Git Workshop – NIAEFEUP

Level-Up 2016

Edgar Passos & Nuno Ramos

What is Git?

"Git is a fast, scalable, distributed revision control system with an unusually rich command set that provides both high-level operations and full access to internals."

- Git Manual

Git is a free-open source version control system created by Linus Torvalds.

Having a distributed architecture, Git is an example of a DVCS (Distributed Version Control System). Rather than have only one single place for the full version history of the software as is common in once-popular version control systems like CVS or Subversion (also known as SVN), in Git, every developer's working copy of the code is also a repository that can contain the full history of all changes.

Preparing your development environment:

For this workshop we will work with Git's command line interface, if you haven't already installed it, go to:

- https://git-scm.com/download/win – For Windows users
- https://git-scm.com/download/mac – For Mac users
- https://git-scm.com/download/linux – For Linux users

We will also be using GitHub to host our project, so, if you have not created an account yet, go to www.github.com and register.

Configuring your Git credentials:

The first thing you should do after you have installed git is to set your username and e-mail address.

You can set your username by executing the following command:

```
git config —-global user.name “Your Name”
```

To set your e-mail, use the following command:

```
git config —-global user.email youremail@example.com
```

Your first repository:

To create a new repository, use the git init command:
If you are already in the repository you want to use, simply use

```
git init
```

If you want to create a repository in another directory, you can use this command with the relative or absolute path to the directory

```
git init path_to_directory
```

This creates a git repository in the chosen folder. Keep in mind that this repository is still local, it is not hosted anywhere online, but we’ll get to that later.

Cloning a repo:

If you are going to work on a repository that you or someone else has created and is already hosted somewhere, you can get a copy of that repository by using the git clone command.
To give you an example, we have created a repository which contains this guide, clone it using the commands

```
git clone https://github.com/NIAEFEUP/git-workshop-levelup-2016.git
```

Checking the state of your repository:

To see the current state of your repository, use the `git status` command. It shows you what branch you are on, the changes you made, how the files in your working directory are seen by git, and if there are any modified or uncommited files.

```
git status
```

-

Adding new or modified files:

Create a small text file in your working directory, named README.md.
If you enter the git status command again, something like this will appear in your terminal:

```
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

README.md

nothing added to commit but untracked files present (use "git add" to
track)
```

`git status` is telling us that a new file, unknown to git, named README.md, was found in the working folder. "Untracked" means that although this file is in the directory of the repository, it is not being treated as part of the repository.
To add this file to our repository, you must use the `git add` command, this command places new or modified files in the staging area. Use it on the new file:

```
git add README.me
```

If you use `git status` again, the file will have been added as a new one. You must do this everytime you create a new file, or edit an existing one.

Commiting your changes:

Your file has now been added, but this change to the repository has not yet been saved, to do that, you must commit the changes to the repository using `git commit.` Commits must come with a message, so the `-m` option must be used to simplify the process:

```
git commit -m “Adding README.md”
```

What this command did was tell git to take a snapshot of the state of the repository and save it. You can combine git add and git commit with the -a option

```
git commit -am “New commit”
```

This will have no effect, however, because we haven't changed anything in the repository.
You can see a history of your commits with the git log command:

```
git log
```

Pushing your changes:

The changes made aren't hosted online ("on the remote") yet, we will use `git push to do that.`
But first, you must create a repository online (your remote), that's what we will use Github for, create a new repository.

```
git remote add origin https://github.com/your_username/your_repo
git push -u origin master
```

Forwards, this repository will be referred to as the origin.

Pulling from the remote:

Sometimes, instead of pushing to a repository, you must pull from it, this updates your working tree with whatever is on the upstream
Create a file on your repository using your browser and save it, you can then use `git pull` to put it in your working folder.

```
git pull
```

Branches:

Git's most useful feature is it's branching, this allows multiple people working on the same project without consistently having to worry about what edits others are doing.
To see a list of the branches of your repository use:

```
git branch
```

`master` is the default and main branch for a git repository, it is the first branch displayed when someone clones or goes to see your repository online.
We are going to create a new branch. We can also use the `git branch` command for this:

```
git branch new_branch
```

to switch to the new branch, use the `git checkout` command:

```
git checkout new_branch
```

You can also use `git checkout` to create new branches, if you use the `-b` option, this allows you to create and instantly switch to a new branch, if we wanted to do that with this new branch, we would have used:

```
git checkout -b new_branch
```

Create a new file on this branch, `branch_file.txt.`
Now, as we have done before, add and commit the file with:

```
git add branch_file.txt
```

Wait, we don't want that file! Undo the change!

```
git reset –-soft HEAD^1
```

`branch_file.txt` is out of the repository again. But I was kidding, we do want it, so add it again and then commit the change

```
git commit -m "Added branch_file.txt"
```

You can record the changes to the remote using `git push,` but because it is a new branch, you must set the upstream branch, similar to what you did before:

```
git push -u origin new_branch
```

Merging your branches:

Switch back to your `master` branch:

```
git checkout master
```

You will see that the `branch_file` is not there. You must merge the branches, that is, bring the changes from another branch into the current one:

```
git merge new_branch
```

`branch_file.txt` will appear on your master. You can now push the changes to the remote repository.

Conflicts and how to solve them:

Create a new branch and switch to it:

```
git checkout -b bad_branch
```

Now edit the first line of `branch_file.txt.` Save and commit the change, then switch to the `new_branch` branch and do the same thing.

Finally, switch back to the `master` branch again and merge `new_branch` and then your other branch:

```
git merge new_branch
git merge bad_branch
```

This will create a conflict in your file, as both branches have made an edit on the same line of the same file. Open your file, and you will see something like this:

```
<<<<<<< HEAD
I hope this creates a conflict
=======
Or else this will look stupid
>>>>>>> new_branch
```

This means that what is shown from <<< to === is the `HEAD` (your current branch) version of the file, and what is shown from === to >>> is what is present in your new_branch branch. To solve this, just delete one of the versions and commit the change. Simple!

Working with Git Stash

Open README.md and make some changes. Get ready to make your commit… Oh no! We did not want to change this file in the master branch… No worries, we will use `git stash` to make this work.

```
Git stash
```

This command will instruct git to stash your changes in a stack, to be accessible later, you can see what's on the top of the stack by using `git stash show.`

Let's repeat our mistake, this time by changing the `branch_file.txt.` Use `git stash` again, and then use the following command

```
git stash list
```

You will see that you have two stashes, `stash@{1}` and `stash@{0},` the latter being the most recent one, to see what is in a specific stash just use `git stash show` with the stash you want to know more about

```
git stash show stash@{1}.
```

Checkout to the `new-branch,` where we originally wanted to make the changes to `branch_file.txt.` You can now pop the stash and the changes made will be applied on top of the branch.

```
git stash pop
```

We still have the changes made to the `README.md,` but we want those in a new branch. We can create a new one with `git checkout -b` or `git branch`, but we will use another command

```
git stash branch readme-branch
```

this will create a new branch and pop the stash, easy!

If at any moment you want to discard the changes in your stash, you can just use

```
git stash drop
```

Rebasing your branches

While we are on a new branch, `readme-branch,` let's change a file and commit the change, then checkout to `new-branch`, and make another change on that branch.
If you now use `git log --all --graph --oneline` you will se a graphical representation of the work done on the repository so far, if you go back to the merges made previously, you will see that the merge is represented by two lines converging in a merge commit.

With multiple branches, this can get confusing, so an alternative is rebasing, this is less used because it can lead to some errors if not done right.

To do it, let's go back to `readme-branch` and rebase `new-branch` on top of it

```
git checkout readme-branch
git rebase new-branch
```

If you use `git status` now, you will see that we are in the all the changes were merged into readme-branch`.` If we use `git log --all --graph --oneline` again, you can see that the graph is now only one line, as if the branches never diverged.

The golden rule of `git rebase` is to only use it with your branches, as this will cause problems when others try to push onto a public branch that you have rebased.