Munich
Germany
Never up to date

Report

Git - Basics and More

General Use Cases and Background Knowledge

Paul Heidenreich

December 2, 2021

Preface

The intention of this document is not to write an absolute beginners guide for git or explain everything in detail again. Therefore, I recommend reading other articles or references throughout the text which helped me a lot to understand how Git really works under the surface.
This report is more a protocol of the problems I encountered when working with git so far. I summarize these problems in so called "User Stories" (they are highlighted in yellow within the text) and describe my solutions to cope with this tasks.

Contents

List of Figures

Abbreviations and Acronyms

e.g. for example

i.e. that is to say

1 Cloning and Creating Git Projects

This chapter describes the workaround to clone git repositories from a remote server or network as well as creating a brand new local git project on your machine, for example (e.g.) when you start your own new software project. Additionally, the proceeding of creating a remote repository of your newly created local git project on GitHub is presented.

1.1 Cloning a Git Repository from Remote

Cloning a git repository is straight forward. Open your terminal or command line and navigate to the location where you want to place the project on your computer's file system.
Before executing the famous **git clone** command take a short breath and ask yourself: "Does this git project has sub-modules and do I also want to clone them too?"
If you say yes of course, then execute in your terminal/command line

```
$ git clone --recurse-submodules <Path/URL/SSH of Git Project>
```

Otherwise, the **git clone** command simplifies to

```
$ git clone <Path/URL/SSH of Git Project>
```

In case, you did not know that the project you already cloned does contain submodules or you simply forgot to clone them in a hurry, there is absolutely no reason to get into panic;)
Keep clam, grab your keyboard and type

```
$ git submodule update --init --recursive
```

Here, the important thing to learn is to use the git way to solve problems. Do not get the idea of just deleting the newly cloned repository and clone it again with the right command!

User Story: *"How do I know if the project I want to clone has any submodules integrated?"*

The first option is to always clone the project with the recurse flag, that is to say (i.e.) submodules get cloned too:

```
$ git clone --recurse-submodules <Path/URL/SSH>
```

The second option is to clone the main project without the recurse flag and checking for the existence of any submodules afterwards by

```
$ git clone <Path/URL/SSH>
$ git submodule status
```

As a result you see all submodules of the project currently missing (there is a little minus sign infront of the SHA key). Clone these submodules into your project repository by executing

```
$ git submodule update --init --recursive
```

1.2 Creating a Local Git Repository inside a Project Folder

Each new git project starts with the **git init** command.

User Story: *"Create a new, local git repository on your machine for the project you want to start or for the one you already started, i.e. you have already produced some files and want to organize your working progress with git."*

In this case, navigate into the directory in your file system which will contain the top level resources of your project or where you have saved the your work done so far. Inside this directory execute

```
$ git init
```

As a result, your directory now contains a **.git** repository which contains everything needed to version your project and track your changes over time. Briefly said, the top level project folder now consists of two main parts:

1. the project's actual working tree, i.e. all the files, code and other work inside this folder and second,

2. the git repository which owns all the snapshots of your work that you put inside this folder at times (the so called commits). This is the **.git folder** inside your project folder created with the **git init** command.

Now, everything is ready to use all the other git functionality.

1.3 Adding a Remote Repository to an Existing Local Git Project

1.3.1 Local and Remote Git Repositories

Before walking through this little workaround, it is advisable to recall oneself into mind the main difference between a local and remote git repository.
As explained in Section 1.2 a local git repository consists of the project's working tree and the

git "archive" (.git folder) recording all the development and changes made so for. By contrast, a remote git repository does only consist of a copy of your local git "archive" made at a time. There is no working tree inside a remote git repository. However, it is possible, via **git clone**, see Section 1.1, to locally reproduce a working tree from this **.git** archive.
Summarizing it all up:

$$\begin{aligned} \textit{Git local Repo} &= \textit{Working Tree} + \textit{.git} \\ \textit{Git remote Repo} &= \textit{.git} \end{aligned}$$

A git remote repository, i.e. a git repository with no working tree, is also called a **bare** repository. Please note, that projects on GitHub are no standard, bare remote repositories. Actually, they are checked out projects belonging to a real remote .git folder somewhere on a server. Interacting with remote repositories containing a working tree can lead to strange behaviorism, see Section 1.3.2 as an example.

Let us have a closer look at this **.git** folder. In fact, the **.git** "archive" rally consists of two main git features. There is, on the one hand

- the **Staging Area** or **Index** and on the other hand we have
- the **git Commit Tree**.

Putting it all together, a whole local git project consists of the **Working Tree** plus the git **Staging Area**/**Index** plus the **Commit Tree**!
In analogy to the the famous OpenGL graphics pipeline, I call these three stages the **git Pipeline**. Since data is pumped from the Working Tree over the Index into the Commit Tree, this comparison seems valid. Chapter 2 introduces and describes the journey through the pipeline in detail. Here, the main thing to remember is, that a local git project has a working tree whereas a remote one does not and the .git folder contains the Staging Area and the Commit Tree!

1.3.2 GitHub as Remote Location

User Story: *"I want to crate a remote location for my software project on GitHub (or any other Server) and connect this repository with my local one."*

Creating a remote GitHub repository requires a GitHub account! Follow the steps below:

1. Go to your GitHub homepage and click the +-icon in the navigation bar (upper right corner) and select **New Repository**.

2. In the Repository Name field, enter a short, descriptive name. I prefer using the project's local name and no white spaces.

3. Enter a brief description. This is optional, but it helps users understand the purpose of your project when it appears in search results or in your repository list.

4. Keep the default Public setting for your project's visibility. If you select Private, the community won't be able to access your project repository.

5. Leave Initialize this repository with a `README` unchecked if you have already created a `README.md` file for your project.

6. Click the Create Repository button to create the new GitHub repository.

Afterwards, the process continues on your local machine at the top level of your local git project folder. Check the status of the remote locations connected with your project with **git remote**. It everything is as expected, add the newly created GitHub remote location to the project by using the `URL` or `SSH` address provided by the GitHub page:

```
$ git remote -v
$ git remote add <Name_of_your_local_Project> <URL/SSH from GitHub>
$ git remote -v  //to see if the new remote location is listed
```

Finally, set the upstream branch. That is, tell your local git repository the name of the remote one (see the second item when creating a remote GitHub project) and the commit-branch to which you want to push back new commits. In General, this branch is called master or main (by default):

```
$ git push --set-upstream <Remote_Repo_Name>
<master/other_branch_name>
```

Now, you are ready to push if there are unpublished, new commits:

```
$ git push
```

Sometimes, setting an upstream branch and pushing your local contents for the first time to GitHub may cause error messages like *fatal: refusing to merge unrelated histories* or *git error: failed to push some refs to remote*. Most of the time, this is due to adding files or some other new content directly on the GitHub web page. A **README.md** file for example. Then, we have actually two separate projects, the local one on your machine and the "local" one on the GitHub web page which started their lives independently from each other but refer to the same remote .git folder. The task is now to tell git, that these two projects are actually one and the same with no risk to enable a merge/pull or push request (in our case pushing local content to GitHub and pulling the README.md file created there). Of course, the git community has already encountered this problem. By executing

```
$ git pull <Project_Name> <remote_branch> --allow-unrelated-histories
```

where the remote branch is master or main you tell git that both projects are the same and pulling the README.md file is safe. See [1] for more information.
However, be very careful with this command and use it only if you definitely know what your are doing!!!

1.3.3 An Arbitrary Server as Remote Location

This is almost the same procedure as described in Section 1.3.1. The only difference is, that there exists probably no web page with a graphical GUI interface, like GitHub, that leads one through the process of creating a remote repository. Furthermore, there are also no additional surprises because we do not have a second, identical local git repository in our web browser (see Section 1.3.2).

First, navigate on the server's file system or to the network drive where you want to generate your remote project destination. Then create a **bare** git remote repository (i.e. a repository without a working tree) with

```
$ git init --bare
```

Afterwards follow the steps already described in Section 1.3.1:

```
$ git remote -v
$ git remote add <Name_of_your_local_Project> <Path_to_remotre_repo/
  URL_to_server>
$ git remote -v
$ git --set-upstream <Remote_Repo_Name> <master/other_branch_name>
```

That's it, now you can push, pull or clone.

1.4 SSH (Secure Shell) Keys

SSH (Secure Shell) connections are very practical to communicate with your remote location safely and without typing the password of your account each time you communicate with your remote server. You can create a SSH key to your GitHub account (where your remote git repositories are) and connect your local git project via a SSH key instead of the URL. Therefore, you have to create an SSH key on GitHub and register your local projects with this key.
This section is not about SSH keys in detail, only how to use them in the git context.

User Story: *"I want to replace the remote URL with an SSH key."*

First thing to do is to list the remote repository connected with your local project and in a second step, replace the URL to your remotes with the SSH key

```
$ git remote -v
$ git remote set-url origin <ssh-key, e.g. from GitHub>
```

Control the result, again with

```
$ git remote -v
```

Eventually, if e.g. **git push** does not work without typing the password, you have to add the SSH key and activate it. In a terminal or command line type

```
$ ssh-add
$ ssh-add-l (to verify everything)
```

On Windows systems make sure that the ssh-command is on your PATH variable so that it works in a CMD (for more Information on this google for "adding ssh to path variable").

Due to security reasons, this procedure of adding the SSH key and authorizing the connection with your finger print key has to be repeated each time the local system gets rebooted.

2 The (local) Git Forest;) - Working Tree, Staging Area and Commit Tree

As shortly mentioned in Section 1.3.1 the three major parts of your local git project are

- the Working Tree or all the files and code you actually write,
- the Staging Area or Index Tree where you prepare new content and changes of your work ready to commit and
- the Commit Tree resembling the whole history and development of your project.

These three items make up the local Git forest which versions and organizes a project in an ongoing process. The meaning of the Working Tree and the Commit Tree is obvious but that holds not for the Staging Area. I will not spend time here explaining the benefits of the Index Tree since it is not important to understand what follows. The interested reader may refer to Section 3.2 or [2].
This chapter is not meant to give another explanation of how to create new commits. The focus lies on these three trees and how you improve the maintenance and administration of the software development process with the help of Git.

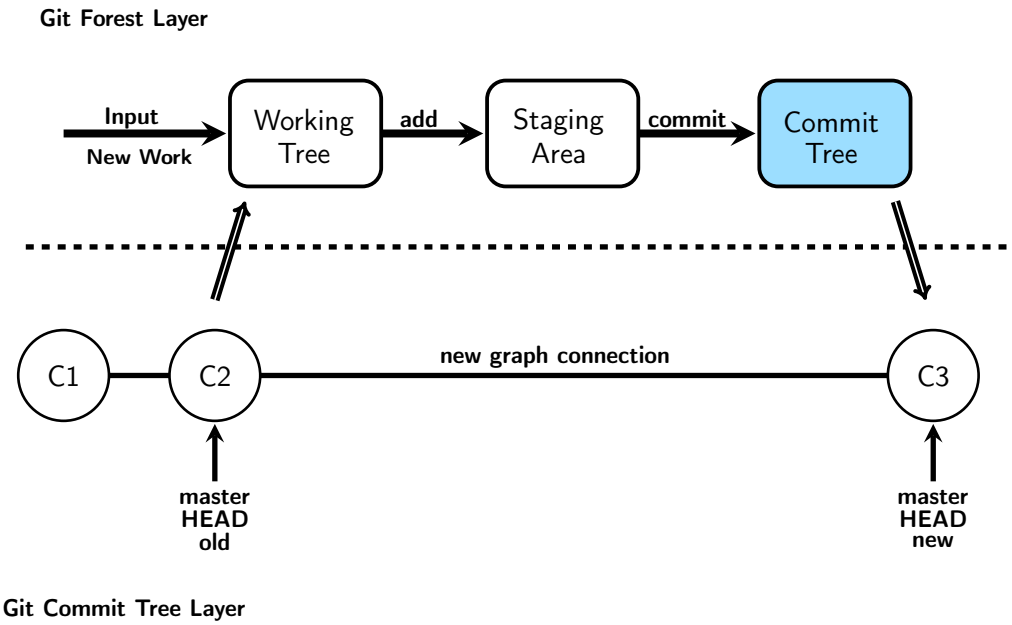

Fig. 2.1: The Git Pipeline: Working Tree, Staging Area and Commit Tree

2.1 Commit, HEAD and local Branches

This section is focused on summarizing briefly Git's meaning of commits, branches and referencing or addressing commits, respectively. We concentrate our investigations on the local .git repository. Concepts of fetch and merge, i.e. a pull operation and remote branching is part of ??.

Commits

Every commit represents a loaf in your commit tree. The whole commit tree represents the progress of your project. A single loaf or commit encapsulates your whole project at the time when the commit was made. That is, a commit is a snapshot of your project and you can extract that project state from just this single commit. Each commit is unique and accessible through its identifying SHA-1 hash key.

The notion of commits is fundamental when using git. **Git is all about commits!** Especially the high level end user API of git is all about creating commits, pointing to or referencing special commits, grouping commits together, looking inside a past commit, removing commits from your tree, restoring them etc. But always remember, one single commit represents and contains your whole project at the time the commit was made.

Next to the commits, Git offers a set of pointers which are directed to special commits by default. These pointers are for addressing commits and investigating your commit tree. The set consist

of:

- HEAD pointer
- branch pointers
- origin/master pointer (refers to your remote commit tree)
- master or main branch pointer

Sections xxx and yyy concentrate on how to work with these pointers using the checkout and reset commands.

Local Git Branches

The term branch often implies a somehow heavyweight and complex construct as part of the commit tree. In fact the contrary is the case. In the context of Git, a branch is nothing more then a named pointer directed on the latest commit in an directed, acyclic (sub-)graph of development in the commit tree. That's it! Again, a branch is only a named pointer to a commit and the branch name represents a colletion of all commits with the same anchestor in the tree. However, although we speak of pointers, these branch pointers are not meant to be moved freely around in your commit tree. They are more like refereces and always refer to the latest commit in such an acyxlic graph. In the following, the term branch means the whole acyclic (sub-)graph and the term branch pointer means the reference to the latest commit in this branch. One special branch is the master or main branch.

Master/Main Branch

The master or main branch reprents the trunk of your commit tree and is crated with the git init command. The master or main branch pointer is always directed to the latest commit added to this master branch. That is, creating a new commit also moves the pointer one commit further.

HEAD pointer, detached HEAD State

The HEAD pointer is also created with the git init command. At the beginning this pointer is attached to the master pointer and refers to the same commit. Attached means, that the HEAD pointer is moved together with the master pointer. However, HEAD is designed as a real pointer and meant to be moved around in your commit tree. It is the git tool to reference arbitrary commits. If the HEAD pointer (and only the HEAD pointer) is decoupled from the master pointer or any other branch pointer and moved backwards in time, i.e. to an older commit in the tree, one calls this state a detached HEAD. To sum it up, the HEAD is directed to the commit

which represents the the project state as you can see it, when you look in your current working tree.

origin/master

This pointer refers to the remote master branch inside the remote .git repository. Since this section only covers the local commit tree, the reader may consult chapter xxx for more information on remote HEAD and master as well as remote branching.
Finally, figurexxx shows a picture of the commit tree in relation with the other trees and references presented. It depicts the situation while adding a new commit and shows how new data gets assembled into a new commit.

2.2 Git Reset

2.2.1 Three Types of Git Reset

2.2.2 Reset with Path Specification

2.3 Git Checkout, Switch and Restore

2.3.1 Detaching the HEAD

2.3.2 Git Restore

Hello World!

3 Miscellaneous

3.1 Setting VS Code as Default Git Editor

VS Code is a very popular and handy editor. By editing git's global .config file it is possible to use VS Code as the default editor for writing new more verbose commits and to use it as git's merge and diff tool. The global .config file contains all important settings configuration parameter like your e-mail address and your name which appears when you write a new commit, for example. A display of this file is possible with

```
$ git config --global -e
```

The following steps are necessary to use VS Code as the default diff tool:

```
$ git config --global diff.tool vscode
$ git config --global difftool.vscode.cmd
  'code --wait --diff $LOCAL $REMOTE'
```

For using VS Code as the default merge tool execute the next two commands:

```
$ git config --global merge.tool vscode
$ git config --global difftool.vscode.cmd 'code --wait --diff $MERGED'
```

Again, check your global .config file and see the changes made so for. Now your are able to use VS Code as your editor of choice. To also use it as the standard commit tool set it as the global editor in general by typing

```
$ git config --global core.editor "code --wait"
```

That's it! VS Code is now your overall git editor.

3.2 The Benefits of having the Staging Area

References

[1] Git. *git-merge - Join two or more development histories together*. Nov. 20, 2021. URL: https://git-scm.com/docs/git-merge.

[2] sonulohani. *What's the use of the staging area in Git?* Ed. by Stackoverflow. Nov. 21, 2021. URL: https://stackoverflow.com/questions/49228209/whats-the-use-of-the-staging-area-in-git.