Tutorial - GIT

Grupo de Linux da Universidade de Aveiro
Eduardo Sousa - eduardosousa@ua.pt

2016

1 Introduction

Welcome to this tutorial. This is a brief introduction to version control software and GIT. This tutorial is separated in three parts:

- Introduction
- First steps
- Advanced commands

Introduction - we will talk about Version Control Systems and GIT.
First steps - we will talk about GIT's basic commands.
Advanced commands - we will talk about GIT's advanced commands.

1.1 What is a version control system?

A version control system (**VCS**) is a software that allows you to manage the changes in files, while retaining previous changes. It will also allow you to work with others more effectively, since it will track the changes for you.

1.2 Why use version control system?

Instead of using other kinds of systems, you can use **VCS** to track the changes in code and in file structure of your project. Just this aspect will allow you to focus in code production, instead of always worrying if you have the right files or what others have changed in the files.

1.3 What is GIT?

GIT is one of the main **VCS** used for software development. **GIT** main features are:

- Speed
- Data integrity
- Distributed
- Non-linear workflow

GIT allows you to track your files and the changes made to them. GIT structures the tracking of files in a tree way, where each node of the tree will represent a different state of the repository. Each node of the GIT tree is called a commit and it is a snapshot of the files that where being tracked at the time of that commit. This tree structure allows you and your colleagues to see the different stages of the tree, in other words it represents the history of the project and the code.

The tree structure also allows for concurrent development, because each branch of the tree is independent and later on the branches can be merged.

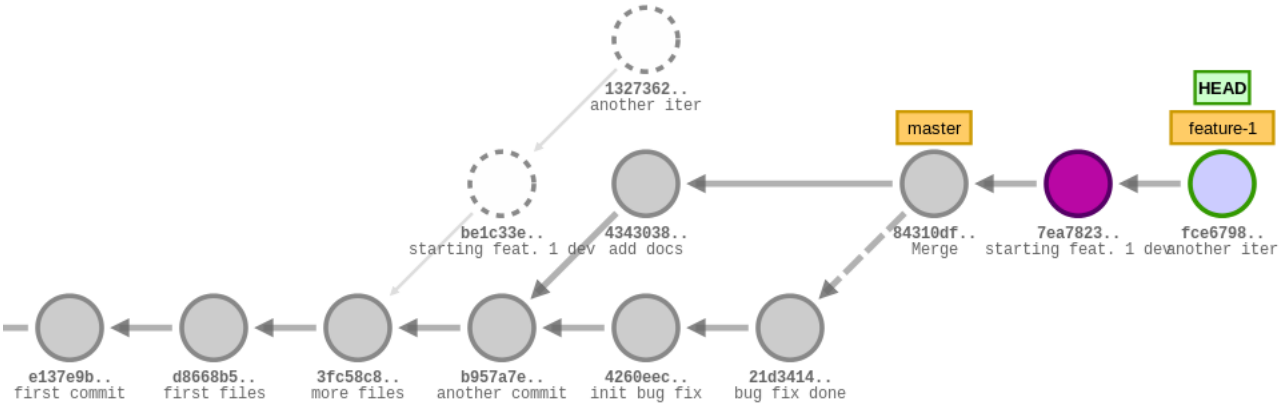

Figure 1: GIT tree structure

2 First steps

This is the beginning of a great journey for you. You will learn the basics of GIT, which in turn will help you until the end of your days as developer.

2.1 Create a local repository

First of all, let's create a new repository, so you that you can start tracking all the changes in your project.

```
git init
touch README.md
git add README.md
git commit -m "Initial commit"
```

Let's dissect each command:

- **git init** - this command will initialize a local git repository in the folder you're in.
- **touch README.md** - will create an empty file called README.md.
- **git add README.md** - will make your local GIT repository start tracking your file.
- **git commit -m "Initial commit"** - will set the actual state of the repository as the first snapshot taken.

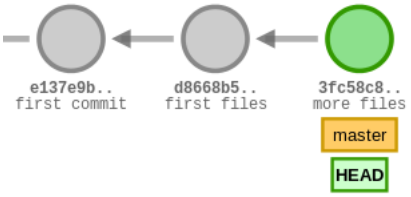

Figure 2: Initial repository state

2.2 Adding files

Let's create a few other files, so we can start tracking them with GIT.

```
touch file_1
touch file_2
mkdir folder_1
touch folder_1/file_1
touch folder_1/file_2
```

Let's check what is the state of the local GIT repository, with **git status**.

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

file_1
file_2
folder_1/

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

There are two ways of start tracking these files. You can add all files with one command or you can add them one by one. If you know that you want to add them all, you can use this command:

```
git add .
```

If you don't want to track all files, you can add them individually like this:

```
git add <file>
```

In this case, we want to add all files, so we will use **git add .** and now let's check the state of the local repository with **git status**.

```
On branch master
Changes to be committed:
   (use "git reset HEAD <file>..." to unstage)

new file:  file_1
new file:  file_2
new file:  folder_1/file_1
new file:  folder_1/file_2
```

2.3 Making commits

Now that we have added some new files and started tracking them in our repository, let's save this state. To save this changes in the repository, we need to make a commit of the changes. To commit the changes, we will use the following command:

```
git commit -m "A message relevant to the changes made in this commit"
```

Now that we know which command to use, let's use it but don't forget to add a meaningful message, so that when others check the commits in the repository, can understand what or why you did those changes.

2.4 Checking the log

Let's check the log to see the changes we have made to our local repository. To check these changes, we will check the local repository log and to do that we must run the following command:

```
git log
```

After running this command, we will get an output similar to this:

```
commit 3070d3621b60b5ebe46b2d58ad6be2537069d79d
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:44:20 2016 +0000

   Added initial files

commit 5814279838033e7b0f14a620c73202f52f11cf99
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:26:39 2016 +0000

   Initial commit
```

What can we check in the log? Well, we can see the commits that were made to the repository, when were they made and the message that the author of the commit wrote for that change.

Now let's say you want to check what a specific commit made. You can verify this with this command:

```
git show <commit hash>
```

Where the **<commit hash>** is the number that appears after the commit word when using the **git log**.

2.5 Updating the remote server

When we are using Git in our computers, the commands we learnt (git init, git add, git commit) have a direct effect on our local repository. But Git was created for teams, so we need a central point for sharing and store code. We call it the remote.

After issuing the previous commands, if you want to make those modifications and code available, you must use the command **git push**.

```
git push
```

But something went wrong, as...

```
fatal: No configured push destination.
```

```
Either specify the URL from the command-line or configure a remote repository using

    git remote add <name> <url>

and then push using the remote name

    git push <name>
```

This is not bad and the message is clear: before publishing something from our local repository to a remote, we must set a remote! To do that we must have access to a Git repository service like GitHub, BitBucket or Code.UA.

Bellow, we show how to create a remote Git repository, with two use-cases: **GitHub** and **Code.UA**. The command **git push** should work normally after you complete this step.

2.5.1 Using Code.UA

For University of Aveiro course's projects, it's recommend to use Code.UA rather than third-party services, like GitHub or Bitbucket.

To create a repository in Code.UA:

1. Login with your universal account in **code.ua.pt**
2. In the top left corner, click in **Projects**
3. Click in **New project**
4. Fill in the required information (choose a good name)
5. Tick, in the **modules** section, the checkbox **Repository**
6. Choose, in the **SCM** list, the item **Git**
7. Confirm the repository creation by clicking in **Create**

Now that we created a new repository in Code.UA, we need to find repository URL, so we can publish the modifications made in our local repository in the remote one.

To do that:

1. In the top right corner, choose the project you newly created
2. Click in the tab **Repository**
3. Copy the url (starts with **https://(...)**)

Then, execute in the path of your local repository:

```
git remote add origin https://(...)
git push --set-upstream origin master
```

And it's done!

2.5.2 Using GitHub

GitHub is the most popular Git repository hosting service. You can and should use it to publish your projects, since GitHub is almost known as the "Programmer's portfolio".
To use GitHub:

1. Create an account on GitHub (on `https://github.com/join?return_to=https%3A%2F%2Fgithub.com%2Faccount&source=login`)
2. After registering and login in, push the **New repository** button
3. Choose a good name for the repository
4. Make the project as **public** or **private**
5. Push the **Create repository** button

After creating a new repository, GitHub provides a set of instructions for creating a new repository on the command line, push an existing repository and more.

The repository URL it's available under the section **Quick Setup**.

Then, execute in the path of your local repository:

```
git remote add origin https://(...)
git push --set-upstream origin master
```

And it's done!

2.6 Creating a branch

GIT supports branches. What are branches? Branches are places where code diverges, meaning that the code will have the same base, but it will differ from branch to branch. Every GIT repository has main branch, that is called master and if you don't branch, all your commits will go there.

Now let's say you need to add a new feature. You don't want to break the master branch while you develop your new feature, so you will create a new branch so that you can work on code. To create the new branch, you will use the following command:

```
  git branch <branch name>
```

Let's create a new branch for our new feature called **feature-1**. After creating your new branch to work on you should check in which branch are you in, by running the following command **git branch**. And you will get the following result:

```
    feature-1
  * master
```

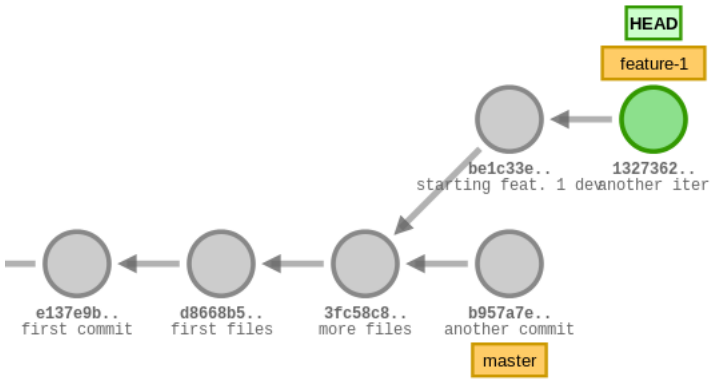

Figure 3: Repository after creating branch feature-1

The star (*) before the name of the branch will tell you in which branch you are and right now you should be in the master branch. But we want to develop our new feature, so we will need to change to the feature-1 branch. To do that we need to run the following command:

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

After we run this command you should check in which branch are you now and the result should be feature-1 as demonstrated in the following:

```
* feature-1
  master
```

Now we can develop our feature in this branch, without changing the master branch.
Let's add a new file called feat_1 and let's add and commit our new feature to the repository. Let's do this by running the following commands:

```
echo "feature 1 developed" >> file_1
git add file_1
git commit -m "Feature 1 complete"
git log
```

And the output should be similar to this:

```
commit 43756632d9dedf6b8756e1dd393d877ee7c81a4d
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 16:55:18 2016 +0000

   Feature 1 complete

commit 3070d3621b60b5ebe46b2d58ad6be2537069d79d
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:44:20 2016 +0000

   Added initial files

commit 5814279838033e7b0f14a620c73202f52f11cf99
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:26:39 2016 +0000

   Initial commit
```

So now our code in feature-1 branch as a new feature that the master branch doesn't have. Let's just check our master branch to check that the change that we made didn't made it to the master branch, by running the following commands:

```
git checkout master
git log
```

The result should be similar to this:

```
commit 3070d3621b60b5ebe46b2d58ad6be2537069d79d
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:44:20 2016 +0000

   Added initial files

commit 5814279838033e7b0f14a620c73202f52f11cf99
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:26:39 2016 +0000

   Initial commit
```

As we can see in the last output, the master branch didn't get changed, while the changes we did are saved in the feature-1 branch.

2.7 Merge vs Rebase

Run this code in your terminal so that the repository matches the next exercise:

```
git checkout master
git branch bug_fix_1
git checkout bug_fix_1
echo "this is a bug fix" >> file_1
git commit -a -m "Bug Fix 1 solved"
git checkout master
```

Now we want to do a lot of things with our repository. We have two things happening, we developed our feature 1 and one of our colleagues developed a bug fix for a bug found in the master branch. Our colleague did his bug fix on the branch bug_fix_1.

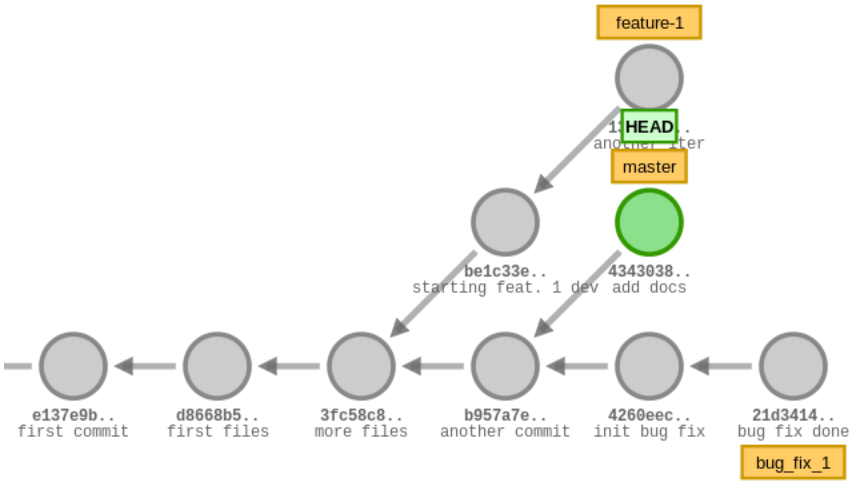

Figure 4: State of the repository after another branch has been created

There are two kinds of operations:

- **Merge** - this operation will join two commits from different branches in a new commit in the branch you are
- **Rebase** - this operation will take the commits in one branch and apply them before the commits of the branch you are in

2.7.1 Merge

So now we want to merge the bug fix into the master branch because we want our master branch to have the less amount of bugs possible. To do this, first we must check if we are in the master branch and then merge the bug_fix_1 branch into the master branch. To check if we are in the correct branch we need to run **git branch** and the output is:

```
  bug_fix_1
  feature-1
* master
```

If this isn't your case you need to run **git checkout master**.

After doing all this, the last thing that we need to do to merge both branches is:

```
git merge bug_fix_1
```

Since we merged the two branches, both of them should be equal. To check that all the commits have made into master branch let's check the log, by running **git log**:

```
commit 3b820eeb454915a81373a569a8a4baf32a6725b1
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 21:31:43 2016 +0000

    Bug fix 1 solved

commit 3070d3621b60b5ebe46b2d58ad6be2537069d79d
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 15:44:20 2016 +0000
```

```
    Added initial files

commit 5814279838033e7b0f14a620c73202f52f11cf99
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 15:26:39 2016 +0000

    Initial commit
```

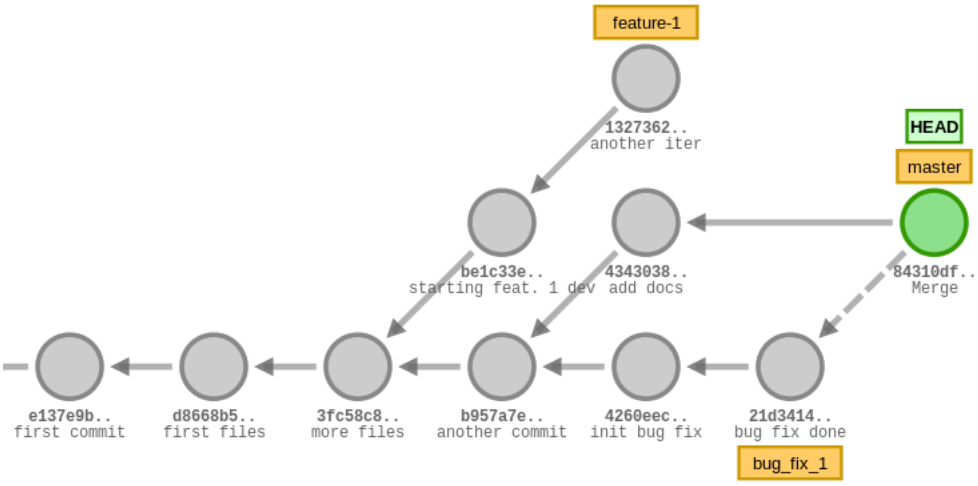

Figure 5: Repository state after the merge

And let's check if the last commit is the same in both branches, by running **git branch -v**:

```
  bug_fix_1 3b820ee Bug fix 1 solved
  feature-1 4375663 Feature 1 complete
* master   3b820ee Bug fix 1 solved
```

Now that the bug_fix_1 branch has been merged into the master branch, we don't need that branch anymore, so let's delete it. To do this run:

```
git branch -d bug_fix_1
```

And now let's check that we did in fact deleted the bug_fix_1 branch and that we didn't lose any information by running again **git branch -v**:

```
  feature-1 4375663 Feature 1 complete
* master   3b820ee Bug fix 1 solved
```

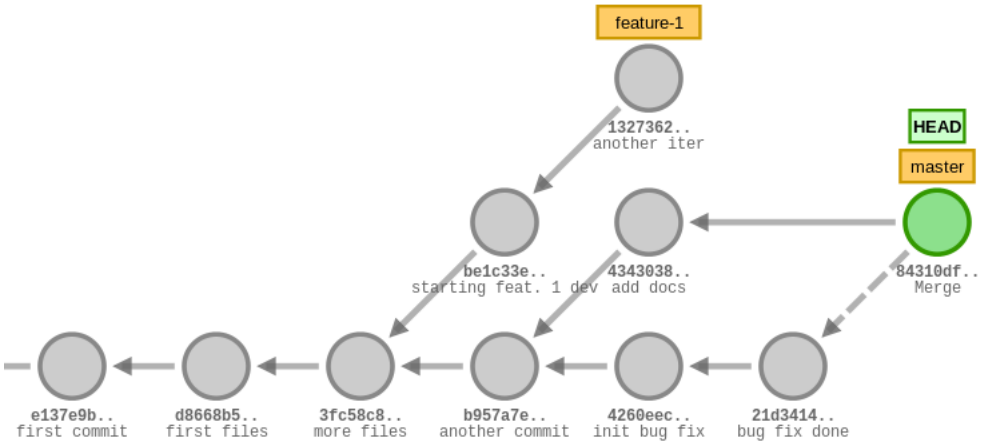

Figure 6: Repository state after deleting branch

2.7.2 Rebase

Now we want to update our feature-1 branch with the latest code from the master branch, to check if the bug fix that we merged into the master branch won't insert any bugs when merged with our new feature. To do this we must change to our feature-1 branch and then rebase it with the commits from the master branch.

```
git checkout feature-1
git rebase master
```

And the output should be something like this:

```
Auto-merging file_1
CONFLICT (content): Merge conflict in file_1
Recorded preimage for 'file_1'
Automatic merge failed; fix conflicts and then commit the result.
```

Since we changed the same file in both branches, we need to fix this. Open the file with vim, using the following command **vim file_1**. The file should be similar to this:

```
<<<<<<< HEAD
feature 1 developed
=======
this is a bug fix
>>>>>>> master
```

Now we must decide what to keep, so let's fix the file, by keeping what we want to keep and removing what is not needed, including **<<<<<<<HEAD** and **>>>>>>>master**. By the end of this conflict resolution, we should have a file equal to this:

```
this is a bug fix
feature 1 developed
```

Now let's add the file and commit the rebase fix:

```
git add file_1
git commit -m "Fixing rebase errors"
```

Now let's check the history of this branch by running **git log**.

```
commit c7cbf0e80fb8564f5cfa5a9f7d20f2002fb7847b
Merge: 76a5e9e 3b820ee
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 23:59:40 2016 +0000

    Fixing rebase errors

commit 76a5e9efc2c59ff487fae6636c9eeb80720ab541
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 23:49:28 2016 +0000

    Feature 1 complete

commit 3b820eeb454915a81373a569a8a4baf32a6725b1
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 21:31:43 2016 +0000

    Bug fix 1 solved

commit 3070d3621b60b5ebe46b2d58ad6be2537069d79d
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 15:44:20 2016 +0000

    Added initial files

commit 5814279838033e7b0f14a620c73202f52f11cf99
Author: John Doe <john.doe@example.com>
Date:   Tue Nov 15 15:26:39 2016 +0000

    Initial commit
```

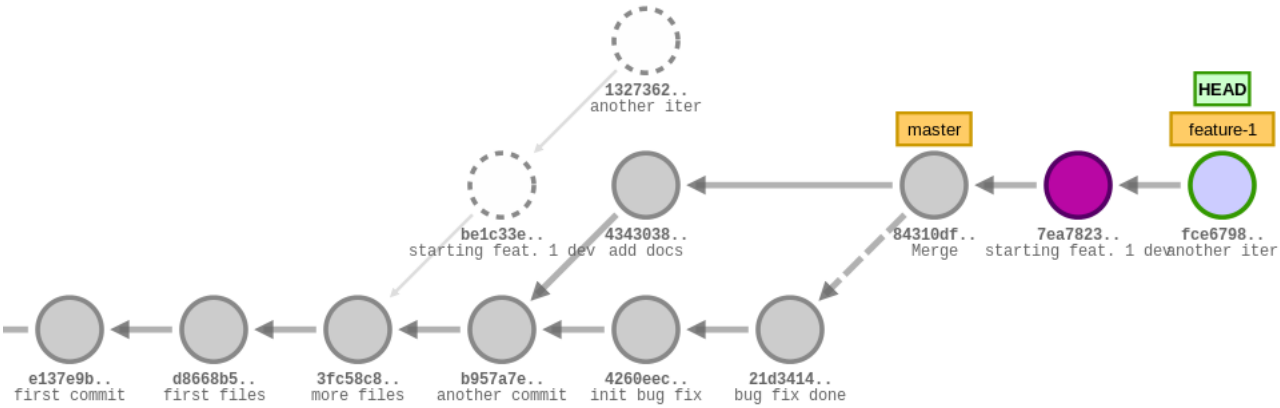

Figure 7: Repository state after rebase

2.8 Reset

Let's say that we did an error when we fixed the rebase error, how can we undo it? Simple we can use **git reset HEAD N**, where **N** represents the number of commits that we want to reset to. Since we only want to undo the last commit we will use N = 1.

```
git reset HEAD~1
```

To check that the **git reset** was successful, let's run **git log**.

```
commit 76a5e9efc2c59ff487fae6636c9eeb80720ab541
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 23:49:28 2016 +0000

   Feature 1 complete

commit 3070d3621b60b5ebe46b2d58ad6be2537069d79d
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:44:20 2016 +0000

   Added initial files

commit 5814279838033e7b0f14a620c73202f52f11cf99
Author: John Doe <john.doe@example.com>
Date:  Tue Nov 15 15:26:39 2016 +0000

   Initial commit
```

We can now check that the rebase never happened.

3 Advanced tricks

3.1 Cherry-pick

3.2 RefLog

3.3 Bisect

3.4