Vim From Scratch: Building An IDE-like Experience

Ianis 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 mostly 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

This book is still in BETA. Please send the comments, ideas and mistakes you find to me: janis.sci@gmail.com.

Change History

0.1

- Initial version

0.2

- Properly separate sections in chapters
- Include the cover
- Update sections
- Change the structure of the book

0.3

- Chapter 3 completely rewritten
- Chapter 4 (Search) added

0.4

- Chapter 6 on formatting and linting is ready

0.5

- Look and feel completely revamped (syntax highlighting added, margins and fonts fixed)
- Added the VimL introduction chapter
- The contents is finalized

0.6

- Section on completion has been added

0.7

- Added the section about Git
- The last chapter - resources, has been vastly reorganised

0.8

- A bonus chapter has been introduced
- So many grammar mistakes fixed

Preface: Why Learn Vim in 2020?

Vim has been around for some time. The first public version was released in 1991, and since then Vim has gotten more popular. Even now, when there are all sorts of editors and very smart IDEs for any language on the market, Vim is still gaining a lot of traction.

In my opinion, there are several reasons for this:

- **Speed**. Vim modes provide a unique editing experience optimized for keyboard-only use and are thus very efficient. More on that later.
- **It's light**. 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, instead of gigabytes as some popular IDEs do.
- **It's extensible**. There are thousands of different Vim plugins out there, 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.
- **Language-agnostic**. You can set up your Vim to work with JavaScript, TypeScript, C++, Ruby or Haskell (you name it), and you don't have to keep a separate IDE for each of the platforms you need.
- **Vim is bottomless**. Every now and then, you get to discover some new piece of utility which helps improve your productivity. It's really crazy how many useful features there are, and how deeply they're thought out.

Vim is usually thought of as something with quite a steep learning curve, mostly because of several key concepts in Vim that will sound unfamiliar to modern developers: modes, or the notion of coding without a mouse (although Vim is really mouse-friendly), and many other

Figure 1: Example of Neovim setup with TypeScript autocompletion

functionalities. You will soon learn that there is nothing scary about those things, and that they are in fact very logical and convenient to use.

Another problem is the fact that Vim may not look particularly 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 keep the focus on building a fully capable modern IDE on top of Vim by extending the configuration (`.vimrc`) file 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 configuration and plugins that you can start using right now with immediate effects on productivity.

This book is supposed to be short and concise and by the end of it you will have a good understanding of how Vim works, and you will be able to build a fully-fledged, modern IDE-like with Vim.

Introduction

In this chapter we'll get to know the very basics of Vim: how to install it, 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 a much older editor called Vi (Vim stands for Vi IMproved). Nowadays there are even more options that are based on Vim or use similar concepts:

- The good old 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.
- Also, there are some preconfigured bundles that you can use to get started (although I wouldn't recommend it): 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 all that much from each other. The main differences are found in the default settings and the location of the configuration file, plus some miscellaneous custom tweaks.

For the purpose of this book we'll be working with Neovim as it seems to be the most popular version now (for a reason), boasting sane default settings. Most of the contents of this book are perfectly compatible with regular Vim, and passages where this is not the case will be indicated with a note. I will be using the words "Vim" and "Neovim" interchangeably.

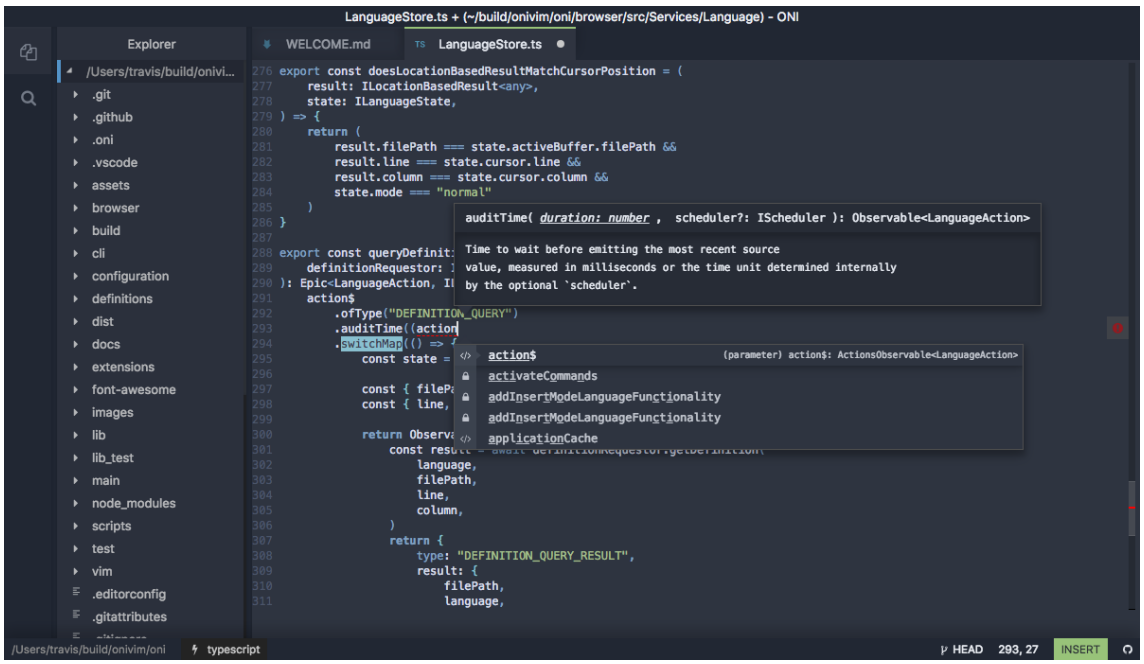

Figure 2: Oni - Beautiful UI on top of Neovim

Configuration

The power of Vim comes from its flexible configuration. As noted earlier, clean Vim is not very useful out of the box. It requires a little bit of tweaking. Tweaking Vim is the very essence of what I like about it. It is absolutely essential to know how to do this because otherwise it doesn't make much sense to use Vim at all.

Vim's configuration is stored in a special file (`~/.vimrc` for Vim and `~/.config/nvim/init.vim` for Neovim). 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.

Installing

Installing (Neo)Vim is a very straightforward process. If you're on MacOS, I recommend to use brew which is a package manager for… well, everything.

```
# Install brew
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/i

# Install Neovim
brew install neovim

# Check the version
nvim --version
# > NVIM v0.3.7
# > ...
```

Installing Neovim on Debian or Ubuntu is equally easy:

```
# Debian
sudo apt-get install neovim

# Ubuntu
sudo apt install neovim
```

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

Now we are ready to learn the basics.

Chapter 1. The Basics

This chapter is about getting comfortable with Vim. You'll learn how to open and navigate files, where to get help, and finally learn about 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 with Neovim just type

```
nvim src/App.ts
nvim src/App.ts src/css/styles/.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.

Boom!

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.

One thing you'll soon start to notice about Vim is that everything is quite 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 as you guessed, 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 to pay attention.

Getting help

Before we go 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 <whatever>`.

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

You can 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

Well, you get the idea.

Here are some tips to work with Vim help:

- There are shortcuts in command-line mode, so instead of `:help tabs` you can type `:h tabs`
- Don't forget about auto-completion. After you type `:help com` you can continuously press `Tab` to switch between different topics.
- Help pages are connected with each other. If you see a highlighted word, that's probably a link that you can put a cursor on and press `K` or `Ctrl + ]` to follow it. `Ctrl + T` will get you back.

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

How to rollback changes

When you think you screw something up, how to fix it?

- `u` is short for “undo”. Press it in normal mode to revert the last operation (this is kinda like CTRL/CMD+Z). Press it multiple times until things are back to normal.
- `<C+r>` (Ctrl + R) to revert the change.
- `Esc` , `:qa!` Close all the files without saving. `Esc` to get to normal mode unless you’re already there.

Modes

When you open Vim, by default you start in NORMAL mode, which 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 get into another mode called INSERT mode. In this mode, you would probably feel as in any other editor/IDE. When you press the keys here you will actually see them appearing on the screen. You can get into the INSERT mode from the normal mode in several 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

There are many other ways to get into the INSERT mode, but we’ll get to them later. When you finish typing, press `Esc` to get back to the normal mode. `Esc` will get you back to the NORMAL mode from any other mode.

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 I'd say they are not as important as these three, so for now, let's focus on them.

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

I have to say, remembering those on the muscle level wasn't easy. It really takes some time and patience. Though when you finally learn, it feels so natural that 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. It forces you to use HJKL instead.

Configuration time: disabling arrow keys

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

Let's add it to our configuration file (`~/.config/nvim/init.vim`).

By default that file doesn't exist, and the directories probably don't exist either. So we need to create the directory first:

```
# -p flag lets us create multiple directories at once
mkdir -p ~/.config/nvim/
```

Open the file (if a file doesn't exist, Vim will create it for you after saving).

```
nvim ~/.config/nvim/init.vim
```

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

```
" Prevent a 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 `nvim` again and try pressing arrow keys. They should not work now.

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 always 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, which in Vim is called "register". Then in normal mode you can press `p` (put or paste, if you like it more) in order 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 buffer, so you can paste it somewhere else later.

There is so much more to grasp here about the registers and how they work, but I will save it for the later chapters.

Key Takeaways

- To open a file `nvim <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 of doing things and also will continue to work on our configuration.

Cursor Motions And Operators

One of the very powerful core concepts of Vim that allows for many wonderful features is the notion that you can combine a text 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 (so, in this case it'll remove text from the current position till the end of line).

But first, let's have a closer look at motions.

Cursor Motions (or how to navigate around)

- `h`, `j`, `k`, `l`
- `w` move to the beginning of the next word
- `b` move to the start of the current word
- `e` move to the last character of the next 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

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.

One very special kind of motion is called “text objects” motion. Text objects are just a way to define things we work in text - for example sentence, paragraph, section, etc. Here are some notions associated with them:

- `(` / `)` move cursor backward / forward one sentence at a time
- `[` / `]` move cursor backward / forward one paragraph at a time

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. Basically 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

By the way, if you want 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 of doing things 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. Let's make a pause here and 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 third party help - Plugins!

Plugins are very helpful for almost 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 ~/.local/share/nvim/site/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.

With `:colorscheme` command you change 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 is finally looks nice.

Now we're talking!

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.

With it, you can use objects like these:

- `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

There are a couple of operators we didn't talk about before. The first is called `y` (yank), and the other is `p` (paste). They are used for copying and pasting stuff. 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. 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).