Extracted from:

Modern Vim

Craft Your Development Environment with Vim 8 and Neovim

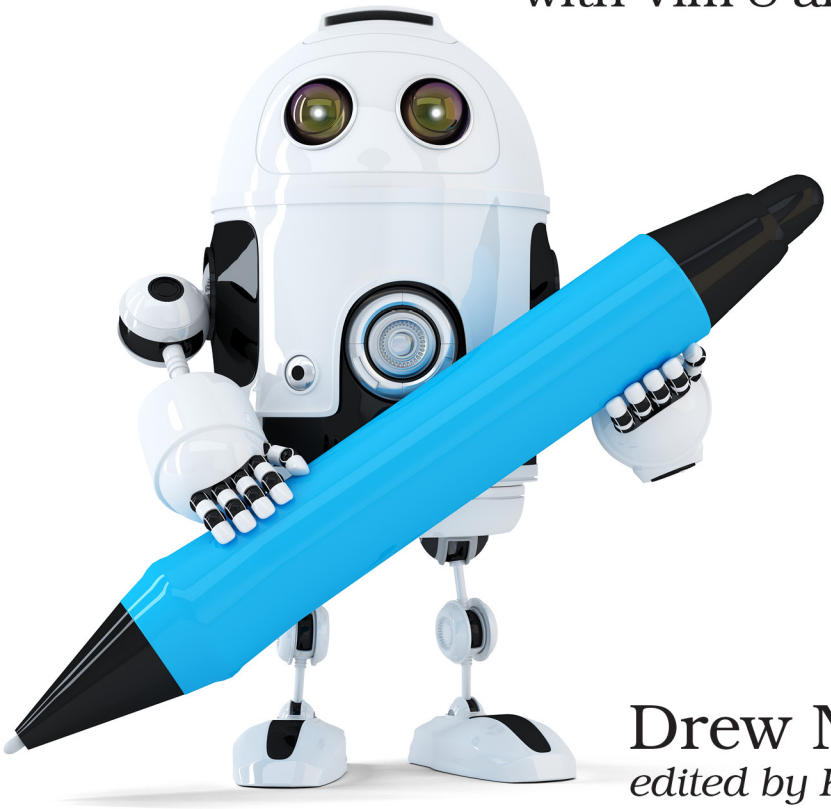
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Modern Vim

Craft Your Development Environment with Vim 8 and Neovim



Drew Neil

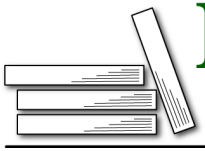
edited by Katharine Dvorak

Modern Vim

Craft Your Development Environment with Vim 8 and Neovim

Drew Neil

The Pragmatic Bookshelf
Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-262-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—November 1, 2017

Tip 10

Running a Build and Navigating Failures

When you run a build tool and everything works fine, you can usually disregard any output the tool produced. But when the build tool fails, it may emit output that contains clues about the line of code where the failure occurred. Using Vim's compiler plugins and the :make command, you can run a build tool and capture the output so that you can refer to it later.

Better still, Vim can parse any references to filenames and line numbers, allowing you to quickly jump to the line of code where an error originated. However, the fact that Vim's built-in :make command runs synchronously can be irritating, especially for long-running builds. The Dispatch plugin solves this by providing various adapters that allow you to run build tools asynchronously.

Preparation

To follow the examples in this tip, you'll need to install Tim Pope's Dispatch plugin.[1] You can install it to your bundle package like this:

```
⇛ $ cd $VIMCONFIG/pack/bundle/start
⇛ $ git clone https://github.com/tpope/vim-dispatch.git
```

Dispatch was ahead of its time when it was released in 2013. Vim had no support for running external commands asynchronously, and Neovim didn't even exist. Make sure you check out the *Introducing dispatch.vim* screencast,[2] and enjoy the unusual fanfare that announced the plugin's release.

The Dispatch plugin supports several different adapters, allowing you to pick a strategy for running async commands that suits you. For Vim 8, I recommend using the tmux adapter. For Neovim, I recommend the neovim adapter.

Enabling the tmux Adapter

The Dispatch plugin has a built-in tmux adapter, which runs programs in a tmux pane or window. To use this adapter, start a tmux session and then launch Vim inside of that session:

```
⇛ $ tmux
⇛ $ vim
```

1. https://github.com/tpope/vim-dispatch
2. https://vimeo.com/63116209

When you run Vim inside of a tmux session, Dispatch will select the tmux adapter by default. I recommend reading *tmux 2 [Hog16]* to learn more about tmux.

Enabling the Neovim Adapter

For Neovim, you'll want to install an extra plugin: vim-dispatch-neovim[3] by Richard Adenling. This adds a neovim adapter, which makes Dispatch run programs using Neovim's terminal emulator. You can install it to your bundle package like this:

```
⇛ $ cd $VIMCONFIG/pack/bundle/start
⇛ $ git clone https://github.com/radenling/vim-dispatch-neovim.git
```

When using Neovim, Dispatch will select the neovim adapter automatically (even if you're running Neovim inside of a tmux session). This is my preferred way of using Dispatch. I find it satisfying being able to run programs asynchronously without relying on tools such as tmux. I recommend reading Chapter 5, *Neovim's Built-in Terminal Emulator*, on page ? to learn more about Neovim's terminal emulator.

Vim 8 Job Adapter for the Dispatch Plugin

If you'd like to use Vim 8's job control feature for running tasks asynchronously, you might want to try out an experimental branch of Dispatch. Instead of installing the master branch, you could install the job branch like this:

```
⇛ $ cd $VIMCONFIG/pack/bundle/start
⇛ $ git clone -b job https://github.com/tpope/vim-dispatch.git
```

The job strategy has the benefit of being self-contained, meaning that you can use Dispatch to perform async tasks without having to depend on tmux. But this strategy also has some limitations. For example, jobs aren't well-suited for commands that read from stdin, so if you're running a command that has an interactive mode, or needs to show a password prompt, the task may fail in unexpected ways. For this reason, the job branch remains experimental. With the prospect of a built-in :terminal command coming soon (see the appendix, *What's Next For Modern Vim*), it may be that a terminal adapter would render these issues moot.

3. https://github.com/radenling/vim-dispatch-neovim

Setting Up the Demo Project

The source code that accompanies this book includes a good-day directory. In there you'll find a simple TypeScript project called good-day. Switch to that directory and use npm to install the TypeScript compiler and its dependencies:

```
⇛ $ cd code/good-day
⇛ $ npm install
❮ good-day@1.0.0 /Users/drew/code/good-day
  ├── tslint@5.7.0
  └── typescript@2.5.2
```

The TypeScript compiler binary is installed in the node_modules/.bin directory. You can execute it like this:

```
⇛ $ ./node_modules/.bin/tsc --version
❮ Version 2.5.2
```

For this tip, it's important you are able to run the TypeScript Compiler without having to specify the full path. To make this work, modify your environment so that the ./node_modules/.bin appears at the start of your path. The npm bin command outputs the full pathname for that directory:

```
⇛ $ export PATH=$(npm bin):$PATH
⇛ $ tsc --version
❮ Version 2.5.2
```

Now, run tsc to build the project:

```
⇛ $ tsc
```

Open the build/greeting.html file in a browser with JavaScript enabled and you should see a greeting message wishing you a good day.

Making Vim Call the Compiler (a Naive Approach)

Open the sample TypeScript file in Vim:

```
⇛ $ vim src/greeting.ts
```

You can use Vim's :!{cmd} command to invoke the TypeScript compiler on the current file:

```
⇛ :!tsc --outDir build %
```

Note that the :!{cmd} runs synchronously, meaning that you can't interact with Vim until the command finishes execution. The % symbol is a shorthand for the filepath of the active buffer (:help cmdline-special). This command

generates a greeting.js file in the build directory. When everything goes smoothly, the compiler produces no output on stdout.

Now, let's see what happens when things don't go so smoothly. The good-day directory contains a break-things.diff patch file. Apply the patch by running:

```
⇛ :!patch % break-things.diff
⇛ :edit! src/greeting.ts
```

Now, try compiling the TypeScript file again:

```
⇛ :!tsc --outDir build %
❮ src/greeting.ts(9,7): error TS2322: Type '1' is not assignable...
  src/greeting.ts(23,22): error TS2345: Argument of type '"now"'...
  [Process exited 2]
```

The TypeScript compiler prints a couple of error messages. These can help you fix the issues by showing you the filename, line number, and column number where each error originated. But you've got a problem: With your next keystroke, the output from :!{cmd} is dismissed. That's inconvenient if you want to refer to those error messages.

There are two things about the :!{cmd} workflow that we could improve upon. First, it would be better if the output from the build was captured in such a way that we could easily refer to it later. Next, it would be handy if there was an option to run a build asynchronously so that you could continue to interact with Vim while the build executes. We'll tackle each of these issues one by one.

Capturing Compiler Output with :make

If you want to capture the output from an external program, you should reach for the :make command (:help :make). As the name suggests, you can use this to run a build that's configured by a Makefile. But you can also use the :make command to run other types of builds. In this demonstration, we'll continue to use the TypeScript compiler.

To use the :make command, you need to configure two options: 'makeprg' and 'errorformat'. These depend upon each other, so if you change the value of 'makeprg', it's likely that you'll also want to change the value of 'errorformat' (and vice versa). The best way to ensure that you change both options simultaneously is by loading a compiler plugin.

For the demonstration in this tip, you can use this simple compiler plugin:

compiler/typescript.vim
```
let current_compiler = "typescript"
CompilerSet makeprg=tsc\ $*\ --outDir\ build\ %
CompilerSet errorformat=%+A\ %#%f\ %#(%l\\\,%c):\ %m,%C%m
```

To install this, create an after/compiler directory in your $VIMCONFIG directory. Then copy the typescript.vim file there:

```
⇛ $ mkdir -p $VIMCONFIG/after/compiler
⇛ $ cp code/compiler/typescript.vim $VIMCONFIG/after/compiler/
```

We're using the after/compiler directory (rather than compiler) to make sure this compiler overrides any other typescript compilers you may have installed on your system. Open the src/greeting.ts file in Vim, then activate the typescript compiler by running:

```
⇛ :compiler typescript
```

That sets buffer-local 'makeprg' and 'errorformat' options to the values specified in the compiler plugin. Now you can compile your TypeScript file just by running:

```
⇛ :make
❮ !tsc  --outDir build src/greeting.ts
```

Vim executes the command specified by the 'makeprg' option, then populates the quickfix list using any output produced by that command. If you're not seeing any output, use the break-things.diff patch and run :make again. To inspect the output from the build, open the quickfix window:

```
⇛ :copen
```

You can traverse the list of errors using the commands: :cfirst, :cprev, :cnext, :clast. These commands allow you to quickly jump between errors, fixing them as you go.

TypeScript Support for Vim

As I write this, Vim has no built-in support for TypeScript (although that may have changed by the time you read this). If you'd like to add TypeScript support, you can install the typescript-vim plugin[a] by Leaf Garland. With this installed, Vim will recognize the .ts extension and enable syntax highlighting for TypeScript files. It also includes a compiler plugin for TypeScript.

a. https://github.com/leafgarland/typescript-vim

Running :make Asynchronously

The killer feature of the :make command is that it populates the quickfix list, allowing you to navigate easily between any error messages. But there's a downside to this command: It runs synchronously. That means you can't interact with Vim until the build completes. This is of little consequence if your build completes quickly, but for long-running builds, it can interrupt your workflow.

You can simulate a long-running build by installing the tardyscript compiler, which is provided with this book's source code. This is identical to the typescript compiler we used earlier, except that it sleeps for five seconds before launching the build.

```
⇛ $ cp code/compiler/tardyscript.vim $VIMCONFIG/after/compiler/
```

Enable the tardyscript compiler and start another build:

```
⇛ :compiler tardyscript
⇛ :make
❮ !sleep 5;tsc --outDir build src/greeting.ts
```

This build takes at least five seconds to complete. During that time you can't interact with Vim in any way. (Now for the moment we've been building up to...)

The dispatch plugin provides a :Make command (note the capital "M"), which behaves like an asynchronous version of Vim's built-in :make command. Try it out, once again using the tardyscript compiler:

```
⇛ :compiler tardyscript
⇛ :Make
```

If you're using the neovim adapter, Dispatch opens a terminal buffer in a horizontal split and runs the program there. With the tmux adapter, Dispatch creates a new tmux pane and runs the program there. If the build fails, the output is used to populate the quickfix list and the quickfix window opens automatically. If the build succeeds, the quickfix window doesn't open automatically.

Dispatch also provides a :Make! variation. Try it out and observe the differences:

```
⇛ :Make!
```

With the neovim adapter, Dispatch uses jobstart() to run the program in the background. With the tmux adapter, Dispatch creates a new tmux window and runs the program there, while keeping the current tmux window active.

The quickfix window doesn't automatically open when the program exits, but you can open the quickfix window at your convenience using the :Copen command.

To summarize: the :Make command lets you run a build in the *foreground*, while the :Make! command lets you run a build in the *background*. A foreground build is appropriate for shorter tasks ("build this file"), while a background build is more suitable for long running tasks ("build the whole project"). Whichever method you choose, you can continue to operate Vim while the build executes. It makes no difference if the build takes five seconds or five minutes.

When a build is running asynchronously, you should be cautious about saving changes to files, because this could affect the build. As a rule of thumb, it's okay to use Vim for reading code, but not wise to make changes while a build is running.

Incorporating Compiler Plugins to Your Workflow

When you get the hang of Vim's compiler plugins, you might find yourself wanting to use :make for more than just building the project. For example, the quickfix list could be useful if you want to lint all of the files in your project. Or if you want to run your test suite. Check out the next tip to see how the Dispatch plugin can further streamline your workflow.

If you want to get the most out of Vim's quickfix feature, you should invest some time on understanding how compiler plugins work. Check out *Tip 30, Understanding Compiler Plugins* for some pointers.