Version Control Systems, Build Automation and Continuous Integration -- Exercises

Systems Engineering Laboratory 1

Note: Please treat these exercises also as professional work. For example, instead of *asdfg* use more meaningful commit messages like *Add acceleration feature* or *Fix #5* (you can find detailed advice in the How to Write a Git Commit Message and the Git Style Guide posts).

1 Initialization

In most of the exercises we use command line tools instead of IDEs, since only those can be used in a CI environment. Moreover, you can learn more this way for beginner exercises.

You will need the following software to complete the lab (these are available on the virtual machine template in the laboratory, but you can use your own machine too):

- Java JDK 11 or newer
- Git command line
- Apache Maven 3.6.0 or newer
- A simple IDE like Visual Studio Code
  - Extensions: Extension Pack for Java, Sonarlint

Important: If you use your own machine, then check that Java JDK is configured correctly: use JDK and not JRE, set `PATH` and `JAVA_HOME` (see here in details).

```
java -version
javac -version
```

1. *Fork* the https://github.com/ftsrg-retelab/base project to your GitHub account. If you don't have a GitHub account yet, then please register one, confirm the e-mail address and then you can fork the mentioned project.
2. Initialize your development environment. You will use
   - CLI tools (marked by [**CLI**]),
   - any type of IDE or editor (marked by [**IDE**]) on your desktop to edit Java code. You should also have a git client, either built it into the IDE or a CLI version.
   - Browser (marked by [**WEB**]) on your desktop to use GitHub web UI and other online CI tools

1.1 GitHub access token

Important: Since 2021 you cannot authenticate with passwords to GitHub from the command line. You should create first a personal access token (PAT), and use that from the git commands.

Important: *Personal access tokens* are like passwords and have the same inherent security risks. Always assign as few permissions as possible to our token!

1. On the GitHub interface, click on your own profile, then select the *Settings* menu item.
2. At the bottom of the left side bar, select the menu item <> *Developer settings*.
3. In the left sidebar, select *Personal access token* and then *Fine-grained tokens*.
4. Click on the *Generate new token* button.
5. Give the token a name, set its expiration date to the closest time after the last lab (MIT-4).
6. From the accesses (*Repository access*), select the *Only selected repositories* option and select the previously *forked* repository under our *own* user.
7. Finally, click the *Generate token* button.
8. The generated token must be used instead of **password** during the *git push* operation.

2 Markdown exercises

1. [**CLI**] Clone your own project using the Git client.
2. [**CLI**] Modify the README file in the root of the project, add new comments using **Markdown syntax**. Use at least 3 types of formatting!
3. [**CLI**] Check which files are not currently under version control (`git status`), add any new or modified files to the current index (`git add`), commit them (`git commit`), then push the new content to the repository instance on GitHub (`git push`).
   - When executing the `git commit` command for the first time, the configuration of the user and email address may be missing, these can be done with `git config user.email "you@example.com"` and `git config user.name "Your Name"` to do.

3 GitHub Flow Exercises

1. [**WEB**] Open the project on **your** GitHub.
2. [**WEB**] Click **Settings**.
3. [**WEB**] Enable the **Issues** option in the **General** menu on the left.
4. [**WEB**] Read the project's README to familiarize yourself with the application you will be working with.
5. [**WEB**] Define a new feature or change request and create a bug ticket (*Issue*) on the GitHub web interface for your **own** project. To solve the task, it is not necessary to define a complicated function, but try to define a realistic function in the context of the project (for example, add support for emergency braking).
6. [**WEB**] Create a development (*branch*) to perform these implementation tasks.
7. [**CLI**] Download the remote repository (*pull*) and switch to the newly created branch (*checkout*).
8. [**IDE**] Complete the implementation and test it in a local runtime (`mvn test`).

9. [**CLI**] (*Commit and push*) the changes to the remote repository.
10. [**WEB**] Based on the resulting changes, create a *pull request* for **your own** project on the web interface (**DO NOT** select the original base repository!) so that the change can be discussed within the development team. In the text of the pull request, write *"Fix #1"*, indicating that bug ticket 1 will be closed.
11. [**WEB**] Examine the *pull request* created, and if it is OK, accept it, read it back into the master branch, and delete the branch used for development. (We will deal with the scanning options provided by the pull request in the next measurement.)
12. [**CLI**] Download the changes from the remote repository and check them using the `git log` command.

Further information can be found here:

- Mastering Issues
- Understanding the GitHub flow

4 Merge conflict

1. Figure out how to generate a merge conflict.
2. [**CLI**] Create two new branches (`branch-A`, `branch-B`).
3. [**CLI**] Checkout `branch-A`, edit one line in a file and commit the changes.
4. [**CLI**] Checkout `branch-B`, edit the same line in the same file on a different way and commit the changes.
5. [**CLI**] Switch back to the master branch and merge `branch-A` and `branch-B` afterwards. Check the results and if a merge conflict occurs, then resolve it.

5 GitHub Actions

We compile and test the project in our own Git repository using GitHub Actions.

1. [**WEB**] Open the project in **your** GitHub repository.
2. [**WEB**] Click **Settings**.
3. [**WEB**] In the **Actions/General** menu on the left, select the **Read and write permissions** option from the **Workflow permissions** options. Save your changes.
4. [**WEB**] In the **Code security and analysis** menu on the left, enable the **Dependency graph** option by pressing the **Enable** button.
5. [**WEB**] Open the **Actions** tab of the project.
6. [**WEB**] Select the '**Java with Maven**' workflow from the options offered. This will bring up an interface where you can edit the `maven.yml` file created based on the sample.
7. [**WEB**] Check the `yml` (YAML) file to see if JDK 11 is set for compiling! For syntax help, see https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-java-with-maven.
8. [**WEB**] When you are finished, click the 'Start commit' button to finalize the file. In principle, this will start an instance of the created Actions workflow. Track the progress of your workflow on the Actions page.

9. **[WEB]** Interpret the result of the compilation on the Actions interface (detailed log files can be accessed by clicking on the name of the commit, look for the 'Build with Maven' step within the build task).
10. **[IDE and WEB]** Make some modification to cause a compilation error. Commit and upload your changes, then check the results on the Actions page.
11. **[IDE]** Fix the compilation error and then add a new unit test to the project.
12. **[IDE and WEB]** Finalize and upload your changes, then check the results on the Actions page.

6 Add external dependencies

1. **[IDE]** Implement a tachograph module that records the following values to a single collection. Use the Google Guava library's `Table` class to implement the collection.
   - current time
   - joystick position
   - reference speed
2. **[IDE]** Go to http://mvnrepository.com or http://search.maven.org/ and search for the `guava` dependency.
3. **[IDE]** Add the found dependency to the maven project.
4. **[IDE]** Create a simple unit test that checks if elements are included in the collection.
5. **[IDE and WEB]** Finalize and upload your changes, then check the results on the Actions page.

7 Implement additional GitHub Actions CI workflow (iMSC)

Now that the build is running successfully on the GitHub Actions CI server, there are a few more things to set up.

1. The Maven output in the CI server log is difficult to read due to the many dependency download messages. Add the `--no-transfer-progress` option to the `mvn` command. Has the log become more readable?
2. Specify to cache Maven products (e.g. downloaded dependencies): description.

8 Finishing the lab

Commit all changes, upload to the remote repository, and verify that the GitHub Actions runs correctly.

Important: Do not delete your own GitHub repository, you will also work on it in further laboratories.