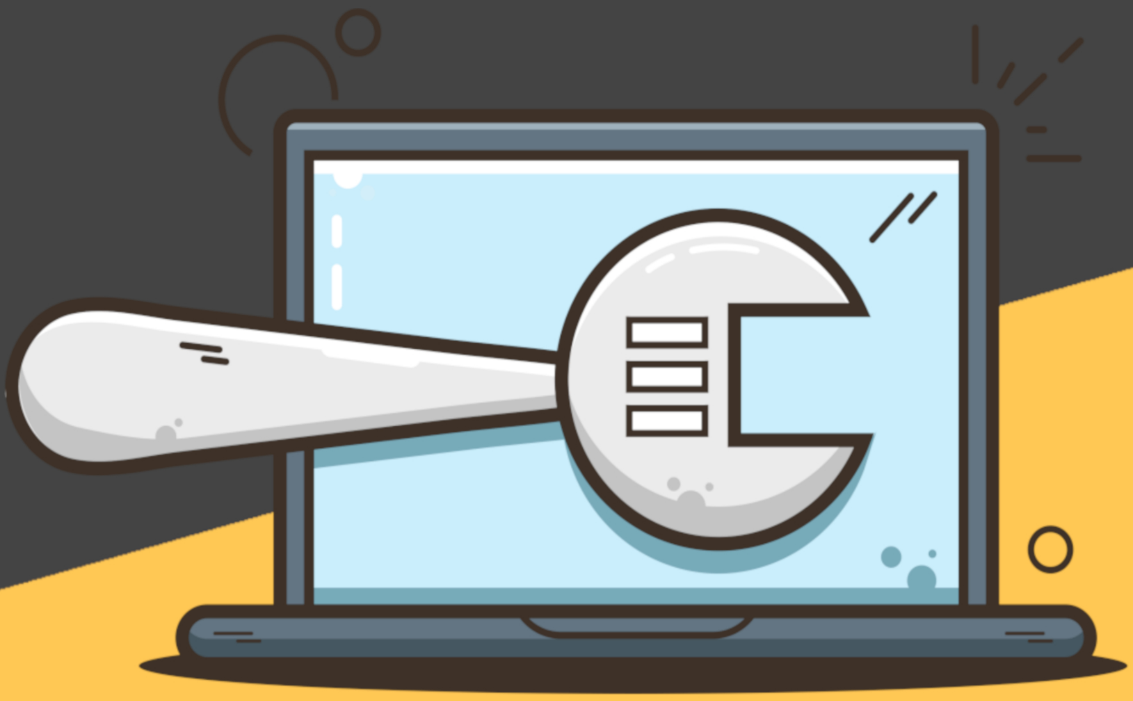
Building your

Mouseless Development Environment



Matthieu Cneude

Contents

Building Your Mouseless Development Environment

You will love my book! !!

An h1 header

An h2 header

An h3 header

An h4 header

Paragraphs are separated by a blank line.

2nd paragraph. *Italic*, **bold**, ~~deleted text~~.

We can create hypnens — like this —.

Superscript: H_2O is a liquid. 2^{10} is 1024.

Inline code: What is the difference between `>>=` and `>>`?

I love to do `rm -rf / && echo "lala"`

Here is a literal backtick `` ` ``.

Smallcaps:

Small caps

Itemized lists:

- this one
- that one
- the other one

> Block quotes are written like so.
>
> They can span multiple paragraphs, if you like.

Use 3 dashes for an em-dash. Use 2 dashes for ranges (ex., “it’s all in chapters 12–14”). Three dots … will be converted to an ellipsis.

Here’s a numbered list:

1. first item

2. second item
3. third item

Code block delimitation. Without mark:

```
define foobar() {
    print "Welcome to flavor country!";
}
```

With bash mark:

```bash
cd lala && rm -rf lolo
echo "lala" | grep "a" | less
echo "lolo"
```

With line numbers:

```bash
200 cd lala && rm -rf lolo
201 echo "lala" | grep "a" | less
202 echo "lolo"
```

Lineblocks:

The limerick packs laughs anatomical
In space that is quite economical.
        But the good ones I've seen
  So seldom are clean
And the clean ones so seldom are comical

Without lineblocks

The limerick packs laughs anatomical In space that is quite economical. But the good ones I've seen So seldom are clean And the clean ones so seldom are comical

---

A compact list:

- youpi
- youpla
- boom

A loose list:

- First paragraph.

  Continued.

- Second paragraph. With a code block, which must be indented eight spaces:

  ```
  { code }
  ```

Now a nested list:

1. First, get these ingredients:

   - carrots
   - celery
   - lentils

2. Boil some water.

3. Dump everything in the pot and follow this algorithm:

1. find wooden spoon
2. uncover pot
3. stir
4. cover pot
5. balance wooden spoon precariously on pot handle
6. wait 10 minutes
7. goto first step (or shut off burner when done)
8. Do not bump wooden spoon or it will fall.

Another way to nest lists:

1. one
2. two

Cut off line items:

- item one
- item two

```
{ my code block }
```

Not cutting off:

- item one

- item two

  { my code block }

Checking some items:

- ☐ an unchecked task list item
- ☒ checked item

Numbered examples:

(1) My first example will be numbered (1).
(2) My second example will be numbered (2).

Explanation of examples.

(3) My third example will be numbered (3).

(4) This is a good example.

As (4) illustrates, …

Here’s a link to a website, to a local doc, and to a section heading in the current doc. Write me!

See the internal-link.

Labels:

Here's a footnote [1].

Tables can look like this:

Table 1: Shoes sizes, materials, and colors.

Name	Size	Material	Color
All Business	9	leather	brown
Roundabout	10	hemp canvas	natural
Cinderella	11	glass	transparent
Cinderella	11	glass	transparent

(The above is the caption for the table.) Pandoc also supports multi-line tables:

Keyword	Text
red	Sunsets, apples, and other red or reddish things.
green	Leaves, grass, frogs and other things it's not easy being.

12	12	12	12
123	123	123	123
1	1	1	1

Centered Header	Default Aligned	Right Aligned	Left Aligned
First	row	12.0	Example of a row that spans multiple lines.
Second	row	5.0	Here's another one. Note the blank line between rows.

Right	Left	Default	Center
12	12	12	12
123	123	123	123
1	1	1	1

One	Two
my	table
is	nice

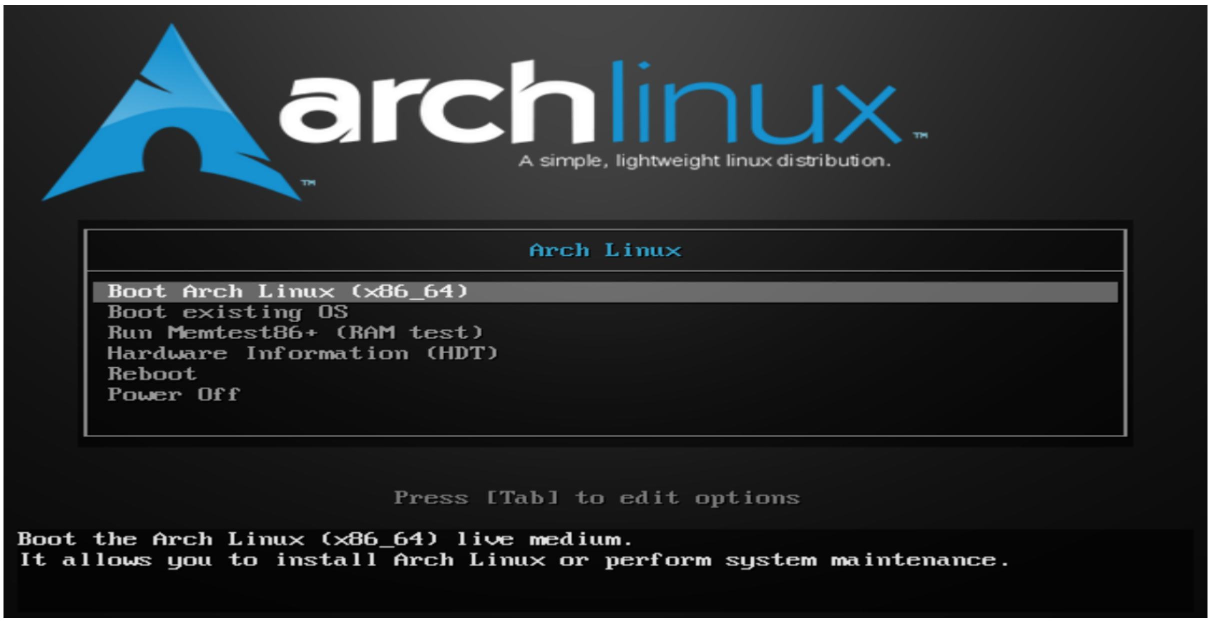
Here's a definition list:

[1] Some footnote text.

apples Good for making applesauce.
oranges Citrus!
tomatoes There’s no “e” in tomatoe.

Again, text is indented 4 spaces. (Put a blank line between each term and its definition to spread things out more.)

Images:



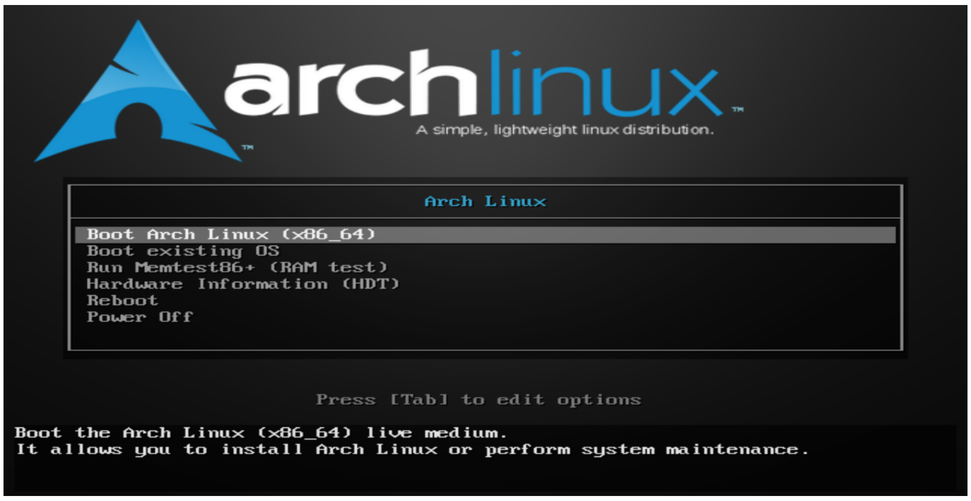
```
Arch Linux

Boot Arch Linux (x86_64)
Boot existing OS
Run Memtest86+ (RAM test)
Hardware Information (HDT)
Reboot
Power Off

Press [Tab] to edit options

Boot the Arch Linux (x86_64) live medium.
It allows you to install Arch Linux or perform system maintenance.
```

Figure 1: example image



```
Arch Linux

Boot Arch Linux (x86_64)
Boot existing OS
Run Memtest86+ (RAM test)
Hardware Information (HDT)
Reboot
Power Off

Press [Tab] to edit options

Boot the Arch Linux (x86_64) live medium.
It allows you to install Arch Linux or perform system maintenance.
```

Figure 2: example image but smaller

Inline math equation: $\omega = d\phi/dt$. Display math should get its own line like so:

$$I = \int \rho R^2 dV$$

And note that you can backslash-escape any punctuation characters which you wish to be displayed literally, ex.: ‘foo‘, *bar*, etc.

This should be on another page

First Title

Second Title

- If you know Linux, pass this part
- Concept of distribution
- Concept of repository
- The kernel
- The shell
- CLIs
  - Mainly what a Mouseless Development Environment about
- The manual, your best friend
  - Intimidating
- Try `--help` , `help` `-h` if you want some help about a CLI

Pre-requesites

- Computer with empty hard drive
- Flash drive
- Another computer to follow with this book
  - Arch wiki can be useful too.

Why Arch Linux?

- Rolling release
  - Because of that, process can change
- AUR

Before Installing

Arch Wiki

- Best resource for Linux out there.
- It’s likely they have a solution if you have a problem during installation

Burning ISO on key

```
cd && rm -rf lala
cd && rm -rf lala
```

`cd && rm -rf lala`

- Download the Arch Linux ISO
- Burn it on your key
    - If you’re on Windows, use rofi
    - You can use dd on Linux
- Go to your bios and choose the boot option
    - You need to change the order
    - Something with USB in the title

Creating Your Arch Linux Installer

The Windows Manager i3

You thought that having icons everywhere on your desktop with a beautiful start menu was the better and only way to interact with your computer?

Many Linux distributions, and many Operating Systems (OS) in general, like Windows or macOS, have their own desktop environment. It's basically a set of icons, windows, toolbars, widgets, wallpaper and other functionalities which give to the users "easy" ways to do what they want.

These desktops are meant to be used by everybody. However, everybody has different needs and aspirations when using a computer, and it's even more true for us, developers.

If you use the terminal pretty often (and you should), a desktop environment might be useless for you, even if you don't even realize it. Indeed, thanks to the terminal, a developer have already access to a tool powerful enough for him to do most of his work, without the need of a full blown desktop.

I will present you today an alternative to a desktop environment: a window manager. More precisely, I will show you how i3 windows manager (i3wm) works. It will give you an amazing flexibility and a total control on your system, without even using the mouse!

"But I love my mouse!" you might think, while sadness assault your heart. Trust me, if your fingers don't have to leave your keyboard, your happiness will go through the roof. You just need to try for enough time to be convinced.

In this article, we will:

- Learn how to install i3.
- Learn why using i3 can be very efficient.
- Learn how work i3.
- Configure and try i3 together.

Ready? Let's go!

{{< picture src="/images/2020/i3_windows_manager/i3-final-configuration-low-resolution.jpg" webp="/images/2020/i3_windows_manager/i3-final-configuration-low-resolution.webp" alt="Final result (low resolution)" caption="Final result (low resolution)">}}

Installing i3

First things first: i3 is only available for Linux. Sorry, fellow Mac users.

If you already have a Linux distribution installed and you want to try i3, you might be able to do it on top of your actual desktop. Depending on your distribution, it will be more or less easy.

Using Lubuntu with i3, for example, is pretty straightforward: the option to log in with i3 will be given automatically after install. If you use another distribution, you might have to do some minor configuration.

I personally use i3 on Arch Linux and I would recommend using a Linux distribution you know before trying i3. It will be easier for you to focus on i3 itself, instead of struggling with a new Linux distribution.

You can use as well virtual machines to install Linux and try i3 on top of Windows, macOS or whatever other OS you have. The free VMWare Player is the best option if you want to take this path. Tested and approved!

To install i3 itself, simply use your package manager. For example:

- Debian / Ubuntu / Lubuntu: `sudo apt-get install i3`
- Arch Linux: `sudo pacman -S i3`

After installing and launching i3, a popup will ask you if you would like to create a configuration automatically. Answer with a big “yes”. We will directly modify this configuration below.

The talkative i3 might ask you as well if `mod4` is the modifier you want. Again, answer “yes”. We will come back to this modifier later.

Why Using i3?

Light But Powerful

The first advantage using a Windows Manager (WM) instead of a full desktop environment is its lightness. i3 doesn’t use much resource, which let you spare memory and CPU for everything else. In short, it doesn’t get in your way.

My experience show me that we never have enough resource, especially if you’re used to run 23098832 docker containers at the same time. Many desktop environment (Unity?) are very heavy and buggy nowadays.

The simplicity of i3 will procure to your mind the peace it deserves to focus on more important problems. You know, your code for example.

Minimalist

A desktop environment has many functionalities we don’t necessarily use. The funny thing is: I didn’t know it was a problem before trying something as light as i3.

My experience with i3 is very similar to my experience with Vim: IDE and desktop environments come with a full package of software and options you didn’t choose by yourself. In the contrary, Vim, tmux or i3 come with a pretty blank state you need to personalize.

Instead of trying to shut off cumbersome functionalities (which is not always possible), you just add the functionalities you truly need.

A Windows Manager like i3 showed me that a status bar and an application launcher are enough. Thankfully, i3 comes with both.

Flexible

Another annoyance with regular desktop environments: the *windows* positioning, especially when you open a new *window*.

It always felt random to me, which means that you always need to position your *windows* manually after opening them with the sacrosanct mouse. We’re all used to do this, of course; it doesn’t mean that there are no better, more efficient and comfortable ways to manage *windows*.

The great i3 will always position *windows* following a specific layout. No surprise, *windows* positioning become highly consistent.

Even better, you can change this layout easily with specific keystrokes. It might look constraining at first glance, but at the end it makes the system way simpler than traditional desktop environments. It’s faster, too.

This is one of the greatest strength of i3. Stop spending your time moving *windows* and get important things done!

Finally, as many tools I use on a daily basis, you can configure easily i3 to answer your own personal needs. As stated above, you can change any keystroke you want, launch specific application at startup, always launch specific applications on specific virtual desktop (called *workspaces*) and configure everything you need.

This article will show a possible configuration for i3 while explaining each step, for you to be able to personalize it afterward.

The Mouse Is Not Your Best Friend

Long time readers of The Valuable Dev won't be surprised to see that i3 can be entirely used without a mouse. Having your hand constantly on your keyboard is a comfort you can't give away after tasting it for enough time.

To be honest with you, this is one of the most profound discovery I made regarding my programming workflow for the last decade.

With i3, it's quick to launch applications, move *windows* through *workspaces*, resizing them, and do many other actions using a couple of keystrokes.

How To Use i3?

Writing Your Own Cheatsheet

Writing i3's keystrokes somewhere as you learn and modify them is helpful. Not only for i3, but for any application using keystrokes extensively such as Vim or tmux.

In order to do so, I personally use Joplin, a free, open source, and powerful note taking application, similar to evernote, to write every keystroke I need. It serves as my reference when I forget what keystroke do what action.

General organisation

Like tmux, i3 store its information in a tree data structure. Let's see what each node can represent.

{{< picture src="/images/2020/i3_windows_manager/i3-tree-structure.jpg" webp="/images/2020/i3_windows_mana
tree-structure.webp" alt="Final configuration of i3">}}

Workspaces

At the top of the tree data structure, you'll find the *workspace*. It's simply the equivalent of a virtual desktop. You can have as many *workspaces* as you want.

Creating a *workspace* is simple: you just need to switch to it and create *containers* in it to make it persistent. More on that below.

Containers

A *container* contains one or multiple *windows*. Its *windows* will be positioned depending on the *container*'s layout.

There are three different layouts possible:

- **Split** - Each *window* share the *container* space and are split horizontally (*splith*) or vertically (*splitv*). This is the default layout.
- **Stacked** - The focused *window* is visible and the other ones are stacked behind. You can change the *window*'s focus via keystrokes easily. You have access to the list of *windows* open too, at the top of the *container* itself.
- **Tabbed** - This layout is similar as the stacked layout, except that the *windows*' list is vertically split, and not horizontally.

Note that a *container* can contain other *containers* as well; however, in practice, you won't often think about *containers* but more about *windows*.

Windows

A *window*, where an application is running, can be created in a *container*. It will automatically position itself and be in focus, depending on the *container*'s layout. You can move them around or even change the layout of the *container* using keystrokes.

There are two different sorts of *windows*: *fixed windows* (by default) and *floating windows*. You can precise which application should use a *floating window*. In that case, the *window* will act like any other *window* in any other desktop environment: you can focus on them and move them with your mouse.

Note that a *floating window* is not affected by a *container*'s layout.

Floating windows are useful for applications which don't need a fullscreen *window*, like pop ups or color grabbers, for example.

The Default Shortcuts

Like tmux or Vim, i3 use a special key for (almost) every keystroke. This key is called a `modifier`. By default, it's the *windows* or *cmd* key. On most keyboard, it has a Windows sign on it.

You can change this modifier in the configuration. Personally, I like to use the *windows* key.

In this article, the modifier key will be designed as `$mod` for every keystroke using it.

Enough mumbling, let's practice. I strongly advise you now to have i3 open to follow along with me, and to try the modifications we do by yourself.

Let's create a new terminal by typing `$mod + Enter`. You know what? Let's create another one! Hit `$mod + Enter` again. As you can see, the default layout of your *container* will automatically fit the two *windows* running two terminals on your *workspace*.

Now, try to focus on the second *windows* by using `$mod + arrow key`. Simple, isn't it?

You can now move the *windows* around and see how they resize themselves automatically: hit `$mod + shift + arrow key`.

You might think at that point that this system is totally dumb. Don't lie! I was pretty disappointed the first time I tried it, since I was very much used to move my *windows* around with my mouse. Now, it's difficult for me to use something else than a *window* manager like i3!

Let's try to change the layout of your *container*. You can use:

- `$mod + e` - Switch to *split* layout (splith or splitv depending on your screen)
- `$mod + s` - Switch to *stacked* layout
- `$mod + w` - Switch to *tabbed* layout

You can try to create and move *windows* using each layout to see the differences.

Finally, let's see other useful keystrokes which will be essential to write and try our new i3 configuration:

- `$mod + shift + r` - Reload i3's configuration. You need to use it each time you modify your configuration file, to apply the changes to your current i3 session.
- `$mod + shift + e` - Logout and quit i3. We will modify that later.

Configuring i3

Configuration Files

Different config files will be loaded by i3 in a precise order, overriding each other:

1. `~/.config/i3/config` (or `$XDG_CONFIG_HOME/i3/config` if set)
2. `~/.i3/config`
3. `/etc/xdg/i3/config` (or `$XDG_CONFIG_DIRS/i3/config` if set)
4. `/etc/i3/config`

If you choose to generate your config file when you run i3 for the first time, you'll need to modify the file `~/.i3/config` .

Default Configuration

Let's dive a bit more into i3's config, for you to understand its possibilities. First, open your configuration file with whatever editor you want.

We'll review in this subsection the beginning of the configuration file.

If you want a good and complete documentation of i3, it's just here. This article is a quick overview of i3 for you to understand why it can be useful for you. After that, the documentation should be your reference for everything else.

The first line of the config will define your modifier key (`$mod`), as I explained above. You can modify it if you like here.

You can see that you can define variable in i3's configuration using the keyword `set` followed by the variable name (`$mod` here) and its value (`Mod4`). It's very practical not to have to modify every single keystroke each time you change your modifier.

To see every possible value for i3's modifier key, you can run in your terminal `xmodmap` .

Below in the file, you'll see the line `bindsym $mod+Return exec i3-sensible-terminal` . `bindsym` allows you to bind a symbol to a command. Internally, a symbol is mapped to a keycode (a key on your keyboard). To see this mapping, you can run in your terminal `xmodmap -pke | less` .

Here, the symbol `Return` is used. If you prefer using directly a keycode instead of a symbol, you can use the command `bindcode` instead of `bindsym` .

This keystroke will execute (using the command `exec`) `i3-sensible-terminal` , a wrapper script which will try to find and open an instance of your terminal.

Let's continue our exploration. Below you'll find the line `bindsym $mod+Shift+q kill` , which allows you to kill a *window*. Depending on the application running in that *window*, some operation might be done before closing. For example, Firefox will save the current session.

Program Launcher

With i3, there is no start menu where you can find the applications installed on your system. Instead, you can launch your favorite software using a program launcher.

The line `bindsym $mod+d exec dmenu_run` define a keystroke to launch `dmenu` , a simple and powerful application to create menus. We will use it again below.

To launch the program launcher, simply hit `$mod+d` . You'll see at the top of your screen a discrete bar appearing. From there, you can search an application and launch it by searching it and hitting `Enter` .

If you don’t like this way of launching software, you can add your own launcher or even a start menu if you want later.

Focusing and Moving Windows

You can focus or move the different *windows* in your current *workspace* using `$mod+jkl;` or `$mod+shift+jkl;` . However, since I use Vim pretty extensively, I like to use `hjkl` to move around. Don’t forget that many other CLI uses these keys as well!

We will now modify i3’s default configuration. Don’t forget to hit `$mod + shift + r` to apply the changes.

Let’s modify the binding for focusing *windows* as following:

```
# change focus
bindsym $mod+h focus left
bindsym $mod+j focus down
bindsym $mod+k focus up
bindsym $mod+l focus right
```

Below in the file, you’ll find the keystrokes to move *windows*. You can change them as well:

```
bindsym $mod+Shift+h move left
bindsym $mod+Shift+j move down
bindsym $mod+Shift+k move up
bindsym $mod+Shift+l move right
```

That’s all! Now, you can use Vim keys to change focus or move your *windows*.

Splitting Containers

{{< picture src="/images/2020/i3_windows_manager/i3-split-horizontally.jpg" webp="/images/2020/i3_windows_ma
split-horizontally.webp" alt="Two windows in a container split horizontally" caption="Two windows in a container split horizontally">}}

Sometimes, you’ll want to open new *windows* on the side or below the *windows* already open. By default, to open a *windows* horizontally, we would need to hit `$mod+h` . However, we use this keystroke to focus *windows*.

Let reconfigure the splits as following:

```
# split in horizontal orientation
bindsym $mod+Ctrl+h split v

# split in vertical orientation
bindsym $mod+Ctrl+v split h
```

Note that changing orientation will create a new container. You can see its delimitation by changing its layout. I personally like to have only one container (it’s easier to manage). Therefore, I never use these keystrokes.

We can now continue to modify our config: let’s go into the *workspace* configuration.

Workspaces

Defining Workspaces

You can open and switch to *workspaces* with the same keystrokes. Here’s the default configuration you’ll normally find in your current configuration file:

```
# Define names for default *workspaces* for which we configure key bindings later on.
# We use variables to avoid repeating the names in multiple places.
set $ws1 "1"
set $ws2 "2"
set $ws3 "3"
set $ws4 "4"
set $ws5 "5"
set $ws6 "6"
set $ws7 "7"
set $ws8 "8"
set $ws9 "9"
set $ws10 "10"

# switch to workspace
bindsym $mod+1 workspace number $ws1
bindsym $mod+2 workspace number $ws2
bindsym $mod+3 workspace number $ws3
bindsym $mod+4 workspace number $ws4
bindsym $mod+5 workspace number $ws5
bindsym $mod+6 workspace number $ws6
bindsym $mod+7 workspace number $ws7
bindsym $mod+8 workspace number $ws8
bindsym $mod+9 workspace number $ws9
bindsym $mod+0 workspace number $ws10

# move focused container to workspace
bindsym $mod+Shift+1 move container to workspace number $ws1
bindsym $mod+Shift+2 move container to workspace number $ws2
bindsym $mod+Shift+3 move container to workspace number $ws3
bindsym $mod+Shift+4 move container to workspace number $ws4
bindsym $mod+Shift+5 move container to workspace number $ws5
bindsym $mod+Shift+6 move container to workspace number $ws6
bindsym $mod+Shift+7 move container to workspace number $ws7
bindsym $mod+Shift+8 move container to workspace number $ws8
bindsym $mod+Shift+9 move container to workspace number $ws9
bindsym $mod+Shift+0 move container to workspace number $ws10
```

What does it mean?

- The first part defines the variables for each *workspace*. You have 10 of them by default.
- The second part define keystrokes to switch to these *workspaces*.
- The third part define keystrokes to move *windows* to specific *workspace*.

From there, you can configure your *workspaces* as you wish. Here's my own configuration as an example:

```
set $terms "1: terms"
set $web "2: web"
set $db "3: db"
set $file_manager "4: files"
set $mail "5: mails"
set $documents "6: documents"
set $mindmap "7: mindmap"

# switch to workspace
bindsym $mod+1 workspace $terms
bindsym $mod+2 workspace $web
bindsym $mod+3 workspace $db
bindsym $mod+4 workspace $file_manager
bindsym $mod+5 workspace $mail
bindsym $mod+6 workspace $documents
bindsym $mod+7 workspace $mindmap
bindsym $mod+8 workspace 8
bindsym $mod+9 workspace 9
bindsym $mod+0 workspace 10

# move focused container to workspace
bindsym $mod+Shift+1 move container to workspace $terms
bindsym $mod+Shift+2 move container to workspace $web
bindsym $mod+Shift+3 move container to workspace $db
bindsym $mod+Shift+4 move container to workspace $file_manager
bindsym $mod+Shift+5 move container to workspace $mail
bindsym $mod+Shift+6 move container to workspace $documents
bindsym $mod+Shift+7 move container to workspace $mindmap
bindsym $mod+Shift+8 move container to workspace 8
bindsym $mod+Shift+9 move container to workspace 9
bindsym $mod+Shift+0 move container to workspace 10
```

If you have Font Awesome installed, you can even add some fancy icons.

Another functionality I find practical: the possibility to come back to your last *workspace* with the same keystroke. For example, if you are on the *workspace 1* and you hit `$mod+2`, you'll switch to *workspace 2*.

Then, if you hit `$mod+2` again, you'll go back to your previous *workspace*, *workspace 1*.

To enable this functionality, you need to add to your configuration:

```
`workspace_auto_back_and_forth yes`
```

Opening Applications in a Specific Workspace

You have as well the possibility to always assign precise application to defined *workspaces*. Let's say that you always want to open Firefox in the *workspace 2*, `$web`, from the example just above. You can add in your configuration:

```
assign [class="firefox" instance="Navigator"] → $web
```

You can assign the application using its class (its general identifier), its instance (an identifier which is specific to some *windows*), or both.

To get the `class` name and the `instance` name of a precise application' *window*, you need to run the application itself, open a terminal and run the following:

```
xprop | grep WM_CLASS
```

Then, click on the *window* where the application is running. Something similar to the following line will appear in your terminal:

```
WM_CLASS(STRING) = "Navigator", "firefox"
```

The first element is always the `instance`, the second always the `class`. Be careful: the case matters.

Resizing Windows

If you continue to go through the configuration, you'll find keystrokes for fullscreen and diverse ways to reload i3's config.

Below, you'll find keystrokes to resize *windows*. Let's modify them, again to match the Vim key bindings *hjlk*.

```
bindsym h resize shrink width 10 px or 10 ppt
bindsym j resize grow height 10 px or 10 ppt
bindsym k resize shrink height 10 px or 10 ppt
bindsym l resize grow width 10 px or 10 ppt
```

You'll notice that your first need to go in mode resize before being able to resize. It means that you first need to hit `$mod+r` (you'll see the word *resize* appearing at the bottom of the screen), then you can use `$mod + hjkl`.

We can as well set `focus_follows_mouse` to `no`. Otherwise, each time you'll hover on a *windows*, it will be automatically focused. Pretty annoying to me.

Locking Screen

To lock your screen for your coworkers not sending embarrassing messages on Slack on your behalf, you'll need the software `i3lock`. You can try to launch it to see if it's installed; if it's not, simply install it.

By default, the lock screen is a boring white screen which will destroy your eyes. If you don't care, you can let it as it is.

Otherwise, you can configure it to display a wonderful wallpaper as follow:

```
set $i3lockwall i3lock -i /path/to/my-wonderful-image.png -t
```

It's important to note that you can only use *png* files with `i3lock`. No *jpeg* supported!

My own wallpaper is a bit more complex. I use a script to take a screenshot of the current screen, automatically "pixelize" it, save it, and use it as a lock screen. With a lock screen that good, every coworker will pray your skills and your swag. Expect a big salary increase, too.

To run the script, you'll need to install `imagemagick` and `scrot`. Here it is:

```
img=/tmp/i3lock.png

scrot $img
convert $img -scale 10% -scale 1000% $img

i3lock -u -i $img
```

Simply save this script somewhere (I like to have a `script` subdirectory in `~/.config/i3`) and create a new variable in the i3's configuration file:

```
set $i3lockwall sh ~/.config/i3/scripts/lock.sh
```

Finally, add a keystroke to lock your screen:

```
bindsym $mod+Ctrl+Shift+l exec --no-startup-id $i3lockwall
```

We'll see that it's possible to lock your screen using `dmenu` as well (you know, the software we use already for our application launcher). More on that below.

Last but certainly not least: to unlock your computer, type your user's password and press *Enter*.

Lock, Shutdown and Reboot Menu

Let's find and delete the following line:

```
bindsym Mod1+Shift+e exec "i3-nagbar -t warning -m 'You pressed the exit shortcut. Do you really wan
```

This allows you to log out of i3. However, we'll now implement a better way to do so.

Let's create a menu with `dmenu` to have a nice and discrete way to lock your screen, log out i3, suspend your computer, hibernate, reboot, shutdown everything, or conquering the world.

Simply add the following to your configuration:

```
# shutdown / restart / suspend...
set $mode_system System (l) lock, (e) logout, (s) suspend, (h) hibernate, (r) reboot, (Ctrl+s) shutd

mode "$mode_system" {
    bindsym l exec --no-startup-id $i3lockwall, mode "default"
    bindsym e exec --no-startup-id i3-msg exit, mode "default"
    bindsym s exec --no-startup-id $i3lockwall && systemctl suspend, mode "default"
    bindsym h exec --no-startup-id $i3lockwall && systemctl hibernate, mode "default"
    bindsym r exec --no-startup-id systemctl reboot, mode "default"
    bindsym Ctrl+s exec --no-startup-id systemctl poweroff -i, mode "default"

    # back to normal: Enter or Escape
    bindsym Return mode "default"
    bindsym Escape mode "default"
}

bindsym $mod+BackSpace mode "$mode_system"
```

Now, after reloading the config, if you hit `$mod+BackSpace` , you'll see on the bottom of your screen a message which precise what you can do. You can hit `l` , `e` , `s` , `h` , `r` or `Ctrl+s` depending on what you want to do.

If you want to just close the menu, hit *Escape* or *Return*.

Wallpaper

When you switch to a *workspace* which doesn't have any *window* yet, it's nice to have a wonderful wallpaper. The easiest solution: using `feh` , a simple software which can display images.

First, You need to install it.

Then, you can add to your configuration:

```
exec --no-startup-id feh --no-fehbg --bg-fill '/path/to/your-favorite-holiday-picture.jpg'
```

Floating windows

Some applications are easier to use with *floating windows*. In short, it's a *window* you can move with your mouse.

For example, pop-up and task dialog should be floating. To do so, add these two lines in your configuration file:

```
# floating pop up automatically
for_window [window_role="pop-up"] floating enable
for_window [window_role="task_dialog"] floating enable
```

You can specify what application should be open with a floating *window* automatically, using the `class` and `instance` of the application, as explained above (see the *workspace* section).

For example:

```
for_window [class="Keepassx"] floating enable
```

Thanks to this line, the application `Keepassx` will automatically be opened in a floating *window*.

Colors and Style

You can configure colors and style for i3 directly in the configuration file. If you have your terminal colors defined in the Xresources file (typically `~/.Xresources`), you can reuse them directly with i3 if you use the `set_from_resource` directive.

For example, `set_from_resource $foreground foreground #C6C6C6` will define a variable `$foreground` which will use the `foreground` color defined in `Xresources` or, if it's not defined, the fallback color `#C6C6C6` .

You can then use the variable `$foreground` everywhere you need it in the configuration. Here's an example of my own:

```
# get color from XResource configuration - variable_name XResource fallback

# special
set_from_resource $foreground foreground #C6C6C6
set_from_resource $background background #1C1C1C
set_from_resource $cursorColor cursorColor #C6C6C6

# black
set_from_resource $black1 color0 #303030
set_from_resource $black2 color8 #444444
set $trueblack #000000

# red
set_from_resource $red1 color1 #d75f5f
set_from_resource $red2 color9 #d75f5f

# green
set_from_resource $green1 color2 #87AF87
set_from_resource $green2 color10 #87AF87

# yellow
set_from_resource $yellow1 color3 #ffd75f
set_from_resource $yellow2 color11 #ffd75f

# blue
set_from_resource $blue1 color4 #87afd7
set_from_resource $blue2 color12 #87afd7

# magenta
set_from_resource $cyan1 color5 #afafd7
set_from_resource $cyan2 color13 #afafd7

# cyan
set_from_resource $cyan1 color6 #afd7ff
set_from_resource $cyan2 color14 #afd7ff

# white
set_from_resource $white1 color7 #4E4E4E
set_from_resource $white2 color15 #ffffff
```

Instead of `set_from_resource` , you can of course define your own variables, such as `set $black #000000` .

Configuration of the i3 Bar

You can configure the status bar at the bottom of the screen too of course, using a different configuration file. You'll find below an example you can copy in a new file, for example `~/.config/i3/i3status.conf` . Feel free to modify it, of course.

You can customize the data displayed in the bar using shell scripts, too.

```
# i3status configuration file.
# see "man i3status" for documentation.

# It is important that this file is edited as UTF-8.
# The following line should contain a sharp s:
```

```
# ß
# If the above line is not correctly displayed, fix your editor first!

general {
    interval = 1
    colors = true
    color_good="#FFFFFF"
    color_degraded="#ffd75f"
    color_bad="#d75f5f"
}

order += "volume master"
order += "battery 0"
order += "disk /"
order += "disk /home"
order += "cpu_usage"
order += "load"
order += "tztime local"

tztime local {
        format = " %Y-%m-%d %H:%M:%S "
}

disk "/home" {
     format = " %avail "
}

disk "/" {
     format = " %avail "
}

cpu_usage {
    format = " %usage "
}

load {
    format = " %5min 5min "
}

volume master {
    format = " %volume "
    format_muted = " %volume "
    device = "default"
    mixer = "Master"
    mixer_idx = 0
}

battery 0 {
    format = "%status %percentage "
    format_down = " DOWN "
    status_chr = " CHARGING "
    status_bat = " BATTERY "
    status_unk = " UNKNOWN "
    #last_full_capacity = true
    status_full = " FULL "
    path = "/sys/class/power_supply/BAT%d/uevent"
}
```

To use this status bar, you need to specify the file path in i3's configuration file. Normally, the block `bar {` is already there, you just need to modify it according to your needs.

For example:

```
bar {
    status_command i3status --config ~/.i3/i3status.conf
    # Disable all tray icons
    tray_output none
    colors {
        background $black2
        statusline $white2
        separator $white2

        #                  border background text
        focused_workspace  $background $background $red1
        active_workspace   $black2 $black1 $white2
        inactive_workspace $black1 $black2 $foreground
        urgent_workspace   $red1 $red1 $white2
        binding_mode       $background $red2 $white2
    }
}
```

You can notice that we use the color variables defined above, to make the bar even prettier! Wonderful.

Managing Your Screen

If you have two, three or 389 screens, you might notice that you have no way to configure them.

Installing `arandr` can help you in that regard. You need to install it.

Then, let's make its *window* floating:

```
for_window [class="Arandr"] floating enable
```

You can now run it (using your application launcher, remember?) and configure your screen orientation and placement as you wish.

This is nice and all, but this configuration is not persistent. Here's a way to fix that:

1. Configure `arandr` as you wish
2. Save the layout from `arandr`. This will create a script similar to the following:

```
xrandr --output VGA-1 --off --output eDP-1 --primary --mode 1600x900 --pos 0x0 --rotate normal --out
```

I then use `dmenu` to create a menu to manage my different screen configurations. For example, you can add in the i3's config:

```
set $mode_display Config monitor resolution - My first config (d) - My second config (t)

mode "$mode_display" {
    bindsym d exec --no-startup-id xrandr --output VGA-1 --off --output eDP-1 --primary --mode 1600x
    bindsym t exec --no-startup-id xrandr --output eDP-1 --primary --mode 1600x900 --pos 0x809 --rot

    # back to normal: Enter or Escape
    bindsym Return mode "default"
    bindsym Escape mode "default"
}

bindsym $mod+x mode "$mode_display"
```

1. First, we set a variable `$mode_display` which will display the menu we want.
2. Then, we bind `d` and `t` to the content of the script created before.

With this method, you can create many screen configurations and switch from one to another easily.

{{< picture src="/images/2020/i3_windows_manager/i3-final-configuration.jpg" webp="/images/2020/i3_windows_r
final-configuration.webp" alt="Final configuration of i3" caption="Final result (high resolution)">}}

Will You Stick With i3?

That's all folks! With this first overview of i3, you should be able to do whatever you want without having a cumbersome desktop environment going in your way.

In this article, we learned:

- i3 is a very light and minimalist *windows* manager, yet powerful and flexible.
- What are *workspaces*, *containers*, fixed and floating *windows*.
- How to configure i3.
- How to configure the lock screen *i3lock*.
- How to create a shutdown menu.
- How to personalize the status bar.
- How to manage your screen configurations with *arandr*.

There's much more to discover about i3 and I invite you again to go through its very complete documentation.

{{}} * i3 official documentation * My personal configuration - Matthieu Cneude {{}}

The Terminal Multiplexer tmux

Do you want a powerful, flexible, and automated terminal experience?

Let me introduce you to tmux. It's one of the most important tool for my development environment.

If you never heard about tmux, fear not, this article will explain the core ideas:

- What is tmux, and what benefits it can bring you.
- How to configure tmux, step by step.
- What are the best tmux plugins you can use.
- How to automate the creation of tmux sessions.

While reading, I strongly advise you to have tmux open on the side to play along with it.

I advise you as well to *create* your own cheatsheet with the commands you'll learn today. They will be easier to memorize if you write them, and you'll have a personalized reference when your memory will fail you.

A last thing: it can be useful if you know already the basics of Vim, since the configuration we will build together mimic some Vim keystrokes. It's however not mandatory.

What's tmux?

Installing tmux

Before I can show you what's tmux, we surprisingly need to install it. If you use a Unix/Linux based system, you can find it via your usual package manager:

- Arch linux: `sudo pacman -S tmux`
- Ubuntu / Debian: `sudo apt-get install tmux`
- Fedora / CentOS: `sudo dnf install tmux`
- Mac OS: `brew install tmux`

What Is tmux?

{{< picture src="/images/2019/tmux_boost/tmux-opened.jpg" webp="/images/2019/tmux_boost/tmux-opened.webp" alt="Screenshot of tmux opened" caption="tmux in action">}}

tmux is a terminal multiplexer. You can create multiple tmux `sessions` totally independent of your terminal emulator.

If you already know GNU-screen (another terminal multiplexer), tmux is similar but more powerful and easier to config.

To understand the concept, let's try to create a tmux `session`. Open a terminal and type `tmux`. A new `session` will be created and attached to a client, your terminal.

You can list every `session` currently running by typing `tmux list-sessions`. Normally, you'll only have one `session`, with the number 0 as name.

Now, let's run this in your terminal:

```
while :; do echo 'This will never end, except if you hit CTRL+C'; sleep 1; done
```

This loop will run forever if you don't stop it! Now, close the terminal. Did the loop stopped? Not at all! It continues to run *in the background*, because the tmux `session` itself is still alive!

{{< picture src="/images/2019/tmux_boost/tmux-alive.jpg" webp="/images/2019/tmux_boost/tmux-alive.webp" alt="Tmux is still alive!">}}

You don't believe me? You're right; you need to experiment by yourself.

1. Again, open a new terminal and type `tmux list-sessions` . You will see the `session_name` displayed before a colon (`:`). Normally, it's `0` .
2. Reattach the `session` to your freshly opened terminal by typing `tmux attach-session <session name>` .

Your infinite loop is back, and you can see that it was still running in the background! Let's really kill it this time, by pressing `ctrl+c` .

"That's nice and all your zombie stories, but what's the point?" could ask many readers. Well, let's find out.

Why Using tmux?

Background Operations

As we saw above, you can detach a tmux `session` from a client (the terminal) and you can attach it back. It means that you can run whatever you want in the background, even if you have no terminal open.

Let's imagine that you need to run a very long script on your remote server. You could:

1. Connect to your remote server via SSH.
2. Launch tmux on the remote server.
3. Run a script which takes hours.
4. Close the SSH connection. The script will still run on the remote server, thanks to tmux!
5. Switch off your own computer and go home.

More Terminals! Everywhere!

Tmux allows you to create multiple terminals on a single screen. This is the functionality I use the most.

You might think: "Well, great, many terminal emulators can do the same, like terminator".

That's true, but tmux is more powerful and consume less resources. You can configure it easily and precisely, according to your specific needs.

It works very well with Vim too, which makes it mandatory if you want to create a mouseless development environment.

Protection Against Terminal Crashes

Since your tmux `session` is independent of your terminal, you don't need to worry anymore if you close it or even if it crashes. You can always reattach your `session` afterward, in a new and shiny terminal!

Saving tmux Sessions

It's possible to save tmux `sessions` in a file and reopen them later, even after switching off your computer.

Remote Pair Programming

A tmux `session` can be attached to many clients (terminals), even via SSH. Which means that you can do pair programming with another developer using tmux and Vim, by sharing the same `session` !

How to use tmux?

General Organization

Let's see in more details how to use tmux. This part should answer many potential questions you might have, at that point.

Here's an example what kind of hierarchy you can with tmux:

{{< picture src="/images/2019/tmux_boost/tmux-working-diagram.jpg" webp="/images/2019/tmux_boost/tmux-working-diagram.webp" alt="tmux organizational diagram">}}

When you launch tmux, it will create a `tmux-server` , a `session` , a `window` and a `pane` .

tmux Server

The tmux server manage every single tmux `session` . If you kill it, you kill every `session` as well. You can try it by yourself with the command `tmux kill-server` .

Sessions

We spoke about `sessions` before. You can detach them from a client (and let them run in the background) and attach them back.

Windows

In tmux, a `window` represent an entire screen. You can have multiple `windows` open in one `session` . You can access each `window` via a tab in the tmux status bar, at the bottom.

Panes

You can split your `windows` in `panes` to have multiple terminals on one screen. These `panes` are independent by default, but you can synchronize them too.

tmux Workflow

tmux Keystrokes

Let's speak about the workflow itself. It's very easy to use tmux with the keyboard only: that's why it's part of my mouseless development environment. Another great strength!

However, tmux needs a way to separate its own shortcuts with the CLIs's shortcuts running in the different `panes` . Imagine, for example, that you have Vim running in tmux: the keystrokes you use in Vim should be different from the ones you use in tmux.

If they overlap, you might accomplish different actions in tmux *and* in Vim, at the same time. This is sometimes what we want, but most of the time it's confusing.

That's why most of the keystrokes in tmux need to be done *after* entering a prefix key. This prefix key is `ctrl+b` by default.

Let's try it: go back to a tmux `session` and press `ctrl+b` , then hit `"` . Congratulation! You just opened a new `pane` . You can close it by simply typing `exit` .

If you think that the keystrokes are awkward and difficult to remember, no worries: we will change them later.

I will represent, in this article, the keystrokes which *must* be hit after the prefix keyas follow: `prefix key -> "` . In this example, it means that you need to hit the prefix key, *then* the key `"` .

Command Prompt

You can execute special tmux commands via a command prompt:

1. Type the prefix key `ctrl+b`
2. Type `:`
3. Welcome to the command prompt! Now, type `split-window`
4. You just created another `pane` !

You can execute many commands in tmux via some keystrokes or via this command prompt.

{{< picture src="/images/2019/tmux_boost/tmux-command-prompt.jpg" webp="/images/2019/tmux_boost/tmux-command-prompt.webp" alt="tmux command prompt" caption="tmux prompt">}}

Managing tmux Sessions

Here are the most useful tmux commands to manage your `sessions` :

- `tmux list-sessions` - List tmux `sessions`
- `tmux new-session -s hello` - Create a new `session` named "hello"
- `tmux kill-session -t hello` - Kill the `session` named "hello"
- `tmux kill-server` - Kill the tmux server and, as a result, every `session`

If you use a tool to automate the creation of your `sessions` , like `tmuxp` , you will barely need to use these commands. More on that later!

Configuring tmux

Let's now create our own configuration to make tmux keystrokes more user-friendly. This configuration is not the best-configuration-in-the-universe-and-beyond, but adding step by step what you might need and playing with tmux at the same time will teach you how tmux works.

I encourage you to personalize this configuration following your own needs.

It's time to configure tmux. Create the configuration file `~/.tmux.conf` . Again, you should have tmux open too, to see what the configuration is doing.

Changing The Default Prefix Key

The default prefix key `ctrl+b` (`C-b` in tmux) is not very easy to remember or even type on a keyboard.

I prefer using `ctrl+space` (`C-space`). It's easy to type and I can remember it easily since my Vim's leader key is `space` . It brings some consistency in my workflow.

Let's modify the prefix key by adding the following in the configuration file:

```
unbind C-b
set -g prefix C-Space
```

Here, the `-g` flag means global: the prefix key will be set for every `session` , `window` and `pane` in tmux. You could set the option only for a precise window for example, directly in the command prompt using `set -w prefix C-Space` .

It can be useful if you want to try multiple prefix key on the fly.

Reloading tmux Config File

Each time you change your config file, you need to reload it in tmux to apply the changes.

Let's set a keystroke to reload the config file easily:

```
unbind r
bind r source-file ~/.tmux.conf \; display "Reloaded ~/.tmux.conf"
```

Here, we have two commands combined into one, on the second line:

1. We bind the key `r` to reload the config file.
2. We display a message when we hit `prefix -> r`.

Let's stop tmux by typing `exit`. Then, run tmux again and, if you hit `prefix key -> r`, you'll normally see the message `Reloaded ~/.tmux.conf` displayed at the bottom of your tmux `session`
.

From now on, each time you want to see the effect of your configuration changes, you need to reload the config file.

Mouse Mode

What? A mouse in a mouseless environment?

Well, using the mouse is sometimes practical, especially since you might use it in other software, like your browser.

Enabling the mouse allows you to use it for:

- Selecting `panes`
- Selecting `windows` (via the status line)
- Resizing `panes`
- Scrolling `windows`
- Copying text

Let's add it in the configuration:

```
set -g mouse on
```

You can do everything with your keyboard too, and I encourage you to do so.

Splitting Panes

{{< picture src="/images/2019/tmux_boost/tmux-two-panes.jpg" webp="/images/2019/tmux_boost/tmux-two-panes.webp" alt="Tmux with two panes" caption="A tmux window split in two panes">}}

One of the most common operation you'll do in tmux is splitting a `window` into multiple `panes`.
Let's add some useful keystrokes in our config:

```
# v and h are not binded by default, but we never know in the next versions...
unbind v
unbind h

unbind % # Split vertically
unbind '"' # Split horizontally

bind v split-window -h -c "#{pane_current_path}"
bind h split-window -v -c "#{pane_current_path}"
```

You might think that something is reversed here, between my keystroke `v` (for vertical) and the tmux flag `-h` (for horizontal). Well, yes and no.

The flag `-c` execute a shell command, and `#{pane_current_path}` simply bring you back where you were in the filesystem.

Resizing Panes

You can resize the `panes` with your mouse or by typing `prefix key -> alt + arrow key` .

Navigating Panes

To navigate from `pane` to `pane` , you can use `prefix key -> arrow key` by default. However, this action is so common I like to use the following:

```
bind -n C-h select-pane -L
bind -n C-j select-pane -D
bind -n C-k select-pane -U
bind -n C-l select-pane -R
```

The flag `-n` means that these binding don't use the prefix key.

Why using `ctrl+hjkl` ? Well, as I explained in my article Vim for beginners, it's always better to keep your fingers on the home row keys. Using the arrow keys force you to move your hands. If you stop doing it, you'll understand how comfortable it is.

More History! MORE!

One of the functionality I love in tmux is being able to have thousands of lines from the terminal outputs. You need to add the following to your config file:

```
set -g history-limit 100000
```

You can then perform some search on the past output very easily, especially if you use the search plugins I describe below.

Windows

{{< picture src="/images/2019/tmux_boost/tmux-two-windows.jpg" webp="/images/2019/tmux_boost/tmux-two-windows.webp" alt="tmux with two windows" caption="a tmux session with two windows">}}

We know how to split the current `pane` , but we don't know yet how to create new `windows` and how to navigate between them.

Let's add some more keystrokes to our config file:

```
unbind n  #DEFAULT KEY: Move to next window
unbind w  #DEFAULT KEY: change current window interactively

bind n command-prompt "rename-window '%%'"
bind w new-window -c "#{pane_current_path}"
```

The keystroke `prefix key -> w` will create a new `window` , and `prefix key -> n` will allow you to rename the current `window` .

You can see some tabs in the status line, at the bottom left of your tmux `session` . Each tab represents one `window` .

When you have more than one `window` , you can then select them with `prefix key -> <number>` . `<number>` is the `window` number you want to display.

You can see that a little star in the `window` 's tab indicate what `window` is currently displayed in your terminal. If you think that it's not really readable, no worries: we will improve the status bar later.

You can notice as well that the `windows` ' numbers begin by 0. However, I find it more practical to set the first `window` to 1 instead.

To stay consistent, let's do the same with the `panes` by adding in the config:

```
set -g base-index 1
set-window-option -g pane-base-index 1
```

You might think it's annoying to do `prefix key -> <number>` , since you'll navigate between `windows` often. I like to use `alt+k` to go to the next `window` , and `alt+j` to go to the previous one, without using the prefix key:

```
bind -n M-j previous-window
bind -n M-k next-window
```

Copy Mode

{{< picture src="/images/2019/tmux_boost/tmux-copy-mode.jpg" webp="/images/2019/tmux_boost/tmux-copy-mode.webp" alt="tmux copy mode" caption="tmux in copy mode">}}

Let's speak briefly about tmux modes. There are two of them:

- The default mode
- The copy mode

We actually use the default mode of tmux. You can think of it as the equivalent of the Insert Mode in Vim. It allows you to type whatever you want in the different `panes` .

When you are in copy mode, you have the possibility to navigate in your terminal, using Emacs or Vi key bindings. Since I'm in love with Vi/Vim, let's add in the config:

```
set-window-option -g mode-keys vi
```

Now, let's try the copy mode! Hit `prefix key -> [` . You can come back to the default mode by hitting `q` .

You can see that you're in copy mode thanks to the two numbers on the top right corner of your `pane` . They represent the total number of lines which are below the visible output (on the left) and the ones which are above (on the right).

From there, you can use the vanilla Vim keystrokes to navigate your current terminal output. Again, I wrote about the basic Vim keystrokes in this article.

You can use, for example:

- `Ctrl-u` - Scroll up
- `Ctrl-d` - Scroll down
- `/` - Search

Exactly like in Vim!

Unsurprisingly, you will use copy mode mainly to copy content.

I you tried to copy terminal outputs already from tmux with your mouse, you might be confused: it doesn't really work.

Indeed, tmux by default doesn't copy anything in your system clipboard, but in one of its `paste buffer` . However, for convenience, I prefer using the system clipboard each time I copy something.

Let's configure that now:

```
unbind -T copy-mode-vi Space; #Default for begin-selection
unbind -T copy-mode-vi Enter; #Default for copy-selection

bind -T copy-mode-vi v send-keys -X begin-selection
bind -T copy-mode-vi y send-keys -X copy-pipe-and-cancel "xsel --clipboard"
```

To stay consistent with Vim's default keybinding, we did the following:

1. Changing the keystroke to select text from `space` to `v` .
2. Changing the keystroke to copy text from `enter` to `y` .

Then, we pipe the action to `xsel` , to copy the selection to the system clipboard. If you don't have `xsel` installed or if you prefer using `xclip` , you can replace the last line in your configuration with this one:

```
bind -T copy-mode-vi y send-keys -X copy-pipe-and-cancel "xclip -i -f -selection primary | xclip -i
```

`xsel` or `xclip` will work well on Linux systems. For mac, you can apparently use pbcopy.

Be aware as well that you can past what's in the tmux `past buffer` , by using `prefix key -> ]` , if you don't want to use the system clipboard.

And voila! Now, copy pasting in tmux will normally work like a charm. You can as well copy using the mouse: select what you want and enter `y` without releasing the mouse button. If you release it, the selection will disappear before you had the chance to copy it.

Integrate Your tmux With Vim

There are some more configuration you need in order for tmux to work seemingly with Vim. If you don't use Vim, you can pass this section.

First, if you use Neovim in tmux, you might experience a cursor problem: it doesn't change from rectangle to pipe when you go from Normal Mode to Insert Mode.

Adding this line in your configuration might help:

```
set -g -a terminal-overrides ',*:Ss=\E[%p1%d q:Se=\E[2 q'
```

Another improvement you might want: navigating between tmux `panes` and Vim `windows` using the same keystrokes, `ctrl+hjkl` .

This config will do exactly that:

```
# Smart pane switching with awareness of Vim splits.
# See: https://github.com/christoomey/vim-tmux-navigator

is_vim="ps -o state= -o comm= -t '#{pane_tty}' \
    | grep -iqE '^[^TXZ ]+ +(\\S+\\/)?g?(view|n?vim?x?)(diff)?$'"
bind -n C-h if-shell "$is_vim" "send-keys C-h"  "select-pane -L"
bind -n C-j if-shell "$is_vim" "send-keys C-j"  "select-pane -D"
bind -n C-k if-shell "$is_vim" "send-keys C-k"  "select-pane -U"
bind -n C-l if-shell "$is_vim" "send-keys C-l"  "select-pane -R"
bind -n C-\ if-shell "$is_vim" "send-keys C-\\" "select-pane -l"
```

That's all!

Now you have a personalized tmux configuration you can modify to fit your precise needs. There are a lot of tmux configuration available out there, if you need some inspiration.

You might think that it will take you forever to remember these keystrokes, but I believe they are overall pretty logical. Moreover, if you have written your own cheatsheet, I bet you know most of them already.

tmux Colors

{{< picture src="/images/2019/tmux_boost/tmux-solarized-theme.jpg" webp="/images/2019/tmux_boost/tmux-solarized-theme.webp" alt="tmux with solarized theme" caption="the status bar looks finally better">}}

Right now, the status bar of tmux is quite ugly: you can't really see properly the `windows` displayed and it's very... green. Boring!

You can personalize the status bar of tmux extensively.

Changing tmux colors can help to bring clarity in the status bar. One popular theme you can try is `solarized`. You can copy past this file in your config.

You can look at my own personalized design too as well, for inspiration.

Enhancing tmux With Plugins

Even if we have now a very powerful tool, we can improve tmux even more by adding some useful plugins.

tmux Plugins Manager

To manage our tmux plugins, we need the tmux Plugin Manager.

To install it, follow these steps:

1. Clone the project into your home directory: `git clone https://github.com/tmux-plugins/tpm ~/.tmux/pl`
2. Add this line in the configuration: `set -g @plugin 'tmux-plugins/tpm'`
3. Add this **at the end** of tmux's configuration: `run -b '~/.tmux/plugins/tpm/tpm'`

You can add any plugin you want in your config and install them with `prefix key -> I`, after reloading the config.

A Better Search With CopyCat

As we saw, tmux copy mode allow you to search in the output of the current `pane`. However, if you want advanced search capabilities, you can try tmux copycat.

What are the cool benefits, you might ask?

- You don't need to be in copy mode to search something.
- You can search using regex or plain strings.
- You have access to keystrokes to select filepaths, git status output, urls and IP addresses to copy them easily.
- Your search is automatically selected. You can copy it using yank (the `y` key we configured earlier) directly.

Like in Vim, you can use `n` and `N` to jump to the next or previous result, respectively.

To install copycat, copy this line to your config:

```
set -g @plugin 'tmux-plugins/tmux-copycat'`
```

Then, you need to reload your tmux config and hit `prefix key -> I` to install copycat.

Fuzzy Search With fzf And Extrakto

You'll need fzf touse this plugin. If you don't have it already, I advise you to install it: it's a very powerful tool which will enhance your terminal experience even more!

In order to install extrakto, add the following to your config file:

```
set -g @plugin 'laktak/extrakto'
```

Then, reload it and hit `prefix key -> I` to install the new plugin.

You can now fuzzy search the output of your terminal using `prefix key -> tab`. Pressing `enter` will copy the search in your clipboard.

Automating The Creation of tmux Sessions

The power of tmux doesn't stop here. There are many tools out there which allow you to automate tmux `session` creation via YAML (or JSON) config files.

You can automate everything, from the number of `windows` and `panes` you want to, to the command lines to execute in which `pane`.

The two most known tools to create `sessions` from config files are tmuxinator and tmuxp.

Here's a simple example of a `tmuxp` config file:

```
session_name: dotfiles
start_directory: $HOME/.dotfiles

windows:
  - window_name: nvim dotfiles
    layout: tiled
    panes:
      - nvim +FZF
  - window_name: terms
    layout: tiled
    panes:
      -
  - window_name: example
    layout: tiled
    panes:
      - shell_command:
          - ls
          - ls -lah
```

This configuration set up a `session` with 3 `windows` containing one `pane` each. In the first `window`, `nvim` will be executed. In the third `window`, two commands will run, one after the other: `ls` and `ls -lah`.

You need to create every `tmuxp` config file in the directory `~/.tmuxp`. You can then load them using `tmuxp load <session_name>`.

This configuration is simple, but you can personalize your tmux `sessions` way, way more.

Welcome To tmux World

Now that you've played around with tmux, you can use it to its maximum. Don't hesitate to try new configurations or new plugins. Experimentation is key, for you to find what fits the best.

What did we learn in this article?

- The shell is a powerful tool we should leverage to improve our development workflow.
- A tmux server manage all your tmux sessions. These sessions include windows and panes.
- You can configure tmux depending on your own workflow, needs and personality.
- Every keystrokes include the prefix key. It's easier to remember and doesn't conflict with other programs.
- You can enhance tmux even further with useful plugins.

To dig more into tmux, I would definitely recommend you to look at the well written manual (`man tmux`).

There are many boring tasks we repeat day after day: creating, copying, moving or searching files, launching again and again the same tools, docker containers, and whatnot.

For a developer, the shell is a precious asset which can increase your efficiency over time. It will bring powerful tools at your fingertips, and, more importantly, it will allow you to automate many parts of your workflow.

To leverage these functionalities, you'll need a powerful and flexible shell. Today, I would like to present your next best friend: the Z shell, or Zsh.

If you look at the documentation (around 450 pages for the PDF version), Zsh can feel daunting. There are so many options available, it can be difficult to come up with a basic configuration you can build upon.

We'll build, in this article, a basic Zsh config. I'll explain the meaning of (almost) everything along the way, including:

- What's a Unix shell.
- Why Zsh is a good choice.
- How to install Zsh.
- A brief overview of:
    - Useful environment variables.
    - Aliases.
    - The Zsh options.
    - The Zsh completion.
    - The Zsh prompt.
    - The Zsh directory stack.
- How to configure Zsh to make it Vim-like.
- How to add external plugins to Zsh.
- External programs you can use to improve your Zsh experience.

Are your keyboard ready? Are you fingers warm? Did you stretch your arms? Let's begin, then!

Brief Unix Shell Overview

A shell *interpret* command lines. You can type them using a prompt in an *interactive shell*, or you can run shell scripts using a *non-interactive shell*.

The shell run just after you logged in with your user. You can imagine the shell as the layer directly above the kernel of Unix-based operating systems (including Linux). Here's the charismatic Brian Kernighan explaining it casually with his feet on a table.

When you use a graphical interface (or GUI), you click around with your mouse to perform tasks. When you use a shell, you use plain text instead.

If you use a graphical interface (like a windows manager or a desktop environment), you'll need a *terminal emulator* to access the shell. In the old days, a terminal was a real device. Nowadays, it's a program.

The shell gives you access to many powerful programs. They are called CLIs, or Command Line Interfaces.

At that point, you might wonder: why using a shell, instead of a graphical interface?

- It's difficult to get a graphical interface right, especially if your software has many functionalities. It can be simpler to build a CLI to avoid some complexity.
- CLIs are usually faster.
- A developer deals often with plain text. CLIs are great for that.
- Many shells, like Linux shells, allow you to pipe CLIs together in order to create a powerful transformation flow.
- It's easier to automate textual commands rather than actions on a graphical interface.

{{< blockquote author="The Pragmatic Programmer" >}} Play around with your command shell, and you'll be surprised at how much more productive it makes you. {{< /blockquote >}}

A shell is the keystone of a Mouseless Development Environment, and the most powerful tool you can use as a developer.

{{}} {{}} {{}}

Why Zsh?

There are other Linux shells available out there, including the famous Bash. Why using Zsh?

- The level of flexibility and customization of Zsh is crazy.
- You have access to a powerful auto-completion for your favorite CLIs.
- The Vi mode is golden for every Vim lovers.
- There is an important and active community around Zsh.
- Bash scripts are (mostly) compatible with Zsh.

Framework Or No Framework?

You'll see many advising you to install a Zsh framework with a crazy number of plugins, options, aliases, all already configured. The famous ones are Oh My Zsh and prezto.

I tried this approach for years and I think the drawbacks outweigh the benefits:

- I have no clue what's included in these frameworks. When I read their documentations, I can't possibly remember everything it sets. Therefore, I barely use 10% of the functionalities.
- Zsh has already many functionalities and options, it's even more daunting to have a framework on top.
- A framework is a big external dependency which brings more complexity. If there is a conflict with my own configuration or a bug, it can take a long time to figure out what's happening.
- A framework impose rules and way of doing I don't necessarily want, or need.

Don't get me wrong: these frameworks are incredible feats. They can be useful to get some inspiration for your own configuration. But I wouldn't use them directly.

Let The Party Begin

We'll now configure Zsh. If the files or folders I'm speaking about don't exist, you need to create them.

This configuration was tested with a Linux based system. I have no idea about macOS, but it should work.

Installing Zsh

You can install Zsh like everything else:

- Debian / Ubuntu: `sudo apt install zsh`
- Red Hat: `sudo yum install zsh`
- Arch Linux: `sudo pacman -S zsh`
- macOS (with brew): `brew install zsh`

Then, run it in a terminal by typing `zsh` .

Zsh Config Files

To configure Zsh for your user's session, you can use the following files:

- `$ZDOTDIR/.zshenv`
- `$ZDOTDIR/.zprofile`

- `$ZDOTDIR/.zshrc`
- `$ZDOTDIR/.zlogin`
- `$ZDOTDIR/.zlogout`

In case you wonder what `$ZDOTDIR` stands for, we'll come back to it soon.

Zsh read these files in the following order:

1. `.zshenv` - Should only contain user's environment variables.
2. `.zprofile` - Can be used to execute commands just after logging in.
3. `.zshrc` - Should be used for the shell configuration and for executing commands.
4. `.zlogin` - Same purpose than `.zprofile`, but read just after `.zshrc`.
5. `.zlogout` - Can be used to execute commands when a shell exit.

We'll use only `.zshenv` and `.zshrc` in this article.

Zsh Configuration Path

By default, Zsh will try to find the user's configuration files in the `$HOME` directory. You can change it by setting the environment variable `$ZDOTDIR`.

Personally, I like to have all my configuration files in `$HOME/.config`. To do so:

1. I set the variable `$XDG_CONFIG_HOME` as following: `export XDG_CONFIG_HOME="$HOME/.config"`.
2. I set the environment variable `$ZDOTDIR`: `export ZDOTDIR="$XDG_CONFIG_HOME/zsh"`.
3. I put the file `.zshrc` in the `$ZDOTDIR` directory.

Most software will use the path in `$XDG_CONFIG_HOME` to install their own config files. As a result, you'll have a clean `$HOME` directory.

Unfortunately, the file `.zshenv` **needs to be in your home directory**. It's where you'll set `$ZDOTDIR`. Then, every file read after `.zshenv` can go into your `$ZDOTDIR` directory.

Zsh Basic Configuration

Environment Variables

As we saw, you can set the environment variables you need for your user's session in the file `$HOME/.zshenv`. This file should only define environment variables.

For example, you can set up the XDG Base directory there, as seen above:

```
export XDG_CONFIG_HOME="$HOME/.config"
export XDG_DATA_HOME="$XDG_CONFIG_HOME/local/share"
export XDG_CACHE_HOME="$XDG_CONFIG_HOME/cache"
```

You can as well make sure that any program requiring a text editor use your favorite one:

```
export EDITOR="nvim"
export VISUAL="nvim"
```

You can set some Zsh environment variables, too:

```
export ZDOTDIR="$XDG_CONFIG_HOME/zsh"

export HISTFILE="$ZDOTDIR/.zhistory"    # History filepath
export HISTSIZE=10000                   # Maximum events for internal history
export SAVEHIST=10000                   # Maximum events in history file
```

I already explained the first line. For the other ones, they will:

- Store your command line history in the file `.zhistory` .
- Allows you to have a history of 10000 entries maximum.

Here's my .zshenv file, if you need some inspiration.

Aliases

Aliases are crucial to improve your efficiency. For example, I have a bunch of aliases for git I use all the time. It's always easier to type when it's shorter:

```
alias gs='git status'
alias ga='git add'
alias gp='git push'
alias gpo='git push origin'
alias gtd='git tag --delete'
alias gtdr='git tag --delete origin'
alias gr='git branch -r'
alias gplo='git pull origin'
alias gb='git branch '
alias gc='git commit'
alias gd='git diff'
alias gco='git checkout '
alias gl='git log'
alias gr='git remote'
alias grs='git remote show'
alias glo='git log --pretty="oneline"'
alias glol='git log --graph --oneline --decorate'
```

I like to have my aliases in one separate file (called, surprisingly, `aliases`), and I source it in my `.zshrc` :

```
source /path/to/my/aliases
```

Here are all my aliases.

Zsh Options

You can set or unset many Zsh options using `setopt` or `unsetopt` . For example:

```
setopt HIST_SAVE_NO_DUPS         # Do not write a duplicate event to the history file.
unsetopt HIST_SAVE_NO_DUPS       # Write a duplicate event to the history file
```

You can already do a lot of customization only using these options.

Zsh Completion System

The completion system of Zsh is one of its bigger strength, compared to other shells.

To initialize the completion for the current Zsh session, you'll need to call the function `compinit` . More precisely, you'll need to add this in your `zshrc` :

```
autoload -Uz compinit; compinit
```

What does it mean?

The `autoload` command load a file containing shell commands. To find this file, Zsh will look in the directories of the *Zsh file search path*, defined in the variable `$fpath` , and search a file called `compinit` .

When `compinit` is found, its content will be loaded as a *function*. The function name will be the name of the file. You can then call this function like any other shell function.

{{}} What about `;` ? It's just a handy way to separate commands. It's the same as calling `compinit` on a new line. {{}}

Why using autoload, and not sourcing the file by doing `source ~/path/of/compinit` ?

- It avoids name conflicts if you have an executable with the same name.
- It doesn't expand aliases thanks to the `-U` option.
- It will load the function only when it's needed (lazy-loading). It comes in handy to speed up Zsh startup.

The `-z` option tells Zsh that your function is written using "Zsh style". I'm not sure what's the "Zsh style", but it's an idiomatic way to autoload functions.

Then, let's add the following;

```
_comp_options+=(globdots) # With hidden files
source /my/path/to/zsh/completion.zsh
```

The first line will auto-complete dotfiles.

The second line source this file which is from the Zsh framework prezto. I basically pruned everything I didn't want. The original file is here.

If you open this completion file, you'll see many instances of the command `zstyle` . It can be used to define and lookup styles for Zsh. You can play around with it to understand what it does.

Now, the auto-completion should work:

- If you type `cp` and hit the tab key, you'll see that Zsh will auto-complete the command.
- If you type `cp -` and hit the tab key, Zsh will display the possible arguments for the command.

{{< picture src="/images/2020/zsh/auto_complete.png" webp="/images/2020/zsh/auto_complete.webp" alt="Zsh auto-completion in action">}}

Pimp My Zsh Prompt

What would be the shell experience without a nice prompt? Dull. Tasteless. Depressing.

Let's be honest here: Zsh default prompt is ugly. We need to change it, before our eyes start crying some blood. My needs are simple:

- The prompt needs to be on one line. I had display problems with two lines.
- The prompt needs to display some git info when necessary.

From there, I created my own prompt from another one. It looks like that:

{{< picture src="/images/2020/zsh/prompt.png" webp="/images/2020/zsh/prompt.webp" alt="Zsh prompt">}}

If you open the prompt script, you'll see that it's pretty simple:

- I set two environment variables: `$PROMPT` and `$RPROMPT` . The first one format the left prompt, the second display git information on the far right.
- You can add some formatting styles using, for example, `%F{blue}%f` to change the color, or `%Bmy-cool-prompt%b` to make everything bold.

This prompt doesn't need any external dependency. You can copy it right away and modify it as much as you want.

Here's everything you need, to create the prompt of your dream.

To load the prompt, you need to add something like that in your `zshrc` :

```
fpath=(/my/path/to/zsh/prompt $fpath)
autoload -Uz name_of_the_prompt_file; name_of_the_prompt_file
```

The first line will add the folder containing the prompt to `$fpath`, as discussed above. It will ensure as well that any function declared in the folder `/my/path/to/zsh/prompt` will overwrite every other ones with the same name, in other `fpath` folders.

The second line autoload the prompt itself.

This prompt require font awesome 4 for the git icons. You can download the font and install it, or you can change the icons.

Zsh Directory Stack

Zsh has commands to push and pop directory on a directory stack. Bash as well, if you didn't know.

By manipulating this stack, you can set up an history of directory visited, and be able to jump back to these directories.

First, let's set some options in your `.zshrc`:

```
setopt AUTO_PUSHD           # Push the current directory visited on the stack.
setopt PUSHD_IGNORE_DUPS    # Do not store duplicates in the stack.
setopt PUSHD_SILENT         # Do not print the directory stack after pushd or popd.
```

Then, you can create these aliases:

```
alias d='dirs -v'
for index ({1..9}) alias "$index"="cd +${index}"; unset index
```

What does it do?

- Every directory visited will populate the stack.
- When you use the alias `d`, it will display the directories on the stack prefixed with a number.
- The line `for index ({1..9}) alias "$index"="cd +${index}"; unset index` will create aliases from 1 to 9. They will allow you to jump directly in whatever directory on your stack.

For example, if you execute `1` in Zsh, you'll jump to the directory prefixed with `1` in your stack list.

You can as well increase `index ({1..9})` to `index ({1..100})` for example, if you want to be able to jump back to 100 directories.

Here's how it looks like:

{{< picture src="/images/2020/zsh/directory_stack.png" webp="/images/2020/zsh/directory_stack.webp" alt="Zsh directory stack example">}}

Zsh By Default

When you're ready psychologically to set Zsh as your default shell, you can run these commands:

- For Linux: `chsh -s $(which zsh)`
- For macOS: `sudo sh -c "echo $(which zsh) >> /etc/shells" && chsh -s $(which zsh)`

A good soul on Reddit whispered me that Zsh is now the default shell from macOS Catalina onwards, so you don't necessarily need the above command.

Zsh is now part of your life. Congratulation!

Zsh With Vim Flavors

For editing purposes, Vim is my best friend. I love when CLIs use some Vim key binding, and Zsh gives you even more than that.

Activating Vi Mode

Zsh has a Vi mode you can enable by adding the following in your `.zshrc` :

```
bindkey -v
export KEYTIMEOUT=1
```

You can now switch between INSERT and NORMAL mode (called as well COMMAND mode) with `esc` . I write the different modes in uppercase here for clarity, but it doesn't have to be.

The second line `export KEYTIMEOUT=1` makes the switch between modes quicker.

Changing Cursor

A visual indicator to show the current mode (NORMAL or INSERT) could be nice. In Vim, my cursor is a beam `|` when I'm in INSERT mode, and a block ` ` when I'm in NORMAL mode. I wanted the same for Zsh.

You can add the following in your `zshrc` , or autoload it from a file, as I did.

```
cursor_mode() {
    # See https://ttssh2.osdn.jp/manual/4/en/usage/tips/vim.html for cursor shapes
    cursor_block='\e[2 q'
    cursor_beam='\e[6 q'

    function zle-keymap-select {
        if [[ ${KEYMAP} == vicmd ]] ||
            [[ $1 = 'block' ]]; then
            echo -ne $cursor_block
        elif [[ ${KEYMAP} == main ]] ||
            [[ ${KEYMAP} == viins ]] ||
            [[ ${KEYMAP} = '' ]] ||
            [[ $1 = 'beam' ]]; then
            echo -ne $cursor_beam
        fi
    }

    zle-line-init() {
        echo -ne $cursor_beam
    }

    zle -N zle-keymap-select
    zle -N zle-line-init
}

cursor_mode
```

You can now speak about beam and block with passion and verve.

Vim Mapping For Completion

To give Zsh even more of a Vim taste, we can set up the keys `hjkl` to navigate the auto-completion menu.

First, add the following to your `zshrc` :

```
zmodload zsh/complist
bindkey -M menuselect 'h' vi-backward-char
bindkey -M menuselect 'k' vi-up-line-or-history
bindkey -M menuselect 'l' vi-forward-char
bindkey -M menuselect 'j' vi-down-line-or-history
```

We load here the Zsh module `complist`. Modules have functionalities which are not part of the Zsh's core, but they can be loaded on demand. Many different modules are available.

Here, the module `complist` give you access to `menuselect`, to customize the way you can move your completion cursor around.

The command `bindkey -M` bind a key to the command `menuselect`.

Editing Command Lines In Vim

You can use as well your favorite editor to edit Zsh commands:

```
autoload -Uz edit-command-line
zle -N edit-command-line
bindkey -M vicmd v edit-command-line
```

Here, we autoload `edit-command-line`, a function from zshcontrib, a melting pot of contributions from Zsh users.

The function `edit-command-line` let you edit a command line in a text editor. Great! That's what we wanted.

We already saw `bindkey -M`. If you add `vicmd` to it, the keystroke will only work when you're in NORMAL mode (called COMMAND mode in the Zsh documentation). You can find all possible Zsh modes (keymaps) here.

Zsh Plugins

The term "plugin", as I use it, has nothing official. People often speak about Zsh plugins as external pieces of configuration you can add to your own.

There are many of these plugins available for Zsh. Many of them are part of Zsh frameworks.

Zsh Completions

By default, Zsh can auto-complete already many popular CLIs like `cd`, `cp`, `git`, and so on.

The plugin zsh-completions add even more auto-completions. The list of the newly supported CLIs is here

If you don't use any of the program listed, you don't need this plugin.

I added `zsh-completion` as a git submodule in my dotfiles. Then, you can automatically add every auto-completions to your `fpath`, in your zshrc:

```
fpath=(/path/to/my/zsh/plugins/zsh-completions/src $fpath)
```

You don't need to load every completion file, one by one. If you look at the beginning of one of these files, you'll see `compdef`. It's a function from Zsh which load automagically the completion when it's needed. The completion file itself only needs to be included in your `fpath`.

You can as well cherry-pick the specific completions you want.

Zsh Syntax Highlighting

What about syntax highlighting in Zsh? That's what zsh-syntax-highlighting is about.

You can source it directly:

```
source /path/to/my/zsh/plugins/zsh-syntax-highlighting/zsh-syntax-highlighting.zsh
```

You should source this plugin at the bottom of your `zshrc` . Everything loaded before will then be able to use syntax highlighting if needed.

Jumping To A Parent Directory With bd

Do you like to type `cd ../../..` to come back to the great-grand-parent of the current folder?

Me neither.

It's where bd can help you. Imagine that you're in the folder `~/a/b/c/d` . You can jump directly to `a` with the command `bd a` .

The Zsh auto-completion is even included. Awesomeness!

To use it, you need to source the file bd.zsh.

I created in my config a file bd, which source the file `bd.zsh` . Then, I can autoload `bd` in my `zshrc` : `autoload -Uz bd; bd` . It's the same technique used by Zsh Syntax Highlighting.

Custom Scripts

Using a shell allows you to automate many parts of your workflow with shell scripts. That's a huge benefit you should take advantage of.

I keep most of my scripts in one file and I document them (roughly) for me to remember what's in there, and for others to get inspired.

I source the functions in my `.zshrc` , but you could autoload them too.

While working, ask yourself what tasks you do again and again, to automate them as much as you can. This is the real power of the shell, and it will make your whole workflow more fun.

External Programs

A shell without CLIs would be useless. Here are my personal favorites to expand Zsh functionalities.

Multiplex Your Zsh With tmux

I've already written about tmux here. It's a terminal multiplexer with a ton of functionalities.

It allows you to split your terminal in different shells, in different windows, synchronize them, and much more. You can even extend it with plugins, which can automate tmux itself.

Fuzzy Search With fzf

The fuzzy finder `fzf` is a fast and powerful tool. You can use it to search anything you want, like a file, an entry in your command line history, or a specific git commit message.

I wrote (or copied and pasted) a bunch of scripts using zsh too, to search through git logs or `tmuxp` projects.

There are different ways to install `fzf` . You'll need first the executable. Then, I would recommend sourcing the files:

- `key-bindings.zsh` , which will include some practical keystrokes like `Ctrl-h` or `Ctrl-t`
- `completion.zsh` , for `fzf` auto-completion.

If you use Arch Linux, you'll need to install the package `fzf` and simply source these two files in your `zshrc` :

```
source /usr/share/fzf/completion.zsh
source /usr/share/fzf/key-bindings.zsh
```

Otherwise, you'll need to follow the installation process from fzf's README file.

The Z-Shell Is Now Yours

You should now have a clean and lean Zsh configuration, and you should understand enough of it to customize it.

What did we learn with this article?

- Zsh reads its configuration files in a precise order.
- You can set (or unset) many Zsh options depending on your needs.
- The completion system of Zsh is one of its best feature.
- Zsh directory stack allow you to jump easily in directories you've already visited.
- If you like Vim, Zsh allows you to use keystrokes from the Vim world. You can even edit your commands directly in Vim.
- External plugins can be found on The Internet, to improve even further the Zsh experience.
- You should go crazy on shell scripting, to automate your workflow as much as you can.
- External programs can enhance your experience with the shell, like `tmux` or `fzf` .

All your colleagues will be jealous. Guaranteed.

{{}} * Zsh official documentation * Awesome Zsh plugins * Profiling Zsh {{}}

Vim

“Vim is not for me!”

I heard this sentence many, many times, when my colleagues dare to look at my screen. Indeed, I’m doing everything in Vim, if it involves writing or coding.

Of course, I explain to them that learning the very basics of Vim can be very beneficial:

- They could run Vim everywhere.
- They could customize it as they need.
- They could edit easily files on remote systems or in docker containers when only Vi (the ancestor of Vim) or Vim are available.
- Many CLI use Vim-like key bindings. Learning the very basics of Vim can help you navigate in many other CLI application, like Less for example.
- Vim can edit very large text files without slowing down. Log files, anyone? In general, Vim is crazy fast.
- Everything they learn in Vim won’t change from version to version, contrary to the majority of code editors or IDEs. How many times did I screamed to the sky, because the huge preference menu in PHPStorm totally changed?
- They would learn a new and really *fun* way for creating and editing any content. This is the main reason why I use Vim so much. It’s basically the gamification of typing.

Still, they often don’t even try to understand what Vim is about. How can somebody judge without *seriously* trying?

I can’t blame them: I had exactly the same bias, not long ago. However, when I tried to learn to use Vim, when I tried to understand how it works (not only learning two shortcuts randomly), I fall in love.

Today, I would simply like to share with you how I learned the very basics of Vim quickly:

- First, we will see that good typing techniques are essential if you want to unleash its full power. Heck, without even using Vim, rock solid typing techniques are very beneficial for any developer out there.
- Second, we will speak about the different Vim modes. This is one of the main reason why Vim is so unique, and so powerful.
- Third, I will give you the basic Vim shortcuts (keystrokes) for you to be efficient as quickly as possible, and some tips to remember them easily. Don’t be afraid: they are very logical and, therefore, easy to remember.

After that, you can speak about Vim with experience and confidence, even if it’s for saying: “naaaah, Vim is not for me!”.

I won’t ask you to replace your IDE with Vim from one day to another. Simply, try to use it to edit some configuration or other text files. Practicing what you will learn in this article is the key for you to really understand how Vim works and why it’s so popular, even decades after its creation.

Now that we clarified some key points, let’s dive into the wonderful world of Vim!

Prerequisites: The Power Is In Your Fingers

When I decided to learn Vim some years ago, I wanted to do it right. Vim allows you to forget your hands typing on your keyboard or grabbing your mouse, and let you really focus on the most important thing: the content you're creating.

The Mouse: Your False Best Friend

One of the advantage of Vim is to let your hands on the keyboard, without the constant need to grab your mouse.

I see you're afraid: your mouse is like your third hand! It's so useful and easy! Why would it be good not to use it?

I'm sorry to disappoint you. Your mouse is a bit like an implant a doctor would have put on your body at a very young age, telling your parent that it's the best device around there to do something on a computer. You like it because you're deeply used to it, for a very long time.

Ask yourself: why on earth, if the mouse was so perfect, your favorite IDE has 341324 *keyboard shortcuts*? Maybe because… using your keyboard is faster? Easier? More efficient? More *comfortable*?

Your mouse is not your best friend. It's just a friend. Your keyboard is the real brother-from-another-mother here. The power comes from it, and Vim is conceptually designed for you to harness and unleash this power.

That's why I deeply believe that before learning Vim, you need to learn the basic techniques how to use your keyboard as efficiently as possible.

The advantages are enormous, even without speaking about Vim:

- You will type faster and more accurately.
- The room for progression (speed and accuracy) will be enormous.
- You won't focus on your keyboard anymore. Not even a bit.

If you already use these techniques, that's great! You can directly go to the next chapter.

Efficient Typing: The Two Rules

We all agree that thinking, for a developer, is more important than knowing how to type. That said, it's still nice to feel in control of your tools, and obviously, the keyboard is one of the most important!

It's very fulfilling to see your typing improving day after days, months after months, even years after years. The room for progression is huge, even if it's pretty quick and easy to learn the basics.

The first rule you need to learn is placing your hand correctly. Since an image is sometimes better than a flow of words:

Source

The keys `a`, `s`, `d`, `f` and `j`, `k`, `l`, `;` are called the *row keys*. They are the starting points for your hands. From there, you'll be able to grab any other key as efficiently as possible.

You'll notice that there are little bumps on the `f` and `j` keys on your keyboard: they are indicators for you to know where to put your indexes, without looking. When your indexes are at the good position, simply place the other fingers on the other row keys.

The second rule you need to train for is to try not looking at your keyboard, while you're typing. Of course, if you don't remember where a key is, look at it, but only after trying blindly where you think it is. Even if you have no idea where this damn key is. You will sometimes surprise yourself.

Before following these two rules, I was only typing with two fingers. It felt totally foreign to use these new hand placement. Now, I could not type differently: it's more efficient and more comfortable.

Don't be afraid! It's not hard to learn good typing techniques, I promise. It took me one to two weeks to really get it.

The First Week

When you decide to use the two rules I described above, you need to try to follow them *all the time*. We need 100% commitment here. If you surprise yourself using your bad technique again, which will happen, don't worry: simply come back to the good ones. This is part of the learning process, not a horrible failure.

The first three days are the most difficult. You will alternate between good and bad technique without even noticing it. You will do mistakes, and you will be slow.

Fortunately, at the end of the week, the amount of mistake you will make will decrease, and you will less and less need to watch your keyboards.

The Second Week

You will notice during the second week the amount of mistakes decreasing even more, and your need to watch your keyboard will disappear.

At the end of the week, you will see your typing speed already improving, compared to your old way of typing. The good feelings of reward will begin to please your brain.

Speed and Accuracy

During your two weeks of initial training, you shouldn't focus on speed or even accuracy. Just type, as much as you can, and don't worry about anything else yet, not even the mistakes you make.

After that, you can focus on speed and accuracy: how fast you can type, making as fewer mistakes as possible.

To train these good typing techniques, from the beginning of your learning experience to the end of your life, you can use typing software which can drastically help you.

Here's a list of my favorite ones:

- Type Racer
- Online Typing Test WPM
- Speed Coder

Vim Or Neovim?

{{< picture src="/images/2019/vim-beginner/vim-neovim.jpg" webp="/images/2019/vim-beginner/vim-neovim.webp" alt="Vim or Neovim? Neovim!">}}

Now that we know the fundamentals for an efficient typing, let's install Neovim.

Neovim? What's this new weird thing, you might rightfully ask?

Neovim is a refactor of Vim and, therefore, is compatible with everything Vim related. I would definitely recommend using it, instead of the regular Vim, since it's optimized out of the box.

You can choose to use Vim as well, but bear in mind you might need to compile it with the options you want.

Here are the official links for both software:

- Neovim
- Vim

Since Neovim and Vim are *almost* identical (with a different philosophy), I will continue to call these two software using the generic term *Vim*, whatever you're using.

Vim's Configuration

One of the main advantage of Vim is its configuration. Everything is configurable. It's insane, I tell you. You can shape your editor according to your specific needs. How great is that?

For Neovim, the configuration file should be there: `~/.config/nvim/init.vim` .

Let's open it with your favorite editor (which might be soon replaced by Vim!) and let's add some basics things. I will explain later the purpose of it:

```
noremap <Up> <Nop>
noremap <Down> <Nop>
noremap <Left> <Nop>
noremap <Right> <Nop>
```

and

```
set clipboard+=unnamedplus
```

You can use in Vim's configuration a whole programming language, called `Vimscript` , in order to shape your editor. When I was saying that Vim can adapt to your *specific* needs, I wasn't lying.

Don't worry though, you don't need to learn `Vimscript` in order to use Vim.

Now, let's launch Vim and, without trying to do anything else, let's see the basics you need for you to get started.

First Step: The Modes

{{< picture src="/images/2019/vim-beginner/normal-mode-power.jpg" webp="/images/2019/vim-beginner/normal-mode-power.webp" alt="Vim's normal mode will give you the power">}}

After launching Vim, you will see a welcome screen displaying the very basic commands you can use. This screen will disappear as soon as you begin to type some content.

Vim is not like many editors with GUI, where you can simply type on your keyboard and your content will magically appear on your screen. Try to type `x` , for example: nothing seems to happen.

This is because Vim has *modes*: one mode is for inserting some content, another mode is for navigating and editing the same content. These modes are called respectively `insert` mode and `normal` mode.

This section is meant for you to understand the most important Vim modes. You will understand them better when you will try the main Vim keystrokes by yourself.

Normal Mode

You're currently in `normal` mode, and, in this mode, you can't insert any content. However, you can use keystrokes to move where you want and edit what you want: inserting, changing or deleting a word, sentence or even paragraph. Without using your mouse once!

Think of it as a system of shortcuts which allows you to target exactly what you want to edit. Shortcuts are very efficient, even GUI editors or IDE allow you to use many of them, to improves your comfort and your efficiency.

The difference between your default GUI editor and Vim at the shortcut level is significant: the shortcuts in Vim make sense, most of the time. That's why it's so easy to learn Vim. As you will see, it's easy to remember these keystrokes since they follow a certain logic.

For example, the shortcut `ctrl+shift+n` to find a file in your favorite GUI editor is difficult to remember because it's difficult to link what you know (opening a file) and what you want to learn (the keystroke).

We will come back to the `normal` mode later, when we will learn the main Vim's keystrokes.

Insert Mode

Look at your cursor in Vim: it should normally be a square. Now, let's type our first `normal` mode keystroke: `i` . Bam! Magical things appear! Your cursor becomes a pipe, or a line. It's impressive, I know. You can see as well that in the bottom left corner, the indicator `--INSERT--` appeared. My friend, welcome to the `insert` mode!

{{< picture src="/images/2019/vim-beginner/vim-insert-mode.jpg" webp="/images/2019/vim-beginner/vim-insert-mode.webp" alt="The default mode of Vim: Normal Mode">}}

The keystroke `i` means `i` nsert. That's what I meant when I was saying that the shortcuts in Vim make sense. They are simple, too!

In `insert` mode, you can finally type your content: go ahead, type anything you want. To come back to `normal` mode and stop inserting, simply type `esc` or `ctrl-c` . The cursor becomes a square again, and the `--INSERT--` indicator disappears.

That's how you work with Vim: juggling between `normal` mode to place your cursor whenever you want and edit your content, and `insert` mode in order to insert some *new* content!

Visual Mode

There is a third important mode in Vim you will use often: `visual` mode. Its goal? Selecting your content. From there, you can modify, edit or copy your selection.

To enter `visual` mode, you might guess it already, you need to type `v` in `normal` mode. You will see the indicator `--VISUAL--` appearing at the bottom left corner of your Vim instance.

Second Step: The Basics Keystrokes

{{< picture src="/images/2019/vim-beginner/vim-book.jpg" webp="/images/2019/vim-beginner/vim-book.webp" alt="Some books are very good to learn Vim">}}

Now that we understand the general concept of Vim and its main modes, let's see the most important keystrokes in `normal` mode you need to be aware of. I encourage you to try them in Vim as you read.

Many of these keystrokes, especially the movement keystrokes, work in `visual` mode as well.

Writing Your Own Cheatsheet

You can find countless cheatsheets on the Internet which will give you as many Vim keystrokes as you want. When I was learning how to use Vim, I used one of them.

At the same time, I was writing as well *my own cheatsheet*.

Why? Personally, it helped me a lot memorizing these keystrokes. Writing is a powerful way to make the information yours.

Organize your cheatsheet as you like, and use whatever you want (paper, evernote, mindmaps…). For mine, I'm using Joplin, a free note taking system.

Bonus points if you write your cheatsheet… in Vim!

The rest of the article will give you the basics keystroke you need for using Vim. Be aware that Vim is full of subtleties which give you power and efficiency, depending on the context. It's a never ending learning process which makes this editor so interesting, so fun to use and highly rewarding!

As a friend of mine said once: "it's basically the gamification of writing!"

The ex command

You can compare the ex command as a menu on a GUI. A big and powerful menu.

If you type `:` in `normal` mode, your cursor will end up automatically at the bottom of Vim. From there, you can type any command.

Here are the most basic ones:

- `:help` to access Vim's help. This help is insanely complete. If you don't remember how to quit Vim for example, you can type `:help quit` .
- `:q` to `q` uit Vim (`:q!` to quit without saving. Imagine that you yell at your editor you want to quit, whatever the consequences!).
- `:w` to `w` rite (save the current file open).
- You can even combine some ex command: `:wq` to write and quit.
- `:e <path>` to edit a file. The path can be absolute or relative.

Searching

I wrote a whole article about searching in Vim, but, for now, the keystroke `/` should be enough. If you use it, you will go again at the bottom of the Vim's instance. From there, type your search and press `enter` .

You can go to the `n` ext found occurrence by typing `n` . To go to the previous one, simply use `N` .

Forget the arrow keys, embrace hjkl

To be totally honest with you, this was the hardest part for me: not using the arrow keys to move your cursor.

As I said in the first part of the article, your fingers should be on the row keys. First, for your typing to improve, and second because the Vim's keystrokes you can use in `normal` mode are especially designed for you to follow the good typing technique. In short, they are all around the row keys, to prevent your hands to move too much. Only your fingers should.

If you follow the typing techniques we saw above, using Vim `normal` mode will be rewarding and efficient.

Now, try to reach the arrow keys from the row keys: yes, you need to move your hand! This is definitely not what we want.

That's why, instead of using the arrow keys, you should use the keys `h` , `j` , `k` and `l` to move respectively left, down, up and right.

It's difficult at first: you will try to use the arrow keys to move your cursor, and more than once. That's why we previously disabled them in the configuration!

How to remember what does `h` , `j` , `k` and `l` ?

- `h` moves your cursor to the left, and `l` moves it to the right. It makes sens, since `h` is on the left of the sequence `hjkl` , and `l` is on the right.
- `j` moves your cursor down. You can remember it since `j` looks like an arrow which points down (with a bit of imagination). Another mnemonic method: the key `j` has a little bump on the bottom of the key, which means the cursor will go down.
- `k` is the only letter left, so it has to go up. I always imagine a Ninja Turtle jumping, saying "Kowabunga"! It's not even the good spelling (it would be "Cowabunga") but it works for me. Please, don't judge me.

I see your mind full of questions. Fear not, dear reader! I've got you covered for this one, with a revolutionary AAA game everybody will speak about in twenty years. A snake game I made, where you *must* use `hjkl`.

Insert Mode

We saw previously that the keystroke `i` moved your soul in the reassuring world of `insert` mode.

There are other handy keystrokes to do so, introducing welcome subtleties:

- `i` for `i` nserting content before the current character.
- `a` for inserting content `a` fter the current character.
- `A` for inserting content `A` fter everything. It will move your cursor to the end of the line and enter `insert` mode.
- `o` `o` pen a new line, below the current one, and allow you to insert your content.
- `O` `O` pen a new line above the current one.
- `r` `r` eplace one character by another one.
- `esc` and `ctrl-c` will bring you back to `normal` mode, if you are in `insert` mode.

Moving Horizontally

Moving is important: after all, targeting what you want to change is one of the main goal of the `normal` mode. Here's how to move on a line:

- `w` move forward your cursor to the next `w` ord
- `b` move `b` ack your cursor to the previous word
- `0` will move your cursor to the beginning of the current line.
- `^` will move your cursor to the first non blank character on the current line.
- `$` will move your cursor to the end of the current line.

Moving Vertically

You can move vertically simply by searching the word you want to move on (see above to learn how to search). There are other ways to move vertically, though:

- `ctrl-u` move your cursor `u` pward half a screen.
- `ctrl-d` move your cursor `d` ownward half a screen.
- `gg` will `g` o `g` o (go go?) at the beginning of the document.
- `G` will `G` o at the end of the document

Undo and Redo

What would we do without the essential undo and redo?

- `u` will `u` ndo the last modifications you did.
- `ctrl-r` will `r` edo. Since `r` is for `r` eplacing, we need to use `ctrl-r` for `r` edo. You can thing of it as you being in `ctrl` of your content.

Third step: the Keystroke Language

{{< picture src="/images/2019/vim-beginner/vim-language.jpg" webp="/images/2019/vim-beginner/vim-language.webp" alt="Vim as a very powerful keystroke-language!">}}

In Vim, some keystroke can be combined to form sentences, describing an action. I know, it sounds weird, but it's brilliant: it will help you tremendously to remember all these keystrokes.

These sentences are so common that you will associate easily what you know already (the sentence) by what you need to learn (the keystrokes).

Even better: knowing that Vim has a keystroke language will push you to combine them instinctively to do what you need to do, and, in many cases, it will work!

It's magic, I tell you.

Let see some basic keystrokes which *need* to be combined:

- `d` for `d` elete
- `c` for `c` hange

With these operators, you can act on text objects. Simply put, a text object is a set of character. In Vim, a `word` is a text object, as well as a `sentence` or a `paragraph` .

Here are some combined keystrokes I use all the time:

- `diw` will `d` elete `i` nside the `w` ord. It will delete the current word under the cursor.
- `ciw` will `c` hange `i` nside the `w` ord. It will delete the current word under the cursor and switch to `insert` mode. In short, you… change the word!

You can try to *change a word* or *delete a word*, it works as well and introduce some subtleties. I let you find what could be the keystrokes for these!

You can as well use `y` (`y` ank) as a basic combination keystroke, in order to copy some text. For example, you case use `yiw` for `y` ank `i` nside `w` ord, which will copy the word under the cursor.

To past, simply use `p` or `P` .

Finally, you can combine these keystrokes with some movements.

For example:

- `d$` will `d` elete from your cursor to the end of line. You can use as well the alias `D` .
- `dgg` will `d` elete everything from the cursor to the beginning of the file.
- `ggdG` will delete everything in the file!

That's all! With these keystrokes, you should already be able to do whatever you want in Vim! Be creative, try as many keystroke combinations as you like.

What's Next?

Congratulations! Now, you know the very basic of Vim.

This article, combined with a good cheatsheet will already bring you pretty far on the road of productivity.

You might think: "great, and what's the big deal?".

Vim has countless more tricks under its belt. As I was saying above, you can code its behavior using `Vimscript` but you can as well use plugins created by the community, for any need you have. If you miss something not available in vanilla Vim, a simple search on the Internet will give you, most of the time, the plugin(s) you need!

However, I would recommend to use Vim's help before installing millions of plugins. Remember: by default, Vim can already do *a lot*.

To summarize what we learned with this article:

- Having proper typing techniques will make you faster, more accurate and will allow you to focus solely on your code, not on your keyboard or on your hands. Even more importantly, you will feel more in control.

- You need to use the row keys `h` , `j` , `k` and `l` to navigate in Vim, to really appreciate its comfort, its power, and its fun!
- Write your own cheatsheet as you learn new keystrokes, and try to find your own mnemonic to retain them. It will accelerate your learning process. To remember easily, associate what you already know with what you want to learn.
- Try for yourself the keystrokes explained here, for you to memorize them using your muscle memory.

Even if you still don't like using Vim at that point, at least you tried seriously to use it, and you will be able to do so when you don't have any other choice. You can now claim, as a man full of experience, that you don't like Vim... even if you tried to understand it!

Now, if you want to continue getting better at Vim, what's next?

- You should go through `vimtutor` . If you use Neovim, you just have to execute the ex command `:Tuto` . If you use Vim, simply type `vimtutor` in your shell. It will remind you some keystrokes from this article, and will teach you even more.
- Vim is a game: the goal is to use as few keystrokes as you can to accomplish what you want to do. In that spirit, Vim Golf can be pretty fun.
- I would heavily recommend reading the book Practical Vim, from Drew Neil. You will learn a lot from it.
- Look at the related resources at the end of the article: they will improve your Vim skills drastically.

From that point, you should have enough material to become a Vim Master!