Git 101: Understanding Git Workflow

BSFM - Quant Division

by Gallo Alessandro

October 2024

The Purpose of this Guide

This paper serves as a beginner-friendly guide to understanding the basic concepts and operations of Git, a widely used DVCS *(Distributed Version Control System)* that intelligently tracks changes in files. This tool is especially valuable when multiple team members are simultaneously making changes to the same project, and understanding the basic commands of this instrument is essential for the optimal organization of a project.
In the following pages, you will grasp the key concepts of Git, like creating repositories, copying projects locally to work on them, creating branches for additional features and merging all the updates to the main branch. By the end, readers will develop a solid foundation in Git's core workflow and acquire the skills needed to collaborate effectively, track project history, and manage code versions efficiently.

1 VCS: the safeguardian of your code

A version control system, also known as *VCS*, tracks the history of changes as individuals and teams collaborate on projects, making it easier to manage software versions, collaborate efficiently and ensure code stability. With a *VCS*, developers can recover any previous version of the project at any time while making changes to it, saving them from the common issue of having to recover important deleted parts of the code. There exist two primary types of *VCS*:

- **Centralized VCS (*CVCS*)**: in *CVCS*, there is a single server that stores all the versioned files and developers check out files to work on them (e.g Apache Subversion and Perforce).
- **Distributed VCS (*DVCS*)**: in *DVCS*, every developer has a local copy of the entire project history, allowing them to work independently without needing constant access to the server (e.g Mercurial and Git).

As can easily be deduced, *DVCS* is the most advanced type of version control that has gained popularity in modern development practices. Unlike *CVCS*, where developers must connect to a central server to retrieve project history and make updates, in *DVCS*, every developer has a complete copy of the project repository, including its entire history, on their local machine, making collaboration more flexible and offering greater control over versioning.
Among all the DVCS, Git is the one that is most distinguished from the others. Created by Linus Torvalds in 2005, Git offers a lightweight, fast, and highly flexible approach to version control, making it the tool of choice for both open-source and enterprise-level projects. The primary features of Git include:

- **Branching and Merging**: Git's branching model is lightweight and efficient. Developers can easily create, switch, and merge branches, allowing them to work on different features or bug fixes in isolation without affecting the main codebase. Merging allows teams to bring together all changes seamlessly.
- **Staging Area**: Git introduces a staging area where changes can be reviewed and selectively staged before committing them. This allows developers to precisely manage which changes are included in a commit and when those changes are committed.
- **Tracking Changes**: Git keeps a detailed history of all changes, allowing developers to see who made changes, when, and why. This is valuable in understanding the evolution of the codebase during its life cycle.
- **Fast Performance**: Git is designed to handle everything from small projects to large-scale repositories with millions of files efficiently. It performs local operations like cloning, committing and branching with exceptional speed.

Although the learning curve of Git may seem quite steep and could scare anyone who approaches the tool, with this paper, the reader will be able to make the most of all the commands offered by Git to collaborate effectively and maintain high-quality code.

2 Getting Git Ready: Download and Setup

Before starting with Git's basic commands, we need to install them on our device. This guide will cover all the principal *Operating Systems* (Windows, MacOS and Ubuntu).

2.1 Download for Windows

The downloading process is the easiest among the three operative systems. Windows users can easily download Git from the web page below and follow the instructions to download the latest version: https://git-scm.com/downloads
After downloading the latest version of the installer execute the program, follow the instructions and Git will be installed on your machine.

2.2 Download for MacOS

Unlike the downloading process for Windows, the procedure for MacOS is slightly more complex. Below will be presented the two principal methods to download Git:

Install homebrew if you don't already have it, then execute the following command on MacOS Terminal::

```
$ brew install git
```

Install MacPorts if you don't already have it, then execute the following command on MacOS Terminal:

```
$ sudo port install git
```

It's important to know that these are only two of the possible methods to install Git on MacOS. For further methods, check this link: https://git-scm.com/downloads/mac

2.3 Download for Linux

Last, let's understand how to install Git on Linux using the command line. For simplicity reasons, we will cover only the installation for the Ubuntu/Debian distribution:

```
$ apt-get install git
```

Obviously, Ubuntu is not the only Linux distribution. If the reader has a different distribution, they should check how to install Git on their specific distribution on this page: https://git-scm.com/downloads/linux

3 Setting Up your Git

After installing Git on your device, you need to set it up. First of all, you need the username and the email related to your GitHub account (don't worry if you don't have an account, you can sign up on the GitHub website[1]).

3.1 Setting Global Variables

Firstly, we need to verify if the current version of Git is the one we installed, and then we can insert the GitHub username and email to configure Git[2]:

```
# Check Git version
$ git --version
  git version 2.47.0.windows.1

# Set the Git username
# It must be your GitHub username
$ git config --global user.name <your-github-username>

# Confirm that you have set the Git username correctly
$ git config user.name
  <your-github-username>

# Set the Git email
# It must be your GitHub email
$ git config --global user.email <your-github-email>

# Confirm that you have set the Git email correctly
$ git config user.email
  <your-github-email>
```

3.2 Personal Access Token

Before starting with Git workflow and the basic commands, we need to focus on the authorization process. To interact with the GitHub repository using Git with the command line and use commands that can alter the remote repository, we should authenticate with a specific code (it might seem a bit excessive, but this is the price to pay to keep all your *"precious"* university projects safe).

[1]For the official GitHub registration guide click here

[2]The commands used in the following example are the same for every *Operating System*. Windows is used only for convenience, and the output of the commands can be different on different OS. Moreover, to be clear, the parts of the code denoted by `#` are comments, `$` denotes the input line of the Terminal and elements in `< >` indicate command parameters.

In GitHub, this authentication code is referred to as a *PTA* (Personal Access Token). It possesses the same capabilities to access resources and perform actions on those resources as the token owner and is additionally restricted by any scopes or permissions granted to the token. For example, if a token is configured with an admin scope, but the user is not an organization admin, the token will not give administrative access to the organization[3]. GitHub offers two types of PTA: ***Fine-grained PTA*** and ***classic PTA*** (this guide will only cover classic PTA for simplicity[4]).
To create a classic PTA, follow these steps:

1. Click on Settings after clicking your profile image in the upper-right corner.
2. Open the Developer Settings in the left sidebar.
3. Under the section for Personal access token, click on Tokens (classic) and generate a new token.

Once you have followed the guided procedure[5] and stored your PTA to use when Git asks for your password, you're ready to start learning the basic command of Git!

4 Basic Git Commands

Before diving into this section, let's imagine how useful it can be Git for everyday coding. Imagine you want to start a new coding project (the reason for this project doesn't matter; it could be a university assignment or starting a no-profit project to help cute cats). After writing several lines of code, fixing boring bugs and drinking tons of coffee, you finally have the first version of your code. Now, you want to modify some features of your code, but you want to avoid copying your files and ending up with tons of identical files. You also don't want to work on the same file, risking to modify something incorrectly. How could you do that? You can use Git and its functions!
Now, let's not waste more time in chit-chat and dive into the magical world of Git commands.

4.1 Initializing and Cloning of a Repository

Before working on our projects, we need to create a *local repository* on our device. But what is a repository? Specifically, a Git **repository** is a virtual storage for our project, it allows you to save versions of your code, which you can access when needed. However, repositories alone don't work if a branch is not created!
A Git **branch** is an independent line of development within a project, and every repository has its main branch. In addition, you can create additional branches at any checkpoint of the main branch to develop new versions and merge them to the main branch later (this feature will be covered later in this guide).

[3]For additional information about GitHub organizations click here.
[4]Additional information about Fine-grained PTA will be added to successive versions of this guide.
[5]For the official GitHub guide on PTA creation click here.

You can work with repositories by initializing a new one or cloning an existing one if it's already online. Let's analize both the methods:

- `git init`: This is a one-time command you use during the initial setup of a new repo. Executing this command will create a new `.git` subdirectory in your current working directory and will also create a new main branch. This command also has additional options that can be pretty useful for a developer; for example, by adding the option `-b` or `--initial-branch` at the end of the command, you can set a custom name for the project's main branch. Let's give some examples:

```
# Firstly, let's move to the project directory
# Change this name with the path of your directory!
$ cd <code-path>

# Execute this to initialize your repository
# without any additional option
$ git init

# You can also select the name of the main branch
# by adding -b and the name of the branch
$ git init -b <branch-name>

# Add -q to print only error message and
# suppress all the other outputs
$ git init -q -b <branch-name>
```

  To inspect and delve deeper into this command's additional options, write `git init --help` to be redirected to the command page of GitHub.

- `git clone`: If a project has already been set up in a central repository, the clone command is the most common and practical way for users to obtain a local development clone. Like `git init`, cloning is generally a one-time operation. Once a developer has received a working copy, all the operations are managed in their local repository. This command usually creates a copy or clone of remote repositories (both personal and of other users). To use it, you need to pass a repository URL to `git clone`. Git supports a few different network protocols and corresponding URL formats, like HTTPS and SSH[6]. For simplicity, only the HTTPS protocol will be used in the following examples:

```
# To clone a repository execute this command
$ git clone <repo-url>

# To clone this repository into a specific
# directory execute this command
$ git clone <repo-url> <directory>
```

[6]For additional information about cloning a repository click here.

```
# To clone the repository and only the n most
# recent commit history execute this command
$ git clone --depth=<n> <repo-url>

# To clone only a specific branch of a
# repository you can run this command
$ git clone --branch <branch-name> --single-branch
   <repo-url>

# This command may be helpful when
# you are developing multi-branch projects
```

To inspect and delve deeper into this command's additional options, write `git clone --help` to be redirected to the command page of GitHub.

4.2 Saving Changes

After a mind-blowing coding session and half a litre of coffee, you finally finished version 1.0 of your project and are ready to start the next version. However, before starting another inspired coding session, you want to save the current version and check it even after tons of changes in the future version. How can you do it? Using the committing feature of Git!
Saving changes on Git is pretty different from saving files on any other editor (e.g. Microsoft Word, PowerPoint, VS Code, etc.) since it usually means overwriting the previous version of a file with a new one. With any other *VCS*, especially Git, when we *"save"*, we don't overwrite the last version of our files. Moreover, we don't directly save modifications with one single command, instead, we need to write multiple commands, but before discussing this command, we need to introduce the Git staging area.
The Git **staging area** is an intermediate space where changes are gathered before committing them to the repository. It allows you to review and organize the modifications you want to include in the next saving (better called *commit*), giving you control over which changes are saved.
Now that we know what the staging area is, we can dive into the fundamental commands of this subsection:

- `git add`: This is the first command to execute to save new changes. The main purpose of this command is to transfer changes in the working directory to the staging area. Specifically, it is very useful for our workflow because it permits us to select the files to put in the staging area. This means you can make various edits to unrelated files, then go back and divide them into logical commits by adding related changes to the stage and committing them one by one. Now let's see how to use this command:

```
# After modifying our file we execute the following
# command to insert it into the staging area
$ git add <file-name>
```

```
# If you want to add every change you made
# you can execute this command
$ git add .
```

To inspect and delve deeper into this command's additional options, write `git add --help` to be redirected to the command page of GitHub.

- `git status`: This command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. The status output does not show you any information regarding the committed project history.
  The execution of this command is quite simple, just type `git status` to retrieve all the information you need about the staging area. For more details about the options for this command, type `git status --help` to be redirected to the command page on GitHub.
- `git restore` and `git reset`: Sometimes, you want to remove some files from the staging area, but the previous command can not be used to eliminate files in the staging area. To do this, we need `git restore` and `git reset`, two commands used to undo changes in your working directory and to restore files to a previous state. These commands allow you to discard changes made to tracked files, returning them to the state of the last commit.
  Specifically, using `git restore --staged` we can remove a specific file from the staging area without affecting the working directory. This is useful if you've accidentally staged a file or changes and want to *"unstage"* them without losing the actual changes in your file. To do it with more than one file, you can use `git reset` to clear the staging area. Essentially, these commands move the files back from the staging area to the working directory for further editing. Here is how to practically use these commands:

  ```
  # Once a file is added to the staging
  # area you can remove it using this
  $ git restore --staged <file-name>

  # If more files were added to the staging
  # area you can remove them with this
  $ git reset
  ```

  For more details about the options for these commands, type `git restore --help` and `git reset --help` to be redirected to the command page on GitHub.
- `git commit`: This can be considered the principal command to save code modifications when you use Git. Basically, it captures a snapshot of the project's changes stored in the staging area. These snapshots can be thought of as "safe" versions of a project that Git will never change unless you explicitly ask it to. Obviously, to store changes in the staging area,

you must execute the command `git add` first. It's important to note that once you execute this command, the modifications will not be sent to the online GitHub repository but will be saved on your machine. This feature is crucial because it permits the developer to work in an isolated environment instead of working directly on the remote repository. Now let's analyse how to use this command and its options:

```
# With this command you commit the staged snapshot,
# but it will launch a text editor similar to Vim
# and its usage won't be covered in this guide
$ git commit

# Adding -a you can commit every change noticed
# by Git without using git add before
$ git commit -a

# This is the suggested option for using commit.
# In fact, by adding -m you can avoid the
# opening of the text editor and
# set the message in the command line
$ git commit -m "<commit-message>"

# With this command you combine the advantages
# of the previous two
$ git commit -am "<commit-message>"

# Passing this option, you will modify the last
# commit instead of creating a new one.
# Staged changes will be added to the previous
# commit and the message won't be modified
git commit --amend

# By using -m, you can modify the commit
# message of the last commit along with
# the action performed by adding --amend
git commit --amend -m "<commit-message>"
```

For more details about the options for these commands, type `git commit --help` to be redirected to the command page on GitHub.

- `git log`: With this you can get a detailed list of past commits, including information like the commit hash, author, date, and commit message. This command is helpful for reviewing the history of changes, tracking the evolution of the project, and identifying specific points in the repository's timeline.

The execution of this command is quite simple, just type `git log` to retrieve all the information you need about the commit history. For more details about the options for this command, type `git log --help` to be redirected to the command page on GitHub.

- `git push`: This command is used to upload local repository content to a remote repository. It transfers changes from your local branch to the corresponding branch on the remote server, making your work available to others. Typically, after committing changes locally, you use `git push` to share your updates with collaborators or back them up and synchronise your remote repository with your local one.
  This command presents complex options that will not be discussed in this guide, but for more details about the options for this command, type `git push --help` to be redirected to the command page on GitHub.

4.3 Hiding the Unnecessary: how to use .gitignore

Before delving into the next section of this guide, we must take some time to discuss an essential feature of Git: the `.gitignore` file. This file is a configuration file used to specify which files and directories should be ignored by Git when tracking changes. It tells Git not to include certain files in version control, preventing them from being committed to the repository. This is especially useful for excluding machine-specific files, such as temporary files or logs that are irrelevant to the project's source code.
The `.gitignore` file can be customized to ignore files based on patterns, such as specific file types, entire directories, or files with certain names. Each line in this file represents a rule for ignoring specific files, and Git will apply these rules when performing version control operations like staging or committing.
The `.gitignore` file itself is typically committed to the repository, so all collaborators can benefit from the same exclusion rules, ensuring consistency across different development environments. It plays a key role in keeping the repository clean and focused only on relevant source code and assets.

5 Branching Out: Navigating Your Git Journey

Imagine you are working on a personal project to create the next revolutionary to-do list app, and it's going great. You've got the basics down, but now you want to add new features, like dark mode and an "infinite tasks" button. Git provides an intelligent way to keep your work organized using branches instead of mixing all your changes into the main code and risking chaos. By creating separate branches, you can work on different features independently and avoid breaking your main project. Now let's learn how to navigate through Git branches and merge changes without losing your mind:

- `git branch`: This command helps you manage branches. You can use it to create a new branch, list all existing branches, or delete old branches when you are done with them. Think of it as creating a safe workspace for each feature you want to develop to avoid unnecessary chaos. Now let's analyze how to use this command and its options:

  ```
  # With this command you can create a new branch
  $ git branch <branch-name>

  # If you want to list all the branches
  # you can use this command
  $ git branch

  # To list both local and remote branches use this
  $ git branch -a

  # If you want to rename the current branch
  # you can use this command
  $ git branch -m <new-branch-name>

  # With this, you can delete the specified branch
  # if it has already been merged with another branch.
  # Git ensures you will not lose important work.
  $ git branch -d <branch-name>

  # With this, you can force the deleting process,
  # imposing Git to delete also non-saved changes.
  $ git branch -d <branch-name>
  ```

  To inspect and delve deeper into this command's additional options, write `git branch --help` to be redirected to the command page of GitHub.

- `git checkout`: This command is used to switch between branches or restore files in your working directory. It allows you to move from one branch to another, updating the working directory to reflect the state of the target branch. Moreover, it can be used to switch between previous commits using the commit hash code. Now let's see how we can use this command:

  ```
  # With this command you can switch to another branch
  $ git checkout <existing-branch-name>

  # This creates a new branch and
  # immediately switches to it
  $ git checkout -b <new-branch-name>

  # This moves you back to the main branch
  # after working in a different branch
  $ git checkout main
  ```

```
# With this you can switch to a previous
# commit using its hash code
# You can get the hash code of the previous
# commits using the command git log
$ git checkout <commit-hash>
```

To inspect and delve deeper into this command's additional options, write `git checkout --help` to be redirected to the command page of GitHub.

- `git merge`: This is used to combine changes from one branch into another. It integrates the work done in a feature or separate branch into the current branch, allowing you to merge new features, fixes, or updates into the main project. Git attempts to merge the changes automatically, but if there are conflicting modifications, it will prompt you to resolve them manually. Now let's see how we can use this command:

```
# After sending the last commit for the branch
# you want to merge, execute these commands
$ git checkout main
$ git merge <new-feature-branch-name>

# Remember to delete the branch
# once you have merged it
$ git branch -d <new-feature-branch-name>

# If some conflict arises, you can abort
# the process using this command
$ git merge --abort

# Or you can fix the problems indicated,
# add the modifications to the staging area
# and retry to merge
```

To inspect and delve deeper into this command's additional options, write `git merge --help` to be redirected to the command page of GitHub.

Conclusions

If you have made it this far, it means you should now have all the basics to work on your project independently. In the future, we will add more sections on complex topics and explore the current ones in greater depth. For this reason, we invite you to follow us on LinkedIn and GitHub to stay updated.
For additional questions, I invite readers to contact me on Linkedin for any reason related to this guide.