Setting up a CI development environment

Kovács Gábor

5 March 2018

1. Introduction

Today we set a CI development environment including the developer machine and the central server side, and show that in work with a *Hello, World!* Java application. The CI development environment of a Java application is the most complex, that is the reason behind our language the choice.

The server side is emulated with a virtual machine on the developer computer, which runs Ubuntu Linux as operating system and has the following Linux packages preinstalled:

- Git server
- Jenkins
- Oracle Java 8, Apache Ant and Apache Maven for Java development
- Sonatype Nexus

2. Preparing the server side

On the server side, we install `git`, `ant`, `maven` and Java from Ubuntu packages. All commands below are issued as root user.

```
> apt-get update
> apt-get install git
> apt-get install ant
> apt-get install maven
```

```
> apt-get install python-software-properties software-
    properties-common
> apt-add-repository ppa:webupd8team/java
> apt-get update
> apt-get install oracle-java8-installer
```

The `ant` installation is in `/usr/share/ant/`, the `maven` installation is in a `/usr/share/maven/`. The local `maven` repository is going to be `~/.m2/repository` in the home directory of the user running Jenkins. The Java installation is in a version dependent directory under `/usr/lib/jvm/`.

Both Jenkins and Nexus are going to be run as standalone Java applications downloaded from their web pages, so no separate installation steps are necessary.

3. Git

The Git service can be accessed by registering the developer's public keys on the server. For that we create a new user called `git`, and let all developers know about its credentials so that they can register their keys, and access the service without having to enter the password each time the source code is pushed from the developer machine.

```
> sudo bash
> adduser git
```

With the command below issued on a developer machine, we can copy the public key of a developer to the central Git server.

```
> ssh-copy-id -i ~/.ssh/id_rsa.pub git@10.211.66.6
```

We are going to store the Git repositories under `/opt/git/`, let our application be under the `hello.git/` subdirectory, and initialise there a new repository. Finally, we as this directories can be created only by the root user, we set up the rights such that the `git` user is the owner of all contents, and the `git` group has write access.

```
> sudo mkdir /opt/git
> sudo chown -R git:git /opt/git/
> cd /opt/git
> mkdir teszt.git
```

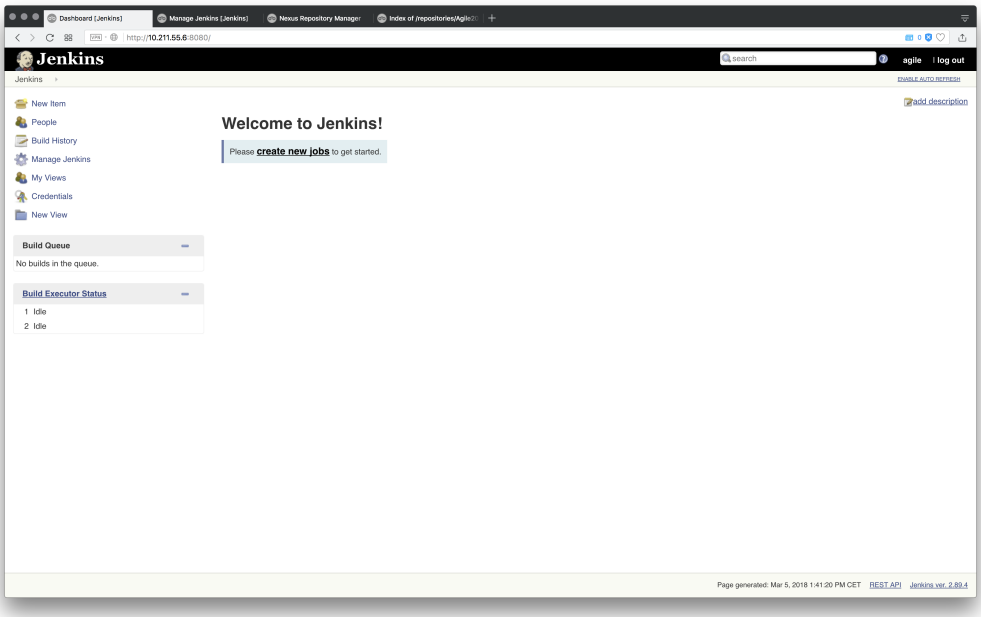
```
> cd teszt.git/
> git init --bare
> chown -R git:git ../hello.git/
> chmod -R g+ws ../hello.git/
```

4. Jenkins

We can download the Jenkins CI service from `http://jenkins-ci.org/`, and the standalone version can be run with a `java -jar` command. All configuration settings are stored in the `~/.jenkins/` directory, in the home directory of the user running Jenkins.

```
> java -jar jenkins.war
```

Managing Jenkins setting can be done on its web interface, that is accessible at port 8080 of the server machine.



1. ábra. Jenkins management page

For our project, we need the plugins Ant, Git and Maven, which should

be available after the startup of the service without any additional installation necessary. However the configuration setting of these plugins must be provided for Jenkins.

In the *Manage Jenkins* menu and the *Global Tool Configuration* submenu we can set the location of our Git, Ant, Maven, Java installations we prepared in Section 2.

In the JDK section, we set the parameters of our Java installation, and set the JAVA_HOME environment variable.

JDK installations	
Name	Oracle Java 8
JAVA_HOME	`/usr/lib/jvm/java-8-oracle`
Install automatically	false

In the Git section, we set the `git` executable.

Git	
Path to Git executable	git
Install automatically	false

In the Ant section, we set the parameters of our Ant installation, and the ANT_HOME environment variable.

Ant installations	
Name	Apache Ant
ANT_HOME	`/usr/share/ant`
Install automatically	false

In the Maven section, we set the parameters of our Maven installation, and the MAVEN_HOME environment variable.

Maven installations	
Name	Apache Maven
MAVEN_HOME	`/usr/share/maven`
Install automatically	false

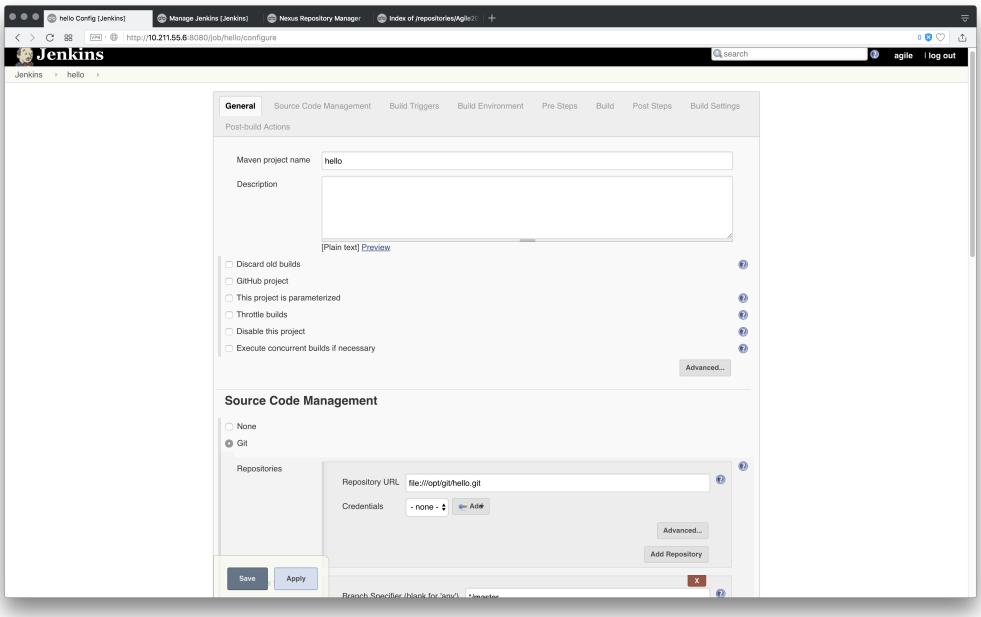
We can leave all other parameter at the default value for now.

Now that we have configured Jenkins, we can create a new *Maven project* with the name *Hello* by clicking on the *New Item* menu. (For C project, we should choose *Freestyle project.*) The project configuration page is shown in Figure 2.

First we set the project description.

Maven project name	Hello
Description	Hello, world!

Our next task is to configure how Jenkins can access the sources of our



2. ábra. The configuration page of our Hello project

project. We choose Git, and provide the path to the sources. As Jenkins is not run as `git` user, we may want to provide passwordless access for this user too just like we did that for the developers. First we generate new keys, then we add the public key to the `git` keyring.

```
> ssh-keygen -t dsa -b 1024
> ssh-copy-id git@localhost
```

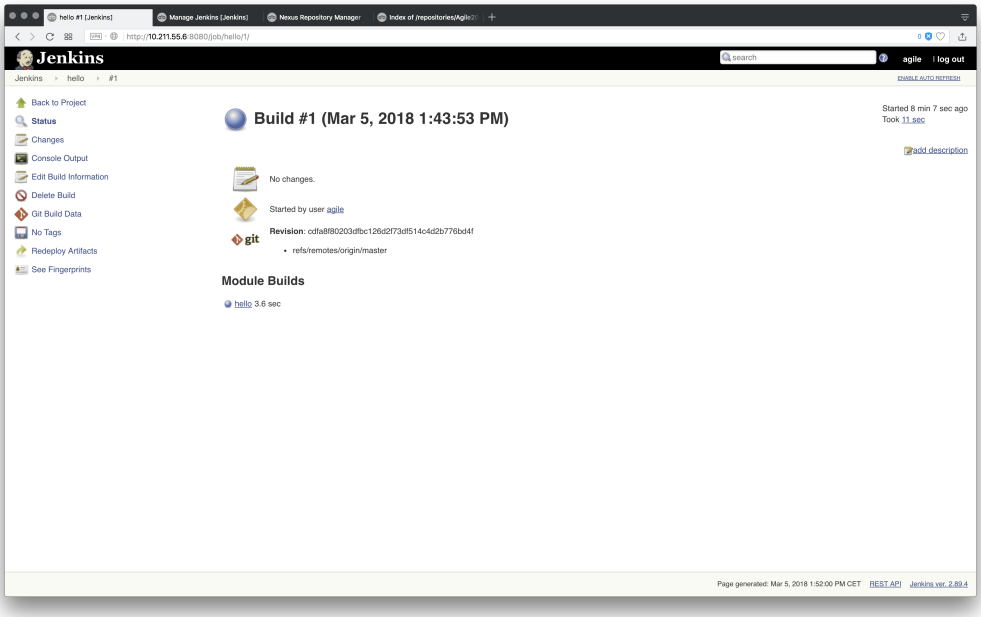
Source Code Management	Git
Repositories	
Repository URL	`ssh://git@localhost/opt/git/hello.git`

The build itself can be triggered manually from the menu, or whenever new code appears in the Git repository, or periodically, which is run as a cron job. Usually it is a good idea to run a 10 minutes build on each commit by developers, and daily, preferably at night a full build

Build Triggers	
Build periodically	selected
Schedule	40 3 * * *

The build itself is done by the Maven build engine based on the project's POM configuration file. Here we use only two actions, `clean` for clearing the results of the previous build, and `install` to perform the build and copy the results to the local Maven repository (`~/.m2/repository`). We may consider using other tasks like `test` to run the tests or `deploy` to deploy the application to the production server .

Build	
Root POM	`pom.xml`
Goals and options	`clean install`



3. ábra. A build of the hello project

5. The Hello, world project

We assume that we have already made passwordless login possible to the server machine as shown above.

Let us create an empty Maven project in our favourite IDE, we used NetBeans. Alternatively, we can issue the command below on console to do the same:

```
>mvn --version mvn archetype:generate -DgroupId=hu.bme.
    tmit.agile -DartifactId=hello -DarchetypeArtifactId=
    maven-archetype-quickstart -DinteractiveMode=false
```

Then we have a `hello` directory containing a `pom.xml` file and a `src/main/java/` subdirectory, where we can put our sources codes. The `pom.xml` has the group identifier `hu.bme.tmit.agile`, the artifact identifier `hello` and the version number `1.0-SNAPSHOT`, there three attributes together identify a Maven resource uniquely. The external dependencies must be set inside the `dependencies` tag, in each `dependency` tag we have to give the group identifier, the artifact identifier and the version number of the JAR file we would like to be made available to be used for our project. When set Maven automatically downloads these JARS from the repository they are available in, and puts the copy under the `~/.m2/repository` directory in the development machine. To make the JAR file build from the project executable, we provide some settings under `build` tag, which sets the `Main-Class` attribute in the MANIFEST of the JAR file.

```
<?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 h
ttp://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>hu.bme.tmit.agile</groupId>
  <artifactId>hello</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.
        sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>
```

```
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
          <configuration>
            <archive>
              <manifest>
                <mainClass>hu.bme.tmit.agile.hello.
                    Hello</mainClass>
              </manifest>
            </archive>
          </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Finally let us write our complex software.

```
package hu.bme.tmit.agile.hello;

public class Hello {
    public static void main(String[] args) {
        new Hello().hello();
    }

    public void hello() {
        System.out.println("Hello, world!");
    }
}
```

Now we just have to get this code to the CI infrastructure. Therefore we initialise a Git repository in the source directory of our project, and check its status.

```
> git init
Initialized empty Git repository in /Users/kovacsg/
   Documents/munka/code/NetbeansProjects/hello/.git/
> cd .git/
```

```
> ls
HEAD        config      hooks       objects
branches    description info        refs
> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be
     committed)

        pom.xml
        src/

nothing added to commit but untracked files present (
   use "git add" to track)
```

We have two files unknown for Git: `pom.xml` and the `src/` directory. Let us make them version controlled, and check what happens in the meantime inside Git.

```
> git add pom.xml
> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   pom.xml

Untracked files:
  (use "git add <file>..." to include in what will be
     committed)

        src/
```

```
> git add src/main/java/hu/bme/tmit/agile/hello/Hello.
   java
> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   pom.xml
        new file:   src/main/java/hu/bme/tmit/agile/
           hello/Hello.java


> git commit -a -m 'initial'
```

After `git commit` all of our files are version controlled. Next we have to share our changes with the other developers. For that, we set the repository we created on our server. We shall refer to that as `origin`, and we shall call the current version of our source `master`, that is we create the master branch.

```
> git remote add origin ssh://git@10.211.55.6/opt/git/
   hello.git
> git push -u origin master
Counting objects: 15, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (15/15), 1.48 KiB | 756.00 KiB/s,
    done.
Total 15 (delta 2), reused 0 (delta 0)
To ssh://10.211.55.6/opt/git/hello.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from
    origin.
```

Our sources have arrived to the CI server, so we can trigger a build from the menu of the project by clicking the *Build Now* button. In the *Build Histo-*

ry we can see the result of the build – blue if successful and red if failed. The built JAR file is under `~/.m2/repository/hu/bme/tmit/agile/hello/1.0-SNAPSHOT` directory, and it can be executed with the `java -jar` command. The `hu/bme/tmit/agile` tag is the group identifier, the `hello` the artifact identifier, and the `1.0-SNAPSHOT` is the version number as given in the `pom.xml`.

```
> java -jar hello-1.0-SNAPSHOT.jar
Hello, world!
```

6. Sharing the results

The JAR file we just created is available in a directory of the server, we have to make that accessible for all developers with a repository manager service. The repository manager assigns an URL to a local directory, and makes its contents available on the web, and it can also mirror the resources of other repository managers.

We chose Sonatype Nexus to be our repository manager. The application itself can be downloaded as a zip file, which we have to extract first, and then we can run it:

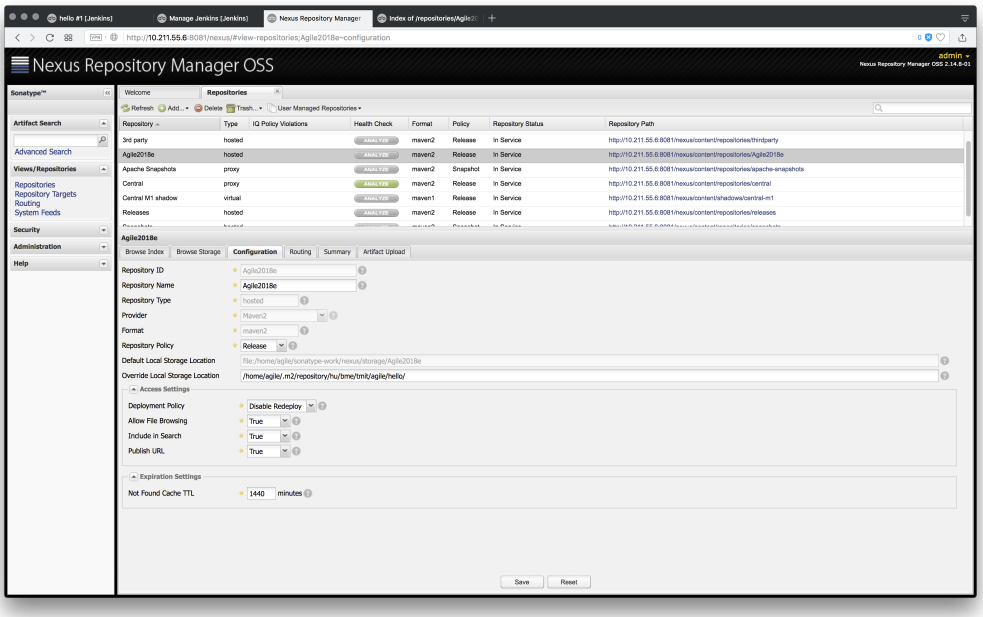
```
> cd nexus
> bin/nexus console
```

After its startup, we can open its management web page (`http://10.211.55.6:8081/nexus`), where we log on as administrator with the default password. We open the *Repositories* menu, and create a new hosted repository. The identifier we give is going to appear in the URL, the name is arbitrary. As Override Local Storage Location we should set the Maven repository of our compiled project.

After saving the changes, the Maven resources are available on the web `http://10.211.55.6:8081/nexus/content/repositories/Agile2018e/`.

This repository can be referred in a `pom.xml` project descriptor of a Maven project created by another developer.

```
<project>
    ...
    <repositories>
        <repository>
```



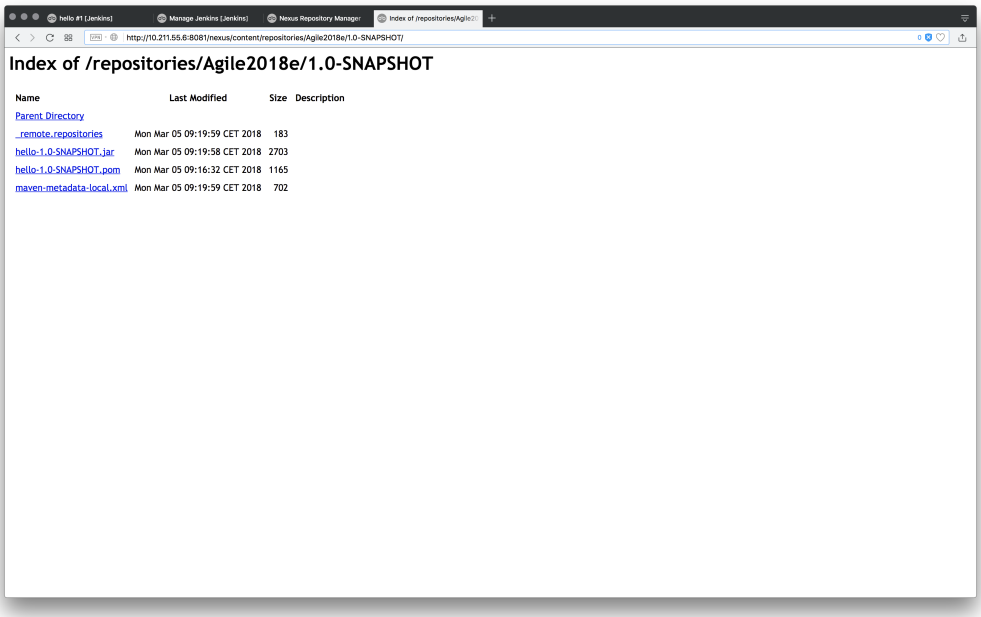
4. ábra. Configuring the Nexus repository

```
            <id>agile</id>
            <url>http://10.211.55.6:8081/nexus/content/
                repositories/Agile2018e/</url>
        </repository>
    </repositories>
    ...
</project>
```

The artifacts in this repository can be used as dependencies by all developer in their own modules. So we have achieved the continuous integration of code written by separate developers, or work groups.

7. Non-Java project

For non-Java project, we have to use *Freestyle project* in Jenkins when creating a new item. Just like in the Java case we have to set up the version control configurations. The difference is in the build. While for Java Maven



5. ábra. Our projects repository

performs all build related tasks based on the `pom.xml`, we may not have such tool available for other languages, therefore a build script must be written to do the compilation, possibly linking, test execution, installation, deployment etc. tasks. The script itself must be defined in *Execute shell* test area. The execution of this script can be triggered just like in the Java case by a user action, a commit or periodically. The script has to take care of the installation as well, and copy the build artifact to a remote machine.