

# Neovim no mundo da Lua

---

**5 de agosto de 2022 - 18 min**

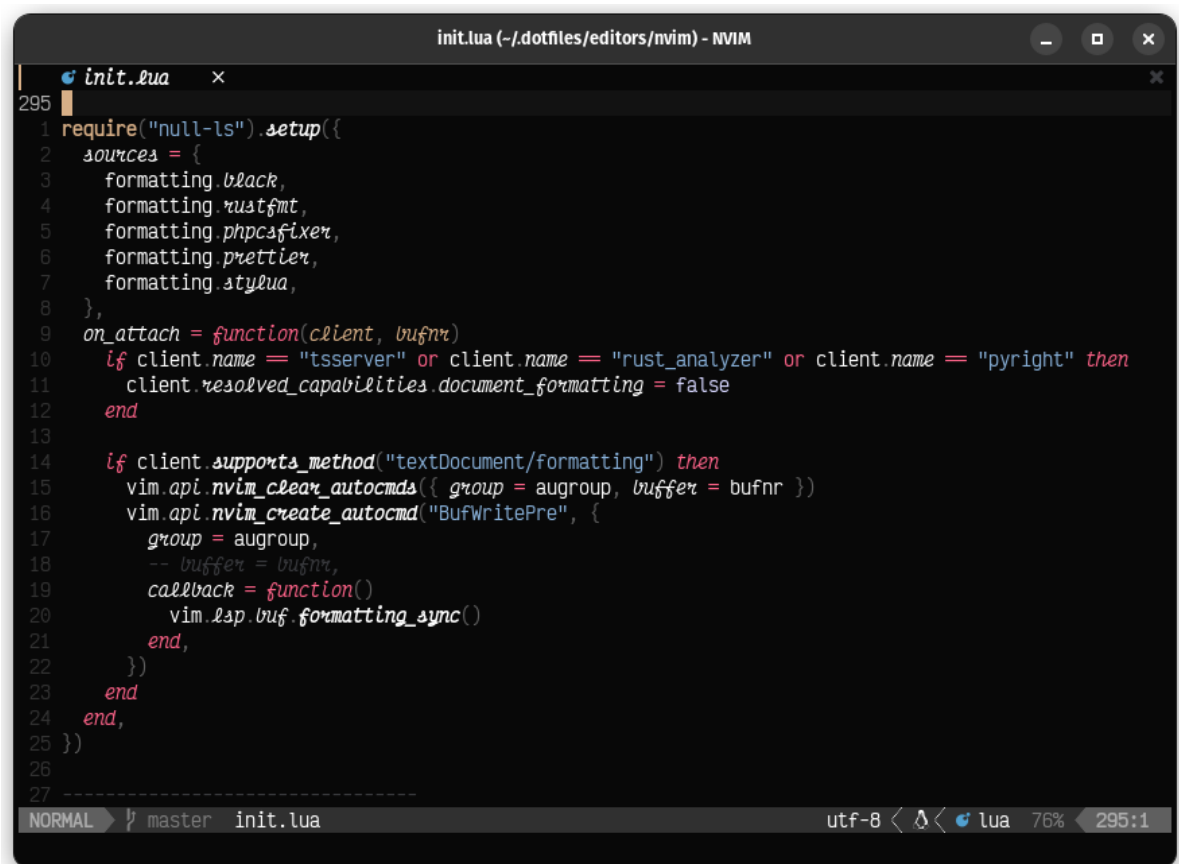
[fonte](#)

Já faz um certo tempo que uso o Neovim como o meu principal editor de código, e até o momento, não mudei de ideia. Escrevi alguns artigos a respeito deste editor e [um deles](#) foi sobre a criação de um arquivo de configuração simples, que funcione tanto para o Neovim, quanto para o Vim. Este ainda funciona perfeitamente, mas como tudo no mundo da informática, sempre há outra forma de atingir os mesmos objetivos. Um desses modos é fazer a configuração do Neovim usando a linguagem Lua.

Recebi alguns pedidos para fazer o guia de uma configuração em Lua, mas procrastinei até realmente precisar configurar o Neovim do zero novamente. A oportunidade veio recentemente, quando precisei configurar um Ryzentosh (Hackintosh com processador Ryzen) para questões de trabalho, e lá estava um Neovim padrão para configurar. Antes de entrar no guia propriamente dito, preciso falar sobre alguns detalhes.

## A linguagem Lua

---



```
init.lua (-/.dotfiles/editors/nvim) - NVIM
init.lua x
295
1 require("null-ls").setup({
2   sources = {
3     formatting.black,
4     formatting.rustfmt,
5     formatting.phpcsfixer,
6     formatting.prettier,
7     formatting.stylua,
8   },
9   on_attach = function(client, bufnr)
10    if client.name == "tsserver" or client.name == "rust_analyzer" or client.name == "pyright" then
11      client.resolved_capabilities.document_formatting = false
12    end
13
14    if client.supports_method("textDocument/formatting") then
15      vim.api.nvim_clear_autocmds({ group = augroup, buffer = bufnr })
16      vim.api.nvim_create_autocmd("BufWritePre", {
17        group = augroup,
18        -- buffer = bufnr,
19        callback = function()
20          vim.lsp.buf.formatting_sync()
21        end,
22      })
23    end
24  end,
25 })
26
27 -----
NORMAL master init.lua utf-8 lua 76% 295:1
```

### Exemplo de código em Lua

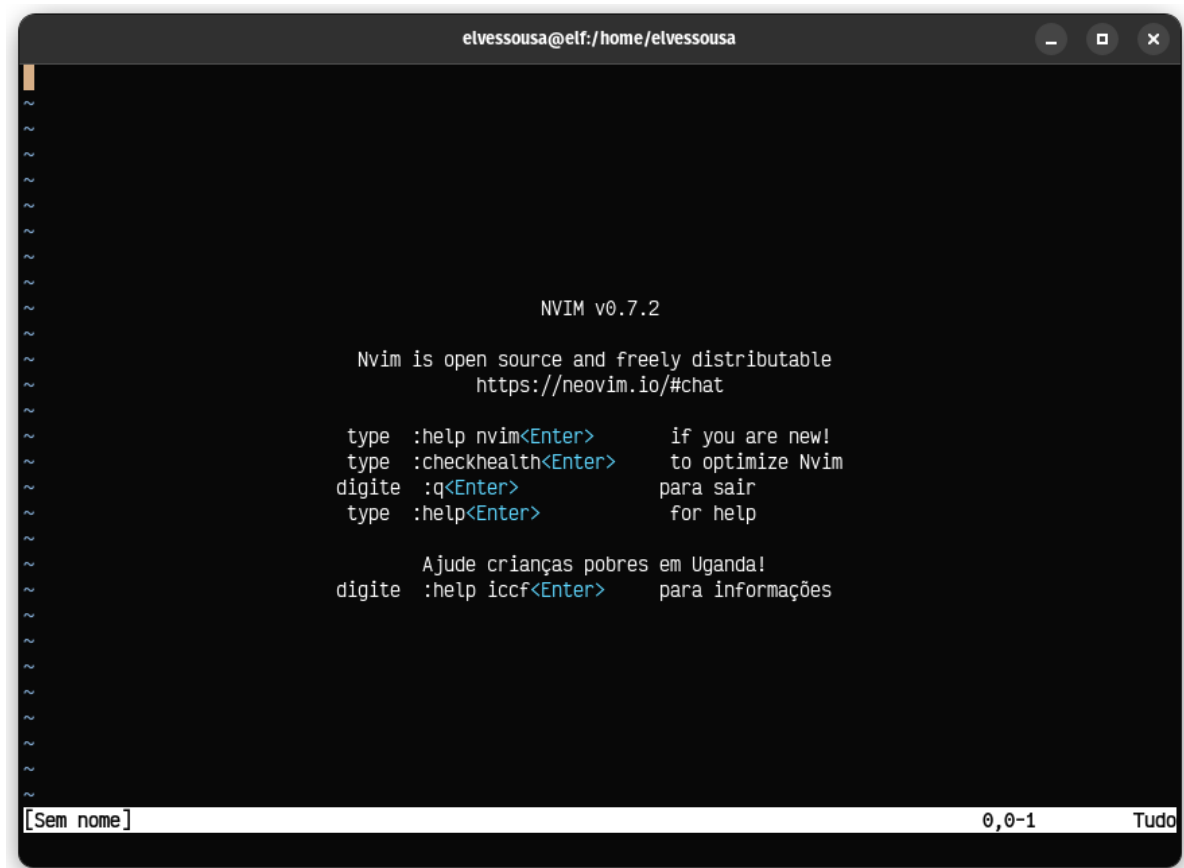
Lua é uma linguagem de programação criada pelos brasileiros Roberto Ierusalimschy, Waldemar Celes e Luiz Henrique de Figueiredo na PUC/RJ. É uma linguagem de alto nível, multiparadigma, leve, com tipagem dinâmica e coleta de lixo automática.

Ela foi feita para automatizar e estender aplicações mais complexas, como projetos da Petrobrás, motores de jogos, sistemas embarcados, e outras aplicações como o Neovim.

Como ser uma linguagem satélite sempre esteve em seus objetivos, "Lua" é um nome mais que apropriado. A facilidade em aprendê-la e sua velocidade considerável tem sido pontos positivos em sua adoção.

Este artigo não irá te tornar um mestre na linguagem, nem é este o objetivo. Mas você saberá o suficiente para configurar o Neovim.

## Sobre o arquivo de configuração



## Neovim

No `init.vim` que fiz em [outro artigo](#), foi possível criar um único arquivo que contém tudo o que precisa ser configurado para ter uma experiência confortável com o Neovim. Já com o `init.lua` que será criado aqui, isso também é possível, e é o que proponho neste artigo. Em outra oportunidade iremos separá-lo em vários arquivos, como é o padrão de mercado.

Se você está começando no Neovim agora, não tente seguir este guia. Comece com o arquivo de configuração em VimScript criado no [outro artigo](#), e assim que estiver mais confortável com o ambiente, volte aqui!

## Arquivo de configuração

Para começar, crie um arquivo `init.lua` no diretório `.config/nvim`, que se encontra na pasta do seu usuário:

```
$ cd ~/.config/nvim/  
$ touch init.lua
```

# Configuração das opções

O objetivo é basicamente traduzir o `init.vim` que já tenho para a linguagem Lua.

Em VimScript, cada opção é configurada com a palavra reservada `set`. Em Lua, usa-se `vim.opt.[nome-da-opção]`, que, convenhamos, não é muito legível se repetido diversas vezes. Então, resolvi alocar o `vim.opt` em uma variável local chamada `set` (uau!). Assim tudo fica mais familiar e legível.

A seguir as opções que uso:

```
-----  
-- Opções  
-----  
local set = vim.opt  
  
set.background = "dark"  
set.clipboard = "unnamedplus"  
set.completeopt = "noinsert,menuone,noselect"  
set.cursorline = true  
set.expandtab = true  
set.foldexpr = "nvim_treesitter#foldexpr()"  
set.foldmethod = "manual"  
set.hidden = true  
set.inccommand = "split"  
set.mouse = "a"  
set.number = true  
set.relativenumber = true  
set.shiftwidth = 2  
set.smarttab = true  
set.splitbelow = true  
set.splitright = true  
set.swapfile = false  
set.tabstop = 2  
set.termguicolors = true  
set.title = true  
set.timeoutlen = 0  
set.updatetime = 250  
set.wildmenu = true
```

```
set.wrap = true
```

Se você já usou o Neovim, provavelmente já conhece várias dessas opções, mas, caso não conheça, eis a explicação de cada uma:

- `background=dark`: aplica o conjunto de cores para telas escuras. Não somente o fundo da tela, como pode parecer.
- `clipboard=unnamedplus`: habilita a área de transferência entre o Neovim e os demais programas do sistema.
- `completeopt`: modifica o comportamento do menu de auto-completar para se comportar mais como uma IDE.
- `cursorline`: destaca a linha atual no editor.
- `expandtab`: transforma tabulações em espaços.
- `foldexpr` e `foldmethod`: estas opções foram colocadas para melhorar o comportamento da supressão de código no TreeSitter.
- `hidden`: esconde buffers não usados.
- `inccommand=split`: mostra substituições em uma divisão da janela, antes de aplicar no arquivo.
- `mouse=a`: permite o uso do mouse.
- `number`: mostra o número das linhas na lateral.
- `relativenumber`: mostra as linhas a partir da atual. Útil para auxiliar em comandos que usam mais linhas.
- `shiftwidth=2`: quantidade de espaços ao indentar o texto.
- `splitbelow` e `splitright`: configura o comportamento da divisão da tela com o comando `:split` (dividir a tela horizontalmente) e `:vsplit` (verticalmente). Neste caso, as telas sempre se dividirão abaixo da tela atual e à direita.
- `swapfile = false`: inibe a criação de arquivos `.swp` do Vim.
- `tabstop=2`: número de espaços para as tabulações.
- `termguicolors`: amplia o número de cores utilizáveis, caso o emulador de terminal suporte.
- `title`: mostra o título do arquivo.
- `ttimetypelen=0`: tempo em milissegundos para aceitar comandos.
- `updatetime`: tempo em milissegundos que os servidores de linguagem usam para verificar erros.

- `wildmenu`: mostra um menu mais avançado para sugestões de auto-completar.

Para vê-las em ação, basta sair do Neovim e entrar de novo (você sabe fazer isso, não é?), ou entrar no modo de comando `shift + :` e escrever `luafile %`. Isto irá ler o arquivo `init.lua` e aplicar as opções na instância atual.

## Sintaxe

Para adicionar suporte à sintaxe automática para os arquivos abertos:

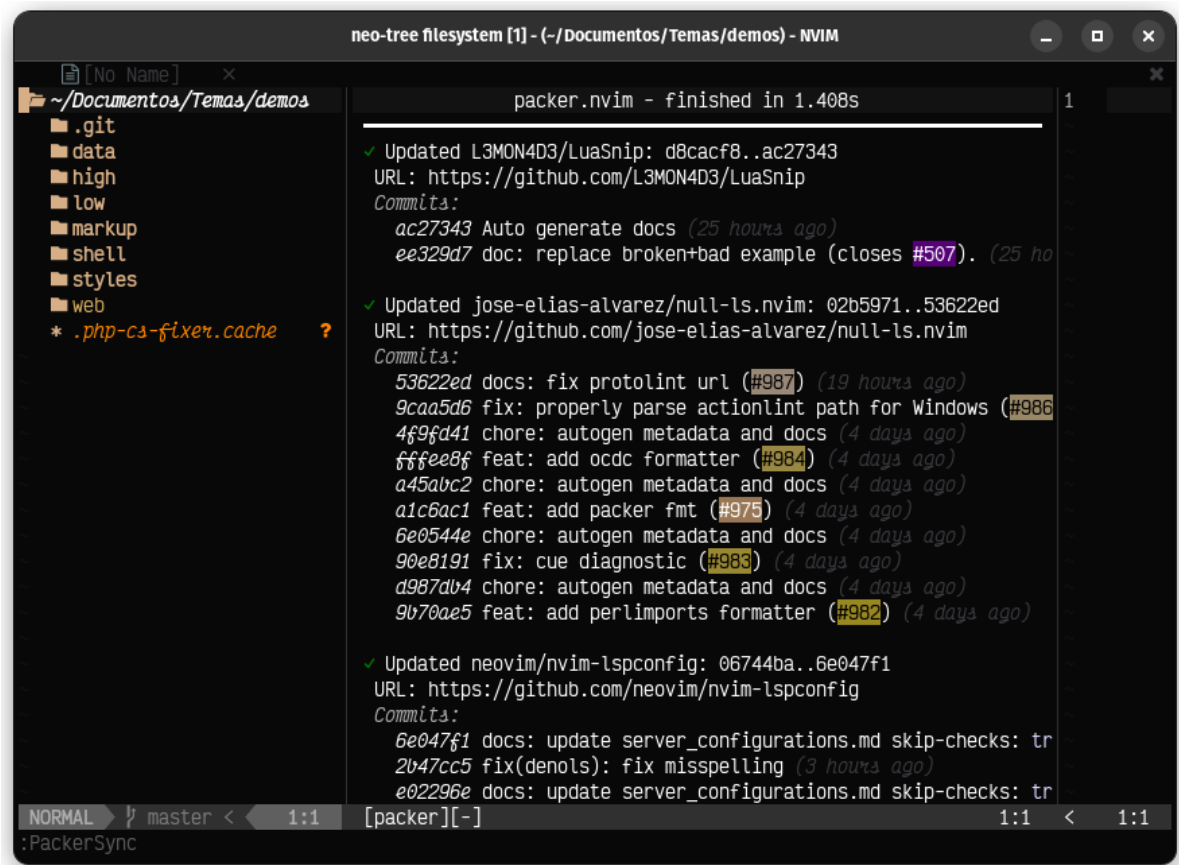
```
vim.cmd([[
  filetype plugin indent on
  syntax on
]])
```

Note o `vim.cmd`: ele possibilita que seja usado VimScript no arquivo `.lua`. Neste caso, é uma forma bastante prática de escrever menos código.

---

## Extensões

### Gerenciador de plugins



### *Packer: Gerenciador de extensões para o Neovim escrito em Lua*

Para instalar plugins, é necessário instalar um gerenciador primeiro. O que será usado neste exemplo é o **Packer**, feito em Lua e suporta, obviamente, a configuração nesta linguagem.

Se você está usando macOS ou Linux, basta executar este comando no terminal:

```
$ git clone --depth 1 https://github.com/wbthomason/packer.nvim\
~/.local/share/nvim/site/pack/packer/start/packer.nvim
```

Após reiniciar o Neovim, você terá uma série de comandos que irão facilitar muito a instalação, atualização e remoção de plugins.

Para acrescentá-lo no nosso arquivo de configuração basta adicionar as seguintes linhas:

```

-----
-- Plugins
-----

local packer = require("packer")

-- Inclusão do packer.nvim no Neovim
vim.cmd([[packadd packer.nvim]])

packer.startup(function()
  -- Plugins são listados aqui
end)

```

Note o `require("packer")`: essa é a maneira que se importa arquivos em Lua. Para quem já usou NodeJS, isto é bem familiar. Adiciona-se após isto um comando em VimScript para que o Neovim reconheça a existência do pacote do Packer.

Para acrescentar extensões a serem instaladas, basta incluir `use` "usuario-no-github/repositório", na função anônima em `packer.startup()`. O primeiro plugin a ser adicionado será o próprio Packer:

```

packer.startup(function()
  -- Gerenciador de plugins
  use("wbthomason/packer.nvim")
end)

```

Desta forma, podemos atualizar o Packer assim como os demais plugins.

## Plugins a serem usados nesta configuração

Eis a lista de extensões:

```

packer.startup(function()
  -- Auto-completar
  use("hrsh7th/cmp-buffer")
  use("hrsh7th/cmp-cmdline")
  use("hrsh7th/cmp-nvim-lsp")
  use("hrsh7th/cmp-path")

```



```

use("hrsh7th/nvim-cmp")
-- Motor de snippets
use("L3MON4D3/LuaSnip")
use("saadparwaiz1/cmp_luasnip")
-- Formatação
use("jose-elias-alvarez/null-ls.nvim")
-- Servidor de linguagens
use("neovim/nvim-lspconfig")
use("williamboman/nvim-lsp-installer")
-- Analisador de sintaxe
use("nvim-treesitter/nvim-treesitter")
-- Gerenciador de plugins
use("wbthomason/packer.nvim")
-- Utilitários
use("windwp/nvim-autopairs")
use("norcalli/nvim-colorizer.lua")
use("lewis6991/gitsigns.nvim")
-- Dependências
use("nvim-lua/plenary.nvim")
use("kyazdani42/nvim-web-devicons")
use("MunifTanjim/nui.nvim")
-- Navegador de arquivos
use("nvim-telescope/telescope.nvim")
-- Interface
use("akinsho/bufferline.nvim")
use({ "nvim-neo-tree/neo-tree.nvim", branch = "v2.x" })
use("nvim-lualine/lualine.nvim")
-- Tema
use("elvessousa/sobrio")
end)

```

Segue a explicação breve da função de cada um deles.

## Sugestões de preenchimento de código

- **Neovim CMP:** motor para as sugestões de código;
- **CMP Buffer:** adiciona sugestões com base no texto digitado nos arquivos abertos (buffers).
- **CMP CmdLine:** sugestões para a linha de comando no Neovim.
- **CMP nvim-lsp:** sugestões de código com base no servidor da linguagem de programação atual.

- **CMP Path:** sugestões para caminhos de pastas.

## Motor de snippets

- **LuaSnip** e **Cmp LuaSnip:** necessários para o funcionamento do Neovim CMP

## Formatação

- **Null LS:** provê ferramentas diversas, como diagnóstico e formatação de código.

## Servidor de linguagens

- **Neovim LSP Config:** disponibiliza o uso dos servidores de linguagem pelo Neovim.
- **Neovim LSP Installer:** instala servidores de linguagem pelo próprio editor.

## Analizador de sintaxe

- **Treesitter:** analisador de sintaxe.

## Gerenciador de plugins

- **Packer:** gerenciador de extensões.

## Utilitários

- **Neovim Autopairs:** fecha automaticamente parênteses, colchetes e chaves.
- **Neovim Colorizer:** exibe cores HEX, RGB ou HSL no editor.
- **Neovim GitSigns:** mostra alterações do git na lateral

## Dependências

- **Plenary:** framework para plugins em Lua para Neovim. Necessário para que algumas extensões funcionem.
- **Neovim Web Devicons:** habilita o uso de fontes de ícones nos plugins. Detalhe: você precisa ter uma fonte "Nerd Font" instalada no sistema e aplicada no seu emulador de terminal.

- **NUI:** biblioteca de componentes para interface, usada por alguns plugins.

## Navegador de arquivos

- **Telescope:** buscador de arquivos.

## Interface

- **Bufferline:** barra de abas.
- **Neo Tree:** árvore de diretórios e arquivos. Alternativa ao NetRW.
- **Lualine:** barra de estado, semelhante ao Airline.

## Tema

- **Sobrio:** tema feito por mim e que uso diariamente, agora com suporte ao TreeSitter.

## Instalação das extensões

Para instalar, como fizemos da outra vez com o VimPlug, é necessário usar um comando para fazer a instalar. Porém, ao contrário do VimPlug, um só comando do Packer é responsável por remover, adicionar ou atualizar as extensões: `:PackerSync`.

Ao usar este comando, uma divisão lateral irá aparecer e mostrar as alterações que o Packer fez em sua instalação do Neovim. Lembre-se de sempre usar este comando ao adicionar ou remover plugins de sua lista.

## Inicialização das extensões

Além de instalar os arquivos com o Packer, é necessário inicializar os plugins para que possam ser usados no Neovim. Alguns funcionam sem opções, outros precisam de mais configurações. Para ver algumas extensões em funcionamento, adicionaremos as mais simples primeiro:

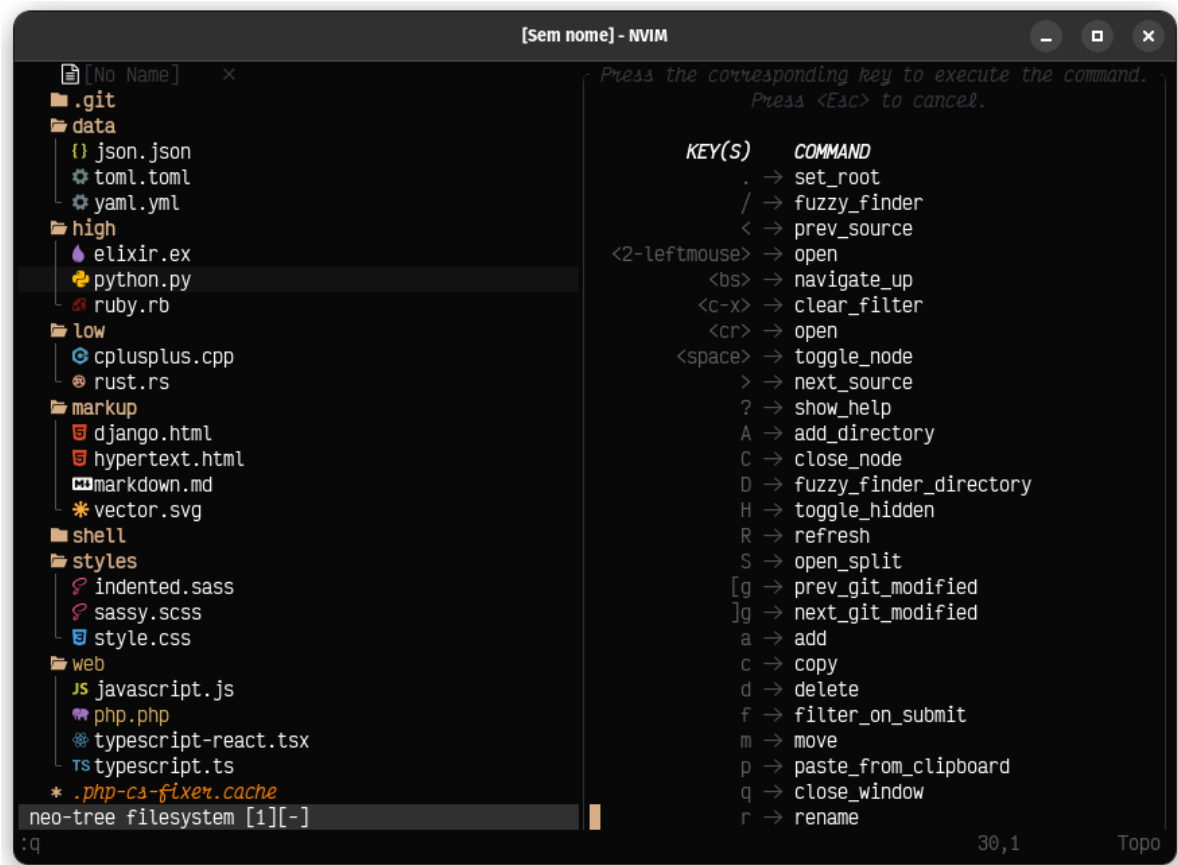
```
-----  
-- Plugins
```

```
-----  
-- Autopairs  
require("nvim-autopairs").setup({  
    disable_filetype = { "TelescopePrompt" },  
})  
  
-- Colorizer  
require("colorizer").setup()  
  
-- Git signs  
require("gitsigns").setup()  
  
-- Bufferline  
require("bufferline").setup()  
  
-- Lualine  
require("lualine").setup()
```

Este é um padrão. Geralmente os plugins são inicializados com `require("nome-do-plugin").setup()`.

Note que no plugin *Autopairs*, eu coloquei uma opção para desativar o funcionamento dele na janela do *Telescope*, para evitar comportamentos indesejados.

## Neo Tree



## Neo Tree: Árvore de diretórios

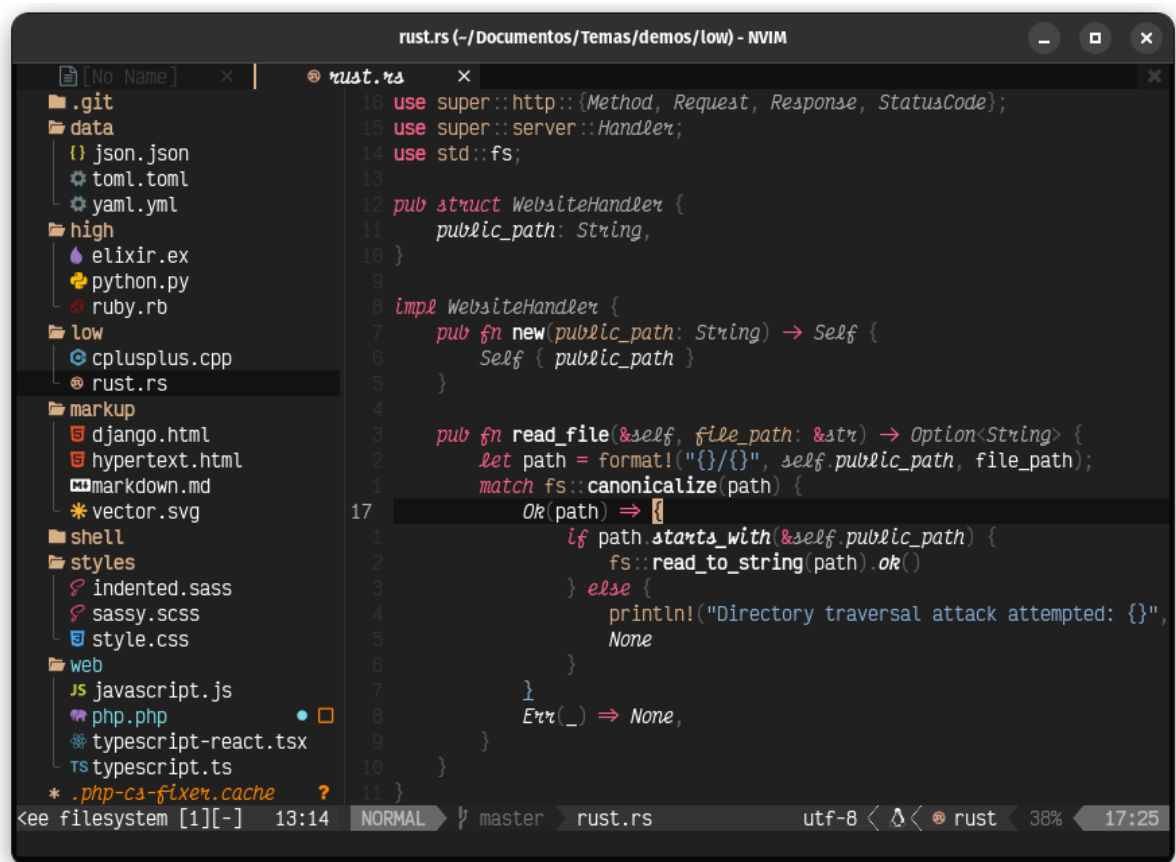
Neo Tree é um substituto ao NetRW que vem por padrão com o Neovim. Por padrão ele já funciona bem, mas recomendo adicionar as opções abaixo:

```
-- Neo tree
require("neo-tree").setup({
  close_if_last_window = false,
  enable_diagnostics = true,
  enable_git_status = true,
  popup_border_style = "rounded",
  sort_case_insensitive = false,
  filesystem = {
    filtered_items = {
      hide_dotfiles = false,
      hide_gitignored = false,
    },
  },
  window = { width = 30 },
})
```

Explicação das opções:

- `close_if_last_window = false`: mantém o Neo Tree visível caso não haja mais arquivos abertos.
- `enable_diagnostics = true`: exibe se há erros ou avisos no arquivo, conforme o servidor de linguagem.
- `enable_git_status = true`: exibe informações de alteração do Git, caso esteja em um repositório.
- `popup_border_style = "rounded"`: arredonda a borda dos diálogos.
- `sort_case_insensitive = false`: ignora maiúsculas e minúsculas ao ordenar arquivos.
- `hide_dotfiles = false`: mostra arquivos ocultos.
- `hide_gitignored = false`: mostra arquivos ignorados no controle de versão.
- `window = { width = 30 }`: define a largura da divisão do Neo Tree em 30% da tela.

## Esquema de cor

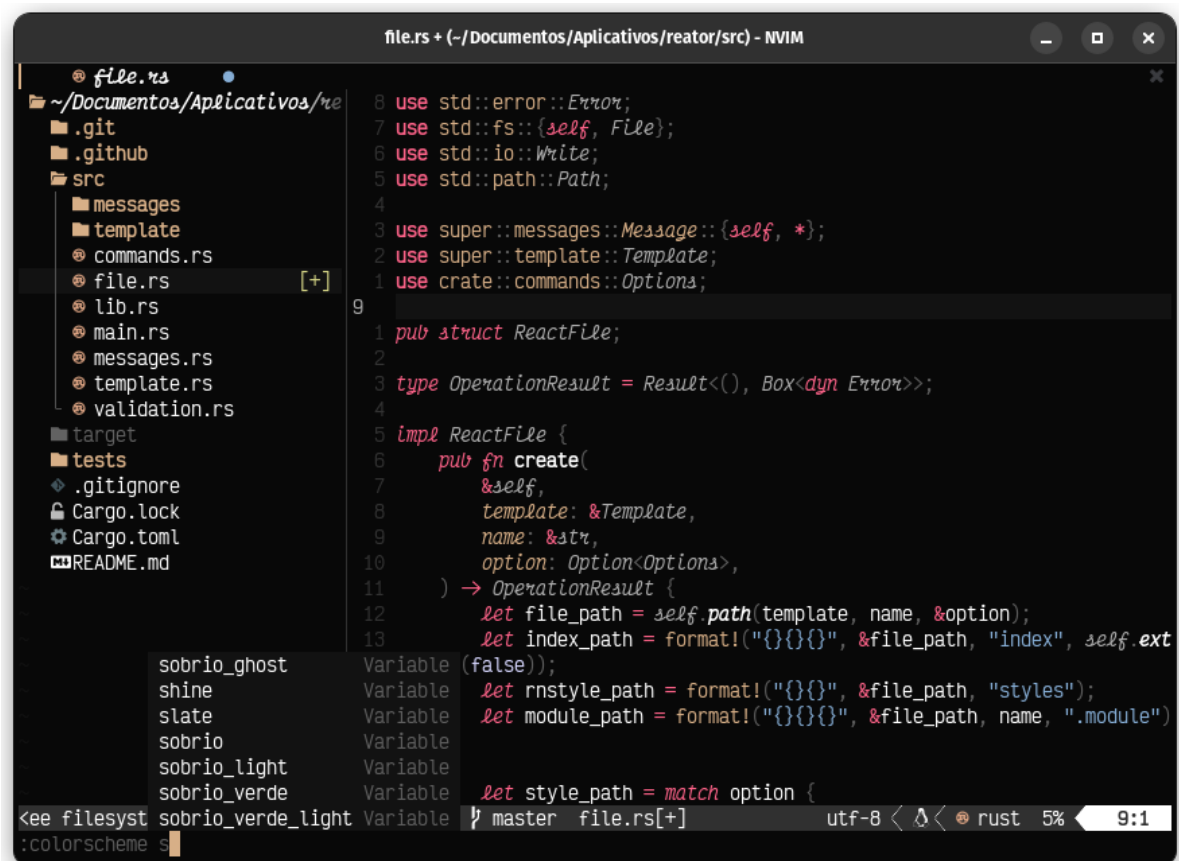


*Sobrio: O tema que fiz e uso no dia-a-dia*

Dependendo do tema que você utilizar, a configuração irá variar. Se o tema que você escolheu tiver configuração disponível em Lua, o processo é muito semelhante aos dos plugins. No caso do meu tema, Sobrio, eu somente o acrescentei no `vim.cmd` que citei no começo deste artigo:

```
-----  
-- Esquema de cores e sintaxe  
-----  
vim.cmd([[  
    filetype plugin indent on  
    syntax on  
    colorscheme sobrio  
]])
```

## Variações



*Sobrio Ghost: versão do tema Sobrio, com fundo transparente*

Este tema que fiz tem outras variantes, como uma versão clara (Sobrio Light), com cores alternativas (Sobrio Verde) e com fundo transparente (Sobrio Ghost).

# Configuração da sintaxe

---

TreeSitter é uma extensão responsável por cuidar de como a sintaxe é apresentada na tela. O resultado geralmente é bem melhor que o analisador de sintaxe que vem por padrão no Neovim.

Segue abaixo uma configuração compatível com o meu fluxo de trabalho:

```
-----  
-- Sintaxe  
-----  
require("nvim-treesitter.configs").setup({  
  ensure_installed = {  
    "bash",  
    "c",  
    "cpp",  
    "css",  
    "elixir",  
    "fish",  
    "graphql",  
    "html",  
    "javascript",  
    "json",  
    "lua",  
    "markdown",  
    "markdown_inline",  
    "php",  
    "python",  
    "regex",  
    "ruby",  
    "rust",  
    "scss",  
    "sql",  
    "toml",  
    "tsx",  
    "typescript",  
    "vim",  
    "yaml",  
  },  
  highlight = { enable = true },  
})
```



```
indent = { enable = true },
autotag = {
  enable = true,
  filetypes = {
    "html",
    "javascript",
    "javascriptreact",
    "svelte",
    "typescript",
    "typescriptreact",
    "vue",
    "xml",
  },
},
})
```

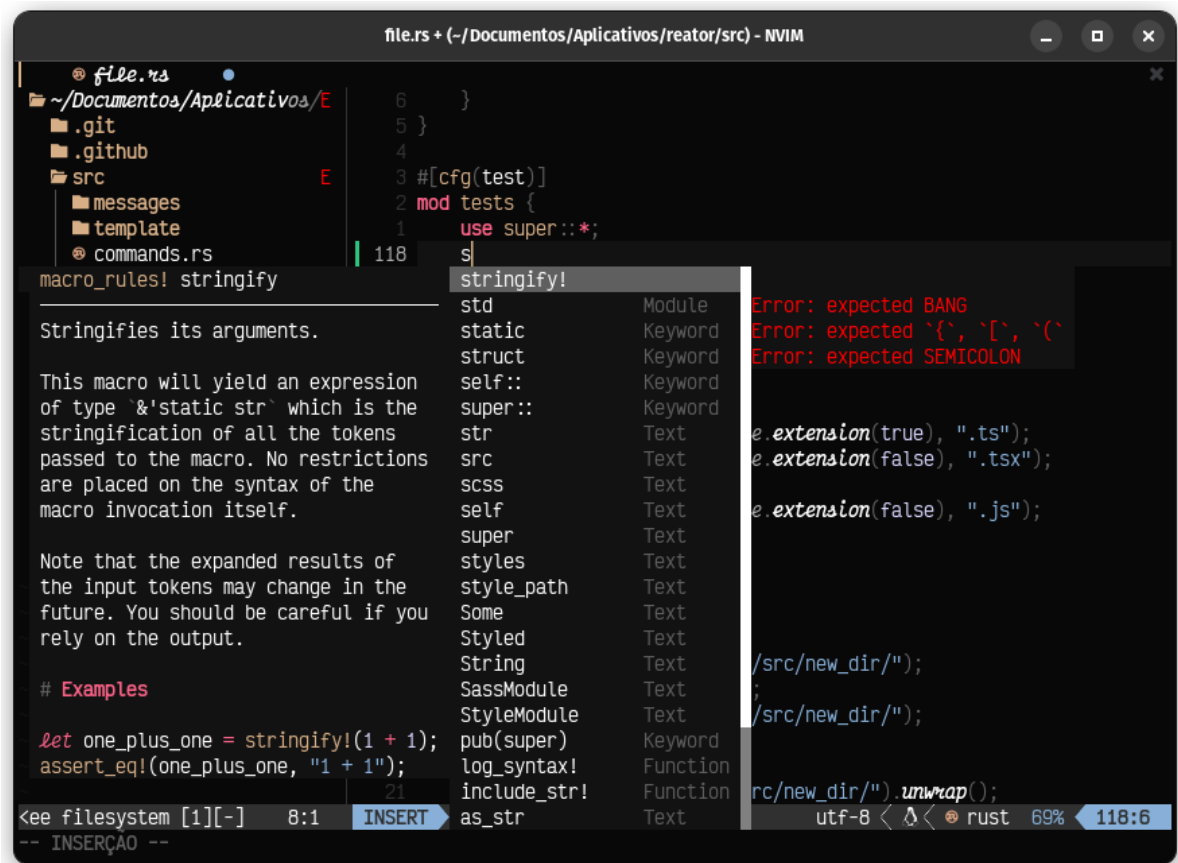
Breve explicação das opções:

- `ensure_installed`: lista de linguagens para serem instaladas no TreeSitter. Se quiser, use apenas a opção `all` e o TreeSitter usará todas as linguagens que ele tem suporte, mesmo as que você jamais usou ou usará.
- `highlight`: habilita a colorização do TreeSitter.
- `indent`: habilita a indentação do código.
- `autotag`: interpreta as tags de XML ou HTML nas linguagens listadas em `filetypes`.

---

## Sugestões de código

---



## *nvim-cmp*

Neovim CMP ou nvim-cmp é a extensão que habilita as sugestões de código no Neovim. Para ele funcionar, várias outras extensões relacionadas precisam estar presentes.

```
-----
-- Auto-completar
-----

local cmp = require("cmp")

cmp.setup({
  snippet = {
    expand = function(args)
      require("luasnip").lsp_expand(args.body)
    end,
  },
  mapping = cmp.mapping.preset.insert({
    ["<C-b>"] = cmp.mapping.scroll_docs(-4),
    ["<C-f>"] = cmp.mapping.scroll_docs(4),
    ["<C-Space>"] = cmp.mapping.complete(),
    ["<C-e>"] = cmp.mapping.abort(),
    ["<CR>"] = cmp.mapping.confirm({ select = true }),
  }),
})
```

```

sources = cmp.config.sources({
  { name = "nvim_lsp" },
  { name = "luasnip" },
  { name = "buffer" },
  { name = "path" },
}),
})

-- Configuração para tipos específicos de arquivo
cmp.setup.filetype("gitcommit", {
  sources = cmp.config.sources({
    { name = "cmp_git" },
  }, {
    { name = "buffer" },
  }),
})

-- Auto-completar do modo de comando
cmp.setup.cmdline("/", {
  mapping = cmp.mapping.preset.cmdline(),
  sources = { { name = "buffer" } },
})

cmp.setup.cmdline(":", {
  mapping = cmp.mapping.preset.cmdline(),
  sources = cmp.config.sources({
    { name = "path" },
  }, {
    { name = "cmdline" },
  }),
})

```

A maior parte deste código, eu inseri conforme a sugestão disponível no próprio repositório do autor. A única coisa que fiz foi adaptá-lo para uso com o motor de sugestões *LuaSnip*.

Na opção `snippet`, configura-se qual motor de sugestões será usado. Em `mapping`, adicionam-se atalhos para os comandos do CMP. Em `sources`, você ordena as fontes de sugestão por prioridade, no caso do código acima, em ordem de prioridade estão:

1. Sugestões da linguagem de programação
2. Códigos pré-prontos do LuaSnip
3. Palavras escritas no arquivo atual
4. Endereços de diretórios do seu computador

O bloco `cmp.setup.filetypes` permite que você configure o comportamento do Neovim CMP para um determinado tipo de arquivo. Neste caso, foi configurada regras para uso do `cmp_git` em operações relacionadas ao Git.

Já `cmp.setup.cmdline` configura as opções para o modo de comandos do Neovim. O fato de haver dois blocos é porque comandos podem começar com os caracteres `/` ou `:` e em cada caso as prioridades de sugestões mudam.

---

## Formatação de código

---

*NullLS* é uma extensão que disponibiliza vários serviços com base no servidor de linguagens embutido no Neovim. Um desses serviços é a formatação de código.

Por padrão, nada é feito. É necessário adicionar algumas configurações para que o NullLS entre em ação. Segue abaixo uma configuração exemplo para quem usa Python, Rust, PHP, JavaScript/TypeScript, HTML, CSS/SCSS e Lua.

```
-----  
-- Formatação  
-----  
  
local diagnostics = require("null-ls").builtins.diagnostics  
local formatting = require("null-ls").builtins.formatting  
local augroup = vim.api.nvim_create_augroup("LspFormatting", {})  
  
require("null-ls").setup({  
  sources = {  
    formatting.black,  
    formatting.rustfmt,  
    formatting.phpcsfixer,  
    formatting.prettier,
```

```

        formatting.stylua,
    },
    on_attach = function(client, bufnr)
        if client.name == "tsserver" or client.name ==
"rust_analyzer" or client.name == "pyright" then
            client.resolved_capabilities.document_formatting = false
        end

        if client.supports_method("textDocument/formatting") then
            vim.api.nvim_clear_autocmds({ group = augroup, buffer =
bufnr })
            vim.api.nvim_create_autocmd("BufWritePre", {
                group = augroup,
                -- buffer = bufnr,
                callback = function()
                    vim.lsp.buf.formatting_sync()
                end,
            })
        end
    end,
})

-----
-- Auto commands
-----

vim.cmd([[ autocmd BufWritePre <buffer> lua
vim.lsp.buf.formatting_sync() ]])

```

A opção `sources` serve para listar as fontes de formatação de código que serão usados. A [lista](#) de formatadores suportados estão no repositório do projeto. Note que não basta apenas adicionar na lista, o executável de cada um precisa estar instalado na máquina para funcionar.

A função em `on_attach` possui duas utilidades:

1. Desabilita a formatação de alguns servidores, para evitar que o NullLS pergunte qual fonte usar toda vez que aplicar a formatação, ou que tenha comportamentos estranhos.
2. Criar um `autocmd` para aplicar a formatação ao salvar, caso haja suporte para o arquivo atual.

# Instalação de formatadores

Cada formatador tem sua maneira de instalar, mas no geral, eles estão disponíveis de três formas:

1. Gerenciador de pacotes da linguagem de programação usada no desenvolvimento: Cargo para Rust, PIP para Python, NPM ou Yarn para JavaScript/TypeScript.
2. Gerenciador de pacotes do sistema: Homebrew no macOS; Pacman, APT, DNF, etc. para distribuições Linux.
3. Compilar direto do código-fonte. Se você não é um usuário do Gentoo Linux ou deseja contribuir com o desenvolvimento dos servidores de linguagem, não recomendo esta opção.

## Exemplos de instalação

Os formatadores usados nesta configuração foram Black (Python), Rustfmt (Rust), PHP CS Fixer (PHP), Prettier (JavaScript e derivados, HTML, CSS e derivados) e Stylua (Lua).

Para instalar o Stylua, o Prettier, PHP CS Fixer e o Black usando os gerenciadores de suas respectivas linguagens:

```
$ cargo install stylua
$ pip install black
$ (sudo) npm i -g prettier
$ composer global require friendsofphp/php-cs-fixer
```

Para instalar o Stylua com o Pacman no Arch Linux:

```
$ sudo pacman -S stylua
```

Pelo Homebrew, é possível instalar o PHP CS Fixer da seguinte maneira:

```
$ brew install php-cs-fixer
```

Lembrando que o Homebrew não é somente para macOS. Ele está disponível para Linux e Windows, também.

---

# Servidores de linguagem

O Neovim possui um detector de linguagem acoplado. Ele pode ser configurado do zero, mas para facilitar este processo, a equipe do Neovim disponibilizou a extensão *Neovim LSP Config*. Com ela, basta inicializar o `lspconfig` mencionando o nome do servidor de linguagem que deseja ativar no Neovim. Segue um exemplo abaixo:

```
-----  
-- Servidores de linguagem  
-----  
  
local lspconfig = require("lspconfig")  
local caps = vim.lsp.protocol.make_client_capabilities()  
local no_format = function(client, bufnr)  
    client.resolved_capabilities.document_formatting = false  
end  
  
-- Capabilities  
caps.textDocument.completion.completionItem.snippetSupport =  
true  
  
-- Python  
lspconfig.pyright.setup({  
    capabilities = caps,  
    on_attach = no_format  
})  
  
-- PHP  
lspconfig.phpactor.setup({ capabilities = caps })  
  
-- JavaScript/Typescript  
lspconfig.tsserver.setup({  
    capabilities = caps,  
    on_attach = no_format  
})  
  
-- Rust  
lspconfig.rust_analyzer.setup({  
    capabilities = snip_caps,  
    on_attach = no_format
```

```

}))

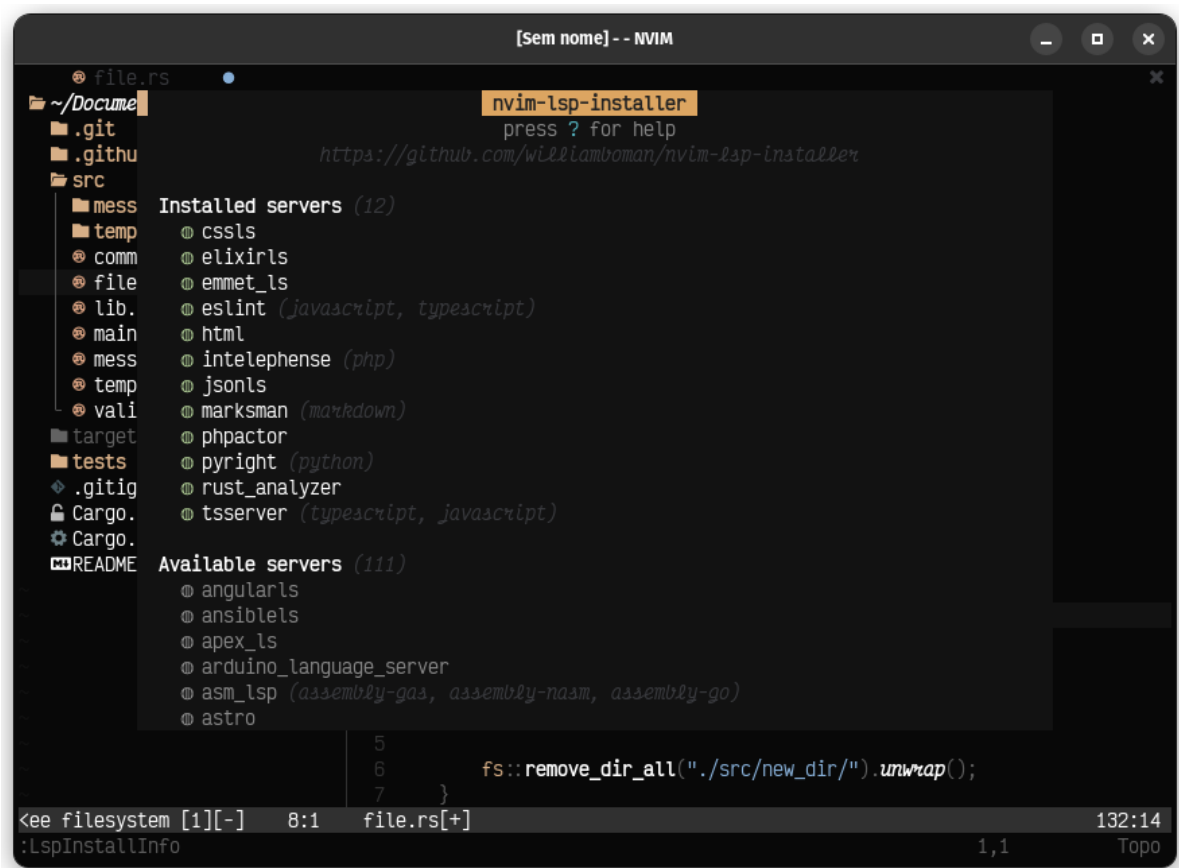
-- Emmet
lspconfig.emmet_ls.setup({
  capabilities = snip_caps,
  filetypes = {
    "css",
    "html",
    "javascriptreact",
    "less",
    "sass",
    "scss",
    "typescriptreact",
  },
})

```

A opção `capabilities` foi usada para adicionar snippets e sugestões de código com base na linguagem de programação. Os servidores configurados acima foram *Pyright* para Python, *PhpActor* para PHP, *TSServer* para Javascript/TypeScript, *Rust Analyzer* para Rust e *Emmet* para tornar a criação de HTML e CSS/Sass algo mais suportável.

Servidores de linguagens também devem ter seus executáveis presentes na máquina para funcionar. Felizmente a extensão *Neovim LSP Installer* ajuda nisso. Basta executar o comando `:LSPInstallInfo` e uma lista com diversos servidores irá aparecer.





### Diálogo do LSP Installer

Para instalar, basta pressionar **i** sobre o nome do servidor desejado. Para atualizar, pressione **u**. Para desinstalar, pressione **x**. Tudo é bem visual, e se estiver perdido é só pressionar **?** para acessar a ajuda.

## E o CoC?

Apesar de ser muito útil e bem mais simples de configurar, ele é mais lento. Antes de escrever este artigo fiz testes no dia-a-dia, e o Neovim com LSP configurado em Lua se mostrou bem mais responsivo.

Provavelmente, o fato do CoC ser feito em TypeScript e VimScript, linguagens mais lentas do que o Lua, tenham sido a causa dessa impressão.

## Atalhos de teclado

Aqui não há novidade. Da mesma forma que fiz com as opções no começo do artigo, aloquei as funções com nomes menos legíveis em variáveis locais com nomes parecidos com os usados no VimScript.

-----

```

-- Atalhos de teclado
-----

local map = vim.api.nvim_set_keymap
local kmap = vim.keymap.set
local opts = { noremap = true, silent = true }

-- Leader
vim.g.mapleader = " "

-- Vim
map("n", "<F5>", ":Neotree toggle<CR>", opts)
map("n", "<C-q>", ":q!<CR>", opts)
map("n", "<F4>", ":bd<CR>", opts)
map("n", "<F6>", ":sp<CR>:terminal<CR>", opts)
map("n", "<S-Tab>", "gT", opts)
map("n", "<Tab>", "gt", opts)
map("n", "<silent> <Tab>", ":tabnew<CR>", opts)
map("n", "<C-p>", ':lua
require("telescope.builtin").find_files()<CR>', opts)

-- Diagnóstico
kmap("n", "<space>e", vim.diagnostic.open_float, opts)
kmap("n", "[d", vim.diagnostic.goto_prev, opts)
kmap("n", "]d", vim.diagnostic.goto_next, opts)
kmap("n", "<space>q", vim.diagnostic.setloclist, opts)

local on_attach = function(client, bufnr)
  -- Habilitar auto-completar com <c-x><c-o>
  vim.api.nvim_buf_set_option(bufnr, "omnifunc",
    "v:lua.vim.lsp.omnifunc")

  -- Mapeamentos.
  local bufopts = { noremap = true, silent = true, buffer =
bufnr }
  kmap("n", "gD", vim.lsp.buf.declaration, bufopts)
  kmap("n", "gd", vim.lsp.buf.definition, bufopts)
  kmap("n", "K", vim.lsp.buf.hover, bufopts)
  kmap("n", "gi", vim.lsp.buf.implementation, bufopts)
  kmap("n", "<C-k>", vim.lsp.buf.signature_help, bufopts)
  kmap("n", "<space>wa", vim.lsp.buf.add_workspace_folder,
bufopts)

```

```

kmap("n", "<space>wr", vim.lsp.buf.remove_workspace_folder,
bufopts)
kmap("n", "<space>wl", function()
    print(vim.inspect(vim.lsp.buf.list_workspace_folders()))
end, bufopts)
kmap("n", "<space>D", vim.lsp.buf.type_definition, bufopts)
kmap("n", "<space>rn", vim.lsp.buf.rename, bufopts)
kmap("n", "<space>ca", vim.lsp.buf.code_action, bufopts)
kmap("n", "gr", vim.lsp.buf.references, bufopts)
kmap("n", "<space>f", vim.lsp.buf.formatting, bufopts)
end

```

## Detalhes finais

Uma coisa que detestei assim que comecei a usar esta configuração, foi os erros em textos virtuais na mesma linha. Raramente o erro é conciso o suficiente para caber na tela, e no caso de várias ocorrências a leitura fica bem prejudicada.

The screenshot shows a Neovim editor window titled 'typescript-react.tsx (-/Documentos/Temas/demos/web) - NVIM'. The code is a TypeScript file with several import statements and a function definition. Multiple floating error messages are displayed on the right side of the editor, each corresponding to an import statement that cannot be found. The errors are:

- Cannot find module 'react' or its corresponding package
- Cannot find module '../components/PageHeader' or its corresponding package
- Cannot find module '../components/Input' or its corresponding package
- Cannot find module '../components/Textarea' or its corresponding package
- Cannot find module '../assets/images/icons/warning.svg' or its corresponding package
- Cannot find module '../components/Select' or its corresponding package
- Cannot find module '../services/api' or its corresponding package
- Cannot find module 'react-router-dom' or its corresponding package

The code in the background is as follows:

```

typescript-react.tsx
20 import React, { useState, FormEvent } from 'react';
19 import PageHeader from '../components/PageHeader';
18
17 import Input from '../components/Input';
16 import Textarea from '../components/Textarea';
15
14 import warningIcon from '../assets/images/icons/warning.svg';
13 import './styles.css';
12 import Select from '../components/Select';
11 import api from '../services/api';
10 import { useHistory } from 'react-router-dom';
9
8 function TeacherForm() {
7   const history = useHistory();
6
5   const [name, setName] = useState('');
4   const [avatar, setAvatar] = useState('');
3   const [whatsapp, setWhatsapp] = useState('');
2   const [bio, setBio] = useState('');
1   const [subject, setSubject] = useState('');
21 const [cost, setCost] = useState('');
1   const [scheduleItems, setScheduleItems] = useState([
2     { week_day: 0, from: '', to: '' },
3   ]);
4
5   function addNewScheduleItem() {
6     setScheduleItems([...scheduleItems, { week_day: 1, from: '', to: '' }]);
7   }

```

The status bar at the bottom shows 'NORMAL', 'master', 'typescript-react.tsx', 'utf-8', 'typescriptreact', '10%', and '21:39'. There is also a message from LSP[phactor] indicating it has finished indexing.

### Erros em texto virtual

Com o código abaixo, os erros aparecem em caixas flutuantes ao manter o cursor em cima do erro.

```

-----

-- Mensagem flutuante

-----

vim.diagnostic.config({
  float = { source = "always", border = border },
  virtual_text = false,
  signs = true,
})

-----

-- Auto commands

-----

vim.cmd([[ autocmd! CursorHold,CursorHoldI * lua
vim.diagnostic.open_float(nil, {focus=false})]])

```

The screenshot shows a Neovim editor window titled 'typescript-react.tsx (-/Documentos/Temas/demos/web) - NVIM'. The editor is displaying a TypeScript file with several import statements and a function definition. A diagnostic error is shown in a red box on the left side of the editor, indicating that the module './../components/Input' cannot be found. The error message is: 'typescript: Cannot find module './../components/Input' or its corresponding type declarations.' The editor also shows a list of imports and a function definition for 'TeacherForm'.

```

typescript-react.tsx (-/Documentos/Temas/demos/web) - NVIM
hypertext.html • | @typescript-react... x
3 import React, { useState, FormEvent } from 'react';
2 import PageHeader from '../components/PageHeader';
1
4 import Input from '../components/Input';
Diagnostics:
1. typescript: Cannot find module './../components/Input' or its corresponding type declarations.
3 import warningIcon from '../assets/images/icons/warning.svg';
4 import './styles.css';
5 import Select from '../components/Select';
6 import api from '../services/api';
7 import { useHistory } from 'react-router-dom';
8
9 function TeacherForm() {
10   const history = useHistory();
11
12   const [name, setName] = useState('');
13   const [avatar, setAvatar] = useState('');
14   const [whatsapp, setWhatsapp] = useState('');
15   const [bio, setBio] = useState('');
16   const [subject, setSubject] = useState('');
17   const [cost, setCost] = useState('');
18   const [scheduleItems, setScheduleItems] = useState([
19     { week_day: 0, from: '', to: '' },
20   ]);
21
22   function addNewScheduleItem() {
23     setScheduleItems([...scheduleItems, { week_day: 1, from: '', to: '' }]);
24   }

```

NORMAL | master > 13 typescript-react.tsx | utf-8 | typescriptreact 2% | 4:26  
1 alteração; antes de #2 5 seconds ago

*Erro em caixa flutuante*

Se isto não te incomoda, é só descartar esta parte.

## Conclusão

Pronto! Estas linhas são o suficiente para ter um Neovim funcional com um arquivo de configuração em Lua.

Se você seguiu exatamente o que foi mostrado até aqui, o seu `init.lua` ficou com cerca de 380 linhas, bem mais do que a outra versão em VimScript (154 linhas, somando o `init.vim` e o `coc-settings.json`).

Para ser sincero, pensei que seriam mais de 500 linhas, mas consegui resumir bastante. O fato de eu ter feito em um arquivo só, contribuiu para isso, incentivando a usar só o essencial. Fiz dessa forma também, porque até o momento onde escrevo estas linhas, não encontrei um tutorial que usasse só um arquivo. No geral, são pastas e mais pastas com diversos arquivos Lua, que causam bastante confusão.

Aos mais experientes, obviamente esta configuração não atende em tudo, mas no meu caso, fiquei bem satisfeito com o resultado. Em um próximo artigo, este arquivo será separado em "módulos", para melhorar a extensibilidade desta configuração.

Se quiser conferir o código completo, deixei o link para o meu repositório de `.dotfiles` abaixo.

Até a próxima!

## Links

- [Meus dotfiles](#)
- [Sobrio - Tema](#)