Vim For Developers: Practical Introduction Into Vim

Yanis Triandafilov

Abstract

In this book we are focusing on configuring our Vim from scratch in order to build a modern, fully-capable IDE on top of it. It is written for people who would like to try Vim out but don’t know how to start and are scared or disappointed by the lack of modern IDE-like features essential for a productive development workflow.

Contents

Please send the comments, ideas, and mistakes you find to me: janis.sci@gmail.com.

Preface: Why Learn Vim in 2023?

Vim has been around for some time (the first public version was released in 1991). Why is it still gaining traction now when there are all sorts of editors and smart IDEs for any language on the market?

There are several reasons for this:

- **Editing speed**. Vim modes provide a unique editing experience optimized for keyboard-only use and are thus very efficient. More on that later.
- **Vim is very lightweight**. The program starts in just a few milliseconds. If you run Vim and open your process inspector, you'll notice it only takes up several megabytes of RAM, not gigabytes like some popular IDEs.
- **It's extensible**. There are thousands of different Vim plugins, which can give you almost anything a modern IDE would need, e.g., code completion, fuzzy file search, jumps between definitions, etc.
- **Close to the OS**. Vim is just one tool of many you can use to perfect your development experience. It integrates nicely with other terminal tools. For example, you can easily sort lines in your editor using `sort` on any Unix machine.
- **Language-agnostic**. You can set up your Vim to work with JavaScript, TypeScript, C++, Ruby, or Haskell — no need to keep a separate IDE for each language.
- **Vim is bottomless**. Every now and then, you get to discover some amazing features. It's crazy how many useful features there are and how deeply they're thought out.

So why is Vim considered hard to learn?

Vim is usually thought of as something with quite a steep learning curve. Some concepts are absent in popular editors and may feel weird in the beginning (for example, modes). But you will soon realize that there is nothing scary about those things and that they are, in fact, very logical and convenient to use.

Figure 1: Example of Vim setup with TypeScript autocompletion

Vim may look less impressive at first glance. New users expecting to see modern IDE features are bound to be disappointed. Newbies are both scared of unfamiliar concepts and disappointed by the lack of essential features for productive dev work.

Those were my first impressions as well.

With this book, I decided to try something new. Yes, we'll still walk through the basics and learn all the common Vim concepts and idioms. But at the same time, we will focus on building a fully capable modern IDE on top of Vim by extending the configuration (`.vimrc` / `init.vim`) and adding all the necessary plugins as we go.

Every new chapter of this book will both teach you about the "Vim way" and will also introduce some configurations and plugins that you can start using right now with immediate effects on productivity.

By the end of this book, you will have a good understanding of how Vim works and be able to build a fully-fledged, modern IDE-like experience on top of it.

Introduction

In this chapter we'll learn how to install Vim, what kinds of Vims are out there, and why it's important to start configuring it from the very beginning.

The Flavors of Vim

Vim itself is an upgrade of an older editor called Vi (Vim stands for "Vi IMproved"). Nowadays there are even more options that are based on Vim or use similar concepts:

- Vim which got the background processing capabilities with the latest (eighth) release
- MacVim provides a more pleasant UI (on OSX)
- Neovim provides some additional features such as background processing, and a built-in terminal.
- Oni, a full IDE with all the bells and whistles based on Neovim.
- There are preconfigured bundles (I wouldn't recommend it for educational purpose): janus, spacevim
- Almost all modern editors and IDEs have a "Vim mode" (so you can use your favorite IDE and still enjoy the basic Vim keybindings and modes).

Those different versions don't really differ from each other all that much. The main differences are found in the default settings and the location of the configuration file, plus some miscellaneous custom tweaks. Though, NeoVim and Vim diverge from each other quite a lot (with introductory of Lua and Vim9 Script).

For the purpose of this book we'll be working with regular **Vim**. Still, 99% of what you'll learn is compatible with any other version.

Configuration

The power of Vim comes from its flexible configuration. Clean Vim is not very useful out of the box. It requires a little bit of tweaking. It is

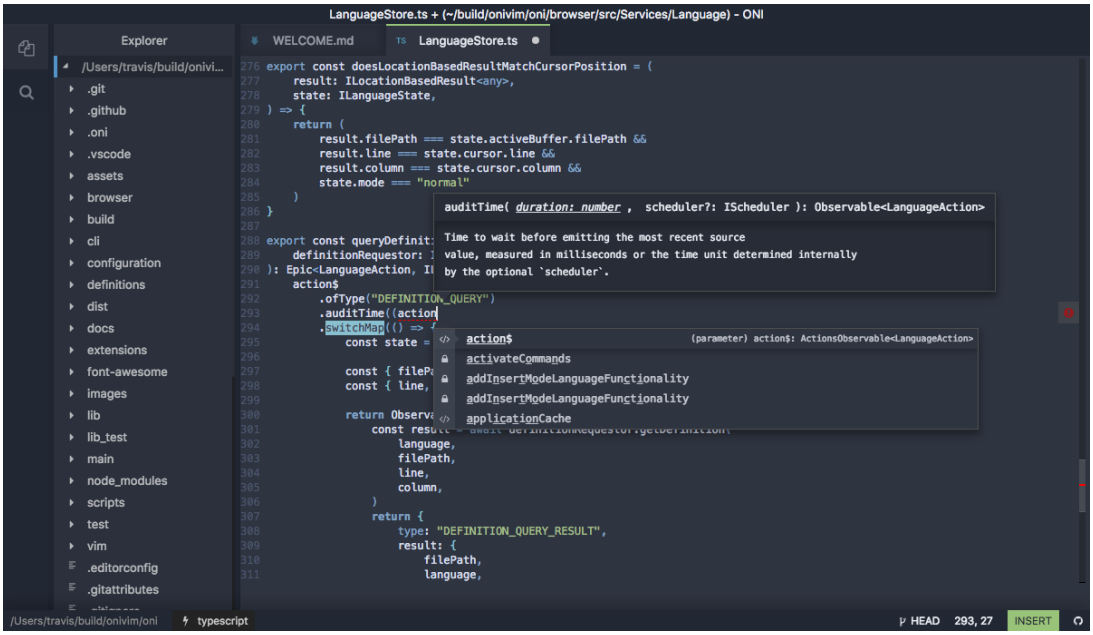

Figure 2: Oni - Beautiful UI on top of Neovim

absolutely essential to know how to do this, otherwise it doesn't make much sense to use Vim at all.

Vim's configuration is stored in a special file (`~/.vimrc`). Upon installation, that file won't exist yet. So we will need to create it first, and then during the course of this book, we'll be adding different settings onto it, one by one. And our modest editor will gradually transform into a beautiful and powerful IDE.

Installation

Installing Vim is a very straightforward process. If you're on MacOS, I recommend using brew, a package manager for... well, everything.

```
# Install Vim
brew install vim


# Check the version
vim --version
# > VIM - Vi IMproved 9.0 (2022 Jun 28n
# > ...
```

Installing Vim on Debian or Ubuntu is equally easy:

```
# Debian
sudo apt-get install vim


# Ubuntu
sudo apt install vim
```

If you work with some other platform, please refer to this official guide.

Now let’s learn the basics.

Chapter 1. The Basics

In this chapter, you'll learn how to open and navigate files, where to get help, and will finally understand modes. If it feels a bit dense, don't worry. We'll recap most of the concepts in the later chapters in more detail.

Editing Files

First things first. In order to edit a file type

```
vim src/App.ts
vim src/App.ts src/css/styles/app.css  # to open multiple files
```

When you're done, press `Esc` to make sure you're in the **normal mode** (more on modes later). Then type `:wq` to write the changes and quit or `:q!` to exit without saving. Press enter.

Now you know how to exit Vim.

Alternatively,

- to quit without saving you could press `ZQ`, or
- `ZZ` to write the current file and quit.

All key combinations are mnemonic.

For example, `wq` is easy to remember as `(w)rite + (q)uit`. If you have multiple buffers and/or tabs open and you want to quit and save all of them, then you can type `:wqa`, which stands for `(w)rite + (q)uit + (a)ll`.

This mnemonic nature of operators and commands is important for remembering all those combinations and helping your muscle memory, so it worths paying attention.

Getting help

Before we went too far, let's talk about getting help.

Vim comes with a quite comprehensive help manual. And when I say comprehensive, I mean you don't even need to google what you're looking for.

Instead you just go the command-line mode and type `:help <query>` .

Help pages are organized with keywords, and are connected with each other forming a help network.

Try something like:

- `:help tabs` to learn about tabs in Vim
- `:help navigation` to learn about motions
- `:help help` to get meta
- `:help 42` for an intriguing Easter egg

Remember, when you type `:` in **normal** mode, you cursor is moved into the bottom part of the screen. There you can continue typing the command.

Here are some tips to work with Vim help:

- Instead of `:help tabs` you can use a shorcut `:h tabs`
- Auto-completion. After you type `:help com` you can continuously press `Tab` to switch between different topics starting with `com` .
- Help pages are connected with each other. If you see a highlighted word, that's a link that you can put a cursor on and press `K` or `Ctrl + ]` . `Ctrl + T` will get you back.

Vim help is very organized and always up-to-date. It should be your first place to look for information. That is, after StackOverflow. Just kidding.

How to rollback changes

When you screw things up, how to fix it?

- `u` is short for "undo". Press it in normal mode to revert the last

operation (this is kinda like `CTR/CMD+Z`). Press it multiple times until things are back to normal.
- `<C+r>` (Ctrl + R) to revert the undo (like `Ctr+Y`).
- `Esc`, `:qa!` Close all the files without saving. `Esc` to get to normal mode unless you're already there.

Modes

When you open Vim, you start in **NORMAL** mode

Normal mode is optimized for navigation and text manipulations other than typing (copy-pasting, for example). This is why, if you try to type something right away, you will probably be confused by whatever is happening on the screen.

If you want to start typing, you first need to switch to **INSERT** mode. In this mode, you would probably feel as in any other editor/IDE. When you type keys you will actually see them appearing on the screen.

How to get from NORMAL to INSERT mode?

There are many ways, for example

- `i` to simply start typing from where your cursor is (i = "insert")
- `a` to start typing from the next character (a = "append")
- `I` to start typing from the beginning of the current line
- `A` from the end of the current line

When you finished typing, press `Esc` to get back to the normal mode. `Esc` will get you back to the NORMAL mode from any other mode. `Esc` is your friend.

There is also the **VISUAL** mode, which is there to help you select a piece of text; for example, if you want to copy it into the buffer or delete the entire thing.

- `v` to start selecting in visual mode (v = visual)
- `V` to select the whole line

Then, while in visual mode with some text selected, you can type `d`

to delete it and get back to NORMAL mode.

There are other modes as well, but for now let's focus on those three.

Basic Navigation

OK, let's learn to navigate our file.

To move your cursor one character at a time you can press:

- `h` move left
- `j` move down
- `k` move up
- `l` move right

Remembering those on the muscle level wasn't easy. It really takes some time and patience. However, when you finally learn, it feels so natural there's no way you ever go back to arrows again.

There is one thing that helped me learn faster using the HJKL keys: a piece of configuration that remaps the arrow keys so they simply stop working. This will force you to use HJKL instead.

Configuration time: disabling arrow keys

Remember, I told you that this book is focused on configuration. Well, let's not waste any more time and start doing that!

Vim configuration lives in `~/.vimrc` file.

By default it doesn't exist. But this is ok, we can still open it for editing, and Vim will create the file once we finish.

```
vim ~/.vimrc
```

Now go into the INSERT mode (by pressing `i`) and type:

```
" Prevent user from using arrow keys
noremap <Up> <NOP>
noremap <Down> <NOP>
```

```
noremap <Left> <NOP>
noremap <Right> <NOP>
```

As an alternative, you can copy those four lines into the buffer and insert them with `Ctrl/Cmd + V`. You still need to be in insert mode in order to do this.

Now press `Esc` to get back to Normal mode, then save the file and exit with `:wq`.

That's it.

Now you can open some other file with `vim` again and try pressing arrow keys. They don't work anymore.

How to update Vim configuration?

An alternative to closing and reopening Vim to re-read the settings is to run `:source $MYVIMRC` in the **command-line** mode.

`:source` is a command that can read any Vim script file and execute it right away. `$MYVIMRC` is a special variable that refers to your configuration file.

Copying and pasting in Vim

Copy-pasting in Vim is quite simple, though it might surprise you a little when you don't know how it works inside.

If you want to copy text you first select it in visual mode and then you press `y` ("yank"). That will put that piece of text in a special buffer, in Vim terms, a **register**. Then in normal mode you can press `p` ("put" or "paste") to paste it under the cursor.

Instead of `y` you can use `d` ("delete"), which works like "cut" - it will delete the text but will also put it into the buffer, then you can paste it somewhere else.

Key Takeaways

Phew! It was a lot, but we did it.

Here's what we've learned:

- To open a file `vim <filename>`
- To save and quit Vim, press `Esc`, then `:wq` or `:wqa`
- To abandon the changes: `q!` and `:qa!`
- To get help `:help <whatever you're looking for>`
- To rollback the changes `u` and `Ctrl+R` to undo
- Three important modes are NORMAL (the default one), INSERT (for typing text), and VISUAL (for selecting text)
- Teach yourself to navigate with `h`, `j`, `k`, `l`
- Copy text with `y`, cut with `d`, and paste with `p`.

Chapter 2. Deeper Dive

In this chapter, we are going to learn a little but more about **the Vim way** and continue working on our configuration.

Cursor Motions And Operators

Here's a very cornerstone of Vim editing.

You can combine an **operator** (like `d` - delete) and a **motion** (like `$` - that moves cursor to the end of line).

The result is that this operator is applied to the chunk of text described by the motion (in this case it removes text from the current position till the end of line).

Let's start with motions.

Cursor Motions (or how to navigate around)

We've already met with HJKL motions that allows us to navigate text one character at a time.

Here are some more motions. Try each one, and try to get confortable with them:

- `h`, `j`, `k`, `l`
- `w` move to the beginning of the next word /
- `b` move to the start of the current word
- `(` / `)` move to the next/previous sentence
- `{` / `}` move to the next/previous paragraph
- `G` go to the last line of the file
- `<number>G` go to line number
- `gg` to the beginning of file
- `$` jump to the end of the line (`g_` - to the last non-blank)
- `^` jump to the first non-blank character of the line (`0` - to the beginning)
- (double back-tick) will get you to the previous position

Normally, when I open a file, I start by pressing `}` multiple times, to get to the place I'm interested in, then pressing `w` multiple times to navigate within the row, jumping from one word to another, and then `h` / `l` to navigate by characters.

A good idea might be write those down and keep at sight when working. Getting used to those takes some time, but eventually pays off.

You can also jump to a specific character:

- `f{char}` puts your cursor on the first occurrence of {char} to the right
- `F{char}` same thing, but in the opposite direction
- `t{char}` puts your cursor just one character before the first occurrence of {char} to the right
- `T{char}` same thing, but in the opposite direction
- `;` is a special motion that repeats the latest motion

You can also combine motions with numbers; for example, you can type `7w` to skip 7 words or `5j` to go 5 lines down.

Using motions with operators

As said before, you can combine those motions with text operators and numbers in order to achieve your goal. For example,

- `d$` deletes the text from the current character until the end of the line
- `5x` deletes five characters
- `dG` removes everything from the current line till the end of the file

Text Objects (:help text-objects)

There are a number of motions called "text object selections" that can be only used after an operator or in visual mode but not on their own.

- `aw` / `iw` stand for "a word" and "inner word"
- `ap` / `ip` a paragraph / inner paragraph

There are many more, but you get the idea. Prepend a text object with `a` or `i` to be able to do things like:

- `daw` delete the entire word including spaces after it
- `yi{` copy into buffer everything inside the {} block
- `ci"` remove text and start typing within quotes
- `dat` delete the whole tag (as in HTML tag) and the spaces after it

Don't forget, to revert your changes, press `u` (undo). Vim undo history is extremely powerful, but we'll talk about it a bit later.

Here's another example. Imagine you have a JavaScript function:

```
function hello() {
  // some code here
}
```

You need to replace the body of that function. You just need three keys for that `ci}` which reads as "change inner } - curly brackets". Change operator works the same way as delete, but it also leaves you in the INSERT mode so you can start typing right away.

The **Vim way** is to combine motions and operators whenever possible. A Vimmer would never type `x`, `x`, `x`, `x`, `x` in order to delete a word with five characters. They will press `5x` instead, or even better, `daw`.

It is your most important task to learn using those combinations as this is pretty much the essence of Vim.

Phew, there's a lot of information to digest.

You might wanna break here, and go get some coffee. And then let's talk about something else.

Plugins

As we are slowly moving forward, we want our Vim to become more useful, and in order to do that we will probably need some help.

Plugins are very helpful for anything Vim doesn't have out of the box. Syntax highlighting? Auto-completion? Snippets? File manager? There's a plugin for that!

Let's start by installing a plugin manager. There are several out there and they work more or less the same way. I prefer vim-plug which is very minimalist and it "just works".

We need to run a little script to install it.

```
curl -fLo ~/.vim/autoload/plug.vim --create-dirs \
    https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim
```

Now let's add our first plugin. Open the configuration file and put at the very beginning of the file, before anything else:

```
" ===========
" Plugins
" ===========
call plug#begin('~/.vim/plugged')

" Collection of color schemes
Plug 'rafi/awesome-vim-colorschemes'

call plug#end()

" Use the colorscheme
colorscheme OceanicNext
```

Now reload the configuration with `:source $MYVIMRC` and run `:PlugInstall`. It will install the plugins.

Reload the configuration again, and you should see the color theme changed.

The `:colorscheme` command changes the color theme, and the plugin we installed is just a bunch of different themes. If you don't like this particular theme, then type `:colorscheme` and then press Tab. By pressing tabs multiple times you should be able to loop through themes

so you can try different ones.

Whenever you want to update your plugins, there's a command for that too. `:PlugUpdate` . `:PlugClean` removes the plugins you don't use anymore from the file system.

OK, so (a) we have a plugin manager now, and (b) our Vim finally looks nice.

Plugin time: more text objects

Now that you know how to install the plugins, let's practice a little bit.

There is a very useful one which adds more text objects. wellle/targets.vim. You can install it by adding it to plugins sections.

Now you can use text objects like:

- `cin)` - change in the next parentheses
- `da,` - delete a comma separated item from the list

Learn more about the plugin here.

The dot operator (.)

Part of the magic of Vim is the amazing **dot operator**. When you press `.` (dot) in the normal mode, it will simply repeat the latest command, whatever it was. A command can be anything you did in normal mode.

Copy-pasting and Registers

Let's talk about copy/pasting operators a bit more. The first is called `y` (yank), and the other is `p` (paste).

Yanking can be combined with text objects. For example, `yaw` will put the current word into the clipboard. `p` will paste it wherever you type it.

In Vim terminology, a clipboard is called a **register**. Unlinke, the system clipboard, there are many registers in Vim. When yanking a piece of text, you can choose which register to put it into with `"ay` .

In that example, you put the text into the register `a`. In order to paste it, you should prepend the paste command with register `"ap`.

Some registers serve a special purpose. For example, `%` register always stores the name of the current file, `/` register holds the latest used search pattern, etc.

Note that the default Vim register is not the same as the system one. When you copy something from StackOverflow, you will not be able to paste it straight away with `p`. You can still do this like this with `"*p"`. Here, `*` (`:help quotestar`) is a special register that refers to the system clipboard.

I have this command in my settings to use the system register as default:

```
set clipboard^=unnamed   " Use the system register
```

Now your default Vim register refers to the system clipboard, and you don't need to prepend `p` and `y` commands with `"*` anymore.

Plugin time: Peekaboo

With so many registers, sometimes it's easy to get confused. There is a very nice little plugin, https://github.com/junegunn/vim-peekaboo which opens a bar on the right every time you you press `"` and displays which content is stored in each buffer.

The sidebar is automatically closed on a subsequent key stroke.

Key Takeaways

- Motions allows you to navigate the file in the NORMAL mode (`:help motion`)
- Motions can be combined with operators; for example, `dj` will remove the current line and the line below
- Motions can also be combined with text objects; for example, `cip` will delete the paragraph and put you into the INSERT mode
- Plugins is a way to extend Vim's capabilities (`:help plugin`)

- Dot operator `.` repeats the last command (`:help registers`)
- Registers are like little pockets in which you can put some text. Copying and pasting use registers. Vim has a register associated with every letter on the keyboard (and not only letters).