2. Documentation and Version Control Systems

Miquel Àngel París i Peñaranda

Web Application Deployment

2nd C-VET Web Application Development

Index

The Git Version Control System.

1. Getting Started

In this practical case, we will cover the concepts discussed about the distributed version control system Git.

Step 1. Installation

On any GNU/Linux system based on Debian/Ubuntu, we can install Git from the terminal using the apt command.

As always, we first refresh the package list:

```
$ sudo apt update
```

And then install the Git package:

```
$ sudo apt install git
```

Step 2. Checking the Version

To know which version we have installed, we will use:

```
$ git -version
```

Output:

```
root@server:/home/user# git --version
git version 2.34.1
root@server:/home/user#
```

Step 3. Accessing Help

If we type the `git` command without any arguments from the console, we will see a help menu with the supported parameters and the most commonly used commands, along with a brief explanation.

On the other hand, if we need help with a specific Git command, we can use:

```
$ git help <command>
```

This will give us help for the specific command. Additionally, we can also consult the Git manual pages in Linux environments using `man git`.

Step 4. Configuring Git

After installing Git, we will need to configure some parameters. We will use the command `git config`, which allows us to save configurations at the system, user, or project level.

We will work at the user level, so the configuration will be saved in a hidden directory called `.gitconfig` in our home folder. In general, to set a parameter at the user level, we will use:

```
$ git config --global <parameter> <value>
```

We can check the settings with:

```
$ git config --global <parameter>
```

Next, we will see the different parameters to configure in Git and how to do it:

User identity, consisting of the name and email address:

```
$ git config --global user.name "DAW Student"
```

```
$ git config --global user.email student.daw.@alu.edu.gva.es
```

Default editor for when Git needs us to write a message (for example, the commit message):

```
$ git config --global core.editor nano
```

There are other useful settings as well:

To enable Git to use colored output:

```
$ git config --global color.ui true
```

To instruct Git to convert line endings when working in hybrid Linux/Windows/Mac environments:

```
$ git config --global core.autocrlf true
```

With this option enabled, you may see the following warning when committing a change to the repository:

```
warning: LF will be replaced by CRLF in file.md.
The file will have its original line endings in your working
directory.
```

This indicates that Linux line endings (LF, Line Feed) will be replaced by Windows line endings (CR, Carriage Return, and LF, Line Feed). If you are not sharing code with users of other operating systems and this warning is bothersome, you can keep the option set to `false`.

Finally, if you want to view the entire list of parameters, use:

```
$ git config -list
```

Output:

```
root@server:/home/user# git config --list
user.name=DAW Ruben
user.email=rubenfm2004@gmail.com
core.editor=nano
core.autocrlf=true
color.ui=true
root@server:/home/user# _
```

2. Getting Started with Repositories

To work with a Git repository, the first thing we need to do is initialize it. We can either clone an existing repository or create a new one. For this example, we will create a new repository.

Step 1. Creating and Initializing the Project Folder

First, we open the terminal, navigate to the directory where we want to store the project, and create the project folder:

```
$ mkdir project
```

Now, we navigate into it:

```
$ cd project
```

And initialize the repository here:

```
$ git init
```

Output:

```
root@server:/home/user# mkdir project
root@server:/home/user# cd project/
root@server:/home/user/project# git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/user/project/.git/
```

As you can see, Git informs us that it has created an empty repository.

To check the repository's status, we type:

```
$ git status
```

Output:

```
root@server:/home/user/project# git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

This indicates that we are on the master branch, no commits have been made yet, and nothing is staged for commit.

If we list the hidden files in the current folder (using `$ ls -la`), we will see a `.git` folder that contains all the information related to version control.

Step 2. Creating Content

Now, let's create a new file using the `touch` command:

```
$ touch file1.md
```

Remember, the `touch` command creates an empty file with the specified name without needing to open an editor to add content.

If we check Git's status again, it will show:

```
$ git status
```

Output:

```
root@server:/home/user/project# touch file1.md
root@server:/home/user/project# git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file1.md

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

As we can see, `file1.md` is marked as **untracked**, which means this file is not under version control yet.

Step 3. Tracking the File

To track the file in the version control system, we need to add it using:

```
$ git add file.md
```

If we now check the repository's status:

Output:

```
root@server:/home/user/project# git add file1.md
root@server:/home/user/project# git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   file1.md
```

We can see that `file1.md` is now a new file in the repository and is staged for the next commit.

Step 4. Committing the File

To commit the file to the repository, we will make our first commit:

```
$ git commit -m "Adding the first commit"
```

Output:

```
root@server:/home/user/project# git commit -m "Adding the first commit"
[master (root-commit) 3a19b43] Adding the first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.md
```

We use the `-m` parameter to specify the commit message. If no message is provided here, the default editor will open to ask for a commit message.

Notice that the first line shows the SHA-1 hash of this commit: `c7e8b82`.

If we check the repository's status now:

```
$ git status
```

Output:

```
root@server:/home/user/project# git status
On branch master
nothing to commit, working tree clean
```

This indicates that everything is up to date, and the working directory matches the repository.

We can also view a log of the commits using:

```
$ git log
```

Output:

```
root@server:/home/user/project# git log
commit 3a19b439b6f7f1390c6fd05516f1b4f3759c5b73 (HEAD -> master)
Author: DAW Ruben <rubenfm2004@gmail.com>
Date:   Mon Oct 21 14:19:47 2024 +0000

    Adding the first commit
```

Or more concisely with:

```
$ git log --oneline
```

Output:

```
root@server:/home/user/project# git log --oneline
3a19b43 (HEAD -> master) Adding the first commit
```

3. Deleting Files

Deleting files is a common task in projects. With Git, we can delete files in two ways: by deleting the file from the working directory and then from the repository, or by deleting it directly from the repository, which will also remove it from the working directory.

Let's start with the files `tmp1.md` and `tmp2.md`.

```
$ touch tmp1.md tmp2.md
$ git add tmp1.md
$ git add tmp2.md
$ git commit -m «Afegint fitxers temporals»
```

A. Deleting Locally and Committing to the Repository

For this example, let's delete the file `tmp1.md` created earlier.

Step 1. Delete the File Locally

First, we delete the file from the working directory:

```
$ rm tmp1.md
```

Then check the status:

```
$ git status
```

Output:

```
root@server:/home/user/project# touch tmp1.md tmp2.md
root@server:/home/user/project# git add tmp1.md
root@server:/home/user/project# git add tmp2.md
root@server:/home/user/project# git commit -m "add temporal files"
[master fc0e77f] add temporal files
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tmp1.md
 create mode 100644 tmp2.md
root@server:/home/user/project# rm tmp1.md
root@server:/home/user/project# git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    tmp1.md

no changes added to commit (use "git add" and/or "git commit -a")
```

At this point, the file deletion is not yet staged for the next commit. We are shown two options to proceed:

`git add` or `git rm` to stage the deletion for the next commit.

`git restore` to discard the deletion and keep the file.

Step 2. Prepare for the Next Commit

We add the file to stage the deletion:

```
$ git add tmp1.md
```

Check the status:

```
$ git status
```

Output:

```
root@server:/home/user/project# git add tmp1.md
root@server:/home/user/project# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    tmp1.md
```

The deletion will be confirmed in the next commit. We could undo the deletion using `$ git restore --staged tmp1.md` at this point.

Step 3. Confirming the Changes

Finally, we confirm the deletion in the repository:

```
$ git commit -m "Deleted tmp1.md"
```

Output:

```
root@server:/home/user/project# git commit -m "deleted tmp1.md"
[master 6ef4246] deleted tmp1.md
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 tmp1.md
```

B. Deleting Directly from the Repository

In this second option, we will delete the file directly from the repository, which will also remove it from the working directory.

Step 1. Delete the File from the Repository

To delete the file from the repository, we use:

```
$ git rm tmp2.md
```

If we check the file's status now, we see that we only need to commit to apply the deletion.

Output:

```
root@server:/home/user/project# git rm tmp2.md
rm 'tmp2.md'
```

Step 2. Confirm the Changes

As usual, we confirm the changes with:

```
$ git commit -m "Deleted tmp2.md"
```

Output:

```
root@server:/home/user/project# git commit -m "deleted tmp2.md"
[master 4e1fa59] deleted tmp2.md
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 tmp2.md
```

4. Moving Files

To move or rename files, we can use the same methods as for deletion: first removing the file from the working directory and then adding the changes for the next commit, or we can do it directly.

As a preliminary step, let’s create a file that we will move in the working directory and add it to the repository:

```
$ touch tmp_mv.md
$ git add tmp_mv.md
$ git commit -m "created tmp_mv.md"
```

A. Local Rename and Extension to the Repository

Step 1. Local Rename

To rename the file, we use the `mv` (move) command:

```
$ git mv tmp_mv.md tmp_mv_1.md
```

If we check the VCS (Version Control System) status:

```
$ git status
```

Output:

```
root@server:/home/user/project# git mv tmp_mv.md tmp_mv_1.md
root@server:/home/user/project# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    tmp_mv.md -> tmp_mv_1.md
```

As we can see, it indicates that the file `tmp_mv.md` has been deleted, and a new untracked file has been created, `tmp_mv_1.md`. Essentially, a renaming operation could be considered as the deletion of the original file and the creation of a new one with the same content.

Step 2. Add New Files and Changes in the Next Commit

Now we need to add the deletion of the old file for the next commit:

```
$ git add tmp_mv.md
```

As well as adding the new file to version control:

```
$ git add tmp_mv_1.md
```

With this, the VCS status will be:

```
$ git status
```

Output:

```
root@server:/home/user/project# git add tmp_mv.md
fatal: pathspec 'tmp_mv.md' did not match any files
root@server:/home/user/project# git add tmp_mv_1.md
root@server:/home/user/project# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    tmp_mv.md -> tmp_mv_1.md

root@server:/home/user/project# _
```

Now, it indicates that the file has been renamed.

Step 3. Commit the Changes

To commit the changes:

```
$ git commit -m "Renamed tmp_mv.md to tmp_mv_1.md"
```

B. Direct Rename in the Repository

Following the previous example, let's create the file `tmp_mv.md` again to rename it as `tmp_mv_2.md`:

```
$ touch tmp_mv.md
$ git add tmp_mv.md
$ git commit -m "Added tmp_mv.md for the second rename"
```

Step 1. Move the File Directly in the VCS

To rename the file directly in the VCS, we use `git mv`:

```
$ git mv tmp_mv.md tmp_mv_2.md
```

This is equivalent to having renamed the file locally and incorporated the changes and generated files for the next commit. If we check the VCS status:

```
$ git status
```

Output:

```
root@server:/home/user/project# touch tmp_mv.md
root@server:/home/user/project# git add tmp_mv.md
root@server:/home/user/project# git commit -m "added tmp_mv.md for the second rename"
[master d770e99] added tmp_mv.md for the second rename
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tmp_mv.md
root@server:/home/user/project# git mv tmp_mv.md tmp_mv_2.md
root@server:/home/user/project# git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    tmp_mv.md -> tmp_mv_2.md
```

We can see that we are at step 2 from the previous section. Now, we just need to confirm the changes.

Step 2. Confirming the Changes

```
$ git commit -m "Renamed tmp_mv.md to tmp_mv_2.md"
```

5. Undoing Changes Between the Staging Area and Working Directory

When we have changes pending in the staging area (staged), we can undo them using the Git subcommand `checkout`.

For example, let's modify the file `tmp_mv_1.md` and add a line (we will use the `echo` command with redirection to add lines to the file):

```
$ echo "Test to undo changes" >>  tmp_mv_1.md
```

If we check the VCS (Version Control System) status now:

```
$ git status
```

Output:

```
root@server:/home/user/project# echo "test to undo changes" >> tmp_mv_1.md
root@server:/home/user/project# git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   tmp_mv_1.md

no changes added to commit (use "git add" and/or "git commit -a")
```

We will see that the file is marked as modified. However, if we want to undo these changes, as Git suggests, we can use the command `git restore <file>` to restore it:

```
$ git restore tmp_mv_1.md
$ git status
```

Output:

```
root@server:/home/user/project# git restore tmp_mv_1.md
root@server:/home/user/project# git status
On branch master
nothing to commit, working tree clean
```

The `git restore` command was introduced in Git version 2.23 (2019). Previously, the `git checkout` command was used (which can still be used), and it was also used to work with different branches. With the introduction of `git restore` and `git switch` in version 2.23, this ambiguity was resolved.

6. Other Interesting Operations

A. Ignoring Files in Git: The `.gitignore` File

Often, we do not want certain types of files to be saved in the repository: executables, compressed files resulting from packaging, library directories, directories or files containing configuration, passwords, or connection parameters, etc.

For this, we use the `.gitignore` file, which is saved in the root of the working directory. In this file, we specify the different filename patterns that should never be included in version control.

This file contains several lines following these rules:

Each line contains a pattern.

Blank lines can be used as separators to improve readability.

Comments start with `#`.

The `!` prefix will include the specified files in Git (override the ignore rule).

The wildcard `*` can be used to indicate any sequence of characters.

The question mark `?` replaces any single character.

Regular expressions like `[0-9]`, `[ao]` can be used.

Here is an example of a `.gitignore` file:

```
# Ignore the files named temporal_6.txt and temporal_7.zip
temporal_6.txt
temporal_7.zip
# Ignore files with the extensions zip, gz, changes, or deb:
*.zip
*.gz
*.changes
*.deb
# Ignore log files in the log folder,
# as well as extensions log0, log1, log2, etc.
log/*.log
log/*.log[0-9]
# Ignore all files in the images directory
imagenes/*
# Ignore all files ending in 'a' or 'o'
# in the compiled directory
compilados/*[ao]
```

Now, we just need to add this file to version control and commit the changes.

B. Reverting to a Previous Commit

To revert to the state of a previous commit, we use the `git revert` command as follows:

```
git revert <SHORT SHA-1 OF COMMIT TO REVERT>
```

To successfully apply `git revert`, there must be nothing in the staging area. Additionally, you will need the selected editor (vim, nano, etc.) installed to avoid errors.

Let's see an example.

Step 1. Check the different commits made:

As we've done before, to see the commits made, we use the command:

```
$ git log --oneline
```

Output:

```
root@server:/home/user/project# git log --oneline
c1c5663 (HEAD -> master) renamed tmp_mv.md to tmp_mv_2.md
d770e99 added tmp_mv.md for the second rename
3815cbf renamed tmp_mv.md to tmp_mv_1.md
809f9ac create tmp_mv.md
4e1fa59 deleted tmp2.md
6ef4246 deleted tmp1.md
fc0e77f add temporal files
3a19b43 Adding the first commit
```

Step 2. Locate the commit where the change occurred and revert it:

If we want to recover the file `tmp1.md` that was deleted in commit `6dc5cad`, we would do:

```
$ git revert 6dc5cad
```

You will be asked for a new message for the log, and a new commit will automatically be created with this message and the applied changes.

If we now inspect the commit log again:

```
$ git log --oneline
```

Output:

```
root@server:/home/user/project# git log --oneline
8cb5874 (HEAD -> master) Revert "deleted tmp1.md"
c1c5663 renamed tmp_mv.md to tmp_mv_2.md
d770e99 added tmp_mv.md for the second rename
3815cbf renamed tmp_mv.md to tmp_mv_1.md
809f9ac create tmp_mv.md
4e1fa59 deleted tmp2.md
6ef4246 deleted tmp1.md
fc0e77f add temporal files
3a19b43 Adding the first commit
root@server:/home/user/project#
```

C. Removing Untracked Files

We can remove untracked files (files not under version control) from the working directory using the command:

```
$ git clean -f
```

For example, if we create three files:

```
$ touch f1 f2 f3
```

And check Git's status:

```
$ git status
```

Output:

```
root@server:/home/user/project# git clean -f
root@server:/home/user/project# touch f1 f2 f3
root@server:/home/user/project# git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        f1
        f2
        f3

nothing added to commit but untracked files present (use "git add" to track)
root@server:/home/user/project#
```

We can clean untracked files with:

```
$ git clean -f
```

If we also want to remove unused directories, we would use:

```
$ git clean -f -d
```

And check Git's status:

```
$ git status
```

Output:

```
root@server:/home/user/project# git clean -f
Removing f1
Removing f2
Removing f3
root@server:/home/user/project# git clean -f -d
root@server:/home/user/project# git status
On branch master
nothing to commit, working tree clean
root@server:/home/user/project#
```