Version Control Systems, Build Automation and Continuous Integration

Systems Engineering Laboratory 1

1 Version Control Systems, Build Automation and Continuous Integration

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.

2 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.

2.1 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.

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

2.2 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.

3 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 the framework of this exercise, we use the Maven solution to compile Java projects.

TASK To learn about Maven, read the Maven section of the Build tools summary.

4 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.
- **GitHub Actions** is a service of GitHub, which can be used to automate typical workflows. This can also basically be customized using a simple text configuration file, where you can perform steps consisting of actions (e.g. translation or installation) in response to events (e.g. new commit or pull request, at given times).

In this exercise, we use GitHub Actions to learn the basics of continuous integration frameworks.

4.1 GitHub Actions

The GitHub Actions configuration files are placed in the `.github/workflows` directory, where you can define individual workflows. In the case of a workflow, you must specify in case of which events it should run. A workflow consists of tasks (job), the steps associated with the task are executed in a runner. The runtime environment can be a container or a virtual machine. A step can execute commands or call actions defined by GitHub or external sources.

The following example configuration file written in YAML demonstrates the most important concepts:

```yaml
# This is a basic workflow to help you get started with Actions

name: CI

# Controls when the action will run. Triggers the workflow on push or pull request
# events but only for the master branch
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

# A workflow run is made up of one or more jobs that can run sequentially or in parallel
jobs:
  # This workflow contains a single job called "build"
  build:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest

    # Steps represent a sequence of tasks that will be executed as part of the job
    steps:
    # Checks-out your repository under $GITHUB_WORKSPACE, so your job can access it
    - uses: actions/checkout@v2

    # Runs a single command using the runners shell
    - name: Run a one-line script
      run: echo Hello, world!
```

5 Preparing for the laboratory

- Git Lectures Page or Basic Git Tutorial
- The Maven Basics section of Build tools wiki page.
- Core concepts for GitHub Actions

5.1 Additional materials (optional)

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