Chapter 1: Installation

LazyVim for Ambitious Developers - Chapter 1: Install

So you think you're ready to install Neovim? Actually, you still have one decision to make.

Neovim can run in a lot of different contexts (You can even run it inside VS Code!) By default it is a terminal program, but there are also tons of GUIs available. I have tried almost all of them, and, honestly, I don't think they have any inherent advantage over running Neovim directly in the terminal.

We will discuss hooking up Neovim to the Neovide GUI much later in the book, but for starting out, I recommend running Neovim in a terminal. A very good terminal, to be specific.

Choosing a Terminal

To get the best Vim editing experience, you want a GPU accelerated terminal. What's that mean? Basically that you will be using the chip designed to render photo-realistic video for rendering source code. Makes as much sense is using it for AI, right?

You will need to do your own research on the following options, so ask your favourite search engine or the AI Chat bot de jour to help you decide:

- Kitty Terminal is my personal preference. I find it well-documented, easy to configure and has all the features I need.
- Alacritty is probably the winner for raw speed, but configuration is awkward, and it is less featureful.
- Wezterm has some very nifty features, but I found the documentation to be lacking and had trouble getting some aspects to work.
- Windows Terminal, if you are a Windows user, does claim to be GPU accelerated, though I found that Neovim was sometimes unresponsive in it.
- If you're already on the Warp Terminal train and you just can't live without it, Neovim *will* work inside it. I found the experience a little choppy, and I didn't enjoy the look and feel of Warp (or the fact that I needed to sign in to use it)

So install one or more of those until you find one that you like. You *can* use other terminal emulators if you want to. You probably won't even notice that the experience is inferior, but

I can promise that if you later switch to a GPU-accelerated experience, you'll notice the improvement.

Setting Up Your Terminal Font

LazyVim and its plugins look beautiful in a terminal, and you would almost not believe that they are not a GUI application. To do this, they depend on special fonts that have a TON of glyphs for coding related things. Most notably, this gives you access to icons representing the type of file you have open, but also provide nice frames and window-like behaviour in the terminal.

To get the best LazyVim experience, you will need to install one of these special fonts and configure your terminal to use it. Indeed, you should really be using one of these fonts in your terminal even if you aren't a heavy Neovim user. A lot of modern terminal apps (there's a phrase, eh?) look better if you have them installed.

Visit Nerd Fonts for more information and to choose a font. I personally use the VictorMono Nerd Font because it has a unique typeface for the italic font that I like for code comment blocks.

You can choose from many of the most popular programming fonts. Downloading and installing them is very much operating system dependent, so I'll leave the Nerd Fonts website to explain it to you.

Install Neovim

Installing Neovim is generally one of the least problematic installs you will encounter as Neovim works pretty much anywhere software can be installed, and it only relies on standard system dependencies. The chief problem is that no matter which operating system you use, you are spoiled for choice!

You can visit the Neovim home page and click the "Install Now" button to get the latest instructions for your operating system of choice (or of necessity) from the Neovim developers.

Which Version should I install?

Neovim development happens at a super fast pace compared to their release cycle, so it is not uncommon for folks to run the latest nightly build. I have only rarely encountered bugs in builds cut from the master branch on Github, so it's generally safe. I usually run off the latest stable release when it comes out, and then when some new plugin update says "here's a cool feature if you use Neovim nightly," I'll install the latest build instead.

I suggest starting with the stable version of Neovim for now, which at time of writing is 0.10.0. LazyVim does tend to add features from the nightly release, so if you start to get as excited about this distro as I am, it will be a perfectly natural progression to switch to the pre-release.

Windows

I generally recommend using the Windows Subsystem for Linux (WSL) and doing all development in there. WSL is way outside the scope of this book, but it is well-documented by Microsoft and many online tutorials. Once you have chosen a WSL-compatible Linux distribution, set it up, and have it running in your chosen terminal, you can install Neovim using the Linux instructions below.

If you have a reason to—or preference for—developing on native Windows, the easiest thing to do is grab the MSI installer from the Releases section of the `neovim/neovim` repository on GitHub.

If you already use Winget, Chocolatey, or Scoop to manage packages on your Windows machine, there is a Neovim package in each of them.

Note that if you use Windows without WSL, you will need to install a C compiler in order to get treesitter support. This is not a trivial task. It is documented in the nvim-treesitter/nvim-treesitter GitHub repo, so I won't go into detail here.

MacOS

I recommend first installing Homebrew if you don't have it already by following the instructions at brew.sh.

Once you have brew up and running, the command `brew install neovim` will instal Neovim.

If you want to live on the edge, `brew install --HEAD neovim` will install the latest nightly version of Neovim, which is probably, but not guaranteed to be, stable.

I find the brew experience to be much kinder than other MacOS installation options for Neovim, so if you aren't already a Homebrew user, I strongly suggest exploring setting it up. There are other open source tools that you will want to install as we get deeper into the LazyVim journey, and brew will be the easiest way to get them all.

If you don't want to use Homebrew, things are a bit more annoying. The Neovim dev team doesn't maintain a MacOS installer, so you'll have to download a tarball and extract it, then link to the binary from somewhere on your system path. If you don't know what any of that means, honestly, use Homebrew, it's easier!

Linux

If Neovim isn't available using your distribution's default package manager, you have a very strange Linux distribution, indeed!

So just run `sudo pacman -S neovim`, `sudo apt install neovim`, `sudo dnf install neovim`, or the appropriate command for whichever more esoteric package manager you prefer.

If you want a nightly version, you may find the instructions on the neovim/neovim GitHub Releases page, or will have to dig into your distro's documentation.

You will also need to install a C compiler in the unlikely event that your Linux distribution didn't come with one. For most distros, just install the `gcc` package and you should be good to go.

Try Neovim Raw (If You Dare)

Once you have Neovim installed, you can try it out by simply typing `nvim` (or `nvim <filename>` to open a specific file) into the terminal you installed a few sections ago. If it is installed correctly and on your path, you'll get an unappealing looking editor that looks like it was forked from something written in the 90s that had the express intent of looking like it was written in the 70s.

So, at least it's honest?

Unfortunately, you're now trapped. To save you the frantic "how do I exit Vim" google search, the command to quit is `Escape` followed by the three characters `:q!` followed by `Enter`: `<Escape>` `<Colon>` `q` `<Exclamation>` `<Enter>`.

Seriously, "How do I exit Vim" is one of the top three autocompletes on Google for "How do I exit…". Apparently only a Samsung TV plus and full screen mode on MacOS are less intuitive to get out of!

> **TIP:** If you want to, you can run the command `<Escape>:Tutor<Enter>` to open an interactive text file that you can read through and edit while learning the basics of Neovim. I do recommend doing this at some point, but now may not be the right time, as a lot of things that are "normal" in the Vim tutor are different (better!) using LazyVim. The rest of this book does **not** assume you have gone through the tutor, but it also won't necessarily cover everything that is available there.

Install LazyVim

Now that you have Neovim up and running, let's get it configured to look like it was developed this century.

Installing LazyVim requires a bit of work with `git`. Since you are reading this book, I am assuming you are a) a software developer and therefore b) familiar with `git`. You can probably use whichever visual git tool you are familiar with. But… now that you have that fancy GPU accelerated terminal emulator, I say put it to good use.

The git commands to install LazyVim are more or less the same for the various operating systems, though paths and environment variables are different.

Start with a clean slate

> Tip: If you already have a NeoVim config and you want to try LazyVim without losing your existing configuration, set the `NVIM_APPNAME=lazyvim` environment variable. Skip to the `Clone the starter template` section below, and clone into `~/.config/lazyvim` instead of `~/.config/nvim`. If you want to make the changes permanent, either set `NVIM_APPNAME` in your shell's startup file or rename the config folder to `nvim`.

First, remove or back up all existing Neovim state. This step is largely optional if you've never used Neovim before, but I recommend making sure the following directories have been removed or moved:

Clean up: Windows with Subsystem for Linux, MacOS, and Linux

Remove or back up (with `mv` instead of `rm`) the following directories:

```
rm -rf ~/.config/nvim
rm -rf ~/.local/share/nvim
rm -rf ~/.local/state/nvim
rm -rf ~/.cache/nvim
```

Clean Up: Windows without WSL

The location of the config and data folders is a little bit different, but the idea is the same as for the Unix systems. Just use Powershell commands instead of the Unix core-tools:

```
Move-Item $env:LOCALAPPDATA\nvim $env:LOCALAPPDATA\nvim.bak
Move-Item $env:LOCALAPPDATA\nvim-data $env:LOCALAPPDATA\nvim-data.bak
```

Install other recommended dependencies

I strongly recommend installing `lazygit`, `ripgrep` and `fd`, which are used by LazyVim to provide enhanced git, string searching, and file searching behaviours. Most operating system package managers will have these available for trivial installation. You can find more specific installation instructions on their respective GitHub repositories under jesseduffield/lazygit, BurntSushi/ripgrep and sharkdp/fd respectively.

Clone the starter template

You'll use a `git clone` command to download the starter template and copy it into the user config directory for Neovim, then remove the `.git` folder.

The starter is just that: a starter. So you won't ever need to pull changes from this repo. Instead, LazyVim will manage updating itself and all its plugins for you. The only reason the starter is a git repo is that it's easy for the LazyVim maintainers to maintain. From your point of view you're just downloading the current state of the repo and don't need to know about the past or future state.

git clone: Windows with Subsystem for Linux, MacOS, and Linux

On Unix systems, use these commands:

```
git clone https://github.com/LazyVim/starter ~/.config/nvim
rm -rf ~/.config/nvim/.git
```

git clone: Windows without WSL

On native Windows, use these commands:

```
git clone https://github.com/LazyVim/starter $env:LOCALAPPDATA\nvim
Remove-Item $env:LOCALAPPDATA\nvim\.git -Recurse -Force
```

The Dashboard

Ok, you have completed the most difficult section of this book and you're finally ready to start LazyVim! Use the same terminal command as before: `nvim`.

You'll see a flurry of activity as LazyVim sets everything up and downloads the plugins it thinks are essential. You may see it compile and install a bunch of treesitter grammars; if you see a message to "Show More" use `G` (i.e. `Shift+g` to skip to the end. Once everything is installed, you'll see a summary of the plugins that were installed inside a window managed by a plugin called `lazy.nvim`

The `lazy.nvim` plugin should not be confused with `LazyVim` itself, though both are maintained by the same person. `lazy.nvim` is strictly a plugin manager, whereas `LazyVim` is a collection of plugins and configurations that ship together. One of those plugins is `lazy.nvim`.

We'll be covering most of the plugins that ship with LazyVim later in this book, so for now, once you get to the `lazy.nvim` screen, you can press the `q`. The plugin will interpret this as `quit lazy.nvim` and the window will close.

Now you can see the LazyVim dashboard, which is the first thing you'll see every time you start LazyVim. It's a little more friendly than the out of the box Neovim experience:

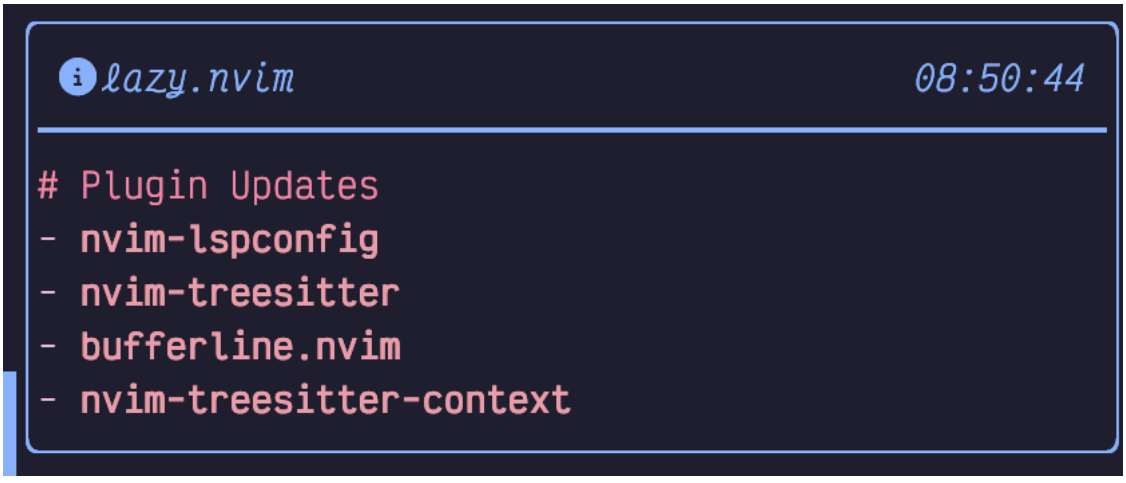

nvim ~
LAZYVIM
Z
Z
z
z
Find File f
New File n
Recent Files r
Find Text g
Config c
Restore Session s
Lazy Extras x
Lazy l
Quit q
Neovim loaded 25/70 plugins in 117.48ms

As you can see, there are several commands that allow you to interact with the dashboard via a single keystroke. Most importantly, of course, is the `q` keystroke to quit!

Most of these options are self-explanatory, but we'll discuss a few of them more deeply in later chapters.

Lazy.nvim Plugin Manager

When you first open LazyVim, it checks for any plugins that are available to be updated, and gives you an overview in a message notification that will look something like this:



Because Neovim is pretty barebones by default, LazyVim ships with a ton of useful plugins ready to go. And there's a good chance they are out of date because plugin development in the Neovim world happens at a ridiculously fast pace.

In the old old days, plugin management was a completely manual process. In the less old, but still old days, it was managed by a variety of plugins that did the job but felt like they were lacking something.

Then came the plugin manager called `lazy.nvim`, created by the same person that later created LazyVim.

`Lazy.nvim` has a ton of slick features, most notably loading plugins only when needed (hence the name "Lazy") so that your editor is lightning fast to start up. It also has a nice UI for managing plugins installation and updates.

You can access this UI from the dashboard simply by pressing the `l` key, which is labelled in the dashboard as `Lazy`. The label should probably be `Lazy PLugin Manager` to make it a bit more clear, but now you know what `Lazy` means so you won't forget.

If you are not actively displaying the dashboard, you can show the plugin manager at any time by entering Space mode. We'll cover Space mode in detail in the next chapter, but for now: First make sure you are in Normal mode by checking the lower left corner of the active window. If not, press `Esc` to enter Normal mode. Then press `Space` to enter space mode, followed by `l` to bring up the lazy.nvim plugin manager.

(Don't worry, those keybindings will all be second nature within a week.)

The Lazy plugin manager interface looks like this:

The window that has popped up is called a floating window. You'll see these in a few different situations, usually when there is interactive data that you need to work with (like a web modal). This particular floating window comes with its own set of keybindings. The keybindings are listed across the top, and pay attention to the fact that all of them are capitalized, so you need to use the `Shift` key when invoking them.

Realistically, the only keybinding I use on a regular basis is `S`, for `Sync`. This is the equivalent of running install, clean, and update in one single action, so it has the effect of guaranteeing that the versions of plugins that are actually installed are exactly consistent with the ones specified in the LazyVim configuration.

So when the "Plugin updates available" notification pops up, just press `Space-l` and then `S` and wait for the sync to complete. Then press `q` to close the Lazy.nvim Plugin mode and floating window and return to what you were doing.

A Note on Managing Dot Files

If you work on multiple different computers, you'll quickly find that you don't want to set up your LazyVim configuration separately on all of them. LazyVim does not have the equivalent of VS Code's "settings sync", though such plugins exist.

An alternative I recommend instead is to store your config files in a git repository. You'll probably find there are a few other files you want to keep in there such as your `.gitconfig` and `.zshrc` / `.bashrc` / `.config/fish/config.fish`. If you use GitHub Codespaces, you may already manage some dot files with git.

If not, my personal recommendation is to follow the advice in the excellent blog article Dotfiles: Best way to store in a bare git repository from the Atlassian blog.

Before distributions like LazyVim came along, it was very common for people to store their Vim configuration in a public repository, and borrow ideas from each other. This practice is not quite dead, and you can find my own dot files on GitHub in the dusty-phillips/dotfiles repository.

Summary

In this chapter, we briefly discussed the history of Vim, Neovim, and LazyVim, and why they are still relevant today. Then we covered the importance of GPU accelerated terminals and Nerd Fonts.

We figured out how to install Neovim and its dependencies under whichever operating system(s) you use, and finally, installed LazyVim from its starter template.

In the next chapter, we'll discuss Vim's core feature: Modal Editing, and dig into the many things you can do with your keyboard in LazyVim.

Chapter 2: What is Modal Editing, Anyway?

Chapter 2: What is Modal Editing, Anyway? - LazyVim for Ambitious Developers

As you may have guessed from the letters on the dashboard, LazyVim is very keyboard-centric. As many actions as possible can be performed without moving your hands between mouse and keyboard. That's not to say that it's impossible to use the mouse. You can click anywhere in the editor, interact with buttons and modals when they pop up, use the scroll

wheel or gestures to scroll, and resize editor panes by dragging their borders, for example. But you can also do all of these things using the keyboard, and usually more efficiently.

More importantly, you can do most things by holding at most two keys, and usually just one. You will only rarely have to contort your hands into painful (and dangerous) positions to `Control + Shift + Alt + <some key>`.

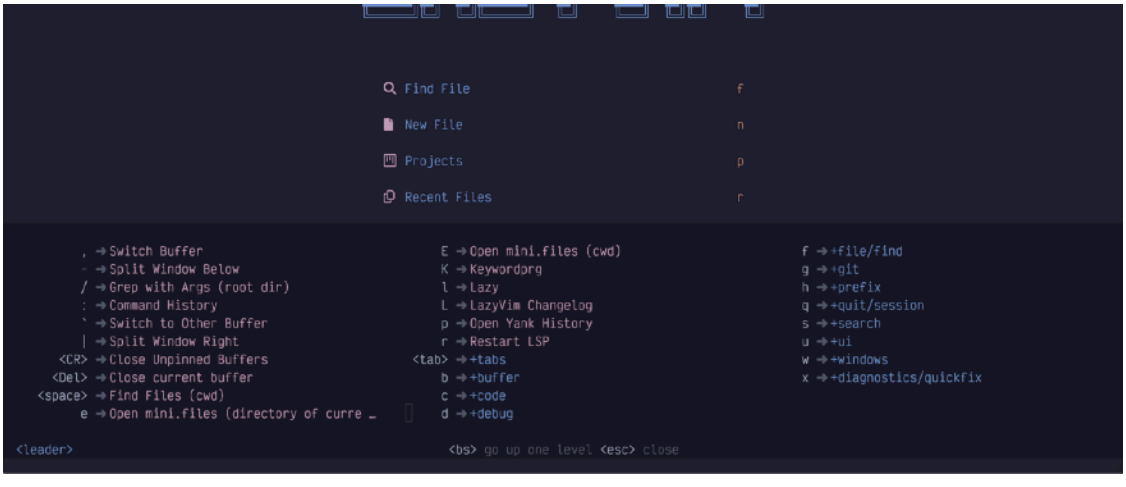
How does Vim do this? Modal editing.

Introduction to Modal Editing

"Modes" in LazyVim simply mean that different keystrokes mean different things depending on which mode is currently active. For example, when you start the editor up, you are in a "Dashboard Mode", and the most common interpretation of keystrokes in that mode are listed right there on the dashboard. This discoverability of keybindings in a given mode is a common theme in LazyVim, and a huge improvement over the opaque default behaviour of Neovim itself.

To see what I mean, press the spacebar to enter "Space mode". Space mode is a LazyVim concept; it does not exist in a raw Neovim installation (though you can install various plugins to recreate the effect if you want Space Mode without LazyVim).

Entering Space mode pops up a menu along the bottom of your screen. If you have the dashboard open, it will look something like this (my menu contains some customizations, so yours won't be identical):



That's a big menu. The important thing to focus on right now is the `f` key, which we will use to understand modal editing.

If you are in Dashboard mode and press the `f` key, you will open the `Find file` dialog using a plugin we'll discuss later called Telescope. However, now that you are in Space mode, if you press the `f` key, it will open the `file/find` Space mode submenu.

This is the crux of what modal editing means: The behaviour of a given key depends on the current mode. As indicated by the line at the bottom of the Space mode menu, you can press the `Escape` key to exit Space mode and return to the dashboard. Go ahead and do that.

Now you're back in Dashboard mode, and you can press the `n` key to create a new, empty buffer.

Pay close attention to the lower left corner of that buffer, where you'll see the word `INSERT` in green:

Remember how I said Space mode is a LazyVim concept? Insert mode is a Vi concept, that the successors Vim, then Neovim, and now LazyVim have all inherited. In Insert mode, the vast majority of keystrokes do what you would expect in any editor: they insert text. So you can touch type as with any other editor!

You can access *some* keyboard shortcuts in Insert mode using `Control` and `Alt` keys. For example, you can hit `Control-r` to enter the "Registers" mini-mode, which pops up a list of "registers" you can paste from. We'll cover registers in detail later. For now, it is enough to know that `Control-r` followed by the plus key (i.e. `Shift-=`) will paste text from the clipboard in Insert mode.

However, you will much more often change to *Normal* mode to perform any non-text-entry operations, including pasting text.

To get into Normal mode from Insert mode, hit the `Escape` key. The cursor will change from a bar to a block and the indicator in the lower left corner will change to a blue `NORMAL`:

In Normal mode, pressing various keyboard characters will not insert text like it does in Insert mode. For example, pressing `p`, rather than inserting a literal `p` character into the document, will instead paste from the system clipboard.

Vim and Neovim aren't very discoverable, but they ARE extremely memorable. As often as possible, the keyboard shortcuts to perform an action start with a letter that makes sense for the action being performed. You might think `p` stands for "paste", but in fact the concept has been around for longer than the clipboard mnemonic. You are welcome to think of it as "paste" if that's easier for you, but in Vim parlance, it actually stands for "put", and we'll use that word in different contexts throughout the book.

For some contrast, the `Control-r` key that pops up the list of registers in Insert mode does **not** pop up a list of registers in Normal mode. Instead, `Control-r` means "redo" (aka undo an undo). In order to enter the Registers mini-mode from Normal mode, you would press the `"` (quote, as in Shift-apostrophe) key instead.

If that sounds confusing, don't worry. Your brain and muscle memory will adapt more quickly than you expect and you'll always understand that behaviours in Normal mode are not the same as in insert mode.

To be honest, I hardly ever use non-text-entry commands in Insert mode. I find it is easier to switch back to Normal mode and then perform the command from normal mode. It doesn't usually take a higher number of keystrokes to do so and I don't have to hold down multiple keys at once.

As I mentioned, the universal key to exit Insert mode and return to Normal mode is `Escape`. And that brings us to an important point: You will be using this key a lot, but moving your hands from the home row to the Escape key in the upper left corner and back again is somewhat inefficient.

There are a few common workarounds to this situation:

- If you have a customizable keyboard you can put the escape key in a more accessible location. This is what I do. I have a Kinesis Advantage 360, and I remapped the keys so that escape is in the "thumb key" section of this admittedly bizarre keyboard. It's as easy to hit as enter, space, and backspace, other keys that I use very frequently.
- Your operating system is probably also capable of remapping keys. A lot of users replace the largely useless `Capslock` with the `Escape` key. (If you ever go back to the keyboard chaining editors descended from Emacs, including VS Code: For these editors it can be more comfortable to remap `Capslock` to the commonly-held `Control` key, especially on laptop keyboards).
- Neovim itself is also able to remap keys. We'll discuss how to do this in LazyVim later. One common pattern is to map a series of uncommon keystrokes that you wouldn't likely type together when inserting text to the escape key. So you can set it up to map something like `jk`, `jj` or `;;` in Insert mode to switch to normal mode. I've tried this and don't care for it as it introduces a timing thing when you hit the first

character and Neovim is waiting to see if you're going to type a command or let text insertion continue, but you might like it.

- The `Control-C` keyboard combination also works to exit Insert mode, with no remapping required. I don't like this because it's two keystrokes and on my Dvorak keyboard, `Control-C` is harder to hit than on a qwerty keyboard where C is on the bottom row near the `Control` key.

Don't worry about actually changing it for now; just start getting used to using `Escape` where it is and see if you find it annoying.

Once you're in Normal mode, you'll obviously want to get back to Insert mode to enter text at some point! There are several different ways to do this that we'll discuss later. As a taste, here are a couple of the most common ones:

The `i` key always inserts text *before* the current cursor position. This means that you could (very clumsily) move your cursor left by pressing `i <Escape> i <Escape>` repeatedly. When you press `i`, you insert text before the current position, and then escape takes you out of Insert mode at that new "before" position.

Commonly, you want to enter Insert mode *after* the current cursor position. To do that, use the `a` key instead (mnemonic: i = **I**nsert Before, a = **Append**, although I usually think of it as **A**fter).

You'll find that you need to alternate between these a lot as you are navigating a document because the various navigation commands we'll cover later will often put you just before or just after the position you need to insert at. So it's important to remember both of them.

Two other very common operations are to insert at the very beginning or the very end of the current line. You *could* use navigation commands to move to the start or end and then use `i` and `a`, but it's easier to use the commands `I` and `A` instead (The difference is that they are capitalized, so you need the Shift key with them).

A note on Keybinding Mnemonics

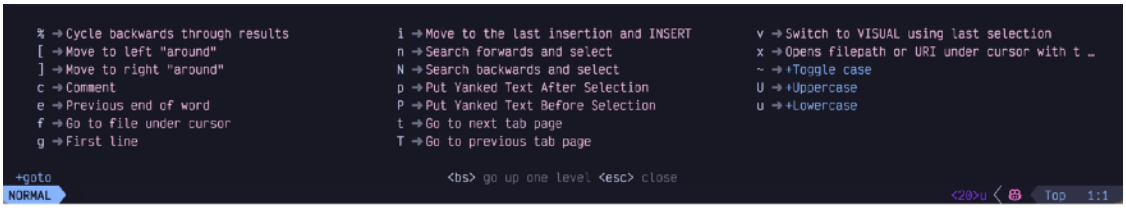
It is very common for related keybindings like these to be assigned to the lowercase and uppercase versions of the same key. You will often find that the lower case version means `do something` and the uppercase version means either `do the same thing only BIGGER` or `do the opposite thing`, depending on the situation. In this case, `i` and `a` mean "insert one character before or after the cursor" and `I` and `A` are "insert before or after the cursor, only BIGGER (i.e. at the beginning or end of the line)".

To illustrate the "do the opposite thing" situation, consider the `o` and (shifted) `O` keys, which are two new ways to get into Insert mode.

The `o` key is used to enter Insert mode on a new line below the current one. I've heard the mnemonic as "**O**pen a new line above/below" to help you remember the otherwise not terribly memorable `o` command. And in the classic "do the opposite thing" scenario, the shifted `O` means "create a new line *above* the current one and enter Insert mode on it".

Let's discuss one final *very* useful command that takes two keystrokes, one after the other: `gi`. That is a single press and release of `g` followed by `i`.

This effectively means "Go to the last place you entered insert mode, and enter Insert mode again". In this case, the `g` key is actually switching to a new mini-mode I call "Go To" mode, though not all the commands accessible from it are strictly related to going places. You can see the entire list of commands available in "Go To" mode by pressing the `g` key in Normal mode and waiting for the menu to pop up at the bottom of the window:



We'll cover most of them later, but notice that the `i` key is in there labelled `Move to the last insertion and INSERT`. So if you forget how to go to the last insertion point, you can enter Go To mode and scan the menu to find the `i` again.

Try all of those commands (`a`, `i`, `o`, `A`, `I`, `O`, and `gi`) repeatedly, entering some text and pressing `Escape` to return to Normal mode. Then try it again. Move your cursor around the text using the mouse (we'll get to keyboard navigation soon, I promise), and try using the commands again to see how they behave in new locations.

Get *really* comfortable with switching between Normal and Insert mode. You might think you'll spend most of your time in Insert mode, but the truth is code is edited far more often than it is written afresh, and you'll be alternating between them constantly.

Visual Mode

The other major mode that LazyVim inherits from its ancestors is "Visual" mode. Visual mode is used to select text. In general, you can enter Visual mode and then use many of the same navigation keys you would use in Normal mode to move your cursor around. Since we haven't covered those navigation keystrokes yet, I'm going to defer a detailed discussion of Visual Mode until we have the necessary foundation.

Command Mode

Command mode is different from the other modes we've seen, which were mostly either submenus or editor-level major modes. You can get into command mode from Normal mode by using the `:` (i.e. `Shift-<semicolon>`) command. In LazyVim, this will pop up a little widget where you can type what is known as an "Ex Command." This name comes from `vi`'s predecessor, `ex`, which hasn't really been used (other than as part of Vim) in decades.

Essentially, you can enter a wide variety of commands into this widget and expect certain behaviours to happen as a result. It is actually more similar to the VS Code command palette than anything else, though it is a quite different beast.

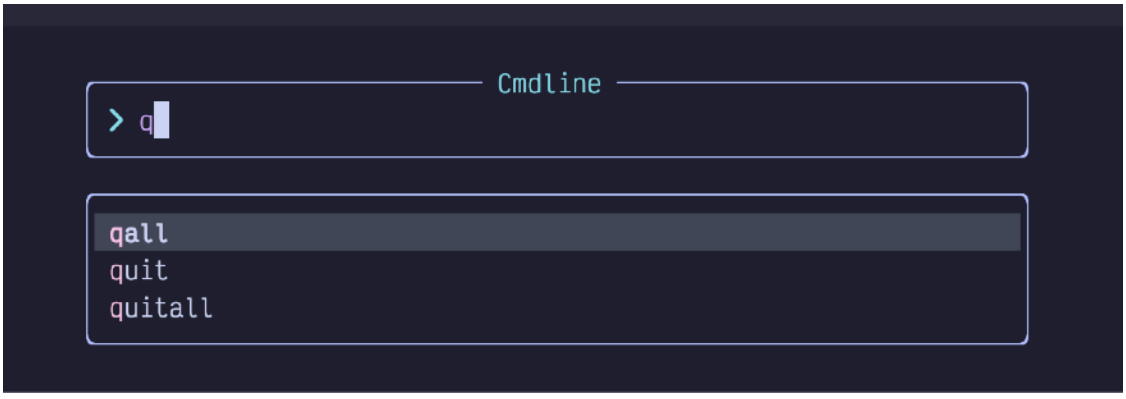
You already know one ex command from the previous chapter! Remember `<Escape><Colon>q!<Enter>` the command to exit the editor? You now know that the `Escape` is to enter Normal mode from whatever mode you are in. The colon is used to switch to Command mode, and the `q` is short for quit (You could type the full word `quit` if you didn't feel the need to conserve keystrokes). The exclamation point says "without saving" and the `Enter` means "submit the ex command".

As another example, let's consider the `write` ex command. Type `:` followed by `write myfile.txt` like this:

Press `Enter` to confirm and execute the command.

> Note: Most commands can be shortened to their shortest unique common prefix. You can type `:w myfile.txt` instead of `:write myfile.txt`. The most popular commands even have special combined commands, so `:wq` will save and exit, although you'll probably prefer `:x` as it's even shorter.

Command mode is kind of weird because it's kind of like an Insert mode in the sense that you can type text into it, and some of the keybindings that work in Insert mode also work in command mode (including `Control-r` to paste from a register). But other keybindings work differently in command mode. The most important one is the `Tab` key, which will do a sort of "tab completion" on the command. For example, `:q<Tab>` pops up a menu like this:



This damn completion menu is surprisingly unintuitive to navigate. You're probably going to want to bookmark this section or take some notes or something until you get used to it!

First, if you want to select a different entry in the menu, you would surely think you can use the arrow keys. Which you can, but it's a mind-mess because you need to use `Left` and `Right` to move the cursor `Up` and `Down`. I know! WTF, right?

This is mostly because the menu looks different in LazyVim than it did in the original Vim, but the keys haven't been remapped. So instead, I suggest using `Tab` and `Shift-Tab` to select different entries from the menu. It's easier to remember and much easier on the muscle memory: Tab once to show the menu, tab again to cycle through the menu.

Second, there is some nuance around *confirming* one of those menu entries. In the above example, you can just press `Enter` to confirm the selection **and execute it**. However, there are often cases where you want to confirm the selection and then continue editing the command. An excellent example is the `:e` or `:edit` command.

This command is used to open a file on your filesystem, but you have to type the entire path to the file. For example, if you have the following directory structure:

```
.
└── foo
    ├── bar
    └── baz
    │   └── fizz.txt
```

...and you have Neovim open, you would have to type the following to open the `fizz.txt` file:

```
:e foo/baz/fizz.txt
```

That's a lot of typing if you need to get to deeply nested directories. Luckily, you *can* use tab completion for this. You can type `:e f<tab>b<tab><tab><tab>` to get `foo/baz`, but at this point the menu is still open:

If you press `Enter` now, it's going to open the `baz` folder instead of just confirming the selection, which is not what you want. And if you pres `Tab` again it will cycle through the menu some more.

Instead, you have a couple of options. The `Down` arrow key will move "into" the selected directory, allowing you to tab through the files inside it. Alternatively, use the `Control-y` (`y` for "yes") key combination. This will confirm the `baz` selection and close the menu but leave you in command mode. Now you can press tab again to complete the `fizz.txt` portion of the command.

It **is** possible to remap these keys to be more like other software, and I honestly think this is one thing LazyVim should do by default. I haven't found a combination that I like, though, so I just stick with the default keybindings.

You probably won't spend a lot of time in command mode. There are easier ways to open files in LazyVim, for example, as well as to quit the editor. And if you need to do something more complex with command history, there is a special window you can use to edit commands with Insert and Normal mode that we will cover later.

For now, remember `<Tab>` and `Control-y` and you'll be able to navigate the Command menu when you need to. There are other keybindings you can use to edit commands, but unless you find yourself annoyed by certain repeated tasks, I wouldn't worry about them.

The most important command, by the way, is `:help`. Vim was created before folks had ready access to the Internet, so it has a tradition of shipping all of its documentation with the editor. So for example, if you can't remember the keyboard shortcut to `put` text, try `:help put`. Or, if you want to know what the `Control-R` keyboard shortcut does, try `:help CTRL-R`. Of course, the Vim help documents have been indexed by your favourite search engines and AI chat bots, so you can go all new-school and ask them if you prefer.

Summary

In this chapter, we became comfortable with the concept of modal editing and the most important LazyVim modes. There are other mini-modes and one major mode that will come up as we progress through this book, but becoming comfortable with Normal, Insert, and Command mode (and how to switch between them) will take you a long way on your LazyVim journey.

In the next chapter, we'll learn a whole bunch of different ways to move the cursor around inside a document.

Chapter 3: Getting Around

Chapter 3: Getting Around - LazyVim for Ambitious Developers

Software developers spend far more time editing code than we do writing it. We're always debugging, adding features, and refactoring.

Indeed, the most common thing I ever do is add a print/printf/Println/console.log at some specific line in the codebase.

If you are coming from the more common word processing or text editing ecosystems, navigating code is the thing that is most different in Vim's modal paradigm. Even if you're used to Vim, some of the plugins LazyVim ships by default suggest different methods of code navigation from the old Vim standbys.

In VS Code, often the quickest way to get from one point in the code to another is to use the mouse. For minor movements, the arrow keys work well, and they can be combined with `Control`, `Alt`, or `Cmd/Win` to move in larger increments such as by words, paragraphs, or to the beginning or end of the line. There are numerous other keyboard shortcuts to make getting around easier, and the Language Server support allows for easy semantic code navigation such as "Go to Definition" and "Go to Symbol".

Vim also supports mouse navigation, but you'll likely reach for it less often once you train up on the navigation keymappings. LazyVim has keybindings for the same Language Server Protocol features that VS Code has, and they are often more accessible. The big difference with Vim is the entire keyboard's worth of navigation commands that are opened up to you when your editor is in Normal mode.

Seeking Text

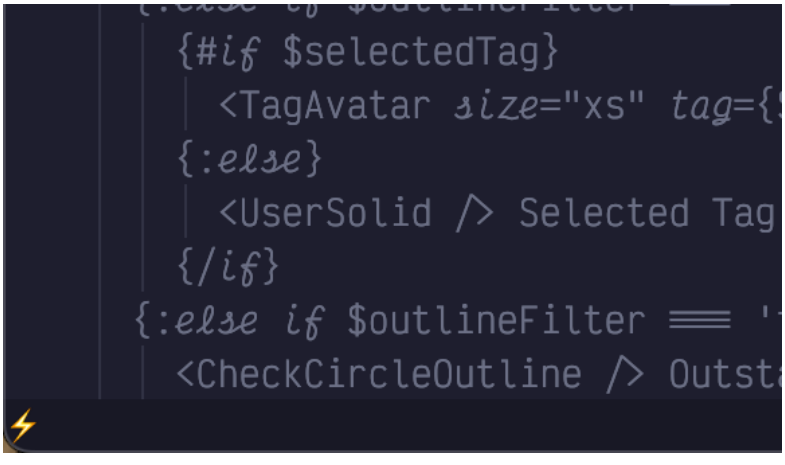
LazyVim ships with a plugin called `flash.nvim`, which was created by the creator of LazyVim and integrates very nicely with it.

This plugin provides a code navigation mode that has been available in various vim plugins (starting with one called EasyMotion) for many years, and has historically been quite controversial. A lot of long-time Vim users think it breaks the Vim paradigm. I won't go into the details as to why, but I will acknowledge that this was true in older iterations of the paradigm and is much less true in modern versions such as `flash.nvim`.

If you can see the code you want to navigate to (i.e. because the file is currently open and the code is scrolled into view), `flash.nvim` is almost always the fastest way to move your cursor there. It admittedly takes at least three keystrokes, but those three keystrokes require no mental math or incrementally "moving closer" to the target until you get there, which are two of the less efficient problems that come up with certain other Vim navigation techniques (as well as in non-modal editing).

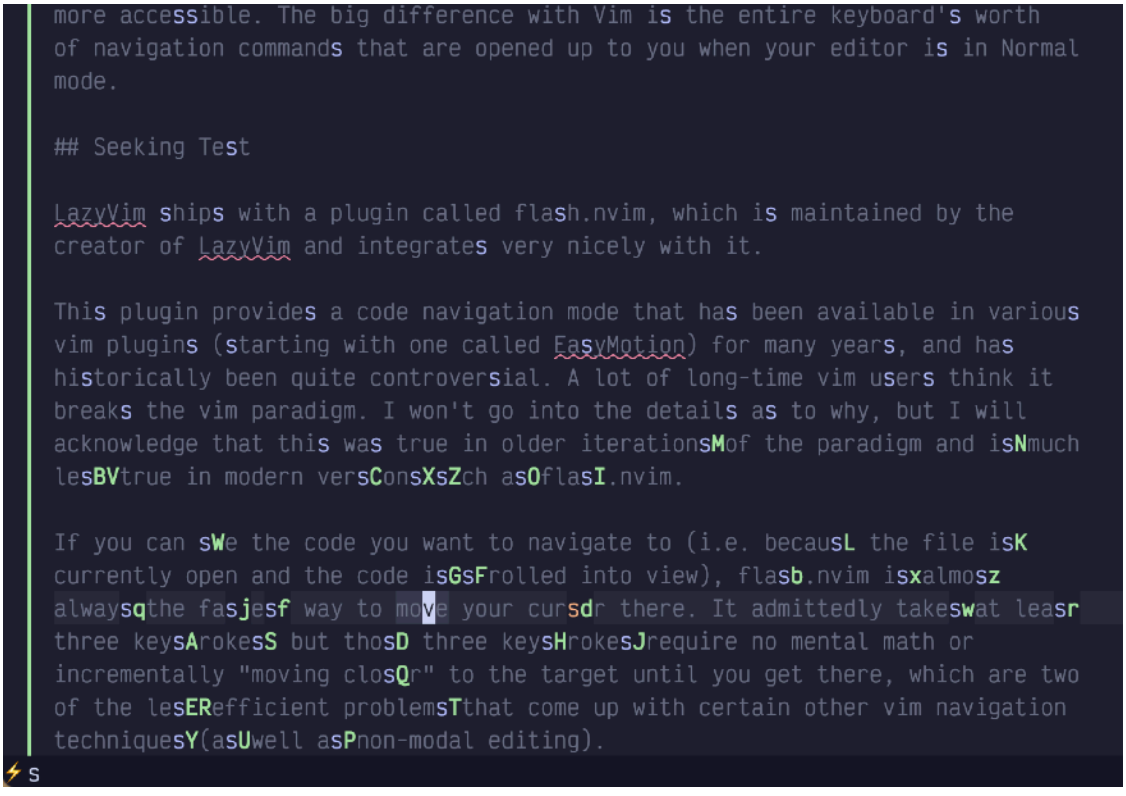
To invoke `flash`, press the `s` key in Normal mode. My mnemonic for `s` is "s stands for **s**eek", although I've also heard it referred to as "sneak" or "search" mode. Searching in LazyVim is a different behaviour (it doesn't care if the text is currently visible or not), and "sneaking" sounds a little too dishonest, so I use "Seek".

The first thing to notice when you press `s` is that the text fades to a uniform colour and there's a little lightning symbol in the Mode indicator indicating that Flash mode is active:



Since you know where you want the cursor to be, your eyes are probably looking right at it, and you know exactly what character is at that location. So after entering seek mode, simply type the character you want to jump to.

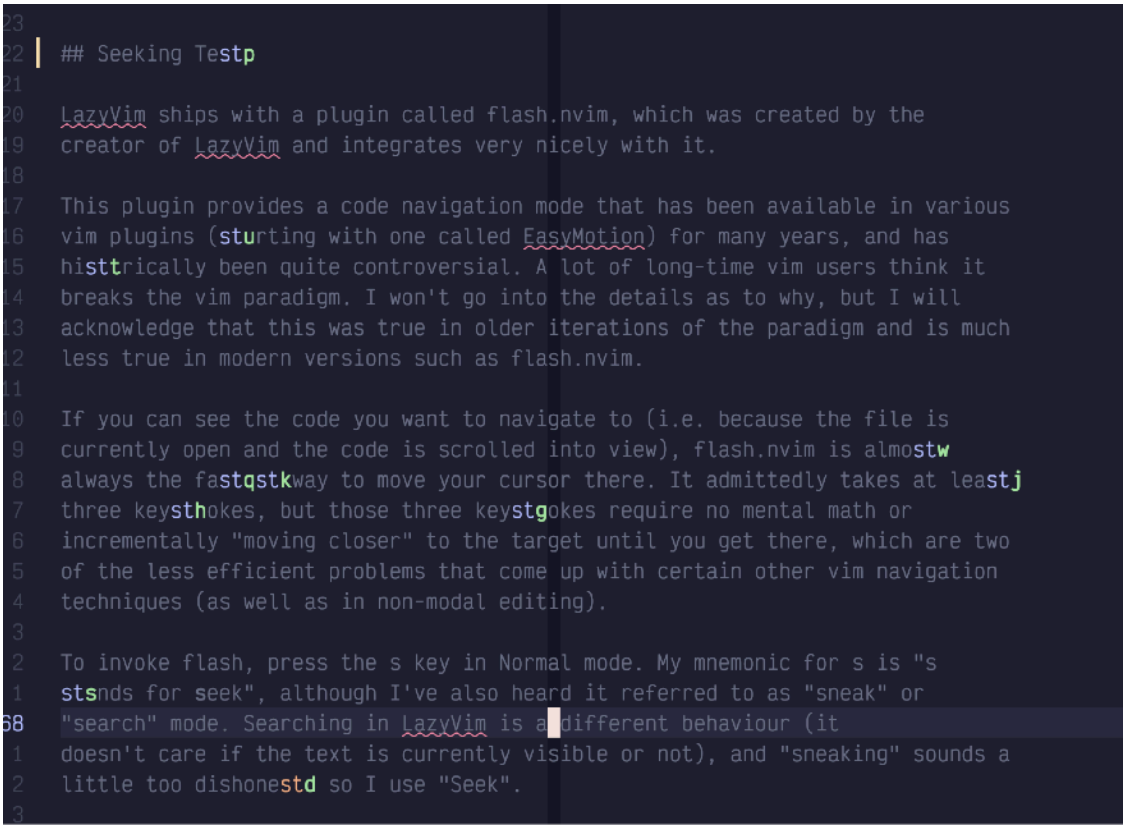
For example, in the following screenshot, I want to fix the (intentional) typo in the heading of this section, changing `Test` to `Text`.



I have hit `ss`, and every single `s` in the screenshot has turned blue, including capitals. There is an `s` character beside the flash icon in the status bar indicating that I have seeked an `s`.

In addition, *beside* (to the right) of all the `s` characters nearest to the cursor (which is in the bottom paragraph) have a green label beside them. If I wanted to jump to any of those `s` characters, I would just have to type that label and boom, I'd be there.

However, the character I want to hit is too far away to have a unique label, as there are a lot of `s` characters in my text. No matter! I just have to type the character to the right of the target `s` character, which is a `t`. Now my screen looks like this:



Now, all instances of `st` in the file are highlighted in blue, and since there aren't as many `st` as `s`, all of those instances have a label beside them. The text I want to move to is labelled with a `p`, so I press `p` and my cursor is moved to the `s` character I wanted to change. Now I can type `rx` to replace the `s` with an `x` (we'll discuss *editing* code in a later chapter, but now you've had a taste of it).

If you have multiple files open in splits (which we'll also discuss in detail later), Seek mode can be used to move your cursor *anywhere* on the screen, not just in the currently active file.

Seek mode does have drawbacks however, at least the way `flash.nvim` implements it. There are some characters you can't move to directly because you run out of text to search for before a labelled match is in that location. For me this happens most often when I want to edit the end of a line. If I type `sn` because I want to edit a line that has `n` as the last character, but there are a bunch of `n` characters closer to my cursor than the one I want to move to, flash may not label the `n` I want to move to, and it won't accept a carriage return as a "next character" input.

For this reason, I don't seek near ends of lines. Instead, I'll seek to a word somewhere in the middle of the same line and then use `A` which, as you may recall, will put me in Insert mode at the end of the line. Alternatively, if I don't want to enter insert mode, I will use the `$` symbol (`Shift+4`), which is the Normal mode command for "Move cursor to end of current line".

Scrolling the screen

Seek mode only works if the text you want to jump to is visible on the screen. You can't label something you can't see! Often, this means you want to use search or one of the larger or more specific motions discussed later, but there are also a few keybindings you can use to scroll the screen so you can see your target and jump to it.

These keybindings are a little unusual by Vim standards because they mostly involve using the control key. How anti-modal! In my experience, these keybindings don't actually get a ton of use. Indeed I've forgotten some of them and had to look them up to write this chapter.

The scrolling keys I use the most are definitely `Control-d` and `Control-u`, where the mnemonic is **d**own and **u**p. They scroll the window by half a screen's worth of text. The cursor stays in the same spot relative to the **window**, which means that it is moved up or down by half a screen's worth of text relative to the **document**.

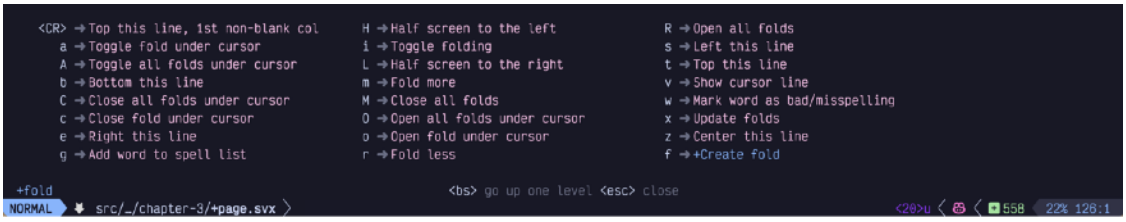
If you need to move even further, you can use the `Control-f` and `Control-b` keybindings, which move by a full page of text. I don't like these ones because I never quite know where the cursor is going to end up and I become disoriented. But it can be handy if you need to scroll something into view quickly to use Seek mode on it. Unlike `Control-d` and `Control-u`, `Control-f` and `Control-b` can be prefixed with a count, so you can type `5<Control-f>` if you need to scroll ahead by `5` pages.

I have no idea why the keys `Control-y` and `Control-e` where chosen to scroll the window by one line at a time. I never use them. These keybindings accept a count, so if you can remember them, they are useful for subtle repositioning of the text. The main advantage of these keybindings is that they don't move the cursor unless it would scroll off the screen, so if you are working on a line and need more visibility but don't want to move the cursor, you could use `Control-y` and `Control-e` to do it.

The reason I don't use these keys (other than lack of a decent mnemonic) is that I prefer to do relative cursor positioning using `z` mode.

Z Mode

The `z` menu is kind an of an eclectic mix of cursor positioning, code folding, and random sub-menus, as you can see by pressing the `z` key while in normal mode:



If that looks like a big menu, you don't know the half of it! There are a ton of other z-mode keybindings that are obscure enough to not deserve mention in the menu! I'll cover the three most useful scrolling related ones here and we'll discuss others later.

The relative cursor keybindings I use exclusively are `zt`, `zb`, and `zz`. These move the line that the cursor is currently on to the top, bottom, or middle of the screen, respectively. When moving to the top or bottom it will leave a few lines of context above or below the cursor.

There are others that will also move the cursor to the first column of the window, but instead of memorizing those shortcuts, I recommend using `zt0`, `zb0`, and `zz0` instead. As we'll discuss later, the `0` command just means "Go to the start of the line". You can also use home if your keyboard has a home key, but `0` is easier to hit on many keyboards.

You can find other scrolling keybindings in the Neovim documentation by typing `:help scrolling`, but the ones I just mentioned will probably more than cover your needs as you learn far more nuanced methods of navigating code.

The first rule of Vim

So there is a holy rule in Vim that I constantly break for valid reasons. Unless you are the very strange combination of weird that I am, you probably should not break it quite so often:

Never use the arrow keys to move the cursor.

The background behind this rule is that it takes a tenth of a second or so to move your hand to the arrow keys on most keyboards, and another tenth of a second to move it back to the home row. I'm not convinced these tenths of a second add up to an appreciable amount of time, even considering the millions of characters I have typed in my lifetime. (Yes, millions. I did the math once).

But I do think the arrow keys on most keyboards can do nasty things to the long term health of your hands, and honestly, the more you get used to the alternative Vim keybindings, the more you'll prefer to use them.

The Vim keybindings for arrow keys seem rather unintuitive when you first look at them: `h`, `j`, `k`, and `l`. These map to the directions, left, down, up, and right. If it seems weird that `l` means "right" instead of `left`, or you're wondering why they skipped `i` since that appears to be an alphabetic sequence, look at your keyboard.

If you are an English-speaker with a standard Qwerty keyboard, the letters `h`, `j`, `k`, and `l` are on the home row under your right hand, in that order, and are therefore the easiest keys to hit on the entire keyboard.

Open a largish file in Neovim (you can use `:e path/to/filename`) and experiment with moving the cursor left, right, up and down using the home row keys. While you do that, I'll tell you why I don't use them because I'm triply abnormal.

First, I'm left handed, so the right hand home row is slightly less accessible feeling. Second, I've been a Dvorak user for two decades. The `j`, `k`, and `l` keys are not on my home row. Third, I use a Kinesis Advantage 360 keyboard, which, among other bizarre layout features, places the arrow keys within reach of my fingers so I don't have to move my hand to hit them.

By a strange twist of fate, these weirdnesses kind of cancel each other out. The `j` and `k` keys happen to be directly above the left and right arrow keys under my dominant left hand. So that's what I use for navigation: `Left` `Right`, `j` `k`. If you are less weird than me, you should probably use the right-hand home row keys the way Vim was designed.

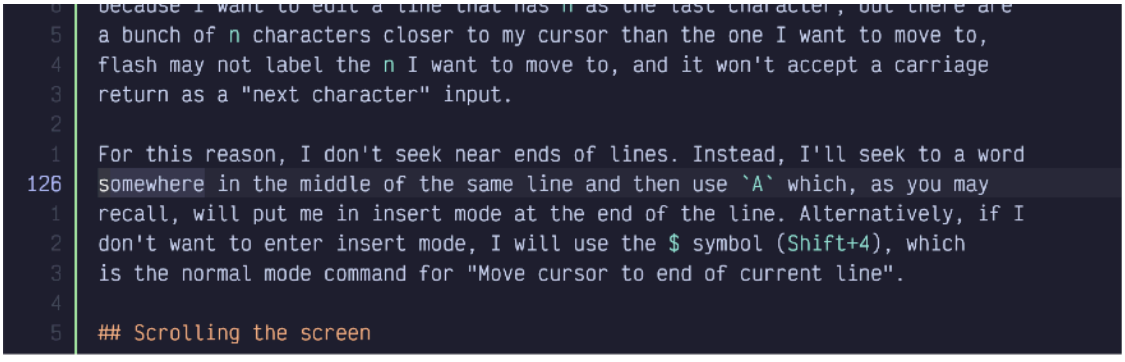
Vim, Neovim, and LazyVim are all *really* good at reusing motions, so you will find that `h`, `j`, `k`, and `l` are used for a lot of different navigation sequences as you progress through this book. Take enough time to really get used to them. But recognize that if you ever have to push these keys more than twice in succession to move the cursor, you're wasting keystrokes.

Counting

The vast majority of commands in Vim can be prefixed with a *count* to repeat the motion multiple times. The count is typically entered as a sequence of digits before the command you want to repeat.

So, for example, to move the cursor up 15 lines, you would enter normal mode and hit the keys `15k`. To move it five characters to the right, use `5l`.

This is why LazyVim has such weird line numbering by default. Consider the following screenshot:



My cursor in this screenshot is on line 126, which is highlighted in the left gutter. It's also is also visible in the lower right corner of my window, though I cropped it out in this screenshot. But directly above line `126` we see the line number 1, and directly below it we also see the line number 1.

Let's say I want to move my cursor to the `Scrolling the screen` heading.

This line has the number `5` beside it, so I don't have to count lines or do any mental arithmetic to figure out the count to use to move my cursor. I just type `5j` and my cursor moves to the desired line.

Now that you know what they are for, I suggest leaving relative numbers on until you get used to them. If you find them distracting or just don't use them, you can change to normal line numbers by editing your LazyVim configuration. Open the file `~/.config/nvim/lua/config/options.lua`, which should have been created for you by LazyVim but currently won't have anything in it other than a comment describing what it is for.

> Tip: You can use the Space mode command `<Space>fc` to quickly find files in the LazyVim configuration directory. This will pop up one of the file pickers that we'll discuss in detail in the next chapter. Type `options` and press `<Enter>` to open the file.

To disable relative file numbers, add this line to the file and save it:

```
vim.opt.relativenumber = false
```

Then reopen Neovim, and you should see the absolute value of line numbers in the left column.

Personally, I find line numbers to not be very useful and I don't like wasting valuable screen width on displaying those characters. As has become a running theme, I recognize that I am somewhat odd! But if you also want to disable line numbers altogether, you'll need another line in `options.lua`:

```
vim.opt.number = false
vim.opt.relativenumber = false
```

Find mode

If you need to move your cursor to a position that is relatively close to its current position, you may want to use LazyVim's Find mode instead of the Seek mode we described earlier. The default Find mode in Neovim is rather limited, but the `flash.nvim` plugin that enables Seek mode makes it much nicer to use.

To enter find mode, press the `f` key. Like Seek mode, a portion of your screen will dim, indicating that you should type another character, and after you do so, all instances of that character *after the cursor* will be highlighted. For example, `fs` will highlight all instances of the letter `s` after the current cursor position.

This is where the similarities between `Find` mode and `Seek` mode end, however. Instead of showing a label, the cursor immediately jumps forward to the first matching character after the cursor. You'll also notice that none of the text before the cursor has been dimmed, and that none of the matching characters in the lines before the cursor are highlighted.

Instead, we need to use counts to jump to later instances of the character. If I want to jump ahead to the third highlighted `s`, I type `3f` and my cursor will move there. However, if you want to jump to a much later `s`, you probably don't want to individually count how many `s` keys there are. Luckily, after you use a count, LazyVim leaves you in find mode, so you can just guess how many `s` characters there are, and then once you are closer, repeat with a new count. If you only want to jump ahead by one `s` character, you don't need to enter a count, just press `f` by itself and you'll move ahead.

If you miscounted or misguessed and jump too far, don't worry! You can take advantage of the fact that (Shifted) `F` means "find backwards", and can also be counted. So if you need to move to the 15th highlighted `s`, it's totally fine to guess `18f`, realize you've gone three too far, and use `3F` to jump back to the previous character.

Moreover, if you know that the character you are looking for is behind or above your cursor in the document, you can enter Find mode with `F` instead of `f` in the first place. This will immediately start a backwards find operation instead of a forward one. And if you know right off the bat that you want to jump back or ahead by three instances of the given character, you can even use a count when you first enter find mode.

There is also a subtle variation of Find mode that I call "To" mode, although the official Vim mnemonic is actual "'til" mode. You enter it with a `t` or `T` depending on what direction you want to go.

"To" mode behaves identically to find mode except that it jumps to *just before* the target character.

You might think that To mode is kind of redundant because you could fairly easily use find mode followed by a single `h` to move the cursor left. But "To" mode is extremely useful when you are combining it with operations to edit the text, which we will discuss later. As a taste, if you use the command `d2ts`, it will delete all text between the cursor and the second `s` it encounters, but leave that `s` alone. This is much easier than the `d2fsis<Escape>` that would be required if you used a find command and then had to enter Insert mode to add the `s` back.

Moving by Words

When `f` or `t` feels too big, and cursors with counts feel too small, you'll most likely want to use the word movement commands. In other editors and IDEs you might be used to getting this functionality by holding `Control`, `Alt`, or `Option` (depending on the operating system and editor) while using the arrow keys.

Neovim is easier; you don't have to move your hands to the arrow key section of the keyboard and you don't have to hold down multiple keys at once.

Instead, you can just enter Normal mode and press the `w` key to move to the beginning of the next word. If you instead want to move to the *end* of the current word, use the `e` key. If you are *already* at the end of the current word, `e` will go to the end of the *next* word.

This is useful when you want to combine it with counts: If you need to move to the end of the word that is two words after the current word, press `3e`. This is the same as pressing `e` three times, which would move to the end of the current word, then the next word, and finally to the end of the word you want to hit. `w` can also be prefixed with a count if you need to move to the beginning of a word that is a certain number of words after the current one.

Use the `b` key if you want to move backwards instead. This will move you to the beginning of the current word, or if you are already at the beginning of the word, it will move to the beginning of the previous word. As usual, use a count to move to the beginning of even more words.

Surprisingly, it takes a bit more work to move to the end of the *previous* word, as you need to press two keys: `g` followed by `e`. The mnemonic for this is "**g**o to **e**nd of previous word". In practice, you'll find that you hardly ever need this functionality for some reason, and the

honest truth is I usually use `be` (`b` to move to beginning of previous word, then `e` to go to end of that word) to move to the end of the previous word. If you do use `ge`, however, it can be combined with a count as well. You'll need to type something like `4ge`, depending on the count. The command `g4e` wouldn't do anything useful.

Collectively, you may occasionally hear the `w`, `e`, and `b` commands referred as the "web" words. It just means "moving by words". These are probably the most common movements you will use, more than individual cursor positions, simply because most editing actions tend to involve changing or deleting a word or sequence of words.

Moving by Words, Only BIGGER

The "shifted" form of the `web` words also move by words, but the definition of "word" is subtly different. Specifically, a capital `W` will move to just after the next whitespace character, where a lowercase `w` will use other forms of punctuation to delimit a word. Consider a method call on an object that looks something like this in many languages:

```
myObj.methodName('foo', 'bar', 'baz');
```

If you cursor is currently at the beginning of that line, a `w` will move your cursor to the period on the line, a second `w` will move you to the `m`, and subsequent `w` presses will stop at the paren and quotes as well.

On the other hand, if your cursor is at the beginning of the line, a `W` will move you all the way to the first quote in the `"bar"` argument, since that is where the first whitespace character is.

As a visualization, here are all the stops on that line of code when you use `w` compared to when you use `W`:

```
myObj.methodName("foo", "bar", "baz")
-----ww---------w-w--w--ww--w--ww--w---->
----------------------W------W-------->
```

The `B`, `E`, and `gE` motions behave similarly, moving in the appropriate direction by whitespace-delimited words instead of punctuation ones.

One thing that is kind of annoying both in Vim and the way LazyVim is configured is that there's no way to navigate between the individual words of `CamelCaseWords` or `snake_case_words`. You can use `fC` or `t_` and similar if you want to, but I will later show you up how to set up the `nvim-spider` plugin that makes navigating these common programming constructs simpler.

Line targets

Very frequently, you need to move to the beginning or end of the line you are currently editing. Often you can use `I` or `A` for this if your goal is to move to that location and enter insert mode, but if you need to move there and stay in Normal mode (e.g. for other purposes such as to delete or change a word) you can use the `^`, `$`, and `0` commands.

If you are familiar with regular expressions, you might know that `^` is used to match the start of text or start of the line and that `$` is used to match the end, so the mnemonic of using these two keybindings to match the beginning and end of the current line will hopefully be less unmemorable than they seem at first.

There is a certain lack of symmetry between the two, however. The `$` (`Shift-4`) command simply means "go to the end of the line", as in the last character before the ending newline, no matter what that character is. The `^` or caret (`Shift-6`) means "go to the beginning of the text on this line". The "of the text" there is important: if your line has whitespace at the beginning (e.g. indentation), the `^` caret will **not** go to the very first column, but will instead go to the first non-whitespace character.

To move to the very beginning of the line, use the `0` key. `0` is the **only** numeric key that maps to a command because the others all start a count. But it wouldn't make sense to start a count with `0`, so we get to use it for "move to the zeroth column".

There is also a command to go to the end of the line excluding whitespace, but I have never used it, probably because I usually have formatters configured to trim trailing whitespace so it doesn't come up.

The two character combination `g_` (g underscore) means "go to the last non-blank character". I guess _ kind of looks like "not a space", so it's kind of mnemonic? I include it to be comprehensive, but you'll likely not use it much. You also have the option of combining other commands you've learned so you don't have to memorize this one off. For example, you can use the three character `$ge` (combining "end of line" with go backwards to end of word) or `$be` to move to the last non-blank on the line. You have options; pick the one that you find is easiest to remember or type!

Jumping to specific lines

If you compile some code or run a linter, you will invariably be given a line number where the error occurred (unless the compiler is particularly useless).

You can jump to a specific line by entering the line number as a count, followed by (shifted) `G`. So `100G` will move your cursor to line 100. Alternatively, you can use the `:100<Enter>` ex command.

`G` is a normal command, though, so you can issue it without a count, in which case the `G` command will always take you to the end of the file.

You can go to the top of the file with `1G` if you want, but since this is such a common operation, you can instead use `gg` (two lower case `g`s). The mnemonic for `g` in all cases is "Go to", and there are a lot of things that can come after a `g` (`:help g` will introduce you to the ones I don't cover, although be aware that LazyVim has overridden some of them).

Since the most common place you are likely to want to "go to" is a line number, the easiest to type `G` and `gg` commands are used for line number navigation.

Jump History

All this jumping around can make you feel a little lost. Luckily, there are two super-useful keybindings for going back to places you previously jumped.

`Control-o` is the non-modal control-based keybinding that I use most often. I should honestly bind it to something more accessible, I use it so much. It basically means "Go to the place I jumped from".

This is super handy when you're editing code deep in a file or module and realize you need to import a library at the top of the file. You can use `gg` to jump to the top of the file, `s` to seek to the line you want to add the import on, and then enter Insert mode to add the import. Now you want to go back to the code you were working on so you can actually use the import. `Control-o` a couple times will take you there.

Neovim keeps a history of *all* your jumps, so you can jump between several locations (perhaps to look up documentation or the call signature for a function) and always find a way back.

If you jump too far, you can use the `Control-i` keybinding to jump *forward* in history. It's just the opposite of `Control-o`. I don't know why `i` and `o` were chosen for these; maybe because they are side-by-side on a Qwerty keyboard? They are used commonly enough that once you learn them, you won't forget.

Summary

Navigating code is a huge topic in Vim. You've already learned enough commands that you can navigate a Vim window more efficiently than most non-modal editors can dream of. But we've actually barely scratched the surface, and we'll be covering a bunch of even more useful code navigation commands in a later chapter.

We covered the LazyVim `Seek` mode to jump anywhere in the visible window, and then the scrolling commands to make sure the thing you want to jump to is visible. Then we covered moving the cursor with the home row key and extended them with counts.

We learned how Find mode differs from Seek mode, even though they are superficially similar. Then we covered some standard keybindings for moving by words and to key places on a line before jumping to specific lines. We wrapped up by covering how to navigate to places you have jumped before.

In the next chapter, we'll learn more about opening files and navigating the Filesystem.

Chapter 4: Opening Files

Chapter 4: Opening Files - LazyVim for Ambitious Developers

In the previous chapter, as a side-effect of learning about command mode, we saw how to open files the old-fashioned Vim way, using the `:edit` command. Another old-school alternative is to open them directly from the terminal shell command line, using `nvim filename`.

Both of these are occasionally handy, but LazyVim pre-configures a few more modern ways of navigating and opening files.

Introducing File Pickers

LazyVim ships with two different "picker" plugins, tools for selecting items from a list. We'll look at both in this chapter with some advice on which to choose. They look a bit differently, but can be used interchangeably.

By default, LazyVim ships with the Telescope picker enabled, so we'll cover that first. It provides a "picker" interface with preview and fuzzy search capabilities. If you've used the command menu in many modern editors (or even Github or Slack), you may know what I'm talking about. The picker itself doesn't care what you are picking, and it is used for a wide variety of tasks built into LazyVim or as third-party plugins, including opening files, selecting open buffers, project-wide search, and more.

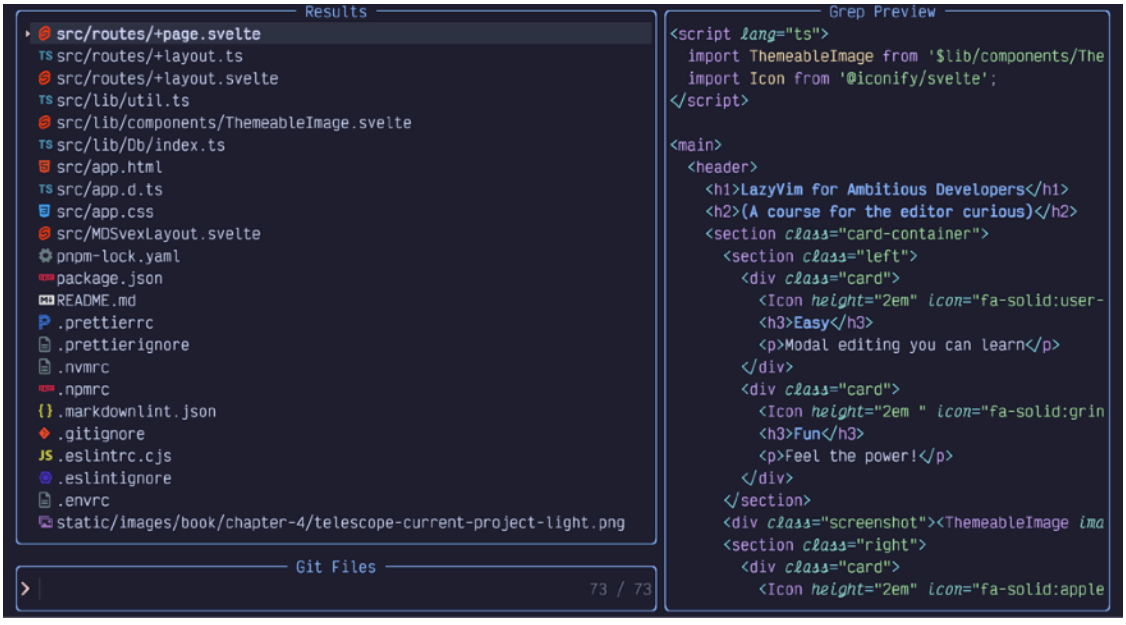
The most common picker task you will perform is to open a file using fuzzy search. I use this command dozens, maybe hundreds of times per day, so it's a good thing it's got a really accessible keybinding.

The file picker is best illustrated while working in a code repository with a lot of files. So close Neovim with `Space q q` and use the `cd` command in your terminal to change to the directory of a project you've been working on recently (If you don't have one close to hand, clone your favourite open source project and use that instead). Then type `nvim` to open Neovim again.

> **Tip:** I had you exit to the terminal above because it's easy to reason about, but it is also possible to change directories from inside LazyVim using the `:cd` command. Type `:cd the/path/to/the/directory` and hit enter, remembering that you can use the `Tab` key to autocomplete the path. Now if you use `:e` to open files, they will be relative to the directory you specified. If you are using a file picker, they may be relative to that cwd or to the project containing the current file, as discussed shortly. Use `:pwd` to see what the current directory is.

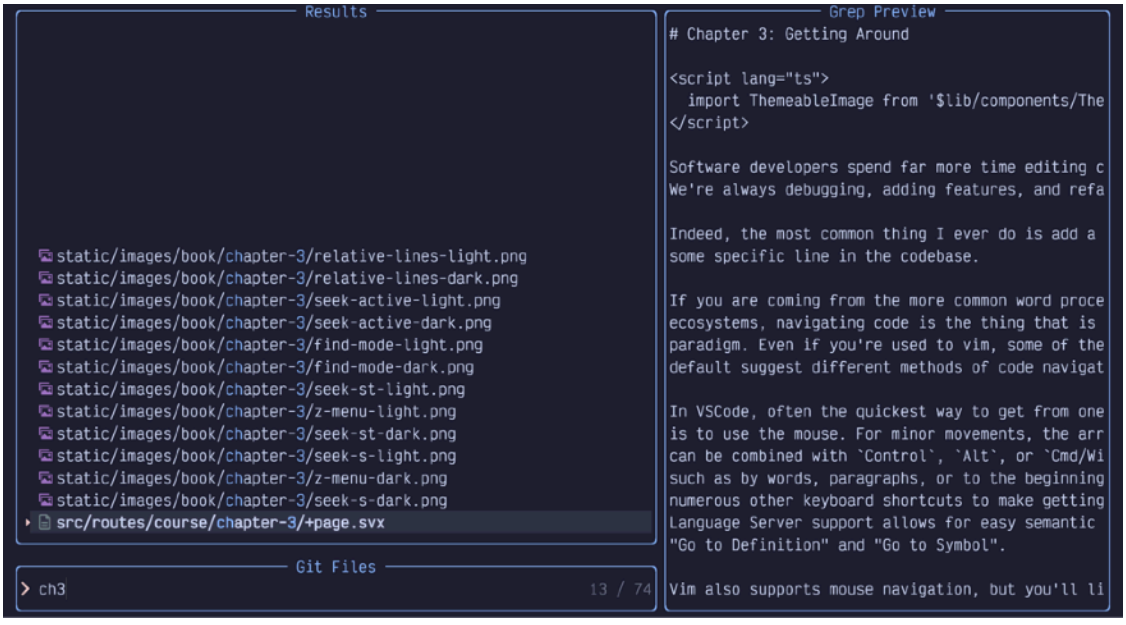
Ok, so you're in the root directory of a large project and you want to open an arbitrary file. Simply press `Space` twice (i.e. `Space Space`) to pop up the "Files In Current Project" picker. As I mentioned, this is the easiest keybinding to type on your entire keyboard. The Space bar on most keyboards is big, and you're hitting it with your strongest digit – the thumb. As usual, just one `Space` will pop up the Space mode menu, and you can see that a second `Space` will present you with "Find Files (root dir)".

For the project containing the current state of this book, the picker looks like this:



The picker is divided into three main areas: the results list in the upper left, a preview of the currently selected file on the right, and the Input area, in this case labelled “Git Files”.

The input area is actively focused and currently in insert mode, so you can just start typing the name of whatever file you want to open. This is a “fuzzy search”, (a concept popularized by Sublime Text) which means you can skip letters, saving you oh-so-precious milliseconds. For example, if I type `ch3`, my list gets filtered down to the following files:

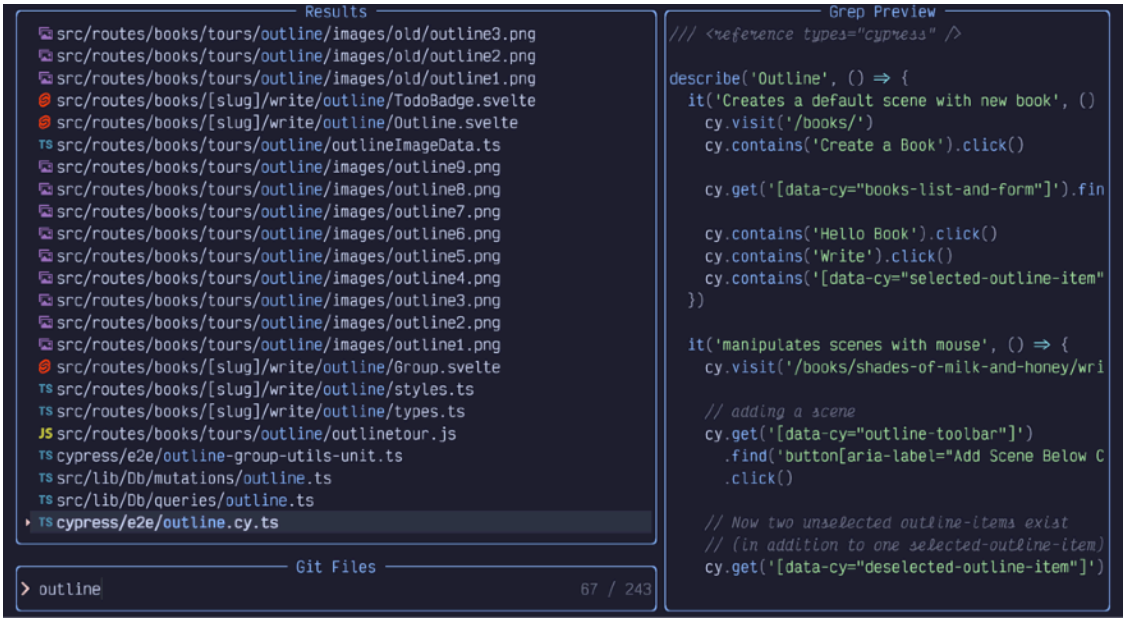


Only files whose paths contain those three characters in order, with possibly other characters in between, are visible. The picker has helpfully highlighted those three letters in the results so you can easily see why it matched (Though, depending on the medium you are reading, it may not be clear in the image.)

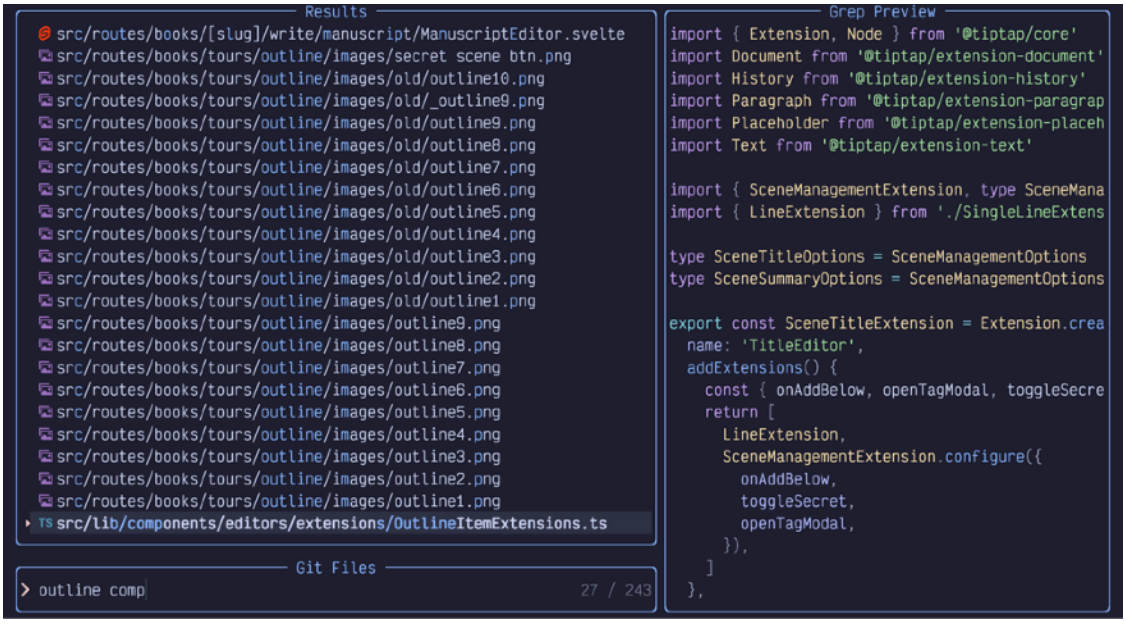
Also notice that by default, the match is case **in**sensitive. I typed the lowercase letter `c`, but it matched the uppercase `C` in the filename. This is usually sufficient to narrow the search results to what you need. However, if you *do* use **any** capitalized letters in your search than it switches to a case sensitive mode (this is sometimes referred to as “smart case”).

That means that `Ch` will match all the `Chapters`, but `cH` will not match anything at all. More interesting, `chF` will *also* not match anything at all because the presence of the capitalized `F` makes the whole thing case sensitive, and the chapters are all named with a capital `C`, so the lowercase `c` is not able to match them.

Another neat picker matching trick: Sometimes you will start typing a word and realize you need to match something *earlier* in the path to distinguish it. For example, I started typing `outline` in these source files from Fablehenge:



Outline is a common word in this app. There are 243 matching files, and I realize I should probably have typed `comp` in front to narrow it to just files in the `component` directory. I *could* switch to Normal mode and edit the beginning of the line, but it's faster to just type `<space>comp`. The picker will interpret the space as "filter the lines again fuzzy matching this new word from the beginning". Here we can see that only `comp...outline` files have been matched:

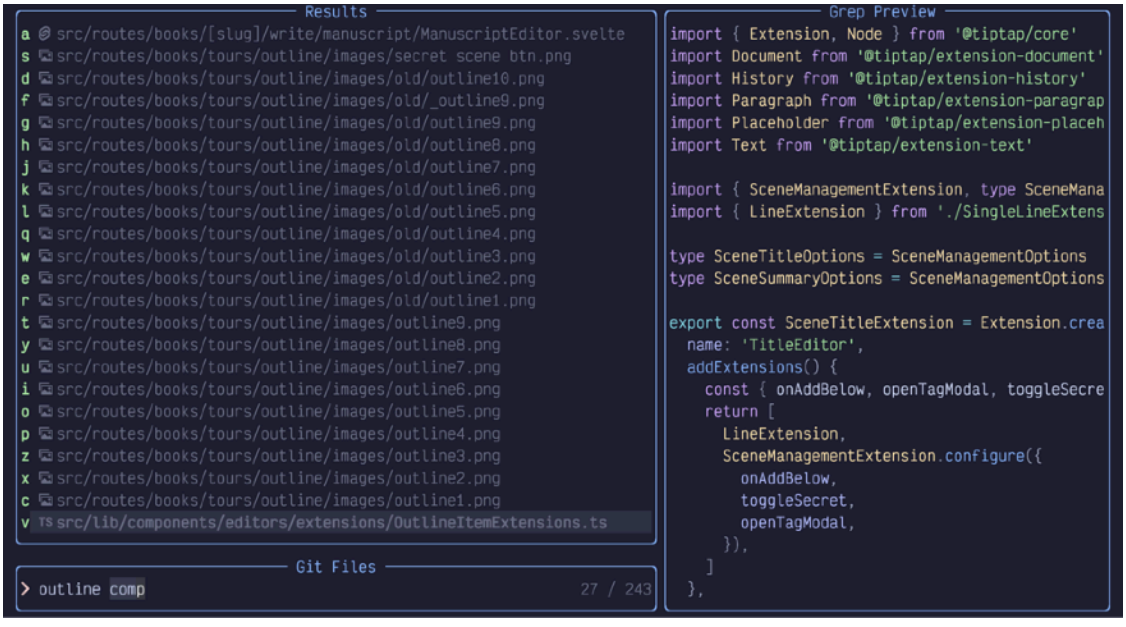


This image might be a bit surprising; the most promising match is obviously the one at the bottom of the list (which is why it is selected). The other 27 matching lines contain all the letters of the word “outline” and all the letters of the word “comp” in order from left to right. However, because of the fuzzy matching algorithm, the two can actually overlap! So on e.g. the second-to-last entry, the `c` of the matching “comp” is *before* the word `outline`, the `o` is **in** it, and the `m` and `p` both come **after** the word outline. The picker doesn’t care, though it will rank matches with the matching letters closer together as more important, so they’ll be visible at the bottom of the results.

You can use the up and down arrow keys to select a different file in the search results, and its preview will show up in the right-hand window. Once you find the file you want to open, press the `Enter` key to open it in the currently active Neovim window.

With Telescope, the input area even has its own normal mode! You can get into it using a *single* press of the Escape key. Now if you press `j` or `k`, you’ll be able to select different files in the list without moving your hand to the arrow keys. Further, the `h` and `l` keys will allow you to move the cursor within the input box and you can use the `i` or `a` keys to enter insert mode at the new location. The “but bigger” `I` and `A` keys allow you to move the cursor to the beginning or end of the line and enter Insert mode as well.

You can even use seek mode, as we discussed in Chapter 3, though it works a bit differently. When you press the `s` key while in the Telescope picker’s normal mode, you can skip the part where you enter a character to search for. Instead, LazyVim will immediately label every line in the picker with a character to the left of the filename:



These characters are labels for each line in the picker. Simply press one of the shown letters on your keyboard, and whichever line the label associated with that letter is on will be selected. Then press `Enter` to actually open the file (or, if it is not a file picker, perform the default action for that picker).

Finally, if you are in the Telescope window and decide you don't want to open any files after all (or you got the information you needed from looking at the preview and therefore don't need to open it all the way), press `Escape` *twice*. Once to enter Normal mode in the Telescope picker, and a second time to close the picker.

If you need to scroll the *preview* window to see something lower down in the file, the same `Control-d`, `Control-u`, `Control-f`, and `Control-b` keys that we discussed in the Basic Navigation chapter can be used.

The difference between "Root" and "cwd"

The `<Space><Space>` command is mapped to "Find Files (Root Directory)". Two other ways to open the file picker are to use `<Space> f` to open the "file/find" menu, and follow it with either `f` again or `F`.

`<Space>ff` is the same as `<Space><Space>`. It opens "Find Files (Root Directory)" and is just another longer way to get there. I assume it exists in both places so that users can choose to map some other action to `<Space><Space>` and still be able to access that functionality; `<Space><Space>` is the easiest keybinding to hit on the keyboard, so it makes sense to assign it to your most common action. If your most common action isn't opening files with the picker, you will still want to be able to do that with the slightly longer `<Space>ff` keybinding.

`<Space>fF`, where the second `F` is shifted, is similar; it is mapped to an action called "Find Files (cwd)". If you run it in your project, you'll probably find that it appears to do the exact same thing as "Find Files (Root Directory)" (depending on how your project is set up), so the purpose of two separate keybindings may be confusing.

Current Working Directory

`cwd` stands for "Current Working Directory", and by default, it refers to whatever directory your terminal was in when you typed `nvim` to open the editor. As I mentioned briefly while discussing tab completion in the command menu, you can change the `cwd` for the entire editor by entering command mode with `:` and then typing `cd path/to/directory` (remember, all commands are followed by a carriage return, so press `Enter` or `Return` afterwards). Now if you use `<Space>fF`, the list of files will be shown relative to the new directory you have changed into.

If you are unsure what directory you are in, you can use the `:pwd` (short for "print working directory") command to have it pop up in a little notification window. `cd` and `pwd` are the same commands used by `bash`, `zsh`, and many other shells for changing and printing the working directory, so they may already be familiar to you.

We haven't discussed splitting your editor or opening new tabs yet, but this is a good time to note that it is actually possible to have *different* working directories for different windows. The command to change just the current window's directory is `:lcd`, short for "local change directory". This can be a powerful way to work on multiple projects at the same time (for example, if you are a full stack developer working on backend and frontend projects). However, the LazyVim concept of a "Root" directory can semi-automate a lot of this.

Root directory

The root directory is not a Vim concept, but is instead a Language Server Protocol (LSP) concept. LSPs are the reason that VS Code became so popular so quickly; the idea was that the editor could call out to an external service running on your computer to find out useful things about the codebase. The LSP powers a lot of useful stuff such as go to definition and references, highlighting errors in your code, and showing documentation for a variable or class. It can even help with formatting and syntax highlighting!

The root directory is the directory that the LSP infers is the "home" directory of the currently open file. How the LSP does this is language (and language server) dependent. For example, in Javascript or Typescript projects it probably searches parent directories for the presence of a `package.json` or `tsconfig.json` file to detect the root directory, whereas in a Python project it might instead look for things like `pyproject.toml` or `poetry.lock`, and Rust projects use the directory that contains a `Cargo.toml`. Or the LSP might just use the presence of a `.git` folder as the "root" of the project's workspace.

The only reason this root directory is "often the same as your `cwd`" is that this is usually the folder you want to work from when you are working on a project, so it's the one you `cd` into before you open Neovim.

This automatic root directory thing can be super useful if you are working on multiple projects. Instead of using `lcd` as discussed in the previous section, you can just open a file in a different project using `:e` or one of the file finding extensions we'll discuss next. Then if you invoke the "Find files (root dir)" command using `<Space><Space>` or `<Space>ff`, it will look for other files in the same root directory as the one you just opened.

However, it can sometimes be confusing, especially if you are working in a so-called monorepo or if you have root directories in places you don't expect. For example, I have a

fairly normal Svelte project that has a `package.json` file in it. This projects uses Cypress for testing, and the Cypress folder has a `tsconfig.json` file in it that causes the Typescript language server to interpret that as a separate root. So if I am working on one of the cypress test files and press `<Space><Space>`, the root directory is considered the Cypress folder and I can only open other cypress tests. But often the thing I *wanted* to do was open a source file in the main folder to see why a test is failing. In this case, I have to press `<Escape><Escape>` to exit the Telescope picker, then `<Space>fF` to open the picker in current working directory mode instead.

Fzf.lua

LazyVim recently shipped a new opt-in picker experience that can be used as an alternative to Telescope. I am convinced folke did this just to make my life as an author harder. They are very similar, but different enough that they need to be documented separately.

The main advantage of `fzf.lua` is said to be that it provides a "familiar" interface to people who are used to using the fzf tool on the command line. The Fzf cli tool is really handy for changing directories or opening files in a deeply nested filesystem by only typing a handful of characters using fuzzy matching. I've used it for years and highly recommend it. That said, I've never really cared whether my editor had the exact same experience as the CLI tool, so I don't consider it a compelling argument.

The other reason `fzf.lua` is recommended is that it is considered to be more performant than Telescope. I haven't really noticed a difference on my machine but if you find Telescope is laggy, you might want to try out `fzf.lua`.

There are a couple downsides to `fzf.lua`, though. The main one is that it runs in a Neovim terminal window, so Normal mode behaves really weird. Notably, pressing escape once closes the picker instead of entering normal mode in the input area like Telescope does.

The second downside is that `fzf.lua` doesn't have the robust plugin ecosystem that Telescope does. In practice, there's only one plugin that I miss when using fzf.lua instead of Telescope, but if you become a Telescope power user, you might find there are many pickers available that aren't there when you use fzf.lua.

I honestly can't tell you which one to use, so I recommend trying both and then deciding for yourself. I personally use Telescope because I hate the Neovim Terminal mode that `fzf.lua` uses.

If you want to try `fzf.lua`, you need to learn about *Lazy Extras*. I go into more detail about Lazy Extras in the next chapter, but the short story is they are optional plugin configurations that you can enable with a single keypress. To enable `fzf.lua`, do the following:

- Type `:LazyExtras<Enter>`
- Move your cursor to the line that contains `editor.fzf`
- Press `x` to install the e**X**tra
- Wait a moment for the related plugins to install
- Restart Neovim

This will disable Telescope and enable `fzf.lua`. You'll see a slightly different picker layout when you open it:

The main difference is that the input area is at the top of the left side in `fzf.lua` instead of the bottom as it is with `Telescope`.

The picker behaves similarly to Telescope, so I'll only highlight the differences here. The biggest one, as I already mentioned, is the weird Terminal-based Normal mode. If you are in the input area and want to enter Normal mode, you have to type the bizarre set of keybindings `Control-\` followed by `Control-n` (we'll go into more detail about Terminal mode in Chapter 15, although not a LOT more detail because I hate it). This Normal mode does allow you to navigate around the window and most keybindings will behave the way you expect Normal mode to behave.

The most notable exception is that Seek mode doesn't behave like it does in Telescope. If you want to get the behaviour of jumping to a line by a label using `fzf.lua`, use the unintuitive `Control-x` keybinding instead.

`Fzf.lua` has several other keybindings configured by default; you can see a helpful menu of them by hitting `F1`. The only ones I use regularly are the annoying Emacs-style `Control-a` and `Control-e` to jump to the beginning or end of the input area , and `Control-d` and `Control-u` to scroll the results window.

Throughout the rest of this book, I will generally assume you are using Telescope rather than `fzf.lua`, though if there are glaring differences between the two I will try to mention them. For now, though, let's move on to a completely different file selecting experience.

The Neo-tree.nvim plugin

Neo-tree creates a left-sidebar file explorer experience that should be familiar to users of many modern IDEs and editors. While, like many of those environments, Neo-tree works with the mouse, it is optimized for keyboard interactions, making it faster to work with once you learn "Neo-tree mode".

I want to be upfront and honest here: I don't personally use Neo-tree. I find that the file pickers we just discussed are the fastest way to open files, and when I need to manipulate the filesystem, I prefer to use `mini.files`, which we will discuss later in this chapter. The primary reason I prefer `mini.files` is that it uses the same keybindings as Vim normal mode. Modes are great, but having more of them than necessary is not!

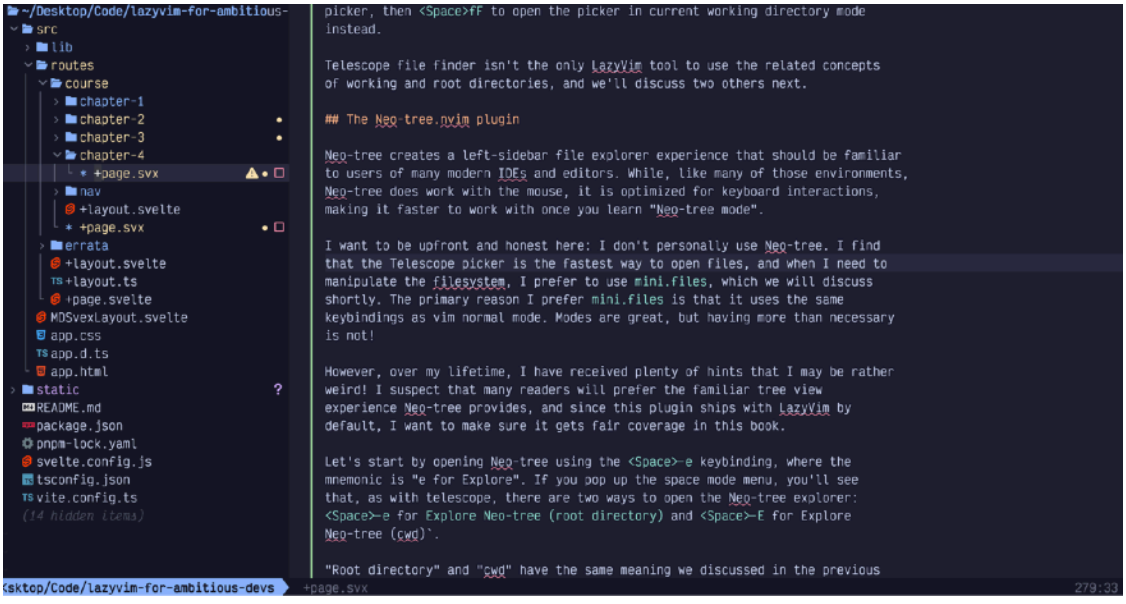
However, over my lifetime, I have received plenty of hints that I may be rather weird! I suspect that many readers will prefer the familiar tree view experience Neo-tree provides, and since this plugin ships with LazyVim by default, I want to make sure it gets fair coverage in this book.

Let's start by opening Neo-tree using the `<Space>-e` keybinding, where the mnemonic is "e for Explore". If you pop up the Space mode menu, you'll see that, as with the fuzzy picker, there are two ways to open the Neo-tree explorer: `<Space>-e` for `Explore Neo-tree (root directory)` and `<Space>-E` for Explore Neo-tree (cwd)`.

"Root directory" and "cwd" have the same meaning we discussed in the previous section, and you will notice the consistent relationship between lowercase and uppercase letters: `<Space>ff` and `<Space>e` both open the root directory, and `<Space>fF` and `<Space>E` both open the current working directory.

> **Tip:** To hide the explorer window, just press `<Space>e` again while it is visible, or press `q` while the explorer window is focused.

When the explorer is opened, it shows all the files and folders in the relevant directory, with all the folders collapsed, except for the one containing the currently active file, if there is one. For example, while editing this file, my Neo-tree looks as follows:



It may not show up clearly in the screenshot, but the cursor is on the file I'm currently editing. I can move that cursor up and down using the arrow keys or the ubiquitous `j` and `k` keys.

Folders are collected to the top of the view. If you move the cursor to one of these folders, you can press the `Enter` key to expand the folder. And if you move it to a file, you can open the file in the current Vim window with the same `Enter` keypress.

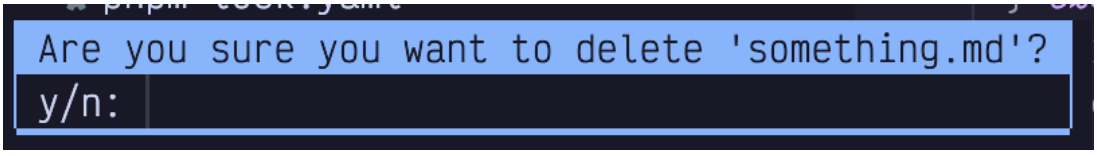
You can also expand and collapse folders and open files by double clicking with the mouse, but my guess is you won't want to do that once you learn proper keyboard navigation.

Speaking of keyboard navigation, yes, `j` and `k` to move up and down can be super slow if there are a lot of files to navigate. All of the commands that we discussed in Chapter 3 can be used to move faster. For example, `10j` will move the cursor 10 lines down with just three keystrokes compared to pressing `j` 10 times, and `Control-d` or `Control-u` can be used to scroll the tree down or up. Most interestingly, `s` can be used to Seek to any line in the Neo-tree view, even if Neo-tree is not currently focused.

Neo-tree will show the root or cwd as the topmost directory. If you need to navigate "up" the tree to a higher-level directory, you will need to use the `Backspace` key.

> **Tip:** Backspace is often coded as <BS> in Vim, so if you see a keybinding or instructions telling you that <BS> does something, they aren't full of (bull)! It just means Backspace.

In addition to navigating and opening files, you can even make changes to the file system using Neo-tree. For example, to delete a file, you can move the cursor over that file and hit the `d` key. You'll be prompted with a popup window asking if you are sure; hit `y` and then `Enter` to confirm it:



To add a file or folder/directory, use the `a` key and enter a new name. Use a trailing slash (`/`) to indicate a folder. You can also use the `A` key in the explorer to add a folder without having to type a trailing slash.

The `r` key can be used to rename the file or folder under the cursor.

To copy or move a file, you can use Neo-tree's pseudo-clipboard. I say "pseudo-" because you can't use this to copy a file to be pasted in MacOS Finder or Windows Explorer; only to other places in the Neo-tree.

To *cut* a file with the intent of moving it somewhere else in the tree, use the `x` command. If, instead, you want to *copy* the file, use `y`. The mnemonic for `y` is `yank`, and is actually the same key you would use to copy text in the normal editor. To complete the move or copy, you'll need to navigate to the folder you want to move or copy it into and use the `p` key (which you may recall means "put" or "paste").

Neo-tree also has a Filter mode that I find quite clumsy; it's really just a cheap imitation Telescope/fzf.lua picker in a smaller window, so I recommend using your chosen picker instead. If you want to use Neo-tree's Filter mode, you can access it using `/` and enter some characters to limit the search results to files that match those characters. Then use the up and down arrows to navigate the list (`j` and `k` won't work here because you're in a sort of Insert mode context).

There is a *ton* of other cool stuff that Neo-tree can do. We will cover its use for buffer, git, and symbol navigation later, for example. In the meantime, you can use the `?` (mnemonic "ask question for help") key while the Neo-tree window is focused to get an overview, or `:help neo-tree` if you want to drink from the firehose.

The mini.files alternative

As I mentioned, I don't actually use Neo-tree for file navigation. I find that it feels kind of "foreign and un-vim-like". To me, it is a completely separate experience that just happens to be embedded in a Vim window. That said, I *also* don't like the tree view sidebar experience

in VS Code and the editors it emulates / is emulated by, so it's possible that tree views just aren't right for me.

These are just **my** opinions, and one of the golden rules of text editors is "all opinions are valid" (otherwise there would be war). A large number of Neovim users love Neo-tree, and you should use it if it matches your mental model.

That said, I'm clearly not alone in these opinions, because LazyVim optionally provides a different file management experience called `mini.files`. It is disabled by default.

> `mini.files` is part of a suite of fairly random Neovim packages known as `mini.nvim`. These plugins are largely independent from each other and provide a lot of common features that in many cases ought to ship with Neovim. Occasionally, the `mini.nvim` plugins are inferior to other plugins that they clone, but a number of them are best in class. `mini.files` is not the only `mini` plugin that ships with `LazyVim`, and we'll touch on others later.

The `mini.files` file manager is kind of like a Neovim-native experience of the columnar view that is popular in MacOs finder, among other file managers. The main reason I like it is that editing the directory listing is just like editing a normal text buffer. I don't have to remember that `a` means "after" in Normal mode, but it means "add file/folder" in Explorer mode. Instead, in mini.files, I use the `o` key to "create a new line below the current line", and then enter the file name in Neovim Insert mode. Later, I tell `mini.files` to sync my changes and it will create the file for the new row.

In order to use `mini.files`, you have to enable it as a Lazy Extra, similar to how we enabled the FZF picker. The exact steps are:

- Type `:LazyExtras<Enter>`
- Move your cursor to the line that contains `mini.files` (Seek mode is fastest)
- Press `x` to install the e**X**tra
- Wait a moment for the plugins to install
- Restart Neovim

Using mini.files

Once installed, you can show the mini.files view using `<Space>fm` and `<Space>fM`. By default, these are *not* quite the same as the `cwd/root` structure we've seen in Telescope, fzf.lua, and Neo-tree. Instead, they are listed in the `<Space>f` menu as follows:

```
m -> Open mini.files (Directory of Current File)
M -> Open mini.files (cwd)
```

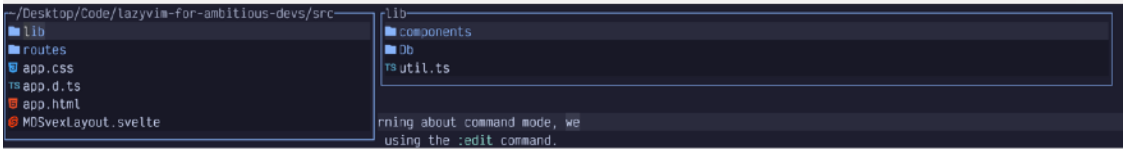
The default `mini.files` configuration doesn't have an open in root option. I like having the ability to open the directory of the currently open file, but I don't like *losing* the ability to open the root of the current project. I show how to address this in the next section where we discuss keybindings.

Instead of a sidebar, the `mini.files` menu shows up as columns of windows (known as Miller columns) side-by-side. For example, here's what happens when I open mini.files to the current working directory of this book:

This book is published as a svelte-kit app. The left-hand pane shows the current working directory, and the right pane shows the contents of the `src` directory, which is the "focused" folder in the left pane.

Interacting with `mini.files` is *very* similar to interacting with a standard vim window. You can use the `j` and `k` keys to move the cursor up and down. If this places your cursor over a folder, the contents of that folder will immediately show up to the right, and if it is over a file, you will see a preview of the file (by default, the previews are smaller than what I have in these screenshots; I'll show you how to change that in the next section if you have the screen real estate for it).

If you want to move "into" a folder to interact with the contents of that folder instead, simply press the `l` key to move "right". Here, I moved my cursor into the `src` folder, which immediately opened the file under the cursor in a new preview window.



Similarly, pressing `h` will move "out" of the current folder. If the cursor is in the left-most column, moving left will open a new left-most column, so you can navigate right up to the root of your file-system if you need to.

To open a file in the currently active Neovim window, press `l` on that file again. The behaviour here may be a bit surprising; the file will open *under* the `mini.files` view, but it won't hide the file menu. This allows you to open multiple files before closing the navigator (which can be done with the `q` key).

The beautiful thing about `mini.files` compared to `Neotree` is that the little windows act like normal editors, and all the navigation features you have become used to are available. For example `Seek` mode can be used to navigate to a file. Press the `s` key and then any number of characters you want to search for. Any matches to the typed characters will be labelled and you can jump to them by typing the indicated label.

Even modifying the filesystem is exactly the same as editing a normal buffer. We haven't really covered editing yet (I'm just as surprised as you are), but here's a quick overview:

- To rename a file or folder, navigate to the line that has it, and enter Insert mode to change or add text.
- Deleting a file or folder uses the command `dd` which is the keybinding to delete an entire line of text in normal Neovim windows.
- Copy a file or folder with `yy` the command to copy (yank) a line of text
- Put/paste a deleted **or** yanked file with `p`.

We'll discuss these commands and more in Chapter 6. The main point is that pretty much any navigation or editing command you learn in the future will work with `mini.files`.

Saving Filesystem Changes

Any modification that you make using these keybindings will not actually be saved on the filesystem until you type the = key, which is a (rare) `mini.files` specific keybinding. I think of it as meaning "make the filesystem *equal* to what I've typed". This will pop up a little window telling you what actions `mini.files` wants to take on your behalf, such as deleting, moving, renaming, or copying files. You can confirm or decline the changes with a `y` or `n` (yes or no, of course).

I encourage you to play with both Neo-tree and mini.files until you can make a decision as to which of the two you prefer. Eventually, you will arrive at one of the following conclusions:

- You prefer Neo-tree and don't need `mini.files`. In this case, revisit the LazyExtras mode and disable `mini.files` with the `x` key.

- You use Neo-tree for some interactions (possibly things we haven't covered yet, such as navigating git, buffers, or symbols) and `mini.files` for others. In this case, you are probably content with the default LazyVim configuration of the `mini.files` extra.
- You are *my kind of weird* and don't want to use Neo-tree at all, preferring only `mini.files`.This exact situation is discussed in the next chapter as we learn more about configuring plugins.

Summary

In this chapter, we learned not one, but four different ways to open files and interact with the filesystem in LazyVim: Telescope, fzf.lua, Neo-tree, and mini.files. Each provides a different mechanism for opening and managing files, and you will find some of them more comfortable than others.

As a side-effect of studying these filesystem tools, we got a tiny preview of configuring plugins and installing LazyVim extras. We will go into more detail of this in the next chapter.

Chapter 5: Configuration and Plugin Basics

Chapter 5: Plugin Basics - LazyVim for Ambitious Developers

I've mentioned plugins a few times previously and you even got to see the `lazy.nvim` plugin manager in action back in Chapter 1. LazyVim has a unique multi-layered approach to managing plugins that requires a bit of description, but is very elegant in practice.

Installing plugins allows you to configure Neovim to do things it can't do by default. Plugins are written in either Lua or VimScript (though most Neovim users prefer Lua-based plugins).

The Three Categories of Plugins in LazyVim

The simplest plugins to use in LazyVim are pre-installed by LazyVim itself. You've used many of them already. Some, such as Neo-Tree, Telescope, and lazy.nvim provide custom UI components to interact with them. Others, such as flash.nvim and which-key provide new commands or modes to work with. Still others operate quietly in the background auto-matching parenthesis or tags and drawing indent guides.

These plugins are preconfigured in LazyVim with (generally) sane defaults. Because they are so well integrated, customizing those defaults is doable, but sometimes requires a few tricks that we will cover in this and later chapters.

The second category of plugin in LazyVim are the so-called "Lazy Extras". These plugins are **not** enabled by default, but can be enabled with just a couple of keystrokes if you want them. Lazy Extras exist to make it easy to install popular plugins with a configuration that is guaranteed to play nicely with the other plugins that ship with LazyVim.

The third category includes third-party plugins that LazyVim has no awareness of. You will have to configure these plugins from scratch and do your own due diligence to ensure that keybindings and visual artifacts don't conflict with the plugins that LazyVim manages. In a non-LazyVim configuration, all plugins fell in this category, and it could be a headache to maintain as plugins evolved and fell out of use over time. In LazyVim, relatively few plugins fall in this category, so the whole experience is much more pleasant.

As a specific example consider these three Neovim plugins for file management, two of which we discussed in the previous chapter:

- `Neo-tree.nvim` ships with LazyVim and is active by default. The LazyVim configuration for Neo-tree does not conflict with other LazyVim plugins by default.
- `mini.files` ships as a Lazy Extra, and is basically a "one click" (or, since this is Vim we're talking about, one keypress!) install that is guaranteed to cooperate well with LazyVim.
- `oil.nvim` is an alternative plugin for filesystem management that LazyVim does not explicitly support. You can install it in LazyVim with a few lines of configuration, but it's not quite as easy to set up as `mini.files` and there is no guarantee it won't have conflicts you need to sort out yourself.

From Neovim's point of view, all these plugins are exactly the same, as NeoVim only knows about third-party plugins. LazyVim just comes with a bit of extra structure that you need to think about when using plugins. Usually this structure simplifies things, but occasionally it adds extra hassle.

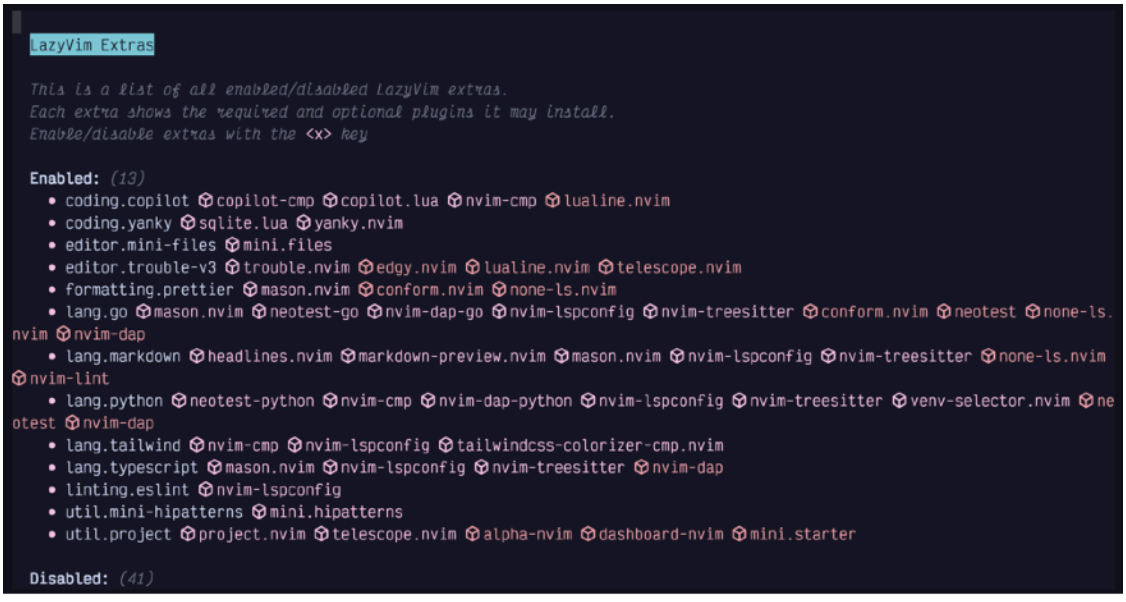
Lazy Extras

In the previous chapter, I covered how to use `mini.files`, but I was pretty terse on the installation instructions. Now we'll get to dive deep.

The Lazy Extras mode can be accessed by pressing `x` from the dashboard. If you aren't on the dashboard, you'll need to enter Command mode with `:` and type `LazyExtras` followed

by the usual `Enter` to confirm a command (Incidentally, you can also show the dashboard at any time by typing the command `:Dashboard`).

Either way, you'll be presented with a list of possible plugins to install. On my setup this looks as follows:



I've installed over a dozen extras at the moment, mostly for the various programming languages I dabble in. You can navigate this file using all the standard navigation commands such as `j` and `k s`.

No matter how you get there, once your cursor is on the extra you want to install (such as `editor.mini-files`) line, just hit the `x` key to install the extra. If you want to uninstall it, do the same thing; move to the appropriate line (now under the list of `Enabled` extras), and hit `x` to disable the extra. The mnemonic here, of course is that `x` means "E**x**tra".

You may need to quit and restart Neovim for `lazy.nvim` to pick up that the extra has been installed and sync it's dependencies.

While we're in the `LazyExtras` screen, I recommend enabling the `lang.*` extras for whichever programming languages you use most frequently. You should also install all the plugins in the "Recommended plugins" section (They have [?] icons beside them) except for:

- `ui.mini-animate` unless you have a much much faster machine than I do. This plugin is extremely laggy on me 2020-era Intel Imac Pro.

- `editor.fzf` unless you have decided that you prefer it to Telescope, as we discussed in Chapter 4.

I wouldn't install any other extras until you've either encountered them later in this book or had a chance to research them after you finish the book. Otherwise, they may change behaviours in ways that I won't have the foresight to write about.

You can find more information on each extra by visiting https://lazyvim.org and clicking the "Extras" menu item on the left menu bar. It includes links to the list of plugins each extra installs as well as the configuration LazyVim brings for that extra.

Disabling a Built-in Plugin

Sooner or later, you're going to want to edit your LazyVim configuration. The out-of-the-box defaults are wonderful, but the odds are that they don't 100% exactly match your personal needs (unless you are the LazyVim maintainer, in which case, I am compelled to say, "Hello, folke, I love LazyVim!").

While the vast majority of LazyVim's default plugins are no-brainers that you want to keep, you may find there are one or two plugins that you just don't need. In most cases, it doesn't actually matter, since LazyVim only loads plugins when you actually use them, so you can just ignore the ones that aren't relevant to you.

In my case, I have disabled only two core plugins:

- `Neo-Tree`, as I foreshadowed in the previous chapter
- `headlines.nvim`, which I find makes my markdown hard to interact with

The LazyVim configuration can be opened from the dashboard by simply pressing the `c` key . Or you can use Space Mode to access the configuration files at any time using `<Space>fc` for "Find Config Files".

This will load the lazyvim config folder in your file picker. This folder is typically `$HOME/.config/nvim`. Neovim loads `$HOME/.config/nvim/init.lua` by default, and if you weren't using LazyVim, this is where you would do all your configuration.

With LazyVim, `init.lua` just uses the lua `require` statement to include the LazyVim configuration infrastructure. **You will normally not have to touch this file**, even though most third party plugins have installation files that assume your configuration is in that file. Instead follow the "LazyVim way" as outlined in this chapter.

In addition to a barebones `init.lua`, LazyVim has put a few configuration files and a bit of folder structure in the configuration directory.

For now, the main thing we need to know is that *any* lua files inside the `lua/plugins` subdirectory will be automatically loaded by LazyVim, no matter what their name is. I have a number of different files in this folder for my custom configurations.

I call the one that holds my disabled plugins `disabled.lua`. The easiest way to create this file is to open one of the existing config files and use either neo-tree or mini.files to create a new file in the same folder, as described in the previous chapter.

When I created my `disabled.lua` file in the `lua/plugins` directory, my intention was to collect all the LazyVim plugins I don't want in it because I assumed LazyVim wouldn't quite match my needs. In reality, it's a really short list! The contents of this file is simply:

```
return {
  { "nvim-neo-tree/neo-tree.nvim", enabled = false },
  { "lukas-reineke/headlines.nvim", enabled = false },
}
```

If there are any other plugins that LazyVim enables by default that you don't want to use, just follow the same syntax. The first argument in each Lua table is a string containing the github repo (with owner) you want to disable. The second argument is to set `enabled = false`. That's it!

> **Tip**: You will inevitably forget the `return` statement at the beginning of a plugins file at some point. Now you know to watch out for it.

If you don't know the Lua language… honestly, don't worry about it. I've never formally studied it, but I've picked up enough by osmosis to easily maintain my Neovim configuration.

If you're less foolish than me, you might want to type `:help lua` and read the official Neovim docs on the topic. followed by `:help lua-guide-api`.

Modifying Keybindings (example)

Keybindings are one of the few things I don't love about working with LazyVim, although it's not strictly LazyVim's fault. I just never quite know where to define the damn keybindings.

There are basically three possible places to configure keybindings, depending on how any one plugin is configured:

- In `.config/nvim/lua/config/keymaps.lua`. This is where you configure or modify keybindings that are not specific to plugins, but rather modify core Neovim functionality.

- In the `keys` field of the lua table (in Lua, a "table" is like a combination of an array and a record or dict in many other dynamic languages) passed to a plugin. This is typically where you map global Normal mode keybindings to set up a plugin. This is what we will do with `mini.files`.
- In the `opts` (options) argument passed into a plugin's configuration. The format of the options for any one plugin are plugin-specific, but many plugins prefer to set up keymaps on your behalf through options instead of having you do the mapping yourself. This is especially true if the keymaps define a different "mode" or only apply if the plugin is currently open or active. I'll give an example of this with `mini.files` as well.

To demonstrate, I want to "fix" the fact that mini files doesn't have a "open in root" option. I like the "open in directory of current file" option, but I also want to be able to open in the root directory.

> Reminder: The root directory is the top level directory of the current project according to the existence of some language-specific file such as `package.json` or `Cargo.toml`. The `cwd` is the current working directory of the editor.

Since we've disabled `neo-tree`, I'm going to steal the `<Space>e` and `<Space>E` keybindings and reuse them for `mini.files`, then I'll remap the existing `<Space>fm` keybinding to open the root so I can access all three models. You can, of course, choose different keybindings if they map better to your mental model or you are keeping neo-tree around.

In a fit of "I'm so meta, even this acronym" (xkcd 917), I used mini.files to create a new file named `extend-mini-files.lua` in my `.config/nvim/lua/config/plugins/` directory. As with the `disabled.lua` file, this file can be named anything so long as it's in the `plugins` directory. I have a habit of prefixing any configuration that I am using to change the defaults provided by LazyVim with the word `extend`.

This makes it easy to distinguish it from non-LazyVim plugins I've installed when I'm listing the directory using `mini.files` or `Telescope`.

Inside this new file, I used this code:

```
return {
  "echasnovski/mini.files",
  keys = {
    {
      "<leader>e",
      function()
```

```
        require("mini.files").open(vim.api.nvim_buf_get_name(0), true)
      end,
      desc = "Open mini.files (directory of current file)",
    },
    {
      "<leader>E",
      function()
        require("mini.files").open(vim.loop.cwd(), true)
      end,
      desc = "Open mini.files (cwd)",
    },
    {
      "<leader>fm",
      function()
        require("mini.files").open(LazyVim.root(), true)
      end,
      desc = "Open mini.files (root)",
    },
  },
}
```

I constructed this by reading through the default configuration for the Telescope find files and Neo-tree plugins, which is conveniently provided on the LazyVim website.

To satisfy the contract with `lazy.nvim`, we need to return a Lua table, wrapped in curly braces. Lua tables can act as an array and a dictionary at the same time. In this case, the first element in the table is the string `"echasnovski/mini.files"`. It doesn't have a named key, so it's kind of like a "positional argument".

The second element in the table is more like a "named argument" in that it is indexed with the name `keys`, and the value is another Lua table. However, that second table acts more like an "array" of three values (three more separate lua tables) because it doesn't have named indices.

It is important to understand that the `keys` field is **merged** with the keys that are provided by the default LazyVim (extras) configuration for `mini.files`. If there are conflicts (such as with `<space>fm`), my values take precedence over the defaults.

This is a powerful feature of LazyVim that allows you to use hosted configuration provided by LazyVim but override it as needed. Older Neovim distros tended not to have this level flexibility, so you were either stuck with their configuration or had to copy the whole thing and edit it, which made updates a nightmare.

To be clear, `keys` is a LazyVim concept (technically, it's actually part of the underlying `lazy.nvim` plugin manager). Any plugin configuration can have a keys array table, and those keybindings will be merged with the default Neovim keybindings, the LazyVim keybindings, your custom global keybindings, AND any other plugin keybindings.

Yes, that's a lot of potential for conflicts, which is why I'm so glad LazyVim has done most of the configuration for me!

Structure of a keys entry

Each item in the `keys` table is another Lua table with (in this case) three fields. The first two fields are positional and represent the keybinding name and the lua callback function that gets called whenever that keybinding is invoked. The third field is a named field, `desc` and provides a string description that will be shown in the Space mode menu.

The keybinding sequence in the first entry is using a standard syntax that comes from Vim. Recall that `<leader>` is an old Vim concept that allows you to configure which key is used as the prefix for custom keybindings. In LazyVim (and indeed, for most modern Neovim users), the leader is `<Space>`. Special keys are indicated to Vim's keybinding engine using angle brackets, so you will often see notations such as `<Space>`, `<Right>`, `<Left>` or `<BS>`.

After the `<leader>` string, we include any additional keys that need to be pressed. For the simple ones, we have `e` and `E` to replace the `Neo-tree` keybindings we disabled with new `mini.files` keybindings. The third one is a bit more complicated, as the `f` indicates that this action will be available under the `file/find` submenu in Space mode, and the `m` indicates which letter will be in this menu.

For the callbacks, we use Lua functions, which always start with `function` and end with `end`. These are anonymous (unnamed) functions, and they don't accept any parameters inside the parentheses. In the function bodies, we call specific code to open `mini.files` the way we want. In two cases I just copied this code from LazyVim's default `mini.files` configuration, and in the third, I cobbled it together by combining code from the `Neo-tree` and `mini.files` configurations. The `LazyVim` global is a handy library with a collection of utility functions to aid with configuration. The `LazyVim.root` function is used to find the root of a project and returns a string that we pass to `mini.files.open`.

Customizing the mini.files Options

As I mentioned, the `keys` table is merged with the default `keys` table that LazyVim has configured for `mini.files`. Similarly, most Neovim plugins can be configured with an `opts` table that contains custom configuration specific to that plugin. If you supply an opts `table`, it will be *merged* with the default LazyVim one (if there is one).

You'll need to read each plugin's documentation (often available on Github, and usually available with `:help plugin-name`) to know exactly what options are available for it. You'll also need to review the default configuration that LazyVim is setting up for that plugin so you understand how it will merge.

In my case, I pass the following `opts` array to `mini.files`:

```
return {
  "echasnovski/mini.files",
  keys = {
    -- the keybindings from above
  },
  opts = {
    mappings = {
      go_in = "<Right>",
      go_out = "<Left>",
    },
    windows = {
      width_nofocus = 20,
      width_focus = 50,
      width_preview = 100,
    },
    options = {
      use_as_default_explorer = true,
    },
  },
}
```

The mappings table in `mini.files` is used to override the default keymappings that are active *while* the `mini.files` view is open. This is different from the *global* keymaps we defined earlier to open `mini.files`. In my case, I have mapped `go_in` and `go_out` to use the arrow keys instead of `h` and `l` because of the left-handed Dvorak Kinesis weirdness I described previously. I don't recommend you make this change; `h` and `l` will work better for most anybody who isn't me.

The window options are there because I have a 32" 6k monitor, which means I can afford to have larger-than-normal explorer columns. Refer to the `mini.files` help for more information on these options.

So now you know a little bit about configuring plugins in LazyVim. It is both a little bit easier and a little bit harder than configuring plugins from scratch:

- It is easier because you only need to change the values that are non-default, instead of setting up an entire configuration, and LazyVim comes with very sane defaults.
- But it is harder because you sometimes have to think about how the option and keybinding merging happens, which wouldn't be necessary if you just had one great big configuration object to begin with. This merging can get quite tricky for plugins that have complicated default LazyVim configurations.

Modifying Existing Options

Sometimes the "merging" behaviour LazyVim uses to overwrite options with the ones you provide in your plugin overrides is too simplistic. This most often happens when you are modifying a plugin that calls or defines a function for options behaviour instead of customizing it.

To support this situation, the `opts` entry in a `lazy.nvim` plugin's configuration table can be a function instead of a static table. The function accepts the previous `opts` table as it was configured by LazyVim as an argument. Your function needs to *modify* this table to suit your desired behaviour.

Note that it **does not** return a new `opts` table; it needs to modify the one that was passed in.

For example, the default LazyVim configuration for the `nvim-cmp` plugin is a pretty long and complicated function. The `nvim-cmp` plugin is responsible for the completion pop-up menu that provides suggestions as you type. It is an insanely useful feature, but I don't like that by default (in LazyVim), selecting a completion is done with an `Enter` keypress. This drives me nuts when editing text as I am right now because I press enter for new lines all the time, often ignoring the pop-up.

There are a couple recipes for modifying this behaviour in the LazyVim docs, and other recipes you can try in the `nvim-cmp` README. In my case, I've configured it as follows:

```
return {
  {
    "hrsh7th/nvim-cmp",
    opts = function(_, opts)
      local cmp = require("cmp")
      opts.mapping = vim.tbl_extend(
        "force",
        opts.mapping,
        {
          ["<Right>"] = cmp.mapping.confirm({ select = true }),
```

```
          ["<CR>"] = function(fallback) cmp.abort() fallback() end,
        }
      )
    end,
  },
}
```

This `opts` function accepts the LazyVim-defined `opts` table as its second parameter. My code changes those `opts` using the `tbl_extend` function provided by Neovim. I add a new `["<Right>"]` key to accept the suggestion (this matches fish shell behaviour) and overwrite the `<CR>` key with abort completion behavior.

This is harder to maintain than if I just had the whole configuration the way I wanted it in the first place, but easier to maintain than if I had to *write* that configuration from scratch in the first place. I am willing to accept that tradeoff for all the places that LazyVim configures things better than I would have done on my own.

Installing third-party plugins

Installing a third-party plugin is little different from configuring a Lazy-Vim provided plugin, except that you don't have to worry about how the keys and opts are merged with a default config.

Simply create a new Lua file in the `plugins` directory (named appropriately for the plugin). Inside the file, return a Lua table where the first entry is the GitHub repo and name of the plugin, with other configuration (`opts` and `keys`, among others) after that name.

For example, I like the `guess-indent.nvim` plugin to set my shift width based on the contents of the file I am currently editing. It is maintained by the github user `nmac427`, so my `plugins/guess-indent.lua` file looks like this:

```
return {
  "nmac427/guess-indent.nvim",
  opts = {
    auto_cmd = true,
    override_editorconfig = true
  },
}
```

The `opts` table depends entirely on what the plugin expects. In this case, I read the `guess-indent.nvim` README and found two options that I wanted to set.

Most modern Lua plugins will be documented as having to call a `setup` function with a Lua table containing the configuration. If the plugin you are trying to set up does not have explicit Lazy.nvim instructions, don't worry! Whatever gets passed into that `setup` function is what you need to include in the `opts` passed to the LazyVim plugin manager.

For example, another third-party plugin I recommend is `chrisgrieser/nvim-spider`, which subtly changes the `w`, `e`, and `b` commands to support navigating within CamelCase and snake_case words. I have a file named `nvim-spider.lua` in my `plugins` directory as follows:

```lua
return {
  "chrisgrieser/nvim-spider",
  keys = {
    {
      "w",
      "<cmd>lua require('spider').motion('w')<CR>",
      mode = { "n", "o", "x" },
      desc = "Move to end of word",
    },
    {
      "e",
      "<cmd>lua require('spider').motion('e')<CR>",
      mode = { "n", "o", "x" },
      desc = "Move to start of next word",
    },
    {
      "b",
      "<cmd>lua require('spider').motion('b')<CR>",
      mode = { "n", "o", "x" },
      desc = "Move to start of previous word",
    },
  },
}
```

This plugin doesn't automatically set up keybindings, so I pass a `keys` = table to the plugin configuration. This array is **not** passed to the plugin. Rather, the keys are parsed by the lazy.nvim plugin manager and added to the global keybindings. It is convenient to keep the keys with the plugin so all the configuration is in one place.

I am satisfied with the default options that `nvim-spider` passes to its `setup` function (after reading the README), so I don't have to pass an `opts` array.

The best resource for finding third-party plugins is the github repository rockerBOO/awesome-neovim. The list is well-maintained and (most importantly) pruned regularly, so there are no outdated or unmaintained plugins on the list.

In practice, LazyVim already ships with the best-in-class versions of most plugins (built-in or as extras), so you won't have to add many, but if you come across any "I wish LazyVim could…" scenarios, the answer is probably "it does and the plugin to do it is listed in the Awesome Neovim repo".

Summary

In this chapter, we learned a little bit about how LazyVim integrates with the wider Neovim plugin ecosystem. It provides sane default plugins and configuration, but makes it easy to customize that configuration for your own needs.

We learned that built-in, extras, and unknown third party plugins are all treated slightly differently (though consistently), and saw examples of how to install some of the plugins I personally find indispensable.

Now that you know how to open files and configure plugins, we can get back to some of the nuts and bolts of modal editing. You already know how to switch between Normal and Insert mode and you can navigate around your code. In the next chapter, we'll cover some basic editing features that blur the line between navigating and inserting text.

Chapter 6: Basic Editing

Chapter 6: Basic Editing - LazyVim for Ambitious Developers

Armed with the navigation keybindings you've already learned and the ability to enter and leave Insert mode at will, your Vim editing experience is getting pretty close to on par with what you might be used to in non-modal editors.

However, moving around and inserting text is a very small part of the life of a software developer. More often, you need to *edit* text. Deleting code, changing code, refactoring code moving code around. It's the majority of what we do.

Yes, you can do all of these things by navigating to where you want to, and entering insert mode. The delete and backspace keys do the same thing in insert mode that they do in other editors. But there are far more efficient tools.

The best part is that you already know most of what you need to take advantage of very powerful editing commands!

The Vim Command Mental Model

The navigation commands such as `s` and `f` and `hjkl` and `web` that you already know are collectively known as *motion* commands. They move the cursor from its current location to a new location.

Most motion commands can be prefixed with a count, so the navigation model is always `<count><motion>`. The effect of a count is usually to repeat the motion a certain number of times, although some commands such as `Shift-G` for "Go to line" will use the count as an absolute value instead. If the count is blank, the "default" count is typically `1`. Even a Seek command which uses labels is allowed to be prefixed with a count (although the count will be ignored).

The `<count><motion>` commands are great for navigation, which is all we've used them for so far, but they can also be combined with a *verb* to do something to the text between the cursor and the location the motion would move you to.

Verbs come first, so the structure is always `<verb><count><motion>`. Navigation is the "default" verb, so if you leave the verb blank (i.e. skip it), your cursor moves to the location indicated by the motion. We'll discuss several important verbs in this chapter.

But the model keeps growing! It turns out, verbs can also be counted. The syntax becomes `<count><verb><count><motion>`. I have never in my life used all four of those in one command, however. Typically you would either do `<count><verb><motion>` OR `<verb><count><motion>`.

This is starting to look like a full fledged grammar (spoiler alert: it is).

This model is nice because it allows you to divide and conquer your learning strategy, and reuse knowledge as you study. First you learned motion commands. Then you learned counts. Now you will learn verbs. If you learn new motion commands or new verbs in the future, you can mix them with all the verbs and motions you already know and they should behave in a predictable way.

Various plugins try to mimic this strategy, and, well, most are successful. My main complaint with `Neo-tree` is that it doesn't operate with the `<verb><motion>` mental model, while `Mini.files` does. Similarly, some folks argue that Seek mode violates the Vim mental model because counts don't make sense. My opinion is that Seek mode simply transcends counts, but it still combines cleanly with verbs so it is a valid Vim model.

A Note on Insert Mode

Like all models, this one is not perfect. For example, you can use counts with the `i`, `I`, `a`, and `A` commands, but it's clear that "enter insert mode" is neither a motion nor a verb.

For example, if you type `5ifoo<Escape>`, Neovim will insert `foofoofoofoofoo` for you. That may not seem very useful, but if you ever want an 80 character `*` ruler to underline a heading, `80i*<Escape>` is pretty nifty!

But the `<count>i` "not-motion" commands *cannot* be combined with verbs like the navigation commands you've learned, so it's important to know the limits of the model.

So now that you understand how the motions you already know can combine with verbs to perform actions other than navigation, you just need to learn some verbs.

Deleting Text

I've previewed this a couple times already, and even if I hadn't, you can probably guess that the verb for deleting text is `d`.

So where `motion` will take you to a specific location in the code, `d<motion>` will delete all the text between the cursor and that location. Here are some examples:

- `dh` to delete the character to the left of the cursor.
- `d3w` to delete three words.
- `3dw` to delete one word, three times.
- `d^` to delete from the cursor to the beginning of the line.
- `d2fe` to delete all text between the cursor location and the second `e` after the cursor, including that second `e`.
- `d2Ta` to delete all text between the cursor and the second `a` *behind* the cursor, *not* including that second `a`.
- `dsfoos` to delete text between the current cursor position and the label `s` that pops up when you use Seek mode to seek to `foo`. Note that Seek mode **always** jumps to the beginning of the word you searched for. This means that if the `foo` you jump to is after the current cursor location, the `oo` will not be deleted, but the `f` will. But if the `foo` you jump to is before the current cursor location, all three letters of `foo` will be deleted.

If any of those are surprising, ignore the `d` and refer back to earlier chapters to refresh your memory of the motions.

So `d` will work with all the motion commands you know, as well as all the motion commands you don't yet, **and** all the motion commands that are defined by plugins you haven't yet installed.

When the delete command is completed, Neovim will still be in Normal mode, and you can immediately perform any other `<verb><motion><pair>` combination.

Changing Text

Sometimes you just want to delete text, but another common task is editing text. Replace a word with another word, change spelling (coincidentally, I just misspelled "change"), delete the rest of the paragraph and replace it with something new, etc.

This can easily be handled by combining the delete verb with Insert mode (e.g. `dwi` will delete a word and enter insert mode.) However, you can save a keystroke by using the `c` verb, which means "**c**hange". If you replace the `d` in each of the examples I outlined above with a `c`, you will effectively get "delete the text and immediately enter insert mode."

Operating to end of the current line

It is very common to want to delete or change from the cursor position to the end of the current line, leaving the beginning of the line intact. These actions happen more often than you would expect in source code editing, so there is a shortcut for them.

Yes you could `d$` and `c$` to delete or change to the end of the line, since `$` is the "jump to end of line" motion. That is the "correct" format for the mental model. However, because this is such a common operation, you can "cheat" with one fewer keystrokes and just use the capitalized `D` or `C` instead.

Note that there is no inverse shortcut verb for "delete to the beginning of the line", so you'll have to use `d^` or `d0` instead, where `^` is the motion to jump to the first non-blank character and `0` is the motion to jump to the first column.

Operating on entire lines

Another common action is to change or delete an *entire* line of text. So much so, in fact, that there are special motions for "the whole line". These motions are accessed by duplicating the verb. This is another place where the mental model kind of breaks down; the interpretation of the motion *depends* on the verb.

In practice, this just means that `dd` deletes an entire line and `cc` deletes it and enters insert mode. These are nice and easy to type, so it makes for a nice shorthand.

You can combine these bespoke motions with counts. `d3d` will delete three lines, and `3dd` will delete one line three times (which is faster to type because you don't have to move your finger off of `d` to hit it twice). Yes, that has the same outcome either way, but the model is such that you can use either of them. Note that there are situations where the two formats may have subtly different behaviours, although in practice I have never encountered surprises.

Some shortcuts for modifying individual characters

Another common operation is to perform a delete or change operation on a single character or specific number of characters. You could do this using `dl` to delete the character under the cursor or `4dl` to delete that character and the three characters that come after it. However, because you do this so often, there is a shorthand verb that doesn't have a motion (or rather, the motion is implied): `x`. For example, you can use `x` to delete an extraneous `u` in words like behaviour if your editor is from the USA but you live in a member of the Commonwealth. The single letter will be deleted, and you'll be back in Normal mode ready to proceed.

The `x` command can be used with a count, so if you want to delete five characters starting with the one under the cursor, just use `5x`.

If you need to go the other direction and delete characters *before* the cursor, use a capital `X`. This, too, can have a count, and it will basically delete that many characters to the left. I rarely use this, since the shift brings us up to two keystrokes anyway, and `hx` or `d4h` is no harder.

If, instead of deleting, you need to replace a character with a different character, use the `r` command. This command will briefly enter insert mode while you type one character, then immediately return to normal mode. Much fewer keystrokes for a common operation (spelling errors are common, right? It's not just me?) than something like `cle<Escape>`. Using `r` with a count is possible, but the behaviour is kind of unhelpful: it will replace the character under the cursor and the appropriate number of characters after that character with the same letter. The only place I can imagine this being helpful is when you copy-paste a password prompt from somewhere and need to replace all the characters in the password with `*`.

Another common operation is deleting the newline at the end of the current line. Use the `J` (`j` stands for "Join Lines") command from anywhere in the line. I use this one a lot. If you need to merge multiple consecutive lines together, `J` takes a count. It generally does the right thing around whitespace (replacing indentation with a single space), but if you need to do a join without modifying whitespace, use the two-character combination `gJ`.

Manipulating Case

If you need to convert a character or sequence of characters to uppercase, use the verb `gU` (That's a shifted `U` for the second character) followed by any standard navigation motion. (Nobody said a verb had to be a single letter, though most are). I find this particular verb frustrating because `g` is normally assigned to the `Go To` motions. In this case, (as with `gJ` above) it is a verb instead.

I guess you can think of it as "Go To and Convert to Uppercase" where `U` is short for `Uppercase`.

The inverse function to convert all text between the current cursor position and the motion destination is to use a lowercase `gu` before the motion. Kind of weird to remember, but it does match the common Vim idiom of "u" means an action and "U" means the same action BUT BIGGER.

The duplicate commands `gUgU` and `gugu` do the same thing as other duplicate verbs, applying the upper/lower case operation to the entire line. It's a rather annoying sequence of keypresses, though, so it is probably better to use Visual mode (discussed in Chapter 8) to select a line with `V` and then apply the visual mode verbs `U` or `u`.

I don't find these commands very useful. I more frequently use the ~ command, which inverts the case of the character under the cursor.

> Tip: If you find yourself doing a lot of case switching work, have a look at the coerce.nvim. It doesn't have a LazyVim extra so you'll need to configure it yourself, but it can be worth the effort.

Repeating Commands

LazyVim doesn't have multiple cursor mode. There *are* plugins to support multiple cursors, but in my experience they don't work very well. Neovim does have multiple cursors on their roadmap, so I am hoping they will come up with a paradigm that integrates nicely with the Vim mental model.

In the meantime, Neovim provides several different tools available for performing an action in multiple places in your code. We'll cover basic repetitions here, and other useful techniques in later chapters.

Once you have performed any verb, you can navigate to another place in the document and repeat that verb with a single keypress: **.** (That's a period, although you will usually hear it referred to as "dot repeat" in this context).

This highlights why `d` and `c` need to be separate verbs, as opposed to using something like `d<motion>i`. When you use `c`, the delete motion **and** the text you inserted is remembered, so you can repeat the entire change with a `.` command. For example, if you want to replace all instances of a variable named `i` with a much better name of `index`, you could jump to the first instance of `i` and type `clindex<Escape>` to "change one character to index". Then you can use Seek mode or other navigation commands to go to the next use of `i`. Now just type `.` to repeat the change and continue to the next instance.

Like motions and verbs, the `.` command can be given a count. However, counts with `.` are a little bit nuanced. Rather than blindly repeating the command `<count>` number of times, it will instead *replace* the count of the command being repeated.

This means that if you use the verb `3dd` to delete three lines, and the next operation you perform is `2.` ("2 dot"), the second operation will delete *two* lines, rather than six.

Recording Commands

Vim's command recording and playback system is extremely powerful. You can trivially record an arbitrary sequence of navigation, editing, and insertion commands, then repeat that sequence on demand at any location you want.

To start a recording, press `qq`. Sorry, but I have no mnemonic to remember `q`. I have a feeling it was just the last available key on the keyboard!

After that, type whatever sequence of navigation, editing, and insertion commands you want to record. Delete words, insert text, change text, search for words (don't use Seek mode, as the replay mechanism will have no idea what label to jump to). Virtually anything you can do in Vim (even `:` commands) can be recorded and replayed later.

When you are finished recording, just press `q` again. The recording will be stored ready for replay whenever you desire.

Appending to a recording

If you partially complete your recording and then realize you need some more information or need to make an edit before completing the recording, you can stop the recording using `q` as usual and do the thing you need to do.

When you are ready to continue recording, use `qQ` to record in *append* mode instead. The main tip here is that you need to make sure your cursor is in a location such that the merged recording will make sense. This usually means the same place it was when you stopped recording, although it may depend on what changes you made in the meantime.

Playing Back a Recording

The easiest and fastest way to play back your most recently saved recording is with a capital `Q`.

It is possible to store and replace multiple recordings at once using *registers* (a stupid name for a storage location that harkens back to humanity's dark days of assembly programming). I will go into more detail about registers in a later chapter.

Undo and Redo

Obviously, these are the most important operations in the whole book! Use the `u` key to undo your most recent change. Note that "most recent change" can be a pretty big whack of text, especially if you haven't exited Insert mode for a while. For example, I wrote this entire paragraph in one Insert session. If I press `u` the entire paragraph will be lost.

That's ok, though, because I can redo using `Control-r`. Like most developers, I use both of these extensively. (Did you know that in the old days of typewriters, secretaries had to get 100% accuracy scores on their typing tests? There was no backspace or delete key, you see).

Neovim actually does an amazing job of keeping track of your entire history, rather than just the most recent suite of changes. So if you make a bunch of changes to get to state B, then undo to state A, and then make a bunch more changes to get to state C, it is still possible to get back to state B (ie: back out of the C changes to state A and go back up the B changes to state B).

It's kind of the same concept as git branches, except your history is *automatically* tracked for every keystroke you make. Working with branches of undo history using raw Neovim commands can feel pretty clumsy, though (read through `:help undo-branches` if you're brave). Instead I recommend configuring and installing the undotree plugin.

About 99.9% of the time, `u` and `Control-r` will be all you need, but that remaining 0.1% can make `undotree` a godsend when you need it.

Summary

In this chapter, we expanded our understanding of the Vim mental model, and then introduced several verbs that can be combined with the navigation motions we were already familiar with.

We discussed a grab-bag of other editing commands before covering how to repeat motions using `.` and command recordings. Finally, we covered undo and redo.

In the next chapter, we'll learn about text objects and some additional nuances of the Vim mental model with operator-pending mode. Combined, these allow us to very quickly perform actions on a huge variety of code concepts.

Chapter 7: Objects and Operator-pending Mode

Chapter 7: Objects and Operator-pending Mode - LazyVim for Ambitious Developers

The navigation and motion commands we've learned so far are certainly very useful, but Neovim also comes with several more advanced motions that can supercharge your editing workflow. LazyVim further amends collection of motions with other powerful navigation capabilities powered by a variety of plugins.

For example, if you are editing text (as I am right now) rather than source code, you will find it useful to be able to navigate by sentences and paragraphs. A sentence is basically anything that ends with a `.`, `?`, or `!` followed by whitespace.

The sentence keybindings are two of the hardest for me to remember. I use it rarely enough that it hasn't become muscle memory, and it doesn't have a good mnemonic I can remember.

Have I built enough suspense? Pay attention, because you will forget this. To move one sentence forward (to the first letter after the whitespace following sentence ending punctuation), type a `)` (right parenthesis) command in normal mode. To move to the start of the current sentence, use `(`. Press the parenthesis again to move to the next or previous sentence or add count if you want to move by multiple sentences.

I hate that the command is `(` since that feels like it should be moving to, you know, a parenthesis! But it's not; it's moving by sentences. Since the `.`, `!`, and `?` characters rarely mean "sentence" in normal software development, I just don't use it that much until I start writing a book (something I keep telling myself I won't commit to again, but it never lasts).

I do use the paragraph motions all the time, though. A paragraph is defined as all the content between two empty lines, and that is a concept that makes sense in a programming context. Most developers structure their code with logically connected statements separated by blanks. The commands to move up or down by one "paragraph" are the curly braces, `{` and `}`, and if you need to jump multiple paragraphs ahead or back, they can, as usual, be prefixed by a count.

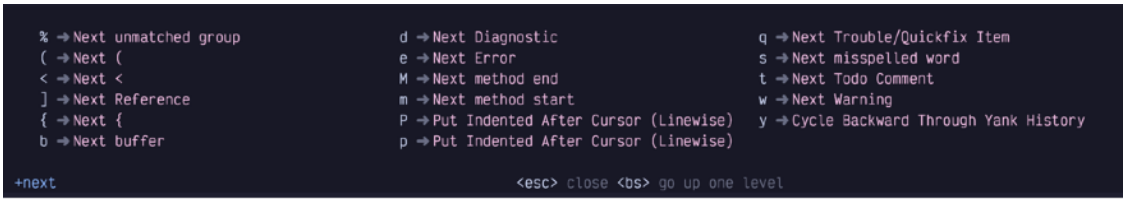
Again, you might expect `{` to jump to a curly bracket, so it is a bit annoying that it means "empty line" instead, but once you get used to it, you'll probably reach for it a lot.

Unimpaired Mode

LazyVim provides a bunch of other motions that can be accessed using square brackets. It will take a while to internalize them all, but luckily, you can get a menu by pressing a single `[` or `]`. Like the sentence and paragraph motions, the square brackets allow you to move to the previous or next *something*, except the *something* depends what key you type after the square bracket.

Collectively, these pairs of navigation techniques are sometimes referred to as "Unimpaired mode", as they harken back to a foundational Vim plugin called vim-unimpaired by a famous plugin author named Tim Pope. LazyVim doesn't use this plugin directly, but the spirit of the plugin lives on.

Here's what I see if I type `]` and then pause for the menu:



Not all of these are related to navigation, and one of them is only there because I have a Lazy Extra enabled for it. We'll cover the motion related ones here.

First, the commands to work with `(`, <, and `{` are quite a bit more nuanced than they look. They **don't** blindly jump to the next (if you started with `]`) or previous (if you used `[`) parenthesis, angle bracket, or curly bracket. If you wanted to do that, you could just use `f(` or `F(`.

Instead, they will jump to the next *unmatched* parenthesis, angle bracket, or curly bracket. That effectively means that keystrokes such as `[(` or `]}` mean "jump out". So if you are in the middle of a block of code surrounded by `{}` you can easily jump to the end of that block using `]}` or to the beginning of it using `[{`, no matter how many other curly-bracket delimited code blocks exist inside that object. This is useful in a wide variety of programming contexts, so invest some time to get used to it.

As a shortcut, you can also use `[%` and `]%` where the `%` key is basically a placeholder for "whatever is bracketing me." They will jump to the beginning or end of whichever parenthesis, curly bracket, angle bracket, or square bracket you are currently in.

That last one (square bracket), is important, because unlike the others, `[[` and `]]` do *not* jump out of square brackets, so using `[%` and `]%` is your only option if you need to jump out of them.

Jump by Reference

Instead of jumping out of square brackets as you might expect, the easy to type `[[` and `]]` are reserved for a more common operation: jumping to other references to the variable under the cursor (in the same file).

This feature typically leverages the language server for the current language, so it is usually smarter than a blind search. Only actual uses of that function or variable are jumped to instead of instances of that word in the middle of other variables, types, or comments as would happen with a search operation.

As an added bonus, LazyVim will automatically highlight other variable instances in the file so you can easily see where `]]` or `[[` will move the cursor to.

Jump by language features

The `[c`, `]c`, `[f`, `]f`, `[m`, and `]m` keybindings allow you to navigate around a source code file by jumping to the previous or next class/type definition, function definition, or method definition. The usefulness of these features depends a bit on both the language you are using and the way the Language Service for the language is configured, but it works well in common languages.

By default, those keybindings all jump to the *start* of the previous or next class/function/method. If you instead want to jump to the *end*, just add a `Shift` keypress: `[C`, `]C`, `[F`, `]f`, `[M`, and `]M` will get you there.

Note that these are *not* the same as "jump out" behaviour: if you have a nested or anonymous function or callback defined inside the function you are currently editing, the `]F` keybinding will jump to the end of the nested function, not to the end of the function after the one you are currently in.

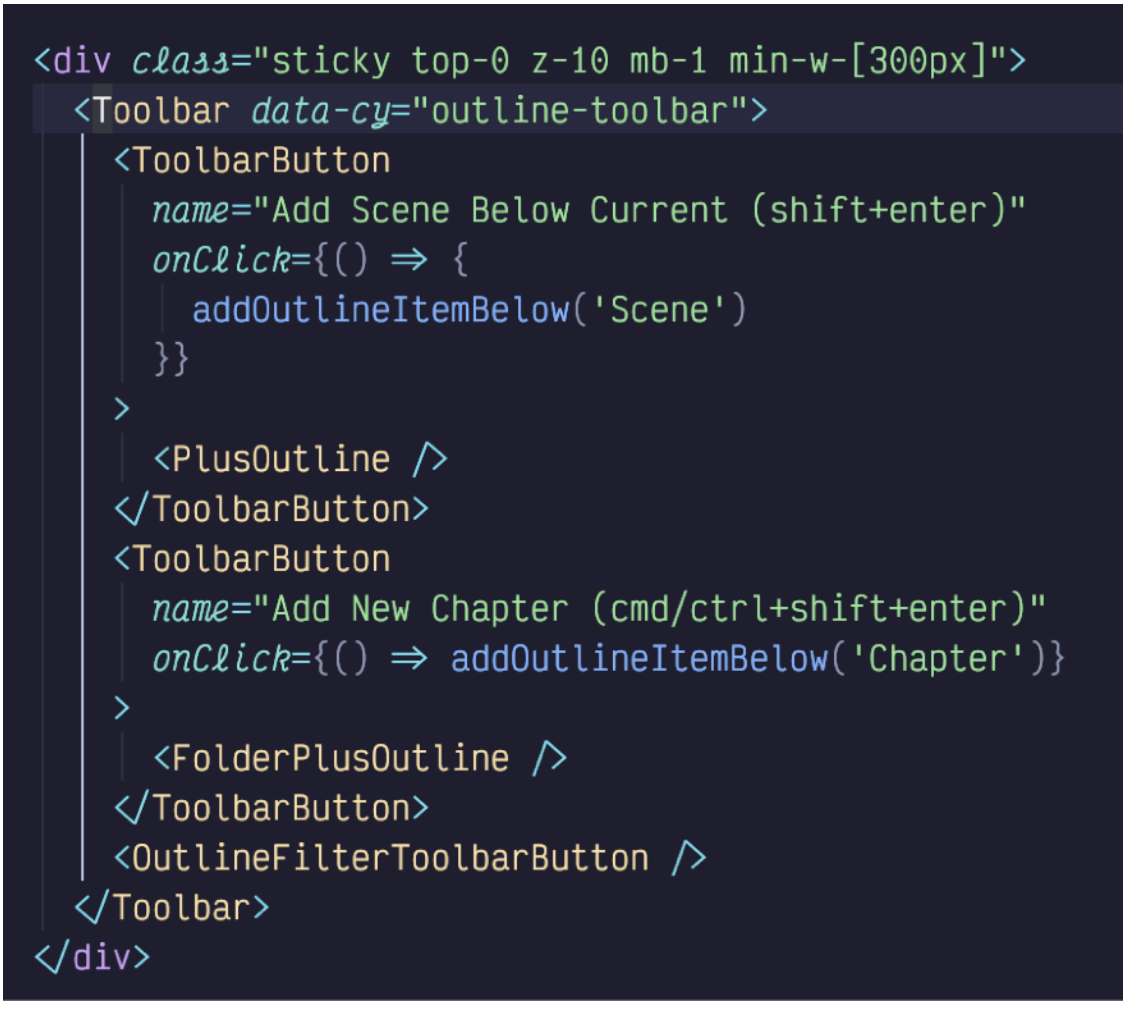
I personally don't use these keybindings very much as there are other ways to navigate symbols in a document that we will discuss later. But if you are editing a large function and you want to quickly jump to the next function in the file, `]f` is probably going to get you there faster than using `j` with a count you need to calculate, or even a `Control-d` followed by `s` to go to seek mode.

Jump to end of indention

If you are working with indentation based code such as Python or deeply nested tag-based markup such as HTML and JSX you may find the `mini.indentscope` extra helpful. You can install it by visiting `:LazyExtras` and searching for `indentscope`.

This plugin provides the `[i` and `]i` pairs.

LazyVim is configured with a plugin called `indent-blankline` which helps you visualize the levels of indentation in a file. Here's an example from a Svelte component I was working on recently:



This Svelte code uses two spaces for indentation. Each level of indentation has a (in my theme) grey vertical line to help visualize where that indentation level begins and ends, and the "current" indentation level is highlighted in a higher contrast colour.

If I enable the `mini.indentscope` plugin, these lines get a pretty animation when I move the cursor. More importantly,it adds the unimpaired commands `[i` or `]i` to jump out of the current indentation level ; it will go either to the top or the bottom of whichever line is currently black.

I use this functionality all the time when editing Python code and Svelte components. I use it less often in other languages where `[%` and `]%` tend to get me closer to where I need to go next. But the visual feedback of indent guides can be super helpful, even in bracket-heavy languages; I may be surprised by which curly bracket I will "jump out" to, but the indent guides are always obvious.

Jumping to diagnostics

I don't know about you, but when I write code, I tend to introduce a lot of errors in it. Depending on the language, LazyVim is either preconfigured or can be configured to give me plenty of feedback about those errors, usually in the form of a squiggly underline (If you aren't seeing squiggly underlines, go back to Chapter 1 and pick a better terminal).

These squiggly underlines are usually created by integration with compilers, type checkers, linters and even spell checkers, depending on the language. Some of them are errors, some are warnings, some are hints. Some are just distractions, but most of them are opportunities to improve your code.

Because I am so incredibly talented at introducing problems in my code, a common navigation task I need to perform is "jump to the next squiggly line". Collectively, these are referred to as "diagnostics", so the key combinations are `[d` and `]d`. If you only want to focus on errors and ignore hints and warnings, you can use `[e` and `]e`. Analogously, the `[w` and `]w` keybindings navigate between only warnings.

Misspellings can be found with `[s` and `]s`. This tripped me up when I started this book because I expected the `]d` to take me to the squiggly underlines under misspelled words, but it doesn't. I need `]s` instead.

Finally, if you use `TODO` or `FIXME` comments in your code, you can jump between them using `[t` and `]t`.

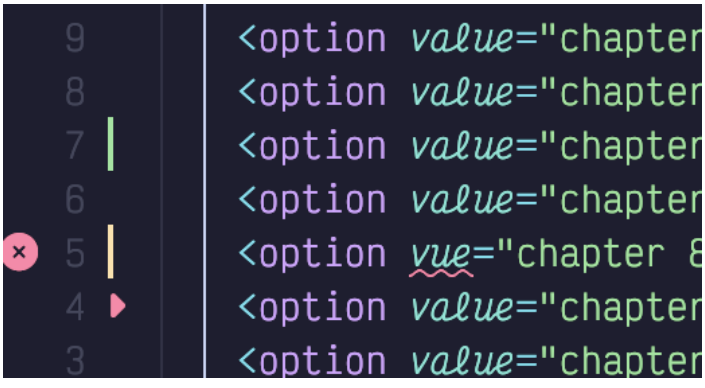
Note that unlike most of the previous `]` and `[` keybindings, it is not possible to combine diagnostic jumps with a verb. So `d[d` will not delete from the current location to the nearest diagnostic. This is (probably) just an oversight in how LazyVim defines the keybindings.

Jumping to git revisions

This is actually my favourite of the square bracket pairs: `[h` and `]h` allow you to jump to the next git "hunk". If you aren't familiar with the word (or if you're from a generation that thinks it means a gorgeous man), a "git hunk" just refers to a section of a file that contains modifications that haven't been staged or committed yet.

A lot of my editing work involves editing a large file in just three or four places. For example, I might add an import at the top of the file, an argument to a function call somewhere else in the file, and change the function that receives that argument in a third place. Once I've started editing, I may have to jump back and forth between those locations. `]h` and `[h` are *perfect* for this, and I don't need to remember my jump history or add named marks (essentially bookmarks) to do it.

Even better, LazyVim gives you a simple visual indicator as to which lines in the file have been modified, so you have an idea where it's going to jump. Have a look at this screenshot:



On the left side, to the right of the line numbers, you can see a green vertical bar where I inserted two lines, an orange bar where I changed a line, and a small red arrow indicating that I deleted a line. (as a bonus, it also shows a diagnostic squiggly and a red x to the left of the line numbers on the where my modification introduced the error). If I place my cursor at the top of the file and type `]h` three times, I will jump between those three places.

Like diagnostics, `[h` and `]h` cannot be combined with a verb.

Text Objects

Combining verbs with motions is very useful, but it would often be more helpful to combine those same verbs with *objects* instead of motions. Vim comes with several common objects, such as words, sentences and the contents of parenthesis. LazyVim adds a ridiculous pile of other text objects.

The grammar for objects is `<verb><context><object>`. The verbs are the same verbs you have already learned for working with motions, so they can be `d`, `c`, `gU`, etc.

The context is always either `a` or `i`. As you know, these are two commands to enter Insert mode from Normal mode. But if you have already typed a verb such as `d` or `c`, you are technically not in Normal mode anymore!

You are in the so-called "Operator Pending Mode". The navigation keystrokes you are familiar with are generally also allowed in operator pending mode, which is the real reason you can perform a motion after a verb. But if a plugin maintainer neglects to define the operator-pending keymaps, you end up with situations where you can navigate but not perform a verb, as we saw with git hunks and diagnostics above.

It doesn't make sense to switch to Insert mode after an operator, so the `a` and `i` keystrokes mean completely different things. Typically, you can think of them as `around` and `inside` (though in my head I always just pronounce them as "a" and "in"). The difference is that `a` operations tend to select everything that `inside` selects **plus** a bit of surrounding context that depends on the object that is defined.

For example, one common object is the parenthesis: `(`. If you type the command `di(`, you will delete all the text inside a matched pair of parenthesis. But if you instead type `da(`, you will delete all the text inside the parenthesis as well as the `(` and `)` at each end.

To see a list of all possible text objects in LazyVim, type `da` and pause. Here's what I see:

Let's cover (most of) these in detail next.

Textual objects

The operators `w`, `s`, and `p` are used to perform an operation on an entire word, sentence, or paragraph, as defined previously: word is contiguous non-punctuation, sentence is anything that ends in a `.`, `?`, or `!`, and paragraph is anything separated by two newlines.

The difference between `around` and `inside` contexts with these objects is whether or not the surrounding whitespace is also affected.

For example, consider the following snippet of text and imagine my cursor is currently at the | character in the middle of the word `handful` in the second sentence:

```
This snippet contains a bunch of words. There are a hand|ful of
sentences.

And two paragraphs.
```

If I want to delete the word `handful` while I'm at that character, I *could* type `bde` to jump to the back of the word, then delete to the end of the word. Or I can use the `inside word` text object and type `diw`.

Either way, I end up with an extra space between `a` and `of` because `diw` is `inside` the word and doesn't touch surrounding whitespace.

If I instead type `daw`, it will delete the word and *one* surrounding space character, so everything lines up correctly afterward with a single space between `a` and `of`.

There is also a `W` (capitalized) operator that has a similar meaning to the captial `W` when navigating by words: It will delete everything between two whitespaces instead of interpreting punctuation as a word boundary.

Similarly, I can use `dis` and `das` from that same cursor position to delete the entire "There are a handful of sentences." sentence. The former won't touch any of the whitespace before `The` or after the `.`, while the latter will sync up the whitespace correctly.

Finally, I can delete the entire paragraph with `dip` or `dap`. The difference is that in the former case, the blank line after the paragraph being deleted will still be there, but in `around` mode, it will remove the extra blank.

Typically, I use `i` when I am changing a word, sentence or paragraph, with a `c` verb, since I want to replace it with something else that will need to have surrounding whitespace. But I use `a` when I am deleting the textual object with `d` because I don't intend to replace it, so I want the whitespace to behave correctly as if that object never existed.

Quotes and Brackets

The operators `"`, `'`, and `` ` `` operate on a string of text surrounded with double quotes, single quotes or backticks. If you use the command`ci"`, you will end up with your cursor in Insert mode between two quotation marks, where everything inside the string was removed. If you use `da"`, however, it will delete the quotation marks as well.

As a shortcut, you can use the letter `q` as a text object and LazyVim will figure out what the nearest quotation mark is, whether single, double, or backtick, and delete that object. I don't use this, personally, but I guess it would save a keypress on double quotes.

Similarly if you want to apply a verb to an entire block contained in parenthesis or curly, angle, or square brackets, you just have to type one of those bracketing characters. Consider, these examples: `di[`, `da(`, `ci{` or `ca<`. As with quotes, the `i` versions will leave the surrounding brackets intact, and the `a` version will delete the whole thing.

The shortcut to select whatever the nearest enclosing bracket or parenthesis type is is the `b` object. (Mnemonic is "bracket").

These actually work with counts so you can delete the "third surrounding curly brackets" instead of the "nearest surrounding curly brackets" if you want to. I can never remember where to put the count, though! If your memory is better than mine, the syntax is to place the count **before** the `a` or `i`. So for example, `d2a{` will delete everything inside the second-nearest set of curly brackets. I'm not sure if that makes sense, so here's a visual:

```
class Foo {
    function bar() {
       let obj = {fizz: 'buzz'}
    }
}
```

If my cursor is on the colon between `fizz` and `'buzz'` then you can expect the following effects:

- `di{` will delete `fizz: 'buzz'` but leave the surrounding curly brackets.
- `c2i{` will remove the entire `let obj =` line and leave my cursor in Insert mode inside the curly brackets defining the function body.
- `c2a{` will do the same thing, but *also* remove those curly brackets, so I'm left with a `function bar()` that has no body.
- `d3i{` will remove the entire function and leave me with an empty `Foo` class.

You can also delete things between certain pieces of punctuation. For example, `ci*` and `ca_` are useful for replacing the contents of text marked as bold or italic in markdown files.

If you want to operate on the entire buffer, use the `ag` or `ig` text object. So `cag` is the quickest way to scrap everything and start over and `yig` will copy the buffer so you can paste it into a pastebin or chatbot. The `g` may seem like an odd choice, but it has a symmetry to the fact that `gg` and `G` jump to the beginning or end of the file. If you need a mnemonic, think of `yig` as "yank in global".

Language features

LazyVim adds some helpful operators to perform a command on an entire function or class definition, objects, and (in html and JSX), tags. These are summarized below:

- `c`: Act on class or type
- `f`: Act on function or method
- `o` Act on an "object" (the mnemonic is a stretch) such as blocks, loops, or conditionals
- `t` Act on a HTML-like tag (works with JSX)
- `i` Act on a "scope", which is essentially an indentation level (only available if the aforementioned `mini.indentscope` extra is installed)

Git Hunks

Remember the git hunks we discussed in unimpared mode? You can act on an entire hunk with the `h` object. So one way to quickly revert an addition is to just type `dih`. But you probably won't do this much as there are better ways to deal with git, as we will discuss in Chapter 15.

Next and last text object

The text object feature is great if you are already inside the object you want to operate on, but LazyVim is configured (using a plugin called `mini.ai`) so that you can even operate on objects that are only *near* your cursor position.

Once installed, the next and last text objects can be accessed by prefixing the object you want to access with a `l` or `n`.

Consider the `Foo.bar` Javascript class again:

```
class Foo {
    function bar() {
       let obj = {fizz: 'buzz'}
    }
}
```

If my cursor is on the `o` of the `let obj` line, I can type `cin{` to delete the contents of the `fizz: 'buzz'` object and place my cursor there in insert mode. I can save myself an entire navigation with just one extra `n` keystroke. I think this is a really neat feature, but I tend to forget it exists… Hopefully writing about it here will help me remember!

Seeking Surrounding Objects

The `flash.nvim` plugin that gave us `Seek` mode, has another trick up it's sleeve: the holy grail of text objects. After specifying a verb, you can use the `S` key (there is no `i` or `a` required) to be presented with a bunch of paired labels around the primary code objects surrounding your cursor.

As an example, I'm going to lean on that `Foo` class again. I have placed my cursor on the `:` and typed `cS`. The plugin identifies the various objects surrounding my cursor and places labels at both ends of each object:

The labels in this image are in green, and (typically) go in alphabetical order from "innermost" to "outermost". The primary difference from Seek mode is that each label comes in pairs; there are two `a` labels, two `b` labels, and so on. The text object is whatever is between those labels.

If the next character I press is `a` (or enter, to accept the default), then I will change everything inside the curly brackets defining the `obj`. If I press `b`, it will also replace those curly brackets. Pressing `c` will change the entire assignment and `d` will change the contents of the function. Hitting `e` replaces the curly brackets as well, and `f` changes the full function definition. The `g` label is the contents of the class, while `h` changes the entire class.

This is a super useful tool when you need to change, delete, or copy a complex structure that does not immediately map to any of the other objects.

Seeking Surrounding Objects Remotely

The `S` Operator-pending mode is useful for acting on objects that surround the cursor, but if your cursor isn't currently within the object you want to select, it won't suffice. You could use `s` to navigate to inside the object followed by `S` to select it, but you can save yourself a few keystroke by instead using the `R` operator.

With a mnemonic of "**R**emote", `R` is easy to use, but hard to explain. It is an operator-pending operation, so you need to type a verb, first, followed by `R` (as with `S` , there is no `i` or `a`) required.

At this point, LazyVim is essentially in Seek mode, so you can type a few characters from a search string to find matches anywhere on the screen. However, instead of showing a single label at any matches for the string you searched for, `flash.nvim` will automatically switch to surrounding object mode, and show pairs of labels of all constructs that surround the matching locations.

To put the icing on the cake, you can also perform a remote seek on any kind of object without using the surround mode. In this case, you would type a verb followed by a lowercase `r` (it still means "remote"). This will also put you in Seek mode, and you can start typing the matching characters. Single (normal Seek mode, rather than Surround Seek mode) labels will pop up, and you can enter a character to temporarily move your cursor to that label, just like normal Seek mode. But when your cursor arrives there, it is automatically placed in Operator-pending mode again. So you can now type any other operator such as `aw` or `i(`. Once the operation completes, your cursor will move back to where it was before you entered the remote seek mode.

As a specific example, the command `drAth2w` will delete two words starting at the word "At" that gets the label `h`, then jump your cursor back to the position it was at before you started the delete. In other words, it is the same as the command `sAthd2w<Control-o>`, which will seek to the word "At" at label h, then delete two words, and use `Control-o` to jump back to your previous history location. The remote command is a little shorter, but it's another one that I tend to forget to use. My brain goes into "move the cursor" mode before it figures out "delete" mode, so by the time I realize I could have done it remotely, it's too late.

Operating on surrounding pairs

We've already seen the text objects to operate on the contents of pairs of quotation marks or brackets, but what if you want to keep the content but change the surrounding pair?

Maybe you want to change a double quoted string such as `"hello world"` to a single quoted `'hello world'`. Or maybe you are changing a `obj.get(some_variable)` method lookup to a `obj[some_variable]` index lookup, and need to change the surrounding parenthesis to square brackets.

LazyVim ships with the `mini.surround` plugin for this kind of behaviour, but it's not installed by default. It is a recommended extra, so if you followed my suggestion to enable all the recommended extras, you may have it already. The keybindings for this

"surrounding" mode kind of violate the Vim mental model, in my opinion, but I'll teach you the default keybindings first, then show you how to change them.

Adding surrounding pair

The default verb for adding a surrounding pair is `gsa`. That will place your editor in operator-pending mode, and you now have to type the motion or text object to cover the text you want to surround with something. Once you have finished inputting that object, you need to type the character you want to surround it with, such as `"` or `(` or `)`. The difference between the latter two is that, while both will surround the text with parentheses, the `(` will also put extra spaces inside the parentheses.

That may sound complicated, but it should make sense after you see some examples:

- `gsai[(` will select the contents of a set of square brackets (using `i[`) and place parentheses separated by spaces inside the square brackets. So if you start with `[foo bar]` and type `gsai[(`, you will end up with `[( foo bar )]`.
- `gsai[)` does the same thing, except there are no spaces added, so the same `[foo bar]` will become `[(foo bar)]`.
- `gsaa[)` will place the parentheses *outside* the square brackets, because you selected with `a[` instead of `i[`. So this time, our example becomes `([foo bar])`.
- `gsa$"` will surround all the text between the current cursor position and the end of the line with double quotation marks.
- `gsaSb'` will surround the text object that you select with the label `b` after an `S` operation with single quotation marks.
- `gsaraa3e*` will surround the remote object that starts with `a` that is labelled with an `a` followed by the next three words with an asterisk at each end of the three words.

Depending on the context it can be a lot of characters to type, but it's typically fewer characters than navigating to and changing each end of the pair independently.

Delete surrounding pair

Deleting a pair is a little easier, as you don't need to specify a text object. Just use `gsd` followed by the indicator of whichever pair you want to remove.

So if you want to delete the `[]` surrounding the cursor, you can use `gsd[`.

If you want to delete deeply nested elements, you need to put the count *before* the verb. So use `2gsd{` to delete the second set of curly braces outside your current cursor position. As a specific example, if your cursor is inside the `def` of the string `{abc {def}}`, type

`2gsd{` will result in `abc {def}`, leaving the "inner" curly braces around `def`, but removing the second outer set around the whole.

Replace surrounding pair

Replacing is similar to deleting, except the verb is `gsr` and you need to type the character you want to replace the existing character with *after* you type the existing character.

So if you have the text `"hello world"` and your cursor is inside it, you can use `gsr"'` to change the double quotes to single quotes: `'hello world'`.

Navigate surrounding characters

Performing operations on surrounding pairs or on the entire contents of the pair is convenient, but sometimes you just want to move your cursor to the beginning or end of the pair. You can often do this using Seek mode, Find mode (e.g `f)` will jump to the nearest closing parenthesis) or the Unimpaired mode commands such as `[(`, but there are other commands that are more syntax aware if you need them.

The easiest one has been built into Vim for a long time. If your cursor is currently on the beginning or ending character of a parenthesis, bracket, or curly brace pair, just hit `%` to jump to its mate at the other end of the selection. If you use `%` in Normal mode when you aren't on a pair, it will jump to the nearest enclosing pair-like object. This only works with brackets, though, so arbitrary pairs including quotes are not supported.

The `mini.pairs` plugin comes with `gsf` and `gsF` keybindings, which can be used to move your cursor to the character in question. I don't use these much (too much typing!), and the `mini.ai` plugin that provides a similar feature using the `g[` and `g]` shortcuts. These shortcut need to be followed by a character type, so e.g. `g[(` will jump back to the nearest surrounding open parenthesis, and `g]]` will jump to the nearest closing square bracket. If you give it a count, it will jump out of that many surrounding pairs.

Highlighting surrounding characters

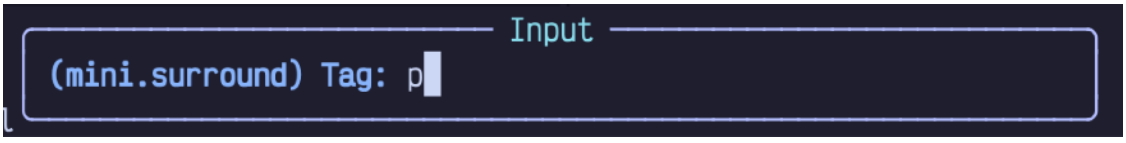
If you just need to double check where the surrounding characters are, you can use something like `gsh(`, where `h` would mean "highlight". This can sometimes be used as a dry run for a delete or replace operation that is using counts so you can double check that you are operating on the pairs you think you are.

Bonus: XML or HTML Tags

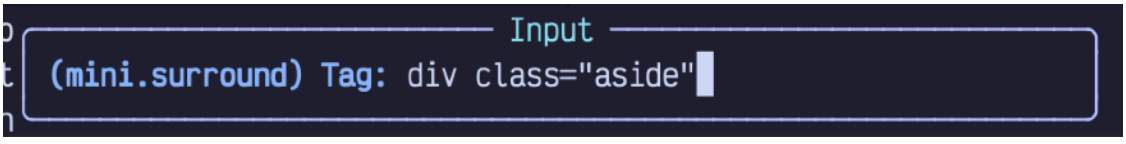
The surround plugin is mostly for working with pairs of characters, but it can also operate on html-like tags.

Say you have a block text and you want to surround it with `p` tags. String together the command `gsaapt` (For additional fun, try pronouncing it). That is `gsa` for "add surrounding" followed by `ap` for "around paragraph. So we're going to add something around a paragraph. Instead of a quote or bracket, we say the thing we are going to add is a `t` for tag.

`mini.surround` will understand that you want to add a tag, and pop up a little prompt window to enter the tag you want to add. Type the `p` for the tag you want to create. You don't need angle brackets; just the tag name:



If the tag you want to add has attributes, you can add them to the menu. `mini.surround` is smart enough to know that the attributes only go on the opening tag.



Modifying the keybindings

I love the surround behaviour. I use it a lot. So much that I quickly got tired of typing `gs` repeatedly. I decided to replace the `gs` with a `;` so I can instead type `;d` or `;r` instead of `gsd` or `gsr`. For adding surrounds, I decided to leverage the fact that double keypresses are easy to type, so I used `;;` for that action instead of `gsa` or even `;a`.

In order for this to work correctly, I also had to modify the `flash.nvim` configuration to remove the `;` command. (By default the `;` key can be used as a "find next" behaviour for the `f` and `t` keys, but the way flash is designed, you don't need a separate key for it; just press `f` or `t` again).

If you want to do the same thing, just create a new lua file named whatever you want (mine is `extend-mini-surround.lua`) inside the `config/plugins` directory.

The contents of the file will be:

```lua
return {
  {
    "echasnovski/mini.surround",
    opts = {
```

```
      mappings = {
        add = ";;",
        delete = ";d",
        find = ";f",
        find_left = ";F",
        highlight = ";h",
        replace = ";r",
        update_n_lines = ";n",
      },
    },
  },

  {
    "folke/flash.nvim",
    opts = {
      modes = {
        char = {
          keys = { "f", "F", "t", "T" },
        },
      },
    },
  },
}
```

Since we are modifying *two* plugins, I put them inside a Lua table, which `lazy.nvim` is smart enough to parse as multiple plugin definitions. The first of these passes `mappings` to the `opts` that are passed into `mini.surround`. These will replace the default keybindings that LazyVim has defined for that table (the ones that start with `gs`).

The second definition also passes a custom `opts` table. It replaces the default keys, which include `;` and `,` with a new table that only defines `f`, `F`, `t`, and `T`.

> **Tip:** If I had known that `;` was being rebound by `flash.nvim`, I could have found this solution by reading the config for `flash.nvim` on the LazyVim website and seeing what needed to be overwritten. However I wasn't able to figure out where the `;` was being defined, and ended up asking for help on the LazyVim Github Discusions. People are really helpful there, and I encourage you to come say hello if you have any questions.

Summary

In this chapter, we learned a bunch of advanced code motion techniques that LazyVim gives us by its reimplementation of Unimpaired mode. Then we learned what TextObjects are and took in a crash course on the many, many text objects LazyVim provides.

We then covered the exceptionally useful `s` motion, which can be used to pick text objects on the fly, as well as the remote variation of `S` and `s`

We wrapped up by going over several new verbs that can be used to work with surrounding pairs of texts such as parenthesis, brackets, and quotes.

In the next chapter, we'll cover clipboard interactions and registers, as well as an entire new Visual mode that can be used for text selection.

Chapter 8: Clipboard, Registers, and Selection

Chapter 8: Clipboard, Registers, and Selection - LazyVim for Ambitious Developers

Vim has a robust copy and paste experience that predates the operating system clipboard you are used to in other editors. Happily, the LazyVim configuration sets up the Neovim clipboard system to work with the OS clipboard automatically.

In fact, you already know how to cut text to the system clipboard: Just delete it.

That's right. Any time you use the `d` or `c` verb, the text that you deleted is automatically cut to clipboard. This is usually very convenient, and occasionally somewhat annoying, so I'll show you a workaround to avoid saving deleted text later in this chapter.

Pasting text

Pasting (typically referred to as "putting" in Vim) text uses the `p` command which I mentioned briefly in Chapter 1. In Normal mode, the single command `p` will place whatever is in the system clipboard at the current cursor position. This is usually the text you most recently deleted, but it can be an URL you copied from the browser or text copied from an e-mail or any other system clipboard object.

The position of the text you inserted can be somewhat surprising, but it usually does what you want. Normally, if you deleted a few words or a string that is not an entire line, it goes

immediately after the current cursor position. However, if you used a command that operates on an entire line, such as `dd` or `cc` to delete an entire line, the text will be placed on the next line. This saves a few keystrokes when you are working with line-level edits, a common task in code editing.

The `p` command can be used with a count, so in the unlikely event you want to paste 5 consecutive copies of whatever is in the clipboard, you can use `5p`.

When you paste with `p`, your cursor will stay where it was, and the text is inserted after the cursor. If you want to instead paste the text *before* the current cursor position, use a capital `P`, where the shifting action is interpreted as "do `p` in the other direction." As with `p`, the text will be inserted directly before the cursor position unless it was a line-level edit such as `dd`, in which case it will be placed on the previous line.

If you are already in Insert mode and need to paste something and keep typing, you can use the `Control-r` command, followed by the + key. The `r` may be hard to remember, but it stands for "**r**egister," and we'll go into more detail about what registers really are, soon.

Copying Text

Copying text requires a new verb: `y`. It behaves similarly to `d` and `c`, except it doesn't modify the buffer; it just copies the text defined by whatever motion or text object comes after the `y`.

"Why `y`?" you might ask? It stands for "yank", which is Vim speak for "copy." I have no idea why `vi` called it "yank," but my guess is that it was a reverse acronym. The original authors probably noticed that `y` was currently free on the keyboard and decided to come up with a word that matches it. The concept of a clipboard or copy/paste had not been standardized yet, so they were free to use whatever terminology worked for them.

The `y` command works with all of the motions and text objects you already know. It is especially useful with the `r` and `R` Remote Seek commands. If you need to copy text from somewhere else in the editor (even a different file) to your current cursor location, `yR<search><label>p` is the quickest way to be on your way without adding unnecessary jumps to your history.

The `yy` and `Y` commands yank an entire line, and from the cursor to the end of the line, analogous to their counterparts when deleting and changing text.

LazyVim will briefly highlight the text you yanked to give a nice indicator as to whether your motion command copied the correct text.

Selecting Text First

Your Vim editing experience so far has not involved the concept of selecting text. Isn't that weird? In normal word processors and VS Code-like text editors, you have to select text before you can perform an operation such as deleting, copying, cutting, or changing it. Considering how awkward text selection is in those editors (you have to use your mouse or some combination of shift, and cursor movements, with extra modifier keys to make bigger movements), it's amazing anyone gets anything done!

In Vim-land, you normally perform the verb first and follow up with a text motion or object to implicitly select the text before manipulating it. This is *usually* the most effective way to operate, but in some situations it is convenient to invert the mental model and highlight text before operating on it.

This is where Visual mode comes in. Visual mode is a Vim major mode, like Normal and Insert mode. Technically, there are three sub-modes of Visual mode. We'll start with "visual character mode" and dig into the other two shortly.

To enter Visual character mode, use the `v` command from normal mode. Then move the cursor using most of the motions that you are used to from normal mode. I say "most" only because Visual mode keymaps are independent of normal mode keymaps, and plugins occasionally neglect to set them up for both modes. LazyVim is really good about keymaps, though, so you will rarely be surprised.

> Tip: You can also get into Visual mode by clicking and dragging with your mouse.

Once you have text selected in visual mode, you can use the same verbs you usually use to delete, change, or yank the selection. You can even use single character verbs like `x` (which does the same thing as `d`) or `r` to replace all the characters with the same character. After you complete the verb, the editor automatically switches back to Normal mode. You can also exit visual mode without performing an action using `Escape` or another `v`.

You can exit Visual mode temporarily without completely losing your selection. From Normal mode, use the `gv` ("**g**o to old **v**isual selection") command to return to the selection. This is useful if you are about to perform a visual operation and realize you need to look something up, make edits, or copy something from elsewhere in the file, then go back to the selection.

Use the `o` (for "**o**ther end") command to move the cursor to the opposite end of the block. Useful if e.g. you've selected a few words, and realize you forgot one at the other end of the block. You can't get into Insert mode from visual mode, so the `o` command gets reused for this purpose.

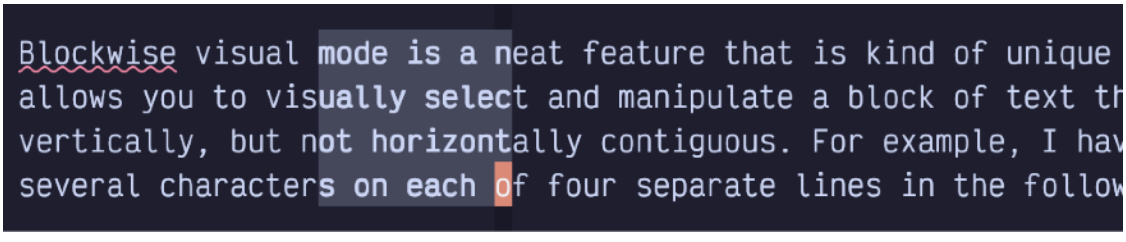
Line-wise Visual Mode

The `v` command is useful for fine-grained selections, but if you know that your selection is going to start and end on line boundaries, you can use a (shifted) `V` instead, to get into line-wise visual mode. Now wherever you move the cursor, the entire line the cursor lands on will be selected.

Other than selecting entire lines, the main difference with Line-wise Visual mode is that when you apply a verb that manipulates the clipboard, (including `d`, `c`, and `y`), the lines will be cut or copied in line-wise mode. When you put them again they will show up on the next or previous line instead of immediately before the cursor.

Block-wise Visual Mode

Blockwise Visual mode is a neat feature that is kind of unique to Vim. It allows you to visually select and manipulate a block of text that is vertically, but not horizontally contiguous. For example, I have selected several characters on each of four separate lines in the following screenshot:



To enter block-wise visual mode, use `CTRL-v` instead of `v` or `V` for visual and line-wise visual mode.

In plain text like this, Visual Block mode doesn't appear to be very useful, but it is handy if you need to cut and paste columns of tabular data in a csv file or markdown table, for example. I don't use it for that functionality terribly often, but when I need it, I know there is no other way to efficiently perform the action I need.

> Tip: If you use `Control-V$`, you will get a slight adaptation of Visual Block Mode where the block extends to the end of each line, in the block. This is handy if you need the block to extend to the longest line as opposed to the line your cursor is currently on.

Visual Block Mode can also be used as a (poor) imitation of multiple cursors. If you use the `I` or `A` command after selecting a visual block, and then enter some text followed by `Escape`, the text you entered will get copied at either the beginning or end of the visual

block. A common operation for this feature is to add `*` characters at the beginning of Markdown ordered lists or a block comment that needs a frame.

Registers

Registers are a way to store a string of text to be accessed later (so think of the Assembly-language definition of the word). In that regard, they are no different from a clipboard. In fact, the system clipboard in Vim **is** a register that LazyVim has set up as the default register.

But Vim has dozens of other registers. This means you can have *custom clipboards* that each contain totally different sequences of text. This feature is pretty useful, for example, when you are refactoring something and need to paste copies of several different pieces of text at multiple call sites.

There are several different types of registers, but I'll introduce the concept with the named registers, first. There are over two dozen named registers, one for each letter of the alphabet.

To access a register from Normal mode, use the `"` character (i.e, `Shift-<Apostrophe>`) followed by the name of the register you want to access. Then issue the verb and motion you want to perform on that register.

So if I want to delete three words and store them in the `a` register *instead* of the system clipboard, I would use the command `"ad3w`. `"a` to select the register, and `d3w` as the normal command to delete three words. And if I later want to put that same text somewhere else, I would use `"ap` instead of just `p`, so the text gets pasted from the `a` register instead of the default register.

`"ad<motion>` will always *replace* the contents of the `a` register with whatever text motion or object you selected. However, you can also build up registers from multiple delete commands using the capitalized name of the register. So `"Ad<motion>` will *append* the text you deleted to the existing `a` register.

I find this useful when I am copying several lines of code from one function to another but there is a conditional or something in the source function that I don't need in the destination. Copy the text before the conditional using `"ay` and append the text after the conditional using `"Ay`, and then paste the whole thing in one operation with `"ap`.

I can copy totally different text into the `b` register using e.g `"byS<label>`. Now I can paste from either the `a` or `b` register at any time using `"ap` and `"bp`.

If you forgot which register you put text in, just press `"` and wait for a menu to pop up showing you the contents of all registers. If that menu is too hard to navigate, you can instead use the `<Space>s"` command to open a picker dialog that allows you to search all registers. Just enter a few characters that you know are in the register you want to paste, use the usual picker commands to navigate the list, and hit `Enter` to paste that text at the last cursor position.

If you're in the `<Space>s"` picker dialog, you'll notice a bunch of other registers besides the named alphabet registers. I'll discuss each of those next.

Clipboard registers

In LazyVim, by default, the registers named `*` and + are always identical to the default (unnamed) register, and represent the contents of the system clipboard.

To understand why, we need some history: Vi had registers, and then operating systems got excited about the ideas of clipboards and vi users wanted to copy stuff to the system clipboard. A default (non-lazy) Vim configuration means that if you want to copy text to the system clipboard, you have to always type `"+` before the `y`. The three extra keystrokes (`Shift`, `'`, and +) can get pretty monotonous in modern workflows where you're copying stuff into browsers, AI chat clients, and e-mails on a regular basis.

On top of that, some operating systems (Unix-based, usually) actually have *two* Operating System clipboards, an implicit one for text you select, and one for text you explicitly copy with `Control-c` (in most programs). This text would be stored in the `"*` register and the OS lets you paste it elsewhere with (typically) a middle click.

I recommend sticking with LazyVim's synced clipboard configuration, but if you already have muscle memory from using Vim the old way or you're just tired of deleted text arbitrarily overwriting your system clipboard, you can disable this integration so that the three registers behave as described above instead of being linked together. To do so use `space f c` to open the `options.lua` configuration file and add the following line:

```
vim.opt.clipboard = ""
```

Speaking of having your clipboard contents randomly overwritten, if you know in advance that you don't want a specific delete or change operation to overwrite the clipboard contents, use the "Black Hole" register, `"_`. So type `"_d<motion>` to delete text without storing it in any registers including the system clipboard.

If you want to *copy* the contents of one register to another register, you can use the ex command `:let @a = @b` where `a` and `b` are the names of the registers you want to copy to and from. The most common use of this operation is to copy the contents of the system

clipboard (which may have come from a different program) into a named register so it doesn't get lost the next time you issue a verb. For example, `:let @b = @+` will copy the system clipboard into register `b`.

The last yanked or last inserted text

Whenever you issue a `y` command without specifying a destination register, the text will always be stored in the `"0` register *as well as* the default register. And it will stay in `"0` until the next yank operation, no matter how many deletes or changes you do to change the default register.

So if you yank the text `abc` and then delete the text `def`, the `p` command will paste the text `def`, but you still can paste `abc` using `"0p`.

You can also use the `".` (period) register to paste the text that was most recently inserted. So if you type the command `ifoo<Escape>` somewhere in the document and move somewhere else in the document and type `".p`, it will insert the word "foo" at the new cursor position. `".` is a register that you may occasionally want to copy into a named register if you have inserted text you want to reuse. Use the previously discussed `:let @c = @.` command to do this.

The delete (numbered) registers

The numbered registers *should* be really useful, but I find them rather confusing. The registers `"1` through `"9` always contain the text that you most recently changed or deleted, in ascending order. So after a delete operation, whatever was in `"1` gets moved to `"2`, `"2` moves to `"3` and so on, and whatever is in `"9` gets dropped.

I can *never* remember the order of my recent deletes, so I would normally have to use the `"` menu to see the contents of the numbered registers. It's handy that my recently deleted text is stored and I can find it this way. However, I usually use the `yanky.nvim` plugin (discussed later in this chapter) instead, so the numbered registers are not that useful to me.

There is also a "small delete register" that can be accessed with `"-`. Whenever you delete any text, it will be stored in the numbered registers, but if that text is less than one line long, it will *also* be stored in this minus register. I have little use for this feature, as the majority of my changes are smaller than one line. That means it gets cleared before it drops out of the numbered registers.

The current file's name

The file that you are currently editing is stored in the `"%` register. It is always relative to the current working directory of the editor (usually the folder you were in when you started Neovim). The only time I ever want to access this register is to copy the filename to the system clipboard with `:let @+ = @%` so I can paste the filename into the browser or my terminal.

Recording to registers

Remember the recording commands I told you about in Chapter 6: `qq` to record and `Q` to play back the recording? Turns out I was a little overly simplistic there.

Recorded commands are actually stored in a named register. In this case, I arbitrarily chose the `q` register when I said to use `qq` to start recording. But you can just as easily store it in the `a` register using `qa` or the `f` register using `qf`.

The `qQ` command to "append to recording" operation is analogous to the capitalized `"A<command>` used to append to a register. In this case, `Q` is still an arbitrary name, and you can append a recording to a different named register besides `q`, use `qA` or `qZ`, for example.

Having multiple sets of recordings can be really handy when you are performing a complex refactoring that requires you to make one of several different repetitive changes in different locations across your codebase.

The `Q` command to play back a recording always plays back the most recently recorded command, regardless of register. If you want to play back from a different register, you would use the `@` command, followed by the name of the register. So if you recorded using `qa`, you would play it back with `@a`. As a simple shortcut, `@@` will always replay whichever register you most recently *played* (which is different from `Q` which always plays back the most recent *recording*).

Editing recordings

To be clear, recordings are placed in normal registers. So if you record a sequence of keystrokes to a register using `qa` and then put the register using `"ap`, you will actually see the list of Vim commands you recorded.

This can be useful if you mess up while recording and need to modify the keystrokes. After recording the keystroke, paste it to a new line using e.g. `"ajp`. At this point it's just a normal line of text that happens to contain vim commands. You can modify it to add other Vim commands, since they are all just normal keystrokes.

For example, let's say I recorded a command as `qadw2wdeq`, which records to the `a` register (`qa`), deletes a word (`dw`), skips ahead two words (`2w`), and then deletes the next word (`de`), then ends the recording with `q`. But too late, I realize I should have skipped over 3 words, not two words.

I can use `"ap` to paste the contents of the recording, which will look like this: `dw2wde`. Then I can use `f2` to jump to the `2` digit, followed by `r3` to replace it with a `3`. Now I can use `"ayiw` to replace the contents of the register with `dw3wde`.

Now if I want to play back that modified command, I can just use `@a` as usual.

The yanky.nvim plugin

Yanky has some niceties such as improving the highlighting of text on yank and preserving your cursor position so that you can keep typing after pasting, but its primary feature is better management of your clipboard history. LazyVim also configures it with several new keybindings to make putting text more pleasant.

The plugin is not enabled by default, but it is a recommended extra, so if you followed my suggestion of installing all recommended extras back in Chapter 5, you may have it enabled already. If not, head to `:LazyExtras`, find `yanky.nvim` and hit `x`. Then restart Neovim.

Now that Yanky is enabled, the easiest interface to see your clipboard history can be accessed with `<Space>p`. It pops up a picker menu of all your recent clipboard entries. Up to a hundred entries are stored, which is a lot more than you get in the numbered registers, and it stores your yanks, not just your deletes and changes. If you need to paste something that is no longer in the clipboard, `<Space>p` is probably the quickest way to find it.

Another super useful keybinding is `[y`. If you invoke this command *immediately* after a put operation, the text that was put will be replaced with the text that was cut or copied prior to the most recent yank. And if you press it again, it will go back one more step in history, up to 100 steps. So if you aren't sure exactly which numbered register a delete operation is in, or you want to access text that was yanked but is no longer in the `"0` register, you can use `p[y[y[y...` until you find the text you really wanted to pasted. If you go too far, you can cycle forward with `]y`.

LazyVim also creates some useful keybindings to improve how text is put, especially with respect to indentation. The two most useful useful are `[p`, and `]p`, (The capitalized versions `[P`, and `]P` are just duplicates that you can safely ignore).

These commands will paste the text in the clipboard on the line above or below the current line, depending on whether you used `[` or `]`. You may think this would be identical to the automatic line-wise pasting described above, but it's slightly different for two reasons:

- First, it pastes on a new line regardless of what command was used to cut or copy the text that is in the clipboard.
- Second, it automatically adjusts the indentation of the text on the new line to match the indentation of the current line.

So if you're moving code into a nested block and need to change indentation, use `]p` instead of relying on line-wise paste. Then you don't have to format it afterwards, (Not that formatting is hard in LazyVim; it happens on save).

You can also use `>p` `<p`, `>P`, and `<P` to automatically add or remove indentation when you put code.

Summary

This chapter was all about selecting and copying text. We learned the `yank` verb for copying text and then dug into the various Visual modes that can be used for selecting text.

Then we learned that Vim has multiple clipboards called registers, and how to cut and copy to or paste from those registers. We even went into more detail about using registers to record multiple separate command sequences before discussing the yanky.nvim plugin to make your pasting life a little easier.

In the next chapter, we'll learn about various ways to navigate between symbols in related source code files as well as how to show and hide code with folding.

Chapter 9: Buffers and Layouts

Chapter 9: Buffers and Layouts - LazyVim for Ambitious Developers

No matter what programming language you are working with, it is inevitable that you will be working on multiple files at a time. And in multiple areas within the same file.

Like all coding editors (other than Notepad), Neovim has a robust system for working with multiple files. LazyVim is configured with a powerful buffer, file, and window management system that may feel familiar at first, but is actually far more powerful than your average editor.

Some terminology

Sometimes it seems like every window management system uses the same words for different things. If you read the documents for e.g. tmux, emacs, kitty, vim, and i3, you'll end up with multiple definitions for words like "window", "pane", "tab", and "layout".

I'll stick with the Vim definitions of these words so that you can switch between this book and most Vim and vim plugin help files, tutorials, and documentation, without getting confused. Unfortunately, this may mean you get confused when interacting with any other software!

This list goes roughly from least to most specific, though understand that the relationship between most of these elements is a graph rather than a tree; it's not a strict hierarchy.

- **Server**: Neovim can run in a server mode and can have multiple clients attached. This means you can have multiple views into the same Neovim instance, and those views could be from different terminals or GUI software, web browsers or even a VS Code Extension. You will probably not need think about the Neovim server, and I won't mention it again in this book. But if you want to do something interesting such as connect to an existing Neovim instance to open a commit message rather than opening Neovim in a new window, you now know that this is possible.
- **Client**: A Neovim application that you are actually running. Normally connected to its own independent server but can be configured to connect to an existing or remote one. A client is what starts up when you type `nvim`, but other clients include GUIs such as Neovide or VimR.
- **Tab**: One client can have multiple tabs. Each tab is a full screen layout that is more or less independent from the other tabs. You can have different buffers visible and different configurations of window splits on each tab. Only one tab is visible at any one time. This is a much different paradigm from VS Code and many other environments where each split has its own set of tabs.
- **Window**: Also known as a "pane" or a "split", a window is a section of the screen that is dedicated to viewing a buffer. Every tab has one or more windows on it. Every window is normally entirely visible; there is no overlap between windows (except in the case of floating windows such as the ones that pop up when you open a picker or Lazy Extras). If a buffer's contents don't fit in a window the window can be scrolled.
- **Buffer**: This is Vim's word for a file that is currently open and available to be viewed/edited. One buffer can be displayed in multiple windows, which means you can have two side-by-side views into the same file at different scroll positions or you can view the same buffer in multiple tabs. If a buffer is visible in two places, they will have the exact same contents (other than scrolling position). There is only ever **one**

buffer open for each file, no matter how many views of the buffer are visible in different windows or tabs.

- **Fold**: Within any one view of a buffer, it is possible to "collapse" a section of that file (for example a function, class, or indentation level) into a single line, effectively hiding the contents. This allows you to view two disjointed sections of the same file at the same time while keeping the unrelated information between those two sections hidden.
- **File**: A file that exists on disk. Each buffer is linked to at most one file, though it is possible to have buffers with no file (sometimes called "scratch" buffers, a word borrowed from Emacs parlance). The contents of a buffer may not be the same as the contents of the file on disk if the buffer has not been saved.

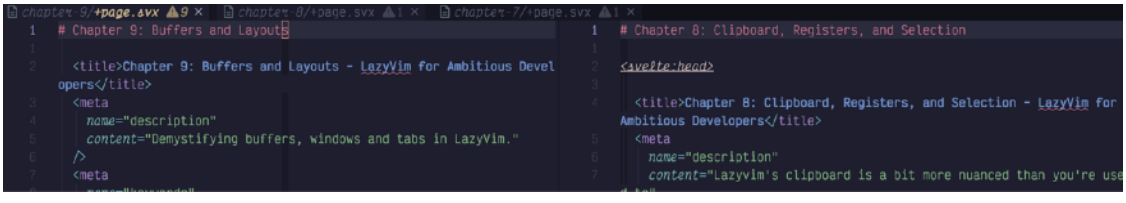
So far in this book, all your interactions have been with one or more buffers in a single window in a single tab. Now, things are about to get much more ~~complicated~~ interesting.

Buffers

We'll start with buffers. If you've used Telescope, Neo-tree, or mini.files to open multiple files, you may well think that a buffer is a tab. In this view, I have three buffers open, only one of which is currently visible:

This is a buffer line, not a tab bar. I repeat: **Those are not called tabs**. Yes, I know they look like tabs in any other software, ever, but that is because LazyVim has configured the buffer line to look like tabs. With the buffer line visible, you may actually not need to use (real) Vim tabs very often, but in Vim, tabs are a completely different concept.

No matter how many **windows** you have open, there is only one buffer line. In the following screenshot, I have the same three buffers open, and two of them are visible in in separate windows, side by side. But there is still only one buffer line along the top of the editor.



This implies that buffers are a "global" concept. There is one collection of buffers for the entire Neovim client, and you can access any of those buffer from any window (or tab).

You can, of course, use the mouse to select different buffers from the buffer line with a single click. But why would you do that when there are **so many** ways to access buffers with the keyboard in LazyVim?

Navigating between open buffers

The absolute easiest way to switch between buffers is using the `H` and `L` (i.e. `Shift-h` and `Shift-l`) keys. By this point you are hopefully intimately familiar with the fact that `h` means left and `l` means right for cursor movement. If you just press the shift keys, you will switch the buffer visible in the currently active window to whichever buffer is to the left or right of the current buffer in the buffer line.

I, of course, don't use these because of being that left-handed, dvorak layout, split keyboard using mutant I mentioned in Chapter 3, but you probably should. I instead use the `[b` and `]b` commands which map to the same thing.

Annoyingly, you will find that these keybindings do not accept counts. So you cannot, by default, use `2L` to jump two tabs to the right. This frustrated me because I know the underlying `:bnext` and `:bprev` commands *do* accept a count.

It turns out that LazyVim maps these to a `BufferLineCycleNext` command provided by the underlying plugin, `bufferline.nvim`, and that plugin doesn't, as far as I can tell support counts.

Upon investigation it sounds like the `BufferLineCycle*` commands exist because the plugin can configure some kind of sorting mechanism on the buffer list. But LazyVim isn't configured to use that mechanism. So we can use the old-fashioned commands instead. To do so, create a new file in your `plugins` configuration folder named (something like) `extend-bufferline.lua`:

```
return {
  "akinsho/bufferline.nvim",
  keys = {
    {
      "L",
      function()
        vim.cmd("bnext " .. vim.v.count1)
      end,
      desc = "Next buffer",
    },
    {
      "H",
```

```
      function()
        vim.cmd("bprev " .. vim.v.count1)
      end,
      desc = "Previous buffer",
    },
    {
      "]b",
      function()
        vim.cmd("bnext " .. vim.v.count1)
      end,
      desc = "Next buffer",
    },
    {
      "[b",
      function()
        vim.cmd("bprev " .. vim.v.count1)
      end,
      desc = "Previous buffer",
    },
  },
}
```

The `vim.v.count1` variable is set whenever a keybinding is called with a count, so it can be accessed inside the callback and passed to the Vim command using string concatenation (the `..` operator). Restart neovim and you can do things like `3L` to jump three buffers to the right on the buffer line.

Another keybinding you will want to reach for when jumping between buffers is `<Space><Backtick>`. This one simply jumps between the current file and the file that was most recently opened in the current window. In Vim parlance, this is referred to as "the alternate file."

If you have a large number of buffers open, the buffer line can get awfully crowded. At some point, it will show two arrows to the left and/or right of the buffer bar so you can tell that there are "hidden" buffers. When you navigate through buffers, it will always ensure the active buffer is visible. Here's a very full buffer line with four buffers hidden to the left and two hidden to the right:

If you have this many buffers open, you may find it easier to use Telescope or fzf.lua (depending which you have enabled) to search through the open buffers. The keybinding to

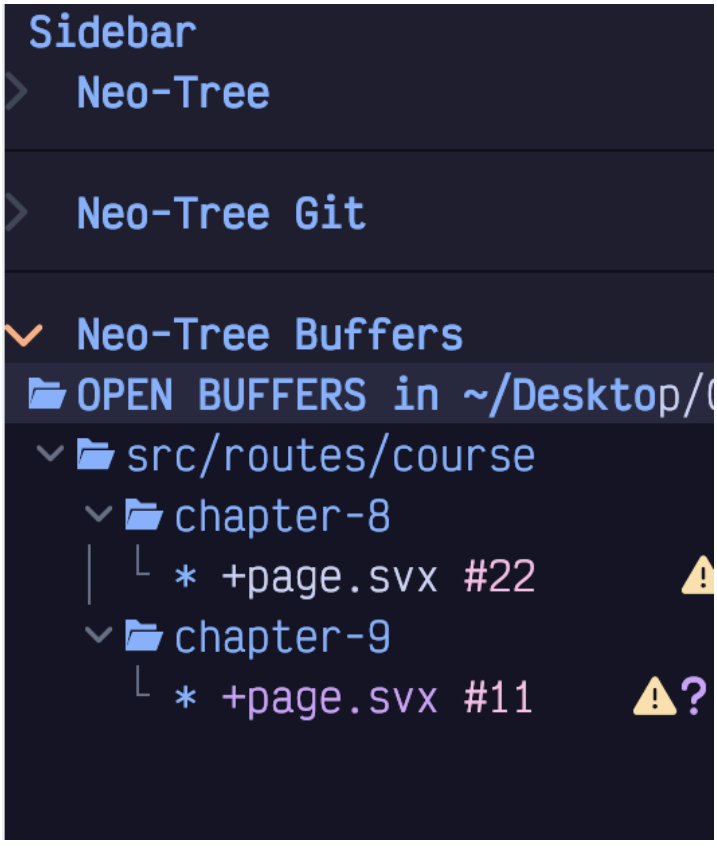
pop up a filterable, scrollable list of buffers is `<Space><comma>`. It has the exact same contents as the buffer line, but it's a different interaction effect.

This is useful if you are working on large projects that have so many files that searching through them with `<Space><Space>` is difficult. If you open the relatively low number of files that you actually need to access as active buffers, they will be easier to filter in the list of open buffers from `<Space><comma>`.

Alternatively, you can use NeoTree to navigate open buffers. If you show the NeoTree sidebar, you'll see that it has some "accordian"-style widgets, named `Neo-Tree`, `Neo-Tree Git` and `Neo-Tree Buffers`. NeoTree has a variety of sources for navigating tree-like interfaces, though these three are the only ones preconfigured by LazyVim.

To switch between the accordions (NeoTree calls them "sources"), you can click a different heading (e.g. Neo-Tree Buffers) with the mouse, or use the < and > keys to cycle through them. Alternatively, use the `<Space>be` key sequence to show the "buffer explorer".

Once the Neo-Tree Buffers view is visible, it will look something like this:



Note that if you have buffers open that are not rooted in the current working directory, they won't show up in Neotree. You'll have to use Telescope for those.

> **Tip:** If you tend to have the Neotree sidebar open all the time, you might want to consider disabling the bufferline across the top of your screen. It shows the same information and there's no reason to spend screen real estate (that most precious commodity) on having both of them visible all the time. Just add `{ "akinsho/bufferline.nvim", enabled = false }` to your `disabled.lua`. Be aware that this will disable certain buffer management keybindings, however.

Closing Buffers

You will commonly want to close the current buffer without closing the split(s) it is currently open in. The keybinding for this is `<Space>bd`, where `<Space>b` pops up a useful menu of other buffer related functions, and `d` means "delete". You aren't actually deleting the underlying *file* when you do this; you're just deleting the buffer from Vim's memory: i.e. closing it.

You can also just press `bd` if you currently have Neotree's buffer view focused.

I find that closing buffers is too common a task to deserve three keys, so I have added the following to the `keys` array I defined in `extend-bufferline.lua`:

```
{
  "<leader><delete>",
  LazyVim.ui.bufremove,
  desc = "Close current buffer"
},
```

Here are several other commands you can use to close buffers:

Keybinding	Description	Mnemonic
`<Space>bD`	Close buffer and the window split it is in.	**D**elete buffer, but "bigger"
`<Space>bl`	Close all buffers to the right in the tab line	
`<Space>bh`	Close all buffers to the left in the tab line	

Keybinding	Description	Mnemonic
`<Space>bo`	Close all buffers other than the active one	"**o**nly" this buffer
`<Space>bP`	Delete all non-pinned buffers	"P" is opposite of "p"

The last one needs some clarification. You can toggle a "pin" on any active buffer using `<Space>bp`. You'll see a pin icon show up to the left of the buffer name. The only purpose of this pin is to keep it open if you want to close all the "less important" (unpinned) files using `<Space>bP`. I personally don't use buffer pinning very much, so for me, `<Space>bP` is a shortcut for "close all buffers"; useful when I complete one task and am ready to start another.

One last tip for deleting buffers. If you spend a lot of time in the Telescope buffer picker (from `<Space>,`), you can set up a keybinding for that picker that allows you to delete files from it without first activating the buffer in a window. I have this mapped to `<Alt-D>`. To do so, create an `extend-telescope.lua` file that contains the following:

```lua
return {
  "nvim-telescope/telescope.nvim",
  opts = {
    pickers = {
      buffers = {
        mappings = {
          i = {
            ["<A-d>"] = function(...)
              return require("telescope.actions").delete_buffer(...)
            end,
          },
          n = {
            ["<A-d>"] = function(...)
              return require("telescope.actions").delete_buffer(...)
            end,
          },
        },
      },
    },
```

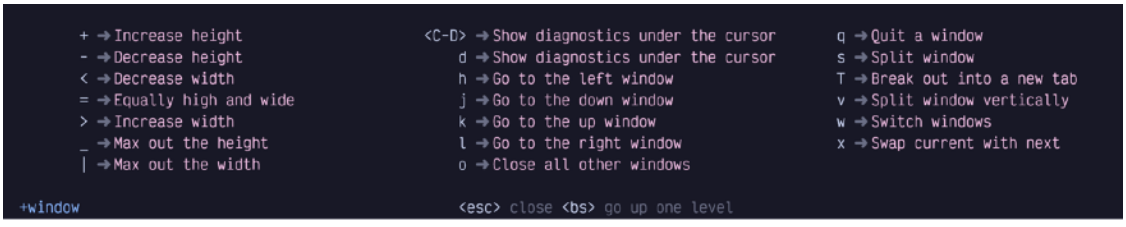
```
  },
}
```

Now that you know all about buffers, let's discuss windows.

Windows

In most modern environments, "windows" refer to the OS-level windows such as the terminal you are running Neovim in. Since Vi predates such environments, they were able to use the word window to refer to what are nowadays more commonly described as "panes" or "splits" in other environments, and the name has stuck.

All the default Vim window keymaps are available in a submode that is accessible via `Control-w`. If you hit that key-combination, LazyVim will pop up a menu with the windowing commands you can use:



We'll go into detail of some of those in a bit, but first, I need to acknowledge that moving between windows happens a lot, and that Control key is not always the most comfortable thing to access. So I recommend adding the following keymap to your global config (press `<Space>fc` to open a picker into the config directory, and then filter for the `keymaps.lua` file):

```lua
vim.keymap.set("n", "\\\\", "<C-w>", {
  desc = "Show Window menu",
  remap = true
})
```

This maps the backslash key to `Control-w` so that the single keypress `\` shows the same menu that two-fingered `Control-w` does. So for example, that means that instead of pressing `<Control-w>v` to create a vertical split, I can use `\v` instead. I will assume you have added this keymap in the rest of this discussion.

Creating Window Splits

Windows in LazyVim can be created on the fly at any time. To split the current window in half "vertically" with one window on the left and a new window on the right, use the `\v` key.

When you create a split, the new window will contain another view of the buffer you were already viewing, side by side or one above the other. But once the split is opened, you can switch the buffer in that split using any of the buffer management commands or by opening a new file with any of the tools we've previously discussed for opening files.

To create a horizontal split between two windows one above the other, use `\s`. The Mnemonic for this is unfortunately just "split". They weren't able to reuse `\h` because that is saved for navigation.

Note: LazyVim also allows you to create a vertical split with `<Space>-|` where the `|` is the vertial bar when you `Shift-Backslash`, and `<Space>-<Minus>` for a horizontal split. I never use these because `\v` is one fewer keys to press and the `\s` is requires less hand movement than `<Space>-<Minus>` on my keyboard.

Creating Splits When Opening files

You already know you can open a file in the current window from Neo-Tree by moving your cursor to the file and pressing `<Enter>`. You can also use the `s` key in Neo-Tree to open it in a *vertical* split (unlike the `\s` key to create a horizontal split in a normal buffer). The shifted form, `S` in Neo-Tree is used to create a horizontal split. To be honest, this is one place where I think Neo-Tree's unconventional keybindings are superior to the built-in.

If you are using Telescope or fzf.lua to open files, you'll use yet another set of keybindings! To open a file in a vertical split from telescope, use the `Control-v` keybinding (this works in both Insert and Normal mode in the Telescope prompt area, but only in Insert mode with fzf.lua). To open it a horizontal split you use `Control-x` (I know: WTF, right?)

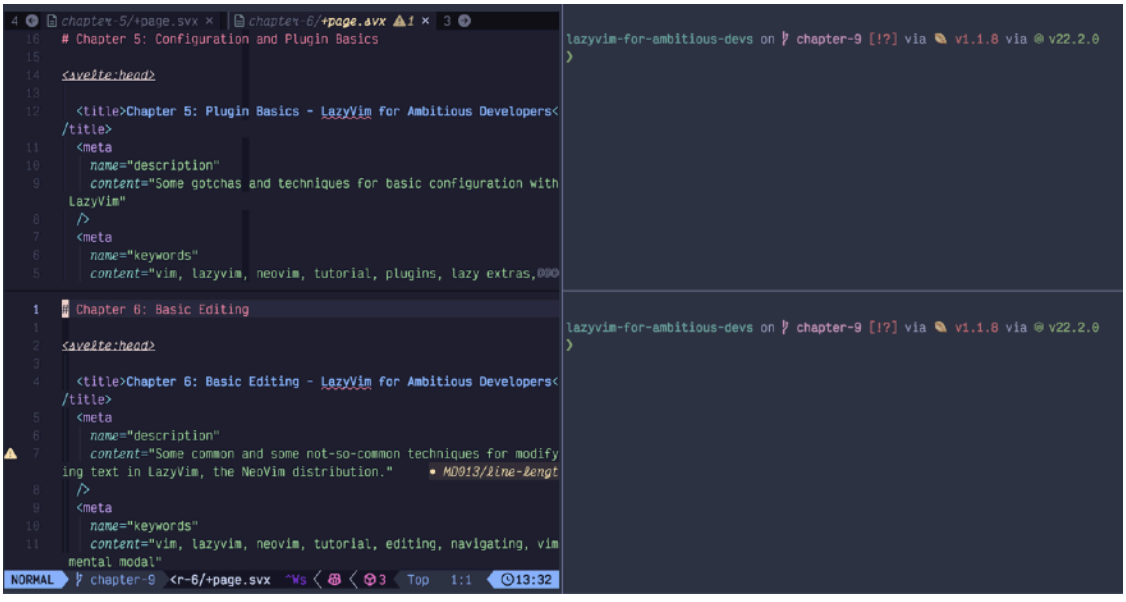
Finally, If you use `mini.files`, you can open a file in a split using the same keybindings as in a normal window (`<Control-w>v` or `\v` and `<Control-w>s` or `\s`), thanks to a PR I submitted so I could shorten this chapter!

Navigating between windows

You can move your cursor between window splits by holding the control key along with any of the `h`, `j`, `k`, or `l` home row arrow-key directions. It can also be prefixed with numerical counts if you want to skip over a window to get to the next one.

Alternatively, you can use the same keys with `\`. So `\h` will move to the window to the left of the current one.

This is also a good time to mention the `mrjones2014/smart-splits.nvim` plugin, which can be configured to navigate between Vim windows and Kitty, Wezterm, or Tmux panes with the same keybindings. Consider this screenshot:



I have three Kitty Terminal panes open. The left one is running Neovim with two windows in it, one above the other. The right is split into two normal terminal panes. By default, if I want to navigate between the three Kitty panes, I have to use one set of keybindings, and if I want to navigate between the two Neovim windows, I have to use another set of keybindings. With the [smart-splits.nvim] plugin, I can navigate between all the windows with the same keybindings, no matter where my cursor is.

Setting up the terminal integration for smart splits is beyond the scope of this book (documentation on the README in the github repository should be sufficient), but to configure the smart-splits plugin in Neovim, create a file in the plugins directory called e.g. `smart-splits.lua`:

```
return {
  "mrjones2014/smart-splits.nvim",
  build = "./kitty/install-kittens.bash",
  keys = {
    {
      "<A-h>",
      function()
        require("smart-splits").move_cursor_left()
      end,
      desc = "Move to left window",
    },
    {
```

```
      "<A-l>",
      function()
        require("smart-splits").move_cursor_right()
      end,
      desc = "Move to right window",
    },
    {
      "<A-j>",
      function()
        require("smart-splits").move_cursor_down()
      end,
      desc = "Move to below window",
    },
    {
      "<A-k>",
      function()
        require("smart-splits").move_cursor_up()
      end,
      desc = "Move to above window",
    },
  },
}
```

If you are using WezTerm or Tmux you won't need the `build =` line, but for all three environments, you'll also need to add some configuration to your Kitty, WezTerm, or Tmux configuration.

Closing a Window Split

You can close a window at any time using `\q` where the mnemonic is "**q**uit", although you aren't actually quitting anything. You're just closing the split.

If, instead, you want to close all splits except the active one, use `\o` for "**o**nly this window" or "close **o**ther".

Resizing Windows

In my unconventional opinion, the easiest way to resize Vim splits is to use… the mouse. There is a vertical bar between vertical splits that you can click and drag on. The mouse cursor doesn't change to give you any feedback that you can click and drag on it, but it works.

For horizontal splits (when two windows are one above the other), there is no obvious bar to click. But you can actually just click on the status bar of the "upper" window to move it up or down.

If you insist on using the keyboard, the keybindings are in the \ menu: \+ and \– to increase or decrease the height of the active window in a horizontal split, and \> or \< to increase or decrease the width of a vertical split. They only move by one row or column at a time, so you will almost certainly want to prefix these commands with a count greater than 10.

To change everything to a "default" size, use \= which will make all the windows "equally high and wide." I use this keybinding frequently whenever I open a "temporary" window such as Neo-tree or the Copilot Chat window to ensure the other windows share the remaining space equally.

Tabs

Tabs in Vim are rather unusual. Some other paradigms might describe them as "Layouts". All tabs are connected to the same list of currently open buffers, which is different from most tab models, but each tab can be split into different window layouts. So you might have one tab with three vertical splits and a second tab with four windows open in a grid, for example. In each of the seven splits you can have any buffer you like open, possibly in multiple locations.

LazyVim has a dedicated tab menu that is accessed by pressing <Space>-<Tab>:

To create a new tab, just press <Space>-<Tab>-<Tab>. If you want to "move" the currently open window split into a new tab, use \T (that's a capital T). This will effectively close the current window split and create a new tab with the same buffer in that tab.

After you create a tab, you'll likely have trouble finding it again! The tabs are grouped at the right end of the buffer line:

This screenshot has **two** tabs, numbered 1 and 2 at the right with an X beside them. The three buffers at the left *are not tabs*. Have I emphasized that too many times?

Unfortunately, other than numbers, the tabs don't do anything to make themselves look unique; it is not possible to know which buffers are active in each tab or what layout they have.

To navigate between tabs, you can click the numbers, or you can use the default vim keybindings of `gt` and `gT` to go to the next or previous tab. Alternatively, the `<Space><Tab>[` and `<Space><Tab>]` keybindings provided by LazyVim can also switch tabs. To go to a specific tab by number, use that number as the count when calling `gt`. For example `3gt` will show the tab number 3 rather than jumping three tabs to the right.

There are several ways to close a tab:

- Just close the last window in the tab (i.e. with `\q`) and the tab will disappear.
- The `<Space><tab>d` keybinding will close all the windows in a tab and then the tab. The buffers will stay open.
- Click the X icon to the right of the tabs in the tab bar.

Code Folding

Vim's code folding system is almost too robust, probably because it has had many iterations of "best practices" over the years. LazyVim is configured with the current best practices, so you generally only need a small subset of the complete list of folding commands.

If you are unfamiliar with the concept, code folding allows you to hide entire sections of code by collapsing them into a single line. Visually, this has a similar effect to splitting a window horizontally and then reading two sections of the same file above and below the split, but when you use folding only one view of the buffer is visible and it scrolls as a single entity.

Consider this section of code:

```
<script lang="ts">
  $: syncSourcesOfTruth($selectedSceneText)

  function clearExistingTimeout(outlineitemid: string) {
    if (saveTimeouts[outlineitemid] == undefined) {
      clearTimeout(saveTimeouts[outlineitemid])
      delete saveTimeouts[outlineitemid]
    }
  }

  // called both on blur and on a timeout when the user stops typing
  function save(sceneTextToSave: SceneText) {
    if (!$draft) return
    clearExistingTimeout(sceneTextToSave.outlineitemid)
    updateSceneTexts(sceneTextToSave.outlineitemid, $draft.id, sceneTextToSave.text)
  }

  function debouncedSave(sceneTextToSave: SceneText) {
    clearExistingTimeout(sceneTextToSave.outlineitemid)
    saveTimeouts[sceneTextToSave.outlineitemid] = setTimeout(() => save(sceneTextToSave), 500)
  }

  async function addTodo() {
    if (!$book || !$selectedOutlineItem) return
    const todoid = await addTodoMutation($selectedOutlineItem.id, {
      bookid: $book.id,
      text: '',
```

While I'm editing, imagine that I am interested in the `clearExistingTimeout` function at the top of the screenshot and the `addTodo` function at the bottom, but not currently interested in the contents of the two `save` callbacks. I can collapse those sections and my screen looks like this:

```
<script lang="ts">
  $: syncSourcesOfTruth($selectedSceneText)

  function clearExistingTimeout(outlineitemid: string) {
    if (saveTimeouts[outlineitemid] == undefined) {
      clearTimeout(saveTimeouts[outlineitemid])
      delete saveTimeouts[outlineitemid]
    }
  }

  // called both on blur and on a timeout when the user stops typing
> function save(sceneTextToSave: SceneText) {

> function debouncedSave(sceneTextToSave: SceneText) {

  async function addTodo() {
    if (!$book || !$selectedOutlineItem) return
    const todoid = await addTodoMutation($selectedOutlineItem.id, {
      bookid: $book.id,
      text: '',
      completed: false,
    })
    selectedTodoId.set(todoid)
    return todoid
  }

  // Split into another scene at the current cursor
```

Most fold operations are accessible from the `z` mode menu accessible by typing `z` in Normal mode (We discussed some of the `z` mode operations in an earlier chapter when we were dealing with scrolling). To collapse a section of code into a fold, use whatever navigation operations you like to get to that section and type `zc` for "**c**ollapse fold".

To open it again, use `zo` for "**o**pen fold".

Alternatively, if you only want to remember a single keybinding, `za` will toggle a fold, collapsing if you are not on a folded line and expanding if you are.

If you have collapsed some folds and want to quickly get back to a point where there are no folds collapsed, use `zR` to open all folds. I had no idea what mnemonic is supposed to work with `R`, but an early reader helpfully pointed out that `zr` is "reduce folding", so `zR` is "Reduce folding BUT BIGGER".

You can even nest folds by folding already folded code. If you want to open folds recursively, use `zO`, which will open a fold and any folds that are nested under that fold.

The way LazyVim is configured, you don't have much control over what gets folded, but it will usually do something close to what you expect based on where your cursor is in the document. "What you expect" depends on both the LSP and the TreeSitter grammar for the language you are working on, but I find it best to just let it do its thing and not disagree with it.

If you find that you want way more control over code folding, I recommend reading `:help folding` in its entirety. More than likely, you'll decide that actually, you don't want more control over folding and just want LazyVim to handle it for you!

Sessions

After a long hard day of coding, you probably have several buffers open and your splits and tabs configured with all the files in just the right places. Wouldn't it be nice to put the code away for the evening and come back to all those buffers, tabs, and splits just as they were?

LazyVim has built-in session management enabled by default. Simply close LazyVim with `<Space>qq` and be on your way. Tomorrow morning, open it to the dashboard with the `nvim` command and hit `s` to be on your way.

If you forgot to open it right away and the dashboard is long gone, you can use `<Space>qs` to restore Neovim to wherever it was when you last closed it (though any files you modified and saved in the meantime will still have their new contents).

If you have opened Neovim temporarily and want to close it without wiping out the session that was saved the last time you closed it, hit `<Space>qd` at any time after you open Neovim and before you close it. In some contexts (notably, git commit messages), this happens automatically so you don't have to worry about making a commit after you close the editor and then losing your session.

Summary

In this chapter, we learned about Vim's buffers, windows and tabs, and how they are different not only from each other, but also from many other window management paradigms. Vim has the same concepts as other editors, but they are sometimes mixed or named in different ways.

We also covered code folding to make it easier to wrangle large files, and session management to return to your window configuration and come back to it later. This is particularly useful when combined with LazyVim's lightning fast startup time. There's no reason to keep your code editor open consuming memory when you aren't actually, you know, editing code.

In the next chapter, we'll dig into some of the terrific programming language support that LazyVim provides. This is arguably the one thing that made VS Code amazing, but the Vim community has learned from its competitors and eventually outmatched them.

Chapter 10: Programming Language Support

Chapter 10: Programming Language Support - LazyVim for Ambitious Developers

Visual Studio Code brought the world the concept of language servers, and all of the other editors jumped on the idea. Early incarnations of language server protocol in Vim were frustrating and clunky and required plugins that tended to be fragile and complicated.

Then NeoVim decided to build support for language servers into the editor itself. NeoVim's built-in support is still frustrating and clunky, but over time, robust and simple plugins have evolved to make the language server experience almost automatic. LazyVim represents the pinnacle of that evolution.

In addition, NeoVim also has built-in support for TreeSitter, a powerful library for parsing and identifying abstract syntax trees in source code while it is being edited, and LazyVim is configured with the plugins needed to make TreeSitter Just Work™.

Language Server protocol gives us support for things like code navigation, signature help, auto-completion, certain highlighting and formatting behaviours, diagnostics, and more. TreeSitter gives us better syntax highlighting, code folding, and syntax based navigation such as provided by the `s` command you already know.

There are two main tools for working with language servers in LazyVim: various language Lazy Extras, and the Mason.nvim plugin. We'll get to know both of these and then learn how to better use some of the tooling they provide.

The lang.* Lazy Extras

We've already used LazyVim extras for plugin configuration, and I told you to install the extras for whichever languages you use regularly. These extras include preconfigured plugins that give best-in-class support for common programming languages. Most of them ship with preconfigured language servers and many include additional NeoVim plugins that are useful with those languages.

In most cases, once you install these extras things will work out of the box, and you won't have to learn any new keybindings for the commands each language provides. However, it wouldn't hurt to read the Readmes for the plugins the extra installs (accessible by looking up the Extra's documentation on the LazyVim website and clicking the headings) to make sure you aren't missing out on any commands the language provides. For example, the python extra ships with the `venv-selector.nvim` plugin that allows you to activate many types of Python virtual environments either automatically or on demand. LazyVim installs a keybinding to open the virtualenv selector using `<Space>cs` where `<Space>c` is the "Code" submode.

Mason.nvim

The Lazy Extras may not install everything you need. For example, instead of the default Typescript formatting and linting tools, I prefer to use a new hyper-fast up-and-comer called Biome.

To install things like this, you can use the `Mason.nvim` plugin, which is pre-installed with LazyVim. To open Mason, use the `<Space>cm` keybinding. The window that pops up looks similar to the `lazy.nvim` and Lazy Extras floating window, although it ships with annoyingly unrelated keybindings.

Mason is effectively a very large database of programming language support tooling, including language servers, formatters, and linters, along with the instructions to install them.

Mason.nvim *does* assume a certain baseline is already installed on your system; for example if you are going to install something that is Rust-based, you better have a `cargo` binary, and if you are going to install something that requires Python support, Python and pip need to be available. In most cases, if you are coding in a given language, you already have the tools

Mason needs to do its thing. The main thing that Mason takes care of is ensuring that the tools are installed in such a way that other NeoVim plugins can find them.

The hardest part with Mason is knowing what tool you want to install. I was already using Biome when I set up LazyVim, so I knew I was going to need to install editor support for it. That was no problem; just find `biome` in the Mason list (like any window, it is scrollable, searchable, and seekable, and Mason helpfully puts everything in alphabetical order).

But when I started working on this book, I decided I needed an advanced Markdown formatter, and I had no idea which one to use. I could search the window for `markdown` and then press `Enter` on any matching lines, which gives a description and some other information, but I had to do some research with a web browser and AI chat bot, (along with a little trial and error) before I found the right tool for me.

Unfortunately, I can't help you with figuring out what is right for you, but once you find the tool in Mason, just use `i` to install the package under the cursor. The only other command you will use frequently in Mason is `Shift-U` to update all installed tools, and you can look up the rest with `g?`.

Validating Things Installed Cleanly

As good as both LazyExtras and Mason are at installing language servers, linting, and formatting tools, setting them up is one of the places most likely to go wrong, no matter which editor you are using. So now is a good time to introduce several commands to validate that things are working as expected.

First, LazyVim pops up notifications in the upper right corner, as you have seen with the plugin updates. These disappear after a few seconds. Every once in a while, you need to be able to refer back to them.

The secret is to use the keybinding `<Space>sn` to open the "Noice" search menu. Noice is the plugin that provides those little popups. Most often, you'll want to follow this with either an `a` or an `l` to see all recent Noice messages, or just the last one. You can also use `<Space>snd` to dismiss any currently open notifications, but honestly, by the time you've completed those four keystrokes, they notifications have probably disappeared themselves already!

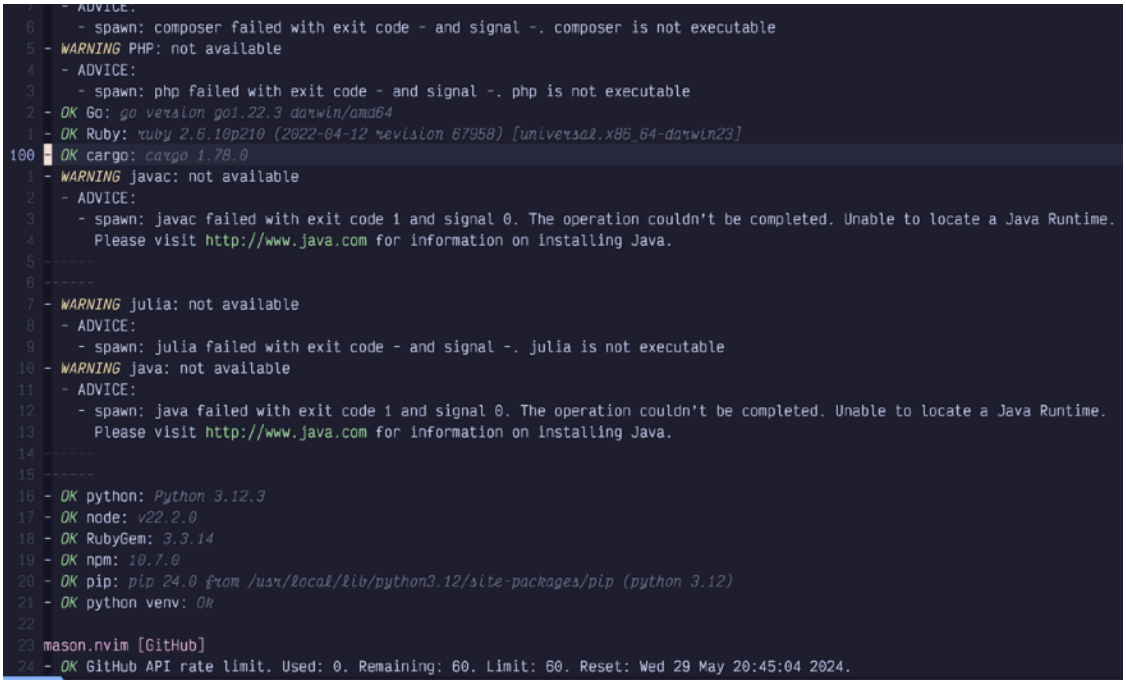
The second command you'll use regularly is `<Space>cl`, which runs the command `:LspInfo`. It displays information about any language servers that are currently running and which buffers they are attached to. For example, while editing this Markdown document, my LSPInfo window looks like this:

In this case, everything looks fine (though I'm surprised the tailwind server is associated with Markdown), but if your LSP isn't behaving correctly, this window might give you a hint as to what the problem might be.

If your LSP is having temporary problems–like showing incorrect diagnostics or unable to find a file you know is there–sometimes it just needs to be given a good kick with `:LspRestart`. The Svelte language server has a nasty habit of not picking up new files, so I've been using this one often enough lately to add a keybinding for it. Most language servers are more cooperative, though.

Two other super useful commands are `:checkhealth` and `:LazyHealth`. Both provide information about the health of various installed plugins. The former is a NeoVim command that plugins can register themselves with to provide plugin health information, while the latter provides LazyVim specific information. There is a lot of overlap in the output, but I find the `:LazyHealth` output is easier to read, and the `:checkhealth` output to be a bit more comprehensive. So I usually use `:LazyHealth` first and switch to `checkhealth` only if `:LazyHealth` didn't provide the information I need.

Don't expect to see green check marks across the board; you'll make yourself crazy. For example, my `checkhealth` output contains a bunch of warnings from Mason:



Tools that I have used recently (and also Ruby for some reason) are installed, and I have warnings for languages that I don't generally need to edit files in. So if you don't code in Java, there's no reason to drive yourself crazy trying to make the `java` warning go away.

Diagnostics

Language Servers fulfill several useful functions, including identifying code problems, linting, formatting, context-aware code navigation and documentation. We'll discuss all of these between this and the next chapter.

We already got a peek at diagnostics in an earlier chapter, when we discussed jumping between error messages with the unimpaired keybindings `[d`, `[w`, `[e`, `]d`, `]w`, and `]e`. Diagnostics show up as little squiggles under specific sections of text and when you jump to them, you usually get a small overlay window telling you what is wrong at that location. For example, I have a simple typo causing an error in this screenshot:

I misspelled “tracingMiddleware”, and I get a helpful error message on that line in the inlay hints, and a window pops up when I navigate to that error with `]d`. This window sometimes has more information than the inlay hint. In addition, the line that imports the correctly spelled variable is showing a hint telling me that it isn’t used.

The colour of the diagnostic conveys the severity–whether it is a hint, a warning, or an error–so you can decide whether it is valuable to fix. I generally try to either fix or silence all errors, as they become less useful if there is much noise.

If the window doesn’t pop up when you navigate to the diagnostic, you can use the `<Space>cd` keybinding to invoke it as long as your cursor is positioned somewhere within the underlined text. You can make the window disappear by moving your cursor with any motion key.

Trouble and Quick Fix

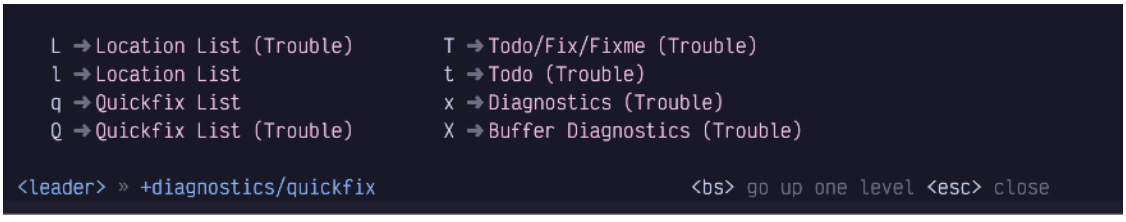
You can also navigate diagnostics using the Trouble menu. Trouble is a LazyVim plugin that provides an “enhanced quick fix” experience. Which is probably meaningless to you if you are new to Vim and don’t know what quick fix means!

The quick fix window is essentially a list of files and line numbers that have been tagged as “interesting” for some reason, where that reason depends on context. It can be used to represent multi-file search results, diagnostics, compiler error messages, and more, depending on how you open it. You can easily hop between the targeted locations, making changes or corrections without losing the context of what you were searching for.

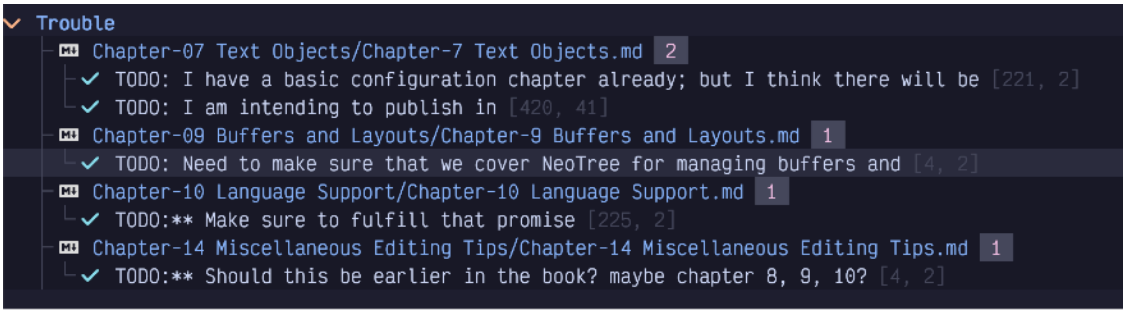
In its simpler form, Trouble is the same thing, just a little bit prettier to look at, with colours, icons, and nice groupings when fix locations are in multiple files.

However, Trouble is rather tricky to talk about right now, because at the time of writing there is a new Trouble v3 beta that can do a lot more than that You can enable the beta from the LazyVim Extras menu, although the odds are that by the time this is published it will already be the default. It supersedes some other plugins, and I will be discussing Trouble v3 instead of its alternatives later in this and the next chapter.

The *contents* of the quickfix and trouble windows depend entirely on how you open them. Most of them are accessible from the `<space>x` (I assume the `x` stands for "fi**X**") menu, which looks something like this:



Let's take to-dos as an example, as I have a lot of them in this book. It's weird saying that because they'll be gone by the time you see it, but this screenshot will live on:



The cool thing about this list of locations is that they are not all in the same file. Without Trouble, I could navigate between to-dos in *current* file using the `[t` and `]t` keys. However, using Trouble, I can navigate between to-dos in multiple files by moving my cursor to the appropriate line in the trouble window and hitting Enter. It will open the file and move the cursor right to the "troubling" line.

Or you can use the commands `[q` and `]q`, which will navigate between quickfix OR trouble locations, no matter which file they are in, without ever focusing the Trouble window.

For diagnostics, open the trouble menu with `<Space>xx` or `<Space>xX`. The lowercase version shows the diagnostics in the current file for a quick overview while the "but bigger" uppercase X shows all the diagnostics in the current workspace (although it depends a bit

on the language servers; some language servers only show you diagnostics for all currently open buffers, not the whole project).

If you're wondering what the "Location List" is, it's a quickfix window that is associated with the current window (NOT buffer). I never use it; my brain can only handle fixing one problem at a time, even if it is in hundreds of files!

We'll meet the Trouble window again when we discuss searching in a couple more chapters.

Code Actions

One of the things that made VS Code seem magical when it came out was code actions. Not that they existed, as the concept has been around for a long time, but that they WORKED. Nowadays, we all kind of take them for granted.

You may be used to accessing code actions by moving your hands to the mouse and clicking a light bulb or right clicking a diagnostic. In LazyVim it is (of course) a keybinding. Navigate to a diagnostic using whatever keybindings work for you (I live by `]d`, personally) and then invoke the `<Space>ca` menu where `c` and `a` mean "code action." A picker menu will pop up with one of sometimes several actions you can take. You can use the arrow keys or `<Escape>` followed by `j` and `k` to navigate between them, or you can enter a number or any text from the line to filter. Hit `<Enter>` to perform the action, or `<Escape><Escape>` to cancel the menu (just one escape allows you to enter normal mode in the search box so you can use the many LazyVim navigation keystrokes that you are becoming used to).

Linting

Linting is *mostly* handled using the `nvim-lint` plugin instead of the LSP. This was a major pain point in my pre-LazyVim days because getting the LSP and linter cooperating often required some serious troubleshooting. And then throw formatting into the mix and you've lost a day or two. To be fair, this was true when I used VS Code, too.

Using LazyVim, it is actually likely that you don't know who is doing the linting for you. I honestly don't. Some of my diagnostics come from the LSP and others come from the linter. I don't bother to question the source of the errors; I just fix them.

The hard part with linting (at least, when it doesn't work automatically) will be making sure that the appropriate linter is installed (Mason has your back here), and configured correctly. If you are lucky and the languages you love to work in have Lazy Extras, then it is probably already configured correctly. Otherwise, you may have to do a little tweaking. The tweaking involved is, sadly, language-dependent, but you'll probably need something like this in a example `extend-nvim-lint.lua` that in your plugins directory:

```lua
return {
  "mfussenegger/nvim-lint",
  opts = {
    linters_by_ft = {
      typescript = {
        -- lint settings for typescript
      }
    },
  },
}
```

Read the `nvim-lint` Readme for more information and refer to the LazyVim documentation for this configuration if you need further clarification.

The nice thing is that once you have your linting configured, the errors will show up using the same diagnostics described above and you can engage with them using the same keybindings, trouble window, code actions, etc.

Formatting

Similar to linting, code formatting *can* be handled by some LSPs, but people have realized that using the language server is often more complicated than just invoking a formatter directly. So LazyVim ships with the `conform.nvim` plugin.

Also similar to linting, if you are lucky, it will Just Work™ after you install the appropriate Lazy Extra and/or Mason tool. However, if you don't like the default formatter (or it's not working), you will have to familiarize yourself with the LazyVim and `conform.nvim` documentation to figure out the exact incantation required.

The only formatter I've had to manually configure is using Prettier for Markdown. It looks eerily similar to the `nvim-lint` configuration:

```lua
return {
  "stevearc/conform.nvim",
  opts = {
    formatters_by_ft = {
      ["markdown"] = { "prettier" },
    },
  },
}
```

Once it's set up (I acknowledge this may be no mean feat), formatting in LazyVim is typically fire and forget: Save your file and it formats. If you want to invoke it manually

without saving, use the `<Space>cf` keybinding. I can't stress how lucky you are that this is the case; without LazyVim, countless hours have been wasted (by me, and by most every vim user) trying to get the autocommands for format on save to work!

Configuring Non-standard LSPs

If you have installed an LSP that LazyVim isn't aware of, you may need to tweak the `nvim-lspconfig` plugin. You will minimally need to let it know that it is available, and possibly to configure it to your needs. For example, one of my favourite programming languages is Rescript, which doesn't have a huge ecosystem and therefore, has no LazyVim extra. I was able to install the language server with Mason easily enough, but I also needed to add the following to my `extend-lspconfig.lua` file for LazyVim to pick it up:

```
return {
  "neovim/nvim-lspconfig",
  opts = {
    servers = {
      rescriptls = {},
    },
  },
}
```

Summary

In this chapter, we learned how LazyVim integrates the language server protocol that VS Code brought to the world. It is *usually* quick and painless, which is more than can be said for manually configuring LSPs. However, there may be some headaches especially around linting and formatting. This is true in any editor, sometimes they hold your hand and sometimes they get in your way. If you get stuck, hit us up in the LazyVim discussions group on Github (but search it first; you're probably not the first person to have trouble).

In the next chapter, we'll learn more about navigating *code* using LSPs, TreeSitter, and several plugins.

Chapter 11: Navigating Source Files

In previous chapters, we've learned many different ways to navigate within a single buffer, as well as between open tabs and windows. This chapter will go into detail of different ways to navigate *between* source files.

Go To Definition

In my opinion, "go to definition" is the most valuable feature language servers have brought us. Major IDEs have supported it for compiled languages for eons, but dynamically typed languages–such as my beloved Python–have always been hell for static analysis, and such features were often pretty hit or miss.

As one of the best-named editor features of all time, go to definition jumps your cursor from whatever keyword it is currently sitting on to the place where that keyword is defined, regardless of what file it is in.

Most often, I use this when I am looking at a call site for a function and want to see the function itself. A simple press of `gd` (go to definition also has one of the more memorable LazyVim keybindings of all time) will take me there.

Depending on how good the LSP for the language I am editing is, this often even allows me to jump into library files or type declarations for third party modules so I can see what is really going on.

Go to definition is context dependent, but will usually do exactly what you expect. If you are looking at a variable, `gd` will jump to wherever the variable is initialized. If your cursor is on a class name, it will jump to wherever the class is defined.

Typically, once you've jumped to a definition and learned what you need to learn from that file, you'll immediately want to jump back to where you started. You can do this easily using `Control-o`, as we discussed in Chapter 3 (and `Control-i` can move forward in your jump history).

Go To References

The inverse of the Go to Definition command is "go to references". If you are looking at a function, variable, type, etc, and want to see all the places that variable is accessed, use the `gr` command.

Unlike with a definition or declaration, there will typically be more than one reference to a given word (A variable in isolation is a useless variable indeed). So when you type `gr` it won't immediately jump to a location. Instead it will pop up a picker view of all the references to the word that was under your cursor, with all the preview and filtering luxury that Telescope and fzf.lua always bring.

It is common to want to perform some action–such as a rename or adding an argument or what have you–at every reference. You *could* keep showing the picker by hitting `gr` again or by using the `<Space>sR` keybinding which resumes your previous picker search. However, it is often much more useful to use the trouble list that we learned about in the previous chapter.

To do that, use `gr` to show the references in a picker as usual. Then use `Control-t` to show each of the files in the trouble window. Now you can use `]q` and `[q` to jump between them without going to the trouble of showing the picker again. If you'd rather use the less fancy quick fix window, use `Control-q` from the picker instead of `Control-t`.

Tip: `Control-t` and `Control-q` work on most pickers. I recommend getting used to using them any time you want a less-temporary list of items than will give you.

Context-specific Help

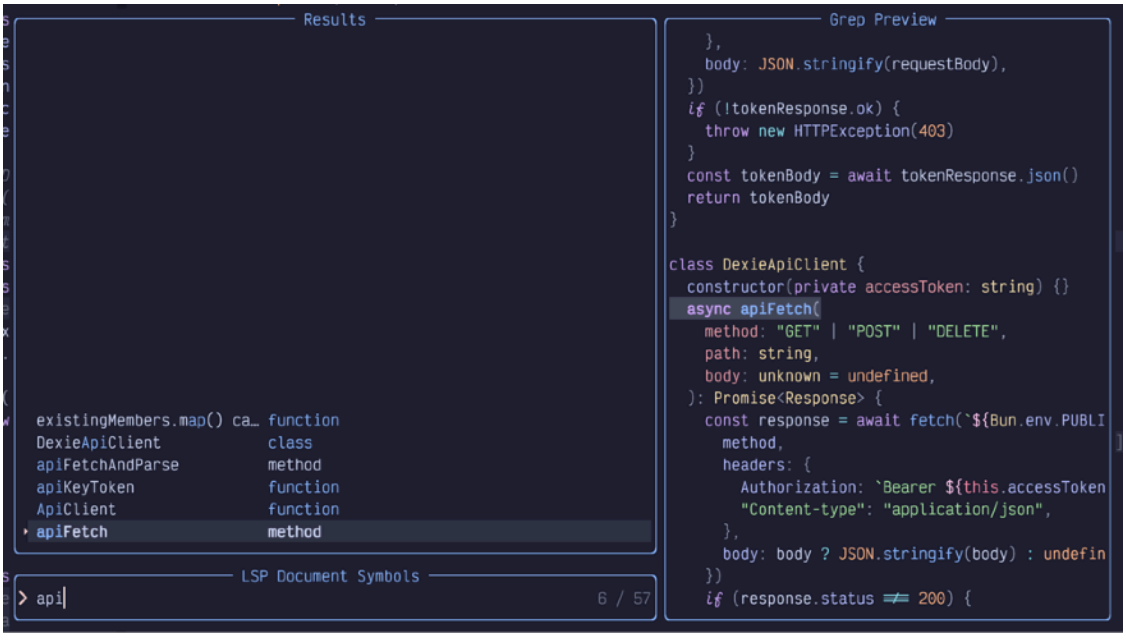
Most non-modal editors show you some help or "hover" text when you hold your mouse over a word or symbol. The quantity and value of this text varies widely depending on the LSP, but usually includes a function signature and documentation for the word under the cursor.

It's probably possible to set up Neovim to show help texts on hover, but why would you move your hand to the mouse when LazyVim has such amazing navigation on the keyboard? Instead, use the (shifted) `K` keybinding. Yeah, `K` is a pretty stupid mnemonic to remember, but `H` and `?` were already taken. In fact, the `K` stands for "keywordprog", which is a legacy Vim concept that has been superseded by language servers in the modern world. So LazyVim reused the keybinding.

Listing symbols

Another handy LSP feature is to search all the symbols in the current file or project. If you are editing a particularly long file and need to jump to a function that is not terribly close to your cursor, you might use the `<Space>ss` command (mnemonic is "search symbols"). As hinted by the double `s`, this is expected to be a fairly common action.

The dialog that pops up should be fairly familiar by now, as it's the usual picker:



So you already know how to use it. However, I want to remind you of a couple Telescope tips that make it more useful:

Most of the time when I'm using this symbol picker, I only care about functions, or sometimes classes. So the fields and properties scattered in the screenshot above are just a distraction. It **is** possible to configure the picker to only show certain kinds of symbols, but I prefer a quick trick that allows me to narrow it down to just functions: type (part of) the word `function`.

Since the picker includes the word "function" in the second column of the results, the picker merrily filters out all the lines that don't have that word in them. Handy.

Better yet, I can input a space *after* the word "function" to inform the picker to perform subsequent searching back in the first column. So "func api" filters all functions that have the word "api" in them.

My second tip is to not forget about the `Control-t` and `Control-q` shortcuts to dump picker results into the quickfix or trouble list. It generates a quick and dirty table of contents of whatever symbols you filtered for.

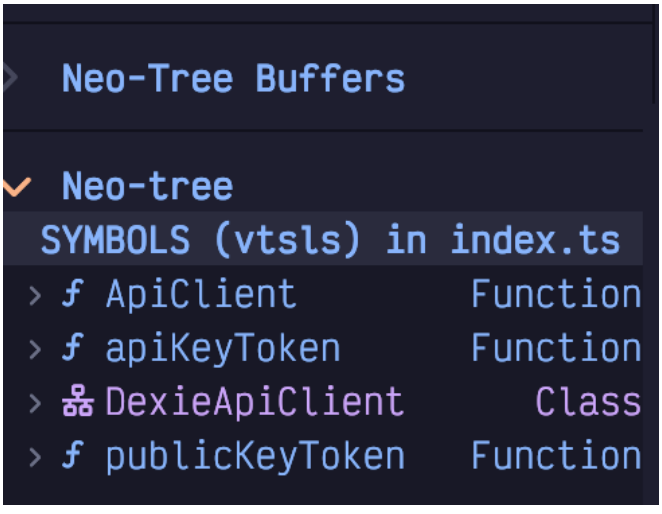
If you want to search all the symbols in your whole project, use the "but bigger" mnemonic. `<Space>ss` will perform such a search. However, be warned that not all LSPs support workspace symbol search. Some only search in currently open files, and even many of those that fully support workspace symbol search are unusably slow.

NeoTree also has a symbols outline

If you like the NeoTree sidebar for file picking, you may appreciate that it also supports a symbol list for the currently focused file. It looks similar to the picker selector but has the (dubious, in my opinion) benefit of always being visible.

At the time of writing, NeoTree claims that the symbol picker is "experimental" in its Readme, so there isn't a keybinding to display it, at least by default.

Instead, use the command `:Neotree document_symbols` to render the symbol picker in your Neotree sidebar:



You can navigate to a symbol in the document either by double clicking it with your mouse or by moving your cursor to the line that contains the symbol you want to jump to and pressing `<Enter>`. You can also use `s` or `S` to open a new view of the buffer at the given symbol in a vertical or horizontal split.

If you find that you just can't get enough of the NeoTree symbol picker, you'll probably want to add a keyboard shortcut for that command. The easiest way is to add the following line to your `keymaps.lua`:

```
vim.keymap.set(
  "n",
  "<leader>sO",
  "<cmd>Neotree document_symbols<cr>",
  { desc = "Document Symbols (Neotree)" })
```

Feel free to use a different keymap if `<Space>s<Shift-O>` doesn't suit.

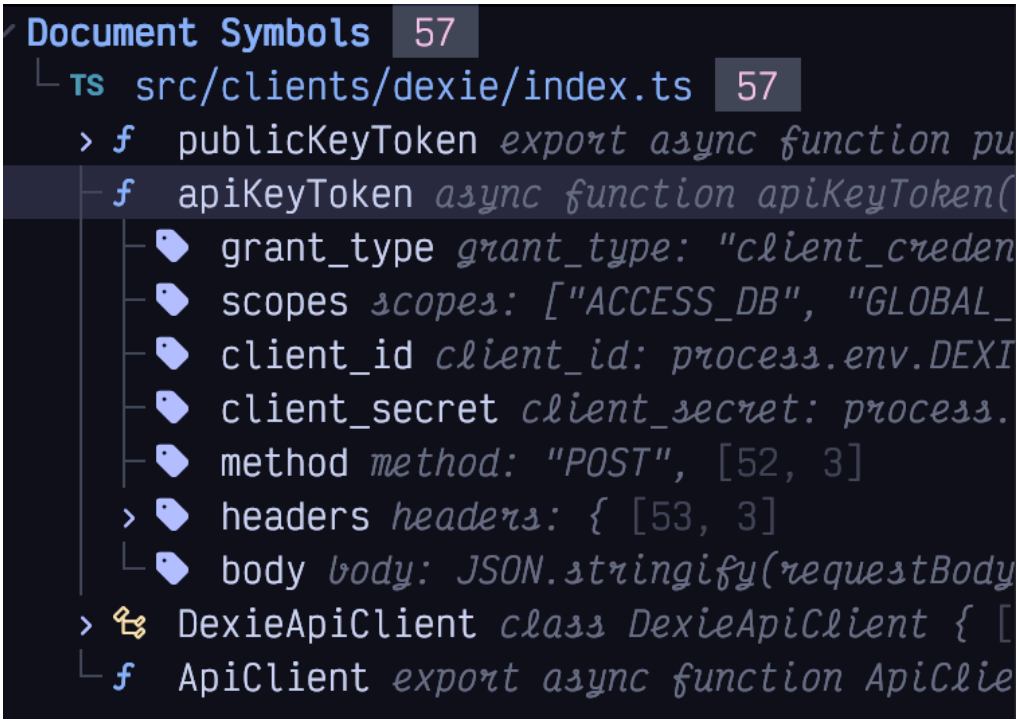
However, it is more common to group keymaps with the configuration for the plugin that the keymap invokes. So you could also do something like this (in any `.lua` file in the plugins directory, say `extend-neotree.lua`):

```
return {
  {
    "nvim-neo-tree/neo-tree.nvim",
    keys = {
      {
        "<leader>sO",
        "<cmd>Neotree document_symbols<cr>",
        desc = "Document Symbols (Neotree)",
      },
    },
  },
}
```

This is especially useful if you decide to extend or customize other aspects of the Neotree configuration in addition to keymaps.

...and so does Trouble!

You can also open a symbols outline using the Trouble plugin. Unlike most trouble windows, it opens in a right sidebar by default. It creates a lovely tree view and you can even collapse and expand the tree nodes using the folds keybindings we discussed in Chapter 9.



You can resize the trouble window using the same keybindings you usually use for resizing windows (\< and \>). As you move the cursor over the trouble window, the symbol it is over will automatically scroll into view.

The fastest way to use the trouble window is to use seek mode. Recall that seek mode can jump to *any* currently visible window, which includes trouble. So if I am currently editing the above file and my cursor is currently somewhere near the end of the file, I can use `spub` to enter seek mode and search for the characters "pub". This will place a label on the `publicKeyToken` in the trouble window. If I hit that label, my cursor jumps to the trouble window and my editor window immediately scrolls to the function in question. Now I just have to hit `Enter` to move the cursor back to the file I'm editing.

Context

The `nvim-treesitter-context` extra is a helpful way to know where you are in the current file. It uses treesitter to figure out which functions and types you are in, and then pins the lines that define those types to the top of the editor. Enable it as usual by visiting `:LazyExtras` and hitting `x` over the line that contains `nvim-treesitter-context`.

This plugin keeps track of which class or function your cursor is currently in. If the function or type definition is so long that the signature scrolls off the screen, it will helpfully copy that signature into the first line or lines of the code window, highlighted with a slightly different background colour.

This is easier to describe with a reference image, so consider this screenshot:

In this image, the first two lines, which are slightly shaded, are providing context, rather than being part of the buffer. The first line tells me that I am in the `DexieApiClient` class and the second line tells me that I'm currently looking at the `forceAddMemberToRealm` function.

Especially notice the relative line number column. The `class DexieApiClient` line is 110 lines above my current cursor position, and the `async forceAddMemberToRealm` line is 28 lines above it. The first visible line of the function is only 14 lines above my current cursor position.

The effect is quite subtle, but the definitions that make their way into this context section tend to be really useful while coding. If they fit on one line I can see function signatures and return types. You really don't notice how often you have to scroll up to see what a variable

is named in a function signature until you don't have to do it anymore! And if you DO need to scroll up to the signature, simply type the relative line number followed by `k` and you're there with no searching required.

If you need to disable the context temporarily, use the keybinding `<Space>ut`. We haven't seen much of the `<Space>u` menu yet, where you can toggle various User Interface effects. This is largely because the default user interface is configured well enough that you don't want to change it often!

Navigating with (book)marks

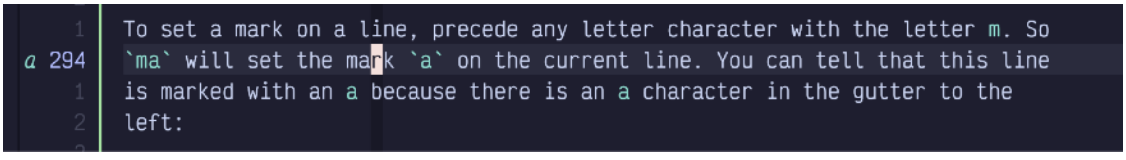
You already know how to navigate through your history with `Control-o` and `Control-i`, and to jump around documents effectively using a wide variety of motions.

Vim also includes a "bookmarks" feature, although it's referred to as "marks" I assume because the `m` character was still free on the vim keymaps.

Marks are built-into Vim and LazyVim has (as usual) added a few minor improvements.

Much like the registers that we covered in an earlier chapters, marks can be assigned to each letter of the alphabet. Additionally, certain punctuation characters represent special system-set marks that you can jump to, but not set.

To set a mark on a line, precede any letter character with the letter `m`. So `ma` will set the mark `a` on the current line. You can tell that this line is marked with an `a` because there is an `a` character in the gutter to the left:

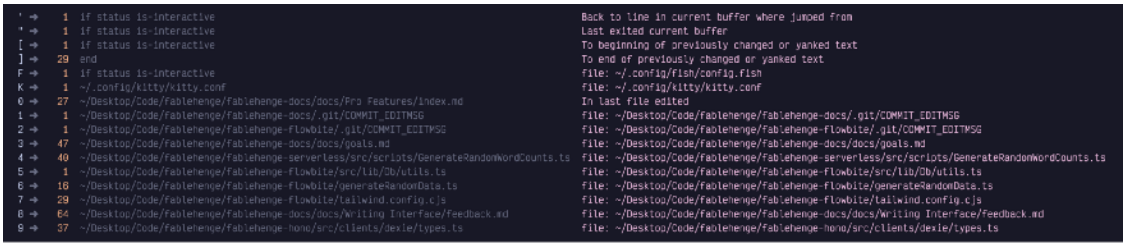


Now I can jump to the line marked with `a` from anywhere *in the current file* by using an apostrophe followed by `a`.

I don't use this very often because other tools tend to be more useful for navigating a file than manually setting a mark. However, if I had marked the line with a capital letter (e.g. `mA`), I would be able to jump to the mark no matter which file was open using `'A`.

So essentially, you can have up to 26 local marks for each file you ever open, as well as 26 global marks that you can access from any file.

Conveniently, if I just type a single apostrophe (in normal mode), LazyVim will pop up a menu of all the marks currently available to jump to:



This list shows the lowercase `a` mark that I've set in this file, several system marks that I can jump to using punctuation (notice the descriptions for each of those marks to the right so you don't have to memorize them), two global marks I use to jump to my kitty and fish configuration files, and the ten numbered marks.

I find the numbered marks to be kind of useless. The essentially refer to the file and cursor location of the last time you closed Neovim. I don't close Neovim that often unless I'm editing a commit message or pull request description in a temporary instance, so my numbered marks are mostly just those kinds of temporary files. If I need to get back to where I was previously, the `<Space>qs` keybinding to restore session is typically more useful than the numbered marks.

The menu that pops up when you press the apostrophe key is usually sufficient to find marks, but you can also use the `<Space>sm` keybinding to search marks in a picker. I don't usually have enough marks active for this to be useful, but if you've got a lot of global and local marks set and you can't remember which letter is associated with a given one, it might help to use the picker to search for the contents of the line you have marked.

Once you've set a mark, you'll eventually be dogged by the question "how do I get rid of it?" Deleting marks is probably up there with "how do I quit vim" for common queries! There isn't a keybinding for deleting marks. Instead, you need to use the command `:delmarks <mark>` to delete the given mark. So to get rid of the `a` mark in this file, I used the command `:delmarks a`. You don't have to be on the marked line to delete the mark.

Marks can be used in place of line numbers in ranges in command mode. For example, If you want to write the text between mark a and mark b to a file, you could do `:'a,'bwrite somefile.txt`. If you've seen the `'<,'>` in front of colon lines when you have text selected, that is because `'<` and `'>` represent the start and end of the most recent visual selection. So rather than manually setting `ma` and `mb` you can visually select the thing you want to write and have the marks pre-filled for you.

You can also use `'<` and `'>` to jump to the beginning or end of the most recent selection even if it has since been deselected.

The other symbol mark that I use a lot is `'.` which jumps to the last place I inserted or changed text. This can sometimes be quicker than a series of `Control-o` keypresses.

Summary

In this chapter, we learned how to navigate code files using goto definition and references, and various document_symbol plugins.

We saw how LazyVim gives us context on our current location in the document and how to look up documentation for the symbol under the cursor.

Finally, we covered vim marks, a more manual process of tracking locations that you may want to jump to.

In the next chapter, we'll learn all about searching and substituting text both in the current file and globally across a project.

Chapter 12: Searching…

Chapter 12: Searching… - LazyVim for Ambitious Developers

As with all the remaining topics in this book, it's kind of amazing that we've gotten this far without covering searching. Find and replace in Vim has always been far more powerful and nuanced than in most editors, which just give you a little dialog with three fields and, if you're lucky, a check box to specify regular expressions.

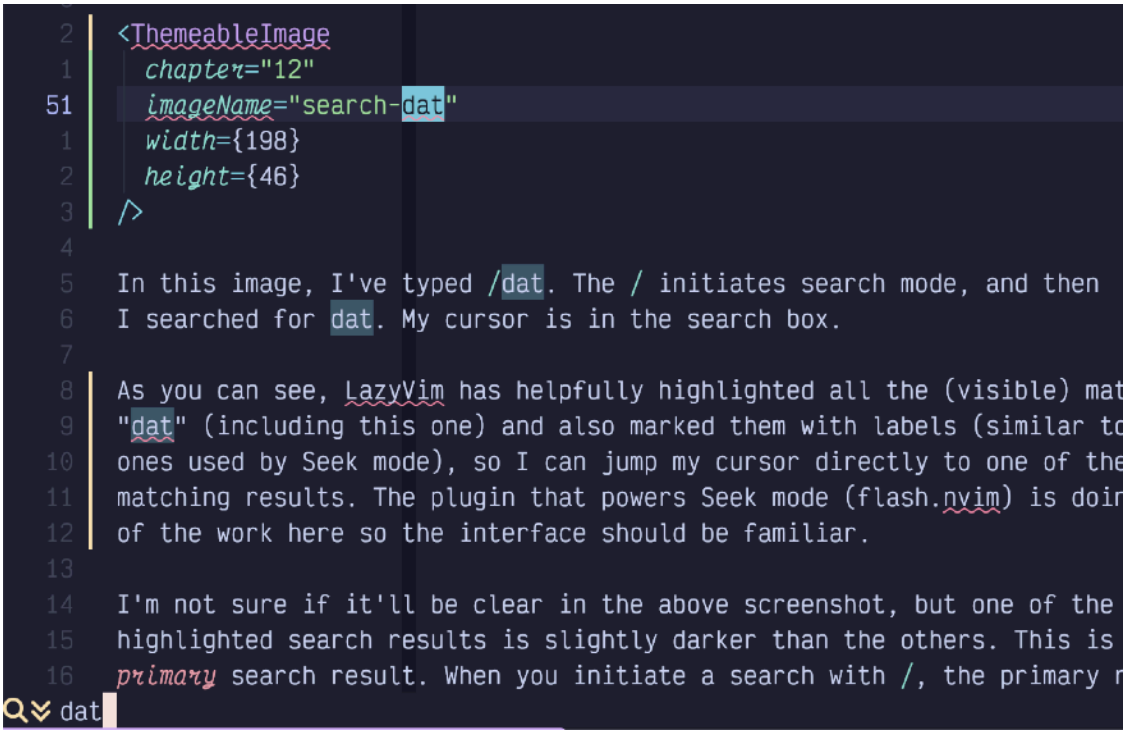
LazyVim, as usual, extends Neovim's already-powerful search feature to make it both easier to use, more useful, and prettier. You already know about the neat Seek (`s`) and treesitter (`S`) modes for navigating to and selecting objects you can see, as well as their remote operator-pending objects counterparts: `r` and `R`. These all work fine so long as the text you are looking for is currently visible. However, when you need to search a file and have it automatically scroll to search results, they are not sufficient.

Search in current file

To search for a pattern in vim use the `/` command in normal mode. The mnemonic is that the `/` key is also the question mark key, and searching for something is a kind of question.

> Even many tools that are not considered "modal" have adopted vi's `/` as a command to invoke search. For example, the exceptional Linear task tracking tool uses `/` to begin a search, as does the ubiquitous GitHub.

The first time you type a (normal mode) `/` in Lazy Vim, there is a good chance you'll lose your cursor! It will not pop up a new window in the editor. Instead, `/` will take over the current file's status bar with a little magnifying glass icon:



In this image, I've typed `/dat`. The `/` initiates search mode, and then I searched for `dat`. My cursor is in the search box.

As you can see, LazyVim has helpfully highlighted all the (visible) matches for "dat" (including this one) The "primary" result will always be the first matching result *after* the point where your cursor was when you hit `/`.

Your cursor will jump to the primary result if you press `Enter` to confirm your search (Press `Escape` to cancel it, as usual).

At this point, you are back in normal mode and can edit the buffer as usual. However, you will notice that all the highlighted results are still highlighted. You can easily jump to the next result using the `n` (for next) key. This command accepts a count, so you can use `3n` to jump to the third result after the current cursor position.

The search will wrap to the top of the document if there are no more matching results at the bottom. If you know how far you need to jump, you can use a count with the `/` command as well, as in `3/something` to jump forward to the third `something`. Figuring out how far you need to jump requires some mental agility, though, so it's usually faster to use Seek mode.

If you `n` too far, you can use `Shift-N` to move the cursor to the previous result instead. And if you *know* you need to jump to a previous result, you can initiate the search with `?` (i.e. `shift-/`) instead of just `/`.

If you have used Vim before, I should warn you that this behaviour of `n` and `N` is different (and more useful) from the default Neovim behaviour. They used to "repeat the last `/` or `?` command," so `n` would continue *up* the document if you started with `?`. The LazyVim model is easier to remember; `n` always means "next down" and `N` always means "previous up".

Ignore case

If you enter your search term as all lowercase letters, LazyVim will ignore case by default, but if you include a capital letter in your search term, it will enable case sensitivity. So searching for `in` will match `in` and `In`, but searching for `In` will only match `In`.

If you expressly want to search for only lowercase matches, you can modify the search term by inserting the two characters `\C` (that `C` is capitalized) somewhere in it.

Conveniently, it doesn't have to be at the beginning of the search term; if there is a `\C` anywhere in the search string, it will make the whole search case sensitive. So, imagine you were looking for the lowercase word "initiate". If you start typing `in` and realize it's matching a bunch of unnecessary `In` because ignore case is enabled, you can append `\C` (so you end up with `in\C`) to switch to ignore case mode before typing `iti` (so the total search string is `in\Citi`).

If you want to disable ignore case temporarily, type the colon command `:set noignorecase`. This will only last until you exit NeoVim, or explicitly enable it again with `:set ignorecase`.

If you want to make the change permanent, open your `options.lua` file and add `vim.opt.ignorecase = false` somewhere in it. Note that now if you want to make any specific search case *in*sensitive, you need to use lowercase `\c` instead of `\C` in the search phrase.

The `\C` trick seems kind of weird at first, but when you think about the alternative used in most code editors, where you have to move your hand to your mouse, target a tiny

checkbox with a label like `wW`, and click it, then refocus the search box and continue typing, you'll probably decide that `\c` is faster.

Regular expressions

Vim searches use regular expressions by default. But they are kind of strange regular expressions.

Ok, I admit that *all* regular expressions are kind of strange. Vim's are only strange in comparison to the pcre-style regular expressions that are common in most modern programming languages. Luckily, if you are searching text, you probably don't need the full complexity the PERL-compatible expressions offer.

I don't have space in this book to instruct in regular expression syntax, so I'll just mention some of the main go-tos and leave you to look up the rest:

- `.` matches *any* single character. If you need to search for a literal period, escape it with `\.`.
- `\S` matches any *non-whitespace* character.
- The `*` character matches the preceding expression zero or more times. Notably, `.*` will match any string of characters of arbitrary length.
- The `\+` string will match the preceding expression one or more times. (This is notably different from most regular expression parsers I've seen, where you don't need the `\` before the + to match one or more). It can be combined with e.g. `\S` to match any word without spaces: `\S\+`.
- `\=` can be used to match the preceding pattern zero or one times. Useful for things like `https\=:` where the "s" is optional. This pattern is usually `?` in most regular expression engines, and in fact `\?` also works for this. However, it would confuse vim when the command to invoke search backwards is `?`, so `\=` wins.
- `\\` matches a literal backslash and `\/` matches a literal forward slash.

In general, if you know Perl Compatible regular expressions, you'll find you need a lot more backslashes in vim. That said, the vast majority of code editor searches are covered by the above.

If you want to "disable" regular expression matching for a specific search, place `\V` at the beginning of the line (or in the middle of the line if you only need to disable it for the remaining part of the search). The "V" stands for "very nomagic", and if you want to be extremely confused, type `:help magic`. It is so confusing, in fact, that you will prefer to learn to just use regular expressions (yes, I am aware how very confusing that is. Vim's interpretation of magic is worse).

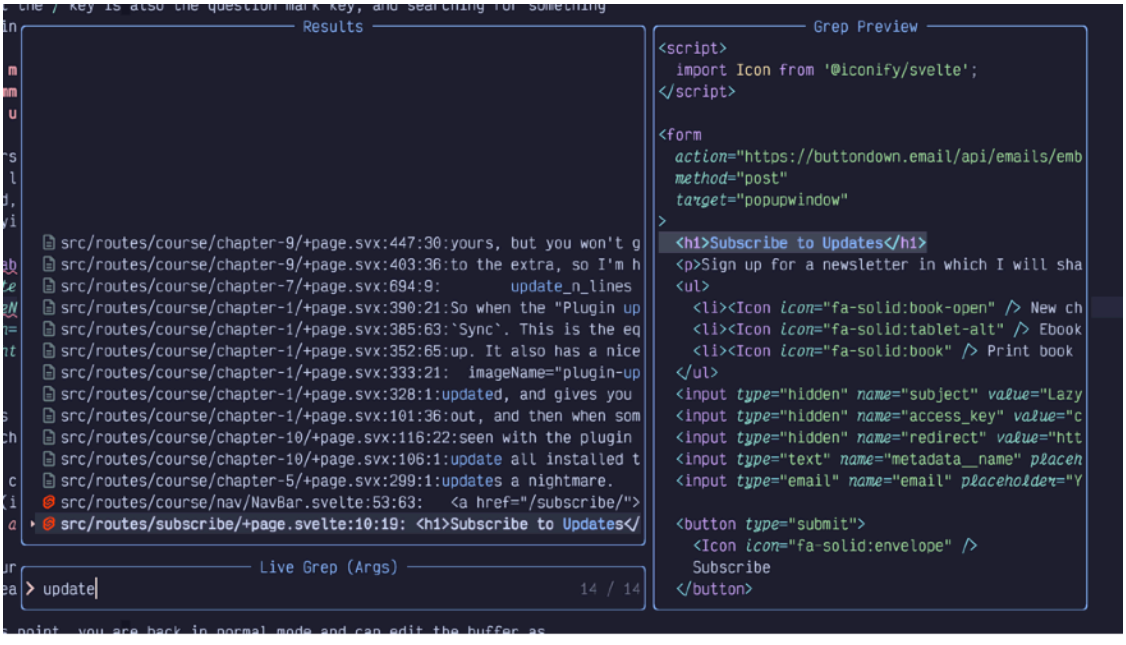
If you desperately need a regular expression to do something you can ~~ask ChatGPT~~ skim through `:help regular expressions` to find the syntax you need. You will come away either enlightened or frustrated.

Search In Project

If you need to search for a word across your entire codebase, instead of just in one file, use the command `<Space>/` instead of just `/`. It will pop up the ever-so-familiar picker, this time in "live_grep" mode.

> Make sure you have ripgrep installed and available on your path as `rg`, as that is what the pickers use under the hood.

Type the string you are looking for. The results will show up in the left side and the file will display in a preview on the right so you can be sure you found the right one:



Remember that you can add labels to Telescope results by pressing `<Esc>s` to enter Seek mode, or to `fzf.lua` results using `Control-x`. I find this more useful in the `live_grep` window because unlike most pickers, a space in `live_grep` is sent as a literal space to `ripgrep`, instead of allowing me to narrow the search results by searching for something earlier in the line.

I should also remind you that since this is a picker, you can press `ctrl-t` while it is open to put all the search results into the Trouble window so you can navigate them while editing (using `]q` and `[q`).

Annoyingly, this search mode is completely different from vim's built-in search. It just passes your pattern to `ripgrep` and behaves the way `ripgrep` does. And `ripgrep` doesn't know about things like Vim's strange regular expression engine. It *does* support regular expressions, but they use maddeningly different syntax from vim. Which is to say, the same syntax as pretty much everything that isn't vim. It's vim that's maddening here, not `ripgrep`. Just so we're all clear.

Ripgrep itself accepts a multitude of command-line options, but by default, the `live_grep` feature doesn't support passing arguments to ripgrep to tweak your query. The Telescope project does provide a `telescope-live-grep-args` extension that you can enable if you want to be a `live_grep` power user

> To the best of my knowledge, fzf.lua doesn't have an equivalent to Telescope's live-grep-args.

Setting up the extension pushes LazyVim's (mostly) reusable plugin configuration system to its limit, though. So let's configure `telescope-live-grep-args` as much for a tutorial on how to handle tricky extension configurations as because you might actually want the feature. There are a lot of neat Telescope extensions out there, and you'll eventually want to know how to set others up that LazyVim doesn't ship by default.

Setting up a Telescope extension

Start by visiting the `telescope/telescope-live-grep-args.nvim`. You'll find installation instructions for `Lazy.nvim` (that's the plugin manager, NOT the LazyVim distro itself) that, at time of writing look like this:

```
-- This is not helpful with LazyVim
use {
  "nvim-telescope/telescope.nvim",
  dependencies = {
    {
        "nvim-telescope/telescope-live-grep-args.nvim" ,
    },
  },
  config = function()
    require("telescope").load_extension("live_grep_args")
```

```
  end
}
```

The reason I added the "not helpful" comment there is the call to `config`. LazyVim already configures Telescope with a fairly complex function that you can find under the `editor` section of the plugins menu on the LazyVim website (click the `Full spec`) tab.

For the most part, LazyVim does a good job of merging its own defaults with any customizations you do with the various plugins it sets up. It's easy to change or remove keybindings or override the options that get passed into a `setup` function, for example.

But it's not easy to override `config`.

You *could* do something like this:

```
config = function(_, opts)
  require("telescope").setup(opts)
  require("telescope").load_extension("live_grep_args")

end
```

This works because the default behaviour of `config` in `lazy.nvim` is to call `require(<the_plugin>).setup(opts)`. So we're basically copying the contents of that function into our custom function. But if LazyVim happened to have a very complicated `config` for Telescope, you would have to copy the whole thing in, and it would eventually get out of date with any changes that LazyVim makes in the future, and that wouldn't be fun for you to maintain. More importantly, it runs a very real risk of clobbering any Telescope-related changes that LazyVim makes with other plugins or extras.

In general, I try very hard to avoid implementing `config` when overriding LazyVim's defaults. It's fine to have a custom implementation of `config` if I am adding a new plugin that LazyVim isn't aware of, since I'm already responsible for maintaining it. But when I'm *customizing* plugins, I try not to override `config`.

The secret, in this case, is to use the "dependencies" feature of `lazy.nvim`. The "Full Spec" on the LazyVim website has an example of how to set up the `nvim-telescope/telescope-fzf-native.nvim` extension, which looks something like this

```
  dependencies = {
    {
      "nvim-telescope/telescope-fzf-native.nvim",
      -- snipped some build instructions
      config = function()
```

```
        LazyVim.on_load("telescope.nvim", function()
          -- snipped loading the extension
        end)
      end,
    },
  },
```

This is showing us how to have a mini-config for a dependent plugin, which is *exactly* what we want. Further, if we customize our spec, `dependencies` is one of the tables that `Lazy.nvim` will *merge* with the parent spec (the one created by LazyVim). So we don't need to copy the above code into our telescope extension file; we only need to create a net new table.

If you have implemented all of the suggestions in this book, you might already have an `extend-telescope.lua` file that has some custom keybindings for deleting buffers. You can either edit that file or create a new one; LazyVim will merge them together either way. Personally, I'll be editing that file to add the following:

```
return {
  { "nvim-telescope/telescope-live-grep-args.nvim" },
  {
    "nvim-telescope/telescope.nvim",
    dependencies = {
      {
        "nvim-telescope/telescope-live-grep-args.nvim",
        config = function()
          LazyVim.on_load("telescope.nvim", function()
            require(
              "telescope"
            ).load_extension("live_grep_args")
          end)
        end,
      },
    },
  }
}
```

It's kind of verbose, but this should be all you need to enable the plugin.

Next, you need to set up a keybinding to call the plugin. You can choose to override the existing `<Space>/` keybinding, or perhaps you would use `<Space>?` if you want to have separate "default live_grep" and "live_grep_args" mode.

Your first attempt, if you are following the live-grep-args README might look like this:

```
-- This won't work
keys = {
  {
    "<leader>/",
    require(
      "telescope"
    ).extensions.live_grep_args.live_grep_args,
    desc = "Grep with Args",
  },
},
```

Unfortunately, this is too easy for LazyVim. :-(Because the `live_grep_args` plugin has been set up to run in a `LazyVim.on_load`, it is not defined at the time the keys array is created.

The solution is to wrap the call in another function, so the import only happens after you press the keybinding. That works because the `onLoad` handler will have been called by that point:

```
keys = {
  -- Other telescope-related keybindings
  {
    "<leader>/",
    function()
      require(
        "telescope"
      ).extensions.live_grep_args.live_grep_args()
    end,
    desc = "Grep with Args (root dir)",
  },
},
```

Before we move on to actually **using** the live-grep-args plugin, there is one more place you need to apply this weird nested function calls trick. Telescope-live-grep-args suggests hooking up `ctrl-k` to the `quote_prompt()` action, like so:

```
-- Don't do this
local lga_actions = require("telescope-live-grep-args.actions")
telescope.setup {
  extensions = {
```

```
    live_grep_args = {
      mappings = { -- extend mappings
        i = {
          ["<C-k>"] = lga_actions.quote_prompt(),
        },
      },
    }
  }
}
```

The table passed into `setup` there just comes from our `opts` array, but we again need to avoid importing `telescope-live-grep-args` at the top-level like that. Instead, we need a new function. But there are a couple gotchas:

- `quote_prompt()` is a function that returns a different function. So we need to invoke that function with a odd-looking `()()` syntax.
- Telescope mappings accept an integer argument (the internal id of the picker), so we need to forward that to the called function.

The resulting `opts` array looks like this:

```
opts = {
    extensions = {
      live_grep_args = {
        mappings = {
          i = {
            ["<C-k>"] = function(picker)
              require(
                "telescope-live-grep-args.actions"
              ).quote_prompt()(picker)
            end,
          },
        },
      },
    },
  }
```

For completeness (and because all the above snippets on their own may not make indentational sense), here is my entire Telescope configuration, including the mappings from Chapter 9 that set up `<A-d>` in the buffer picker:

```lua
return {
  { "nvim-telescope/telescope-live-grep-args.nvim" },
  {
    "nvim-telescope/telescope.nvim",
    dependencies = {
      {
        "nvim-telescope/telescope-live-grep-args.nvim",
        config = function()
          LazyVim.on_load("telescope.nvim", function()
            require(
              "telescope"
            ).load_extension("live_grep_args")
          end)
        end,
      },
    },
    keys = {
      {
        "<leader>/",
        function()
          require(
            "telescope"
          ).extensions.live_grep_args.live_grep_args()
        end,
        desc = "Grep with Args (root dir)",
      },
    },
    opts = {
      pickers = {
        buffers = {
          mappings = {
            i = {
              ["<A-d>"] = function(...)
                return require(
                  "telescope.actions"
                ).delete_buffer(...)
              end,
            },
            n = {
              ["<A-d>"] = function(...)
```

```
                return require(
                  "telescope.actions"
                ).delete_buffer(...)
              end,
            },
          },
        },
      },
      extensions = {
        live_grep_args = {
          mappings = {
            i = {
              ["<C-k>"] = function(picker)
                require(
                  "telescope-live-grep-args.actions"
                ).quote_prompt()(picker)
              end,
            },
          },
        },
      },
    },
  }
}
```

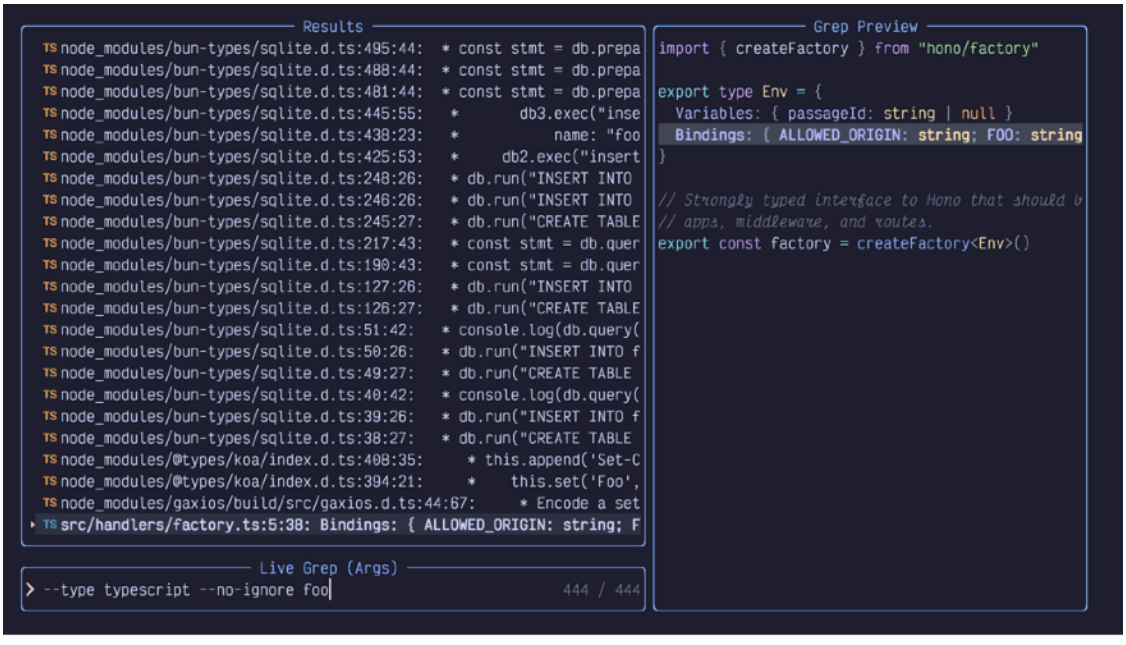
It's a mess, I know. Part of the mess is because Telescope is pretty generic, and that mess would still exist if you were managing your own config. But part of the mess is because we need to cooperate with LazyVim when we enter our customizations, and the config is necessarily more complicated than it would be. As usual, I'm ok with this because I have to do it rarely enough and I appreciate not having to manage *most* of my configuration by myself.

Using Telescope live grep args

Ok that was a pretty big digression, but hopefully you understand a bit better how to configure plugins in those cases where the simple abstractions over plugins that LazyVim provides are too simple.

Now that you have the `telescope-live-grep-args` configured, let's discuss a few tips on how to use it.

You can place any `-` or `--` prefixed args that you would pass to ripgrep (`man rg` on your command line will tell you what these are) at the beginning of your search string to forward those arguments to ripgrep. For example, to run the command `rg --type typescript --no-ignore foo` (which searches for `foo` only in typescript files including files that would otherwise be hidden by things like `.gitignore`), you could type the following into the Live Grep (Args) Telescope picker:



For me, ninety percent of the time when I want to customize `rg` args, it's just to specify a file type, so `--type` (or `-t` to be truly lazy) is all I need.

Often, you'll start typing a command and then realize the search is too broad (or too narrow) and you should have included some ripgrep arguments. It's certainly easy enough to type `<Esc>I` to move your cursor to the beginning of the line to insert them, but another option is to press `Ctrl-k`. If you've set up the plugin as described in the previous section, `Ctrl-k` will call `quote_prompt()`. All that does is convert `whatever the prompt was` to `"whatever the prompt was"`, with surrounding quotes and a space after it.

This is convenient, because if your prompt starts with a search string in quotes, it becomes the pattern, and you can add your other ripgrep arguments *after* the string:

Summary

This chapter was all about search: the `/` shortcut to enter search mode, and the `<Space>/` shortcut to enter find in project mode. The latter is not as powerful as it could be, so we also set up the `telescope-live-grrep-args` plugin to give us a little more control over the project-wide search.

Searching is, of course, only half the story. In the next chapter, we'll cover replacing text, both as part of a search operation and at a more project-wide level.

Chapter 13: ...and Replacing

Chapter 13: ...and Replacing - LazyVim for Ambitious Developers

Vim has a very powerful find and replace mechanism. It... takes some getting used to. On the one hand, it's pretty hard to go back after you've gotten used to the power of Vim substitution. On the other hand, getting used to it can take a lifetime. Substitution predates Vim and even Vi; it goes back to the legendary `ed` by Ken Thompson (who, among

numerous other things, wrote the original paper on regular expressions. It's an interesting read).

Substitution in `ed` was so powerful that it has somehow stuck around for over half a century. Not only is it the primary search and replace mechanism in modern Vim and NeoVim, it is also popular when automating tasks via shell scripting, using `sed` (the stream editor, a sequel to `ed`).

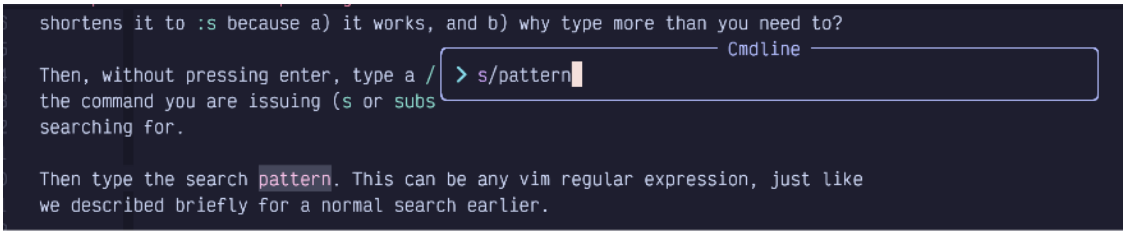
LazyVim, as usual, enhances the substitution command, mostly by showing you live previews of your changes as you type.

Because it is an `ex` command (ex stands for "extended ed", much like its sibling `vi` was latter rewritten as "vi improved"), you access substitution by entering command mode (with a `:`). You could type `:substitute`, but everybody shortens it to `:s` because a) it works, and b) why type more than you need to?

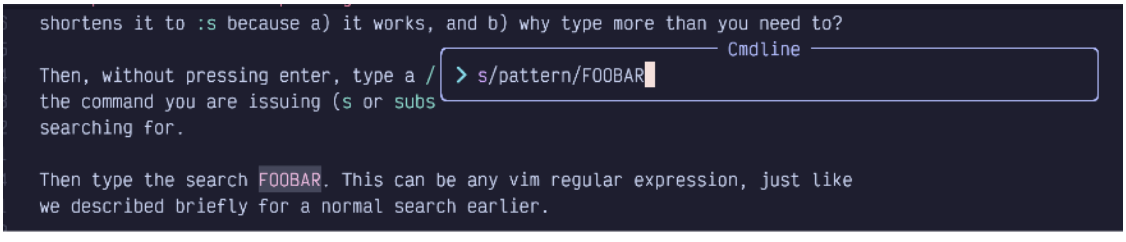
Then, without pressing enter, type a `/`. This is just a separator to separate the command you are issuing (`s` or `substitute`) from the term you are searching for.

Then type the search pattern. This can be any vim regular expression, just like we described briefly for a normal search earlier.

Here you can see that I have typed `:s/pattern` into my editor, and the `pattern` is highlighted on the line that my cursor was on:



Next, type another `/` to separate the *pattern* from the *replacement*, and then type whatever string you want to replace it with. LazyVim will live update all instances of the search term with the replacement term so you can preview what it will look like. Here, I'm going to replace `pattern` with `FOOBAR`:



Now press `Enter` to complete the command and confirm the replacement. So find and replace is as simple as `:s/pattern/replacement<Enter>`. That's not so bad, is it?

Maybe it's not, but we're not done. Not remotely. For one thing, that command will only replace the first instance of `pattern`, and only if the pattern happens to be on the same line as the cursor.

Substitute ranges

Many NeoVim ex commands can be preceded by a range of lines that the command will operate on. The syntax for ranges can be a little confusing, and to this day I still have to look it up with `:help range` if I'm doing anything non-standard.

The simplest possible range is the `.`, which stands for "current line". It would look like `:.s/patttern/replacement`. The `.` between the `:` and the `s` is the range, in this case. You normally wouldn't bother, though because `.` or current line is the default range.

Probably the second most common range you will use is `%`. It stands for "Entire File". If you are used to the find and replace dialog in most editors or word processors, you probably expected it to mean "entire file." But it's not, and if you want to do a find and replace across the entire file, you would need to use `:%s/pattern/replacement` (probably with a `/g` on the end as described in the next section).

You could also set a specific line number, such as `:5s/pattern/replacement` to replace the word `pattern` on line 5. But personally, I would use `5G` to move my cursor to line five and then do a default range substitution instead.

The name "range" implies that you would normally cover a sequence of multiple lines, and you can indeed separate a start and end position using a comma. So, for example, `:3,8s/...` will perform the substitution on lines 3, 4, 5, 6, 7, and 8 (the selection is inclusive at both ends): Here I've started a pattern that is highlighting the word `hello` on lines 3 through 8, but no other lines:

You can also use marks such as `'a` as described in the previous chapter to define the start or end of a range.

The most common way you'll use this is using `'<,'>`, which specifies the range for "the most recent visual selection. Luckily, you won't need to type those characters all that often, because if you select some text using e.g. `Shift-V` followed by a cursor movement, and then type `:`, Neovim will automatically take care of copying that range into the command line.

This means that if you want to "perform a substitution in the current visually selected text," you just have to select the text and type `:s/...`. The range will be inserted between the colon and `s`, so you'll get `:'<,'>s/...`.

If your brain is up for some recursive confusion, you can even use a search pattern to specify one end of the range! In the following example, my cursor was on line 5 when I started the substitution:

The substitution is `:,/hello-10/s/hello/foo`. All those forward slashes in there make it pretty hard to read (looks like a Unix file path!), but it's actually easy to write. Let's break it down from left to right:

- `:` is the normal mode "start a command" trigger.
- There is nothing between the `:` and the `,` so the start of the range is the current line (line 5 in this example).
- The first `/` is a ever-so-slightly more succinct way of saying "the end of the range is the first line after the current cursor position that matches some pattern".
- `hello-10` is the pattern we are searching for to define the end of the range.
- The second `/` marks the end of the pattern. So our full range is `,/hello-10/` and means "from the current line to the line containing `hello-10`."
- The `s` indicates we want to perform a substitution on the lines in that range.
- `/hello/foo` is the pattern "hello" and replacement "foo", like any substitution.

There is a ton of other stuff you can do with Vim ranges, but the truth is, most of them only exist to support outdated editing modes. You will likely find that `%`, `'<,'>`, and `,/pattern/` cover 95% of your use cases. Read through `:help range` once to make sure you know what other sorts of syntaxes are available, and don't be afraid to look them up in the rare instances that one of the above is not sufficient.

Flags (Global and ignore case substitutions)

Did you think we were done with substitutions? Sorry! I'm trying to condense this section to include only that which is actually useful in 2024, but substitutions are just so darn powerful that it's getting away from me.

You can add "flags" at the end of any substitution (after the last `/`) to modify how the search and replace behaves. The most common flag you'll use is `g` which stands for "global". You'll append it more often than not.

By default, `substitute` only replaces the first instance of a pattern on the line. So if I have a file full of the overly cheerful words `hello hello`, then the substitution `:%s/hello/foo` will only replace the first instance on each line:

But if I append `/g` it will replace all the `hello`'s on each line:

I mentioned earlier that the supremely common use case of "replace everything in the file" is `:%s/pattern/replacement/g`. The `%` is "every line", and the `g` means "every instance in each line".

There are almost a dozen flags, but the only other useful ones are `i`, `I`, and (rarely) `c`. The first two explicitly ignore case or disable ignoring case in the term being searched for, and you'll only ever need one or the other depending on whether you have `ignorecase` set in your `options.lua` (it defaults to true in LazyVim). The `c` flag means confirm and is useful if you want to make substitutions in a large file but you know you want to skip some of them; you will be shown each proposed change and accept or reject them one at a time.

Flags can be combined, so `:%s/hello/foo/gc` will do a global replace, confirming each one.

Handy Substitute shortcuts

You don't need to memorize this section, but once you get used to substituting, you'll probably notice that some actions are rather repetitive and monotonous and you'd like them to go faster. Read through these tips so you remember to look them up when you are more comfortable with `:substitute`.

If you leave the pattern part of a substitution blank, (as in `:s//replacement/`), it will default to whatever pattern you last searched for *or* substituted. For example, if you perform these commands in order:

- `/foo` will search for the word `foo`
- `:s//bar` will replace `foo` with `bar`
- `:s/baz/bar` will replace `baz` with `bar`
- `:s//fizz` will now replace `baz` with `fizz`

This can save a little typing when you search for a term and then decide you want to replace it, or when you have substituted something in one file and want to substitute it again in another.

If you just use `:s` without any pattern or replacement, it will repeat the last pattern and replacement you did. But be aware that it will not act on the same range, so if you want to repeat it exactly you'll need to type the range again.

It also won't repeat flags, but you can (usually) append the flags directly to `:s`. For beginners, the most common of these is `:%sg`, which maps to "repeat the last substitution on the entire file, globally." This is helpful when you typed `:s/some-long-pattern/some-longe-replacement` and expected it to do a global replace, but actually it just replaces the first instance on the current line. `:%sg` will repeat the substitution the way you intended it. You might also reach for `'<,'>sg` to replace in the last visual selection.

> Don't forget that you can repeat the last visual selection with `gv` to confirm that it is actually selecting what you thought it does.

If you want to reuse whatever was matched in the pattern in the replacement, you can use `\0` in the replacement string. This is particularly useful when you are using a regular expression that could potentially match different things.

For example, imagine I have the following file:

```
hello world
Hello thrift shop
Hellish world
```

For some reason, I want to add an adjective between the first and second words. This can be accomplished with the command `:%s/[hH]ell\S* /\0green /`:

That command might be a little intimidating if you aren't comfortable with regular expressions, so I'll break it down again;

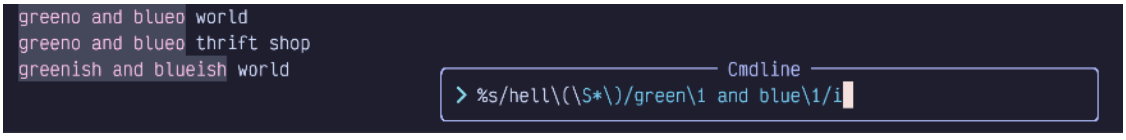
- `:%` means "perform a command on the entire file"
- `s/` means "the command to perform is substitute"
- `[hH]` means "match h case insensitively (see note)
- `ell` means "match the three characters `ell` exactly"
- `\S` means "match any non-whitespace character"

- `*` means "repeat the `\S` match zero or more times", which takes us to the end of the word.
- `/` includes a space and then the end of the search pattern
- `\0` says "insert whatever was matched by the above pattern into the replacement"
- `green` says "insert that text directly into the replacement"

> Note on `[hH]`: this isn't necessary if you don't have `vim.opt.ignorecase=false` in your `options.lua`. An alternative would be to use `/i` at the end of the pattern to force ignoring case for this one search. Then `[hH]` could just be `h`.

You can even reuse **part** of the pattern in the replacement. To do this, place the part you want to reuse in `\(` and `\)`. Then use `\1` to represent whatever was matched between brackets in the replacement portion.

This is easier to understand with an example. If we start with the same three line example as above, we can use the substitution `:%s/hell\(\S*\)/green\1 and blue\1/i` to cause the following nonsense substitution:



The `\(\S*\)` matches the same thing as `\S*` but it stores the result in a *capture*. Then when we want to reuse the capture in the replacement, we use `\1` to refer *back* to whatever was captured on that match.

You might guess from the fact that we're using numbers here that you can have and refer back to multiple captures, and your guess would be correct!

Project-wide search and replace with Spectre

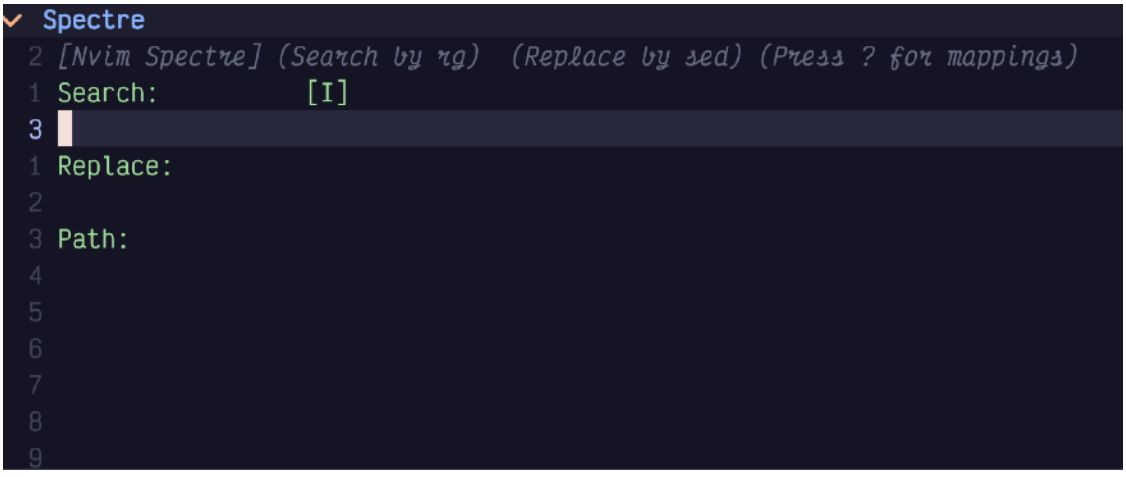
LazyVim ships with a plugin called Spectre to do a global find and replace. In the old days, you would probably do this from the command line using `sed`, the stream-oriented evolution of `ed` that I mentioned. And if you're anything like me, you wouldn't enjoy it.

Spectre does a global find and replace in all files in the current project. Before we see how to use it, I should warn you (as the Specte README does) to commit your files to version control before running Spectre because the work it does is irreversible. You can't undo it, so make sure `git reset --hard` won't cause you to lose any work that wasn't done by Spectre.

Spectre is really just a lightweight UI wrapping `sed` and `ripgrep`, two command line tools we've mentioned before. But that UI is pretty handy, as both tools have some arcane arguments (though `sed` is similar to the `:s` behaviour we just learned).

> **Note:** MacOS ships with a less-robust version of `sed` and you should probably run `brew install gnu-sed` for Spectre to work properly.

To show the Spectre UI, use the keyborad shortcut `<Space>sr`, where the mnemonic is `r` for `r`eplace. The window that pops up is pretty bare-bones:

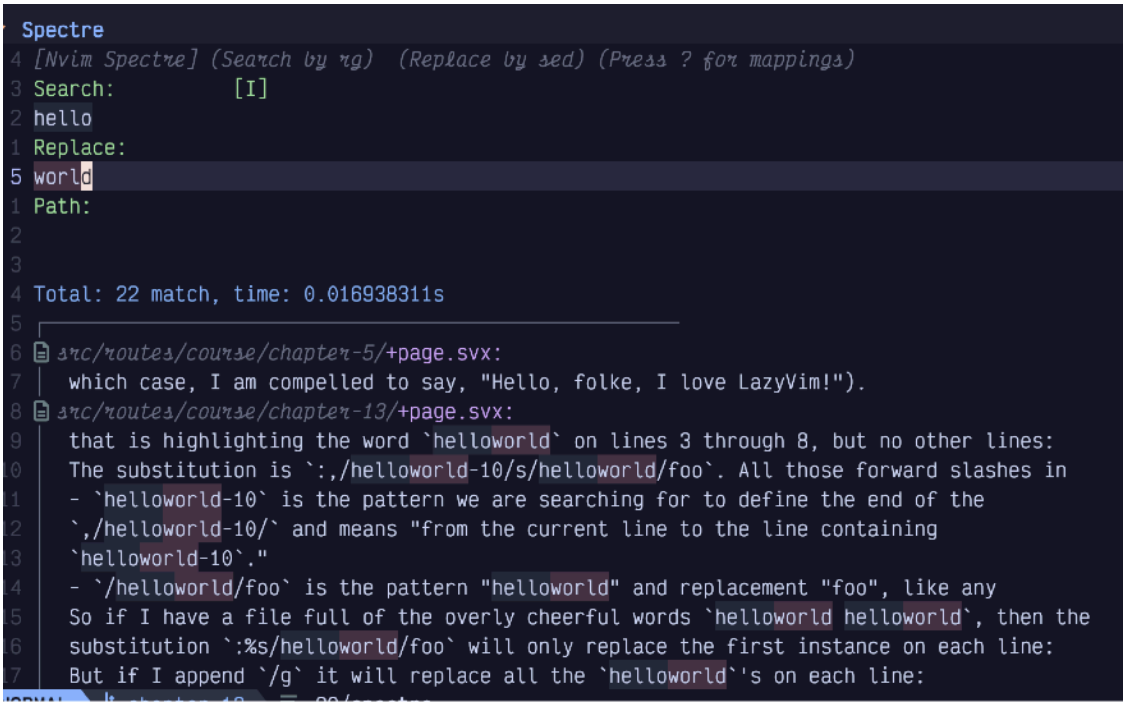


You can navigate around this window using all the normal vim motions, but you'll mostly just want to use `Tab` (in normal mode) to jump between the three fields.

The search field can accept any (vim-style) regular expression, and the replace field can support `\0` and friends so it should start to feel familiar (at least, insofar as `:s` can feel familiar!)

The path field is used to isolate your search to a specific subfolder of the current directory (e.g. `src/` might be a useful path). However Spectre will perform its job faster if you `:cd` or use `NeoTree` or `mini.files` to change to the subdirectory directly before running spectre.

After you have inserted the search and replace text, you will need to press… (this is unintuitive, unless you are used to vim) `Escape` to return to normal mode. This will pop up a pretty nifty preview area that shows all the changes it will make:



Navigate the preview window using whatever motions feel appropriate. If one of the results is something you don't actually want to replace (e.g. a package lockfile), use the `dd` command to remove it from the replace action.

When you are ready to perform the underlying command, use `<Space>R` to perform the replacement action. You can also use `<Space>rc` to replace just one of them. Even though these look like space mode commands, they are only available when the Spectre window is active.

There are a few other useful keybindings in a menu you can pop up with `?`, which I'll leave you to peruse at your leisure.

Perform vim commands on multiple lines

The `:substitute` isn't the only one that can operate on multiple lines at once, with a range. In fact, if you just want to write a few lines out to a separate file, you can pass a range to `:write`. The easiest way to do this is to select the range in visual mode and type `:write`. Neovim will automatically convert it to `:'<,'>write` and only save only those lines.

A note on multiple cursors

NeoVim doesn't have first class multi-cursor support (yet). Historically, Vim coders have considered multi-cursor mode to be a crutch required by less powerful editors that don't have Vim's command modes. More recently, experimental editors such as Kakoune and Helix have demonstrated that multiple cursors can integrate very well with modal editing. Modern developers like multiple selections, and NeoVim is expected to ship with native multiple cursor support in the future (It's currently listed as 0.12+ on the roadmap).

In the meantime, there *are* multiple cursor plugins, but I find them to be clumsy and fragile, and recommend avoiding them at this time. Instead, you can use the commands discussed below or rely on other Vim tools such as repeating recordings (with `q` `Q`, and `@@`), or visual block mode (`Ctrl-v`) with an insert or append that modifies multiple lines.

The `:norm` command

When you first use it, `:norm` feels pretty weird. It allows you to perform a sequence of arbitrary vim normal-mode commands (including navigation commands such as `hjkl` and `web` as well as modification commands like `d`, `c`, and `y`) across multiple lines.

You can even enter insert mode from `:norm`! But you need to know a small secret to get out of insert mode because pressing `<Escape>` while the command menu is visible will just close the command menu. Instead, use `ctrl-v<Escape>`. When you are in insert mode or the command line, the `ctrl-v` keybinding says "insert the next keypress literally instead of interpreting it as a command." The terminal usually renders `ctrl-v<Escape>` as `^[`.

For example imagine we are editing the following file:

```
foo
Bar
fizz buzz
one two three
```

For inexplicable (but pedagogical) reasons, we want to perform the following on each and every line:

- insert the word "HELLO" at the beginning of the line with a space after it
- capitalize the first letter of the first word on the line
- insert the word "BEAUTIFUL" after the first word on each line with spaces surrounding it
- append the word "WORLD" to the end of each line with a space before it

Start by typing `:%norm` to open a command line with a range that operates on every line in the file (`%`) and the `norm` command followed by a space.

Then add `IHELLO` to insert the text `HELLO` at the beginning of each line in the range. Now hit `ctrl-v` and then `Escape` to insert the escape character into the command line. This will return the command to normal mode when it runs.

Now type `lgUl` to move the cursor right (which puts it on the beginning of the first word), then uppercase one character to the right (i.e. the first character of the next word).

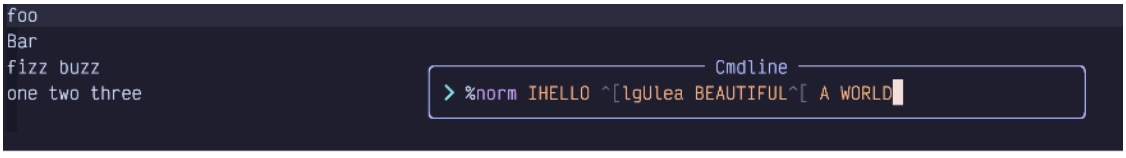
Next is `e` to jump to the end of the word, followed by `a BEAUTIFUL` to append some text after that word. `ctrl-v` and `Escape` will insert another escape character.

Finally, add `A WORLD` to enter append mode at the end of the line and add the text `WORLD`.

The entire command would therefore be:

```
:%norm IHELLO <ctrl-v Escape>lgUlea BEAUTIFUL<ctrl-v Escape> A WORLD
```

Visually, it looks like this, since the `ctrl-v Escape` keypresses get changed to `^[`:



And the end result:

```
HELLO Foo BEAUTIFUL
HELLO Bar BEAUTIFUL WORLD WORLD
HELLO Fizz BEAUTIFUL buzz WORLD
HELLO One BEAUTIFUL two three WORLD
```

Admittedly, it's unclear why you'd want to perform this exact set of actions, but it hopefully shows that anything is possible!

It's pretty common to get the command wrong the first time you try to apply it. Simply use `u` to undo the entire sequence in one go, then type `:<Up>` to edit the command line again.

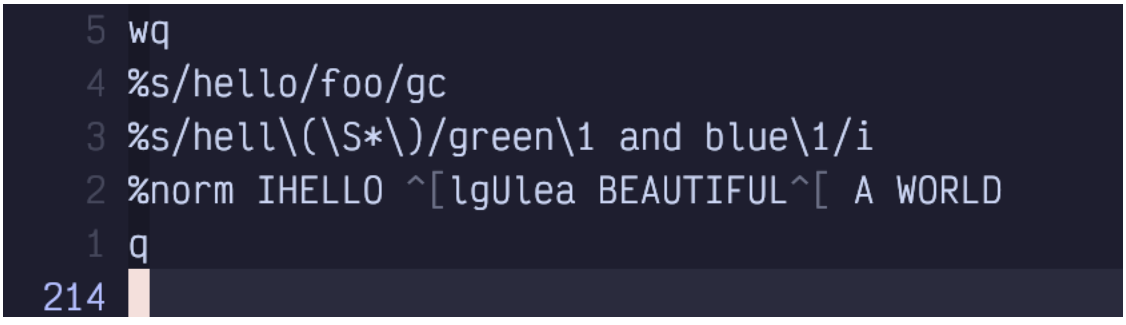
If the command is kind of complicated, you'll probably get annoyed while editing it because you don't have access to all the vim navigation commands you are used to. So now is a great time to introduce Vim's command line editor.

Command Line Editor

To display the command line editor, type `Control-F` while the little `Cmdline` window is focused. Or, if you are currently in normal mode, type `q:`. This latter is not related to the "record to register" command typically associated with `q`. It is instead "Open the editable command line window".

This window is basically what happens when the normal command line editor marries a normal vim window and spawns a magical superpower command line window.

The new magic window shows up at the bottom of the current buffer, just above the status bar, and it contains your entire command line history (including searches and substitutions):



Use `ctrl-u` to scroll up this baby and you'll see every command you ever typed. You can even search it with `?` (search backward is probably more useful than search forward since your commands are ordered by recency).

To run any of those old commands, just navigate your cursor to that line and press `Enter`. Boom! History repeats itself.

Or you can enter a brand new command on the blank line at the bottom of this magic command window (Remember `Shift-G` will get you to the bottom in a hurry).

You will find this window is devilishly hard to escape, though. The escape key doesn't work, because it's reserved for escaping to normal mode while editing the window. The secret is to use `Control-C` close it, although other window close commands such as `Control-w q` (or `\q` if you have set up your keybindings as I suggested in Chapter 9) will also work. You can even run `:q` from inside the command line window.

Most importantly, you can use normal vim commands to *edit* any line in this window. Just navigate to the line, use whatever mad editing skills you have (including other command-

mode commands such as `:s`) to make the line look the way you want it to, return to normal mode, and press `Enter`. The edited command will execute.

I have to confess, I had been using Vim for over a decade before I discovered what this window was for (though I'd gotten myself into it by accident and had trouble getting out more than a few times…). And it instantly became my favourite window. `q:` (or `:Control-f`). It's magical!

Mixing :norm with recording

Recall that the `q` command can record a sequence of commands to a register for later playback. There are several ways you can later apply this recording to a range of lines.

- `:<range>norm @q` will simply execute the `q` register on the entire range, since `@q` is the command to execute register `q`.
- `:<range>norm <ctrl-r>q` will copy the contents of register `q` into the cmdline window.
- `q:<range>inorm <Esc>"qp` will open the command line editor window, insert the word `norm` and copy the contents of register `q` into the line using the normal mode register paste command.

The global command

The `:norm` command operates on a range of lines, and NeoVim ranges must be contiguous lines. It's not possible to execute a command on e.g. lines 1 to 4 and 8 to 10, but not 5 to 7 (other than running :norm twice on different ranges).

Sometimes, you want to run a command on every line that matches a pattern. This is where the `:global` command comes in.

The syntax for `:global` is essentially `:<range>global/pattern/command`, although you can shorten it to `:<range>g/pattern/command`. The pattern is just like any vim search or substitute pattern.

The `command`, however, is kind of weird. Technically, it's an "ex" command, which means "many but not all of the commands that come after a colon, but mostly ones you don't use in daily editing so they are hard to remember".

The most common example is "delete all lines that match a pattern", which you can do with `:%g/pattern/d`.

Another popular one is `substitute`, which you already know. If you precede your substitute with `:%g/pattern`, you can make it only perform the substitution on lines that

match a certain pattern. This pattern can be *different* from the one that is used in the substitution itself. Consider the following arcane sequence of text:

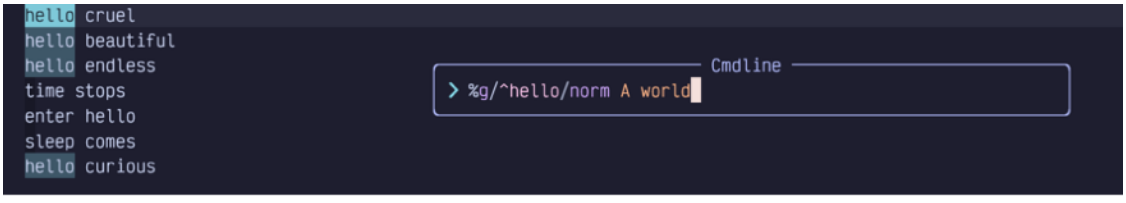
```
:%g/^f/s/ba[rt]/glib
```

What a mess! This is obviously meant to be easy to write, not easy to read. If we wanted it to be slightly easier to read, we'd probably write `:%global/^f/substitue/ba[rt]/glib`.

This command means "perform a global operation on every line that starts with `f`. The operation in this case should be to replace every instance of `bar` or `bat` with the word `glib`."

In my opinion, though, the most interesting use of `:global` is to run a normal mode command on the lines that match a pattern. This effectively means mixing `:global` with `:normal`, as in `:%g/pattern/norm <some keystrokes>`.

As just one example, this will insert the word "world" at the end of every line that starts with "hello":



You can also use global to perform a command on every line that **does not** match a pattern. Just use `g!/` instead of `g/` This is useful, for example, in log files that have exceptions wrapping onto random lines. For example, a rudimentary log file might look like this:

```
2024-03-26T12:00:00 Something happened
2024-03-26T12:01:01 Something happened
2024-03-26T12:01:02 Something super bad happened
  Traceback:
    A bunch of lines I don't care about
2024-03-26T12:02:00 Something else happened
2024-03-26T12:03:58 Cool thing happened
```

and prior to further processing, I might want to remove every line that doesn't start with a date:

That might be a bit eye-watering. Each `\d` means "match a digit", while the final `/d` means "perform a delete operation on the selected lines". The `g!` is the important part; that's the one that means "the selected lines are ones that don't match the pattern".

I don't use `:global` nearly as often as I use `:norm`. But when I do, it is a hyper-efficient way to cause massive changes in a file. It takes some getting used to, and you'll probably be looking up the syntax the first few times you need it, but it's a really terrific tool to have in your toolbox.

Summary

This chapter was all about bulk editing text. We started with substitutions using the `:s[ubstitute]` ex command, and then took a tour of the UI for performing find and replace across multiple files using the Spectre plugin.

Then we learned how to perform commands on multiple lines at once using `:norm` and `:global`, and earned a quick but comprehensive introduction to the command line editing window.

In the next chapter, we'll learn several random editing tips that I couldn't fit anywhere else.

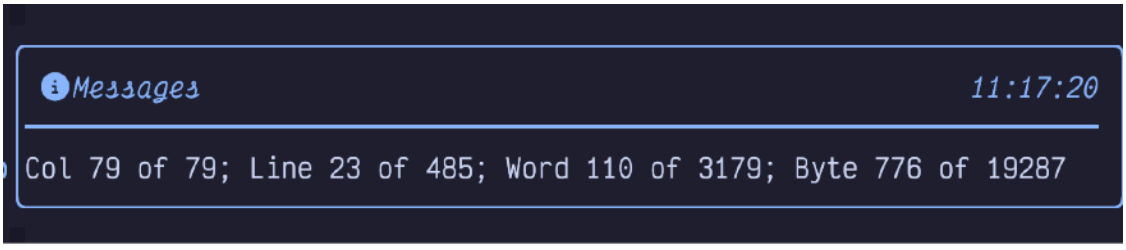
Chapter 14: Miscellaneous Editing Tips

Chapter 14: Miscellaneous Editing Tips - LazyVim for Ambitious Developers

Before we dive into some of the more "IDE-like" behaviours that LazyVim enables, I wanted to collect some tips that can make your editing life a little more fun. This chapter is a bit of a grab bag, and includes some commands and plugins that didn't fit anywhere else.

Word counts

Use `g<Control-g>` to spit out a message containing some helpful info about the current cursor position:



Most notably, the "Word 110 of 3179" tells me that this chapter has over 3000 words in it (I mean, *obviously* I updated this section after I wrote more words!)

Transposed characters

How often do you type so fast that you accidentally transpose two chracters?

Simply use `xp` to swap a character with the one to the right of it. For example, if you have typed `ra` when you meant to type `ar`, put your cursor on the `r` and hit `xp`.

This is not a special custom command. It just leverages the default "delete character" and "put last deleted after the cursor" commands to move the character from its current position to the next one. You can use a similar idea to move other text around. For example, move a word with `dwwP` or use `daaWp` to delete an argument and move it later in a list of arguments.

Commenting and uncommenting code

I used Vim's somewhat clumsy blockwise column mode to add or remove # or `//` characters at the beginning of each line for way too long before I learned there were plugins for commenting code. As of Neovim 0.10, this is actually shipped natively with Neovim, but LazyVim still includes a plugin if you are on an older Neovim version.

The verb for toggling comments is `gc` and can be followed by a motion or text object. So `gc5j` will comment this line and the five lines below it, while `gcap` will comment out an entire block separated by newlines.

This command pairs beautifully with the `s` command to comment out a surrounding text object. For example `gcsh` will comment out the function surrounded by the `h` labels after the `s` is invoked.

To comment out a single line, use the easy-to-type shortcut `gcc`. This command can take a count, so `5gcc` will comment out five lines (a little easier to type than `gc4j`).

As with most verbs, `gc` can also be applied to a visual selection with e.g. `V5jgc`.

The `gc` verb is actually a toggle, so if a line is currently commented, it will uncomment it instead of commenting it a second time. Thus, `gccgcc` is a no-op. However, note that if you have a selection that contains commented and uncommented lines, you will end up with a double comment. This is usually what you want: If you temporarily comment out a block that contains other comments, when you uncomment that block, you probably want the comments to stay commented.

Incrementing and decrementing numbers

If your cursor is currently on a number in normal mode, you can use `ctrl-a` to increment that number. This command is somewhat smart and does the "right thing" if your number needs new digits. So `9` becomes `10` and `99` becomes `100` when you press `ctrl-a` anywhere in the number.

To decrement a number, use `ctrl-x`. I hated these two keybindings for the longest time because they are only occasionally useful, but when they are useful, I couldn't remember them. So I spent a long time manually incrementing numbers and thinking to myself "I need to look up those number increment numbers," but the only keywords associated with this help section were the keybindings themselves!

Eventually I learned about the `:helpgrep` command, which allows you to search the help. Long before I memorized the keybindings, I remembered that `:helpgrep Adding and subtracting` would help me look them up.

But there is actually a mnemonic for these keybindings: `ctrl-a` is "Add", which is easy enough to remember. `ctrl-x` is a little harder, but now that you have `ctrl-a` you'll be able to look it up with :help `ctrl-a` ;-). I'm not sure if it will help anyone else, but I've learned to think of the `x` as "'cross' out one digit to subtract".

Use `g<ctrl-a>` and `g<ctrl-x>` to decrement numbers on consecutive lines with an additional count for each line. This is useful if you are manipulating numbered lists. Say you want to make a list of 10 items. First type `oi1.<esc>` to make a line that says `1.`. Then type `9.` to repeat that command 9 times. Now you have:

```
1.
1.
1.
1.
```

```
1.
1.
1.
1.
1.
1.
```

You can use `V'[` to select the 9 rows that just got inserted, as the `'[` mark is the first character of the previously changed text. Now type `g<ctrl-a>` to increment them and you end up with:

```
1.
2.
3.
4.
5.
6.
7.
8.
9.
10.
```

Not bad for just a handful of admittedly bizarre keystrokes: `oi1.<Esc>9.V'[g<ctrl-a>`!

If you need to insert a new entry in the middle of a list, add the entry, select the lines with the remaining entries, and hit `ctrl-a` to sync them up.

Neovim will smartly just increment the first number it encounters on a line. This means it is easy to for example, manipulate a book's outline even if it contains multiple numbers. Consider this hypothetical outline of a book not unlike this one:

```
Chapter 1: Intro and Install
Chapter 2: 1 Weird modal editing trick
Chapter 3: The numbered marks 1-9
Chapter 4: Navigating things
...
```

Let's say I want to split Chapter 1 into two different chapters: "Intro" and "Install". I can simply add the new chapter using normal text insertion like this:

```
Chapter 1: Intro
Chapter 2: Install
Chapter 2: 1 Weird modal editing trick
```

```
Chapter 3: The numbered marks 1-9
Chapter 4: Navigating things
...
```

Then I can use <Shift-V> to select the chapters originally numbered 2 and higher. Then I use a navigation keystroke (such as `s` or `}`) to jump to the next paragraph. When I hit `ctrl-a`, the chapter numbers are incremented, but the `1` in `1 Weird trick` will not be impacted, nor will the numbered marks indicators.

```
Chapter 1: Intro
Chapter 2: Install
Chapter 3: 1 Weird modal editing trick
Chapter 4: The numbered marks 1-9
Chapter 5: Navigating things
...
```

The dial.nvim extra

If the increment and decrement keybindings sound kind of like that one weird kitchen unitasker that is helpful once a month, you might want to consider installing the `editor.dial` extra from `:LazyExtras`.

This extra comes with the `dial.nvim` plugin which allows you to increment and decrement a bunch of other cool stuff. I mostly use it to swap boolean expressions (both `ctrl-a` and `ctrl-x` will alternate `true` to `false` and vice versa.), but it can also increment words ("first" increments "second"), months ("December" increments to "January"), version numbers, Markdown headers, and more. More importantly, you can extend it with your own patterns if you need to.

Changing indentation

The > and < keybindings can be used in normal mode to indent or dedent text. Most often, you'll use them doubled up (as in << and >>) to change the indentation of the current line. However, you can also change the indentation of any motion. Another common one is `>S<label>` to indent a block of text, and `>ap` will indent an entire blanks-delimited paragraph.

These verbs can get a little confusing when it comes to using counts. You might expect `2>>` to indent the current line by two indentation levels, but instead, it will indent two lines by one indentation level.

If you want to change by multiple indents in one command, you will need to resort to visual mode. To indent the current line by five indentation widths, the quickest way is with `v5>`,

compared to typing ten greater-than symbols. This works with any visual selection, so you can use, for example, `va{5>` to indent an entire block five levels.

Often, all you want to do is "make the indentation correct for this programming language". If `conform.nvim` is configured correctly, the easiest way to do this is to just save the file. LazyVim has format on save enabled by default, and if it can find a formatter, it will use it. You can also use `gq` with a motion or selection (most commonly `gqag` to format the entire file) to apply formatting.

However, if you don't want to save, or aren't using conform, you can also use the = verb. The behaviour of = depends a little on the programming language, but it generally applies the indentation engine to the visually selection (or motion selected) lines as though you had pressed `enter` to start a new line. The end result is that all lines will be indented "correctly" for some definition of "correctly". If you're lucky, it will be the same as your definition!

Reflowing text

I've used the `gw` command a lot while writing this book. It effectively rewraps (`w` for wrap) all the text at the eighty character limit (or any ruler number, configurable with `:set textwidth=<number>`), without breaking words.

Most often, I use `gww` to rewrap the current line so that it has linebreaks at the appropriate position or `gwip` to rewrap an entire paragraph. But `gw` works with any motion or visually selection. To rewrap an entire file, use `gwig`.

This command relies heavily on the existence of newlines. Effectively any two consecutive lines will be joined into a single line (if they fit in 80 characters). For me, this has meant that if I forget to put a newline after a heading, my first paragraph gets tied up into the heading, which is obviously not what I want.

Filtering through external programs

You can also pipe text out to any external program that is a good Unix citizen: processing input on STDIN and outputting it to STDOUT. To do so, visually select the text you want to pipe in visual mode. Then type a `!`. This will open the command window with the visual selection, and is a shortcut for `:'<,'>!`. Then type a command on the path and the selected text will be replaced with the output of that command.

Here are some examples:

- `!grep -v a` will replace the selection with the same text, but any lines that contain the letter "a" will be removed.

- `!tr -s ' '` will call the translate command, replacing all instances of multiple spaces with a single space.
- `!jq` will format the `json` text with `jq`
- `!pandoc -f markdown -t html` is a handy way to quickly write html by starting with simpler markdown syntax.
- `!./my-custom-script` will pipe the command through an arbitrary script you wrote.
- `!python ./something.py` will pipe the command through a python script you wrote.

> If you want to run a command without modifying the text, don't supply a range. For example, `:!mkdir foo` will run the `mkdir` command without overwriting your file content.

I think it is unfortunate that this feature is not used more. Many features that are built into Neovim or supplied as plugins could just as easily be CLI programs that operate on piped input and output. As just one example, the `:sort` command that ships with Neovim is, in my opinion, just bloating the editor when `!sort` can run the external sort utility just as well.

Abbreviations (and filetype configuration)

Vim abbreviations have been around since the earliest days of the editor. They are an easy way to have "shortcut" words that expand to something else entirely without leaving insert mode.

To create a temporary abbreviation, just use the command `:iabbr <shortcut> <expansion>`. You can use Vim's keybinding syntax to represent special characters like `<Enter>`, and `<Tab>`. You can even use `<Left>` to reposition the cursor within the abbreviated text.

For example, the command `:iabbr ifmain if __name__ == "__main__":<Enter>main()<Left>` will expand the text `ifmain<Space>` to the following, and place the cursor inside the parentheses after `main`:

```
if __name__ == "__main__":
    main( )
```

The `i` in `iabbr` means it will work in insert mode, and `abbr` is short for "abbreviate."

Note that I didn't have to explicitly add any indentation after the `Enter` because the python indentation engine takes care of that for me. Note also that the `<Space>` I typed after

`ifmain` was inserted between the brackets. If you need to expand an abbreviation without adding spaces, use the `Control-]` keybinding instead.

And if you need to insert the words `ifmain` without expanding them, type `ifmain<Escape>` to return to normal mode without expanding.

This abbreviation will only exist until I close the editor. To make it permanent, I need to add it to my configuration. Typically, abbreviations only make sense within the context of a single filetype, so I collect mine in the `autocmds.lua` using syntax like this:

```
vim.api.nvim_create_autocmd("FileType", {
  pattern = { "python" },
  callback = function()
    vim.cmd('iabbr ifmain if __name__ ==
"__main__":<Enter>main()<Left>')
    vim.cmd("iabbr frang for i in range():<Enter><Esc>F(i")
    -- Other Python abbreviations
  end,
})
```

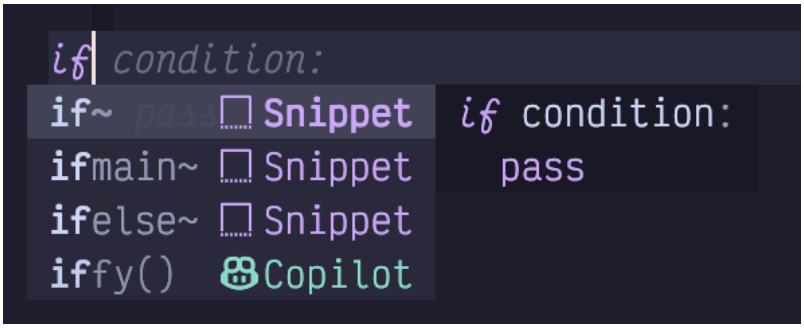
The `frang` abbreviation shows another neat trick: You can use the string `<Esc>` to enter normal mode and move the cursor. I used `F(` to "find the previous open paren" followed by `i` to enter insert mode inside the `range()` parens.

Vim abbreviations have been around forever and do the job well. I still use them probably because I am old, but the world has largely moved on to snippets instead.

Snippets

LazyVim ships with the `nvim-snippets` plugin, an interface to Neovim 0.10's built-in snippets interface. It can load VS Code-style snippets. I'm not going to go into detail of how to configure all that , but I do want to discuss some of the nuanced details of snippets.

`nvim-snippets` integrates with `nvim-cmp`, which we've touched on before, for completions. By default, `nvim-cmp` pops up a pretty menu with a bunch of completions as you type. For example, here's what I see if I type `if` in a Python file:

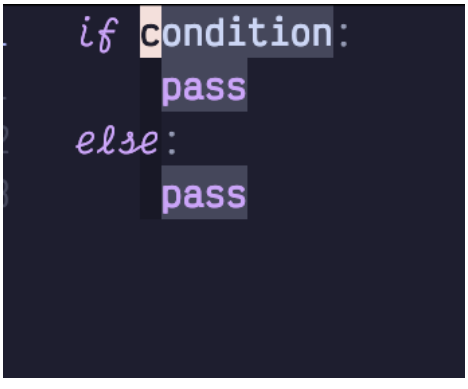


The left column shows possible completions in a neutral text colour. The middle column indicates two (in this case) different completion sources: Snippets or the Copilot AI engine (this isn't enabled by default, but there's an extra for it that we'll discuss in chapter 16). If I was working in a real project, it would likely also include some functions and classes that I've defined.

I can move my cursor up and down the list with the *arrow* keys (`j` and `k` won't work here because I'm still in insert mode). Most completions have a preview box pop up with documentation or an example of the completion.

This snippet was created by the `FriendlySnippets` plugin, which is a massive collection of useful snippets that ships with LazyVim by default. (Also notice that there is also a `ifmain` snippet much like the abbreviation I apparently didn't actually need to define above!)

If I then press the `Control-y` key, which confirms a completion (or `Enter` if you use the LazyVim defaults or `Right Arrow` if you have configured nvim-cmp the way I have), the snippet is inserted my editor:



The editor is currently in "Select" mode, an uncommon mode that is similar (but not really) to Visual mode. In LazyVim's default config, I'm not aware of any way to get into select

mode other than accepting a snippet! So we won't go into too much detail about this mode outside the context of snippets.

The key point is that "condition" is currently highlighted, and I can start typing immediately to overwrite it, almost as though I was in Insert mode. Once the condition has been replaced, I can press the `<Tab>` key, which the `nvim-snippets` plugin interprets as "jump to the next field in the snippet." Now the `pass` inside the `if` is highlighted instead.

The `<Tab>` key only works like this if `nvim-snippets` is aware it is in a snippet that has fields.

Defining new snippets

If the `FriendlySnippets` snippets aren't enough for you, you can define your own snippets using the ubiquitous VS Code Snippet syntax and load them in nvim-snippets. As a quick example, here's how to create a snippet for a boilerplate Svelte component using the lua snippet syntax:

1. If it doesn't exist, create the `~/.config/nvim/snippets/` directory to hold your snippets. This is the default location `nvim-snippets` will look for snippets.

2. Create a subdirectory in that directory for the filetype you want to create a snippet for. You can discover the Neovim filetype of the currently open file with the `:set ft` command. In this case, we'll create `~/.config/nvim/snippets/svelte/`.

3. Create a json file in the `svelte` directory. It doesn't matter what name you give it, but I'll call mine `svelte.json`. It can contain multiple snippets. Here's how my boilerplate component snippet looks:

```
{
 "Boilerplate Component": {
     "prefix": "<scr",
     "description": "Basic svelte boilerplate",
     "body": [
         "",
         "",
         "${2:<div></div>}",
         "",
         "<style>",
         "  $3",
         "</style>"
     ]
```

```
  }
}
```

If you are unfamiliar with VS Code snippet syntax:

- `prefix` is the string you type in insert mode to trigger the snippet. In this case, it is `<scr`.
- `description` is a string that describes it in the completion menu.
- `body` is a list of lines in the snippet.
- `$1`, `$2`, `$3` represent "tab stops" in the snippet.
- `${2:<div></div>}` represents a tab stop with placeholder content that can be typed over.

If I restart Neovim and load a svelte file, I can type `<scr` to insert this snippet. The default output looks like this:

```
<div></div>

<style>
</style>
```

Summary

This chapter introduced a grab bag of editing tips, starting with word counts and transposing characters, and then moving on to managing comments, indentation and formatting.

Finally, we covered the old-but-not-busted abbreviation syntax and the new-hotness Snippets engine that LazyVim ships with.

In the next chapter, we'll start discussing something completely different: version control in LazyVim.

Chapter 15: Source Control

LazyVim ships with several features to manage your source control history, and there are some excellent best-in-class third-party plugins you can use as well. Some of these plugins work with multiple version control systems, although some of them are git-centric. This book will assume you use git because, well, you probably do, even if you use other systems as well.

The Integrated Terminal (A rant)

For reasons I cannot explain, Neovim ships with a terminal emulator. It is bizarre to me that an editor that *runs in a terminal* ships a terminal. It is literally possible to open a terminal, open Neovim, open a terminal in Neovim, and open Neovim in a terminal in Neovim.

Add some nested ssh sessions if you really want to make a mess.

I don't need a terminal in my editor. I have a terminal already, an excellent one. I just use Kitty splits, tabs, and windows when I need a new terminal. The smart-splits plugin allows me to switch between editor and terminal seamlessly and Kitty even manages installing itself over ssh for me.

Or I press `Control-z`, which is the traditional way Vim users used to access a terminal. It is a shortcut that I really wish hadn't gone out of style. Pressing `Control-z` "suspends" Neovim. If you're not in the know, you'll think it closed your editor without saving, because the window disappears and returns you to your terminal.

But fear not! It is merely suspended, as indicated by the `'nvim' has stopped` message on the terminal.

```
lazyvim-for-ambitious-devs on  chapter-15 [!+?] via  v1.1.16 via  v22.3.0
❯ nvim
fish: Job 1, 'nvim' has stopped

lazyvim-for-ambitious-devs on  chapter-15 [!+?] via  v1.1.16 via  v22.3.0 took 11s
✦ ❯ jobs
Job     Group   State   Command
1       99346   stopped nvim

lazyvim-for-ambitious-devs on  chapter-15 [!+?] via  v1.1.16 via  v22.3.0
✦ ❯ fg
```

As this screenshot also shows, you can see the list of stopped (or running) background jobs using the `jobs` command in any shell (I recently switched to fish and highly recommend it). The `fg` (short for foreground) command starts the suspended Neovim process back up. If

you have multiple suspended jobs, the `fg %#` command can be used to choose a specific job id (e.g. `fg %1` will run the job with id `1` in the first column of the `jobs` output).

This is not a Neovim-specific feature. The `Control-z` trick works with (almost) any long-running shell command. You can even set a suspended task to keep running in the background by using the `bg` command instead of `fg` (though if the background job prints to stdout you'll quickly become confused).

Between terminal splits and `Control-z`, there's just no need for the editor to have its own terminal embedded with it. Still, Neovim ships with an integrated terminal, so I should probably explain how to use it.

The Integrated Terminal (for real this time)

You can pop up a terminal at any time in Lazyvim using the keybinding `Control-/`. It will appear in front of all your other editor windows (unless you have the edgy extra enabled, in which case it will show up in the bottom half of the window) and can be dismissed with `Control-/` again.

Neovim's terminal window is a super weird hybrid terminal and vim window. Once the terminal is open, you can use normal mode commands to navigate around it, but only if you know the secret and hard-to-type incantation `<Control-\><Control-n>`. `Escape` WILL NOT put you in normal mode, even though your fingers are, by now, conditioned to hit `Escape` reflexively.

This actually makes sense because `Escape` is a common key to need to type in various terminal programs, so it would be rude for Neovim to steal it.

In Normal mode, you can navigate anywhere in the terminal window using any of the navigation keys including seek and search modes. This can occasionally be helpful if you need to yank some outputted text to the clipboard.

Pressing a key such as `a` or `i` will send you back to "Terminal mode" which effectively just sends every keystroke to the program currently running in the terminal (probably your shell).

Annoyingly, this means you can't use Normal mode to reposition your cursor on the command line; it will go back to wherever it was when you last typed `<Control-\><Control-n>`.

> If you want to use vim normal modes to edit your command line (in any terminal —not just inside Neovim) configure your shell to use "vi mode." All modern shells support some version of this, and it usually allows you to use `Esc` to put the shell

> in a pseudo-normal mode. It gives you commands like `w` and `b` for navigation and basic line-editing commands like `d` and `c` to edit the command line.

There are third-party plugins that try to make the terminal experience more consistent and enjoyable, but in my opinion, they are not worth the trouble. I can just press `cmd-enter` to get a new Kitty terminal pane and have a perfectly normal terminal experience.

Checking your git status

Lazyvim is preconfigured with a handful of carefully configured plugins that make your version control life much better.

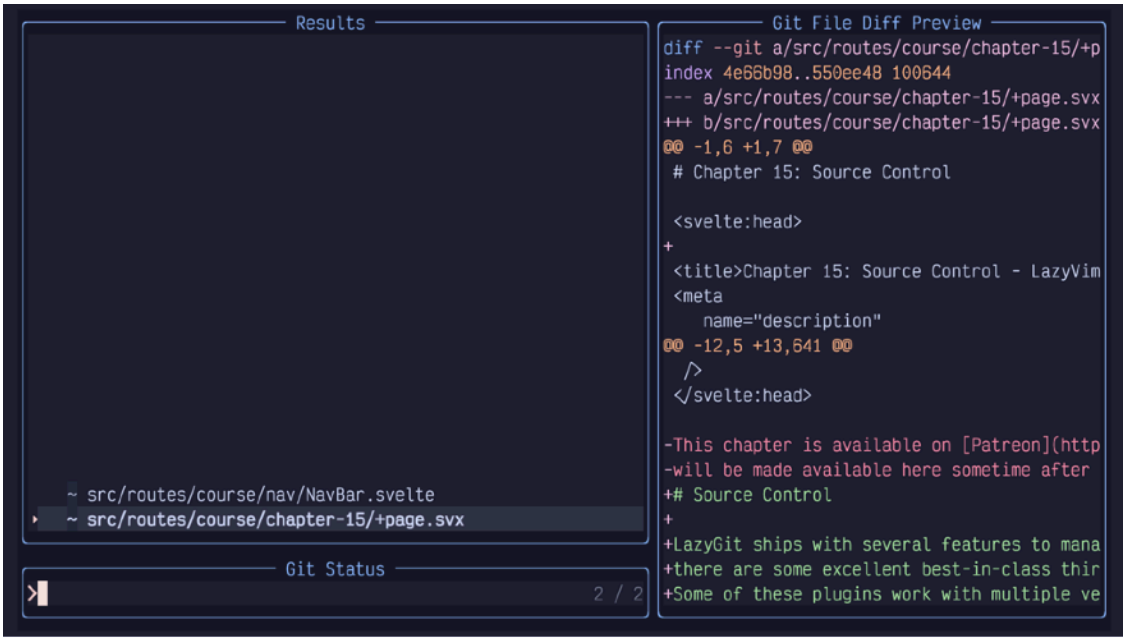
Probably the simplest of these uses your configured file picker (`Telescope` or `fzf-lua`, as discussed in Chapter 4) to list files that have changed since the last commit. This will behave similarly to other file picker operations, except it only lists files that have modifications in git.

You can open it with `<Space>gs`. I use it a lot for switching between files related to whatever feature I am currently working on, and actually prefer it to the buffer picker (which only shows opened files) we discussed in Chapter 9.

The popup behaves slightly differently depending on whether you use Telescope or fzf.lua. I'll explain with Telescope first, and mention how fzf.lua is different afterwards. (I hate that folke makes me do this. Please, can we just collectively settle on one best picker and use it?)

With Telescope

This Telescope screenshot shows that I have modified two files since my last commit:



The preview pain shows the diff of lines I have added and removed. On the left, you can see that I have `page.svx` focused, and a preview of some of the changes in this file on the right.

The confusing bit to pay attention to is the first two columns. They indicate your git status, and their meaning can be devilishly hard to remember. The symbols themselves are straightforward:

- ~ means the file on that line contains modifications since the last commit
- – means it has been deleted
- `?` means it is an untracked file (has been added to the working directory but not staged or committed)
- + means it is a new file that has been staged in git

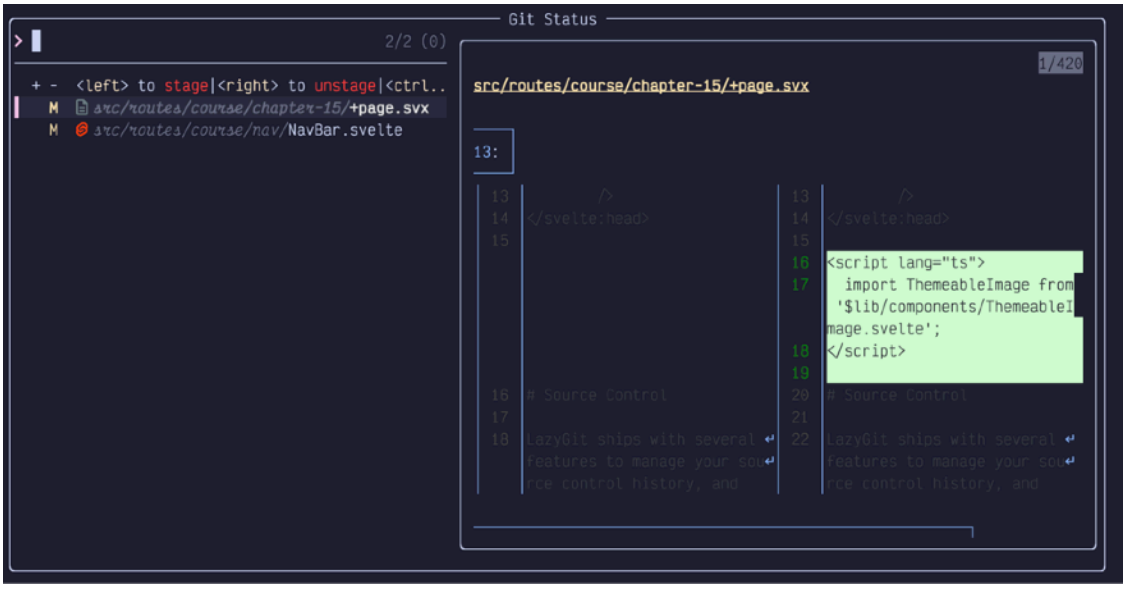
If the sign shows up in the *first* column, it means the file has been staged and will be included in the next commit. If it is in the *second* column, then it means the file is not yet staged. If a ~ is in both columns, some parts of it have been staged and some parts have not.

I had to use this picker quite a few times before I could remember whether column 1 or column 2 mean “staged”, and if all or none of the files are staged it can be hard to tell which column is empty.

In addition to allowing you to effectively view your git status, this picker also allows you to *stage* entire files. To do so, focus a file and hit the `<Tab>` key. If it is staged it will become unstaged and vice versa, moving the symbol between the first and second columns..

With Fzf.lua

Fzf.lua behaves similarly, but not identically to Telescope. If you've installed the `fzf.lua` extra, the same keybinding (`<Space>gs`) pops up an fzf window instead (ensure the delta-pager CLI tool is installed to get the pretty diffs):



The main difference is that you use the left and right arrow keys to stage or unstage a file instead of `Tab`, and you can additionally use the `Control-x` keybinding to reset an entire file to the last commit state. Unlike Telescope, these keybindings are helpfully written across the top of the picker so you don't have to memorize them.

The two columns are labelled + and –. I'm not sure why those symbols were chosen, as they don't reflect whether files or lines are added or deleted. The + column holds files that have been staged to go in the next commit, while the – column holds a status for files that have changed but have not yet been staged. This is the same as Telescope, but it's a little bit clearer with the heading symbols on there.

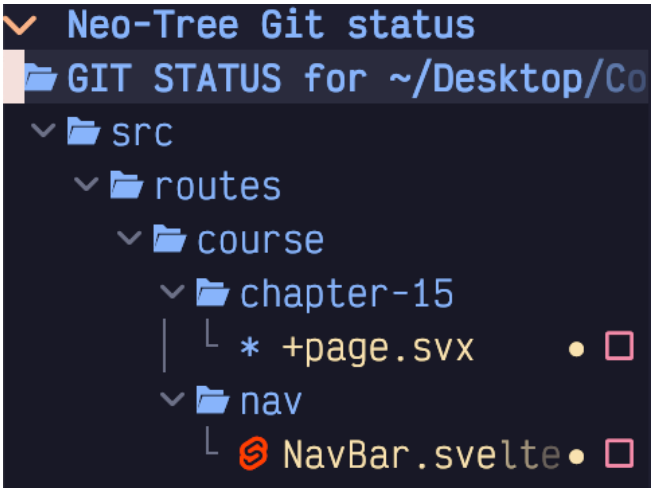
Other pickers

Telescope and fzf.lua both come with pickers to view and search commit history (`<Space>gc`), kind of like a log browser, and to check out a branch. The latter doesn't have a keybinding for some reason, but you can bind one to `:Telescope git_branches` or `:FzfLua git_branches` if you like the picker UI for this task.

> There are a variety of less commonly-used git-related pickers you can find by typing either `:FzfLua` or `:Telescope` and then `Enter` and `git`.

Git Files in Neotree

Neotree also has a Git status viewer. It has the advantage of displaying any changed files inside a folder hierarchy. Here's the same two files from the previous example as rendered in Neotree:

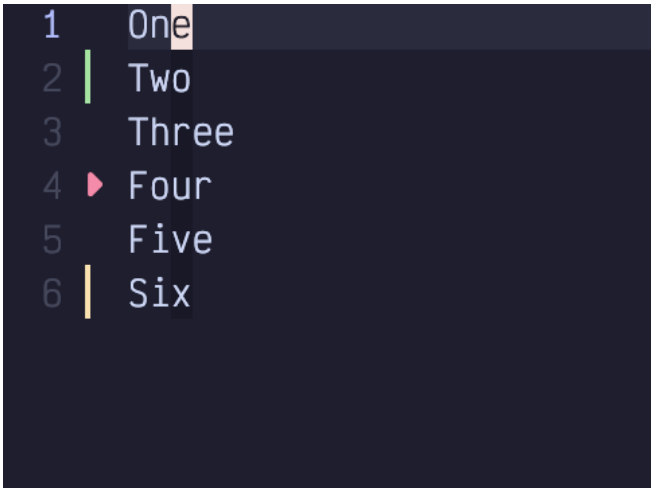


To stage and unstage a file with Neotree, use `ga` (git add) and `gu` (git unstage) while your cursor is over that line. The `A` keybinding will stage all unstaged files.

You can also use `gc` to commit the current state. This pops up a crappy little text entry window that is absolutely not suitable for typing a proper-length commit message, so I suggest avoiding it. You can also use `gp` to push the current branch to the remote repository. I recommend using the `lazygit` integration discussed later instead, but these commands are available if you spend a lot of time in Neotree.

Status of the currently focused file

Every buffer has a couple subtle indications of the changes in that file. Consider this screenshot:

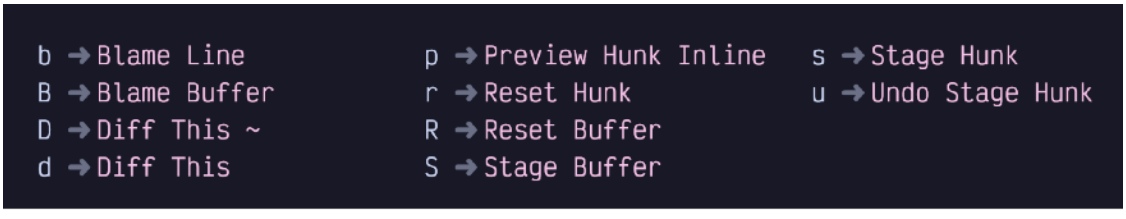


Notice the left sidebar, to the right of the line numbers. it contains a green bar, a small red triangle, and a short orange bar. These indicators all show that lines have been added, removed, and modified, respectively.

Additionally, in the status bar, just to the left of the file progress indicator we see these icons, which summarize the same information:

Staging from the editor

You can add files to git's index (so they are ready to commit) right from the editor. The `<Space>gh` menu (mnemonic is "git hunks", although you may think of it as "git hub" if that's where most of your work happens) has a bunch of interesting subcommands you can leverage for this:



You can use `<Space>ghS` to stage an entire file, which would move it to the left column in the git status pickers we discussed above. If you want to create a patch of a subset of your changes, navigate to the hunk you want to stage (`[h` and `]h` are super handy for this) and hit `<Space>ghs`.

Most people have an unfortunate habit of just committing everything instead of properly curating their history, but if you are one of the rare folks who uses git properly (please be that person), you'll use this command a lot.

You can also reset a hunk (effectively making it the same as it was at the time the last commit was made) using `<Space>ghr`. if you want to reset the entire file, use the "but bigger" `<Space>ghR`. Resetting is a destructive operation, so be careful (though `u` for undo can usually get you back to where you were).

If you accidentally stage a hunk, use `<Space>ghu` to unstage it. Unlike reset, this won't change the file; the changes will still be there; they just won't be staged anymore.

The Blame line (`<Space>ghb`) shows the commit that last changed the line the cursor is currently on, useful for answering the all-important question "Why on Earth did I do that?"

The Preview hunk (`<Space>ghp`) temporarily renders the original and changed version of a hunk (one above the other) so you can see exactly what changed.

The `Diff this` (`<Space>ghd` and `<Space>ghD`)commands do the same except in a side-by-side view that we will discuss later in this chapter.

Personally, I use many of these commands too often for the number of keystrokes required to pop them up. So I've created an `extend-gitsigns.lua` file in my plugins directory that copies them from `<Space>gh` to `<Space>h`:

```
return {
  "lewis6991/gitsigns.nvim",
  keys = {
    {
      "<leader>hb",
      "<cmd>Gitsigns blame_linepcr>",
      desc = "Stage Hunk"
    },
    {
      "<leader>hs",
      "<cmd>Gitsigns stage_hunk<cr>",
      desc = "Stage Hunk"
    },
    {
      "<leader>hS",
      "<cmd>Gitsigns stage_buffer<cr>",
      desc = "Stage Buffer"
```

```
    },
    {
      "<leader>hr",
      "<cmd>Gitsigns reset_hunk<cr>",
      desc = "Reset Hunk"
    },
    {
      "<leader>hR",
      "<cmd>Gitsigns reset_buffer<cr>",
      desc = "Reset Buffer"
    },
    {
      "<leader>hu",
      "<cmd>Gitsigns undo_stage_hunk<cr>",
      desc = "Undo Stage Hunk"
    },
  },
}
```

I got these by copying them from the git-signs config on the LazyVim website and converting it from `map` calls to the `keys =` format.

Lazygit

Lazygit (which is not maintained by folke in spite of sharing the `Lazy` namespace with LazyVim and Lazy.nvim) is a terminal UI tool for interacting with git. It is a separate program that you will need to install with your operating system's package manager (e.g. `brew install lazygit`) if you want to use it.

LazyVim is preconfigured to show lazygit in a terminal window using the keybinding `<Space>gg`. I won't go into all the details of how to use this third-party program. It can do anything git can do in a much more user-friendly interface.

Lazygit takes a bit of study to get used to, but it has helpful menus and mnemonics for its keybindings so the learning curve is relatively gentle.

Ironically, I used lazygit (in its standalone format from the command line) a lot more before I started using LazyVim. I used to stage changes using lazygit, but now I use the `<Space>h` menu we just covered instead.

I also do most of my git work with the exceptional Graphite tool (visit https://graphite.dev for those interested), which simplifies many of the flows I used to use lazygit for (especially

rebasing). I still use lazygit every day; I just don't have it open 100% of the time like I used to.

Diff Mode

Neovim comes with a very powerful, but slightly hard-to-learn diff viewing mode. It shows "before" and "after" files side by side and can even be configured to show the "parent" and changed state if you want a fancy merge tool.

There are several different ways to get yourself into diff mode. The basic way is to specify it when you open two files on the command line:

```
nvim -d file1 file2
```

This opens the indicated files side by side. Most often, you won't have two separate files, though. Instead, you'll want to see the difference between the current file and the staging index, which you can do with the shortcut `<Space>ghd`. Or use `<Space>ghD` to show the differences between the current file and the last commit, regardless of what has been staged.

> **Important:** Once you are done operating in diff mode, it can be tricky to get back to the normal file. The issue is that when a file is in `diff` mode, it stays that way, even if other windows are opened or closed. The secret is to use the `:diffoff` command, which will disable "diff view" for the current buffer. This doesn't close the two side-by-side windows, though; you'll need to use normal window and buffer management tooling such as `<Space>bd` and `<Control-w>q` to do that.

Note that by default, the diff view will collapse any code that is identical between the two files into a single fold. Use the code unfolding command `zo` to expand a section.

Editing Diffs

If you use the `<Space>ghd` command to show your file in diff against the index mode, you can keep editing the file to make additional changes. If you do this, only edit the file on the *right*. This is the "working" file. The file on the left is the "index" file; it shows the staged changes. If you want to "edit"" the file on the left, use the `<Space>ghd`, `<Space>ghr`, and `<Space>ghu` to stage, reset, and unstage hunks from the right side. It is not *forbidden* to edit the index file directly, but it will confuse the diff mode machinery, so stick to staging and unstaging from the right side.

When working with diff view like this, I find that the stage, reset, and unstage keybindings best match the mental model I am used to. However, there are two kind of weird commands built into Neovim that you may sometimes need to reach for: `:diffget` and `:diffput`.

These are more commonly typed as `:diffg` and `:diffp` to save a couple keystrokes.

These commands are most often used in visual mode (or with a range), and they essentially mean that (within that range) we should either "make this file the same as the other file" or "make the other file the same as this file", respectively.

Consider these two files that are slightly different:

The file on the left represents the state of my index, while the file on the right is my working copy. The indexed version was missing the word "Two", so I have added that on the right. It also had an extra "Four Point Five" line that I have removed on the right. And I modified the spelling of the word "Six".

Let's explore a couple ways to make these files identical with `:diffg` and `:diffp`. You can use these commands on either file, but it usually makes sense to operate on only one of them. For this example, assume I'm working on the right-hand file.

I can use any navigation commands to jump to the second line of the file. If you are editing a real git indexed file, the `[h` and `]h` keybinding are probably useful for jumping between hunks. However, when you are in "diff" mode you can also use the `[c` and `]c`, which mean "jump between changes," but *only* when you are in "diff" mode. (in a non-diff window, LazyVim has bound those keys to jump between classes or types.) I usually just use `[h` and `]h`, but in those instances where I am using a diff view that is not attached to git history, `[c` and `]c` should not be forgotten.

So with my cursor on the first line of the file, `[c` or `[h` will jump to the second line, which contains the word `Two` in my file, but not the index.

I want to stage this change, so I type `:diffp`, which means "make the other file the same as this one."

The next line is `Four Point Five` in the left file, but was deleted in the right file. For the sake of argument, let's say I want to "unstage" this change, which is to say "make the right file the same as the left file". To do this from the right window, I can use `Shift-V` to enter visual line mode, and select the lines containing `Four` and `Five` as well as the blank red

space between those two lines representing the deleted line. Now I can type `:diffg` or `diffget` which means "get the contents of the other window and make my window match it." Since `:diffget` and `:diffput` accept ranges, it passes the visual selection with the usual `'<` and `'>` marks.

> Tip: If you find you like the above diff interface, but figuring out which files have differences is frustrating, you may want to configure the `sindrets/diffview.nvim` plugin. I personally just use the git status telescope picker, but the `diffview.nvim` plugin has a nice interface and some handy commands.

Configuring Vim diff as merge tool

Everyone seems to hate resolving merge conflicts. Armed with diff mode and rebasing, I actually find the process kind of enjoyable. The trick is to have a slightly complicated `~/.gitconfig` (and a very large monitor).

I can't help you with the monitor, but the `.gitconfig` needs to look like this:

```
[diff]
    tool = vimdiff
[merge]
    tool = vimdiff
    conflictstyle = zdiff3
[mergetool "vimdiff"]
    cmd = nvim -d $LOCAL $BASE $REMOTE $MERGED \
          -c '$wincmd w' -c 'wincmd J'
```

The `zdiff3` conflictstyle makes diffs a bit easier to read by automatically resolving identical lines. The two `tool =` lines say to use the `vimdiff` merge tool that is configured on the last line.

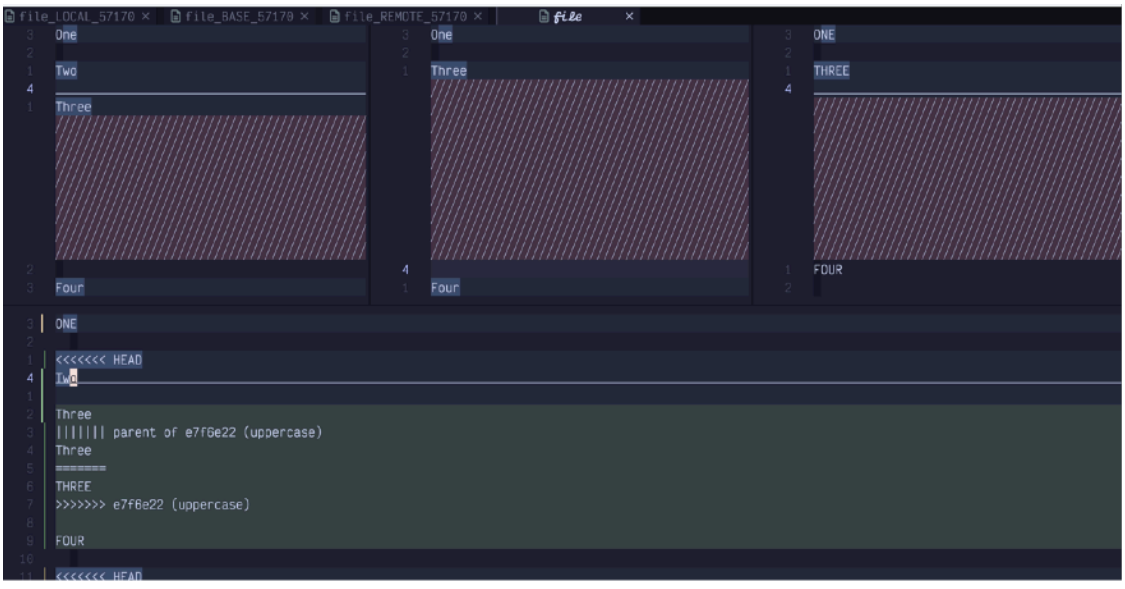
That last line is a command to open Neovim with a whopping FOUR windows open and focuses the appropriate one.

To demonstrate this, I made a new git repo with two branches with conflicting changes. When I went to rebase (I always use rebase rather than merge commits because it allows me to deal with conflicts in the isolation of one change. This is why it's important to me that every commit have only one change!), one branch onto the other, of course, I end up with this error:

```
✦ ❯ git rebase main
Auto-merging file
CONFLICT (content): Merge conflict in file
```

```
error: could not apply f611b6f... Uppercase
Could not apply f611b6f... Uppercase
```

To resolve this conflict, I run `git mergetool`. Because of the git configuration above, iit will open Neovim with these four different diff windows:



There are three windows across the top and one in a big pane (also pain) in the bottom.

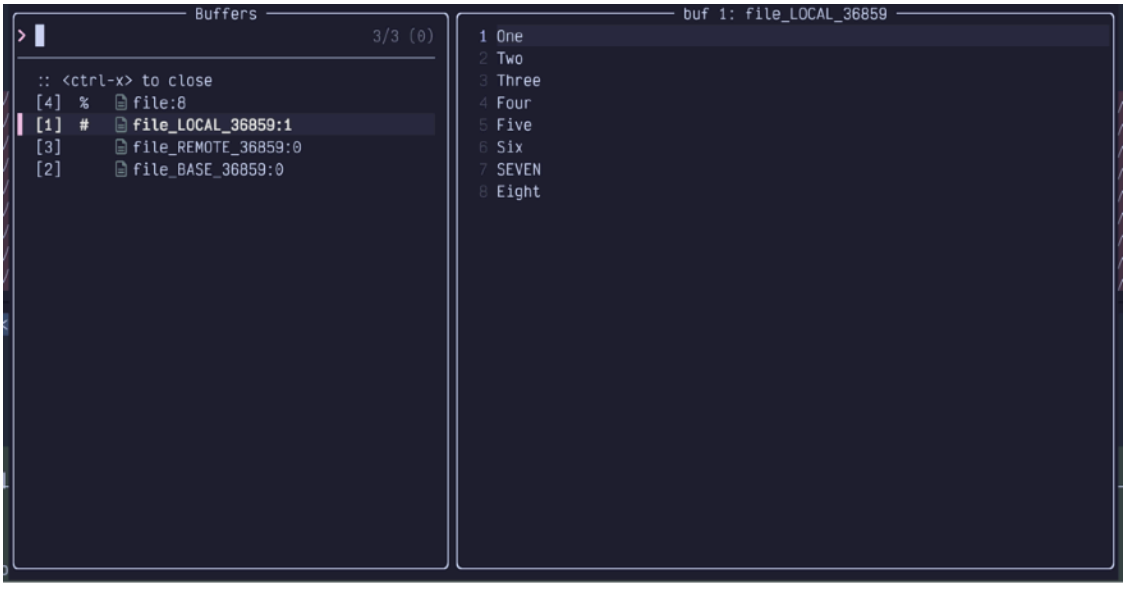
- The *upper-left* window shows the “Local” changes. The meaning of “local” depends on exactly what commands you used to get into the conflict situation. In typical rebase flows, it returns to “the current state of the main branch”. So when there is a conflict, it would contain “the other person’s changes”, so “local” doesn’t seem applicable.
- The middle window contains the “common ancestor” or “base” of the changes. Which is to say, this is the state of the file before either you or “the other person” made any changes. This window is not commonly described in merge-tool tutorials, but I find it can be quite helpful when trying to figure out what changed between the base and each of the two side windows.
- The right window contains the “Remote” changes, which, like local, can be a misnomer. In rebase flows, it usually means “the changes I made on the branch I am rebasing onto main.”
- The lower window contains “the current state of the file”, which at the time the rebase failed includes messy conflict markers. This is the only file you should make edits to.

All four files will feature code folding if there are long sections of common code. Also, if you scroll or move the cursor in the lower file, the upper files will also scroll so everything stays in sync, and an underline in the top three windows will indicate which line the diff tool thinks is the “current” one with respect to the cursor position in the lower window.

Most rebase flows start with using `vag` and `:diffg` from the lower window to make it identical to one of the upper windows. Then you would use `diffget` to get hunks from the left or right window, depending on context. You’ll also usually have to do some manual editing because if it was possible to automatically get all the changes, git would not have thrown a conflict!

The problem is, `:diffg` doesn’t know which window to get things from because there are multiple windows open:

Instead, we need to use the command `:%diffg 2`. The `2` is a buffer number. When you run merge-tool directly from the command line, the buffers are numbered in the order they are open. So `1` is the left-hand buffer, `2` is the middle one, `3` is the right-hand one, and `4` is the lower window. If you aren’t sure, you can use the `<Space><comma>` keybinding to show the buffer list:



In this list, the first column holds the buffer number. This number generally increases monotonically from the most recent time Neovim opened, so it can get pretty high if you've been editing for a while. But when you use `git mergetool`, it typically opens a brand new Neovim instance and `1-4` are expected.

After running `vag` and the `:%diffg 2` command, the bottom window looks the same as the middle window, which is the state everything was before either branch was created. If I used `vag` and then `:%diffg 1` it would look the same as `main`, and `vag` followed by `:%diffg 3` would make it look the same as my branch. Then I could selectively look at differences between buffers and use `:diffg #` to get changes from the left or right one respectively.

Merge conflicts can always be somewhat stressful, but I find the four window view often makes it easier to understand what changed and why. That said, I only reach for it when I'm in a particularly knotty merge situation. Normally, I use the git-conflict.nvim plugin.

git-conflict.nvim

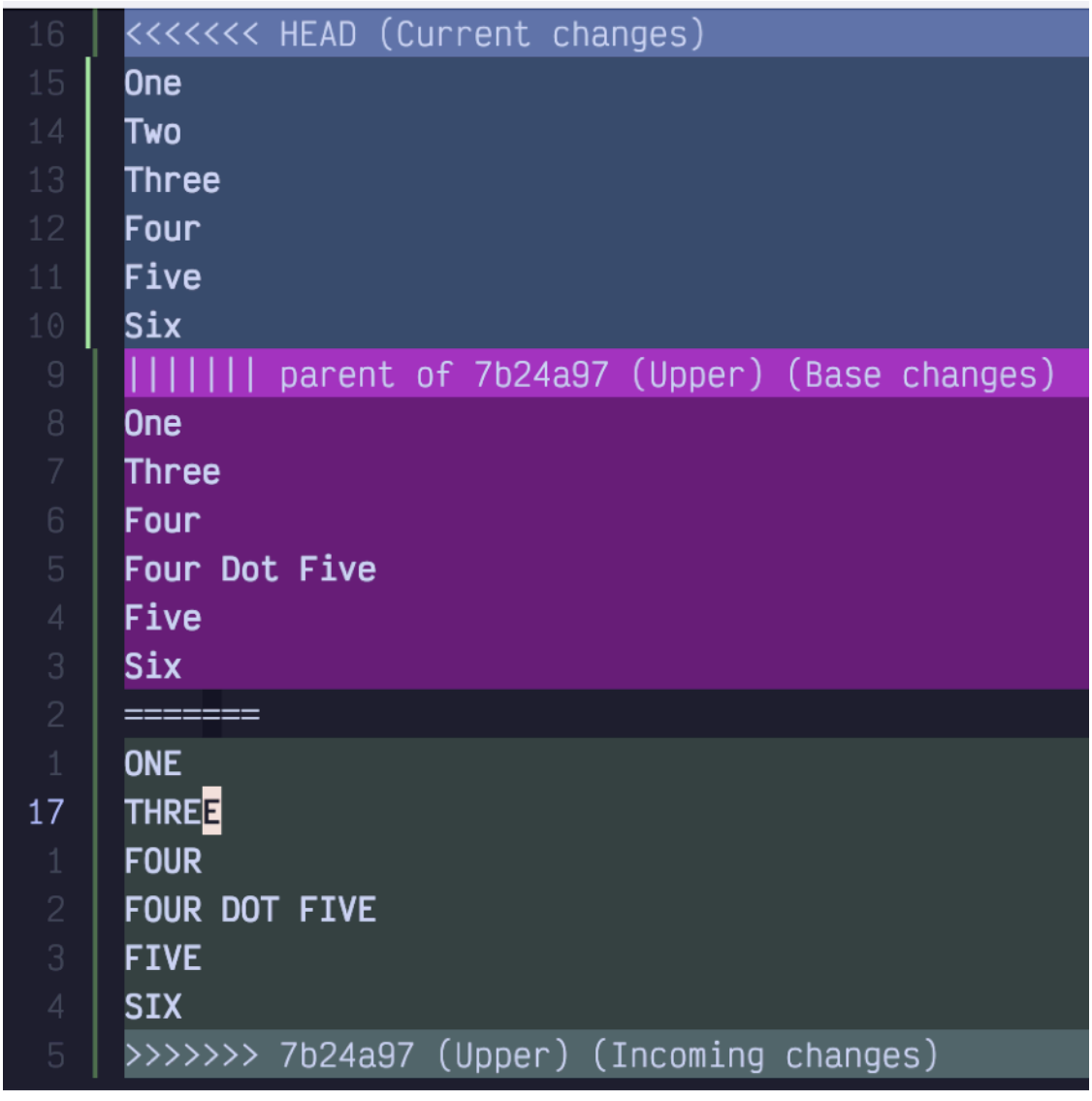
While merge-tool is very helpful when working with particularly complicated merges, for simple conflicts, I usually find it quicker to just edit the file with the conflict markers in it directly. A plugin called `git-conflict.nvim` provides syntax highlighting and some keybindings to help navigate conflicts.

Set it up with a config something like this:

```
return {
  "akinsho/git-conflict.nvim",
  lazy = false,
  opts = {
    default_mappings = {
      ours = "<leader>ho",
      theirs = "<leader>ht",
      none = "<leaderh>0",
      both = "<leaderhb",
      next = "]x",
      prev = "[x",
    },
  },
  keys = {
    {
      "<leader>gx",
      "<cmd>GitConflictListQf<cr>",
```

```
      desc = "List Conflicts"
    },
    {
      "<leader>gr",
      "<cmd>GitConflictRefresh<cr>",
      desc = "Refresh Conflicts"
    },
  },
}
```

I use the `<Space>h` prefix that I set up previously for staging hunks and add a few new commands to it. After enabling this extension, if you open a file with conflicts, it highlights the conflict markers in a different colour. On my plaintext sample file it looks like this:



The conflict markers include the “current” (whatever is on main) code above and the “new” (whatever is being rebased) code below the original or base code (before either change) in the middle.

I can use the `]x` keybinding to quickly jump to the next conflict (in this case there is only one). Then I can use one of the following keybindings to resolve the conflict:

- `<Space>ho` Choose the top version
- `<Space>ht` Choose the bottom version
- `<Space>hb` Choose both

- `<Space>h0` Go back to whatever is in the middle

The `o` and `t` keybindings are hard to remember. Technically they mean "ours" and "theirs", but depending on which order you did a merge or rebase, it doesn't always semantically map to your own or somebody else's commit. I just remember that `o` is before `t` in the alphabet, so it means the upper change.

In all cases, but especially in the latter two, you will likely need to do some manual editing to make the code look correct. This is normal. None of the conflict management extensions uses AI to semantically understand what the changes *intended* to do, so you still need to do that part yourself!

About ninety percent of the time, this plugin is all I need to resolve a conflict. I only use the mergetool when things are particularly hairy or complicated.

Summary

This chapter introduced a lot of different ways of interacting with git and version control from inside LazyVim. You probably won't use all of it, but I wanted to present multiple options so you can decide which ones work best for you.

Perhaps you want to use Lazygit, or maybe you want to stay in the editor and use the functionality that git-signs and native vim diff mode provide. Maybe you want to install some extra plugins such as `git-conflict.nvim` or `diffview.nvim` to streamline your experience (others you might want to look at include `Neogit` and `mini.git`).

Or maybe you don't want to manage this stuff from your editor at all and just want to drop to terminal mode and use `git` or a wrapper like `graphite`. Whatever works for you, LazyVim provides the integrations you need.

In the next chapter we'll admit that it's not 2020 anymore and talk about Artificial Intelligence.

Chapter 16: Configuring Artificial Intelligence

Chapter 16: Configuring Artificial Intelligence - LazyVim for Ambitious Developers

This chapter is available on Patreon and will be made available here sometime after July 17, 2024.

Chapter 17: Debugging

Chapter 16: Debugging - LazyVim for Ambitious Developers

This chapter is available on Patreon and will be made available here sometime after July 25, 2024.

Chapter 18: Testing

Chapter 16: Debugging - LazyVim for Ambitious Developers

This chapter is available on Patreon and will be made available here sometime after July 31, 2024.