A quick look behind the scenes

In this first chapter, I'd like to start with addressing probably one of the biggest misconceptions of git as a version control system - its complexity. Whether you've just started out, or have been using it for quite a while, maybe even years. If we don't explore what the fundamental building blocks of git are, it'll probably always be a bit of a black box and mystery that "kind of works" but feels rather complex and hard to understand.

Turns out this can be easily fixed. As we explore some of git's tools throughout this book, we'll quickly realise that git is actually just an addressable file system and that everything we do can be broken down to simple operations. No black magic, no things that couldn't be done otherwise by hand (not recommended though).

Once we have a better understanding of the inner workings, we can be much more confident and productive in our actions when using git on a daily basis. Moments where we believe we've "messed up" all of a sudden become scenarios that are extremely easy to solve.

What's in a repository

Let's start off by taking a look at what makes a git repository a git repository. Too keep things simple, we'll create a new directory:

```
$ mkdir some-repository
$ cd some-repository
```

Once this is done we initialize a new repository using `git init`:

```
$ git init
Initialized empty Git repository in \
/Users/pascalprecht/some-repository/.git/
```

At this point, our `some-repository` directory has a git repository that includes everything it needs to function. It's also worth noting, in case you aren't too familiar with the `git` command line tool but prefer using some IDE for development, when you create a repository through your IDE, at the end of the day it will also just run `git init`. There's nothing special going on here.

You'll notice that there's now a `.git` directory which, surprise… is the actual repository. Any action that is related to tracking changes in git, will be recorded in there. Despite it being actually rather hard to lose work when using git, if you delete the `.git` directory, your project's version history will be gone forever unless there's a back up. Luckily, it's very uncommon for users to accidentally do that. One of the reasons might be that, at least on Unix systems, every file or directory starting with a `.` is considered a hidden file and therefore not immediately visible to most users.

Now is a very good time to look inside that `.git` directory and see what's happening there. Feel free to either list its contents with a tool of your choice, or use your operating system's file explorer. I personally prefer the `tree` command which can be installed on Linux and macOS using various package managers.

```
.git
  HEAD
  branches
  config
  description
  hooks
      applypatch-msg.sample
      commit-msg.sample
      post-update.sample
      pre-applypatch.sample
      pre-commit.sample
      pre-push.sample
      pre-rebase.sample
      pre-receive.sample
      prepare-commit-msg.sample
      update.sample
  info
      exclude
  objects
      info
      pack
  refs
       heads
       tags

9 directories, 14 files
```

This looks a little bit more overwhelming than it really is. Obviously we can see a bunch of directories and files. Let's inspect a few of them:

- `HEAD` - Probably one of the most important files in a git repository. `HEAD` tells git what the currently checked out branch, tag or commit is. If we inspect its file contents, we'll should see the following:

  ```
  ref: refs/heads/master
  ```

  This means that `HEAD` is pointing to a reference which is located in `refs/heads/master` inside the `.git` repository. Don't worry if that doesn't make too much sense right now. We'll play around with this in a minute.

- `config` - Just like the name says, this is where git related configurations are stored. Most of the time we configure git machine-wide, meaning that there's a `.gitconfig` file in the user's home directory and all configuration

values there apply everywhere, unless overwritten by a project specific configuration.

- `objects` - This is where our data goes every time we add or commit changes to the repository. We'll see in a minute what that looks like in action.
- `refs` - References like branches, remote branches and tags are stored here. As mentioned earlier, our test repository's `HEAD` points to a reference `refs/heads/master`. You might have noticed that there is no such file. That's because there aren't any commits in the repository yet, but let's not get ahead of ourselves.

Alright, now that we have a little bit of a better picture what makes a repository a repository, let's quickly talk about some concepts of how they work, so that shortly after that we can play around with this and discover some interesting things.

Working directory, index, repository

If you feel comfortable with how git repostories generally work, meaning that they have a working directory, an index (or "stage") and the repository itself, then you can safely skip this section and move on to the next part. If not, you might want to stick around as these fundamentals are crucial to make sense of most of the topics discussed in this book. I'll keep it short and to the point.

When working with git repositories, think of your data going through three different stages, as illustrated below.

These stages are:

- **Working directory** - This is basically the root directory of your poject including the entire file tree of the currentlly checked out commit. If you check out a different commit, branch or tag, your working tree will change along with it accordingly. It's important to realise that any additions, changes and deletions of files and directories done here that aren't tracked, won't make it into the repository.
- **Index** - The index, also known as "stage" is where you decide what changes should go into the next commit. It also keeps track of the changes between what is stored in the repository (or at least the currently checked out commit) and the working directory. Every change that should be committed has to go through the index at some point. This is a very important characteristic of a git repository as it enables powerful features and workflows. We'll come back to those later.
- **Repository** - As we've learned, this is where the actual data goes and all commits, branches, tags and other things are stored.

When we work in a repository, we always go through the same process:

1. Make changes (working directory)

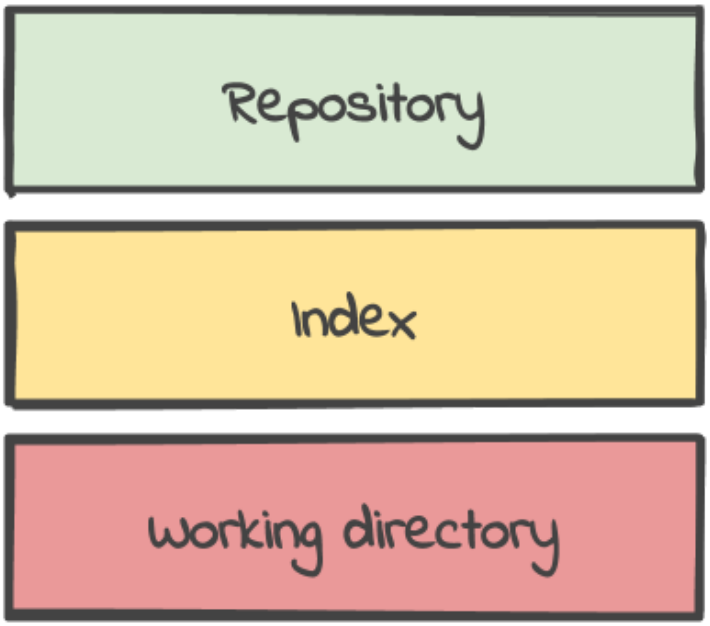

Figure 1: Three stages of a git repository

2. Add changes (index)
3. Commit changes (repository)

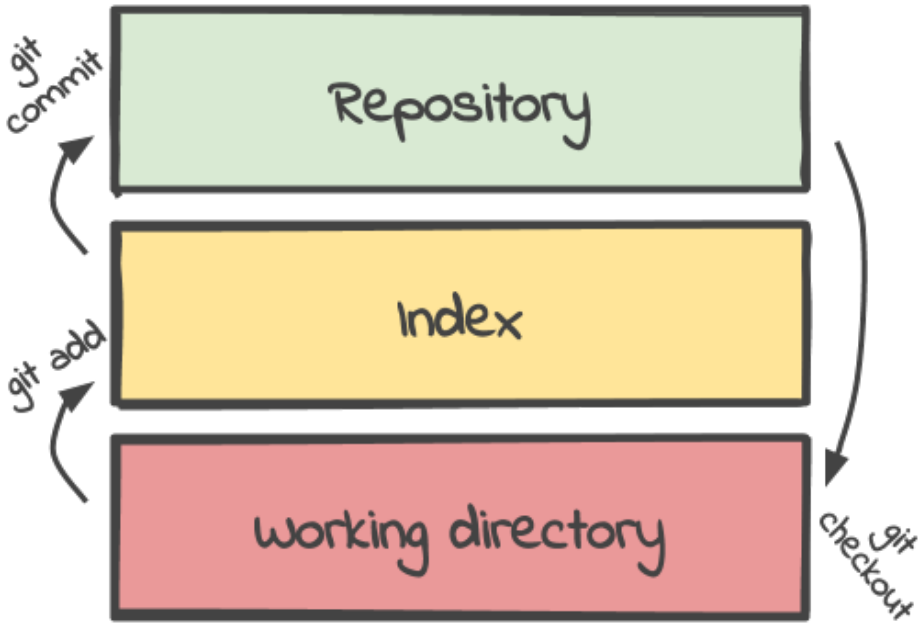

Figure 2: Typical git workflow

There are surely cases where we checkout different branches, or maybe reset to different commits, but from that point on, we keep repeating this process. Notice how there's no word about actually removing changes or commits. This is because, even if we want to remove something, we still technically "move forward" and add changes and commits.

It's hard to lose work in git and in the next section we discuss one of reasons why.

Where our data ends up

One thing that a lot of git users aren't aware of, is that even when changes are added to the index, but not actually committed yet, git will already store the corresponding data in the repository.

There's various technical reasons why this is done. In fact, git wouldn't be able to provide certain features and guarantees if it didn't do it this way. The reason I'm bringing this up is not to bore you with implementation details that aren't

necessary for you to know, but rather to give you better insights of what git is doing and, to emphasize the fact that it's really hard to lose data in git, once it has been tracked, even if it was only added via `git add`. Hopefully git is already starting to get less scary!

We can try this out by creating some file with some arbitrary content:

```
$ echo "Hello World" >> hello-world
```

Obviously, running `git status` will tell us that there's an untracked file in the repository:

```
On branch master

No commits yet

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

        hello-world

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

Next we add the `hello-world` file to the repository's index using `git add` and confirm it by doing another `git status`:

```
$ git add hello-world
$ git status
On branch master

No commits yet

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

        new file:   hello-world
```

Alright, stop right there. Take another look inside the contents of the `.git` directory, specifically in `objects`:

```
  objects
      80
          2992c4220de19a90767f3000a79a31b98d0df7
      info
      pack
```

Remember that this was empty before? Git has created a directory structure composed of a hash that represents the `hello-world` file and its contents. The

hash might look different on your machine if you've created a file with a different name and content. Git has also created the index in `.git/index` which, as mentioned before, tracks changes between the repository and our working directory.

Let's go ahead and do what we'd usually do - commit our changes to the repository:

```
$ git commit -m "Adds hello-world"
[master (root-commit) a39ee7c] adds hello-world
 1 file changed, 1 insertion(+)
 create mode 100644 hello-world
```

The contents of our `.git` directory, again, look different now:

```
.git
  COMMIT_EDITMSG
  HEAD
  branches
  config
  description
  hooks
     applypatch-msg.sample
     commit-msg.sample
     post-update.sample
     pre-applypatch.sample
     pre-commit.sample
     pre-push.sample
     pre-rebase.sample
     pre-receive.sample
     prepare-commit-msg.sample
     update.sample
  index
  info
     exclude
  logs
     HEAD
     refs
         heads
             master
  objects
     80
        2992c4220de19a90767f3000a79a31b98d0df7
     a3
        9ee7ccc01b294670a6c11212184d935c970764
     ce
        bd3f79ac0181ba01d3c8f34a7b859085e3a783
     info
```

```
    pack
refs
    heads
        master
    tags

15 directories, 22 files
```

Most notably, there are now three different directories composed of hashes `80299...`, `a39ee...`, and `cebd3f...`. At least one of them has a different hash on your machine, even if you've created a file with the same name and the same contents. There are probably a couple of questions coming up by now. What are those hashes? Where do they come from and why are there exactly three of those hashed directories? No worries, all of that will make more sense when we explore the **Anatomy of a git commit** in the following chapters.

Another thing that stands out is that, now that we've created our first commit, there's a new file `refs/heads/master`, which, if you remember, didn't exist before. Reading the contents of that file will give us one of the hashes in `objects`:

```
a39ee7ccc01b294670a6c11212184d935c970764
```

What this means and why this file exists in the first place, will be discussed deeply in **A branch is just a pointer**.

What we've learned

- Throughout this chapter we've explored the inner workings of a git repository at least to a degree that we know all of our data, changes and commits are somehow stored as hashes on the file system.
- We've also learned that, even if we don't create commits but simply add our changes to the index, git will already store those changes in the same way, making it rather hard to actually lose work.

With this foundation, we can now take a closer look at the fundamental building blocks of a git commit and how we end up with those hashes in the first place. This will give us an even better understanding to pave the way towards branches and rebasing in general.

Buy the complete book

This was an excerpt of the REBASE book. A complete version can be purchased at rebase-book.com.