Practical Versionning with Git

Guille Polito, Stéphane Ducasse

July 10, 2019

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Getting Started with Git

In this chapter we introduce the basics of Git and VCSs through guided examples. We first start by setting up a repository in a remote server and then load it in our own machine. We then show how we can inspect the state of our repository and save our files into it. Once our changes are saved, we show how we can push our changes to our remote repository in a distant server.

This chapter will assume you have Git already installed in your machine, and that you're using a *nix operating system. For users of other systems such as Windows, an appendix at the end will give some details on different setups and installation procedures. Moreover, you will see that we will approach Git with the *command-line*. Don't be affraid if you've never used it before, it is not as difficult as it may seem and you will get used to it. There is always a first time! Also, we promise you that everything you learn in here can be applied to, and will actually help you better understand, non command-line tools.

1.1 Creating a Repository

A Git repository is a store of files and directories. The big difference between a Git repository and a simple filesystem is that the changes we make are stored as events, like a timeline (Figure 1-1). Git not only stores such a timeline but also allows us to query it, undo some of its changes, and so on.

Before working with such a history/repository we need to set it up. Fortunately, setting up a Git repository is much lightweight than setting up a database repository. Though there are many different ways to create a Git repository, we will start with a simple solution to be up and running as fast as possible. We will study other ways to setup repositories in Chapter **??**.

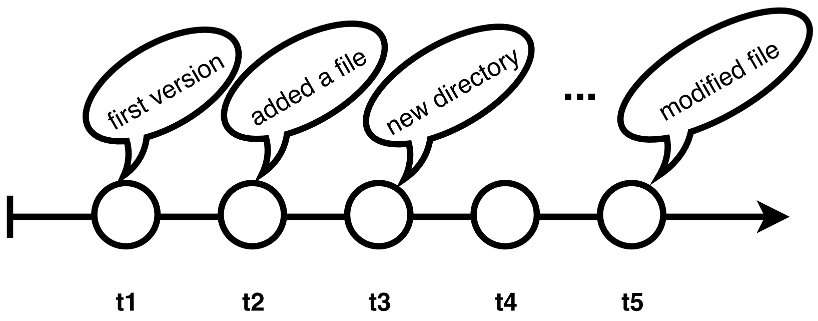


Figure 1-1 A Repository as a timeline of changes

Let's proceed to create a repository in an online hosting service such as GitHub or GitLab. I'll give you some minutes to create an account and log into it. Then you're ready to use the *New Repository* action, within the menu using the + symbol. Figure 1-2 shows the kind of form GitHub provides to its users to create a new repository.

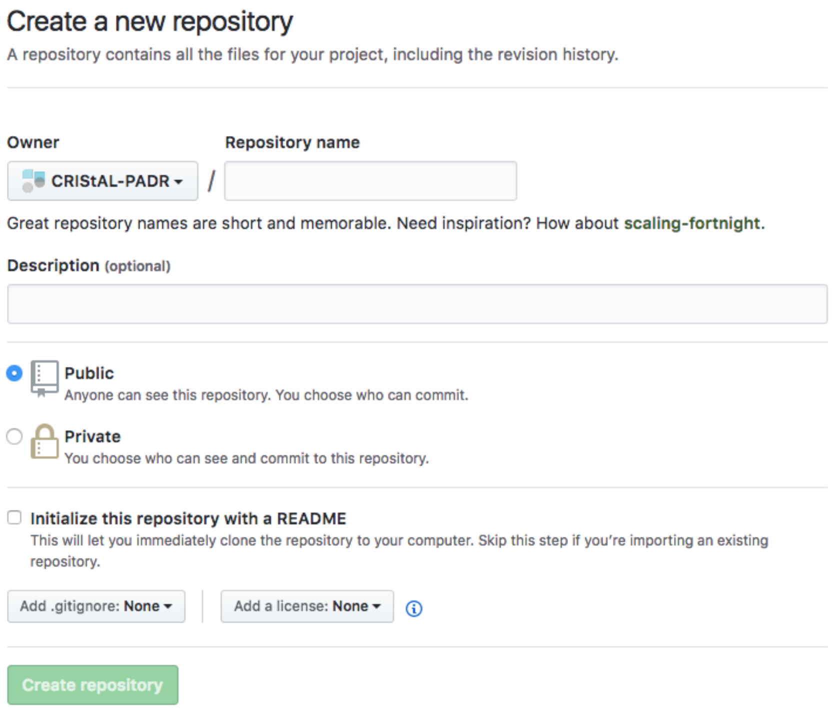
Following the form/wizard will eventually get you a running repository online. You will be most probably then redirected to your repository page. Figure 1-3 shows how such a page looks in GitHub.

We are almost set to work from the command line now. However, we need to set-up our mind around a couple of extra concepts. The repository we just created does not exist in our machine. Actually, it is stored in some server maintained by github/gitlab. To interact with this repository we will need a network connection. We will call this repository, living in a remote server, a **remote repository**.

1.2 `git clone`

Git, constrastingly to other VCSs, is a distributed VCS. This has a lot of consequences in the way we work, that we will study in detail in Chapter **??**. For now, you will have to remember only one thing: instead of being connected all the time to our remote repository, we will copy the repository we just created in our machine. Then we can work against our copy in the repository. Eventually, we will synchronize the state between our local repository with the remote one. This is what makes possible the disconnected or offline workflow that people often praise in Git.

Making a local copy of a remote repository is such a common task that git has a dedicated command for it, the `git clone [url]` command. The `git clone` command receives as argument the url of our repository, that we can


Create a new repository

A repository contains all the files for your project, including the revision history.

Owner: CRIStAL-PADR / Repository name:

Great repository names are short and memorable. Need inspiration? How about scaling-fortnight.

Description (optional)

Public [✓]
Anyone can see this repository. You choose who can commit.

Private
You choose who can see and commit to this repository.

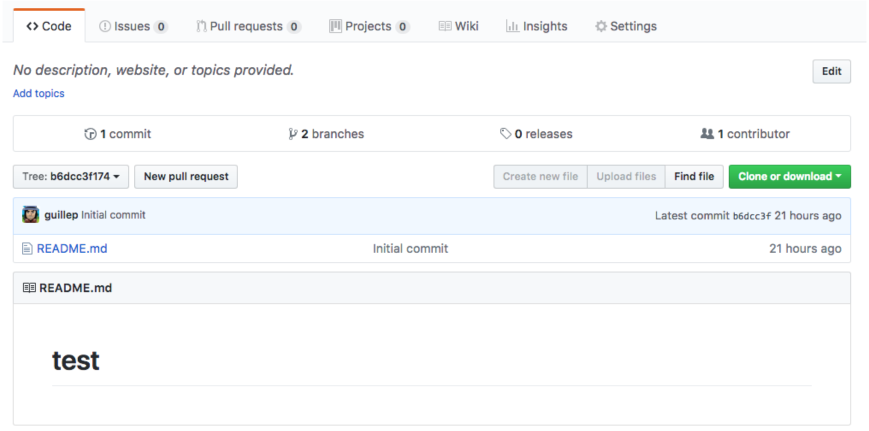
Initialize this repository with a README
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None

Add a license: None

Create repository

Figure 1-2 Creating a New Repository in Github



No description, website, or topics provided.

1 commit | 2 branches | 0 releases | 1 contributor

guillep Initial commit — Latest commit b6dcc3f 21 hours ago

README.md — Initial commit — 21 hours ago

README.md

test

Figure 1-3 A Repository Page for a project called **test** in GitHub

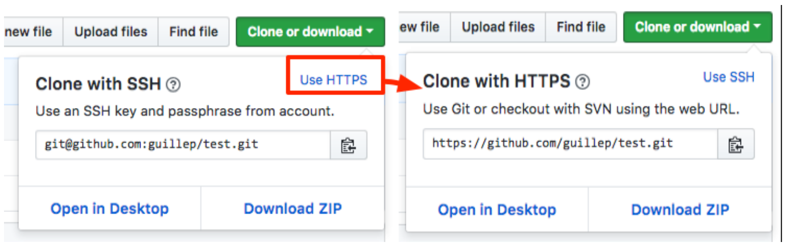

new file
Upload files
Find file
Clone or download
Clone with SSH
Use HTTPS
Use an SSH key and passphrase from account.
git@github.com:guillep/test.git
Open in Desktop
Download ZIP
Clone with HTTPS
Use SSH
Use Git or checkout with SVN using the web URL.
https://github.com/guillep/test.git

Figure 1-4 Getting the HTTPS url of your repository from GitHub

get from our repository page. You will see that your repository page will offer you different url options, the most used being SSH and HTTPS urls. We will use in this chapter HTTPS urls because they have an easier setup, but for those readers that are curious, Section **??** compares SSH and HTTPS, and Section **??** shows how to setup your SSH environment.

To obtain the HTTPS url of your repository, go to your repository's page and look for it under the clone/HTTPS options. As an example, Figure 1-4 illustrates how to get such url from a GitHub project page.

Copy that url and use type your command as in:

```
$ git clone [url]
Cloning into '[your_project_name]'...
remote: Counting objects: 11082, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 11082 (delta 2), reused 6 (delta 2), pack-reused 11076
Receiving objects: 100% (11082/11082), 4.35 MiB | 1.22 MiB/s, done.
Resolving deltas: 100% (4063/4063), done.
Checking connectivity... done.
```

When the command finishes, Git is done creating a directory named as your repository (your_repo_name). We will call this directory the **working directory** as it is where we will work and interact with our repository. The working directory, with all the files and directories it contains, is managed git and linked to our repository. Do not worry, there is nothing else you need to do to keep this link, Git will automatically track your changes for you.

You are now ready to go and start working on your project.

1.3 Making Changes: How does Git track my changes?

Let's now dive in our working directory and start making changes to it. I'll for example create a file called `project.txt`, then open it with a text editor and add some lines to it.

```
$ cd your_project_name
$ touch project.txt
...
```

After some time working, I can use the `ls` command to check the files in my directory.

```
$ ls
project.txt README.md
```

And then the `cat` command to check their contents from the command line.

```
$ cat project.txt
# Done
 - created the repository
 - git clone
 - created this file

# To do
 - commit this file
 - push it to my remote repository
```

```
$ cat README.md
#My Project

This project is an example Git repository used to learn Git.
Check the project.txt file for information about pending tasks.
```

Basically, we have modified some files, but we have done no Git at all. What does Git know about these files at this point?

Git does nothing without your permission

A new important thing to grasp about Git at this point is that it will do nothing until we explicitly ask it to do it. In this sense, Git is not any kind of repository but a transactional repository. All changes we do in our working directory are not stored by git automatically. Instead, we need to explicitly store them using a `commit` command, as we do with transactional database to store any data.

The transactional aspect means also that:

- while we do not commit, we can easily rollback our changes
- the other side of the coin, until we commit all our changes are in a transient state, and we may lose them

We will see more of this *explicitness* applied in other cases throghover these chapters. Sometimes it may seem that Git is just dumb, and that it cannot guess what we want to do when it is obvious. However, the case is that Git has so many possibilities that not guessing is the healthier decision in most cases. Specially when considering destructive operations that may make you lose hours of work.

`git status`

We can then turn our question around: Does Git know something about these files? Git indeed tracks our files to know what should be saved and what should not. We can use this information to see what are actually the changes that happened while we were working. The `git status` command produces a list of the current changes.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
    directory)

  modified:   README.md

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

  project.txt

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

Reading the output of `git status` we can see that it lists several things:

- **Changes not staged for commit.** Lists the modified files that Git already knows.
- **Untracked Files.** Lists the files that are in the working directory but were never added under the control of Git.

Also, `git status` shows some hints about possible commands that we may use next such as `git add` or `git checkout`.

Git and directories

You may have observed a curious behaviour with directories. If you create an empty directory and then use the `git status` command, you'll notice the directory is not listed at all. It even looks like the directory is being completely ignored. For example, if we try adding an empty directory into a new repository Git will actually say *working tree clean* as if there were no changes at all.

```
$ mkdir emptyDirectory
$ git status
On branch master
nothing to commit, working tree clean
```

Indeed, Git does not manage empty directories but only files. Directories are only modelled as paths to get to files. No extra information is tracked for them. In other words, we cannot just store directories into Git. And in case we want to do it for some reason, we need to put files into them.

1.4 Commiting your changes

We would like now to save our changes in our Git repository. This way, if anything happens, we can always recover our work up to this point. We have said before that the operation of saving our work in the repository is called a **commit**. If we try the `git commit` command we will see this is not as direct as expected.

```
$git commit
On branch master

Initial commit

Untracked files:
  README.md
  project.txt

nothing added to commit but untracked files present
```

If we read Git's message, we will notice that though it has correctly we have new files, he is asking us to *add* them to commit.

The Groceries Metaphore

To make it simple, you can see this whole tracking story as going to the supermarket. Imagine you make your grocery list and go to the supermarket. To get our groceries and take them home, we need first to go look for them, put them in our shopping cart, and then go and pay for them. An extra services may propose to take your grocery list and do the groceries for you. But such extra service may be charged extra and takes out some control from you (and can –and will!– make mistakes!).

Same same, Git committing all changes is not its default behavior. Partly because Git's philosophy is to ask the user explicitly what to do, which in this case is translated to asking what to commit. Instead, Git requires us to add the files we want to commit to a list of *added* files, also called in Git terminology the *staging area*, and equivalent to your shopping cart. Once our staging area is full with the changes we want to commit, we need to pay for those changes using the `git commit` command.

A First Commit

Adding changes to your staging area is done through the `git add [file]` command. Let's proceed to add our changes and see what is the status of our repository afterwards.

```
$ git add README.md
$ git add project.txt
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

  modified:   README.md
  new file:   project.txt
```

Now Git says that our two new files are listed as ”to be committed”. Let’s now proceed to save our changes in the repository with the `git commit -m "[message]"` command. The *message* used as argument of this command is a piece of text that we can use to explain the contents of the changes, or the intention of our changes.

```
$ git commit -m "first version"
[master a93c016] first version
 2 files changed, 12 insertions(+), 1 deletion(-)
 create mode 100644 project.txt
```

If we see the status of our repository after the commit we will also see that it has changed. There is nothing to commit:

```
$ git status
On branch master
nothing to commit, working directory clean
```

Add then Commit, all over again

If we repeat the process and we change one of our existing files, we will see something interesting. Commiting our changes in a file we added before requires that we do a `git add [file]` and `git commit` again on the same file, even if Git already knew about it.

```
$ cat project.txt
# Done
 - created the repository
 - git clone
 - created this file
 - commit this file

# To do
 - push it to my remote repository
```

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
    directory)

  modified:   project.txt

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

This is because even if in the surface Git seems to manage files, it actually manages *changes* in those files. Technically speaking, the changes we have done are *new* changes, so we have to tell Git we are interested in those changes.

```
$ git add project.txt
$ git commit -m "Commit is not in ToDo anymore"
[master e14a09f] Commit is not in ToDo anymore
 1 file changed, 1 insertion(+), 1 deletion(-)
```

1.5 Synchronizing with your Remote Repository

So far we have worked only on the local repository residing in our machine. This means that mostly all of Git features are available without requiring any internet connection, making it suitable for working off-line (think on working on the train or with a constrained connection!). However, working off-line is a two-edged weapon: all your changes are also captive in your machine. While your changes are in your machine, nobody else can contribute or collaborate to them. Moreover, losing your machine would mean losing all your changes too. Keeping your changes safe means to synchronize them from time to time with your remote repository.

Git's metaphore for remote synchronization is based on the ideas of *pulling* and *pushing* changes between repositories. Git makes us situate ourselves in our local repository. We bring other's changes by pulling them from remote repositories to our local repository. We send our changes by pushing them from our local repositories to one or many remote repositories.

Getting Remote Changes with `git pull`

Before being able to share our commits in some external server, we need before to update our repository to avoid them being de-synchronized. While you can always try to share your commits by directly pushing (see Section 1.5), you will see with experience that Git favors pulling before pushing. This is, among others, because in your local repository you have complete control to do whatever manipulation you want, what is especially important to solve mistakes and merge conflicts. You cannot do the same in your remote repository.

In our example pulling does not seem really necessary because you are the only person modifying your repository. No new changes happened in the remote repository in the meantime. However, let's imagine that you have done a modification in this same repository from another machine or even a different clone in the same machine (which are totally feasible scenarii). In that case, you would like to update your local repository with those new changes.

Updating our repository is done through the `git pull` command. Pulling will update our database and then update our files.

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1), pack-reused 0
Unpacking objects: 100% (2/2), done.
From https://github.com/guillep/test
   1656797..a2dbd8b  master     -> origin/master
Updating 1656797..a2dbd8b
Fast-forward
 newfile | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newfile
```

A Bit on Merging

We will see in detail in Section **??** that `git pull` performs two different operations: a fetch and a merge. The fetch lookups new commits in remote repositories. The merge, studied in detail in Section **??**, takes the state in the remote repository and your local repository and tries to make a single version of of that. Three different scenarios can actually happen from a pull operation, which will be:

- **Fast-forward**: the updates were applied without needing a merge.
- **Automatic Merge**: the updates were applied without conflicts. Git had to do a merge commit and will ask you for a commit message.
- **Merge Conflict**: The changes you did and incoming changes affect some common files. In this case Git does not know what version to keep (or even if a mixture is possible) and asks you to solve it manually before doing a new commit.

Once the merge is resolved, your working copy is updated with the new version of your repository. Luckily for us, fast-forward and automatic merges are the simplest and more common ones. They require almost no manual interaction other than introducing a message.

Sending your Changes with `git push`

The final step in our Git journey is to share our changes to the world. Such sharing is done by **push**ing commits to a remote repository, as shown in

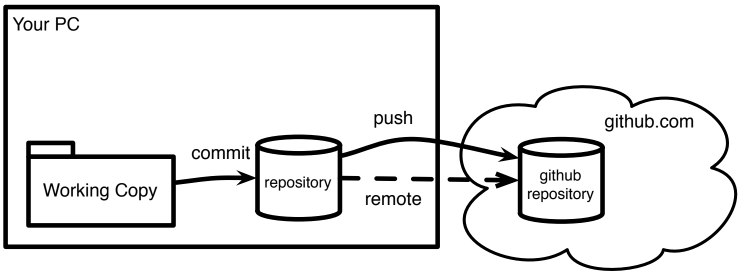


Figure 1-5 Push is an operation that sends commits from your local repository to a remote repository.

Figure 1-5. To push, you need to use the git command `git push [remote] [remote_branch]`. This command will send the commits pointed from your your current branch to the remote [remote] in the branch [remote_branch].

```
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> temp
```

A Branch's Upstream

We can omit the destination branch and remote from the command, relying on Git default values. By default a `git push` operation will push to the so called **branch's upstream**. A branch's upstream is a configuration specifying a pair (remote, branch) where we should push by default that branch. When we clone a repository, the default branch comes with an already configured upstream. We can interrogate Git for the branch's upstream with the super verbose flag in the branch command, *i.e.*, `git branch -vv`, where we can see for example that our **master** branch's upstream is **origin/master**, while our **development** branch has no upstream.

```
$ git branch -vv   # doubly verbose!
  development   1656797 This commit adds a new feature
  master        f269295 [origin/master] First commit
```

When a branch has no upstream, a push operation will by default fail with a Git error. Git will ask us to set an upstream, or otherwise specify explicitly a pair remote/branch for each push.

```
$ git push
fatal: The current branch test has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin test

$ git push --set-upstream origin test
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@github.com:[your_username]/[your_repo_name].git
   b6dcc3f..f269295  master -> test
```

Pushes can get Rejected

In some scenarii Git may reject our pushes, so they are not saved to the remote repository. In general Git rejectes changes when the remote repository has diverged from ours. Of course a rejection may also happen when we don't have write permissions in the remote repository. The typical error shows something like the following:

```
$ git push
To git@github.com:guillep/test.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to
    'git@github.com:[your_username]/[your_repo_name].git'
hint: Updates were rejected because the remote contains work that
    you do
hint: not have locally. This is usually caused by another repository
    pushing
hint: to the same ref. You may want to first integrate the remote
    changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
    details.
```

As the error message says, the remote has changes that we do not have locally. In other words, our push has been rejected because otherwise we would have overwritten the remote changes. Instead, we need to take the remote changes and *mix and match* them with our changes, by applying a pull (Section **??**) and a merge (**??**). After the pull, our repository will have our outgoing changes, but no more incoming changes, and so our push will not be rejected

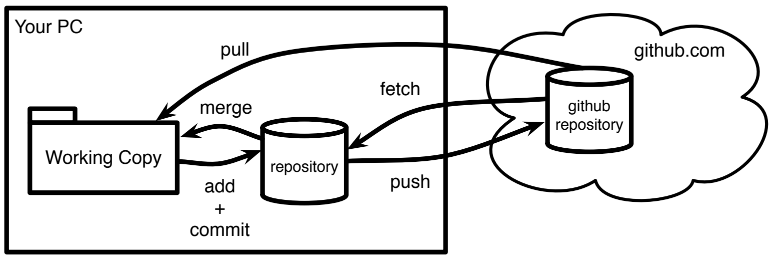


Figure 1-6 Overview of Git basic operations: `add`, `commit`, `pull` and `push` (+ extra `fetch` and `merge`).

1.6 Overview

In this chapter we have studied the basic operations of Git. We have seen that a new repository starts in your local machine with a `git clone` that creates a working copy directory. Changes in our working copy are tracked by Git automatically and we can query the tracking using `git status`. We can then proceed to operate on our changes as illustrated in 1-6.

We have seen that:

- **`git add`** tells Git about files to track for commit
- **`git commit`** finishes a transaction and stores our changes in a commit
- **`git pull`** synchronizes a remote repository with our local repository by doing first a `fetch` and then a `merge`. Different merge scenarios may happen, in which some of them cause conflicts that have to be manually resolved.
- **`git push`** sends our changes to a remote repository. A push can get rejected

1.7 Exercises

1. **Exercise 1.** Create an account in your preferred git repository hosting service, create there a repository and clone it. Then, check its history from the command line. How many commits are there in the repository? Tip: there is a difference if you checked the ”create README.md file” checkbox while creating a repository.

1. **Exercise 2.** Create a file, a file inside a directory and an empty directory. Commit them (remember, `git add`, `git commit`). What can you see there? How does Git manage directories?

1. **Exercise 3.** If you're on a unix system (linux/osx), try changing your file's permissions and check `git status`. How does git treat file per-

missions? Commit your changes and check the log. What can you observe?

1. **Exercise 4**. Push now your changes to your remote repository. Then, clone your repository again in another directory. Tip: try checking the help of the clone command `git clone -h`. Did git save all your files, directories and even permissions?

1. **Exercise 5**. Go back to your first repository, add a new file, commit it and push it. Then go back to the second repository and pull. Inspect the history in both repositories: Is it the same?

1. **Exercise 6**. Check your online repository on your hosting service. Can you see the same state as in your local repositories? Go over the different tools offered by the hosting, they usually give some idea of the activity of the project, try to understand what they are for.