**Introduction to Git**

Git is officially defined as a *distributed version control system* (VCS).

In other words, it's a system that tracks changes to our project files over time. It enables us to record project changes and go back to a specific version of the tracked files, at any given point in time. This system can be used by many people to efficiently work together and **collaborate on team projects**, where each developer can have their own version of the project, distributed on their computer. Later on, these individual versions of the project can be merged and adapted into the main version of the project.

Link to download Git

<https://git-scm.com/downloads>

After installing it, start your terminal and type the following command to verify that Git is ready to be used on your computer:

git –version

**Configuring Your Name & Email**

git config --global user.name "Your Name"

git config --global user.email "your@email.com"

**Repositories**

When working with Git, it's important to be familiar with the term **repository**. A Git repository is a container for a project that is tracked by Git.

We can single out two major types of Git repositories:

* **Local repository** - an isolated repository stored on your own computer, where you can work on the local version of your project.
* **Remote repository** - generally stored outside of your isolated local system, usually on a remote server. It's especially useful when working in teams - this is the place where you can share your project code, see other people's code and integrate it into your local version of the project, and also push your changes to the remote repository.

**Checking the status**

git status

This is a command that is very often used when working with Git. It shows us which files have been changed, which files are tracked, etc.

**Staging files**

From the project folder, we can use the **git add** command to add our files to the staging area, which allows them to be tracked.

git add file.js

To add multiple files, we can do this:

git add file.js file2.js file3.js

Instead of having to add the files individually, we can also add all the files inside the project folder to the staging area:

git add .

By default, this adds **all the files and folders** inside the project folder to the staging area, from where they are ready to be committed and tracked.

**Making commits**

A **commit** is a snapshot of our code at a particular time, which we are saving to the commit history of our repository. After adding all the files that we want to track to the staging area with the \*\*git add\*\* command, we are ready to make a commit.

To commit the files from the staging area, we use the following command:

git commit --m "Commit message"

**Commit history**

To see all the commits that were made for our project, you can use the following command:

git log

The logs will show details for each commit, like the author name, the generated hash for the commit, date and time of the commit, and the commit message that we provided.

The logs will show details for each commit, like the author name, the generated hash for the commit, date and time of the commit, and the commit message that we provided.

The logs will show details for each commit, like the author name, the generated hash for the commit, date and time of the commit, and the commit message that we provided.

git checkout <commit-hash>

To go back to the latest commit (the newest version of our project code), you can type this command:

**Branches**

A **branch** could be interpreted as an individual timeline of our project commits.

With Git, we can create many of these alternative environments (i.e. we can create different **branches**) so other versions of our project code can exist and be tracked in parallel.

When we initialize a repository and start making commits, they are saved to the **master** branch by default.

**Changing branches**

To switch to a different branch, you use the **git checkout** command:

git checkout <branch-name>

For example, you could be working on different features in your code and have a separate branch for each feature. When you switch to a branch, you can commit code changes which only affect that particular branch. Then, you can switch to another branch to work on a different feature, which won't be affected by the changes and commits made from the previous branch.

**To create a new branch and change to it at the same time, you can use the -b flag:**

git checkout -b <new-branch-name>

**To go back to the master branch, use this command:**

git checkout master

**Merging branches**

You can merge branches in situations where you want to implement the code changes that you made in an individual branch to a different branch.

For example, after you fully implemented and tested a new feature in your code, you would want to merge those changes to the stable branch of your project (which is usually the default **master** branch).

To merge the changes from a different branch into your current branch, you can use this command:

git merge <branch-name>

You would replace <branch-name> with the branch that you want to integrate into your current branch.