Version Control Systems, Build Automation and Continuous Integration

Systems Engineering Laboratory 1

Systematic methods and automation can highly enhance the teamwork on software projects. During this laboratory we will learn the basics of the following techniques:

- **Version control systems** to support efficient teamwork.
- **Build automation** to ensure stable and consistent builds.
- **Continuous Integration** to provide automatic and systematic build, test execution and also deployments.

1 Version Control Systems

Version control systems allow to keep all the historical versions of your software for easy tracking. Using version control also benefits team collaboration and improves the efficiency of the development team. In addition, it can be used as a central repository for the data, making build automation and continuous integration possible.

There are historically two different approaches for managing the repositories:

- **Centralized model (CVS and Subversion/SVN)**: there is one server that contains "the repository" and everyone else checks code into and out of that repository. An important feature of these systems is that only the repository has the full history of all changes made. A *checkout* from this central repository will place a "working copy" on the user's machine. This is a snapshot from a certain version of the project on his/her disk.
- **Distributed model (Git)**: In a distributed version control system, instead of a *checkout*, a user will *clone* a repository from a remote server. In return, he/she receives a full-fledged repository, not just a working copy. The user then has his/her own repository on the local machine -- including all of the project's history. A user can do everything on the local machine: commit, inspect history, restore older revisions, etc. Only if the user wants to share the work with the world he/she has to connect to a remote server and *push* the changes of the repository.

Currently, Git is the most popular version control system, we'll use it during the lab.

1.1 Git basics

A *Git branch* is an active line of development. Working on a branch means *creating, deleting and modifying files*, *adding/removing* these changes to the index and finally *committing* them.

Multiple branches can be used during the development. Git has a complex branching model, where branches can be *created*, changes can be done on the branches and finally the branches can be *merged*.

Committed changes in the local and remote repositories may be synchronized by *pulling* (*fetching* and *merging*) the remote changes into the local repository or *pushing* the local changes to the remote repository.

For additional detailed information and command line examples please read the Git Lectures Page or Basic Git Tutorial (Reading one of these two tutorials is required for the Lab).

1.2 Git workflows

There are multiple workflow models proposed for collaboration purposes. A development team should follow one of these workflow models in order to do a systematic development.

- Git Flow is a complex workflow model for large development project and large teams with release, hotfix and feature branches.
- GitHub Flow is a simplified model for small projects and is good for educational purposes.

During the laboratory we'll use the GitHub Flow workflow that proposes the following steps:

1. Creating a new branch for the new features.
2. Doing the changes in the code and committing to the branch.
3. Creating a *pull request* (sometimes referred to as *merge request*), that can be checked by the team.
4. Doing some discussion and changes on the pull request.
5. Merging the pull request back to the master branch.

2 Build Automation

Building complex software projects is a challenging task. Building the different modules and artifacts, tracking the dependencies and doing the integration build requires systematic solution.

In the beginning there was **Make** as the only build tool available. Later, **Ant** introduced more control on the build process and by integrating **Ivy** introduced the dependency management over the network. Ant was the first "modern" build tool, but its XML description language usually didn't fit to the procedural build descriptions.

Maven was the next build automation tool used primarily for Java projects. It addressed two aspects of building software: First, it describes how software is built, and second, it describes its dependencies. Comparing with Apache Ant, Maven uses conventions for the build procedure, and only exceptions need to be written down, while Ant requires developers to write all the commands that lead to the successful execution of some task. Maven introduced first the ability

to download dependencies over the network (later on adopted by Ant through Ivy). That in itself revolutionized the way we deliver software.

Gradle is designed for large multi-project builds. Gradle builds upon the concepts of Ant and Maven, combines good parts of both tools, and introduced a Groovy-based (one of JVM languages) domain-specific language (DSL) instead of traditional XML form for configurations. As a result, Gradle build scripts tend to be more succinct than those written for Ant or Maven. Gradle supports incremental builds and intelligently determines when not to re-execute up-to-date parts of the build tree and dependencies.

In this laboratory we'll use Gradle to build Java projects. For preparation we require to read at least one Gradle with Java quick start guide to learn the Gradle basics (CLI and basic structure of a Gradle project). We propose one of the following guides:

- Getting Started With Gradle: Our First Java Project
- Gradle Quick Start
- Building Java Projects with Gradle

3 Continuous Integration

Continuous Integration (CI) usually refers to integrating, building, and testing code frequently. It means a pipeline, that starts at the version control system and goes through on the building, testing and also deployment phases (sometimes called also continuous delivery or continuous deployment).

A continuous integration framework provides support to execute the required tools in the continuous integration pipeline and manages the build artifacts.

- **Hudson** and its fork **Jenkins** share the same code base and very similar set of features. Both are easy to extend, powerful and free. Their main advantage is in the number of plugins and community support. They are a self-hosted solution and use a web-based GUI to configure and execute the pipelines. Starting from version 2 they provide also support for declarative pipeline description by using a Jenkinsfile.
- **Travis CI** is a hosted continuous integration service for the open source community and is very well integrated with GitHub. The main strength of Travis is its simplicity. Unlike Jenkins, which allows almost unlimited plugins, complicated flows, etc., Travis is based on a single file, `.travis.yml`, that resides in the root of your code and describes the build pipeline.

During the laboratory we'll use Travis to learn the basics of the CI frameworks.

3.1 Travis

For most of the time, Travis knows what should be done without any need to explicitly define the flow. For example, if there is the `build.gradle` file, Travis will understand that it should be compiled, tested, etc. using Gradle. It inspects your code and acts accordingly. One can switch from Ant to Maven to Gradle without making any changes to Travis or the configuration file.

Travis has a strong dependency with Git. In cases when some other version controls system is used, Travis is not a good option. If, on the other hand, you are using Git, working with Travis is like forgetting that CI even exists. Whenever the code is pushed to the repo Travis will detect it and act depending on changes in the code (including `.travis.yml`). If there is a problem, you will be notified by email. Travis notifications can also be integrated to other services such as Slack.

Keeping CI configuration (`.travis.yml`) as integral part of the code brings advantages. Through that configuration, you tell it what to do and it does it.

To use Travis CI four steps are needed:

1. Register to Travis CI.
2. Activate your GitHub Repositories in your Travis profile.
3. Add `.travis.yml` file to your repository.
4. Trigger your first build with a `git push`.

To prepare for the laboratory, please read the following Travis tutorials:

- Travis CI for Complete Beginners
- Getting started
- Customizing the Build (until the *Breaking the Build* section)

4 Preparing for the laboratory

1. Read this lecture note.
2. Read the linked Git Lectures Page or Basic Git Tutorial.
3. Go through the GitHub Flow tutorial.
4. Read one of the proposed *Gradle with Java* tutorials.
5. Read the proposed Travis CI tutorials.

4.1 Additional materials (optional)

- Git Book
- Evolution of Popular Build Automation Tools
- Gradle User Guide
- Continuous Delivery: Introduction to concepts and tools