

APRIL 5, 2022 / #BASH

Command Line for Beginners – How to Use the Terminal Like a Pro [Full Handbook]



German Cocca

```
total 15:53 .
2. Sep 15:53 ..
3. Sep 2015 bin -> usr/bin
19. Sep 09:31 boot
21. Sep 15:58 dev
19. Sep 09:32 etc
21. Sep 15:52 home
7 30. Sep 2015 lib -> usr/lib
34 23. Jul 10:01 lib64 -> usr/lib
96 1. Aug 22:45 lost+found
396 30. Sep 2015 mnt
16 21. Sep 15:52 opt
     0 21. Sep 08:15 private -> /home/encrypted
4896 12. Aug 15:37 proc
560 21. Sep 15:58 root
7 30. Sep 2015 run
4896 30. Sep 2015 sbin -> usr/bin
8 21. Sep 15:51 srv
380 21. Sep 15:45 sys
4896 12. Aug 15:39 usr
4896 23. Jul 16:25 var
root 4896 21. Sep 15:53
root 4896 21. Sep 15:53
```

Hi everyone! In this article we'll take a good look at the command line (also known as the CLI, console, terminal or shell).

The command line is one of the most useful and efficient tools we have as developers and as computer users in general. But using it can feel a bit overwhelming and complex when you're starting out.

In this article I'll try my best to simply explain the parts that make up the command line interface, and the basics of how it works, so you can start using it for your daily tasks.

Let's go! =D

Table of Contents

- [Difference between console, terminal, command line \(CLI\) and Shell](#)
 - [Console](#)
 - [Terminal](#)
 - [Shell](#)
 - [Command line \(CLI\)](#)
- [Why should I even care about using the terminal?](#)
- [Different kinds of shells](#)
 - [A bit of history - Posix](#)
 - [How do I know what shell I'm running?](#)
 - [What shell is better?](#)
 - [A comment about customization](#)
- [Most common and useful commands to use](#)
 - [Git commands](#)
- [Our first script](#)
- [Round up](#)

Difference between console, command line (CLI), terminal and Shell

I think a good place to start is to know exactly what the command line is.

Learn to code — free 3,000-hour curriculum

but the truth is they're actually different things.

Differentiating each isn't necessarily crucial knowledge to have, but it will help clarify things. So let's briefly explain each one.

Console:

The console is the **physical device** that allows you to interact with the computer.

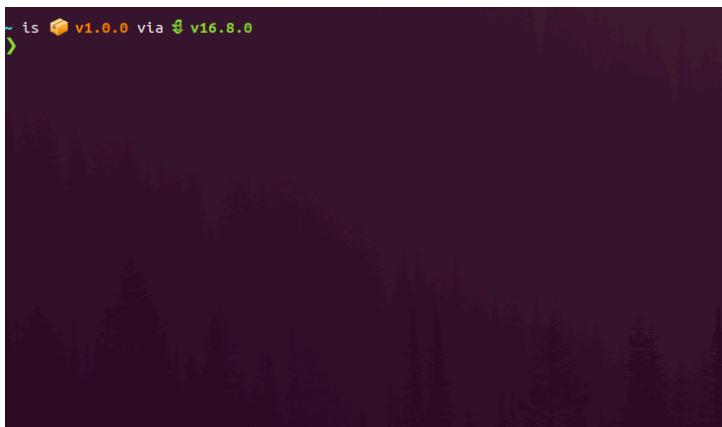
In plain English, it's your computer screen, keyboard, and mouse. As a user, you interact with your computer **through** your console.



Terminal:

A terminal is a text input and output environment. It is a **program** that acts as a **wrapper** and allows us to enter commands that the computer processes.

In plain English again, it's the "window" in which you enter the actual commands your computer will process.



Keep in mind the terminal is a program, just like any other. And like any program, you can install it and uninstall it as you please. It's also possible to have many terminals installed in your computer and run whichever you want whenever you want.

All operating systems come with a default terminal installed, but there are many options out there to choose from, each with its own functionalities and features.

Shell:

A shell is a **program** that acts as command-line interpreter. It **processes commands** and **outputs the results**. It interprets and processes the commands entered by the user.

Same as the terminal, the shell is a program that comes by default in all operating systems, but can also be installed and uninstalled by the user.

Different shells come with different syntax and characteristics as well. It's also possible to have many shells installed at your computer and run each one whenever you want.

In most Linux and Mac operating systems the default shell is Bash. While on Windows it's Powershell. Some other common examples of shells are Zsh and Fish.

Shells work also as **programming languages**, in the sense that with them we can build **scripts** to make our computer execute a certain task. Scripts are nothing more than a series of instructions (commands) that we can save on a file and later on execute whenever we want.

We'll take a look at scripts later on in this article. For now just keep in mind that the shell is the program your computer uses to "understand" and execute your commands, and that you can also use it to program tasks.

Learn to code — free 3,000-hour curriculum

run on any terminal. There's no dependence between both programs in that sense.

Command line or CLI (command line interface):

The CLI is the interface in which we enter commands for the computer to process. In plain English once again, it's the space in which you enter the commands the computer will process.



This is practically the same as the terminal and in my opinion these terms can be used interchangeably.

One interesting thing to mention here is that most operating systems have two different types of interfaces:

- The **CLI**, which takes commands as inputs in order for the computer to execute tasks.
- The other is the **GUI** (graphical user interface), in which the user can see things on the screen and click on them and the computer will respond to those events by executing the corresponding task.

Why should I even care about using the terminal?

We just mentioned that most operating systems come with a GUI. So if we can see things on the screen and click around to do whatever we want, you might wonder why you should learn this complicated terminal/cli/shell thing?

The first reason is that for many tasks, it's just **more efficient**. We'll see some examples in a second, but there are many tasks where a GUI would require many clicks around different windows. But on the CLI these tasks can be executed with a single command.

In this sense, being comfortable with the command line will help you save time and be able to execute your tasks quicker.

The second reason is that by using commands you can easily **automate tasks**. As previously mentioned, we can build scripts with our shell and later on execute those scripts whenever we want. This is incredibly useful when dealing with repetitive tasks that we don't want to do over and over again.

provider for us, or that executes a simpler task like changing our screen wallpaper every hour.

Scripting is a great way to save up time with repetitive tasks.

The third reason is that sometimes the CLI will be the **only way** in which we'll be able to interact with a computer. Take, for example, the case when you would need to interact with a cloud platform server. In most of these cases, you won't have a GUI available, just a CLI to run commands in.

So being comfortable with the CLI will allow you to interact with computers on all occasions.

The last reason is it looks cool and it's fun. You don't see movie hackers clicking around their computers, right? ;)

Different kinds of shells

Before diving into the actual commands you can run in your terminal, I think it's important to recognize the different types of shells out there and how to identify which shell you're currently running.

Different shells come with different syntax and different features, so to know exactly what command to enter, you first need to know what shell you're running.

A bit of history – Posix

For shells, there's a common standard called [Posix](#).

Posix works for shells in a very similar way that ECMAScript works for JavaScript. It's a standard that dictates certain characteristics and features that all shells should comply with.

This standard was established in the 1980's and most current shells were developed according to that standard. That's why most shells share similar syntax and similar features.

How do I know what shell I'm running?

To know what shell you're currently running, just open your terminal and enter `echo $0`. This will print the current running program name, which in this case is the actual shell.



```
~ is 🎉 v1.0.0 via 🐫 v16.8.0
❯ echo $0
bash
```

There's not A LOT of difference between most shells. Since most of them comply with the same standard, you'll find that most of them work similarly.

There are some slight differences you might want to know, though:

- As mentioned, **Bash** is the most widely used and comes installed by default on Mac and Linux.
- **Zsh** is very similar to Bash, but it was created after it and comes with some nice improvements over it. If you'd like to have more detail about its differences, [here's a cool article](#) about it.
- **Fish** is another commonly used shell that comes with some nice built-in features and configurations such as autocompletion and syntax highlighting. The thing about Fish is that it's not Posix complaint, while Bash and Zsh are. This means that some of the commands you'll be able to run on Bash and Zsh won't run on Fish and viceversa. This makes Fish scripting less compatible with most computers compared to Bash and Zsh.
- There are also other shells like **Ash** or **Dash** (the naming just makes everything more confusing, I know...) that are stripped-down versions of Posix shells. This means they only offer the features required in Posix, and nothing else. While Bash and Zsh **add more features** than what Posix requires.

The fact that shells add more features makes them easier and friendlier to interact with, but slower to execute scripts and commands.

So a common practice is to use this "enhanced" shells like Bash or Zsh for general interaction, and a "stripped" shell like Ash or Dash to execute scripts.

When we get to scripting later on, we'll see how we can define what shell will execute a given script.

If you're interested in a more detailed comparison between these shells, [here's a video that explains it really well](#):

If had to recommend a shell, I would recommend bash as it's the most standard and commonly-used one. This means you'll be able to translate your knowledge into most environments.

But again, truth is there's not A LOT of difference between most shells. So in any case you can try a few and see which one you like best. ;)

A comment about customization

I just mentioned that Fish comes with built-in configuration such as autocompletion and syntax highlighting. This come built-in in Fish, but in

[Learn to code — free 3,000-hour curriculum](#)

The point is that shells are customizable. You can edit how the program works, what commands you have available, what information your prompt shows, and more.

We won't see customization options in detail here, but know that when you install a shell in your computer, certain files will be created on your system. Later on you can edit those files to customize your program.

Also, there are many plugins available online that allow you to customize your shell in an easier way. You just install them and get the features that plugin offers. Some examples are [OhMyZsh](#) and [Starship](#).

These customization options are also true for Terminals.

So not only do you have many shell and terminal options to choose from – you also have many configuration options for each shell and terminal.

If you're starting out, all this information can feel a bit overwhelming. But just know that there are many options available, and each option can be customized too. That's it.

Most common and useful commands to use

Now that we have a foundation of how the CLI works, let's dive into the most useful commands you can start to use for your daily tasks.

Keep in mind that these examples will be based on my current configuration (Bash on a Linux OS). But most commands should apply to most configurations anyway.

- **Echo** prints in the terminal whatever parameter we pass it.

```
echo Hello freeCodeCamp! // Output: Hello freeCodeCamp!
```

- **pwd** stands for print working directory and it prints the "place" or directory we are currently at in the computer.

```
pwd // Output: /home/German
```

- **ls** presents you the contents of the directory you're currently in. It will present you with both the files and other directories your current directory contains.

```
ls // Output:  
node_modules  package.json  package-lock.json  public  README.md  src
```

If you pass this command the flag or parameter `-a` It will also show you hidden files or directories. Like `.git` or `.gitignore` files

```
ls -a // Output:  
.  .env  .gitignore  package.json  public  src  
..  .git  node_modules  package-lock.json  README.md
```

- `cd` is short for Change directory and it will take you from your current directory to another.

While on my home directory, I can enter `cd Desktop` and it will take me to the Desktop Directory.

If I want to go up one directory, meaning go to the directory that contains the current directory, I can enter `cd ..`

If you enter `cd` alone, it will take you straight to your home directory.

- `mkdir` stands for make directory and it will create a new directory for you. You have to pass the command the directory name parameter.

If I wanted to create a new directory called "Test" I would enter `mkdir test`.

- `rmdir` stands for Remove directory and it does just that. It needs the directory name parameter just as `mkdir`: `rmdir test`.
- `touch` allows you to create an empty file in your current directory. As parameters it takes the file name, like `touch test.txt`.
- `rm` allows you to delete files, in the same way `rmdir` allows you to remove directories. `rm test.txt`
- `cp` allows you to copy files or directories. This command takes two parameters: the first one is the file or directory you want to copy, and the second one is the destination of your copy (where do you want to copy your file/directory to).

If I want to make a copy of my txt file in the same directory, I can enter the following:

```
cp test.txt testCopy.txt
```

If I wanted to copy the file into a different directory, but keep the same file name, I can enter this:

```
cp test.txt ./testFolder/
```

And if I wanted to copy to a different folder changing the field name, of course I can enter this:

```
cp test.txt ./testFolder/testCopy.txt
```

- **mv** is short for move, and lets us move a file or directory from one place to another. That is, create it in a new directory and delete it in the previous one (same as you could do by cutting and pasting).

Again, this command takes two parameters, the file or directory we want to move and the destination.

```
mv test.txt ./testFolder/
```

We can change the name of the file too in the same command if we want to:

```
mv test.txt ./testFolder/testCopy.txt
```

- **head** allows you to view the beginning of a file or piped data directly from the terminal.

```
head test.txt // Output:  
this is the beginning of my test file
```

- **tail** works the same but it will show you the end of the file.

```
tail test.txt // Output:  
this is the end of my test file
```

- The **--help** flag can be used on most commands and it will return info on how to use that given command.

```
cd --help // output:  
cd: cd [-L|[-P [-e]] [-@]] [dir]
```

Change the current directory to DIR. The default DIR is the value of the ▾
HOME shell variable.

The variable CDPATH defines the search path for the directory containing
DIR. Alternative directory names in CDPATH are separated by a colon : .

A null directory name is the same as the current directory if DIR begins with

....

- In a similar way, the **man** command will return info about any particular command.

```
man cp // output:  
  
CP(1) User Commands CP(1)  
  
NAME  
cp - copy files and directories  
  
SYNOPSIS  
cp [OPTION]... [-T] SOURCE DEST  
cp [OPTION]... SOURCE... DIRECTORY  
cp [OPTION]... -t DIRECTORY SOURCE...  
  
DESCRIPTION  
Copy SOURCE to DEST, or multiple SOURCE(s) to DIRECTORY.  
  
Mandatory arguments to long options are mandatory for short options too.  
  
-a, --archive  
    same as -dR --preserve=all  
  
--attributes-only  
    don't copy the file data, just the attributes  
...
```

You can even enter `man bash` and that will return a huge manual about everything there's to know about this shell. ;)

- **code** will open your default code editor. If you enter the command alone, it just opens the editor with the latest file/directory you opened.

You can also open a given file by passing it as parameter: `code test.txt`.

Or open a new file by passing the new file name: `code thisIsAJsFile.js`.

- **edit** will open text files on your default command line text editor (which if you're on Mac or Linux will likely be either Nano or Vim).

If you open your file and then can't exit your editor, first look at this meme:



And then type `:q!` and hit enter.

The meme is funny because everyone struggles with CLI text editors at first, as most actions (like exiting the editor) are done with keyboard shortcuts. Using these editors is a whole other topic, so go look for tutorials if you're interested in learning more. ;)

- `ctrl+c` allows you to exit the current process the terminal is running. For example, if you're creating a react app with `npx create-react-app` and want to cancel the build at some point, just hit `ctrl+c` and it will stop.
- Copying text from the terminal can be done with `ctrl+shift+c` and pasting can be done with `ctrl+shift+v`
- `clear` will clear your terminal from all previous content.
- `exit` will close your terminal and (this is not a command but it's cool too) `ctrl+alt+t` will open a new terminal for you.
- By pressing **up and down keys** you can navigate through the previous commands you entered.
- By hitting **tab** you will get autocomplete based on the text you've written so far. By hitting **tab twice** you'll get suggestions based on the text you've written so far.

For example if I write `edit test` and **tab twice**, I get `testFolder/test.txt`. If I write `edit test.` and hit **tab** my text autocompletes to `edit test.txt`

Git commands

Learn to code – free 3,000-hour curriculum

you're going to use the terminal for as a developer.

It's a whole lot more efficient to do it from the terminal than by clicking around, so let's take a look at the most useful git commands out there.

- **git init** will create a new local repository for you.

```
git init // output:  
Initialized empty Git repository in /home/German/Desktop/testFolder/.git/
```

- **git add** adds one or more files to staging. You can either detail a specific file to add to staging or add all changed files by typing `git add .`
- **git commit** commits your changes to the repository. Commits must always be accompanied by the `-m` flag and commit message.

```
git commit -m 'This is a test commit' // output:  
[master (root-commit) 6101dfe] This is a test commit  
1 file changed, 0 insertions(+), 0 deletions(-)  
create mode 100644 test.js
```

- **git status** tells you what branch are you currently on and whether you have changes to commit or not.

```
git status // output:  
On branch master  
nothing to commit, working tree clean
```

- **git clone** allows you to clone (copy) a repository into the directory you're currently in. Keep in mind you can clone both remote repositories (in GitHub, GitLab, and so on) and local repositories (those that are stored in your computer).

```
git clone https://github.com/coccagerman/MazeGenerator.git // output:  
Cloning into 'MazeGenerator'...  
remote: Enumerating objects: 15, done.  
remote: Counting objects: 100% (15/15), done.  
remote: Compressing objects: 100% (15/15), done.  
remote: Total 15 (delta 1), reused 11 (delta 0), pack-reused 0  
Unpacking objects: 100% (15/15), done.
```

- **git remote add origin** is used to detail the URL of the remote repository you're going to use for your project. In case you'd like to change it at some point, you can do it by using the command `git remote set-url origin`.

Keep in mind you need to create your remote repo first in order to get its URL. We'll see how you can do this from the command line with a little script later on. ;)

- **git remote -v** lets you list the current remote repository you're using.

```
git remote -v // output:  
origin  https://github.com/coccagerman/testRepo.git (fetch)  
origin  https://github.com/coccagerman/testRepo.git (push)
```

- **git push** uploads your committed changes to your remote repo.

```
git push // output:  
Counting objects: 2, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (2/2), 266 bytes | 266.00 KiB/s, done.  
Total 2 (delta 0), reused 0 (delta 0)
```

- **git branch** lists all the available branches on your repo and tells you what branch you're currently on. If you want to create a new branch, you just have to add the new branch name as parameter like `git branch <branch name>`.

```
git branch // output:  
* main
```

- **git checkout** moves you from one branch to another. It takes your destination branch as parameter.

```
git checkout newBranch // output:  
Switched to branch 'newBranch'
```

- **git pull** pulls (downloads) the code from your remote repository and combines it with your local repo. This is particularly useful when working in teams, when many developers are working on the same code base. In this case each developer periodically pulls from the remote repo in order to work in a code base that includes the changes done by all the other devs.

If there's new code in your remote repo, the command will return the actual files that were modified in the pull. If not, we get `Already up to date.`

- **git diff** allows you to view the differences between the branch you're currently in and another.

```
git diff newBranch // output:
diff --git a/newFileInNewBranch.js b/newFileInNewBranch.js
deleted file mode 100644
index e69de29..0000000
```

As a side comment, when comparing differences between branches or repos, usually visual tools like [Meld](#) are used. It's not that you can't visualize it directly in the terminal, but these tools are great for a clearer visualization.

- **git merge** merges (combines) the branch you're currently in with another. Keep in mind the changes will be incorporated only to the branch you're currently in, not to the other one.

```
git merge newBranch // output:
Updating f15cf51..3a3d62f
Fast-forward
 newFileInNewBranch.js | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newFileInNewBranch.js
```

- **git log** lists all previous commits you've done in the repo.

```
git log // output:
commit 3a3d62fe7cea7c09403c048e971a5172459d0948 (HEAD -> main, tag: TestTag, origin/main, n
Author: German Cocca <german.cocca@avature.net>
Date:   Fri Apr 1 18:48:20 2022 -0300

    Added new file

commit f15cf515dd3ec398210108dce092deb26ff9e12
Author: German Cocca <german.cocca@avature.net>
...
```



- The **--help** flag will show you information about a given command, exactly the same way it works with bash.

```
git diff --help // output:
GIT-DIFF(1)          Git Manual          GIT-DIFF(1)

NAME
    git-diff - Show changes between commits, commit and working tree, etc

SYNOPSIS
    git diff [options] [<commit>] [--] [<path>...]
    git diff [options] --cached [<commit>] [--] [<path>...]
    ...
```

Learn to code — free 3,000-hour curriculum

Now we're ready to get to the truly fun and awesome part of the command line, scripting!

As I mentioned previously, a script is nothing more than a series of commands or instructions that we can execute at any given time. To explain how we can code one, we'll use a simple example that will allow us to create a github repo by running a single command. ;)

- First thing to do is create a `.sh` file. You can put it wherever want. I called mine `newGhRepo.sh`.
- Then open it on your text/code editor of choice.
- On our first line, we'll write the following: `#!/bin/sh`

This is called a **shebang**, and its function is to declare what shell is going to run this script.

Remember previously when we mentioned that we can use a given shell for general interaction and another given shell for executing a script? Well, the shebang is the instruction that dictates what shell runs the script.

As mentioned too, we're using a "stripped down" shell (also known as sh shells) to run the scripts as they're more efficient (though the difference might be unnoticeable to be honest, It's just a personal preference). In my computer I have dash as my sh shell.

If we wanted this script to run with bash the shebang would be `#!/bin/bash`

- Our next line will be `repoName=$1`

Here we're declaring a **variable** called `repoName`, and assigning it to the value of the first parameter the script receives.

A **parameter** is a set of characters that is entered after the script/comand. Like with the `cd` command, we need to specify a directory parameter in order to change directory (ie: `cd testFolder`).

A way we can identify parameters within a script is by using dollar sign and the order in which that parameter is expected.

If I'm expecting more than one parameter I could write:

```
paramOne=$1  
paramTwo=$2  
paramThree=$3  
...
```

Learn to code — free 3,000-hour curriculum

for that so next we're going to code a **conditional** that keeps asking the user to enter the repo name until that parameter is received.

We can do that like this:

```
while [ -z "$repoName" ]
do
    echo 'Provide a repository name'
    read -r -p '$Repository name:' repoName
done
```

What we're doing here is:

1. While the repoName variable is not assigned (`while [-z "$repoName"]`)
 2. Write to the console this message (`echo 'Provide a repository name'`)
 3. Then read whatever input the user provides and assign the input to the repoName variable (`read -r -p '$Repository name:' repoName`)
- Now that we have our repo name in place, we can create our local Git repo like this:

```
echo "# $repoName" >> README.md
git init
git add .
git commit -m "First commit"
```

This is creating a readme file and writing a single line with the repo name (`echo "# $repoName" >> README.md`) and then initializing the git repo and making a first commit.

- Then it's time to upload our repo to github. To do that we're going to take advantage of the [github API](#) in the following command:

```
curl -u coccagerman https://api.github.com/user/repos -d '{"name": "'"$repoName"', "private":false}'
```

`curl` is a command to transfer data from or to a server, using one of the many supported protocols.

Next we're using the `-u` flag to declare the user we're creating the repo for (`-u coccagerman`).

Next comes the endpoint provided by the GitHub API (`https://api.github.com/user/repos`)

[Learn to code — free 3,000-hour curriculum](#)

`repoName` variable) and setting `private` option to `false`, since we want our repo to be public.

Lots of other config options are available in the API, so [check the docs](#) for more info.

- After running this command, GitHub will prompt us to enter our **private token** for authentication.

If you don't have a private token yet, you can generate it in GitHub in [Settings > Developer settings > Personal access tokens](#)

Signed in as

coccagerman



Edit status

Your profile

Your repositories

Your codespaces

Your projects

Your stars

Your gists

Upgrade

Feature preview



Help

Settings

Sign out

 [Emails](#)

 [Password and authentication](#)

 [SSH and GPG keys](#)

 [Organizations](#)

 [Moderation](#)



[Code, planning, and automation](#)

 [Repositories](#)

 [Packages](#)

 [Pages](#)

 [Saved replies](#)

[Security](#)

 [Code security and analysis](#)

[Integrations](#)

 [Applications](#)

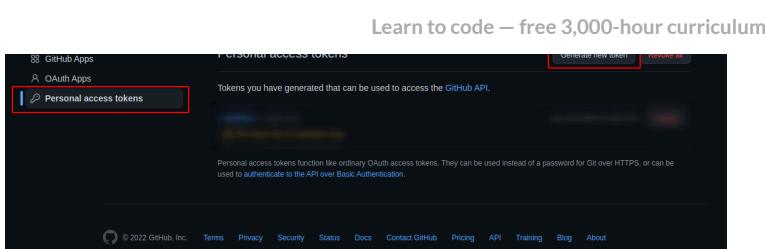
 [Scheduled reminders](#)

[Archives](#)

 [Security log](#)

 [Sponsorship log](#)

 [Developer settings](#)



- Cool, we're almost done now! What we need now is the **remote URL** of our newly created GitHub repo.

To get that we're going to use curl and the GitHub API again, like this:

```
GIT_URL=$(curl -H "Accept: application/vnd.github.v3+json" https://api.github.com/repos/coc
```

Here we're declaring a variable called `GIT_URL` and assigning it to whatever the following command returns.

The `-H` flag sets the header of our request.

Then we pass the GitHub API endpoint, which should contain our user name and repo name

([https://api.github.com/repos/coccagerman/"\\$repoName"](https://api.github.com/repos/coccagerman/$repoName)).

Then we're **piping** the return value of our request. Piping just means passing the return value of a process as the input value of another process. We can do it with the `|` symbol like `<process1> | <process2>`.

And finally we run the `jq` command, which is a tool for processing JSON inputs. Here we tell it to get the value of `.clone_url` which is where our remote git URL will be according to the data format provided by the GitHub API.

- And as last step, we rename our master branch to main, add the remote origin we just obtained, and push our code to GitHub! =D

```
git branch -M main
git remote add origin $GIT_URL
git push -u origin main
```

Our full script should look something like this:

```
#!/bin/sh
repoName=$1

while [ -z "$repoName" ]
do
    echo 'Provide a repository name'
```

```
git init  
git add .  
git commit -m "First commit"  
  
curl -u <yourUserName> https://api.github.com/user/repos -d '{"name": "'"$repoName"'", "p  
GIT_URL=$(curl -H "Accept: application/vnd.github.v3+json" https://api.github.com/repos/<  
  
git branch -M main  
git remote add origin $GIT_URL  
git push -u origin main
```

- Now it's time to test our script! To **execute** it there're two things we can do.

One option is to enter the shell name and pass the file as parameter, like:

```
dash ../ger/code/projects/scripts/newGhRepo.sh .
```

And the other is to make the file **executable** by running

```
chmod u+x  
../ger/code/projects/scripts/newGhRepo.sh .
```

Then you can just execute the file directly by running

```
..../ger/code/projects/scripts/newGhRepo.sh .
```

And that's it! We have our script up and running. Everytime we need a new repo we can just execute this script from whatever directory we're in.

But there's something a bit annoying about this. We need to remember the exact route of the script directory. Wouldn't it be cool to execute the script with a single command that it's always the same independently of what directory we're at?

In come **bash aliases** to solve our problem.

Aliases are a way bash provides for making names for exact commands we want to run.

To create a new alias, we need to edit the bash configuration files in our system. This files are normally located in the home directory. Aliases can be defined in different files (mainly `.bashrc` or `.bash_aliases`).

I have a `.bash_aliases` file on my system, so let's edit that.

- In our CLI we enter `cd` to go over home directory.
- Then we can enter `ls -a` to list all files (includen hidden ones) and check if we have either a `.bashrc` or `.bash_aliases` file in our system.
- We open the file with our text/code editor of choice.
- And we write our new alias like this: `alias newghrepo="dash /home/German/Desktop/ger/code/projects/scripts/newGhRepo.sh"`

And between quotes, define what that alias is going to do ("dash
/home/German/Desktop/ger/code/projects/scripts/newGhRepo.sh")

See that I'm passing the absolute path of the script, so that this command works the same no matter what my current directory is.

If you don't know what the absolute path of your script is, go to the script directory on your terminal and enter `readlink -f newGhRepo.sh`. That should return the full path for you. ;)

- After we're done editing, we save our file, restart our terminal, and voilà! Now we can run our script by just entering `newghrepo`, no matter in what directory we currently are. Much quicker than opening the browser and clicking around to create our repo! =D

I hope this gives you a little taste of the kind of optimizations that are possible with scripting. It certainly requires a bit more work the first time you write, test, and set up the script. But after that, you'll never have to perform that task manually again. ;)

Round up

The terminal can feel like an intimidating and intricate place when you're starting out. But it's certainly worth it to put time and effort into learning the ins and outs of it. The efficiency benefits are too good to pass up!

If you're interested in learning more about the terminal and Bash, Zach Gollwitzer has [an awesome crash course series on youtube](#). He has also great tutorials on other topics such as Node and Javascript, so I recommend that you follow him. ;)

As always, I hope you enjoyed the article and learned something new. If you want, you can also follow me on [linkedin](#) or [twitter](#).

Cheers and see you in the next one! =D

Learn to code – free 3,000-hour curriculum



German Cocca

I'm a full stack developer (typescript | react | react native | node | express) and computer science student. In this blog I write about the things I learn along my path to becoming the best developer I can be.

If this article was helpful, [share it](#).

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. [Get started](#)

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization
(United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can [make a tax-deductible donation here](#).

REST APIs

TypeScript

AI Chatbots

GraphQL APIs

Access Control

PHP

Linux

CI/CD

Golang

Node.js

JavaScript Classes

Express and Node.js

Clustering in Python

Trending Books and Handbooks

Clean Code

JavaScript

Command Line

CSS Transforms

REST API Design

Java

React

Docker

Python

Todo APIs

Front-End Libraries

Python Code Examples

Software Architecture

 Search our news articles, tutorials, and book

[Donate](#)

Learn to code — free 3,000-hour curriculum

[Mobile App](#)



Our Charity

Publication powered by Hashnode [About](#) [Alumni Network](#) [Open Source](#) [Shop](#) [Support](#) [Sponsors](#) [Academic Honesty](#) [Code of Conduct](#) [Privacy Policy](#) [Terms of Service](#)

[Copyright Policy](#)