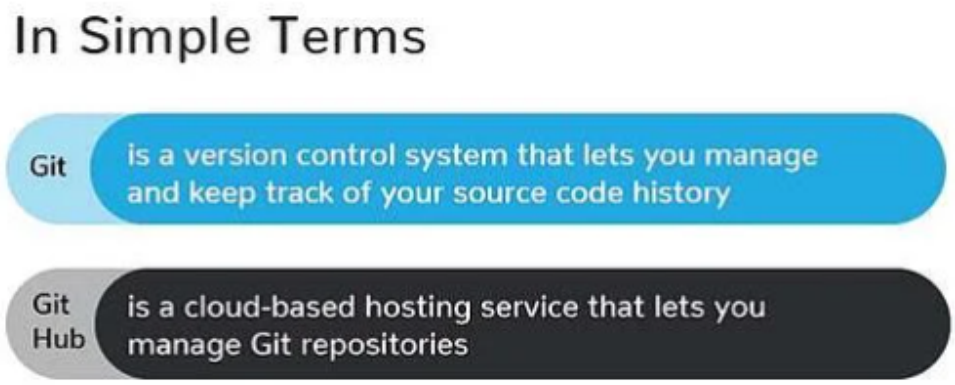
# Git Intro

## Cygwin

First thing's first - go to https://www.cygwin.com/ and install Cygwin, it will make your life a lot easier.
Cygwin is a Linux-like environment for Windows-based systems. It consists of an emulation layer and a collection of tools that provide a Linux look and feel.
It provides support to programmers to make use of Win32 API along with Cygwin API, allowing to port UNIX utilities to Windows without much change in source code.
Advantages of Cygwin over Command Prompt/WSL
There are certain advantages of using Cygwin shell over Windows Command Prompt. Listed below are some of the major ones:

- Provides UNIX shell to Windows, allowing access to a range of utilities from UNIX/Linux world to Windows.
- There is no need to install a full-fledged Linux OS or set up a VM where the requirement is satisfied by resources available via Cygwin.
- Optimal resource usage and system requirements to run on Windows since the environment is emulated and works on top of Windows.
- Ideal for testing/development where the requirement is to use UNIX/Linux utilities on Windows.
- Compatible with older Windows OS like Windows 7 etc., whereas the WSL environment offered by Windows is supported on newer versions only.

## 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.

Basically, it's a massively popular tool for coordinating parallel work and managing projects among individuals and teams. Needless to say, knowing how to use Git is one of the most important skills for any developer nowadays - and it's definitely a great addition to your resume!

**1.Setup**

Git is primarily used via the command-line interface, which we can access with our system terminals.

However, we first need to make sure that we have Git installed on our computers.

You can download Git here: https://git-scm.com/downloads

Click the download link for your specific operating system and then follow through the installation wizard to get things set up on your computer!

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
```

If everything went well, it should return the Git version that is installed on your computer.

**Configuring Your Name & Email**
In your terminal, run the following commands to identify yourself with Git:

```
git config --global user.name "Your Name"
git config --global user.email “your@email.com"
```

Replace the values inside the quotes with your name and email address.

**2. 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.

In this video, we'll only work with local repositories.

**3. Initializing a repository**

To create a new repository and start tracking your project with Git, use your terminal software and navigate to the main folder of your project, then type the following command:
git init
This command will generate a hidden **.git** directory for your project, where Git stores all internal tracking data for the current repository.

**4. Staging and committing code**

Committing is the process in which the changes are '*officially*' added to the Git repository.
In Git, we can consider **commits** to be checkpoints, or snapshots of your project at its current state. In other words, we basically save the current version of our code in a commit. We can create as many commits as we need in the commit history, and we can go back and forth between commits to see the different revisions of our project code. That allows us to efficiently manage our progress and track the project as it gets developed.

Commits are usually created at logical points as we develop our project, usually after adding in specific contents, features or modifications (like new functionalities or bug fixes, for example). Before we can commit our code, we need to place it inside the **staging area**.

### 4.1. Checking the status

While located inside the project folder in our terminal, we can type the following command to check the status of our repository:

```
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.
We can add the untracked project files to the staging area based on the information from the `git status` command.
At a later point, `git status` will report any modifications that we made to our tracked files before we decide to add them to the staging area again.

### 4.2. 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.
We can add a specific file to the staging area with the following command:

```
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.

### 4.3. 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”
```

Inside the quotes, we should write a **commit message** which is used to identify it in the commit history.

The commit message should be a descriptive summary of the changes that you are committing to the repository.
After executing that command, you will get the technical details about the commit printed in the terminal. And that's basically it, you have successfully made a commit in your project!

To create a new commit, you will need to repeat the process of adding files to the staging area and then committing them after. Again, it's very useful to use the **git status** command to see which files were modified, staged, or untracked.

### 4.4. 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.
To go back to a previous state of your project code that you committed, you can use the following command:

```
git checkout <commit-hash>
```

Replace `<commit-hash>` with the actual hash for the specific commit that you want to visit, which is listed with the `git log` command.

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

```
git checkout master
```

### 4.5. Ignoring files

To ignore files that you don't want to be tracked or added to the staging area, you can create a file called `.gitignore` in your main project folder.

Inside of that file, you can list all the file and folder names that you definitely do not want to track (each ignored file and folder should go to a new line inside the **.gitignore** file, by its full path).
You can read an article about ignoring files on this link.

## 5. 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.
That allows us to add new (experimental, unfinished, and potentially buggy) features in separate branches, without touching the '*official*' stable version of our project code (which is usually kept on the **master** branch).

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

### 5.1. Creating a new branch

You can create a new branch using the following command:

```
git branch <new-branch-name>
```

The new branch that gets created will be the reference to the current state of your repository.
It's a good idea to create a **development** branch where you can work on improving your code, adding new experimental features, and similar. After development and testing these new features to make sure they don't have any bugs and that they can be used, you can merge them to the master branch.

### 5.2. Changing branches

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

```
git checkout <branch-name>
```

With that, you switch to a different isolated timeline of your project by changing branches.
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 list the branches for your project, use this command: `git branch`
To go back to the **master** branch, use this command:

```
git checkout master
```

### 5.3. 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.

### 5.4. Deleting a branch

To delete a branch, you can run the **git branch** command with the **-d** flag:

```
git branch -d <branch-name>
```

## 6. Other important Git commands

### 6.1. git clone

This command is used for downloading the latest version of a remote project and copying it to the selected location on the local machine. It looks like this:

```
git clone <repository url>
```

To clone a specific branch, you can use

```
git clone <repository url> -b <branch name>
```

### 6.2. git fetch

This Git command will get all the updates from the remote repository, including new branches.

### 6.3. git push

Git push will push the locally committed changes to the remote branch. If the branch is already remotely tracked, simply use it like this (with no parameters):

```
git push
```

If the branch is not yet tracked, and only resides on the local machine, you need to run the command like this:

```
git push --set-upstream <remote branch> <branch name>
```

### 6.4. git diff

You can use this command to see the unstaged changes on the current branch.
If you want to see the staged changes, run the diff command like this:

```
git diff —staged
```

Or you can compare two branches:

```
git diff <branch1> <branch2>
```

### 6.5. git pull

Using git pull will fetch all the changes from the remote repository and merge any remote changes in the current local branch. You might want to do this a few times per day or at least once at the beginning of your work day, to make sure there are no major conflicts down the line.

### 6.6. git reset

This command unstages the file, but it preserves the file contents:

```
git reset <file>
```

This command undoes all the commits after the specified commit and preserves the changes locally:

```
git reset <commit>
```

This command discards all history and goes back to the specified commit:

```
git reset —hard <commit>
```

### 6.7. git stash (best.command.ever)

The git stash command **takes your uncommitted changes (both staged and unstaged)**, saves them away for later use, and then reverts them from your working copy. You use it as:

```
git stash
```

to hide your changes (you can move to a new branch now, but the changes will stay on the current one).And:

```
git stash apply
```

to bring back your changes.

Here is also in article containing some of the most commonly used git commands with examples:

https://www.freecodecamp.org/news/10-important-git-commands-that-every-developer-should-know/