Munich
Germany
Never up to date

Report

Git - Basics and More

General Use Cases and Background Knowledge

Paul Heidenreich

January 10, 2022

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. Alongside to these stories, theory is given to understand the solutions presented.

The command line is the tool to work with Git in this document. I know that there exists a lot of graphical clients such as *Tortoise Git* or *GitKraken* [6, 3]. However, I recommend using the command line only. Then, you have to know and to understand what you are doing. This is especially helpful if you are still learning to work with Git effectively.

As pointed out at the beginning, the focus is directed to practical problems. In case you want to learn a little bit more of Git's magic, I recommend reading [7].

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

It is common practice to simply call the remote repository "origin" but you can also choose an arbitrary other name, say the project's name for example.

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

That's it, now you can push, pull or clone. Again, "origin" is the default name for the remote repository but if you prefer another name just replace "origin".

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 4.2 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.

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 Chapter 3.

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, cutting off 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 ancestor 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 references and always refer to the latest commit in such an acyclic graph. In the following, 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 represents 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, Fig. 2.1 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. The commit tree is highlighted in blue since we put new information up to this stage. The newly created commit C3 is also highlighted since it contains this new information next to everything which is inside C2 and C1.

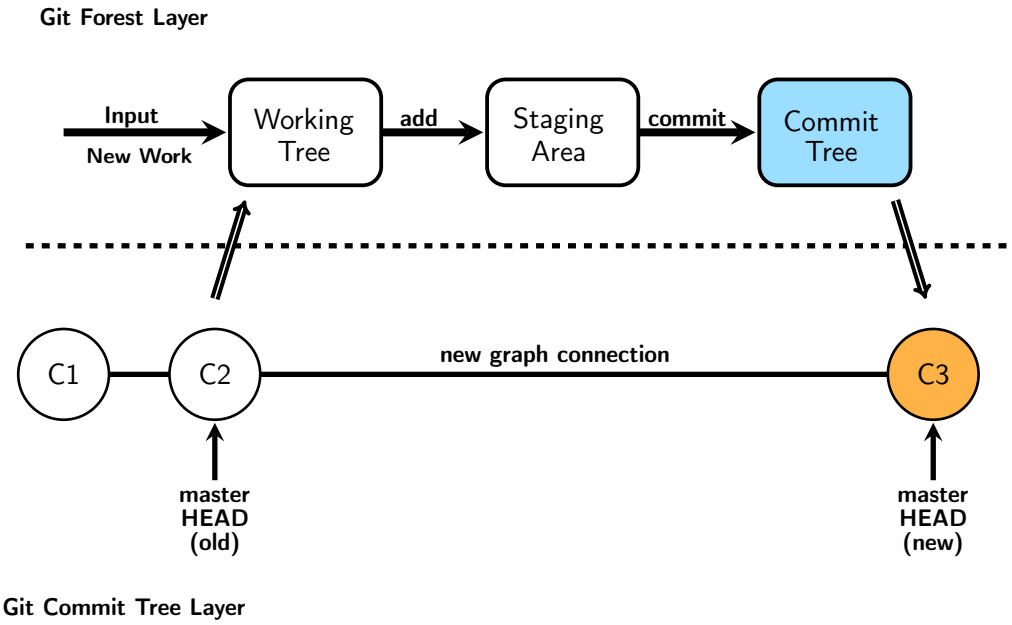

Fig. 2.1: Visualization of creating a new commit

After being familiar with the the Working Tree, the Staging Area, the Commit Tree, the pointer

Git uses and the branch expression, you should read [4]. The source explains in depth the meaning of Git resetxxx. What follows in Section 2.2 and Section 2.3 is a remainder of how to use git's reset and checkout commandsxxx.

2.2 Git Reset

Effectively using Git to organize your software projects requires at least a good understanding of the Git reset xxx command. After knowing how to create new commits with Git add xxx and Git commit, it is especially important to understand how you can revert these actions or even reset your project to a past, stable state. In other words, Git reset xxx moves the HEAD/master pointer to the specified commit.

The Git reset xxx command always refers to a commit specified by the user. Git offers several ways to address commits. The most common one is using a commit's SHA-1 hash key. You do not have to type the whole key, the first five to six signs are enough:

```
$ git reset <5-6 SHA1 digits>
```

Another method often used is giving the position of a commit relatively to the HEAD pointer in the commit tree. For example,

```
$ git reset HEAD~2
```

means commit C1 in Fig. 2.1. Specifying no commit at all like

```
$ git reset
```

is equivalent to

```
$ git reset HEAD~1
```

which is the same as

```
$ git reset HEAD~
```

Of course, these rules all apply for any other Git command requiring a commit as input argument.

2.2.1 Three Types of Git Reset

There are three different ways of moving the HEAD/master pointer using Git reset xxx:

- a soft reset,
- a mixed reset and
- a hard reset.

Git reset –soft

A git reset –soft xxx is more or less the opposite command to git commit xxx. Resetting the Commit Tree softly actually cuts off the latest commit, puts the information added with this commit back into the Staging Area and moves, of course, the HEAD/master reference. Basically, Fig. 2.2 illustrates this situation and the next use case summarizes this process. Again, the stage containing the latest modifications of the Working Tree is highlighted. Note, at this point, there exists no commit in the Commit Tree encapsulating theses changes.

User Story: *"I want to undo the last commit and put the files back into the staging area, that is perform a soft reset."*

According to the different ways of addressing commits, there are the following, equivalent command options:

```
$ git reset --soft <SHA1>   // or
$ git reset --soft HEAD~    // or
$ git reset --soft HEAD~1   // or
```

This is also possible for older commits by adjusting the SHA1 hash key or relative HEAD position. A soft reset with respect to commit C1 would put all uncommitted changes of C2 and C3 in the Staging Area or leave them untracked if the files were newly added in C2 or C3.

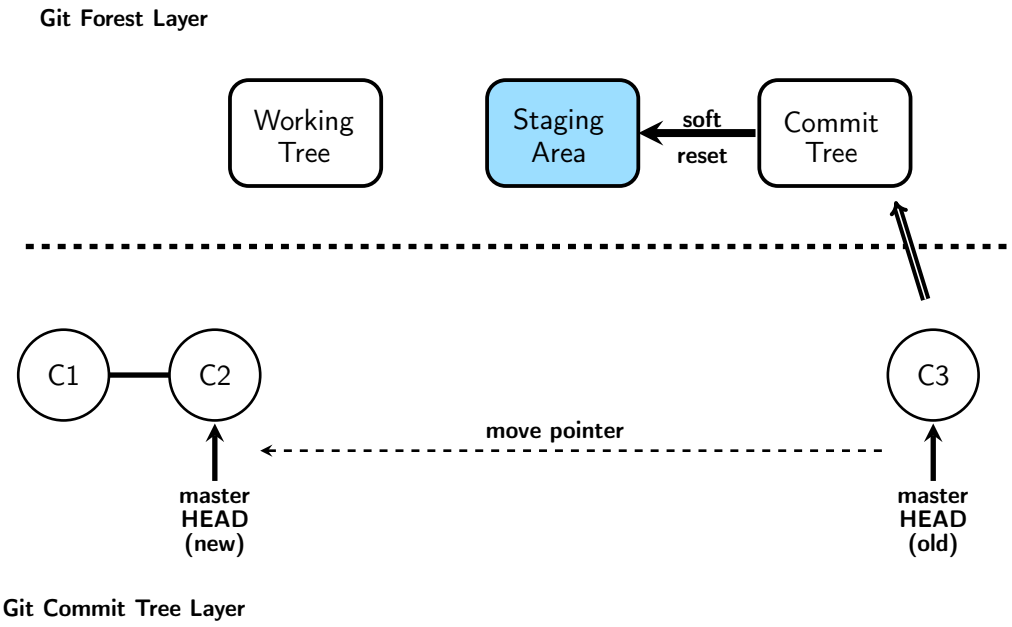

Fig. 2.2: Visualization of Git reset –soft

Note, Git reset does **not** delete any commits. In fact, Git actually never simply deletes any commits. That is not how Git works. Git reset xxx just cuts the connection in the commit tree.

As Fig. 2.2 implies, the commit C3 still exists and is accessible through Git's **reflog**. Handling the reflog is the topic of xxxyyy. At this point, we want to state, that Git never simply deletes commits and it is fairly save to try things out.

Git reset –mixed

Git reset mixed xxx is the second reset type. It is the same as Git reset soft plus it also un-stages the modifications. The mixed reset type is also the default version of Git reset, i.e.

```
$ git reset --mixed <SHA1>
```

and

```
$ git reset <SHA1>
```

is the same. To sum it up, Git reset mixed moves the HEAD/master pointer to the specified commit and leaves all modifications un-staged in the Working Tree. It is the opposite command to Git add xxx. Figure 2.3 shows the situation after a mixed reset from commit C3 to C2. Consider the following use case:

User Story: *"I want to completely reset my project. That is switching the HEAD/master pointer one (or more) commit(s) back and recreating the Working Tree stored in this commit. I DO want to keep the changes made so for inside the Working Tree"*

The command for this operation is the one right above.

However, the behavior is slightly different, if there are currently files staged in the index. Then, a mixed reset does not move the HEAD/master pointer and cut off the latest commit but simply un-stages all files, i.e. only undoes Git add xxx.

User Story: *"I want to un-stage all the files I recently put into the staging area via git add."*

To achieve this, just type

```
$ git reset
```

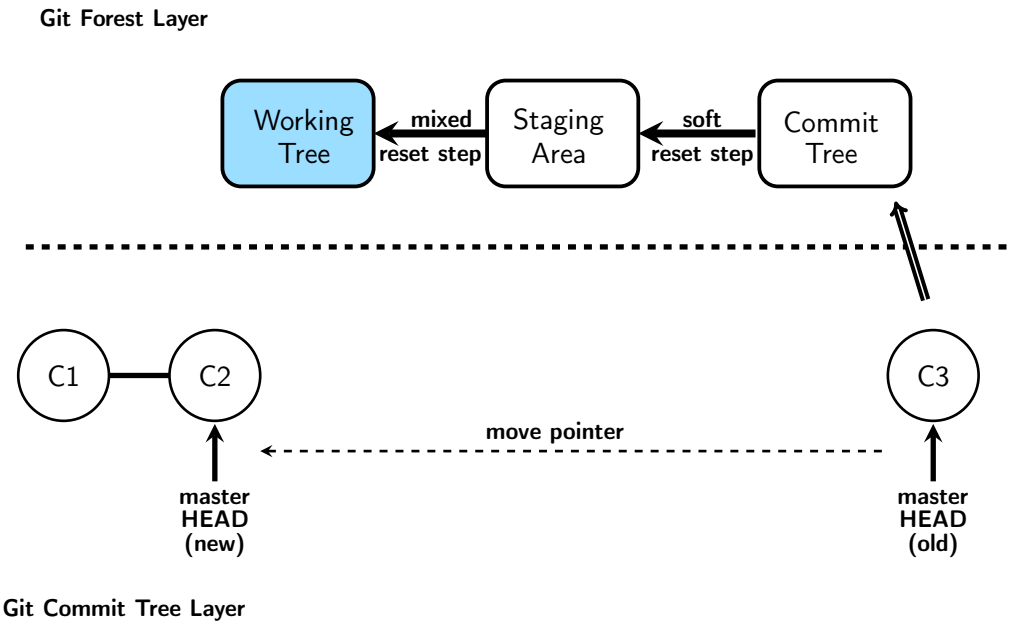

Fig. 2.3: Visualization of Git reset –mixed

Git reset –hard

The last reset type is Git reset hard xxx. Resetting the project hard means a soft reset plus a mixed reset plus removing all modifications even from the Working Tree. A hard reset recreates your project as it is saved in the specified commit. After a hard reset your Working Tree is clean, that is all modifications get removed. Please note, that this also implies, that all modifications which have not been saved in a commit are deleted. Therefore, before executing a hard reset, **ALWAYS** commit all your modifications made so far! Fig. 2.4 shows this situation.

User Story: *"I want to completely reset all the changes inside my Working Tree. That is switching the HEAD/master pointer one (or more) commit(s) back and recreating the Working Tree stored in this commit. I do NOT want to keep the changes made so for inside the Working Tree"*

```
$ git reset --hard <SHA1>
```

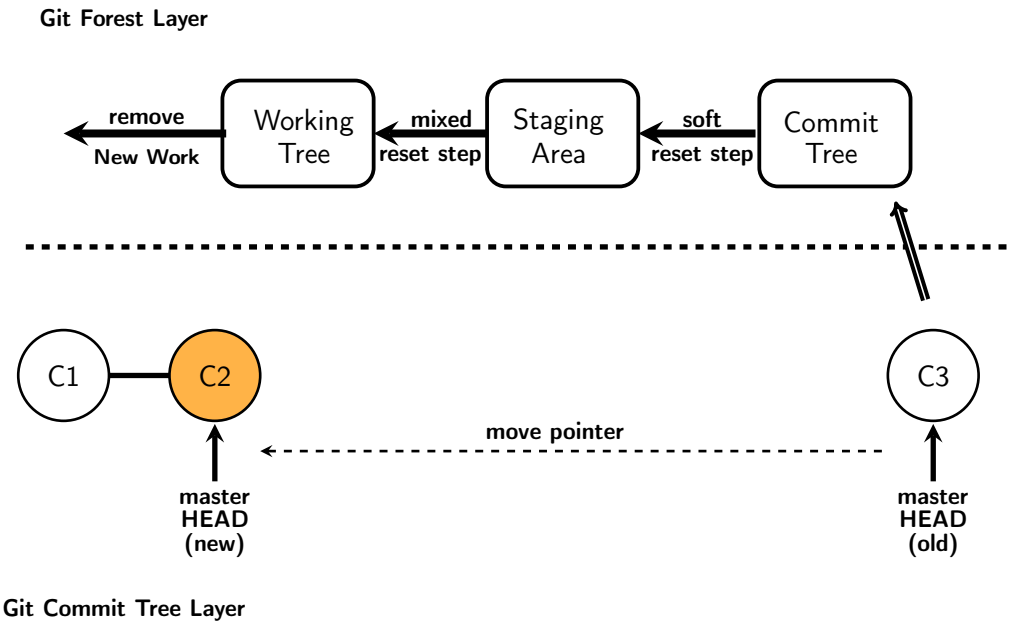

Fig. 2.4: Visualization of Git reset –hard

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 All about Branches

Section 2.1 shortly introduced the notion of branches and mentioned they are more or less special, named commits. We have now a closer look at them and show the basic Git branch handling.

Git basically offers four types of branches, **local non-tracking branches**, **local tracking branches**, **remote tracking branches** and real **remote branches**. The first three types only deal with your local git repository as the name implies. Only the last one really lives outside of your local system in a remote Git repository far, far away. To list all branches currently available, whether local or remote, type

```
$ git branch -a
```

The local git repository may be connected to several other remote system, see Section 1.3.1. You can view the list of all remote locations available via

```
$ git remote -v
```

In other words, these are the locations where you can pull from or push to. However, pulling or pushing from remote requires a little bit more then only the existence and knowledge from at least one remote git repository. Git xxx also needs to know if there are associated branches on the local and remote system, e.g. the local and remote master branch are connected with each other (see Section 1.3.1 till Section 1.3.3). Pulling and pushing only works if git knows the remote counter parts of your local branches. If there are no remote branches specified for one local branch, pulling and pushing is not possible.

3.1 Local Branches

A local branch is a branch that only you (the local user) can see. It exists only on your local machine. You can get a list of all local branches with

```
$ git branch
```

Local Non-Tracking Branches

All local branches which are not associated with any other branch are called **local non-tracking branches**. They are not meant to be published on some remote server and are not designed to

be used collaboratively. You use them for private development tasks on your local machine. It is not possible to do push or pull actions on those branches.

User Story: *"How do I crate a local non-tracking branch on my machine?"*

There are several possibilities using Git's branch, checkout or switch command:

```
$ git branch <branch_name>
$ git checkout -b <branch_name>  # also switches to the new branch
$ git switch -c <branch_name>    # also switches to the new branch
```

Local Tracking Branches

These branches are associated with another branch, i.e. they keep track of another branch, usually they track their remote counterpart.

```
$ git branch -vv
```

shows if a local branch has an associated remote branch. For example the master branch is a local tracking branch since it is mostly associated with it's remote version referred to as origin/master (or origin/main).
Locale tracking branches are meant to work collaboratively on them.

User Story: *"How can I start a local tracking branch based on a real remote branch?"*

```
$ git switch origin/<remote_tracking_branch_name>
```

They `origin` keyword is the name of the remote repository. If your remote repository has another name, you have to replace `origin` with that name in the above command.

Now, pull and push operation are possible on this branch.

Remote Tracking Branch

All remote tracking branches available are listed with

```
$ git branch -r
```

A local remote tracking branch gets automatically added to your local git repository when adding a local tracking branch. Think of your local remote-tracking branches as your local cache for what the real remote branches contain. It is an local image of the remote branch which get updated through git fetch and since git fetch is part of git pull you update this image which each pull request. Actually you do not work directly with this branch as long as you just use pull and push.

Read the section xxx about fetch and merge instead of pull to understand the purpose of this institution. Even though all the data for a remote-tracking branch is stored locally on your machine (like a cache), it's still never called a local branch. (At least, I wouldn't call it that!) It's just called a remote-tracking branch.

3.2 Remote Branches

A remote branch is a branch on a remote location (in most cases origin). You can view all the remote branches (that is, the branches on the remote machine), by running

```
$ git remote show origin/or_any_other_remote_location_available
```

It is not necessary to have a local counterpart of each remote branch on your local system. You can create any time a local remote tracking branch associated with an existing remote branch, see Section 3.1 - local tracking branches.

3.3 Overview

Fig. 3.1 shows the relations between these types of branches graphically.

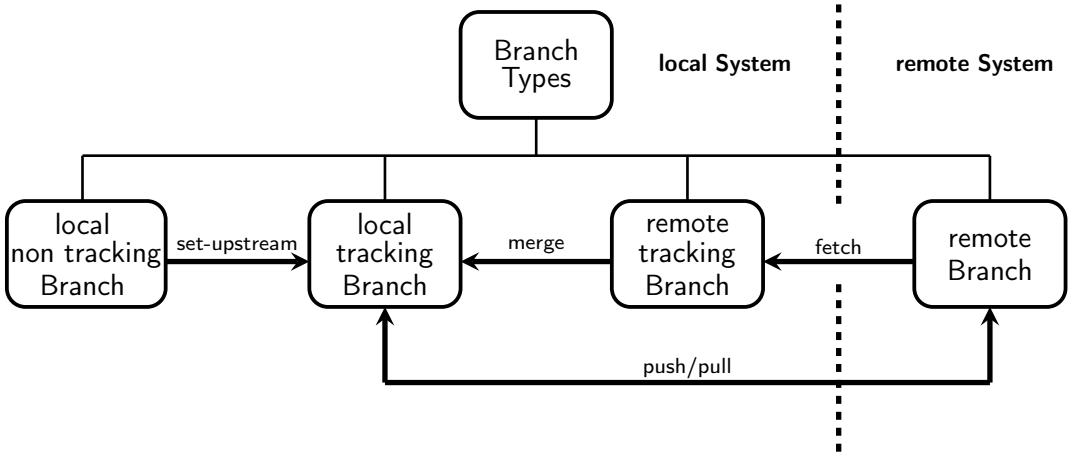

Fig. 3.1: Branch Types and their Relations

3.4 Advanced Dealing with Branches

3.4.1 Transforming a Local Non-Tracking Branch into a Local Tracking Branch

User Story: *"I have created a local (non-tracking) branch and now I want to publish this branch on the remote repository so that others can pull from and push to that branch."*

In other words, you want to transform a local non-tracking branch into a local tracking branch by creating a real remote branch version of your simply local branch. To achieve this, you just have to set an upstream location for your local branch exactly as you did it with your master branch when setting up a remote location, see Section 1.3 (note, `-u` is short for `--set-upstream`):

```
$ git remote -u origin <local_branch_name>
```

It is common practice to name the remote branch after your local branch.
This operation also adds a remote tracking branch locally to your Git repository. Local tracking branches and remote tracking branches always come together.

3.4.2 Merging Branches

The very basic meaning of a Git merge is to create a new commit which has two ancestors, the so called merge commit:

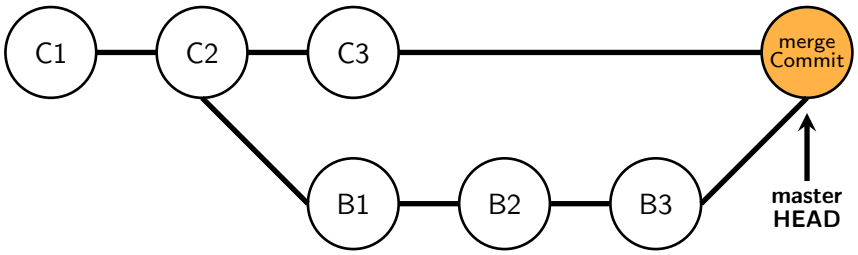

Fig. 3.2: Merge Commit

This orange merge commit gets created when executing Git's `merge` command.

Merging a local non-tracking Branch into master

User Story: *"I have finished work on a local non-tracking branch, local_branch, and now I want to merge local_branch into master branch."*

First of all, clean up your stages by committing every modifications on both branches. Do not continue unless everything is clean.

Merging is always done on the branch into which you want to merge the other branch. That is, for merging `local_branch` into master, you have to switch to the master branch. And there is another thing. Conflicts! If you are sure, that there will be no conflicts, e.g. the merge operation adds only files which do not exist on master, a common way to go is:

```
$ git switch master             # make sure you are on the master
                                # branch
$ git pull origin master        # update your master branch
$ git merge <local_branch>      # create the merge commit
$ git push origin master        # publish your merge operation
```

In case you expect conflicts or you are not sure, it is advisable to separate the creation of a new commit form the merge process:

```
$ git switch master    # make sure you are on the master branch
$ git pull             # update your master branch
$ git merge --no-ff --no-commit <local_branch_name> # postpone commit
  creation
```

The option –no-ff means no fast forward and –no-commit is self explanatory. Checking the Git status reveals if conflicts occurred during the merge process:

```
$ git status
```

You can call your favorite editor (I use VS Code, see Section 4.1 for how to change your editor) to eventually solve the conflicts with

```
$ git merge tool
```

and afterwards commit everything:

```
$ git commit -m"merge branch xxx into master"   # create commit
$ git push    # publish your merge operation
```

There is still one last issue a little bit unsatisfactory. Have a look at your local commit tree and compare it to the remote one as it is depicted in figure xxx after merging and publishing the merge commit:

hier das Bild mit den zwei verschiedenen commit Bäumen xxx

As we can clearly see, both trees differ a little bit. Especially for developers who only see the remote commit tree it is hard to understand what happens in the last commit. It would be nice if we could publish all commits under our local development branch, too (the B commits). Therefore, we take all commits of our development branch and put them at the end of the master branch. In other words, we create a new base commit for our local branch or we rebase all those commits, see figure xxx which illustrates this process.

hier das rebasing bild rein

Putting it all together finally leads to the following command sequence:

```
$ git checkout master
$ git pull
$ git checkout test
$ git pull
$ git rebase -i master
$ git checkout master
$ git merge --no-ff --no-commit <local_branch_name>
```

Again check the status with

```
$ git status
```

and solve the conflicts. Then create the merge commit and publish everything:

```
$ git commit -m"merge and rebade branch xxx into master"
$ git push
```

4 Miscellaneous

4.1 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.

4.2 Stashing

Git Stash-Motivation

With the stash option Git enables the creation of patches representing work still in progress. Lets say you are currently working on branch A (not the master) but you shall fix a bug in branch B immediately. You normally would stage and commit your changes in branch A before

switching to branch B. After committing and pushing the bug fix on branch B you return to branch A. Graphically, the situation is as follows (Fig. 4.1):

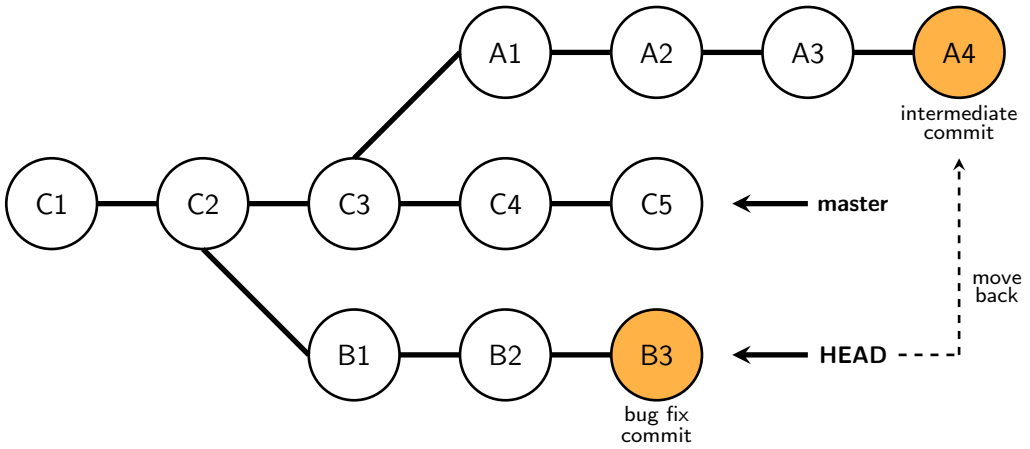

Fig. 4.1: Stash Situation

However, at the time you stooped working on branch A, work was still in progress and commit A4 represents an incomplete state. Therefore you decide to cut off commit A4 and put your modifications back into the Working Tree:

```
$ git reset --mixed HEAD~
```

Now you can finish coding and create a commit representing a stable state of development on branch A. This proceeding is perfectly fine but not the most intuitive way. A better approach would be to conserve the current state of branch A in a temporary, hidden commit which is not a direct part of the commit tree. Git offers the **stashing mechanism** for exactly that purpose.

Another motivation is that you did some hard work the whole and day and you want to save it in a commit like way. Perhaps, the next day, you continue further with a critical section and there could be the situation where you want to switch back to the state of today. Creating a full commit is not what you want, since your work is not really complete and unclear for others who might check out your development branch. Again, a Git stash is what you need.

Git Stash-Usage

Creating a stash is very easy. Just type

```
$ git stash
```

That's it, the stash is now listed in the stash reflog and you can display all stashes of the current branch with

```
$ git stash list
```

You can create as many stashes as you want in a row, the latest stash is referenced with zero, the next with one and so on. If the stash description automatically generated by Git does not fit your needs, the option save enables user defined descriptions

```
$ git stash save <your descriptive text>
```

Sometimes, it is helpful to also put the un-tracked or even the ignored files into a stash because they are relevant for the current development process but not for the whole project. Git offers the options `-u` or `--include-untracked` and `-a` or `--all`, respectively:

```
$ git stash -u
$ git stash -a
```

Furthermore, you can keep all staged files in the index after creating a stash with

```
$ git stash --keep-index
```

This option is also combinable with other options, e.g.:

```
$ git stash -u --keep-index
```

There are two possibilities to recreate the state of your branch stored in a stash. The first one is pop, the second one is apply. Using pop also removes the stash form the stashes reflog list whereas apply keeps the stash listed. Not specifying explicitly a stash reference number always recreates stash number zero:

```
$ git stash apply # recreate and keep stahs number 0
$ git stash pop # recreate and remove stahs number 0
```

Older stashes need the reference number given through the list command:

```
$ git stash pop/apply stash@{x}
```

Recreating a stash may lead to merge conflicts or which have to be solved using the merge tool:

```
$ git merge tool
```

4.3 Submodules

Briefly, you create a submodule in your main project when you add an additional library or some other file bundles which already represent a Git project, i.e. have their own `.git`-repository. Using submodules is most of the time straight forward. The official Git manual page is a good starting point to get the most important information, see [5].

At this point I just want to emphasize two things. The first one is, always initialize a newly added submodule and and especially do this recursively if the submodule contains other submoduels itself:

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

or

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

Omitting this step results in an empty submodule directory the next time your project gets cloned and the cloning operation delivers an error message, that it can not find one or more commits.

The second point is removing a submodule completely from your project which requires a few steps to go.

User Story: *"I want to remove the submodule library called "lib" completely from my project"*

```
$ git submodule deinit lib # deinitialize the submodule
$ git rm --cached demolibapp -r # remove from git
$ rm -rf .git/modules/lib  # delete the git-modules entry
$ git add --all
$ git commit -m "removing extra submodules"
$ git push
$ rm -rf lib # remove everything from the filesystem
```

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.

[3] Unknown. *Git Kraken*. Ed. by Unknown. Dec. 20, 2021. URL: https://www.gitkraken.com/.

[4] Unknown. *Git Tools - Reset Demystified*. Ed. by Unknown. Dec. 4, 2021. URL: https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified.

[5] Unknown. *Git-Tools-Submodule*. Ed. by Unknown. Dec. 19, 2021. URL: https://git-scm.com/book/de/v2/Git-Tools-Submodule.

[6] Unknown. *Tortoise Git*. Ed. by Unknown. Dec. 20, 2021. URL: https://tortoisegit.org/.

[7] John Wiegley. "Git from the bottom up". In: (Dec. 26, 2017).