Configuring Vim

Customizing `~/.vimrc`

Vim is a text editor that is difficult to get comfortable with, but once you have a set configuration that works for you it's very portable and really nice to use when editing files on remote hosts, especially when you take the time to configure it to suit your needs.. Configuring vim requires taking a look at your local user's `~/.vimrc`, and depending on the features you require, you may need to install and manage vim plugins.

For new users, the `~/.vimrc` file can be easy to overlook, but taking some time to figure out the settings and plugins available can help make any beginner much more confortable in the text editor. Below, I'll go over a few settings and plugins I've picked up along the way, and hopefully you can get some use out of them, too. To begin,lets look at some basic syntax for configuring `~/.vimrc` -

```
" A single double-quote is a comment within a .vimrc file

" Two leading double-quotes shows that the line is actual code that can be uncommentted and
ran, like below
""set mouse=a

" A basic set statement, enabling expanded tabs in vim to replace tabs with spaces
set expandtab

" A custom keybinding to trigger the vim command :ColorToggle on pressing Ctrl+C
nnoremap <C-c> :ColorToggle<CR>
" Above, <CR> stands for carriage return, or enter, which submits the command as if ran
manually within vim

" Variables within vim
" Set a custom constant string variable
let hostname = 'home'
" Parse current system hostname (control which settings are applied where, should it matter)
let hostname = system('hostname')[:-2] "

" A variable can be given a scope, below we use a global variable
" This line sets the vim airline plugin theme to use
```

```
let g:airline_theme='kalisi'



" A conditional statement within .vimrc
if !exists('g:airline_symbols')
  let g:airline_symbols = {}
endif

if condition
  echo 'First'
elseif !conditiontwo
  echo 'Not second'
else
  echo 'Fail'
endif

" Define function in vim to remove whitespace
function! TrimWhitespace()
  let l:save = winsaveview()
  keeppatterns %s/\s\+$//e
  call winrestview(l:save)
endfunction

" Call this on every attempt to save a file of types defined below..
autocmd BufWritePre *.cpp,*.h,*.c,*.php :call TrimWhitespace()
```

It should be noted that if the function `TrimWhitespace()` above is defined within your `~/.vimrc`, you can call it at any time from within a vim session by running `:exec TrimWhitespace()`, this is useful if you do a lot of batch editing in vim you can define functions to carry out otherwise tedious changes. Above, I've set vim to automatically strip whitespaces from source code files each time I save by using `autocmd` on the final line.

Now that we know the basics, lets take a look at some builtin vim settings that can be applied to any vim editor right out of the box.

Builtin Settings

If `~/.vimrc` does not exist in your home directory, create it, and customize it to suit your needs. For example, the following `.vimrc` will set your tab size to 2 from the default 4, and convert your tabs to spaces automatically. This is useful when sharing code, as things are more compact and using spaces is less ambiguous than tab sizes across platforms.

```
set tabstop=2 shiftwidth=2 expandtab autoindent mouse=a
```

Here, `tabstop` is the tab size setting, measured in spaces. `shiftwidth` allows vim to compensate according to our tab settings when automatically indenting, etc. `expandtab` converts our tab size setting into actual spaces. `set autoindent` will set vim to automatically indent to our current depth when in insert mode and moving to a new line by pressing enter. This will not insert spaces unless text is input. `mouse=a` enables mouse interaction with split windows, when supported.

If you're missing having numbered lines shown in the lefthand gutter when editing a file in vim, you can enable vim's builtin numberlines by adding the following to your `~/.vimrc`

```
set number
```

To turn on syntax and set a colorscheme

```
syntax on
colorscheme sourcerer
```

Above, we set a colorscheme! Neat, but where and how do we install it? Where did it come from? I got this coloscheme from xero, but there are plenty of options out there if you want to look around yourself, or even make your own!

To get sourcerer working in your vim sessions, copy this file to your `~/.vim/colors/` directory and add the lines above to your `~/.vimrc`, vim will know to check the `~/.vim/colors` directory for the `theme.vim` file, in this case its looking for `sourcerer.vim`

Builtin Unicode Input

To enter Vim's default Unicode input mode, ensure you are in `<INSERT>` mode and press `<Ctrl>+V`. Then proceed to enter your character code following the guidelines below -

Unicode Prefixes

a decimal number (0-255) o then an octal number (o0-o377, i.e., 255 is the maximum value) x then a hex number (x00-xFF, i.e., 255 is the maximum value) u then a 4-hexchar Unicode sequence U then an 8-hexchar Unicode sequence

So, if we wanted the `stopwatch - f2f2` symbol from Font Awesome's Cheatsheet, we would enter `<INSERT>` mode within Vim and press `<Ctrl>+V`, followed by the character keypresses respective to our (4-char) unicode symbol - `u+f+2+f+2`.

Note that Vim will not change appearance or indicate that it is pending input for a character sequence, once pressing `<Ctrl>+V` within `<INSERT>` mode we are not prompted further. This is expected and if the sequence is done correctly Vim will input the Character specified by the sequence input, whether its decimal, octal, hex, or unicode, just be sure to use the appropriate

prefix listed above

Plugin Management

Vim stores plugins within `~/.vim/bundles/` and managing them is made simple using various vim plugin managers. See some of the repositories below for different options. Currently, I am using Pathogen, but there are many options that provide great solutions to vim plugin management.

For example, I might run something like `git submodule add https://github.com/user/plugin ~/dot/.vim/bundles/plugin/` to add a plugin to vim and track it on my dotfiles repository. Since I use stow to manage my dotfiles all of these new files will also reflect on my local user configurations and git will still be able to track them within the single `~/dot/` repository.

For this reason, I prefer Pathogen since it pairs well with Git submodules, but again, to each their own. Other plugin managers may pair well with git in their own ways as well. Consider the options below.

Pathogen plugin manager for Vim, allows for easy installation of useful plugins via `git clone` into a specified directory.. Don't like it? To uninstall Pathogen -

- delete `~/.vim/autoload/pathogen.vim`,
- delete the lines you have added to `~/.vimrc`.

Check out Plug, Vundle, or Dein to name a few alternative vim plugin managers..

Plugins

Below, we'll see a few plugins that I've found to be useful. If you are using Pathogen, any of these plugins can be installed by `git clone https://github.com/user/plugin ~/.vim/bundles/plugin`, but you'll want to check each GitHub for updated instructions on how to configure the plugins to work with vim. For some of the more specific settings and issues I came across, I've provided examples.

Unicode.vim Plugin

The unicode.vim plugin on GitHub adds easy support for Unicode characters, some of the useful commands can be seen below (Mostly taken from the official README.md within the plugin repository linked above)

If you just installed the plugin, run the below to update your unicode tables, just to be sure you have the full list

```
:DownloadUnicode - Download (or update) Unicode data
```

```
:Digraphs        - Search for specific digraph char
:UnicodeSearch   - Search for specific unicode char
```

```
:UnicodeSearch!  - Search for specific unicode char (and add at current cursor position)
:UnicodeName     - Identify character under cursor (like ga command)
:UnicodeTable    - Print Unicode Table in new window
:DownloadUnicode - Download (or update) Unicode data
:UnicodeCache    - Create cache file
:UnicodeTable    - Print Unicode Table in new window
```

For me, this is a very useful plugin when I want to grab a unicode symbol and insert it within a configuration. For example, if using the Ale lint engine you can define symbols that appear within your vim gutter when the linter detects and error or warning within your code. When editing these kinds of configurations within vim, you can use the command `:UnicodeSearch! cancel` to search for a unicode `X` symbol to appear when errors are found. You'll then be able to select from a list of relevant symbols that are installed based on your available fonts, inserting it at your cursor position within the file.

```
:Digraphs        - Search for specific digraph char

    :Digraphs

Outputs the digraph list in an easier way to read with coloring of the
digraphs. If a character has several digraphs, all will be shown, separated by
space.

If you want to display a list with a line break after each digraph, use the
bang attribute (Note, this output also contains the name in parentheses). >

    :Digraphs!

And if you want to display all digraphs matching a certain value, you can add
an argument to the command: >

    :Digraphs! A

displays all digraphs, that match 'A' (e.g. all that can be created with the
letter A or whose digraph matches the letter 'A'.)
Note: This is a silly example, that can take some time. You should always be
able to abort that by pressing |CTRL-C|. To output progress information, call
the command with the |:verbose| command modifier.

If you know the name, you can also search for the unicode name: >
```

```
    :Digraphs copy

will display all Digraphs, where their unicode name contains the word "copy"
(e.g. copyright symbol). Case is ignored. Note, you need at least to enter 2
characters.

:UnicodeTable    - Print Unicode Table in new window
```

See the unicode.vim official GitHub docs for more info

Code Completion in Vim

Check out https://github.com/xavierd/clang_complete/ for code completion. Instructions are within the README there. The path used to setup this plugin is dependent on the `clang` and `libclang` package and setup will be different depending on which version of clang you are using. When installing via `sudo apt install libclang-10-dev` you can expect your path to be the same as the path I use below. Otherwise, find it on your system using `sudo find / -name libclang.so.1`.

After getting your path, set the global variable below within your `~/.vimrc` If you see errors on opening a cpp file you might not have set your clang library path correctly.

```
let g:clang_library_path='/usr/lib/llvm-10/lib/'
```

Alternatively, if you want to automate this a bit and use an environment variable to determine your clang path as I have within my dotfiles repository, you can run the commmands below

```
echo "let g:clang_library_path=$LIBCLANG" >> ~/.vimrc
echo "export LIBCLANG=\""$(sudo find / -name libclang.so.1)"\"" >> ~/.bash_aliases
source ~/.bashrc
```

...Still having issues? I've found with some versions of `libclang` that `ln /usr/lib/llvm-10/lib/libclang.so.1 /usr/lib/llvm-10/lib/libclang.so` seemed to resolve some issue where the plugin would not detect `libclang.so.1` but would detect `libclang.so`. All this command does is create a symbolic link to a new file, `libclang.so`, that simply points to the original `libclang.so.1` file that already exists on our system. If this file does not exist, you don't have clang installed! Run `sudo apt install clang` or the respective command for your package manager to install clang locally, then check for the `libclang.so.1` file again.

Supertab

If you use code-completion, you'll probably miss the tab function that usually brought up a context menu with code snippets. To use something similar, check out supertab, its a really handy and easy to configure. I have no settings related to supertab in my `~/.vimrc`, all I needed to do was clone it into my `~/.vim/bundles/supertab/` directory. From here, entering a vim file with any text

and attempting to type results in a dialog box popup similar to that of an IDE with suggestions.

Linters

Ale is my lint engine of choice, check it out on the official GitHub

My `~/.vimrc` contains the below settings related to Ale, and otherwise uses the default configuration. Ale can run all kinds of linters, which can be configured within your `~/.vimrc` file to be triggered based on the filetype you are editing. Check out the GitHub for more information.

```
" Ale linter settings
" Hover detail info in preview window
let g:ale_hover_to_preview = 1
" Hover detail info in balloons
""let g:ale_set_balloons = 1
" Set custom symbols to Ale gutter when errors / warnings show
let g:ale_sign_error = '□'
let g:ale_sign_warning = '□'
" Change Ale highlight settings
highlight ALEWarningSign ctermbg=Yellow
highlight ALEWarningSign ctermfg=Black
highlight ALEWarning ctermbg=DarkYellow
highlight ALEWarning ctermfg=White
highlight ALEErrorSign ctermbg=DarkRed
highlight ALEErrorSign ctermfg=White
highlight ALEError ctermfg=DarkRed
" Set Ctrl+j/k to move to next/prev errors and warnings
nmap <silent> <C-k> <Plug>(ale_previous_wrap)
nmap <silent> <C-j> <Plug>(ale_next_wrap
```

Ale can be further configured to support a number of languages and features. For my needs, it has worked just fine out of the box. Ale has builtin features which are not modified above, such as auto-completion relative to the language and source code you are editing, which is very useful when working in larger projects as it displays some information on the variable names and their types. Ale supports popup bubbles or text preview on mouse hover or normal cursor overlap of a variable or function, even for standard libraries and includes, which is nice and really helps to make vim feel more like a full featured editor.

Colorizer

This plugin helps a lot if you do any amount of web design or css, and for me came in handy when working with i3wm and other customizations to my desktop environment. Colorizer hightlights the various forms and syntax representations of colors that exist within files you're editing. Below are

my settings related to Colorizer, even those which I've commented out but leave in there should I want to turn them on quickly.

```
" Colorizer plugin settings
" See :h colorizer in Vim for more info
""let g:colorizer_colornames = 0 " Don't color literal names, like red, green, etc
let g:colorizer_auto_color = 0
""let g:colorizer_skip_comments = 1
""let g:colorizer_auto_filetype ='css,html,vim'
nnoremap <C-c> :ColorToggle<CR>
```

Above, I set a command to toggle the Colorizer plugin with `Ctrl+c`, pressing this combination of keys automatically inputs the `:ColorToggle` vim command and then `<CR>` enters the command to be ran. Without the `<CR>`, the command would just appear input at the bottom of our session, waiting for us to hit the enter key to run it. In this case, that was not a useful scenario, but possibly if you find yourself doing complex searches like a find and replace - You could easily create a keybind that would layout the general format of the command quickly, allowing you to make some small changes before running the command like the word to find and what to replace it with.

Ranger

To tie everything together ranger is a really useful tool, while it is not a vim plugin or configuration of any kind, it is a terminal file browser inspired by vim. Ale paired with supertab and ranger for quick filebrowsing and edits can make for a nice and portable configuration allowing you to hop around quickly on a host to edit and preview files. For me, this was really useful when working with Ansible roles as you often cross reference multiple files. Ranger even supports image previews, see the ranger GitHub for more information on configuring ranger to do more, but if you just want to check it out in its default state run `sudo apt install ranger && ranger` and have a look for yourself.

For images in ranger, I recommend installing the GitHub version of ranger for the time being. It brings support for ueberzug, a `pip3` package that can be installed and used to display images within a terminal much more reliably. The GitHub version of ranger has instructions on how to enable this within the default `~/.config/ranger/rc.conf` configuration file generated by running `ranger --copy-config=all`. After following these instructions, just `sudo pip3 install ueberzug`. If you don't have pip3, `sudo apt install python3-pip`.

You may also be interested to check out devicons, a plugin for ranger which displays console icons next to files corresponding to their type. For example, a directory will have a folder icon, a configuration file a gear, etc. This plugin requires the use of Patched Nerd Fonts.

Backup Vim Configurations

To use vim to its full potential, its useful to stay organized when testing out different vim configurations, and providing yourself with a git repository to track your changes is a good way to do so. This way, should problems arise or should your system be lost for any reason, returning to

your preferred setup is not a case a deja vu, but instead a planned restoration of your already backed up settings. This enables you to quickly establish your settings on a new host without over complicating the process or repeating steps across multiple hosts.

To create a git repository storing dotfiles, see information on stow. This same concept can generally be applied to any application that stores local user configurations, but it is very imporant to know exactly what changes will be applied when using stow as it will replace system files - stow(8)

At the very least, run a `cp -r ~/.vim* ~/backup/vim/` from time to time.

Backup Vim With Stow

To backup vim configurations using stow, create the file structure like the below tree by copying your vim settings with `cp -r ~/.vim* ~/backup/vim/`

```
vim/
├── .vim
│   ├── autoload/
│   ├── bundle/
│   ├── colors/
│   ├── doc/
│   ├── .netrwhist
│   ├── plugin/
│   └── .VimballRecord
└── .vimrc
```

If the `~/backup/vim/` directories don't exist, create them. Once this has been created, from the `~/backup/` directory, run `stow --adopt vim/`. this will create a symbolic link to the configuration files and directories on your system, which enables you to edit the files within `~/backup/vim/` and the changes will reflect in the configurations stored within the parent `~/` directory. If these files or configurations already exist in the parent directory, stow will overwrite them. If they do not exist, they will be created / linked to the files in `~/backup/vim/`.

This is a powerful tool when storing configurations in remote repositories, sometimes for various users or configurations based on distributions or window managers.

Personally, I prefer this method over writing a script to handle this manually, but since I did the work at writing the manual backup script, take a look below if you're interested in such a solution. I haven't revised it in quite a while, but it may at the very least provide some ideas.

Backup Scripts

A while back, I created a script that configures vim according to my preferred settings. Though I haven't used or updated this for some time, I'll leave it here an an example of how you could do

this yourself.

Feel free to tweak it to suit your needs, create your own, or find a better one somewhere else. If nothing else, you might get some ideas by reading through the script below -

Depending on your system, the script below attempts to globally configure vim's default settings

```
#!/bin/bash
## Author: Shaun Reed | Contact: shaunrd0@gmail.com | URL: www.shaunreed.com ##
## A custom bash script to configure vim with my preferred settings          ##
## Run as user with sudo within directory to store / stash .vimrc configs    ##
################################################################################

# For easy colorization of printf
GREEN=$(tput setaf 2)
RED=$(tput setaf 1)
UNDERLINE=$(tput smul)
NORMAL=$(tput sgr0)


welcome=( "\nEnter 1 to configure vim with the Klips repository, any other value to exit." \
  "The up-to-date .vimrc config can be found here:
https://github.com/shaunrd0/klips/tree/master/configs" \
  "${RED}Configuring Vim with this tool will update / upgrade your packages${NORMAL}\n\n")

printf '%b\n' "${welcome[@]}"
read cChoice

if [ $cChoice -eq 1 ] ; then

  printf "\nUpdating, upgrading required packages...\n"
  sudo apt -y update && sudo apt -y upgrade
  sudo apt install vim git

  # Clone klips repository in a temp directory
  git clone https://github.com/shaunrd0/klips temp/
  # Relocate the files we need and remove the temp directory
  sudo mkdir -pv /etc/config-vim
  sudo cp -fruv temp/README.md /etc/config-vim/

  sudo cp -fruv temp/configs/ /etc/config-vim/
```

```
  rm -Rf temp/
  printf "\n${GREEN}Klips config files updated"
  printf "\nSee /etc/config-vim/README.md for more information.${NORMAL}\n\n"

  # Create backup dir for .vimrc
  sudo mkdir -pv /etc/config-vim/backup/
  printf "\n${GREEN}Backup directory created - /etc/config-vim/backup/${NORMAL}\n"

  # Set global vimrc defaults to klips settings
  sudo cp /etc/config-vim/configs/.vimrc /usr/share/vim/vimfiles/vimrc

  # Stash the current .vimrc
  sudo mv -bv ~/.vimrc /etc/config-vim/backup/
  printf "${RED}Your local .vimrc has been stashed in /etc/config-vim/backup/${NORMAL}\n\n"

  # Copy our cloned config into the user home directory
  sudo cp /etc/config-vim/configs/.vimrc ~/
  printf "${GREEN}New /usr/share/vim/vimfiles/rc configuration installed.${NORMAL}\n"

  # Reinstall Pathogen plugin manager for vim
  # https://github.com/tpope/vim-pathogen
  printf "\n${RED}Removing any previous installations of Pathogen...${NORMAL}\n"
  sudo rm -f /usr/share/vim/vimfiles/autoload/pathogen.vim

  # Install Pathogen
  printf "\n${GREEN}Installing Pathogen plugin manager for Vim....\n"
printf "\nIf they don't exist, we will create the following directories:\n"
printf  "/usr/share/vim/vimfiles//autoload/    ~/.vim/bundle/${NORMAL}"
  mkdir -pv /usr/share/vim/vimfiles/autoload /usr/share/vim/vimfiles/bundle && \
  sudo curl -LSso /usr/share/vim/vimfiles/autoload/pathogen.vim https://tpo.pe/pathogen.vim
  printf "\n${GREEN}Pathogen has been installed! Plugins plugins can now be easily
installed.\n"\
         "Clone any plugin repositories into /usr/share/vim/vimfiles/bundles${NORMAL}\n"

  # Remove any plugins managed by this config tool (Klips)
  printf "\n${RED}Removing plugins installed by this tool...${NORMAL}\n"
  sudo rm -R /usr/share/vim/vimfiles/bundle/*

  # Clone plugin repos into pathogen plugin directory
  printf "\n${GREEN}Installing updated plugins...${NORMAL}\n"
  git clone https://github.com/ervandew/supertab /usr/share/vim/vimfiles/bundle/supertab && \
```

```
  printf "\n${GREEN}Supertab plugin has been installed${NORMAL}\n\n" && \
  git clone https://github.com/xavierd/clang_complete
/usr/share/vim/vimfiles/bundle/clang_complete && \
  printf "\n${GREEN}Clang Completion plugin has been installed${NORMAL}\n\n"
  vimConf=( "\n${UNDERLINE}Vim has been configured with the Klips repository.${NORMAL}" \
    "\nConfiguration Changes: " )
  printf '%b\n' "${vimConf[@]}"



else
printf "\nExiting..\n"
fi

sudo cat /etc/config-vim/configs/.vimrc-README
```

Revision #7
Created 2019-08-30 08:15:30 UTC by Shaun Reed
Updated 2020-06-15 13:14:54 UTC by Shaun Reed