ComputerTools

Release 0.1

Pierre Marchand

Sep 11, 2023

The purpose of this document is to present useful tools for people who are just learning to code and for people who are more familiar with programming but want to improve their workflow.

How to access this document

This document is written in Sphinx and can be accessed as a website or as a pdf document.

Contact

If you have any suggestions or questions, please contact me via my personal webpage.

Contribute

If you have a GitHub account, you can directly suggest changes or correct typos via a pull request, either by clicking on the edit button on the top right of each page, or directly in the GitHub repository containing the source of this website. If you are not familiar with pull request, you can also open an issue.

Contributors

Thank you to the people who helped to improve this document!

- zmoitier

  1 contribution

- marcellabonazzoli

  1 contribution

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Disclaimers

Before reading the content of this document, you should be aware that

- I have studied very little computer science and informatics in my undergraduate studies. Most of what I know now comes from self-learning, and practice. If you disagree, or if you think there is a better way than what I will present, please share your ideas and *contribute* to this document!
- The target audience for this document is people who have also studied very little computer science and informatics, but still need to code.
- *A workflow is personal*, meaning that what I present may not be entirely adequate to your own way of reasoning. But I hope it can inspire you to think about your workflow, and that it can give you ideas on how to make it more efficient.
- The tools I will present are maintained and well-documented at the time of writing, so I will focus on what they can bring you and refer to their documentation for installation and more technical discussions.
- For those doing theoretical research, I should point out that *writing a* LaTeX *document* is coding.

1.2 Motivation

Thinking about how you work to improve your efficiency is an effort, and it requires to challenge yourself from time to time. In my own experience, people may consider that it is a waste of time, and doing it the *quick and dirty way* is enough. Or they already know one method to accomplish a task that is enough for them, and they want to stick with it. In the short term, it is definitely faster, but I think taking the time to improve your workflow always proves worthwhile, at least in the mid-long term.

For sure, I can testify that during my PhD thesis, it allowed me to go further in my research, more than I could without trying to look for better tools to help me in my work. Without even talking about computer tools, just thinking about the way you can improve your workflow is already a great start! Challenge your habits, try to see where you spend the most time, and look around how people do it.

1.2.1 A simple example

When I was an undergraduate student in applied mathematics, I was always spending a lot of time, not just producing data for my numerical experiments, but post-processing them (organizing the data files, plotting the output, …). At the time, I was more focused on the core of my work, which was solving a partial differential equation. So I did not want to spend too much time on "technical details", and ironically, I was…

It came to a point where it was ridiculous, and something needed to be done. Like some students, I had one executable/script doing everything (computations like solving a linear system, analyses like doing a linear regression, plots, …), and every parameter was hard coded. It was convenient (I thought) since I just had to push a button, and I had directly the results I wanted. But here are the **issues**:

- Surprisingly, nothing works the first time, and having everything in the same code makes you debug everything at the same time.
- The analysis you want to carry over your data is never relevant the first time, so you need to rerun everything each time you want to explore your data, in particular run computations which is the most expensive part.
- The first plots are always ugly, so you also need to rerun everything.
- In case of compiled code, you need to recompile each time you want to change some parameters.

My solution (that may seem obvious to you now, but remember, I was an undergraduate student at the time) consisted in formalizing my numerical experiment as a pipeline where each code component could be implemented separately and needed to take some inputs and create some outputs:

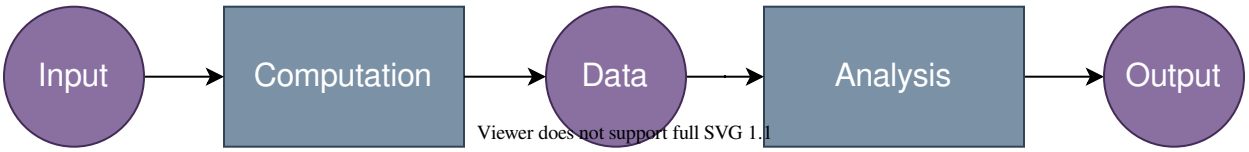


Fig. 1.2.1: Work pipeline

Obviously, it seems more complicated than doing everything in one code/script, and you need to manage inputs and outputs for each code component. But, it has several **advantages**:

- The fact that *Computation* takes *Input* as an argument makes you able to loop over the input parameters, and to generate more data with a simple script.
- If you have several pipelines with similar *Computation* component, you can try to merge them to make one pipeline, that takes more input to recover the exact previous computations.
- You can do as many analyses as you want on *Data*.
- Using well-know formats, such as *csv*, *json* or others (depending on your data) for *Input*, *Data* and *Output* makes you able to use most of the languages/software programs I/O utilities, post-processing and plotting tools. For example, you can easily use the python modules *pandas* and *matplotlib*, LaTeX package *pgfplot*, etc.
- You can save and store your results *Data* and *Output*.

Of course, you should not break your pipeline into too many components, there is a balance to be struck, and this was a very simple/naive example just to show you how changing your workflow actually allows you to **do more in your core work**. In this example, more computations with more inputs, and more analyses using more post-processing components.

1.2.2 Take home message

It is very valuable to take the time to work on your workflow, even for your core work. Another reason is that you are very likely to learn new tools, and thus get new transferable skills that you will probably be able to use throughout your career. Besides, it is also very intellectually satisfying to see we can improve and to see the net profit we get in our daily work life.

1.2.3 The goal

In this document, you will find several introductions to tools that can help you improve the way you code, from the most basic to more advanced. The focus will be more on what each tool can provide, I will not dwell upon all the installation procedures, but references to relevant documentations will be given. More generally, I will try to always give references for you to start looking out and go further. As I said earlier, a workflow is *personal*, and you need to make it your own.

1.3 To follow along

1.3.1 Requirements

This document assumes that you have a unix-like operating system (Linux, macOS, not Windows but see *About Windows*), so that you have access to a terminal (see *Bash*). To follow along the presentations of each tool, you will need a source code editor[1]. There are a plethora of choices.

In the following, I will often point out how each tool integrates with VS Code and its extensions, because this is the editor I use and recommend, but taste and colours are not always the same, so here is a non-exhaustive list of other free, cross-platform and multipurpose source editors.

- Editors with text user interface: emacs, nano, vim or neovim.
- Editors with graphical user interface: gEdit, Sublime Text.

See this Wikipedia article for more editors.

1.3.2 Visual Studio Code

VS Code is a free code editor that can be used with a great variety of programming languages. One of its main advantages is its great extensibility. On its own, it is a relatively simple source editor, but anyone can write extensions to add new features into VS Code, improving support for programming languages or adding new functionalities, to the point it can become a full Integrated Development Environment (IDE) for most programming languages.

Where to start

Let's start from the beginning, you need to install VS Code on your workstation and learn how to use its basic editing features. Since I will not do better than the documentation, I refer to their Setup page for the installation procedure, while you can look at their Get Started page and Basic Editing page for basic editing features.

I want to highlight the Tips and Tricks page, where it is mentioned for example that you can learn how to use basic editing (and more) using *walkthroughs* (on the right in the landing page of VS Code).

In particular, the walkthroughs "Get Started" and "Editor Playground" are really nice, you will learn how to

[1] Or at least you need to be able to modify files somehow. But if you are here, I guess you will not just use bash commands to do that, so a source editor is probably more suited.

- use the command palette `Ctrl/Cmd+maj+P`[2] to call in VS Code command (and that is very *very* **very** important),
- change the look of the editor,
- save and sync the settings of VS Code,
- use multi-cursor, snippets, intellisense, …
- and much more!

Extensions to help you start

Once you took a look at the previous references, you should know how to install extensions on the marketplace (*View* > *Extensions* otherwise). I will only give here some extensions that I think are generally useful and that I used to write this document.

- Project Manager can be used to define "Projects" (or auto-detect git repositories and define a project for each one), which in practice means you can open a project folder easily in a VS Code window (either using `Project Manager: List Projects to Open` in the command palette, or directly in the Side bar on the left).
- LTeX provides spelling, *styling* and **grammar** checking for several programming languages (LaTeX included) in several languages (more than 20 supported languages!) using LanguageTool. I wonder if people really understand how amazing it is.
- Live Share enables sharing your editor in real time remotely, meaning you can invite people in your VS Code window, and collaborate directly writing code together. In practice, everyone can independently move around and edit files, you all have a cursor, and you can also follow a particular person as he makes edits and move around in the project. It is obviously useful for pair programming and teaching.
- Better Solarized is a good theme (a look for VS Code) I use, but again, taste and colours are not always the same. The point is you can use another theme than the default one.
- Draw.io Integration to integrate diagram.net in VS Code, which means you have a graphical interface in VS Code to draw diagrams and more.
- :emojisense: to easily insert emojis (Why not? 😛)

I will recommend more specific extensions when introducing each tool later on.

1.3.3 About Windows

I recommend using a unix-like operating system to code, because most of the tools for programmers are tailored for this.

If you still want to use Windows, there are actually alternatives to a full unix-like system.

- Recent versions of Windows actually comes with an "embedded Linux kernel", this feature is called Windows Subsystem for Linux (WSL). You can run a GNU/Linux environment directly on Windows, but this is not enabled by default. I refer to its documentation for the installation, but I want to highlight the tutorial part of the documentation, and more precisely the page on Best practices for set up which gives a lot of information.
- There exist tools that provide a unix-like environment on top of Windows: Cygwin and MinGW-w64 for example.
- Another option is to install Linux with a dual boot, but it may be quite involved.

In the context of this document, I suspect the first solution is easier to use since it is directly supported by Microsoft, and the goal is just to try the tools introduced later on. Besides, *Visual Studio Code* has support for WSL via the Remote -WSL extension.

[2] It means `Ctrl`, except on macOS where `Cmd` should be used instead.

1.4 Changelog

All notable changes to this project will be documented in this file.

1.4.1 Unreleased

Added

- Add automatic list of contributors
- Add *pyproject.toml*, in particular for dependencies
- Add references for git

Fixed

- typos (#1). Thanks Marcella (@mbonazzoli) !
- typos (#2). Thanks Zois (@zmoitier) !
- fix numerous warnings

1.4.2 [0.1] - 2022-08-18

Added

- First draft of the introductions to some basic tools (bash, git and ssh)
- A section to introduce this document, with notes on the required setup to follow along the document and try out the presented tools.

CHAPTER

TWO

BASIC TOOLS

2.1 Terminal - Bash

One crucial tool a developer needs is a *terminal*. It is a tool to interact differently with your computer, i.e., using a command line interface (CLI). In other words, it allows you to give text commands to the operating system, instead of manipulating graphical elements (windows, icons, …). While the latter is easier to begin with, the former has the advantages to allow scripting and automation of your tasks, to use less resources, and it only uses the keyboard which can make you more efficient.

2.1.1 Terminology

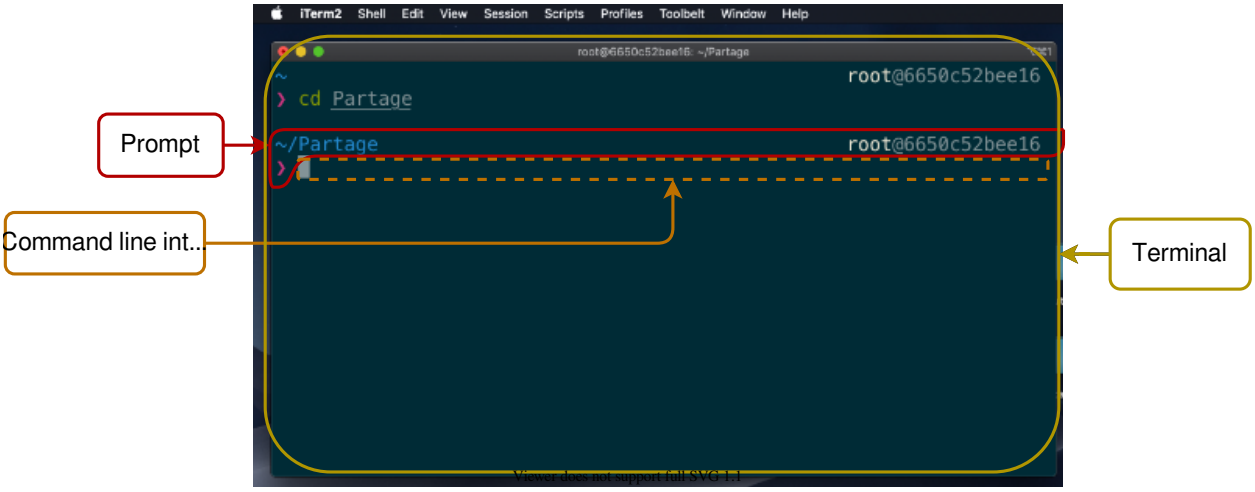


Fig. 2.1.1: Terminology

- Command-line interpreter, or *shell*: a program that processes commands, which allows users to access operating system's services. The syntax (command's names, arguments, …) is specific to the shell. Examples of shells: `sh`, `bash` (the most common), `fish`, and `zsh` which is discussed *here*. A command is typically its name with eventually some arguments. For example,

```
echo 'Hello World!'
```

  where `echo` is the command we wish to execute, and `'Hello World!'` is its argument. Here the command just prints its argument, give it a try 😉

Note: Commands can have multiple arguments, delimited by whitespace characters, and even options, usually with a long name `--option1` and a short name `-opt1` (note the number of dashes). For example:

```
echo -e 'Little darling\nThe smiles returning to the faces' 'Little darling'
```

will print all its arguments and the flag `-e` enables the interpretation of the escape character `\n`.

- Terminal, or *terminal emulator*: a text interface to a shell. In other words, it is the software in which users can access to the command-line interface. Before, terminal refereed to a physical hardware with a keyboard and a monitor. They are now mostly *virtual*, and in practice, it is the window in which we can interact with a shell. Examples of terminals (see list):
  - on macOS: Terminal.app, iTerm2, …
  - on Linux: Konsole, GNOME terminal, …
  - on Windows: Windows Terminal, PuTTY, …
- Command prompt: a sequence of characters in the command-line interface (CLI) that tells you the shell is ready to accept a command. It *prompts* you to give a command. It usually contains other information (path of the current directory, hostname, …).

Note: VS Code comes with its own terminal, which you can use along side having opened files and editing them.

2.1.2 Filesystem

Files are usually displayed in a hierarchical tree structure, where each node is a directory. In Fig. 2.1.2, you can see a typical filesystem for an operating system with two users, Alice and Bob. Both of them have their own directory, called *home directory*, where they can put their documents, install their software programs, etc. Every file and directory is under the *root* directory `/`, which is literally the root of the hierarchical tree structure.

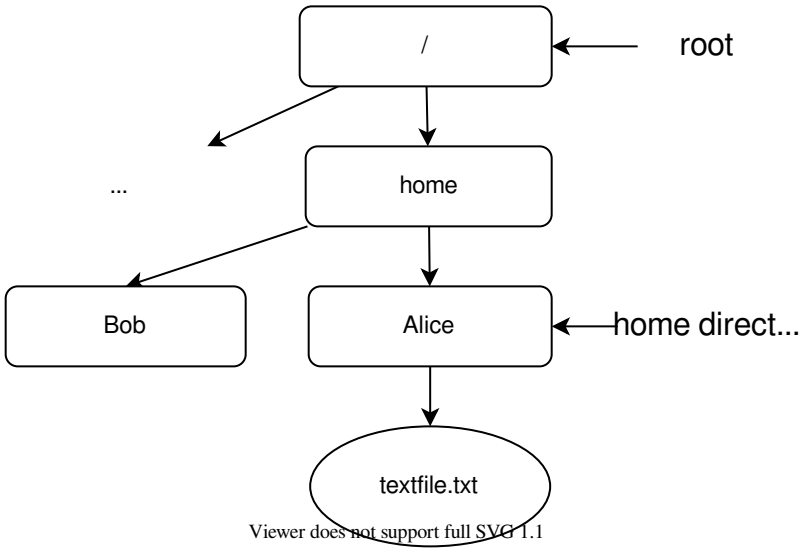


Fig. 2.1.2: Directory hierarchy

Every process (shell, commands that are executed, …) has a *current working directory*, which is simply the directory in which it is working from. For example, a user starting a new shell will usually have the home directory as current working directory.

Now that we understand that each directory and file are parts of a hierarchical tree structure, we need to locate them in this structure. To do that, we use a *path*, a string of characters composed by directory names and a delimiting character: `/` on Unix systems. There are two types of path:

- *absolute path*, which corresponds to the location starting from the root directory. For example `/home/Alice/textfile.txt`.
- *relative path*, which corresponds to the location starting from the current working directory. Assuming the latter is `/home/Bob`, the relative path to `textfile.txt` would be `../Alice/textfile.txt`, where `..` refers to the parent directory.

2.1.3 Variables

Shells can often be seen also as programming languages, in particular you can write scripts with variables, if/for loops, and functions. We will not dive in too much in this direction, but we need to understand at least variables.

You can set variables using = (without whitespaces around), here is an example with a variable containing a string:

```
my_variable="Hello World!"
```

And you refer to the value of a variable using `$`,

```
echo $my_variable
```

This is important to understand because when starting a shell, it sets *environment variables* to record the properties of the new shell session. You can see the active environment variables using the command `env`. In particular, you should find the following variables:

- `HOME` containing the path to the home directory,
- `SHELL` containing the path to the shell,
- `PWD` containing the path to the current directory,
- `PATH` containing a list of important paths, used when compiling code for example,

and many others… It also sets a number of other variables (that you can also print with the command `set`), including the variable `PS1`. This variable contains the command prompt, which was mentionned in *Terminology* . By default, it usually contains only `$`, but it can be customized to display more information.

2.1.4 Navigation

Once you have started a shell session, the first thing you can try is to check where you are on your laptop with the command `pwd` (**p**rint **w**orking **d**irectory).

```
pwd
```

It will print out the absolute path to your current directory, something like `/home/YourName`. To change the current directory, you can use `cd` (**c**hange **d**irectory) followed by the (absolute or relative) path of its new location. For example, if we want to go the root directory.

```
cd /
```

Then, you can check that you are at the root directory using again `pwd`.

Tip:

- Calling `cd` without argument changes the current directory to your home repository.
- Calling `cd -` changes the current directory to its previous location.
- Calling `cd ~` changes the current directory to the home directory.
- Calling `cd ..` changes the current directory to its parent directory.

Note:

- You can make a path combining different shortcuts, for example `cd ~/../..` will change the current directory to the root directory in the previous *example* (home directory, then go up two levels).

To know where to go, you may need to know what are the files and directory contained in a given directory. You can use `ls` (**l**ist **f**iles) to print them out.

2.1.5 Change to the filesystem

We can now start to modify the hierarchical structure adding and removing files and directories.

- To create an empty file named `my_textfile.txt`, use `touch`

```
touch my_textfile.txt
```

- To create an empty directory name `my_directory`, use `mkdir` (**m**a**k**e **dir**ectory)

```
mkdir my_directory
```

- To remove a file name `my_textfile.txt`, use `rm` (**rem**ove)

```
rm my_textfile.txt
```

- To remove a directory named `my_directory`, use `rm` with the flag `-r` or `--recursive` to allow recursive deletion of the directory's content

```
rm -r directory
```

Warning: Be careful when deleting files and directories, it is quite involved/impossible to recover what you delete with `rm` (no recycle bin), and you risk breaking your system by deleting the wrong file or directory.

2.1.6 Tips and tricks

Autocompletion

Use `tab` to autocomplete paths. When writing the beginning of path, use `tab` to autocomplete. If there is not a unique possibility, it will display the different possibility.

Navigate in your history

Use `Up` arrow to navigate through your command history.

Backward research

Use `ctrl-r` to look for previous command calls. Use `ctrl-r` and type the beginning of the command you are looking for in your history, and you want to recover. It will show you the last command you used starting by what you typed, and you can use `ctrl-r` to look for previous commands starting by what you typed.

Man page

To learn how to use a command, use the command `man` with the name of the command as argument. For example `man echo` displays to manual for the command `echo` (type `q` to leave).

explainshell.com

This website explains commands and their flags. Just write down `echo -e`, then it will show you what `echo` is for, and what `-e` means for this command. It is a more visual way to access to the man page mentioned above for a given command and the specific flags you used. Note that I am not affiliated to this website.

2.1.7 Notes for VS Code users

As we previously mentioned, VS Code comes with its own terminal where you can use any shell available on your workstation, so no need for an extension a priori. Everything we mentioned here should work since it does not depend on the terminal, but is related to the bash shell.

That being said, I should point out that VS Code's terminal has some nice features you should be aware of, for examples:

- Terminal processes are restored on window reload, if you reload your VS Code window for example, it will not kill your shell session.
- Every path or URL displayed in the terminal is a link, meaning you can left-click holding `Ctrl/Cmd`[1] to use it (as in the editor). For a file, it will open it in an editor, which is very useful when compiling or debugging.
- You can use "Find" in the terminal using `Ctrl/Cmd+f` as you would expect.

VS Code also added recently a new feature called Terminal Shell Integration. For common shells (including bash), VS Code can understand what is happening inside the shell, which allows him to add some other nice features, for example:

- Command decorations: VS Code recovers the exit code of each command you call and displays blue dot on the left if it succeeded, a red one otherwise.
- Command navigation: you can quickly navigate between commands using `Ctrl/Cmd+Up` and `Ctrl/Cmd+Down`.

[1] It means `Ctrl`, expect on macOS where `Cmd` should be used instead.

I only mentioned some features for VS Code's terminal and shell integration, go look at the documentation to see them all. Besides, they will probably add more features in the near future.

2.1.8 References

Terminology

- Wikipedia for command-line interface, terminal emulator
- Questions on StackEchange: Unix&Linux and superuser
- List of terminal emulators
- Video of Luke Smith defining the terminology.

Filesystem

- Wikipedia for Unix and Unix-like filesystems: Filesystem Hierarchy Standard
- Wikipedia for home directory, root directory, working directory, path

Integrated terminal in VS Code

- VS Code's Documentation on its integrated terminal
- Mastering VS Code's Terminal: a blog with a lot of tips to improve and customize VS Code's terminal

2.2 Git

Git is an essential tool to collaborate and version your code. That is why, we will take the time to understand its benefits, and how to use it. The presentation will be organized around what git allows you to do: versioning, remote backup, synchronization and collaborative work.

Each one of these usages will require the previous ones. For example, you need to know how to version your code to use a remote backup, but you do not need to know a priori how to use git for collaborative work. So that, you can start by reading just what you need. If you want to go further, I give some *references* to other concepts in git that I do not talk about here.

Note that I present command lines to use git, but there exists a lot of graphical interfaces (see this list), but understanding how git works is still necessary to use them. You can also find a presentation I did a few years ago on git with a similar approach.

2.2.1 Setup

Configuration

We refer to this page for installation instructions, but before, check if you do not already have it. Then, the first thing you need to do is to configure git. You should at least set your identity as follows:

```
git config --global user.name "Username"
git config --global user.email username@mail.com
```

Each modification you make to your repository will be associated with this identity.

The flag `--global` just means that this identity will be used in all the repository you work with on your system. You can always set local configuration to override it, and you can also set other types of configuration variables like the editor used to write commit messages, diff tool and so on. We refer to this page if you want to go further, but it should be enough at first. You can check your configuration with `git config --list`.

Create repository

To start locally a repository, go to the folder you want to work with, here `YourRepository`, and use `git init` to initialize your git repository.

This will create a hidden folder `.git` with all the information of the repository. You should not modify anything in `.git`.

2.2.2 Versioning

The first benefit of using git is that it allows you to *version* your source code. It means that git will track your files, save their history efficiently, and give you the possibility to easily navigate through the different versions of your files. Using git, you can forget about versioning your files numbering their name like `file1.txt`, `file2.txt`, `filefinal.txt`, `filefinal1.txt`, ... and all the redundancy it implies.

Note that you can do it **locally**, even if in practice most people also use a remote to back up their repository. This will be discussed afterward, and we will keep it local for the moment.

Create History

Firstly, you need to put a file, here `FirstFile.txt`, in your repository and ask git to track it. To do so, you need two commands:

```
git add FirstFile.txt
git commit FirstFile.txt -m "first file added"
```

The first command makes the file `FirstFile.txt` *staged*, and the second one commit this version of the file in the repository's history, with a small comment. The repository's history can be represented as a graph/tree, where each commit is a node, containing a state of the whole repository, a comment describing the commit, a unique commit ID (a hexadecimal number of 40 digits), the commit date, the committer's name, and email address.

Note that if you do not add the `-m` flag followed by a string, `git` will open your editor (default to `nano`) for you to write a commit message instead.

In Fig. 2.2.1, you can see an illustration of a simple history. Each rectangle represents a commit, so a snapshot in the history of your repository with all the associated information, in particular, a commit ID. The commit `291bb0` is the first one, followed by `e9b2d0` which has a pointer to the previous state. That is why there is an arrow from `e9b2d0` to `291bb0`. Then, `main` is the tip of the history, and represents a *branch*. A branch represents a linear history of your repository, and in practice it is a pointer to the last state of a linear history. Here, we only have one branch. Finally, `HEAD` is the actual state on your computer, if you open a file tracked by this repository, its state will be the one of the commit `HEAD` points to. In this example, `HEAD` points to `main`, so the last state of the history.

Here, the name of the branch I created is called `main`[1].

[1] The current default is `master`, but git shows a message, when creating a repository, explaining this is subject to change, and it recommands to set the default name using `git config --global init.defaultBranch <name>` where `<name>` will be the new default name. I chose to use `main`.

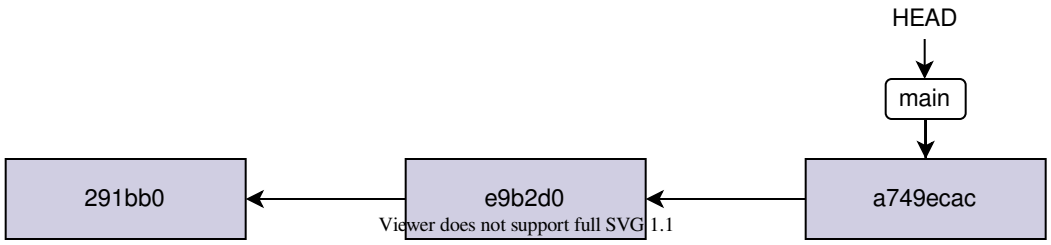


Fig. 2.2.1: Versioning

Note: Why do you need two commands just to update your repository?

It is usually the first point that confuses people discovering git. I refer to some *discussions* on the subject, but the bottom line is that the staged area (so, the files you used `git add` on) allows reviewing your changes before committing them to the repository's history. It also allows separating multiple changes in meaningful commits.

For example, if you add a feature to your code, and fix a bug at the same time. You can add only the changes related to your fix, review them, commit them with a specific description, and then do the same for your new feature. Remember that your commit messages need to be descriptive enough to easily navigate the repository's history.

Navigating through the history

Now that you populated your history's repository, you may want to go back and check previous states of your repository. The current state on your computer (not necessarily the last one) is called `HEAD`, see Fig. 2.2.2.

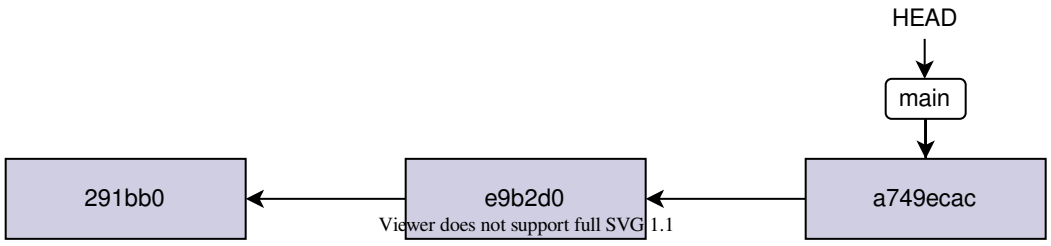


Fig. 2.2.2: Current state to last commit

The command `git log` shows your repository's history, i.e., commit messages, commit IDs, committer's names, and email addresses. And, `git log -2` will only show the information for the two last commits.

To navigate through your history, you can use

- relative references: to check out the second generation ancestor of `HEAD` use `HEAD~2`, see Fig. 2.2.3.
- to return to the last state of your repository, `git checkout main`, see Fig. 2.2.4.
- absolute references: using the commit ID, `git checkout e9b2d0` (first characters are enough), see Fig. 2.2.5

Tip: You are lost? `git status` will tell you where you are and what you can do.

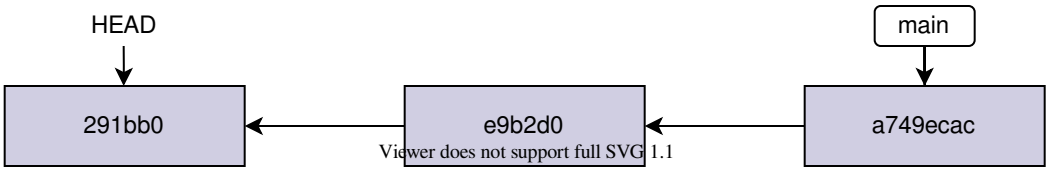


Fig. 2.2.3: `git checkout HEAD~2`

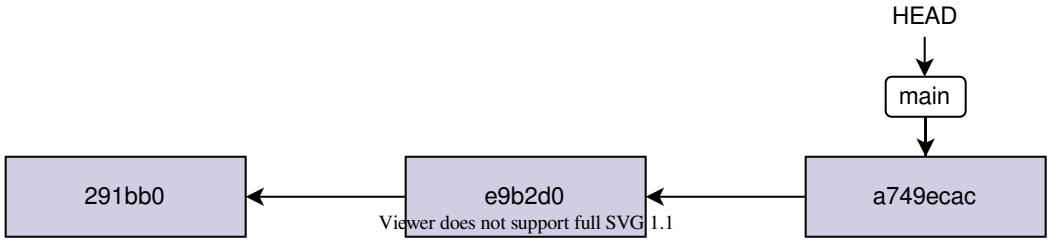


Fig. 2.2.4: `git checkout main`

2.2.3 Back up

Another advantage of git is the possibility to back up your repository in a remote server. It is said to be a *distributed* version-control system (unlike SVN for example), because both your local repository and the remote repository will have the full history after each synchronization.

Set up the remote

First, you need to create a remote repository in GitHub, GitLab, Bitbucket or some other providers (or your own git server).

Providers will usually give you instructions on how to set up your repository (see Fig. 2.2.6 for example). In any case, we need to add the remote URL to the local repository with

```
git remote add origin https://github.com/PierreMarchand20/YourRepository.git
```

The remote is then referenced as `origin`. And, we need to push the local commits to the remote

```
git push -u origin main
```

We have now a remote branch `origin/main`, which is the copy on the remote `origin` of `main` as described in Fig. 2.2.7.

Note: The example uses a https URL, but you can also connect to a git server via a *SSH* URL. This can be useful to avoid having to give a username and a password each time you want to update the remote repository.

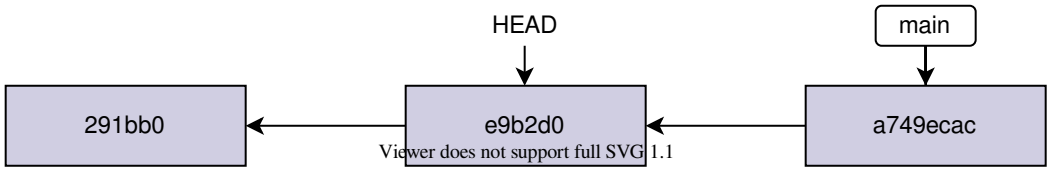


Fig. 2.2.5: `git checkout e9b2d0`

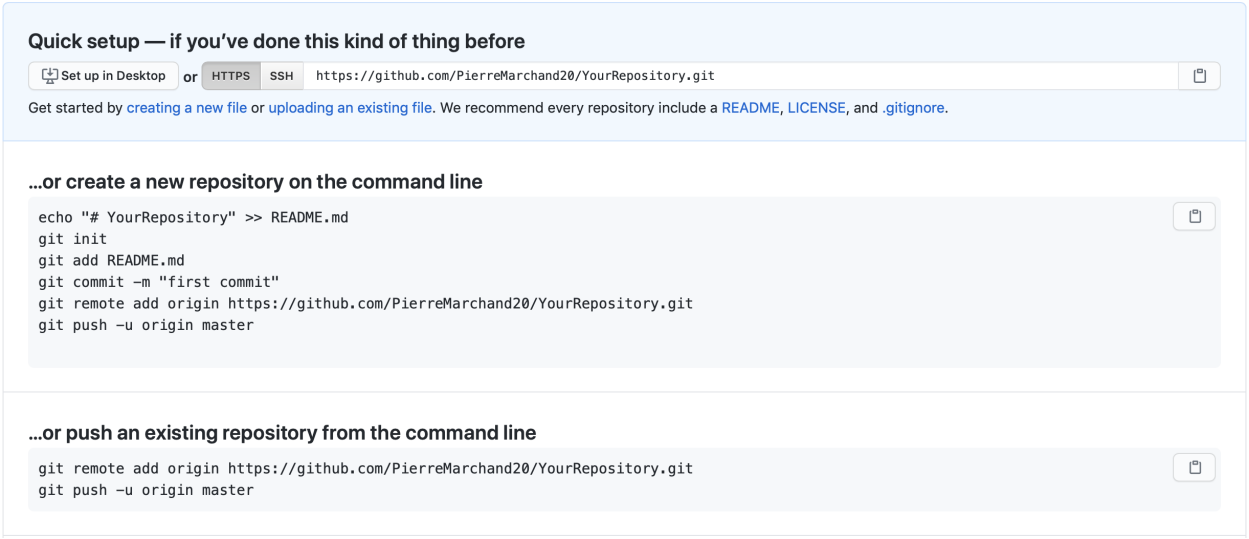
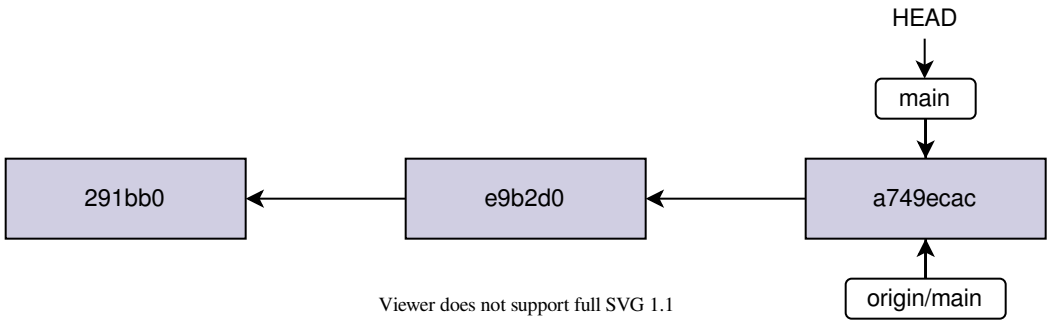


Fig. 2.2.6: GitHub



Viewer does not support full SVG 1.1

Fig. 2.2.7: Remote added

Working with a remote repository

Now, we create a new commit locally, so that the branch `main` is further than the branch `origin/main` on the remote (see Fig. 2.2.8).

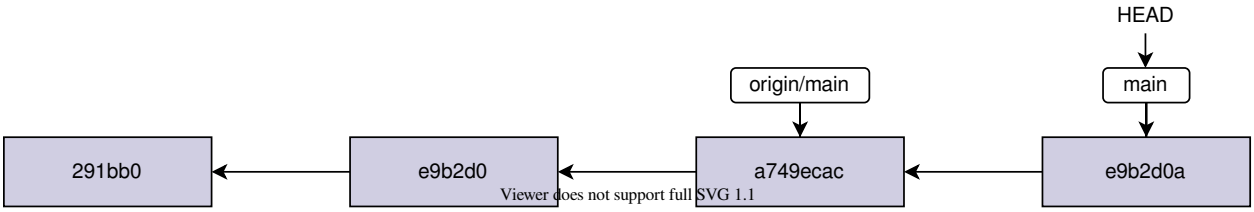


Fig. 2.2.8: Local new commit

We just need to do `git push` to update `origin/main` (by default, git will push to `origin/main`, no need to specify it).

2.2.4 Synchronization

Having a remote repository, you can also use it to synchronize a repository on several computers, let's say `Computer 1` and `Computer 2`.

Update from remote

Imagine you create a new commit locally on `Computer 2`, then you push this new commit to the remote repository. This time, it is `origin/main` that is further than `main` from the point of view of `Computer 1`! We are in the situation illustrated by Fig. 2.2.9.

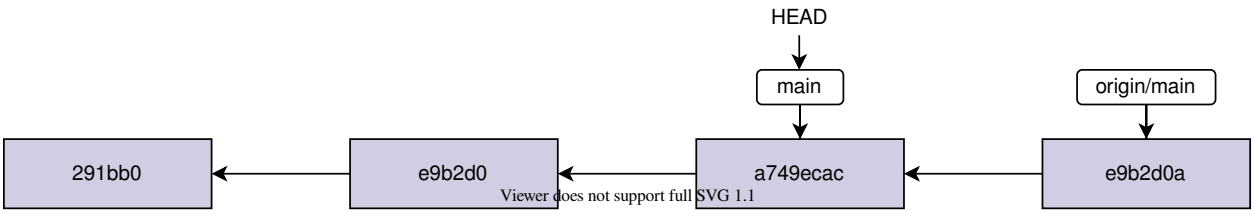


Fig. 2.2.9: Repository on `Computer 1`

To update your local repository, you just need to call `git pull` on `Computer 1`, and it will update `main` adding the last commits from `origin/main`. This is called a *fast-forward merge*, because there is no divergent branches, git just needs to update the local copy of `origin/main` to get the last changes and to move the pointer of the local branch (here `main`) forward to the tip of the remote branch (here `origin/main`).

Note: `git pull` can be seen as the combination of two commands:

- `git fetch` for the local branch. In our case, it updates the local copy of `origin/main`.
- `git merge`, which in the present case will do a fast-forward merge.

Issue

Something wrong can quickly happen with bad practices. Imagine you do a new commit locally on both computers. You push your new local commits from `Computer 2` to the remote repository. This time, `main` and `origin/main` have diverged from the point of view of `Computer 1`, which is described in Fig. 2.2.10.

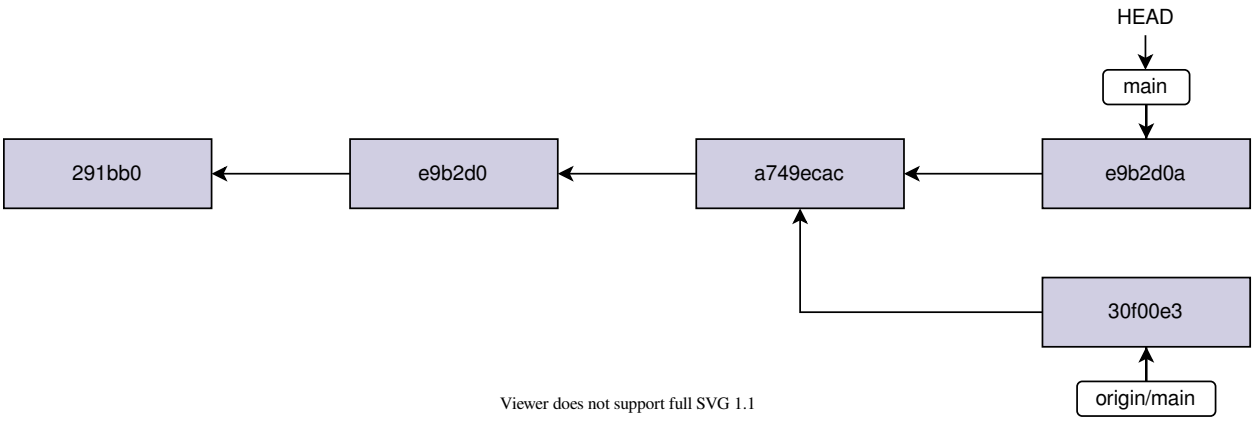


Fig. 2.2.10: Repository on `Computer 1` with diverged `main` branches

Two remarks here:

- It is usually what people discovering git fear the most! But note that it is not specific to git, if you modify one file locally on two computers, you will also have to deal with this situation. Actually, git will tell you that there is an issue if you try `git push` on `Computer 1`, and it will help you solve the issue. So git is a tool to help you deal with this situation, instead of doing everything by hand.
- That being said, you should avoid this situation because it is more likely to break your code. In the case where you are just synchronizing several computers of yours, you can always `git pull` when starting to work on one computer, add/commit all your modifications, and `git push` when you have finished. You should not be in this situation if you follow this workflow.

In case you still encounter this situation (you forgot to commit a change, or to push at the end of a working session for example), we refer to the next section.

2.2.5 Collaboration

If you want to collaborate with someone else, or if you work with a team on a project, then the previous *issue* may occur more often. It is very likely that your coworkers will commit some changes while you are also working on the repository, so that, you will be in the situation described in the Fig. 2.2.10 with divergent branches. To avoid this, you need to adopt a workflow, i.e., a way to work all together with the git repository. There are several solutions depending on how you work with your team/coworkers, the number of contributors, etc. It is an advanced subject, and I give some pointers for more information in the *references*.

But here are some general considerations shared by most of them. They usually aim at:

- making the history's repository/tree as flat as possible. This makes it easier to navigate between commits,
- avoiding situations with diverging branches, and thus, limiting the risks of breaking your code.

And, they usually rely on one of the two following git operations, if not both: `git merge` and `git rebase`. Both commands allow merging two branches, but the outcome is different as we will see.

Merge

Merging is used automatically by git when pulling from a remote which is further than the local branch. But it can also be used to merge two different branches locally. Actually, `git pull` means `git fetch`, which updates locally the remote branch (here, the local copy of `origin/main`), followed by `git merge`, between the remote branch (the local copy of `origin/main`) and the local branch (`main`).

Let us take an example. We have a file `FirstFile.txt` that contains the following three lines:

```
This is the first file
This is the first file
This is the first file
```

On `Computer 1`, we modify it to

```
This is the first file - modified by Computer 1
This is the first file
This is the first file
```

On `Computer 2`, we modify it to

```
This is the first file
This is the first file
This is the first file - modified by Computer 2
```

The first line is modified by `Computer 1`, and the third line is modified by `Computer 2`.

Now, we commit both changes locally, we push the modifications by `Computer 2`, and pull on `Computer 1`. Note that git is safe, if you try to push changes from `Computer 1`, it will be rejected because `main` on `Computer 1` is behind `origin/main`. When pulling on `Computer 1`, because the modifications from both computers are not overlapping, git actually proceeds to merge automatically the changes, and create a commit stating the merge. Then, you just need to push on `Computer 1`, and pull on `Computer 2`, and we obtain a history as Fig. 2.2.11.

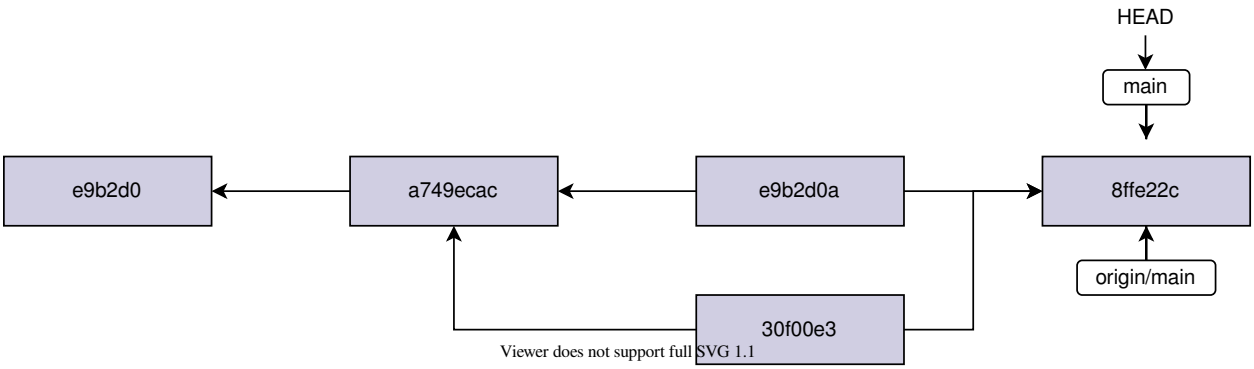


Fig. 2.2.11: Merging

Note: This is the default behaviour of git when calling `git pull`, but in recent versions, it also displays a warning explaining how to use rebase (see next *section*) instead of merge when the changes do not overlap. You can configure the default behaviour with `git config --global pull.rebase true` to change it to use rebase. You can also disable the default strategy to reconcile divergent branches (so no `merge` or `rebase`), and only enable *fast-forward merges* with `git config --global pull.ff only`.

Let us look at the case where the modifications are overlapping. On `Computer 2`, we do the following change instead:

```
This is the first file - modified by Computer 2
This is the first file
This is the first file
```

If we commit locally on both computers, and we push on `Computer 2`. Then, when pulling on `Computer 1`, auto merging fails, and `FirstFile.txt` contains now:

```
<<<<<<< HEAD
This is the first file - modified by Computer 1
=======
This is the first file - modified by Computer 2
>>>>>>> 438c30414304658df44ef2dfd735abea47c7025a
This is the first file
This is the first file
```

We see the change from the local `HEAD` (so, `Computer 1`), and the change from the commit on the remote (so, `Computer 2`). We just need to modify `FirstFile.txt` as we want, then stage it and commit.

Rebase

While `git merge` creates a new commit, as illustrated *here*, `git rebase` changes the base of one branch to put it after the last commit of the other branch. Taking the same example illustrated *here*, we can do `git fetch origin` on `Computer 1` to update the local copy of `origin/main`, and then `git rebase origin/main`. Git will start an "interactive rebase", reviewing each commit from the diverged part of `main`, one after the other, to check if there are overlapping differences with `origin/main`.

If there are overlapping differences between a commit from `main` and `origin/main`, you need to fix them, then use `git add` with the fixed files which will modify the current reviewed commit and use `git rebase --continue` to go onto the next commit to review in the interactive rebase.

With the example from Fig. 2.2.10, using a rebase strategy instead of merge will produce a linear history, see Fig. 2.2.12.

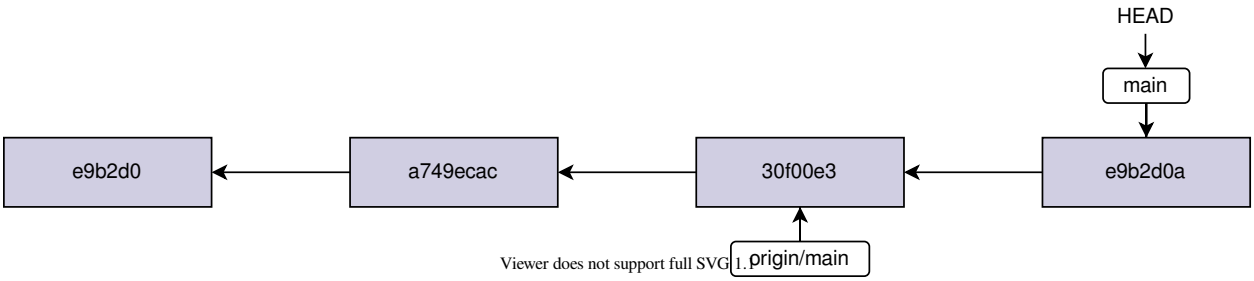


Fig. 2.2.12: Rebasing

where the diverged commit `e9b2d0a` is now behind `30f00e3`. We moved the base of `main` to the tip of `origin/main`.

Note: In recent versions of git, you can use `git pull` instead of using `git fetch` and `git rebase` to apply the same strategy. As mentioned in *Merge*, you need to change its default behaviour with `git config --global pull.rebase true` to do so.

This is particularly useful to avoid an additional commit, and in the case of two different branches, it allows preserving both history. But, there is one golden rule when using `git rebase`. It should **not be used with public branches** For

example, you should not rebase `origin/main` instead of `main`, because it would modify the commit history of the branch shared with other workstation/people.

2.2.6 Notes for VS Code users

VS Code already comes with an extension for git, so that you can do most of the basic git commands (push, pull, add, commit, …) directly via the graphical interface of VS Code. Everything is in "Source Control" in the Activity bar on the left. You can also access it via *View > Source Control*.

- Changes: you can find an overview of all the changes, and if you click on a modified file, it will display the modifications of the file.
- Commit: you can select the changes you want to add, write a commit message and commit.
- etc.

VS Code will also display information directly in the editor.

- Gutter indicators: when opening modified in files in the editor, VS Code will add an indicator on the left to the modified lines.
- Merge conflics: it will add colours to merge conflicts, and buttons to accept either one or both change.

You can find the documentation here with all the features. But git integration in VS Code can go even further with additional extensions.

- GitLens adds an enormous amount of git-related features, among which,
  - Enriched Source controle view with a list of all the commits of the current branch and provide quick access to the modified files in each commit (and show the modifications when clicking on it).
  - Information directly to the editor, with for example, blame annotations (author, date and message from the current line's most recent commit) at the end of the current line, and much more
  - And a lot more!
- Git Graph displays a graphical representation of the repository, from which you can also do most the git commands.

2.2.7 References

General presentations

- Pro Git book by Scott Chacon and Ben Straub, free and available in several languages.
- Introduction to Git with Scott Chacon of GitHub on YouTube.
- An interesting discussion on Quora: What is git and why should I use it?.

Specific discussions

- Discussions on why you need to add and commit here and there.
- Several possible workflows for teams are described here by Atlassian.
- Lists of GUIs here.

Other references

- Tutorial of Infomath
- multilingual interactive git cheatsheet
- open source game Oh My Git!

To go further

- Stashing
- Submodules
- Branches

2.3 SSH

The Secure Shell (SSH) protocol is a network protocol that allows secure access to systems running an SSH server over a network (e.g. the Internet). All the communications from the client (e.g. your workstation for example) to the server (e.g. a remote workstation) are encrypted.

SSH is widely used, examples of applications are

- accessing severs remotely (student workspace hosted by the school, supercomputers, *Git* servers, remote workstation, …) with or without password,
- transferring or syncing files,
- mounting a directory on a remote server as a local filesystem.

SSH client and server are present on most operating systems, including macOS, most distributions of Linux, and even (recent) versions of Windows.

2.3.1 Access to a remote computer

To access a remote computer via ssh, the administrator of the server needs to set up your account. You should have a username and a password.

Password-based authentication

The SSH command to access a remote server with a password is as follows

```
ssh username@hostname
```

where `username` is the username associated with your account on the remote computer and `hostname` is the name associated with the remote computer. The latter can be an IP address or a domain name.

This command will prompt you for your password (associated to your account on the remote computer), and will then open a terminal session on the remote computer.

Note: The first time you connect to a remote computer, it will ask you if you want to add the host to `known_hosts`, which is a file you can find in `~/.ssh`

Key-based authentication

Using SSH, there is an alternative to password-based authentication which uses a public key and a private key. More precisely, it uses an asymmetric encryption algorithm where the public key is used for encryption, and the private key is used for decryption.

The idea is that the user generates both keys, and send the *public* key to the remote computer. Then, each time the user wants to connect to the remote computer, the remote computer will "challenge" the user sending a message encrypted using the public key. The user will send back the decrypted message, which can only be done with the private key. Note this is done automatically behind the scenes using the same command as before.

This approach is usually considered safer than password-based authentication, mainly because the private key never leaves the user's workstation. But remark that if the private key is stolen (by someone accessing your workstation for example), then the remote computer is compromised (as compromised as if your password was stolen). That is why it is advised to add a layer of security by encrypting the private key itself with a passphrase, which is suggested when creating the keys.

Finally, this approach provides passwordless access to the remote computer, meaning you can access several times to the remote computer, without having to type your password each time.

Creating private and public keys

To create a pair of public/private keys, you need to use the following command

```
ssh-keygen
```

It will first ask you about the name of the file (default to `~/.ssh/id_rsa`), and for a passphrase. Note that the latter is optional, if given, it is used to encrypt the private key adding a new layer of security. It will then create two files: `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub` by default, the first is the private key, and the second the public key.

Sending a public key to a remote computer

Once the pair of public and private keys is created, you need to send the *public* key to the remote computer. It can be done with the following command

```
ssh-copy-id username@hostname
```

If the public key name differs from the default, add the flag `-i path/to/my_id_rsa.pub`.

Accessing a remote computer with key-based authentication

If the private key is not encrypted, in other words, if there is no passphrase, then you can connect to the remote computer using the same command as in *Password-based authentication*, and it will not prompt you for your password.

If the private key is encrypted, each time you will try to connect, it will prompt you for the passphrase to use the private key. To avoid this and still have passwordless access to the remote computer, you can use `ssh-agent` that will cache the decrypted key for the rest of your terminal session.

You first need to call this command to set up `ssh-agent`

```
eval $(ssh-agent)
```

and then, you can cache a key using `ssh-add`

```
ssh-add ~/.ssh/id_rsa
```

In this manner, you can access the remote computer using `ssh` as before.

Configuration file

If you want to avoid remembering all the usernames, domain names, and command-line options, you can set up a configuration file where you can write everything once.

This file does not exist by default, so first you need to create it

```
touch ~/.ssh/config
```

The format of the configuration file is as follows

```
Host hostname1
    Option1 value
    Option2 value

Host hostname2
    Option1 value
```

A minimal configuration file using our previous example with `hostname` as the host and `username` as the username gives

```
Host my_host
    HostName hostname
    User username
```

With this configuration file, one can use `ssh MyHost` instead of `ssh username@hostname`. Here we do not gain a lot since this is a simple example, but a configuration file can quickly be useful.

2.3.2 Transferring files

Several file transfer mechanisms based on SSH exist. You can always use either password or key-base authentication,

Secured copy protocol

`scp` simply reads the source file and writes it to destination.

```
scp local_file host:local_file_sent_to_remote
scp host:remote_file remote_file_sent_to_local
```

rsync

`rsync` is highly configurable tool for copying files. It can be used locally, or over a network with SSH. One difference with `scp` is that it uses a "delta transfer algorithm", meaning in particular that only the difference between the source and target files will be sent.

It has many flags allowing for fine-tuning of its behaviour. Here are some:

- `-a` to keep almost[1] everything the same recursively.
- `-u` to *not* update if files at destination are newer.
- `--progress` to show the progress of the transfer

[1] The main exception concerns hard links, see documentation for more information.

2.3.3 SSH Bastion

Usually remote computers are not directly accessible from the Internet. You first need to access to a remote server (called "Bastion") from which you can access to a remote computer. It allows isolating a private network (where the remote computer is) from the Internet, making it only accessible via the Bastion.

To automate the process of first connecting to the bastion, and then to the remote computer, you can use the following command

```
ssh -J username@bastionname username@hostname
```

where `bastion` is the domain name of the SSH bastion. You can also use the following configuration file that defines the bastion host and the remote computer

```
Host my_bastion
    HostName bastionname
    User username

Host my_host
    HostName hostname
    User username
    ProxyJump my_bastion
```

You can then connect to the bastion using `ssh my_bastion`, or directly to the remote computer with `ssh my_host`. It will prompt you for your password twice if you use a password-based authentication, or you can send your public key to both remote servers to use key-based authentication.

2.3.4 Notes for VS Code users

VS Code has useful extensions to work on a remote computer.

- Remote - SSH can be used to open a folder on a remote computer running an SSH server. In practice, this will open a new VS Code window (workspace in VS Code jargon) where you can interact with the remote files and folders, open a terminal on the remote computer, and even install extensions. This way, one can work on this remote folder using all the usual VS Code features (intellisense, code navigation, debugging, and so on) as if it was local. I refer to its documentation, but note that it will find SSH targets defined in *Configuration file*.
- SSH FS is another alternative I use when the previous extension is not available, which can happen if the operating system on the remote computer is not supported. It mounts a remote filesystem as a local workspace folder. The integration with VS Code is not as advanced as with the previous extension since the workspace is local. Features like intellisense, code navigation, debugging will not work since it does not have access the remote environment, but basic features such as syntax highlighting should work and this is often enough.

2.3.5 References

- Wikipedia for Secure Shell, Secured copy protocol and rsync
- Article on SSH keys from ArchWiki
- StackExchange discussions on the subject can be found here and here
- StackOverflow discussion on ssh vs rsync

CHAPTER

THREE

ADVANCED TOOLS

3.1 Terminal - ZSH

It is an alternative to bash, and it is going to be the default shell on macOS starting from Catalina 10.5.

3.1.1 Z Shell

`zsh` is a shell derived from the Bourne shell `sh`, like `bash` we discussed *here*. It is the reason why they both share a lot of features. If you already know how to use `bash` (`cd`, `mv`, `touch`, `mkdir`, ...), don't worry, you won't start from scratch again. Note that to make it your default shell, you can use:

```
chsh -s $(which zsh)
```

Note: You can use any shell interactively, and if you write a script, you can specify the shell using a shebang (`#!/usr/bin/env bash` for bash) at the beginning of your shell scripts.

Here are some interesting features of zsh (see next *section* to activate all of them):

- **Completion** (builtin and improved with zsh-completions): just use `tab` to complete your command line
  - **recursive path completion**: no need to write the full directory/file names
  - **command arguments completion**: `zsh` suggests arguments to the command you wrote with a description of each option (for example, `git` we saw *here*)
  - **command arguments flags**: same as the previous feature, but for flags
  - **variable expansion**:
- **Better history navigation**: you can search for a command in your history with any substring of this command using up and down arrows (with history-substring-search)
- **Autosuggestion**: the last command starting by what you write is suggested and you can use `tab` to autocomplete it (with zsh-autosuggestions)
- **Syntax highlighting**: a few examples (with zsh-syntax-highlighting)
- **Plugin and theme support**: zsh is known for its *plugin managers* that allows installing/activating the previously cited plugins and many others, but it is also known for its highly customizable *prompt*. We are going to see how to benefit from them in the following.

3.1.2 Plugin managers

As we have seen, some features are available via plugins. You could install them by hand, each repository explains how to do it. Usually you have to download them, source them in your `.zshrc` and set some variables. But it can be tricky because the order in which you source them matters, and having a lot of plugins can add a delay when starting a new shell session.

Another possibility is to use a *plugin manager*. There are a lot of them (see *reference*), I personally use Zim that I find fast and easy to use. Besides, it is well-maintained, and the maintainers were quite helpful when I had a question. I tried to use a few other plugin managers, most of them are great, but some added a delay when starting a new shell session, and that is how I tried `Zim`, which is marketed as fast. I was also convinced by the fact they thought about how their project should grow.

The installation process is quite simple, and default configuration should give you most of the features described previously. To add or remove modules, you need to add a line with `zmodule` in `.zimrc` and run `zimfw install`. See documentation here.

3.1.3 Prompts

The benefit in customizing your prompt is that it allows you to display more information. `git`, that we introduced in *here*), is the usual first example. You can display the current branch, and if there are modifications to be committed. But you can also display timing between commands, battery level, and a lot of other information. I personally like to keep it simple, but you do you 😉

Similarly to plugins, you could define a customized prompt by hand. But the risk is to add a delay each time you enter a command because of the loading time of the prompt.

Similar problem, similar solution: people have already defined optimized prompts that allow for customization while avoiding delay most of the time. Two popular prompts are Spaceship ZSH and Powerlevel10k. I personally use the latter at the moment, but they are both fast, customizable and easy to use. The installation process is quite straightforward, and the configuration is done interactively.

Here is an example with Powerlevel10k where I show current folder, current git status (notice the `?1`, which means there is one file not tracked), python virtual environment, time, and a custom prompt that shows ⋆ with `my display`.

3.1.4 References

Z Shell

- Website of zsh.
- Some features of zsh
- Resources about zsh.
- Bash vs zsh on Stackexchange

Plugin managers

- Some plugin managers: zim, oh my zsh, antigen, zplug, zinit, …
- Benchmarks for plugin managers: zim benchmarks, a Reddit thread.
- a Reddit thread on plugin managers.

Prompts

- Customizable and efficient prompts: Powerlevel10k, Spaceship

3.2 Dotfiles

If you followed my previous posts, you may find in your `${HOME}` directory several files starting with a dot, or dotfiles: `.gitconfig` for your git configuration, `.zsh` for you zsh configuration, and `.p10.zsh` if you tried Powerlevel10k.

Of course, there are probably a lot of other dotfiles from

- your editor or your terminal (`.vimrc`, `.bashrc`, etc.),
- software programs or libraries you use (I have one for iTerm2 and matplotlib for example).

This is not an exhaustive list, but `${HOME}` is the place where user configurations are usually stored in plain-text files.

3.2.1 The goal

Since all these dotfiles are just plain-text files, a natural idea is to save them, or to version them, to actually back up your settings, sync them across multiple machines and be able to deploy them quickly. This is particularly useful when getting a new computer, using a remote computer, or for some other reasons we will see in a later post. 😉

That is why, when talking about *dotfiles*, people refer to a *git* repository that stores all these dotfiles, but also contains some scripts (sometimes called *bootstrap*) to set everything on a new environment:

- install the dotfiles into `${HOME}`,
- install utilities (for example, a plugin manager for `zsh` or an enhanced prompt as we have seen in with *zsh*),
- install libraries (via Homebrew and processing a `.BrewFiles` on macOS for example),
- etc.

And it needs to be easy to deploy, with a one-line command in the terminal for example.

3.2.2 How-to

As always, there are lots of possibilities to achieve this, and it also depends on your particular needs, so you need to try and fail to find what is the best for you.

That being said, there exists several tools to do all the heavy lifting. At the moment, I use YADM which I find easy to use and well-maintained. I refer to its documentation for more details, but it really feels like using *git* in your ${HOME}, which could be cumbersome to do directly. Besides, it offers the possibility to run a script, called *boostrap*, when cloning a repository via YADM. You can use it

- to set up your *zsh* or *git* (the `git config` options),
- or to install packages using your system/language package manager

  - on macOS when using Homebrew: you can store a .Brewfile in your dotfile repository and call `brew bundle`,
  - for python when using pip: you can store a requirements.txt file,
- or to set up your OS settings. In particular, for macOS you can look at this file.

> **Warning:** **Beware**, you should not store confidential information on a remote repository (email, password, …). YADM and tools like such offer some encryption features, use them or do not take any risk.

There exists a lot of tools to manage dotfiles, they all have their own strategy (wrapping git, using symlinks, …). The best thing to do is to try by yourself and check which one fits best your needs, here is a non-exhaustive list.

3.2.3 References

Lists of dotfiles

- An unofficial guide to dotfiles

Other tutorials/introductions

- The missing semester from MIT

3.3 Docker