



Architecture & Deployment

2025-2026 v0.1.0 on branch main Rev: d1f684699a511826485586b2ea383cf021631add

Command Line

Learn what a command line interface is and learn the basics of navigating and manipulating your filesystem in a Unix shell.

Table of contents

- [Presentation](#)
- [Back to the command line](#)
 - [What is a Command Line Interface \(CLI\)?](#)
 - [Why use it?](#)
 - [Open a CLI](#)
 - [Install WSL \(Windows users only\)](#)
- [How to use the CLI](#)
 - [Work in progress...](#)
 - [Writing commands](#)
 - [Options vs. values](#)
 - [Options with values](#)
 - [Naming things when using CLI](#)
 - [Auto-completion](#)
 - [Getting help](#)
 - [Interactive help pages](#)
 - [Unix Command Syntax](#)
- [Using the filesystem](#)
 - [The pwd command](#)
 - [The ls command](#)
 - [The cd command](#)
 - [Absolute paths](#)
 - [The . path](#)
 - [The .. path](#)
 - [Path reference](#)

- [Your projects directory](#)
- [The mkdir command](#)
- [The touch command](#)
- [The echo command](#)
- [The cat command](#)
- [Stopping running commands](#)
- [Windows users](#)
- [Vim](#)
 - [WHY?!](#)
 - [How Vim works](#)
 - [Normal mode](#)
 - [Command mode](#)
- [Nano](#)
 - [An alternative to Vim](#)
 - [Editing files with nano](#)
 - [Saving files](#)
 - [Confirming the filename](#)
 - [Setting nano as the default editor](#)
- [The PATH variable](#)
 - [Understanding the PATH](#)
 - [Finding commands](#)
 - [Using non-system commands](#)
 - [Custom command example](#)
 - [Executing a command in a directory that's not in the PATH](#)
 - [Updating the PATH variable](#)
 - [Does it work?](#)
 - [What have I done?](#)
- [Unleash your terminal](#)
 - [Oh My Zsh](#)
 - [Other tools for the command line lover](#)

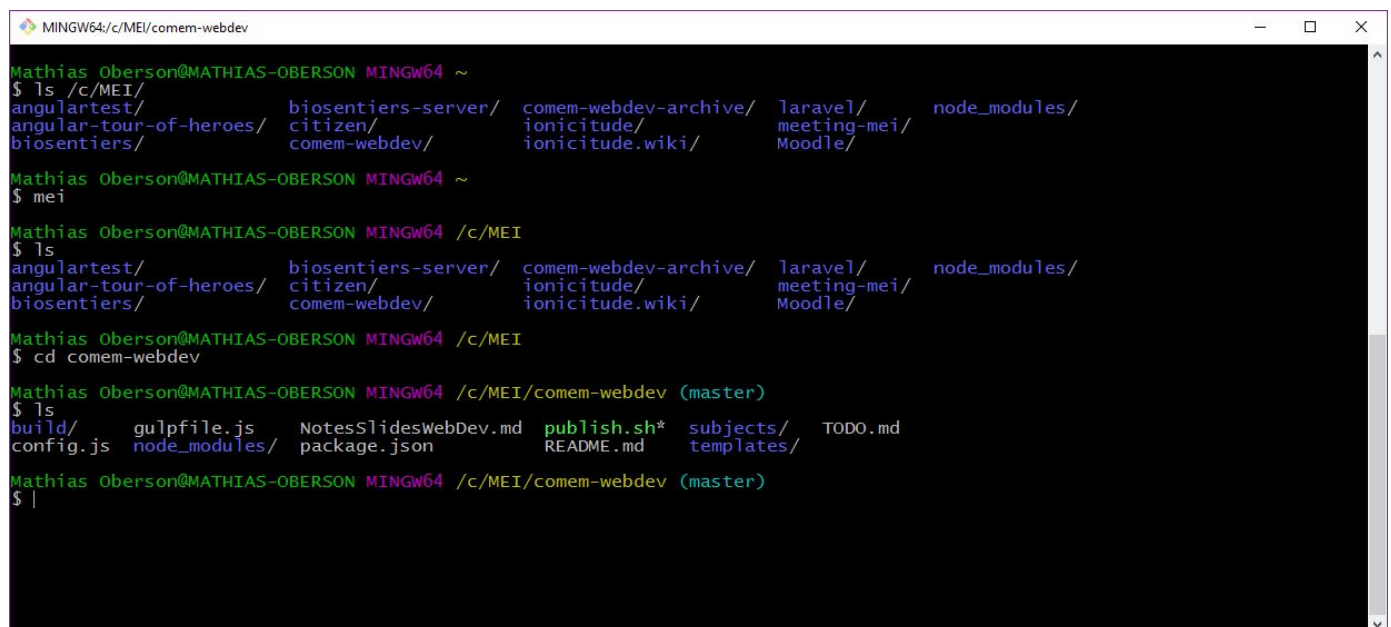
Back to the command line

Command line interfaces are still in wide use today.

What is a Command Line Interface (CLI)?

A CLI is a tool that allows you to use your computer by **writing** what you want to do (i.e. **commands**), instead of clicking on things.

It's been installed on computers for a long time, but it has evolved "a little" since then. It usually looks something like this:



```
MINGW64/c/MEI/comem-webdev
Mathias Oberson@MATHIAS-OBERSON MINGW64 ~
$ ls /c/MEI/
angulartest/      biosentiers-server/  comem-webdev-archive/  laravel/      node_modules/
angular-tour-of-heroes/  citizen/             ionicitude/           meeting-me/
biosentiers/       comem-webdev/        ionicitude.wiki/      Moodle/

Mathias Oberson@MATHIAS-OBERSON MINGW64 ~
$ mei

Mathias Oberson@MATHIAS-OBERSON MINGW64 /c/MEI
$ ls
angulartest/      biosentiers-server/  comem-webdev-archive/  laravel/      node_modules/
angular-tour-of-heroes/  citizen/             ionicitude/           meeting-me/
biosentiers/       comem-webdev/        ionicitude.wiki/      Moodle/

Mathias Oberson@MATHIAS-OBERSON MINGW64 /c/MEI
$ cd comem-webdev

Mathias Oberson@MATHIAS-OBERSON MINGW64 /c/MEI/comem-webdev (master)
$ ls
build/      gulpfile.js  NotesSlidesWebDev.md  publish.sh*  subjects/  TODO.md
config.js   node_modules/ package.json  README.md   templates/

Mathias Oberson@MATHIAS-OBERSON MINGW64 /c/MEI/comem-webdev (master)
$ |
```

Why use it?

A CLI is not very user-friendly or visually appealing but it has several advantages:

- It requires very **few resources** (e.g. memory), which is convenient where resources are scarce (e.g. embedded systems, web servers).
- It can be easily **automated** through scripting.
- Is ultimately **more powerful and efficient** than any GUI for many computing tasks.

For these reasons, a lot of tools, **especially development tools**, don't have any GUI and are only usable through a CLI. Or they have a limited GUI that does not have as many options as the CLI.

Thus, using a CLI is a requirement for any developer today.

Open a CLI

CLIs are available on every operating system.

On **Unix-like** systems (*like macOS or Linux*), it's an application called the **Terminal**.

You can use it right away, as it's the *de-facto* standard.

On **Windows**, the default CLI is called **cmd** (or **Invite de commandes** in French) However, it does not use the same syntax as Unix-like CLIs (*plus, it's bad*).

You also have PowerShell which is better, but is not a Unix-like CLI either.

You'll need to install an alternative.

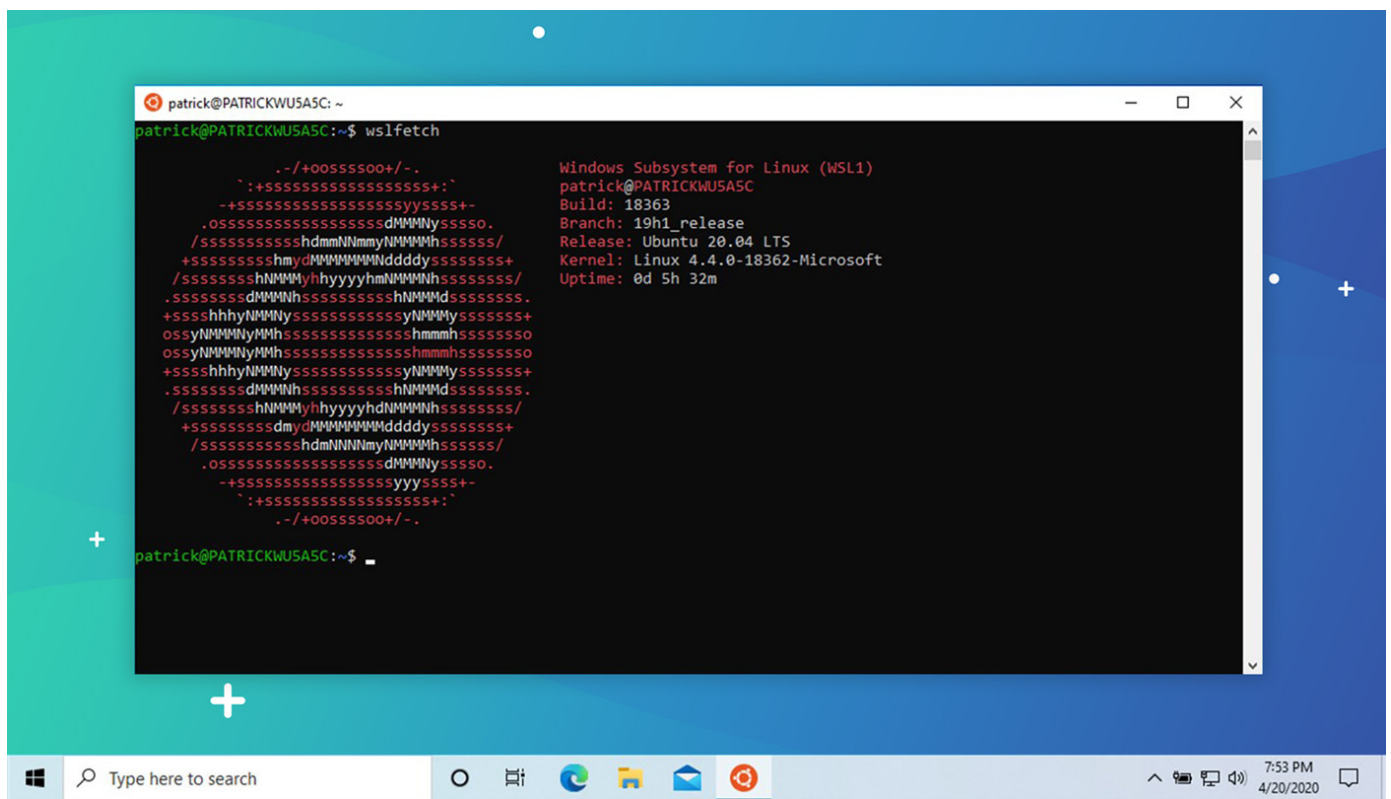


More information

Software terminals are an emulation of old physical terminals like TTYs or the VT100. You will still find references to the term "TTY" in the documentation of some modern command line tools.

Install WSL (Windows users only)

You're going to install the **Windows Subsystem for Linux (WSL)**, a tool that allows you to run a Linux environment on your Windows machine, without using a virtual machine or setting up a dual boot.



Follow [the installation instructions for the WSL](#). The default Linux distribution installed will be Ubuntu, which is perfect for the purposes of this course.

It will ask you for a **username** and a **password**. We suggest you use the same username as for the rest of the course, and that you use the same password as your Windows user account's.

How to use the CLI

When you open the CLI you will find a blank screen that looks like this:

```
$>
```

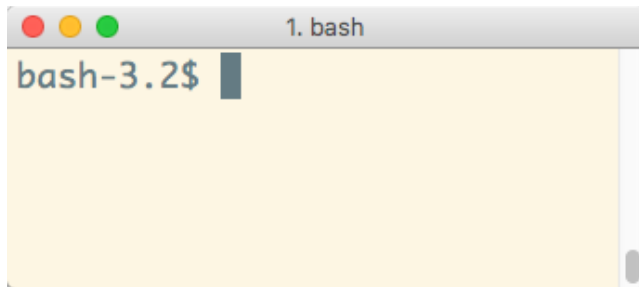
These symbols represent **the prompt** and are used to indicate that you have the lead. **The computer is waiting for you to type something** for it to execute.

Note

The prompt is not always `$>`.

For example, on earlier macOS versions, it used to be `bash3.2$`, indicating the name of the shell (Bash) and its version.

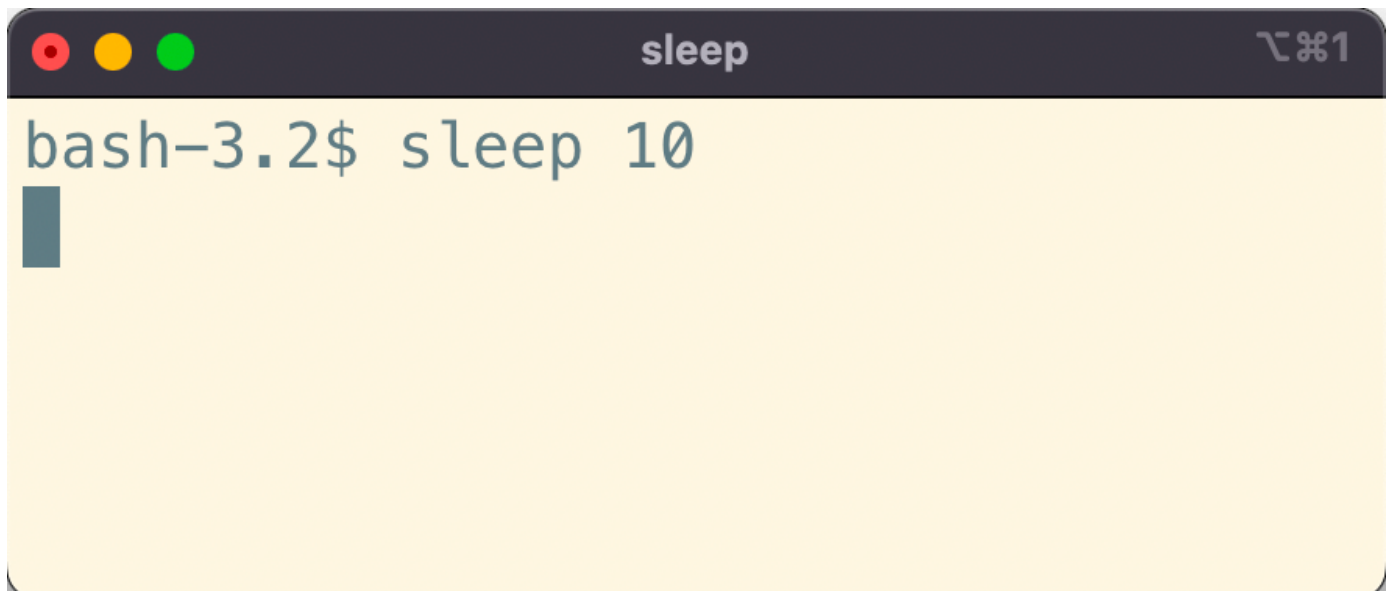
On more recent macOS versions using the Z shell (Zsh), the prompt might indicate your computer's name, your username and the current directory, e.g. `MyComputer:~ root#`.



For consistency, we will always use `$>` to represent the prompt.

Work in progress...

When the computer is working, the prompt disappear and you no longer have control.



When the computer is done working, it will indicate that you are back in control by showing the prompt again.

A terminal window with a dark title bar containing three colored window control buttons (red, yellow, green) on the left, the text 'bash' in the center, and a zoom icon on the right. The terminal area has a light yellow background. It shows two lines of text: 'bash-3.2\$ sleep 10' and 'bash-3.2\$' followed by a dark blue cursor block.

```
bash-3.2$ sleep 10
bash-3.2$
```

 Note

The `sleep` command tells the computer to do nothing for the specified number of seconds.

Writing commands

A command is a **word** that you have to type in the CLI that will **tell the computer what to do**.

The syntax for using commands looks like this:

```
$> name arg1 arg2 arg3 ...
```

Note the use of **spaces** to separate the different **arguments** of a command.

- `name` represents the **command** you want to execute.
- `arg1 arg2 arg3 ...` represent the **arguments of the command**, each of them **separated by a space**.

Options vs. values

There are two types of arguments to use with a command (if needed):

Options usually specify **how** the command will behave. By convention, they are preceded by `-` or `--`:

```
$> ls -a -l
```

We use the `ls` command to **list** the content of the current directory. The options tell `ls` **how** it should do so:

- `-a` tells it to print **all** elements (including hidden ones).
- `-l` tells it to print elements in a **list** format, rather than on one line.

Values not preceded by anything usually specify **what** will be used by the command:

```
$> cd /Users/Batman
```

Here, we use the `cd` command to move to another directory (or **change directory**).

And the argument `/Users/Batman` tells the command **what** directory we want to move to.

Options with values

Values can also be linked to an option:

```
$> tar -c -v -f compressed.tar.gz file-to-compress
```

The `tar` (tape archive) command bundles and compresses files. In this example, it takes **three options**:

- `-c` tells it to **compress** (instead of uncompressing).
- `-v` tells it to be **verbose** (print more information to the CLI).
- `-f` tells it where to store the compressed **file**; this is followed **immediately** by `compressed.tar.gz` which is the **value** of that option.

It then takes **one value**:

- `file-to-compress` is the file (or directory) to compress

There are two values in this example: one linked to the `-f` option, and one used by the overall command.

Naming things when using CLI

You should avoid the following characters in directories and file names you want to manipulate with the CLI:

- **spaces** (*they're used to separate arguments in command*).
- **accents** (e.g. `é`, `à`, `ç`, etc).

They can cause **errors** in some scripts or tools, and will inevitably complicate using the CLI. If you have a `Why So Serious` directory, this **WILL NOT work**:

```
$> ls Why So Serious
```

This command will be interpreted as a call to the `ls` command with **three arguments**: `Why`, `So` and `Serious`.

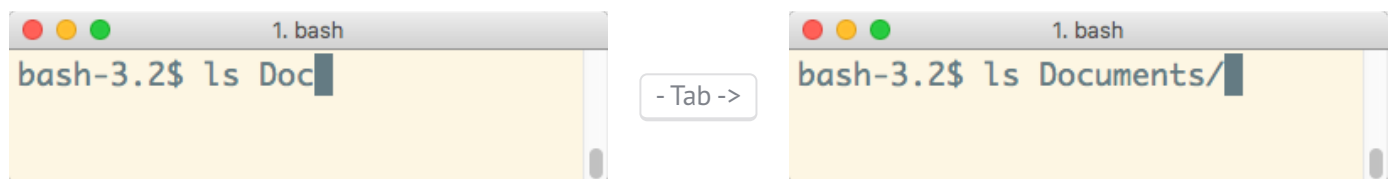
You **can** use arguments containing spaces, but you have to **escape** them first, either with **quotation marks** or **backslashes**:

```
$> ls "Why So Serious"
```

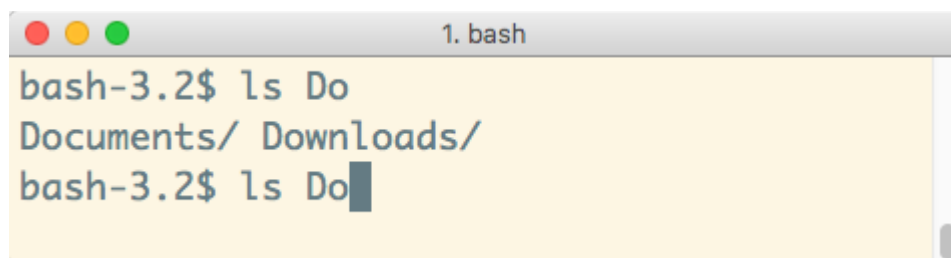
```
$> ls Why\ So\ Serious
```

Auto-completion

It's not fun to type directory names, especially when they have spaces you must escape in them, so the CLI has **auto-completion**. Type the first few characters of the file or directory you need, then hit the **Tab** key:



If there are multiple files or directories that begin with the **same characters**, pressing **Tab** will not display anything. You need hit **Tab** **a second time** to display the list of available choices:



You can type just enough characters so that the CLI can determine which one you want (in this case **c** or **w**), then hit **Tab** again to get the full path.

Getting help

You can get help on most advanced commands by executing them with the **--help** option. As the option's name implies, it's designed to **give you some help** on how to use the command:

```
1. bash
bash-3.2$ tar --help
tar(bsdtr): manipulate archive files
First option must be a mode specifier:
  -c Create  -r Add/Replace  -t List  -u Update  -x Extract
Common Options:
  -b #       Use # 512-byte records per I/O block
  -f <filename> Location of archive
  -v         Verbose
```

Some commands don't have the `--help` option, but there are alternative sources of information depending on what operating system you're on:

- On Linux or macOS, use `man ls` to display the **manual** for the `ls` command.
- On Windows, use `help cd` to display help for the `cd` command; you can also type `help` to list available commands (only system commands).
- If you have Node.js and npm installed, there is also tldr pages: a cross-platform tool that provides simplified and community-driven manual pages.

Interactive help pages

Some help pages or commands will take over the screen to display their content, hiding the prompt and previous interactions.

Usually, it means that there is content that takes more than one screen to be shown. You can “scroll” up and down line-by-line using the arrow keys or the `Enter` key.

To quit these interactive documentations, use the `q` (**quit**) key.

```
1. sh

LS(1)                                BSD General Commands Manual                                LS(1)

NAME
    ls -- list directory contents

SYNOPSIS
    ls [-ABCFGHLOPRSTUW@abcdefghijklmnopqrstuvwxyz1] [file ...]

DESCRIPTION
    For each operand that names a file of a type other than directory, ls
    displays its name as well as any requested, associated information.  For
    :
```

Unix Command Syntax

When reading a command's manual or documentation, you may find some strange syntax that make little sense to you, like:

```
cd [-L|[-P [-e]] [-@]] [dir]
ls [-ABCFGHLOPRSTUW@abcdefghijklmnopqrstuvwxyz1] [file ...]
```

Here are some explanations:

- `[]`: Whatever's inside is **optional** (ex: `[-e]`).
- `|`: You have to **choose between** options (ex: `-L|-P`).
- `...`: Whatever's before can be **repeated** (ex: `[file ...]`).

Depending on the documentation, you will also see symbols like this:

- `<value>`
- `--option=VALUE`

DON'T WRITE `<value>` or `VALUE`. Replace it by an appropriate value for that option or argument.

Using the filesystem

The `pwd` command

When you open a CLI, it places you in your **home directory**. From there you can navigate your filesystem to go to other directories (*more on that later*).

But first, you might want to check **where** you currently are. Use the `pwd` command:

```
$> pwd
/Users/Batman
```

Note

`pwd` means “**p**rint **w**orking **d**irectory”: it gives you the absolute path to the directory you’re currently in.

The `ls` command

Now that you know where you are, you might want to know **what your current directory is containing**.

Use the `ls` command:

```
$> ls
(lots and lots of files)
```

Note

`ls` means “**l**ist”: it lists the contents of a directory.

By default, `ls` doesn’t list **hidden elements**. By convention in Unix-like systems, files that start with `.` (a dot) are hidden.

If you want it to do that, you need to pass the `-a` (**a**ll) option:

```
$> ls -a
```

```
(lots and lots of files, including the hidden ones)
```

The `cd` command

It's time to go out a little and move to another directory.

Suppose you have a `Documents` directory in your home directory, that contains another directory `TopSecret` where you want to go. Use the `cd` (**c**hange **d**irectory) command, passing it as argument **the path to the directory** you want to go to:

```
$> pwd
```

```
/Users/Batman
```

```
$> cd Documents/TopSecret
```

```
$> pwd
```

```
/Users/Batman/Documents/TopSecret
```

This is a **relative path**: it is relative to the current working directory.

Absolute paths

You can also go to a specific directory anywhere on your filesystem like this:

```
$> cd /Users/Batman/Documents
```

```
$> pwd
```

```
/Users/Batman/Documents
```

This is an **absolute path** because it starts with a `/` character. It starts at the root of your filesystem so it does not matter where you are now.

Tip

You also have **auto-completion** with the `cd` command. Hit the `Tab` key after entering some letters.

The `.` path

The `.` path represents the current directory. The following commands are strictly equivalent:

```
$> cd Documents/TopSecret
```

```
$> cd ./Documents/TopSecret
```

You can also *not go anywhere*:

```
$> pwd
/Users/Batman
```

```
$> cd .
```

```
$> pwd
/Users/Batman
```

Or compress the current directory:

```
tar -c -v -f /somewhere/compressed.tar.gz .
```

This does not seem very useful now, but it will be in further tutorials.

The `..` path

To go up into the parent directory, use the `..` path (**don't forget the space between `cd` and `..`**):

```
$> pwd
/Users/Batman/Documents
```

```
$> cd ..
```

```
$> pwd
/Users/Batman
```

You can also drag and drop a directory from your Explorer or your Finder to the CLI to see its absolute path automatically written:

```
$> cd
(Drag and drop a directory from your Explorer/Finder, and...)
$> cd /Users/Batman/Pictures/
```

At any time and from anywhere, you can return to your **home directory** with the `cd` command, without any argument or with a `~` (tilde):

```
$> cd
$> pwd
/Users/Batman
```

```
$> cd ~
$> pwd
/Users/Batman
```



To type the `~` character, use this combination:

- `AltGr-^` on **Windows**
- `Alt-N` on **Mac**

Path reference

Path	Where
<code>.</code>	The current directory.
<code>..</code>	The parent directory.
<code>foo/bar</code>	The file/directory <code>bar</code> inside the directory <code>foo</code> in the current directory. This is a relative path .
<code>./foo/bar</code>	<i>Same as the above</i>
<code>/foo/bar</code>	The file/directory <code>bar</code> inside the directory <code>foo</code> at the root of your filesystem. This is an absolute path .
<code>~</code>	Your home directory. This is an absolute path .
<code>~/foo/bar</code>	The file/directory <code>bar</code> inside the directory <code>foo</code> in your home directory. This is an absolute path .

Your projects directory

Throughout this course, you will often see the following command (*or something resembling it*):

```
$> cd /path/to/projects
```

This means that you use **the path to the directory in which you store your projects**. For example, on John Doe's macOS system, it could be `/Users/jde/Projects`.

Warning

Do not actually write `/path/to/projects`. It will obviously fail, unless you happen to have a `path` directory that contains a `to` directory that contains a `projects` directory...

⚠ **Windows users:** if your username contains **spaces** or **accents**, you should **NOT** store your projects under your home directory. You should find a path elsewhere on your filesystem. This will save you **a lot of needless pain and suffering**.

The `mkdir` command

You can create directories with the CLI.

Use the `mkdir` (**make directory**) command to create a new directory in the current directory:

```
$> mkdir BatmobileSchematics
$> ls
BatmobileSchematics
```

You can also create a directory elsewhere:

```
$> mkdir ~/Documents/TopSecret/BatmobileSchematics
```

This will only work if all directories down to `TopSecret` already exist. To automatically create all intermediate directories, add the `-p` (**p**arents) option:

```
$> mkdir -p ~/Documents/TopSecret/BatmobileSchematics
```

The `touch` command

The `touch` command updates the last modification date of a file. It also has the useful property of creating the file if it doesn't exist.

Hence, it's a **quick way to create an empty file** in the CLI:

```
$> touch foo.txt
```

```
$> ls  
foo.txt
```

The `echo` command

The `echo` command simply **echoes** its arguments back to you:

```
$> echo Hello World  
Hello World
```

This seems useless, but can be quite powerful when combined with Unix features like redirection. For example, you can **redirect the output to a file**.

The `>` operator means “***write** the output of the previous command into a file*”. This allows you to quickly create a simple text file:

```
$> echo foo > bar.txt
```

```
$> ls  
bar.txt
```

If the file already exists, it is overwritten. You can also use the `>>` operator, which means “***append** the output of the previous command to the end of a file*”:

```
$> echo bar >> bar.txt
```

The `cat` command

The `cat` command can display one file or concatenate multiple files in the CLI. For example, this displays the contents of the previous example's file:

```
$> cat bar.txt
foo
bar
```

This creates a new `hello.txt` file and displays the result of concatenating the two files:

```
$> echo World > hello.txt

$> cat bar.txt hello.txt
foo
bar
World
```

Stopping running commands

Sometimes a command will take too long to execute.

As an example, run this command which will wait one hour before exiting:

```
$> sleep 3600
```

As you can see, the command keeps executing and you **no longer have a prompt**. Anything you type is ignored, as it is no longer interpreted by the shell, but by the `sleep`

command instead (which doesn't do anything with it).

By convention in Unix shells, you can always terminate a running command by typing `Ctrl-C` (press the **C** key while holding the **Control** key).

Warning

Note that `Ctrl-C` **forces termination** of a running command. It might not have finished what it was doing.

Windows users

This is how you reference or use your **drives** (`C:`, `D:`, etc) in the Windows Subsystem for Linux (WSL):

```
$> cd /mnt/c/foo/bar
```

```
$> cd /mnt/d/foo
```

Note

If you are using Git Bash, it's `/c` instead of `/mnt/c`.

Copy/Paste

Since `Ctrl-C` is used to stop the current process, it **can't** be used as a shortcut to copy things from the CLI. Instead, the **Windows Subsystem for Linux** (WSL) has two custom shortcuts:

- `Shift-Ctrl-C` to **copy** things from the CLI
- `Shift-Ctrl-P` to **paste** things to the CLI

Vim

Vim is an infamous CLI editor originally developed in 1976 (WHAT?!) for the Unix operating system.

Note

The name comes from “**vi** improved”, because Vim is an improved clone of an earlier editor: vi (from **vi**sual).

WHY?!

Why would you need to learn it?

Sometimes it's just the **only editor you have** (e.g. on a server). Also **some developer tools might open Vim** for user input.

If this happens (*and it will*), there's **one** imperative rule to follow:

DO NOT PANIC!

Open a file by running the `vim` command with the path to the file you want to create/edit:

```
vim test.txt
```

How Vim works

Vim can be unsettling at first, until you know how it works.

Let go of your fear. And your mouse, it's mostly useless in Vim. You control Vim by **typing**.

The first thing to understand which Vim is that it has *3 modes*:

- **Normal** mode (the one you're in when Vim starts).
- **Command** mode (the one to use to save and/or quit).
- **Insert** mode (the one to use to insert text).

To go into each mode use this keys:

From	Type	To go to
Normal		Command
Normal		Insert
Command/Insert		Normal

Normal mode

The **Normal** mode of Vim is the one you're in when it starts. In this mode, you can move the cursor around with the arrow keys.

You can also use some commands to interact with the text:

Command	Effect
	Delete the character under the cursor
	Delete a word, with the cursor standing before the first letter
	Delete the complete line the cursor is on
	Undo the last command
	Enter Command mode (to save and/or quit)
	Enter Insert mode (to type text)

At anytime, you can hit the `Esc` key to go back to the **Normal** mode.

Command mode

The **Command** mode, which you can only access from the **Normal** mode, is the one you’ll mostly use to save and/or quit.

To enter the **Command** mode, hit the `:` key. From there, you can use some commands:

Command	Effect
<code>q</code>	Quit Vim (will fail if you have unsaved modifications)
<code>w</code>	Write (save) the file and all its modifications
<code>q!</code>	Force (!) Vim to quit (any unsaved modification will be lost)
<code>wq</code> or <code>x</code>	Write and quit , i.e. save the file then quit Vim.

Nano



“Nano: a simpler CLI editor to keep your sanity.”

An alternative to Vim

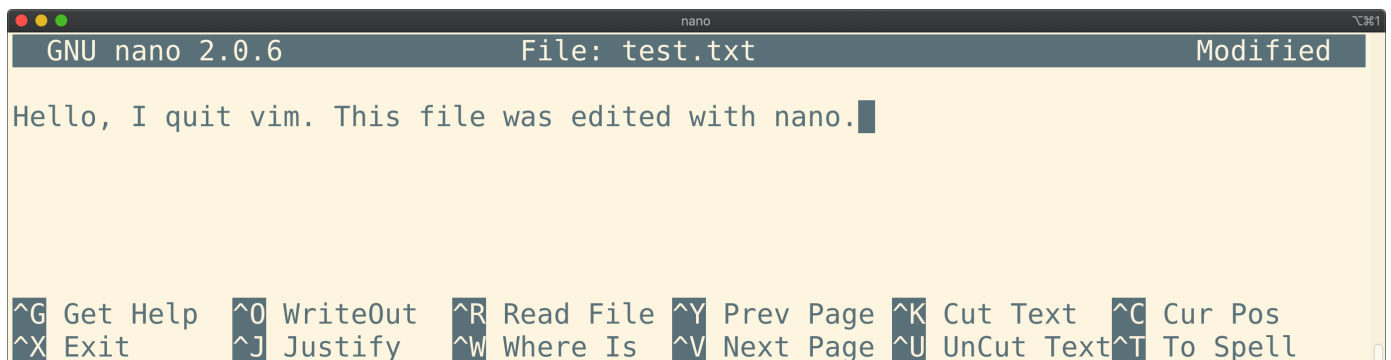
If Vim is a bit too much for you, nano is another CLI editor that is much simpler to use and is also usually installed on most Unix-like systems (and in Git Bash).

You can open a file with nano in much the same way as Vim, using the `nano` command instead:

```
$> nano test.txt
```

Editing files with nano

Editing files is much more straightforward and intuitive with nano. Once the file is open, you can simply type your text and move around with arrow keys:



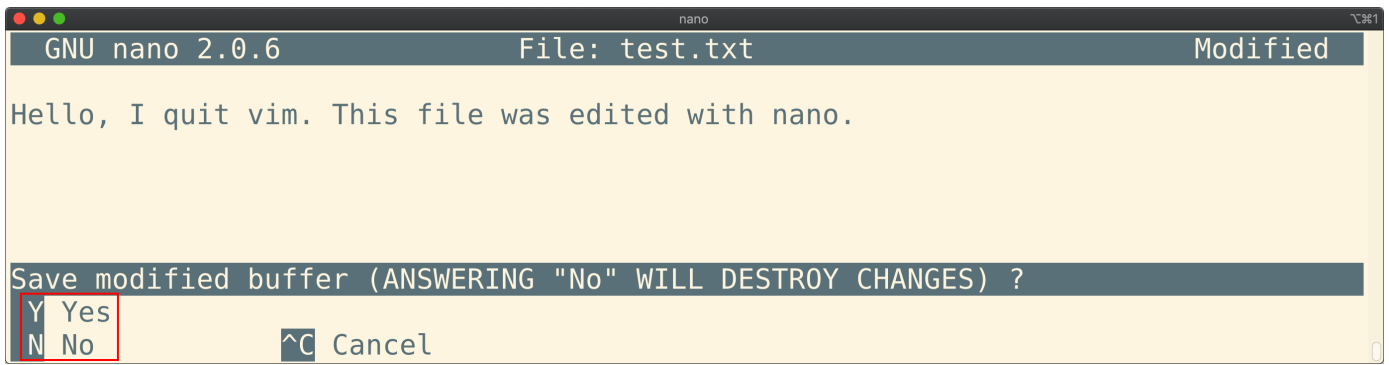
Tip

Nano also helpfully prints its main keyboard shortcuts at the bottom of the window. The most important one is `^X` for Exit. In keyboard shortcut parlance, the `^` symbol always represents the control key.

So, to exit from nano, type `Ctrl-X`.

Saving files

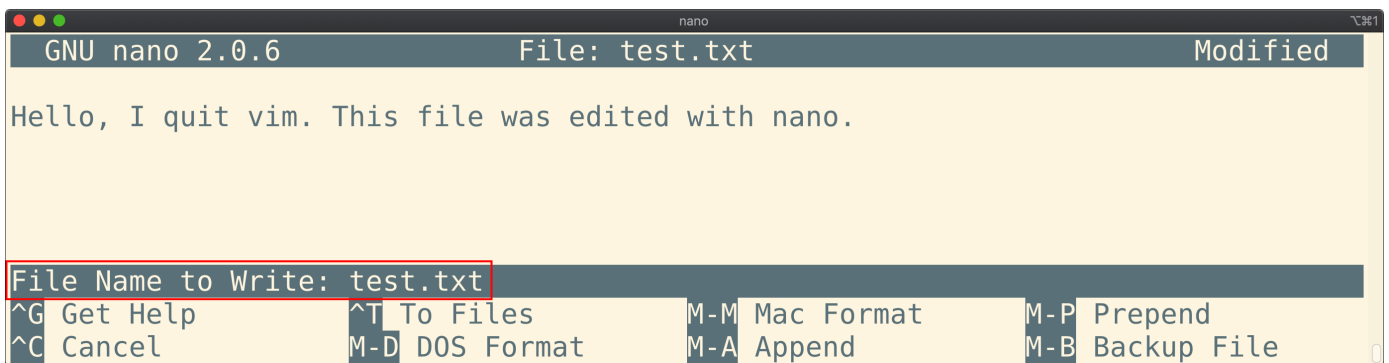
When you exit nano with `Ctrl-X`, it will ask you whether you want to save your changes:

A screenshot of the nano 2.0.6 text editor. The title bar shows 'GNU nano 2.0.6', 'File: test.txt', and 'Modified'. The main text area contains the line 'Hello, I quit vim. This file was edited with nano.' At the bottom, a dark status bar displays the prompt 'Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES) ?'. Below this, a light-colored box contains the options 'Y Yes' and 'N No', with the 'Y' key highlighted by a red rectangle. To the right of this box is the '^C Cancel' option.

Press the **y** key to save or the **n** key to discard your changes.

Confirming the filename

When saving changes, nano will always ask you to **confirm the filename** where the changes should be saved:

A screenshot of the nano 2.0.6 text editor. The title bar shows 'GNU nano 2.0.6', 'File: test.txt', and 'Modified'. The main text area contains the line 'Hello, I quit vim. This file was edited with nano.' At the bottom, a dark status bar displays the prompt 'File Name to Write: test.txt', which is highlighted by a red rectangle. Below this, a light-colored box contains a list of keyboard shortcuts: '^G Get Help', '^T To Files', 'M-M Mac Format', 'M-P Prepend', '^C Cancel', 'M-D DOS Format', 'M-A Append', and 'M-B Backup File'.

As you can see, it tells you the name of the file you opened. Now you can:

- Simply press **Enter** to save the file.
- Or, change the name to save your changes to another file (and keep the unmodified original).

Setting nano as the default editor

Editing the shell configuration will depend on your shell: for Zsh (the default terminal shell on macOS) or Bash shell (the default in Git Bash and most Linux systems), you have to set the **\$EDITOR** environment variable. You can do that by adding the following line to your **~/.zshrc** or **~/.bash_profile** file depending on which shell you are using:

```
export EDITOR=nano
```

Remember that you must **relaunch your terminal** for this change to take effect.

If you are unsure of what shell you are using, type in the following command. The output will display the name of your current shell.

```
$> echo $0  
bash
```

Tip

Now that you know how to use nano, you can edit your Bash profile file with the following command: `nano ~/.bash_profile`.

Note

On Ubuntu, you can list available editors and choose the default one with the following command:

```
$> sudo update-alternatives --config editor
```

The `PATH` variable

When you type a command in the CLI, it will try to see **if it knows this command** by looking in some directories to see if there is an **executable file that matches the command name**.

```
$> rubbish  
bash: rubbish: command not found
```

This means that the CLI failed to find the executable named `rubbish` in any of the directories where it looked.

The list of the directories (and their paths) in which the CLI searches is stored in the `PATH` environment variable, each of them being separated with a `:`.

You can print the content of your `PATH` variable to see this list:

```
$> echo $PATH
/usr/local/bin:/bin:/usr/bin:/custom/dir
```

Understanding the `PATH`

Assuming your `PATH` looks like this:

```
$> echo $PATH
/usr/local/bin:/bin:/usr/bin:/custom/dir
```

What happens when you run the following command?

```
$> ls -a -l
```

1. The shell will look in the `/usr/local/bin` directory. *There is no executable named `ls` there, moving on...*
2. The shell will look in the `/bin` directory. **There is an executable named `ls` there!** Execute it with arguments `-a` and `-l`.
3. We're done here. No need to look at the rest of the `PATH`. If there happens to be an `ls` executable in the `/custom/dir` directory, it will **not be used**.

Finding commands

You can check where a command is with the `which` command:

```
$> which ls  
/bin/ls
```

If there are multiple versions of a command in your `PATH`, you can add the `-a` option to list them **all**:

```
$> which -a git  
/opt/homebrew/bin/git  
/usr/bin/git
```

Note

Remember, the shell will use the first one it finds, so in this example it would use `/opt/homebrew/bin/git` if you type `git`, completely ignoring `/usr/bin/git`.

Using non-system commands

Many development tools you install come with executables that you can run from the CLI (e.g. Git, Node.js, MongoDB).

Some of these tools will install their executable in a **standard directory** like `/usr/local/bin`, which is already in your `PATH`. Once you've installed them, you can simply run their new commands. Git and Node.js, for example, do this.

However, sometimes you're downloading only an executable and saving it in a directory somewhere that is **not in the** `PATH`.

Custom command example

Run the following commands to download a simple Hello World shell script and make it into an executable:

```
$> mkdir -p ~/hello-program/bin
$> curl -o ~/hello-program/bin/hello https://gist.githubusercontent.com/AlphaHydr
$> chmod 755 ~/hello-program/bin/hello
```

Note

The `curl` command is used to download the script file, and the `chmod` command to make that file executable.

You should now be able to find it in the `~/hello-program/bin` directory:

```
$> ls ~/hello-program/bin
hello
```

It's now installed, we can find it using the CLI, but it still cannot be run. Why?

```
$> hello
command not found: hello
```

Executing a command in a directory that's not in the `PATH`

You can run a command from anywhere by writing **the absolute path to the executable**:

```
$> ~/hello-program/bin/hello
Hello World
```

You can also **manually go to the directory** containing the executable and **run the command there**:

```
$> cd ~/hello-program/bin
$> ./hello
Hello World
```

Tip

When the first word on the CLI starts with `/`, `~/`, `./` or `../`, the shell interprets it as a file path. Instead of looking for a command in the `PATH`, it simply executes that file.

But, ideally, you want to be able to **just type** `hello`, and have the script be executed. For this, you need to **add the directory containing the executable** to your `PATH` variable.

Updating the `PATH` variable

To add a new path in your `PATH` variable, you have to edit a special file, used by your CLI interpreter (shell). This file depends upon the shell you are using:

CLI	File to edit
Git Bash	<code>~/.bash_profile</code>
Terminal / <u>Z</u> sh	<code>~/.zshrc</code>

Open the adequate file (`.bash_profile` for this example) from the CLI with `nano` or your favorite editor if it can display hidden files:

```
$> nano ~/.bash_profile
```

Add this line at the bottom of your file (use `i` to enter **insert** mode if using Vim):

```
export PATH="$HOME/hello-program/bin:$PATH"
```

If you're in Vim, press `Esc` when you're done typing, then `:wq` and `Enter` to save and quit. If you're in nano, press `Ctrl-X`, then answer `Yes` and confirm the filename.

Does it work?

Warning

Remember to **close and re-open your CLI** to have the shell reload its configuration file.

You should now be able to run the Hello World shell script as a command simply by typing `hello`:

```
$> hello
Hello World
```

You don't even have to be in the correct directory:

```
$> cd
$> pwd
/Users/jde
$> hello
Hello World
```

And your CLI knows where it is:

```
$> which hello
/Users/jde/hello-program/bin/hello
```

It knows this because the directory containing the script is now in your PATH:

```
$> echo $PATH
/Users/jde/hello-program/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin
```

What have I done?

You have **added a directory to the** `PATH`:

```
export PATH=~/.hello-program/bin:$PATH
```

This line says:

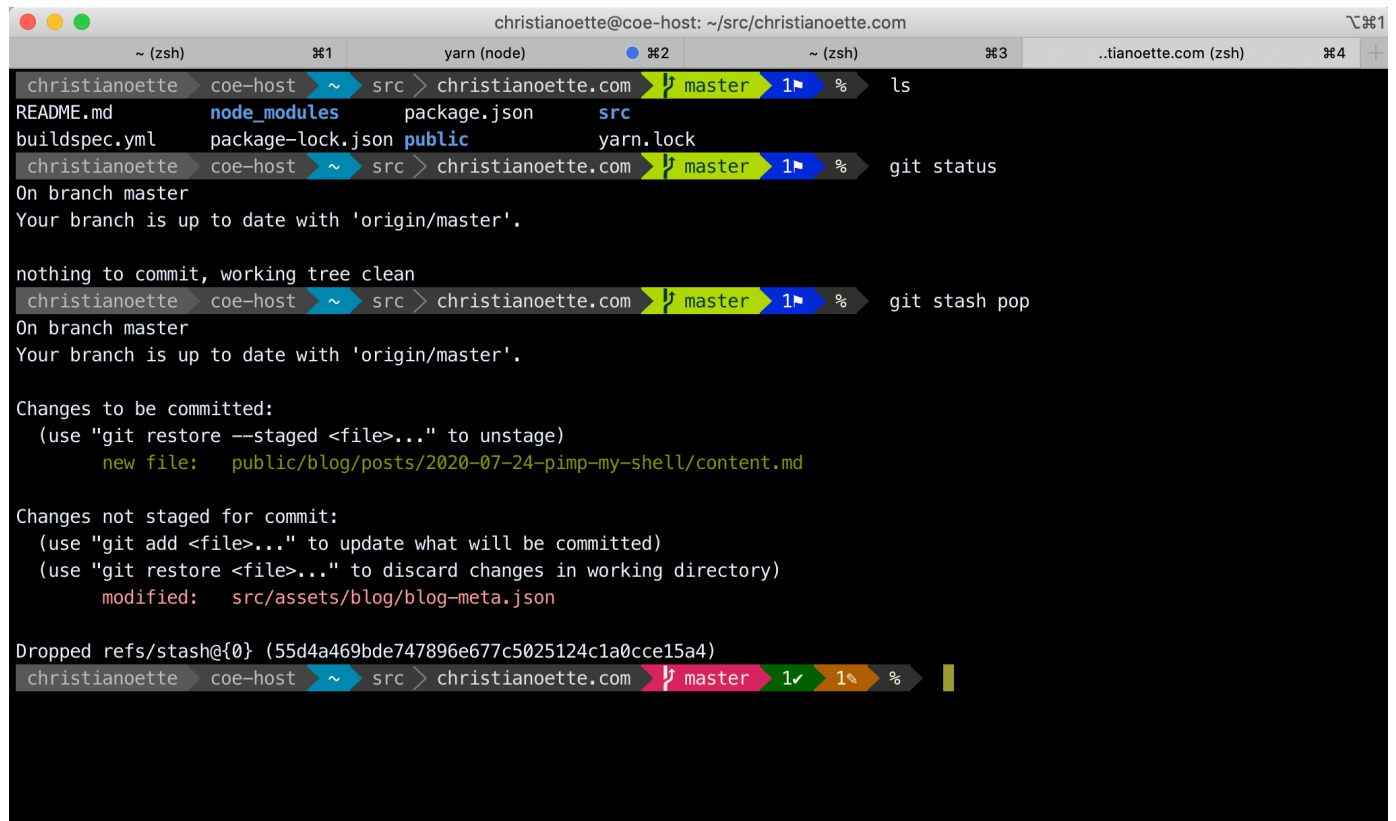
- Modify the `PATH` variable.
- In it, put the new directory `~/hello-program/bin` and the previous value of the `PATH`, separated by `:`.

The next time you run a command, your shell will **first look** in this directory for executables, then in the **rest of the** `PATH`.

Common mistakes

- What you must put in the `PATH` is **NOT** the path to the executable, but the path to the **directory containing the executable**.
- You must re-open your CLI for the change to take effect: the shell configuration file (e.g. `~/.bash_profile`) is only applied when the shell starts.

Unleash your terminal



```
christianoette@coe-host: ~/src/christianoette.com
~ (zsh) %1 yarn (node) %2 ~ (zsh) %3 ..tianoette.com (zsh) %4 +
christianoette coe-host ~ src > christianoette.com master 1▶ % ls
README.md node_modules package.json src
buildspec.yml package-lock.json public yarn.lock
christianoette coe-host ~ src > christianoette.com master 1▶ % git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
christianoette coe-host ~ src > christianoette.com master 1▶ % git stash pop
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   public/blog/posts/2020-07-24-pimp-my-shell/content.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   src/assets/blog/blog-meta.json

Dropped refs/stash@{0} (55d4a469bde747896e677c5025124c1a0cce15a4)
christianoette coe-host ~ src > christianoette.com master 1✓ 1▶ %
```

Oh My Zsh

Command-line shells have been worked on for a very long time. Modern shells such as the Z shell (Zsh) have a whole community that created many plugins to simplify your daily command-line work.

On macOS or Linux, you may want to install Oh My Zsh to fully unleash the power of your Terminal. It has plugins to integrate with Homebrew, Git, various programming languages like Ruby, PHP, Go, and much more.

On Windows, you may want to install the Windows Subsystem for Linux so you can install a Linux distribution like Ubuntu. You can then install Zsh and Oh My Zsh as well.

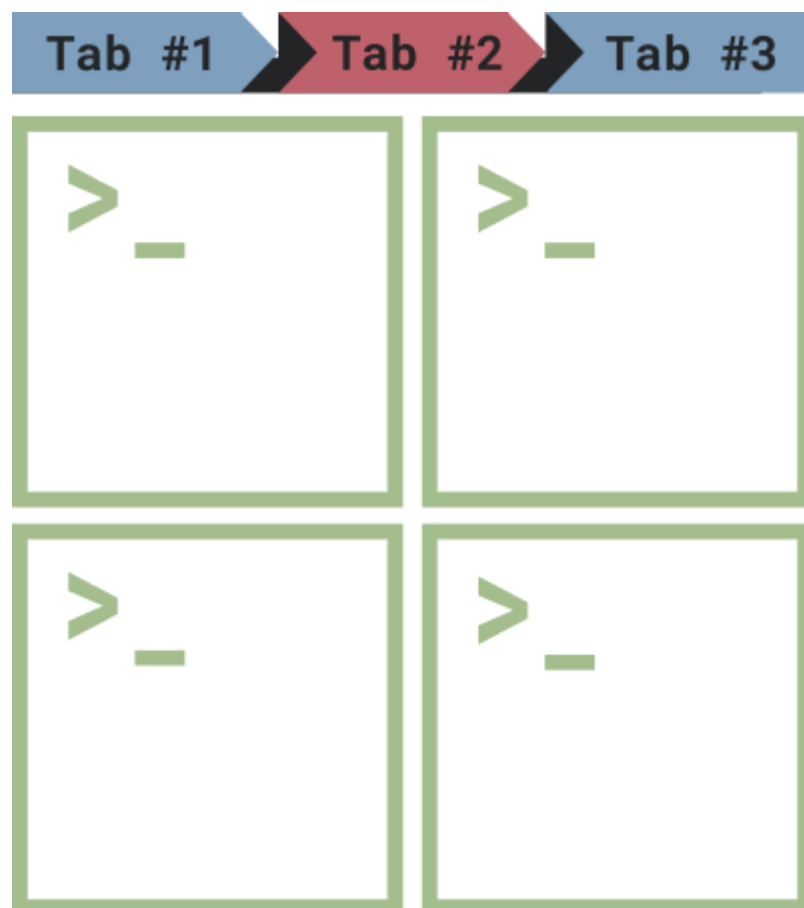
Other tools for the command line lover

- bat: a cat clone with wings

- `tldr`: better `man` pages
- `fzf` to quickly find files
- `ack` to search for text in files (`grep` alternative)
- `rsync` for incremental file transfers (`cp` & `scp` alternative)

A Terminal multiplexer like:

- `tmux`
- `screen`
- `zellij` ❤️



↑ Back to top