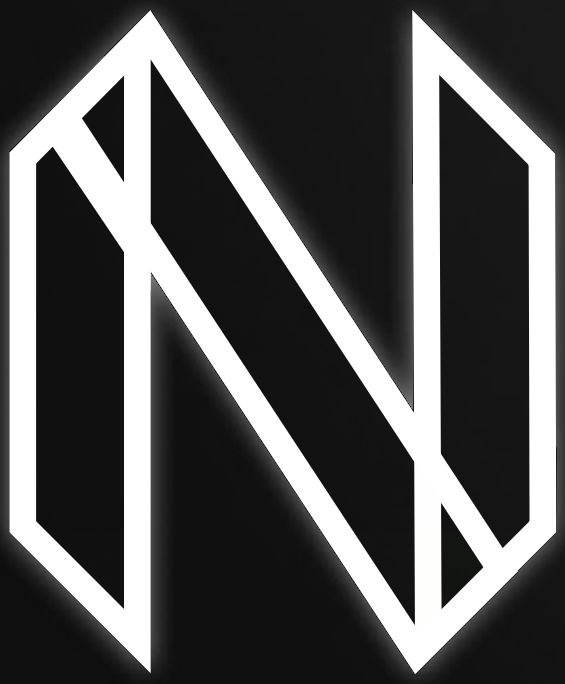


Effective NeoVim Setup for 2024

+ Additional Plugins & Tips

sindo.dev

Introduction

This is a free e-book written by Sindo. To access even more e-books just like this one visit sindo.dev.

GitHub Repository

You can find the dotfiles for my setup here:

 Dotfiles on GitHub

The Backbone of the setup

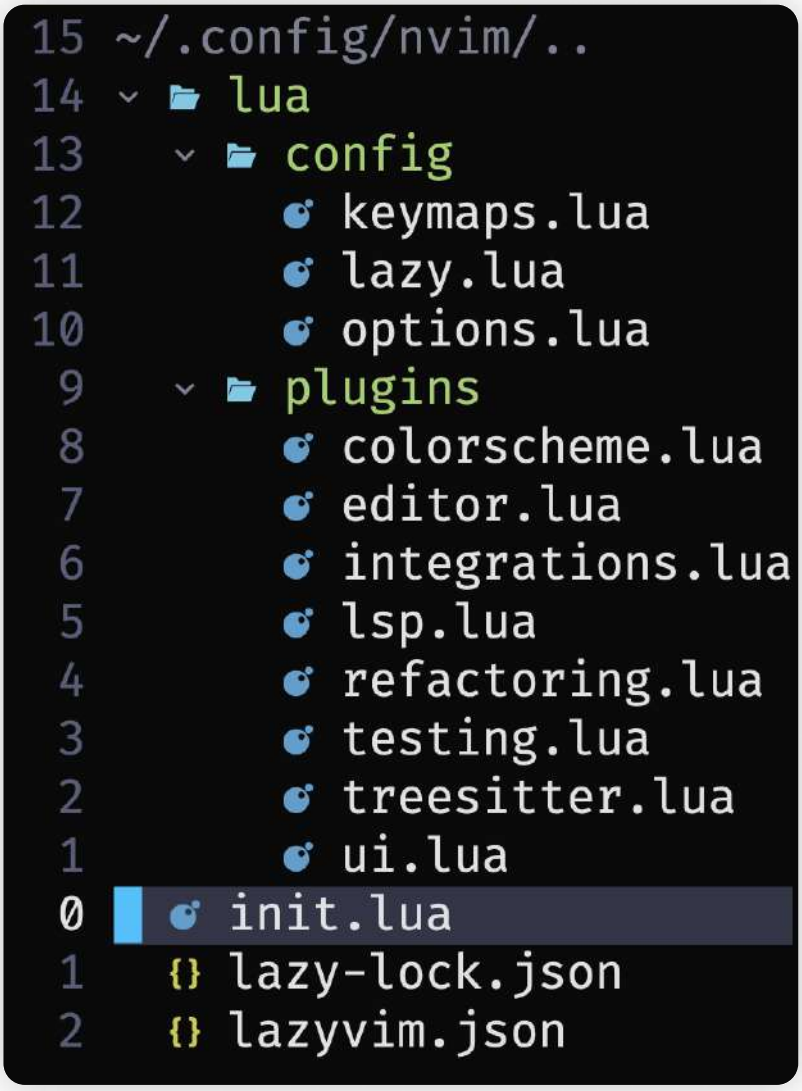
For this setup, we're using LazyVim. It's a strong starting point for many configurations, and it features the Lazy plugin manager at the heart of the configuration.
There are other popular alternatives like NvChad which also use the Lazy plugin manager, so which starter you're using will depend on your own preferences.
In our case, LazyVim is arguably more barebones than NvChad, while still providing the majority of the necessities out of the box.

Installing the LazyVim Starter:

https://www.lazyvim.org/installation

File Strucutre

Here is the file structure of our setup. You can use it for reference as you're going through the e-book. For the entire codebase, make sure to check out the GitHub repository listed on the first page of the e-book.



LazyVim Config

We're keeping the default LazyVim config, but adding in some additional plugins that were included out of the box.

We're also setting the color scheme here, but we'll install Sonokai properly in a later step.

LazyVim Config

```
local lazypath = vim.fn.stdpath("data") .. "/lazy/lazy.nvim"
if not vim.loop.fs_stat(lazypath) then
  vim.fn.system({
    "git",
    "clone",
    "--filter=blob:none",
    "https://github.com/folke/lazy.nvim.git",
    "--branch=stable",
    lazypath,
  })
end
vim.opt.rtp:prepend(lazypath)

require("lazy").setup({
  spec = {
    {
      "LazyVim/LazyVim",
      import = "lazyvim.plugins",
      opts = {
        colorscheme = "sonokai",
        news = {
          lazyvim = true,
          neovim = true,
        },
      },
    },
    { import = "lazyvim.plugins.extras.linting.eslint" },
    { import = "lazyvim.plugins.extras.formatting.prettier" },
    { import = "lazyvim.plugins.extras.lang.typescript" },
    { import = "lazyvim.plugins.extras.lang.json" },
    { import = "lazyvim.plugins.extras.lang.tailwind" },
    { import = "lazyvim.plugins.extras.coding.copilot" },
    { import = "lazyvim.plugins.extras.util.mini-hipatterns" },
    { import = "plugins" },
  },
  defaults = {
    lazy = false,
    version = false,
  },
  dev = {
    path = "~/.ghq/github.com",
  },
  checker = { enabled = true },
  performance = {
    cache = {
      enabled = true,
    },
    rtp = {
      disabled_plugins = {
        "gzip",
        -- "matchit",
        -- "matchparen",
        "netrwPlugin",
        "rplugin",
        "tarPlugin",
        "tohtml",
        "tutor",
        "zipPlugin",
      },
    },
  },
  debug = false,
})
```

Configuring Vim Options

Vim Options

```
vim.g.mapleader = " "

vim.scriptencoding = "utf-8"
vim.opt.encoding = "utf-8"
vim.opt.fileencoding = "utf-8"

vim.opt.number = true

vim.opt.title = true
vim.opt.autoindent = true
vim.opt.smartindent = true
vim.opt.hlsearch = true
vim.opt.backup = false
vim.opt.showcmd = true
vim.opt.cmdheight = 0
vim.opt.laststatus = 0
vim.opt.expandtab = true
vim.opt.scrolloff = 10
vim.opt.inccommand = "split"
vim.opt.ignorecase = true
vim.opt.smarttab = true
vim.opt.breakindent = true
vim.opt.shiftwidth = 2
vim.opt.tabstop = 2
vim.opt.wrap = false
vim.opt.backspace = { "start", "eol", "indent" }
vim.opt.path:append({ "**" })
vim.opt.wildignore:append({ "*/node_modules/*" })
vim.opt.splitbelow = true
vim.opt.splitright = true
vim.opt.splitkeep = "cursor"
vim.opt.mouse = ""

-- Add asterisks in block comments
vim.opt.formatoptions:append({ "r" })
```

The first and arguably most important step of the setup is defining some Vim options that make Vim actually *useable*.

Here are the options that I use, and we'll break them down line by line.

Configuring Vim Options

1. `vim.g.mapleader = " "`: This sets the map leader to space. Map leader is a special key that is used to define custom key mappings.
2. `vim.scriptencoding = "utf-8"`: Sets the encoding used for reading and writing Vim scripts to UTF-8.
3. `vim.opt.encoding = "utf-8"`: Sets the default encoding for the current buffer to UTF-8.
4. `vim.opt.fileencoding = "utf-8"`: Sets the encoding used for reading and writing files to UTF-8.
5. `vim.opt.number = true`: Displays line numbers on the left side of the buffer.
6. `vim.opt.title = true`: Allows Vim to set the terminal window title according to the current file name.
7. `vim.opt.autoindent = true`: Automatically indents new lines based on the indentation of the previous line.
8. `vim.opt.smartindent = true`: Automatically adjusts indentation in certain situations, such as when creating new lines or indenting with the < and > commands.
9. `vim.opt.hlsearch = true`: Highlights all occurrences of the search pattern as you type.
10. `vim.opt.backup = false`: Disables Vim's automatic backup creation.

Configuring Vim Options

11. `vim.opt.showcmd = true`: Shows the partial command you are typing on the last line of the screen.
12. `vim.opt.cmdheight = 0`: Specifies the number of screen lines to use for the command-line.
13. `vim.opt.laststatus = 0`: Hides the status line if there is only one window open.
14. `vim.opt.expandtab = true`: Converts tabs to spaces when you press the Tab key.
15. `vim.opt.scrolloff = 10`: Specifies the minimum number of lines to keep above and below the cursor when scrolling.
16. `vim.opt.backupskip = { "/tmp/*", "/private/tmp/*" }`: Specifies file patterns to be excluded from backup operations.
17. `vim.opt.inccommand = "split"`: Shows the effect of a command incrementally as you type it.
18. `vim.opt.ignorecase = true`: Makes searches case-insensitive by default.
19. `vim.opt.smarttab = true`: Makes the Tab key insert spaces instead of a literal tab character when appropriate.
20. `vim.opt.breakindent = true`: Adds indentation to wrapped lines that are broken at a word.
21. `vim.opt.shiftwidth = 2`: Specifies the number of spaces to use for each step of (auto)indent.

Configuring Vim Options

22. `vim.opt.tabstop = 2`: Specifies the number of spaces that a <Tab> in the text stands for.
23. `vim.opt.wrap = false`: Disables line wrapping, causing long lines to extend beyond the edge of the screen.
24. `vim.opt.backspace = { "start", "eol", "indent" }`: Allows backspacing over autoindent, line breaks, and the start of insert.
25. `vim.opt.path:append({ "**" })`: Appends the recursive wildcard pattern to the 'path' option, enabling searching for files in subdirectories.
26. `vim.opt.wildignore:append({ "*/node_modules/*" })`: Specifies patterns to ignore during file name completion.
27. `vim.opt.splitbelow = true`: Opens new split windows below the current window.
28. `vim.opt.splitright = true`: Opens new split windows to the right of the current window.
29. `vim.opt.splitkeep = "cursor"`: Specifies the behavior when closing a window. In this case, it keeps the cursor in the same position.
30. `vim.opt.mouse = ""`: Disables mouse support in Vim.
31. `vim.opt.formatoptions:append({ "r" })`: Specifies formatting options. In this case, it adds the option to continue comment lines started with `*` in a new line with the same indentation.

Configuring Keymaps

Vim Keymaps

```lua
local keymap = vim.keymap
local opts = { noremap = true, silent = true }

keymap.set("n", "x", '"_x')

-- Increment/decrement
keymap.set("n", "+", "<C-a>")
keymap.set("n", "-", "<C-x>")

-- Select all
keymap.set("n", "<C-a>", "gg<S-v>G")

-- Save file and quit
keymap.set("n", "<Leader>w", ":update<Return>", opts)
keymap.set("n", "<Leader>q", ":quit<Return>", opts)
keymap.set("n", "<Leader>Q", ":qa<Return>", opts)

-- File explorer with NvimTree
keymap.set("n", "<Leader>f", ":NvimTreeFindFile<Return>", opts)
keymap.set("n", "<Leader>t", ":NvimTreeToggle<Return>", opts)

-- Tabs
keymap.set("n", "te", ":tabedit")
keymap.set("n", "<tab>", ":tabnext<Return>", opts)
keymap.set("n", "<s-tab>", ":tabprev<Return>", opts)
keymap.set("n", "tw", ":tabclose<Return>", opts)

-- Split window
keymap.set("n", "ss", ":split<Return>", opts)
keymap.set("n", "sv", ":vsplit<Return>", opts)

-- Move window
keymap.set("n", "sh", "<C-w>h")
keymap.set("n", "sk", "<C-w>k")
keymap.set("n", "sj", "<C-w>j")
keymap.set("n", "sl", "<C-w>l")

-- Resize window
keymap.set("n", "<C-S-h>", "<C-w><")
keymap.set("n", "<C-S-l>", "<C-w>>")
keymap.set("n", "<C-S-k>", "<C-w>+")
keymap.set("n", "<C-S-j>", "<C-w>-")

-- Diagnostics
keymap.set("n", "<C-j>", function()
  vim.diagnostic.goto_next()
end, opts)
```

Let's also define some keymaps that make navigating Vim much more intuitive.

We'll also go over them in detail.

Configuring Keymaps

Firstly, let's make it so that pressing "x" to cut a single character, doesn't actually store that character in the clipboard, overwriting whatever we currently have copied. Instead, we execute the "x" command while storing the result in the underscore register.

We also make incrementing and decrementing numbers easier by just hovering over them with the cursor and pressing the + and - keys respectively.

We map saving and exiting NeoVim to Leader (Space) + q, w, and Q.

We make working with tabs easier, and make splitting windows a little bit faster.

Also, we make moving and resizing windows a bit more intuitive.

Finally, we map Control + j to opening up the diagnostics, so we can inspect warnings and errors in our code quickly.

Colorscheme

Probably the most interesting part of the setup for most people is the color scheme. I personally use Sonokai. Feel free to use your favorite color scheme, just make sure to also update the color scheme in the LazyVim config, like we did before.

Colorscheme

```
return {
  {
    "sainnhe/sonokai",
    priority = 1000,
    config = function()
      vim.g.sonokai_transparent_background = "1"
      vim.g.sonokai_enable_italic = "1"
      vim.g.sonokai_style = "andromeda"
      vim.cmd.colorscheme("sonokai")
    end,
  },
}
```

Language Server Protocols

To actually make our setup functional, we need language server protocols. They power most of our editor's features, such as code completion, syntax highlighting, and refactoring support. In our scenario, we'll be using Mason.nvim to manage installed LSP's. Mason is a package manager for LSP's, which allows us to easily install and manage LSP servers, DAP servers, linters, and formatters. We'll also define some default language servers to be installed on initial startup.

Mason NVIM

```
{
  "williamboman/mason.nvim",
  opts = function(_, opts)
    vim.list_extend(opts.ensure_installed, {
      "luacheck",
      "shellcheck",
      "shfmt",
      "tailwindcss-language-server",
      "typescript-language-server",
      "css-lsp",
    })
  end,
},
```

Language Server Protocols

We also need to make use of the Nvim LSP Config, which allows us to configure specific language servers. In our scenario, we're setting up the TypeScript, TailwindCSS, CSS, HTML and optionally Lua language servers.

LSP Config

```
{
  "neovim/nvim-lspconfig",
  opts = {
    inlay_hints = { enabled = true },
    ---@type lspconfig.options
    servers = {
      cssls = {},
      tailwindcss = {
        root_dir = function(...)
          return require("lspconfig.util").root_pattern(".git")(...)
        end,
      },
      tsserver = {
        root_dir = function(...)
          return require("lspconfig.util").root_pattern(".git")(...)
        end,
        single_file_support = false,
        settings = {
          typescript = {
            inlayHints = {
              includeInlayParameterNameHints = "literal",
              includeInlayParameterNameHintsWhenArgumentMatchesName
= false,
              includeInlayFunctionParameterTypeHints = true,
              includeInlayVariableTypeHints = false,
              includeInlayPropertyDeclarationTypeHints = true,
              includeInlayFunctionLikeReturnTypeHints = true,
              includeInlayEnumMemberValueHints = true,
            },
          },
          javascript = {
            inlayHints = {
              includeInlayParameterNameHints = "all",
              includeInlayParameterNameHintsWhenArgumentMatchesName
= false,
              includeInlayFunctionParameterTypeHints = true,
              includeInlayVariableTypeHints = true,
              includeInlayPropertyDeclarationTypeHints = true,
              includeInlayFunctionLikeReturnTypeHints = true,
              includeInlayEnumMemberValueHints = true,
            },
          },
        },
      },
      html = {},
    },
  },
},
```

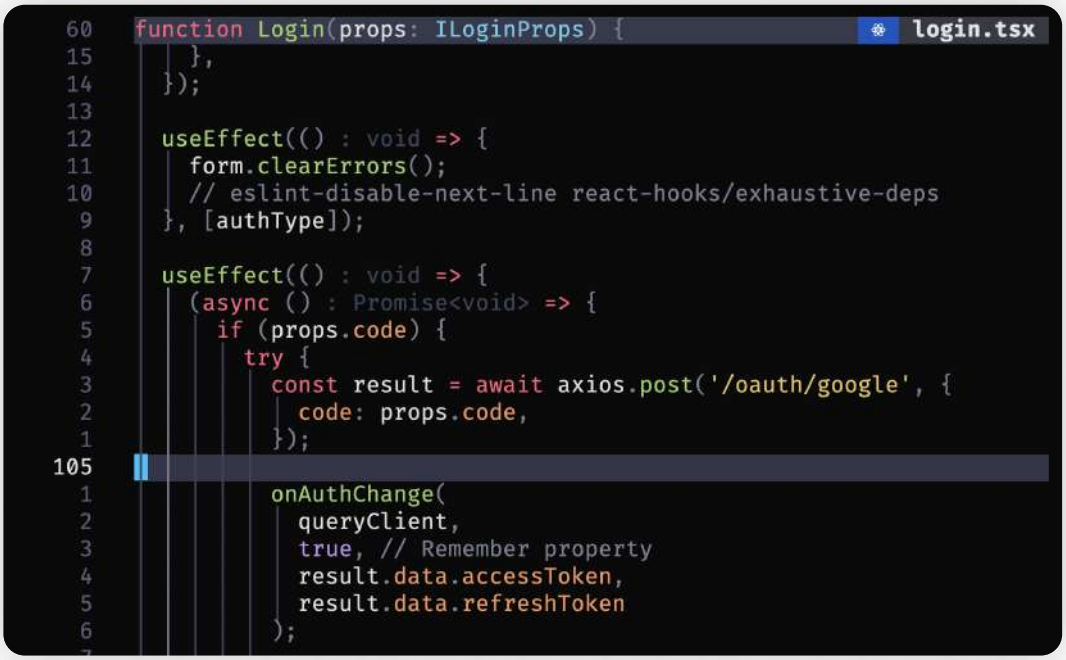
Language Server Protocols

For both TailwindCSS and Typescript, we're defining a custom function for locating the root of the project. The function allows us to find the .git folder, and use that as the root of the project.

We don't want TypeScript single file support, as we usually want to work with a magnitude of files which should be included in the context when developing.

Additionally, we're enabling Inlay Hints. Inlay hints are a new feature in NeoVim, which allows us to view additional information about our code, directly inline. For example, we can see the function's expected TypeScript types.

Here's an example from within my editor:



Language Server Protocols

Keep in mind that Inlay Hints only work with NeoVim versions 0.10 and above. At the time of writing, the latest stable NeoVim version is 0.9.5, meaning you might have to install the Nightly build of NeoVim, or build directly from HEAD in order to use Inlay Hints.

Finally, we'll add nvim-cmp, which provides that beautiful autocomplete dropdown. If you want icons next to your suggestions, make sure to also include the cmp-emoji plugin as a dependency.

Nvim CMP

```
{
  "nvim-cmp",
  dependencies = { "hrsh7th/cmp-emoji" },
  opts = function(_, opts)
    table.insert(opts.sources, { name = "emoji" })
  end,
},
```

Treesitter

While the LSP does the majority of the work, if we want our files to have proper syntax highlighting, indentation and folding, we need additional configurations for different file types. This is where Tree Sitter comes in. Tree Sitter builds a syntax tree for your files and updates it as you edit your code.

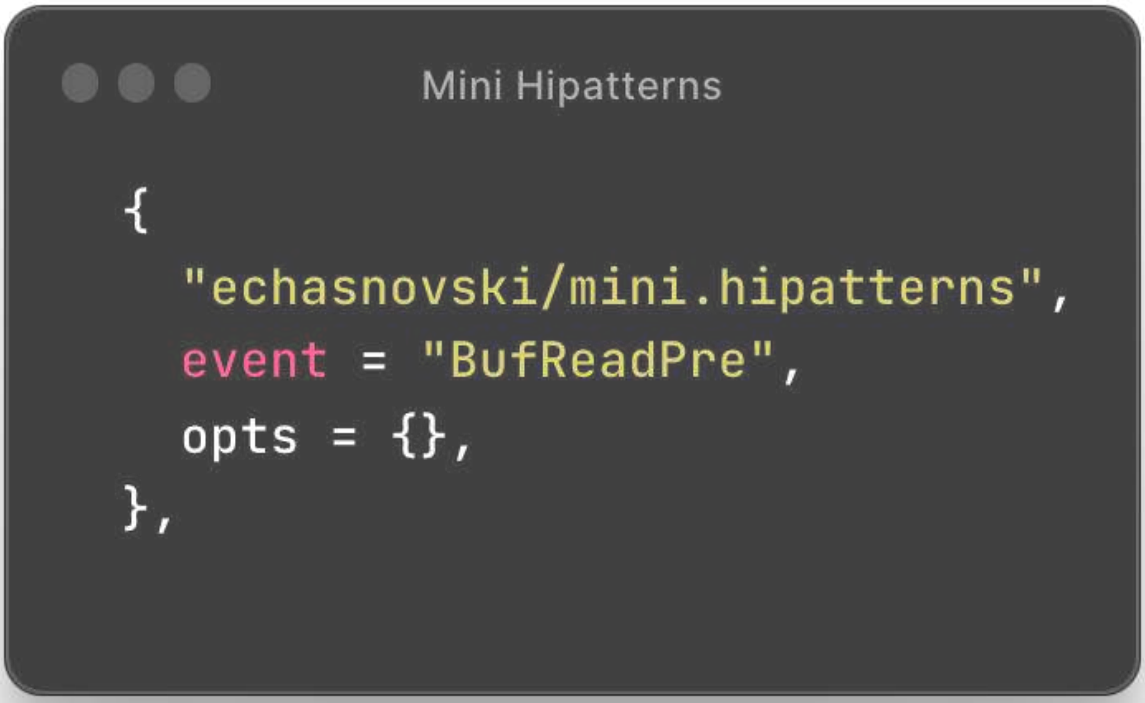
This syntax tree is what allows us to do cool stuff, like syntax highlighting (even though that's the most basic use case). It's an extremely powerful plugin, so for more advanced readers, you can explore the documentation for more info.

Treesitter

```lua
return {
  {
    "nvim-treesitter/nvim-treesitter",
    tag = "v0.9.1",
    opts = {
      ensure_installed = {
        "javascript",
        "typescript",
        "css",
        "gitignore",
        "graphql",
        "http",
        "json",
        "scss",
        "sql",
        "vim",
        "lua",
      },
      query_linter = {
        enable = true,
        use_virtual_text = true,
        lint_events = { "BufWrite", "CursorHold" },
      },
    },
  },
}
```

Mini Hipatterns

Mini Hipatterns is a cool plugin, a part of the Mini.nvim plugin library, that allows us to highlight patterns in text. Most useful to us, is the color highlighting, directly inline. Here's an example from within my editor:



Telescope

Telescope is a visual fuzzy finder, that allows us to quickly search for files, text, buffers, and much more. For now, let's follow the default setup, and also include the telescope-file-browser plugin as a dependency, as we'll be making use of it very soon.

Telescope

```
{
  "telescope.nvim",
  priority = 1000,
  dependencies = {
    {
      "nvim-telescope/telescope-fzf-native.nvim",
      build = "make",
    },
    "nvim-telescope/telescope-file-browser.nvim",
  },
  keys = { },
  config = function(_, opts)
    local telescope = require("telescope")
    local actions = require("telescope.actions")
    local fb_actions =
require("telescope").extensions.file_browser.actions

    opts.defaults = vim.tbl_deep_extend("force", opts.defaults, {
      wrap_results = true,
      layout_strategy = "horizontal",
      layout_config = { prompt_position = "top" },
      sorting_strategy = "ascending",
      winblend = 0,
      mappings = {
        n = {},
      },
    })

    telescope.setup(opts)
    require("telescope").load_extension("fzf")
    require("telescope").load_extension("file_browser")
  end,
},
```

Telescope

Let's start defining some keys. I focus a lot of my keybinds on the semicolon ";" key, and Telescope is no exception. We're adding these into the "keys" section of the Telescope setup. These mostly just call the proper function that executes Telescope when a key combination is pressed.

;f opens up the file search

;r opens up the text search

\\ opens a list of the current buffers

;; resumes the last Telescope search that you closed

Telescope

```
    {
      ";f",
      function()
        local builtin = require("telescope.builtin")
        builtin.find_files({
          no_ignore = false,
          hidden = true,
        })
      end,
      desc = "Lists files in your current working directory,
respects .gitignore",
    },
    {
      ";r",
      function()
        local builtin = require("telescope.builtin")
        builtin.live_grep()
      end,
      desc = "Search for a string in your current working directory
and get results live as you type, respects .gitignore",
    },
    {
      "\\\\",
      function()
        local builtin = require("telescope.builtin")
        builtin.buffers()
      end,
      desc = "Lists open buffers",
    },
    {
      ";;",
      function()
        local builtin = require("telescope.builtin")
        builtin.resume()
      end,
      desc = "Resume the previous telescope picker",
    },
```

Telescope

;e opens up a list of all diagnostics

;s opens up a TreeSitter syntax view

Telescope

```
      {
        ";e",
        function()
          local builtin = require("telescope.builtin")
          builtin.diagnostics()
        end,
        desc = "Lists Diagnostics for all open buffers or a specific
buffer",
      },
      {
        ";s",
        function()
          local builtin = require("telescope.builtin")
          builtin.treesitter()
        end,
        desc = "Lists Function names, variables, from Treesitter",
      },
```

Telescope

Finally, let's also set up the Telescope file browser, which allows for very quick file navigation. I use this to move around 90% of the time, if I don't know the exact file name I'm looking for.
We want to define some default view options, mainly not respecting git ignore to show all files, show hidden files, group files by type, and disabling the preview window.

Telescope

```
{
  "sf",
  function()
    local telescope = require("telescope")

    local function telescope_buffer_dir()
      return vim.fn.expand("%:p:h")
    end

    telescope.extensions.file_browser.file_browser({
      path = "%:p:h",
      cwd = telescope_buffer_dir(),
      respect_gitignore = false,
      hidden = true,
      grouped = true,
      previewer = false,
      initial_mode = "normal",
      layout_config = { height = 40 },
    })
  end,
  desc = "Open File Browser with the path of the current
buffer",
},
```

Telescope

Let's not forget to define some additional keybinds for the file browser in the "config" function of the Telescope setup. We map the capital N key to creating a new file, the "h" key to navigating to the parent directory, and we also add back the Control + d, and Control + u keybinds to quickly jump and and down the list.

Telescope

```
      opts.extensions = {
        file_browser = {
          theme = "dropdown",
          -- disables netrw and use telescope-file-browser in its
place
          hijack_netrw = true,
          mappings = {
            -- your custom insert mode mappings
            ["n"] = {
              -- your custom normal mode mappings
              ["N"] = fb_actions.create,
              ["h"] = fb_actions.goto_parent_dir,
              ["<C-u>"] = function(prompt_bufnr)
                for i = 1, 10 do
                  actions.move_selection_previous(prompt_bufnr)
                end
              end,
              ["<C-d>"] = function(prompt_bufnr)
                for i = 1, 10 do
                  actions.move_selection_next(prompt_bufnr)
                end
              end,
            },
          },
        },
      }
```

Disabling plugins

I personally do not like the dashboard upon opening NeoVim, and the status line as I don't use it often. For this reason, we'll disable the included plugins.

UI

```
{
  "nvimdev/dashboard-nvim",
  enabled = false,
},
{
  "nvim-lualine/lualine.nvim",
  enabled = false,
},
```

Noice.nvim

Noice.nvim is a plugin that replaces the messages, command line, and the pop up menu with nicer visual alternatives.
It's an amazing plugin by default,

but when you try and open up the definition of a variable or function that doesn't have one, an annoying "No information available" pop up spams your screen. For this reason, we'll define a function that suppresses this message. Credit where it's due, I got this from devaslife.

UI

```
{
  "folke/noice.nvim",
  opts = function(_, opts)
    table.insert(opts.routes, {
      filter = {
        event = "notify",
        find = "No information available",
      },
      opts = { skip = true },
    })
    local focused = true
    vim.api.nvim_create_autocmd("FocusGained", {
      callback = function()
        focused = true
      end,
    })
    vim.api.nvim_create_autocmd("FocusLost", {
      callback = function()
        focused = false
      end,
    })
    table.insert(opts.routes, 1, {
      filter = {
        cond = function()
          return not focused
        end,
      },
      view = "notify_send",
      opts = { stop = false },
    })

    opts.commands = {
      all = {
        -- options for the message history that you get with
`:Noice`
        view = "split",
        opts = { enter = true, format = "details" },
        filter = {},
      },
    }

    opts.presets.lsp_doc_border = true
  end,
},
```

Nvim Notify & Bufferline

We'll specify the notification timeout, background color, and view style. Additionally, we'll configure the bufferline plugin to be extremely minimal, which displays tabs as a GUI at the top of NeoVim.

UI

```
  {
    "rcarriga/nvim-notify",
    opts = {
      timeout = 5000,
      background_colour = "#000000",
      render = "wrapped-compact",
    },
  },

  -- buffer line
  {
    "akinsho/bufferline.nvim",
    event = "VeryLazy",
    keys = {
      { "<Tab>", "<Cmd>BufferLineCycleNext<CR>", desc = "Next tab" },
      { "<S-Tab>", "<Cmd>BufferLineCyclePrev<CR>", desc = "Prev tab"
},
    },
    opts = {
      options = {
        mode = "tabs",
        show_buffer_close_icons = false,
        show_close_icon = false,
      },
    },
  },
```

Incline.nvim

Incline.nvim is also a very neat plugin that displays the name of the current file in the top right corner of the window. We'll mostly keep the default configuration, while slightly altering the theme color. Your color scheme may also support specific colors for Incline, so check out your color scheme's documentation.

UI

```
{
  "b0o/incline.nvim",
  dependencies = {},
  event = "BufReadPre",
  priority = 1200,
  config = function()
    local helpers = require("incline.helpers")
    require("incline").setup({
      window = {
        padding = 0,
        margin = { horizontal = 0 },
      },
      render = function(props)
        local filename =
vim.fn.fnamemodify(vim.api.nvim_buf_get_name(props.buf), ":t")
        local ft_icon, ft_color = require("nvim-web-
devicons").get_icon_color(filename)
        local modified = vim.bo[props.buf].modified
        local buffer = {
          ft_icon and { " ", ft_icon, " ", guibg = ft_color, guifg =
helpers.contrast_color(ft_color) }
            or "",
          " ",
          { filename, gui = modified and "bold,italic" or "bold" },
          " ",
          guibg = "#363944",
        }
        return buffer
      end,
    })
  end,
},
```

LazyGit

As developers, an easy to use Git integration is a must. While terminal commands will do most of the time, having a visual overview of the git changes can save a lot of time. For this reason, we're adding LazyGit. Make sure to also install LazyGit on your system. If you're on MacOS, you can install it using brew install lazygit using Homebrew.

UI

```
{
  "kdheepak/lazygit.nvim",
  keys = {
    {
      ";c",
      ":LazyGit<Return>",
      silent = true,
      noremap = true,
    },
  },
  -- optional for floating window border decoration
  dependencies = {
    "nvim-lua/plenary.nvim",
  },
},
```

Dadbod

If you do any work with databases, Vim Dadbod UI is a must have. It's a plugin that allows you to connect to any database, perform queries and mutations on it, and save queries for later use. I personally defined a keybind that opens up Dadbod, while closing the Nvim Tree (which we will install next), and opens u Dadbod in a new tab. This allows me to keep my window layout, and play with the database in a separate tab.

UI

```
  {
    "kristijanhusak/vim-dadbod-ui",
    dependencies = {
      { "tpope/vim-dadbod", lazy = true },
      { "kristijanhusak/vim-dadbod-completion", ft = { "sql",
"mysql", "plsql" }, lazy = true },
    },
    cmd = {
      "DBUI",
      "DBUIToggle",
      "DBUIAddConnection",
      "DBUIFindBuffer",
    },
    init = function()
      -- Your DBUI configuration
      vim.g.db_ui_use_nerd_fonts = 1
    end,
    keys = {
      {

        "<leader>d",
        "<cmd>NvimTreeClose<cr><cmd>tabnew<cr><bar><bar>
<cmd>DBUI<cr>",
      },
    },
  },
```

Nvim Tree

Last but not least, let's add in Nvim Tree. While I mostly navigate with Telescope file search and the Telescope File browser, when working on new or large projects, I like to have a visual overview of where I'm at in the project.

We'll define some default view options, ignore the node_modules folder, make the "t" key open up the file in a new tab, and make sure that the NvimTree only opens up when we start up NeoVim with no file specified.

UI

```lua
{
  "nvim-tree/nvim-tree.lua",
  config = function()
    require("nvim-tree").setup({
      on_attach = function(bufnr)
        local api = require("nvim-tree.api")

        local function opts(desc)
          return {
            desc = "nvim-tree: " .. desc,
            buffer = bufnr,
            noremap = true,
            silent = true,
            nowait = true,
          }
        end

        -- default mappings
        api.config.mappings.default_on_attach(bufnr)

        -- custom mappings
        vim.keymap.set("n", "t", api.node.open.tab, opts("Tab"))
      end,
      actions = {
        open_file = {
          quit_on_open = true,
        },
      },
      sort = {
        sorter = "case_sensitive",
      },
      view = {
        width = 30,
        relativenumber = true,
      },
      renderer = {
        group_empty = true,
      },
      filters = {
        dotfiles = true,
        custom = {
          "node_modules/.*",
        },
      },
      log = {
        enable = true,
        truncate = true,
        types = {
          diagnostics = true,
          git = true,
          profile = true,
          watcher = true,
        },
      },
    })

    if vim.fn.argc(-1) == 0 then
      vim.cmd("NvimTreeFocus")
    end
  end,
},
```

Refactoring

For refactoring, I mainly use two plugins.
First one being Incremental rename, which allows us to rename a variable in place, and update all of it's occurrences instantly.
Secondly, by the legendary ThePrimeagen, refactoring.nvim, which allows us to visually select a code block, and either extract it, or inline it. This is a must have for any larger codebase.

Refactoring

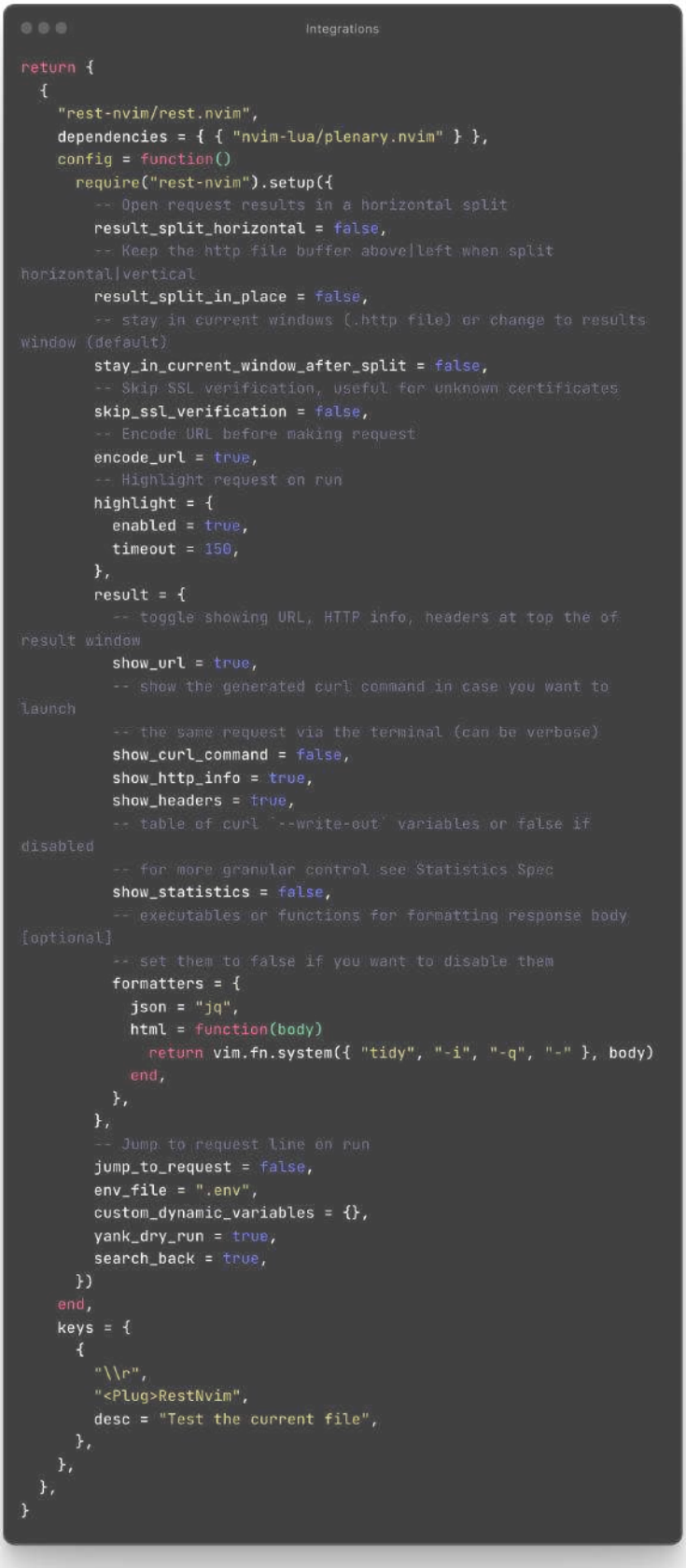
```lua
return {
  -- Incremental rename
  {
    "smjonas/inc-rename.nvim",
    cmd = "IncRename",
    keys = {
      {
        "<leader>rn",
        function()
          return ":IncRename " .. vim.fn.expand("<cword>")
        end,
        desc = "Incremental rename",
        mode = "n",
        noremap = true,
        expr = true,
      },
    },
    config = true,
  },

  -- Refactoring tool
  {
    "ThePrimeagen/refactoring.nvim",
    keys = {
      {
        "<leader>r",
        function()
          require("refactoring").select_refactor({
            show_success_message = true,
          })
        end,
        mode = "v",
        noremap = true,
        silent = true,
        expr = false,
      },
    },
    opts = {},
  },
}
```

Integrations

Since I work a lot with Rest API's, I often have to perform many API requests to validate their functionality, and to inspect the expected result. To make this process faster, we can integrate this process into NeoVim using Rest.nvim.

This is the default config.



Testing

Finally, since I also do a lot of testing in my workflow, I needed something to run my tests from NeoVim, especially targeted tests without having to modify jest.config.js each time I want to test out a different file or folder.
For this reason, we're adding in NeoTest, which is a general plugin that allows you to test many types of tests, based on the adapters you set up for it.
In our case, we're setting up Jest.

This is the default recommended configuration for LazyVim, so no need to worry about the specifics unless you're eager to highly customize the setup.

Testing

Now, let's actually configure the adapters. NeoTest Plenary is recommended, and no additional setup needed, but since I work with monorepos a lot, I need to modify NeoTest Jest to find the proper jest config file and root directory while working in a monorepo. If you don't work within monorepos, you can just use the default setup, like we did with Plenary above.

Testing

Let's also define some keybinds for NeoTest, once again focused around the semicolon key.
We should now be able to open up any Jest file, and execute tests directly from NeoVim.

Testing

```
    keys = {
      { ";tt", function()
require("neotest").run.run(vim.fn.expand("%")) end, desc = "Run File"
},
      { ";tr", function() require("neotest").run.run() end, desc =
"Run Nearest" },
      { ";tT", function() require("neotest").run.run(vim.loop.cwd())
end, desc = "Run All Test Files" },
      { ";tl", function() require("neotest").run.run_last() end, desc
= "Run Last" },
      { ";ts", function() require("neotest").summary.toggle() end,
desc = "Toggle Summary" },
      { ";to", function() require("neotest").output.open({ enter =
true, auto_close = true }) end, desc = "Show Output" },
      { ";tO", function() require("neotest").output_panel.toggle()
end, desc = "Toggle Output Panel" },
      { ";tS", function() require("neotest").run.stop() end, desc =
"Stop" },
    },
```

 Harpoon on GitHub

Extra #1 - Harpoon

Another plugin by the legendary ThePrimeagen - Harpoon! The plugins allows you to “harpoon” files, meaning add them to the harpoon list, and then quickly open those files using a simple GUI. Very useful if you’re working with multiple files constantly, and don’t want to get lost in a sea of tabs and split windows.

 Fugitive on GitHub

Extra #2 - Fugitive

If you don’t quite fancy the GUI nature of LazyGit, you can go the Fugitive route. It allows you to call git commands directly from NeoVim, and makes the results & outputs much cleaner and easier to work with. It also provides some aliases for commands which make your git work more efficient.

 Emmet-vim on GitHub

Extra #3 - Emmet-vim

Emmet Vim allows you to expand abbreviations, similar to Emmet. This is an extremely useful plugin when you're working with a lot of HTML or CSS, as you can write out the entire HTML tree in a single line, and have emmet generate it for you. You can also generate custom snippets to make your workflow even faster.

 LuaSnip on GitHub

Extra #4 - LuaSnip

LuaSnip allows you to define and use custom snippets, with the cursor automatically jumping to the next configurable object in the snippet. Highly useful when writing similar code often, like classes, functions, useEffect hooks, etc. It's included with LazyVim, just make sure to configure it.

 Vim Abolish on GitHub

Extra #5 - Vim Abolish

If you're frustrated with text replacement in Vim, Vim Abolish aims to help you with that. Another plugin by the man himself, tpope, that allows you to replace while keeping in mind the capitalization of the search and plural variations. It also has some cool additional functionality you can check out for string manipulation.

 VimSurround on GitHub

Extra #6 - Vim Surround

Vim Surround allows you to surround pieces of text with a character, or character combination. Adding quotation marks, HTML tags, brackets, and asterisks has never been easier!

Thank you!

Thank you for downloading this resource! I hope you find it valuable. For more resources like this one, be sure to keep an eye on https://sindo.dev as new content is regularly added.

Interested in working together?

Website: ***sindo.dev***
Email: ***sindoonyt@gmail.com***