Lund University
Computer Science
Mattias Nordahl

Software development in teams
EDAF45
2018–11–26

Lab 5 Exercise
Build and continuous integration tools

1 Introduction

This lab will give you some experience in using build tools, as well as continuous integration tools, which together can help automate many of the mundane tasks a developer otherwise has to do him- or herself. You will also get to work with these tools in the Eclipse IDE.

The lab is divided into three major parts; build tools, working in Eclipse, and continuous integration tools. You should work in pairs.

You will work with a terminal initially, and several commands will be given for you to write in the terminal. Be advised that copying and pasting text from PDF documents often incorrectly adds extra spaces or uses the wrong type of quotation marks. ***Always double check the format if copying, or just write it out by yourself instead.***

In later parts of the lab, there will be sections where you are asked to reflect on what you have done, together with an empty box like below. Discuss with your partner and write down your thoughts, and answers to any questions that are asked.

2 The build tool Gradle

As a project grows larger, there may be many commands that need to be run in order to compile, test, create jar files, etc. Running these commands manually after each change of the source code would quickly become both tedious and error prone. Instead, software projects use build tools that document and automate the commands to run. Examples of common build tools are Make, Ant, and Gradle. In this lab, we will use Gradle.

The Gradle build tool is implemented in the language Groovy, which compiles to Java bytecode. We will run the Gradle tool via a wrapper script called gradlew. When this script is run the first time, the Gradle tool (a jar file) is downloaded from the web automatically. This way, you don't need to manually install the Gradle tool on your computer.

Gradle is a very powerful build tool. Here are some things it can do:

- automate common tasks like compiling, producing jar files, and running tests
- support different languages by the use of plugins. There is, for example, a Java plugin.
- incremental builds, i.e., avoid building things that are already up to date. For example, if you run a build, but did not change any source files since your last build, then no compilation or testing will be run.
- automatic download of dependencies, i.e. appropriate versions of library files that your project depends on. These files are downloaded from global repositories like Maven and Ivy.

2.1 Gradle basics

To understand the basics of how Gradle works, first download `JavaGradleProject.zip` from `http://fileadmin.cs.lth.se/cs/Education/EDAF45/2018/labs/L5/JavaGradleProject.zip`, and unzip it. This is an empty Java project, set up to build with Gradle. In the project you will find the following files and directories:

- build.gradle – the build script
- gradle/ – contains the gradle-wrapper.jar file, among other things.
- gradlew – the wrapper script for running Gradle on Unix
- gradlew.bat – the wrapper script for running Gradle on Windows
- src/main/java – where your main Java code should be placed
- src/test/java – where your JUnit test code should be placed

The gradlew script might not be executable, so start by adding execution rights to `user`:

```
chmod u+x gradlew
```

When you run Gradle (using the script `gradlew` or `gradlew.bat`), it by default reads in the file `build.gradle`. This file is the build script and it contains rules for how to build the project. Take a look at `build.gradle`. The only thing it contains is a line of code that says that the Java plugin should be used:

```
apply plugin: 'java'
```

The Java plugin defines rules for building Java projects. It defines a number of tasks, i.e., pieces of work. You can find out which tasks are supported by running the command

```
./gradlew tasks --all
```

As you see, there are quite a few tasks. We will only discuss the following:

- build – compiles any source and test files, creates a jar file, and runs any test files
- compileJava – only compiles the main Java code (not the test code)
- compileTestJava – only compiles the test code (the JUnit tests)
- clean – removes the generated files (class files, jar files, etc.)

Try out a few of these tasks. For example, run

```
./gradlew build
```

Note that all generated files are placed in a newly generated directory `build`. Since there are no source files yet in this project, there will in this case be no generated class files. There will, however, be a generated jar file, although it contains nothing but a manifest. The jar file is named `JavaGradleProject.jar` after the project directory.

Now run the clean task, and note that the build directory with all the generated files is removed.

```
./gradlew clean
```

Tasks depend on each other in a directed acyclic graph. You might have noticed that when you run a task, it lists all dependent tasks it runs (its subtasks). Run the `build` task again. Note that it, for example, depends on the `compileJava` task.

When a task is called, and everything it depends on is unchanged, it will not be run again, but is instead reported to be `UP-TO-DATE`. Run the `clean` task, and then the `build` task twice. Note how several subtasks that were not up to date the first time are now up to date.

Gradle keeps its internal information in a directory `.gradle` in your home directory, and in a separate directory `.gradle` in each project. Any dependencies (library jar files, etc.) downloaded by a build will be stored in the `.gradle` directory in your home directory, so that they can be shared by all your Gradle projects. The Gradle tool itself is also stored there.

2.2 Adding production code

We will now add some Java code to our project. Download `expr-src.zip` from `http://fileadmin.cs.lth.se/cs/Education/EDAF45/2018/labs/L5/expr-src.zip`. It contains the source code for a simple model of expressions. Extract the content into your project, placing it in `src/main/java/`. Now, build your project (`./gradlew build`).

The build should be successful, and in the generated `build` folder you should now find several compiled `.class` files. As before, a jar file will also have been generated, but it is no longer empty; It now contains the generated class files.

2.3 Adding test cases

With the production code in place, it is time to add some tests. Download `expr-test.zip` from `http://fileadmin.cs.lth.se/cs/Education/EDAF45/2018/labs/L5/expr-test.zip`. It contains a number of JUnit test cases for the `expr` package. Extract the content into your project, placing it in your project test folder `src/test/java/`. Open the test files and take a brief look at what they contain.

You may try to build your project again, but you will notice that there are multiple errors. As you know, running JUnit tests requires the JUnit test framework, which we must add as a dependency in our build script. Open the build script (`build.gradle`) and add the following lines.

```
// Specify where to look for dependencies
repositories {
    mavenCentral()
}

// Declare that the task testCompile requires the JUnit framework
dependencies {
    testCompile "junit:junit:4.12"
}
```

Build the project again. This time, gradle will download the JUnit framework, and include it in the build path when running the tests. The build should now be successful.

We can optionally add the following lines as well to the build script.

```
// When running the task 'test'
test {
    // Log passed/failed tests in the console (see also build/reports/tests):
    testLogging.events 'passed', 'failed'

    // This causes tests to always be re-run.
    dependsOn 'cleanTest'
}
```

Build the project once more. The tests will now be cleaned before being run, and thus always run even though nothing has changed. Additionally, each test case is reported as either *passed* or *failed* directly in the terminal.

2.4 Comments and further reading

This has been a short introduction to build tools and Gradle in particular. You can of course do much more. For example, you could configure your build script to not only compile your code and run your tests, but to build runnable jar files, generate documentation and package everything (jar files, source code, test files, documentation etc.) in a neat archive. With a proper build script and work process, doing a release of your project can be turned from an hour long hassle to the push of a button.

As such, you may wish to use Gradle, or some other build tool, in your upcoming projects. There are many resources readily available on the Internet from which you can learn more. Finding and reading some might be a good way to spend your "spike" time in your projects.

For Gradle, a good place to start is at their documentation page (https://gradle.org/docs/), where you can find the User Manual, guides and tutorials and help on many of its plugins, e.g. the Java plugin and the Distribution plugin which can help you build release-ready archives of your application and documentation.

3 Eclipse preparations

Before we start working with continuous integration tools, we will create a git repository at BitBucket and add our project to it. We will also set up an Eclipse project, and see how we can use Gradle from Eclipse.

3.1 Creating repository

Create a new private repository on BitBucket. Next, we will add our code to it.

In the terminal, make sure you are in your project directory. Then, initiate an empty git repository, and add all files to the staging area. Now, check git status.

```
cd JavaGradleProject
git init
git add .
git status
```

You may notice there is a `.gitignore` file. It has been predefined to exclude hidden files from Gradle and Eclipse that need not be in your repository.[1] Verify that only your build script, the gradle wrapper and your source and test files are staged.[2]

2) Next, commit your files. Then connect your repository to BitBucket.

```
git commit -m "Initial project setup."
git remote add origin <fill in repository address from BitBucket>
git push -u origin master
```

[1] https://www.gitignore.io/

[2] If all developers use the same developing tools, e.g. Eclipse, you can add Eclipse's hidden files to your repository too, to make importing easier. In this lab, however, we will only version control source files.

3.2 Creating an Eclipse project

We will now start working in Eclipse, instead of the terminal.

If you prefer another tool or text editor you are free to use that instead, but it might be difficult to follow along the instructions. Alternatively, if you feel more comfortable working through the terminal (and can handle git branching in the terminal) you may skim through and skip this part.

First, let's import our code from BitBucket into a new Eclipse project.

1. In Eclipse, open the Git Repositories view, or the Git perspective.
   Git Repositories view: Window → Show View → Other and select Git Repositories
   Git perspective: Window → Perspective → Open Perspective → Other and select Git
2. Find and click the button Clone a Git repository. Copy and paste your repository URL from BitBucket.
3. Click through the wizard with default options until the clone is complete, ***but take note of where your repository is placed***.
4. Right-click your repository, and choose to Import projects.
5. In the import window, choose the second option `Import using the New Project Wizard`, and click finish.
6. In the wizard, select Java Project and click next. ***OBS!*** After giving your project a name, e.g. PVGLab5, uncheck the option *Use default location* and instead browse to the directory where Eclipse cloned your repository.
7. Then, step through the rest of the wizard using the default settings.

All of your source and test files should now be in your new project, in two separate folders *src/main/java* and *src/test/java*. However, Eclipse will report some build errors in the test folder. You need to add the JUnit framework on the build path. For example, by right-clicking your project and selecting properties. Find Java Build Path → Libraries → Add Library and then select JUnit version 4.

3.3 Using Gradle in Eclipse

The Gradle build tool can be integrated into Eclipse through a plugin, allowing us to continue using our build script. To install the plugin, select the menu option Help → Eclipse Marketplace. In the search field, type 'buildship' and find the *Buildship Gradle Integration 2.0* plugin. Install it, if it is not installed already. You may need to restart Eclipse (File → Restart).

We can now "convert" our *Java* project to a *Gradle* Project, by importing it again.[3]

[3]You could have done the initial import as a Gradle Project immediately, instead of as a Java Project first, but the source folders would then have been incorrect and would have had to be manually fixed. Although our approach is somewhat confusing, it is less error prone.

1. Right-click your project and select Import.
2. Find Gradle → Existing Gradle Project and click next.
3. Make sure that the project root directory is the correct one (where Eclipse cloned your repository), and use all default settings.
4. Once complete, restart Eclipse (File → Restart). This will initiate the Gradle project properly.

You should now be able to see and run all available Gradle tasks in the Gradle Tasks view (*Window → Show View → Other, Gradle Tasks*). Try to run e.g. the `test` or `build` task.

4 Continuous Integration with Jenkins

In any project that has more than a single developer, their code needs to be integrated with each other's eventually. Doing so frequently, or continuously, is one of the core practices in XP. Its main purpose is to prevent and help solve integration problems; The longer time spent without integrating your code, the higher risk you have of running into problems once you do. Continuous Integration (CI) tools help developers to automate the integration, and can help improve your development process in many ways.

There are several CI tools to choose from; Some you typically run on a separate server, others are offered as hosted services. Some providers of git repositories, e.g. BitBucket and GitLab, also offer their own CI services, typically suitable for smaller projects. In this Lab we will use Jenkins, which is one of the more popular CI tools. A Jenkins server has been set up especially for this Lab.

4.1 Creating a build project

The Jenkins server can be accessed from your browser at `https://vm59.cs.lth.se:8443`. At this lab's scheduled time, the Jenkins server will be open for user registration. Start by visiting the server and Create an account. You may chose your username and password freely; at the end of the course all users will be removed.

Once you are logged in, you can create a build project for your code. From the dashboard, click New Item in the top left corner. Enter a name for your project, preferably something unique to you, e.g. `pvglab5-<studentId>`. Then, select Freestyle project and click OK at the bottom of the page. Now you get to configure your project.

Important! By default, *all* users will be able to see and configure *all* projects. To prevent other students from being able to see and access your project, which might result in them interfering with your work by accident, check the box `Enable project-based security`. In the drop-down menu `Inheritance Strategy`, select *Do not inherit permission grants from other ACLs.*

Important! Now, only the permissions explicitly set in this project will apply. Therefore, you must add permissions to yourself, or else you will be locked out from your own project. In the textfield `User/group to add`, enter your own username and click add. Then check all permission boxes to give yourself all available permissions.

Next, go down to the segment **Source Code Management**, and select Git. In the text field `Repository URL`, paste your BitBucket repository URL (you can find this at your BitBucket repository). Now, you must add your BitBucket credentials, to allow Jenkins access to your BitBucket repository. Click the button `Add` and select Jenkins, this will open a new window for adding credentials. The `Kind` should be *Username and password*. Fill in your BitBucket username (should be your e-mail address) and password. Leave the ID blank, but write something unique and recognizable in Description, e.g. your student ID. Now, click Add.

Unfortunately, Jenkins handles credentials globally, so now you must find your credentials in the drop-down list among the credentials of all other students. Once you have selected it, Jenkins will automatically check the credentials against BitBucket in the background. If they are not correct, an error will be clearly shown. Otherwise, proceed to the next step.

Now, go down to the segment **Build**. Click Add build step, and select Invoke Gradle script. Then, choose the second option Use Gradle Wrapper, check the box Make gradlew executable and in the text field for which Gradle task to run, fill in `build`.

If you have followed the instructions up to this point, your build script will be at the root of your repository. If so, you may leave everything else with their default values and Jenkins will find your build script and the Gradle wrapper automatically. If not, you may need to specify where your build script is located. Click Advanced and fill in the text field Build File, e.g. `YourGradleProject/build.gradle`*.*

Click save at the bottom of the page. This takes you to the overview of your project.

You can now try to let Jenkins build your project. In the top left, click Build Now. This will schedule a new build, which you can see in the build history (as #1). Click on the build to see an overview of that particular build instance. Here you can e.g. view what changes were included in this build (none this time, since it was our first build) and the console output. Open the console output; you should recognize it from part 1 of the lab. Did the tests run? Was the build successful?

Reflect briefly on what you have done so far. What happened when Jenkins built your project? How does Jenkins fit into the work flow? Sketch a figure in the box below showing an overview of how Jenkins, BitBucket and your local repository fit together.

4.2 Adding a web hook

Ideally, we want our CI tool to be automated. We want Jenkins to build our project automatically at appropriate times. A practical way of doing this is to use so called hooks. Git provides several hooks, which can be triggered e.g. on commit, on pull, before push, on push, etc. BitBucket (and other repository providers) similarly provide web hooks.

From your project view in Jenkins, click Configure to open the project configuration page. Find the segment **Build Triggers**, and check the box `Build when a change is pushed to BitBucket`. This setting will make Jenkins listen for web hooks from BitBucket. Now save your project.

Then, go to your BitBucket repository, and go to settings. Find and open the Webhooks settings. Click Add webhook and fill in a title of your choice. In the URL field, enter exactly:

```
https://vm59.cs.lth.se:8443/bitbucket-hook/
```

Important! Do not forget to include the ending slash. It is easy to miss but required by the BitBucket plugin in Jenkins. Click save to add your hook.

Now, every time something is pushed to your repository, BitBucket will issue a HTTP(S) request to the Jenkins server, containing all information about the new commit. In turn, Jenkins will schedule a build of your project.

How does the web hook alter your work flow? Sketch an updated figure in the box below that reflects this change.

4.3 Make some code changes

It is time to try out the build process. Go to Eclipse, and open the file TestValue.java in the tests folder. There is a method `testDivByZero()` which have been commented. Uncomment it, save, and commit and push this change. Shortly, you will see in Jenkins that a new build has been scheduled automatically. Does it succeed? No, because we added a new test case but no code that implements the functionality being tested. Check the console output.

To fix this, we will implement the missing functionality. Open the file Div.java in the main source folder. Three lines in the `op()` method has been commented. Uncomment them, and commit and push this change. Yet another build will be scheduled in Jenkins, and this time it will finish successfully!

We intentionally committed code that would not run properly, to see what happened in Jenkins. In an actual XP project, what should we have done instead to avoid such errors?

If we always test our code before committing and pushing it, why is it still a good idea to let a CI tool build it as well? What types of errors will Jenkins find that we might not?

4.4 Branching strategies

We will now experiment with some other development work flows and branching strategies. One goal of XP is to have a fully functional code base at all times, such that it can be released as a working product. As such, we never want any untested or non-working code in our master branch. We are going to try out two different branching strategies together with Jenkins to accomplish that.

4.4.1 Development branch

For keeping the master branch clean, one possibility is to have a separate branch for development, and let Jenkins test it and automatically merge it into master if there are no errors. Developers thus never directly push anything to master, only to the development branch. And if it should happen that a new push accidentally breaks something, the faulty code never reaches the master branch.

From your project overview in Jenkins, click Configure. Under **Source Code Management**, change which branch to build, from `*/master` to `*/development`. Also, for the option Additional Behaviors, click Add and select Merge before build. Specify the repository *origin* and the branch *master*. With this change Jenkins will build the development branch instead, but first merge it into its local master branch.

On successful builds, we would also like Jenkins to push the merged build to the remote master branch (i.e. back to BitBucket). Scroll down to the segment **Post-build Actions**. Click Add post-build action and select Git Publisher. Here, check the two first options `Push Only If Build Succeeds` and `Merge Results`. Then save the new configuration.

Now, let's go back to Eclipse. We want to start working on a new branch called development. Right-click your project, then find and click Team → Switch To → New Branch. Enter the branch name `development` and check the option *Configure upstream for push and pull.* Then click Finish. You are now working on a new local branch!

We will now make some changes to the code by implementing the modulo operator (%). Start by adding some tests for the modulo operator in the files TestPrinting.java and TestValue.java (there are already commented tests which you can simply uncomment). Also create a new expression class called `Mod` (you may copy and rename the `Div` class).

Before implementing the Mod class, try running the tests to see that they fail, e.g. by running the Gradle task `test`, or `build`. Then, complete the implementation and make sure that all tests run successfully. Finally, commit and push your changes. Eclipse will recognize that you are pushing to a new remote branch; Just click Next and/or Finish to confirm.

What happened? Where did you push your code? What does Jenkins do? Did it work? Look in the console output.

Then, in Eclipse, pull new changes from BitBucket. What is new?

4.4.2 Feature branches

Another possible strategy that is sometimes used is to create separate branches for each new feature (or story, or task). We might also combine this with Git pull requests to get more control over when a feature is ready to be merged. Imagine that our customer have a story that ask us to create an `Abs` expression, for calculating absolute values. We will develop this new story in a separate branch.

Let's start by changing our Jenkins project configuration. From your project overview in Jenkins, click Configure. Under **Source Code Management**, change which branch to build, from

`*/development` to `*/story/*`. This will build any branch with the prefix `story/`.

Next, under **Build Triggers**, uncheck the option `Build when a change is pushed to BitBucket`, and instead check the option `Bitbucket Pull Requests Builder`. This reveals several new settings, but we do not need to specify all of them. Fill in the settings like this, leaving the others with their default values:

- Cron: `* * * * *`
  (Five asterisks separated by spaces. This means that Jenkins will poll BitBucket for pull requests once every minute; Unfortunately the plugin does not support web hooks.).
- Credentials: Select your credentials as before.
- RepositoryOwner and RepositoryName: Take these from your BitBucket URL.
  (E.g. if your URL is `git@bitbucket.org/REPO_OWNER/REPO_NAME.git` *then use* `REPO_OWNER` *and* `REPO_NAME` *as owner and name respectively.)*
- Using Git SCM...: Check this box to use the same branch setting as in the previous section.

We are now done with the configuration, so click Save.

Going back to Eclipse, we will now implement the story for absolute values. Create a new local branch, using the remote `master` branch (not the `development` branch) as source, and call it `story/abs`. To save time, we will disregard tests and just add a new class `Abs.java` by copying and renaming `Num.java`. Change the implementation fittingly. Then commit and push your changes.

You will notice that Jenkins does not start a new build automatically, because we have not yet done a pull request. So, at this point we could continue developing our story, and other users could contribute as well, and the story would not be merged into master until we deem it ready. Edit the file again, e.g. by adding a comment, then commit and push. Your `story/abs` branch is now two commits ahead of the master branch.

To create a pull request, go to your repository on BitBucket, and click Branches. You should see the branch `story/abs`. Click it, and then find and click the link *Create pull request*. Since we branched from master, the pull request will automatically target back into master, and the description will be pre-filled with your commit comment. Finish by clicking *Create pull request*.

Now, check in on your Jenkins project. Within a minute, Jenkins should discover the new pull request and schedule a build. What is the result?

Briefly reflect on what you have done. How does your work flow look now?

4.4.3 Multiple developers

When multiple developers are working on separate branches, several interesting cases may occur. Look at the scenarios depicted in figure 1 and figure 2 below, where two developers have branched off from master while working on different stories. They both push several changes on their respective branch, until their stories are done. What happens when they both create pull requests for merging back into master?

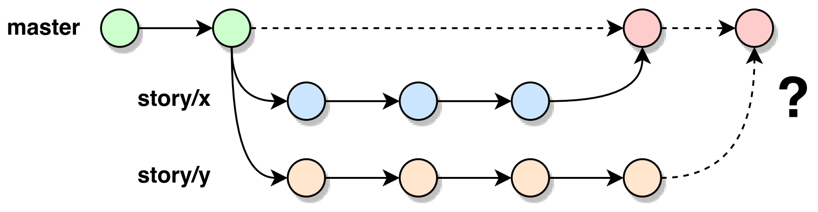

Figure 1: Two story branches, having the same source, both attempting to do a pull request.

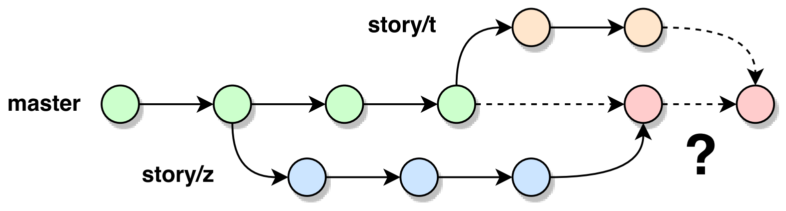

Figure 2: Two story branches from different versions of master, both attempting to do a pull request.

Discuss the depicted scenarios with your partner. Depending on what part of the code the two developers has changed, which different situations may occur? How do you think Jenkins should handle them? Does it? If time permits, experiment with two developers (e.g. using two Eclipse instances with separate repository clones, or two terminals) and try out the scenarios.

4.4.4 Comments

There are of course other alternatives and many variations of branching strategies. Which one you choose to adopt depends on project size and developers' preferences. In smaller projects it may be sufficient to work solely on master, with no additional branches. Larger projects may be further divided into multiple repositories, each possibly having different branching strategies, and thus a more complex build procedure.

Another handy feature, which we have not tried out, is to let Jenkins send alerts to the developers when a build breaks. In the Jenkins project configuration, under **Post-build Actions**, you can e.g. add an E-mail Notification. This enables the person who committed the faulty code, or all team members, to get notifications whenever a build breaks, and when it returns to working order again. If time permits, you may try this out.[4] There are also plugins available to Jenkins for issuing such alerts in other ways, e.g. via Slack.

5 End

This concludes the lab.

This is a fairly new lab with several new additions and changes this year. We would appreciate any feedback in regards to the subjects of the lab (build tools, CI tools, Eclipse), the lab itself, the lab instructions (this document) or anything else. Thank you!

[4]Our Jenkins server does not have a dedicated e-mail server, nor an e-mail address of its own; It is configured to send e-mails through the LU mail server, from mattias.nordahl@cs.lth.se.