

THE EXPERT'S VOICE® IN OPEN SOURCE

tmux Taster

*LEARN HOW TO USE THIS TERMINAL
MULTIPLEXER IN THIS EASY-TO-DIGEST
TASTER VOLUME*

Mark McDonnell

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.



Apress®

Contents at a Glance

About the Author xiii

About the Technical Reviewers xv

Acknowledgments xvii

Introduction xix

■ Chapter 1: Terminal Multiplexer 1

■ Chapter 2: Fundamentals 19

■ Chapter 3: Modifications 31

■ Chapter 4: Copy and Paste..... 37

■ Chapter 5: Pane/Window Management 43

■ Chapter 6: Scripting and Automation 51

■ Chapter 7: Pair Programming..... 59

■ Chapter 8: Workflow Management 67

Index..... 73

Introduction

The standard terminal that comes installed as part of your operating system (whether it be Mac, Linux, or Windows based) is seen by most users to be a harsh and barren wasteland, devoid of emotion and color. Respected by many a neckbeard for its power in all but a few areas where it falls short.

If you're unfamiliar with the (joke) term "neckbeard", it roughly refers to a Unix system administrator type user. You know, the type of person who has an intimate knowledge of the internals of the Linux kernel and could probably write an OS over a weekend – the sort of hardened engineer who fears no terminal environment. Even *they* have experienced times where their terminal fails to do the advanced tasks required to ease their workload; and this is where a terminal *multiplexer* steps in. The focus of this book being one particularly popular multiplexer: *tmux*.

At this stage I won't divulge any further details, as I cover the majority of "why use a multiplexer?" thoughts in the opening chapter of this book. But suffice to say that the information contained in this book should help you not only understand what *tmux* is and how to use its basic features, but it also covers customizations and abstractions; automation via a scripted interface; resolving notorious copy and paste issues; pair programming (using a tool such as Vagrant to emulate a remote server environment), as well as looking at the best ways to manage your workflow with *tmux*.

With all this in mind, I hope you enjoy reading *tmux Taster* and that you'll be sure to get in contact to let me know if, and how, this fantastic tool has helped you to improve your own workflow.



Terminal Multiplexer

The terminal emulators we use on a day-to-day basis do their job admirably. Having direct access to the shell environment is a fundamental way of life for most software engineers, as it gives us a sense of power and efficiency that cannot be matched by GUI-based applications and mouse interactions.

But even a reasonably modern terminal application can fall short in many areas (such as the Terminal.app on Mac OS X). As an example, imagine your terminal window is busy with a long-running process, and you want to interact with another part of your application or project (but you want to keep an eye on the current process, so that you know when it's finished). One way you could do this is to create a new terminal tab and simply switch back and forth. This isn't a very elegant solution or efficient, but it would work.

But some terminal emulators don't let you create tabs, so if you're one of those unfortunate souls, you'll have a worse option ahead of you, which is to use your mouse to reduce the size of your terminal and then open a new instance of your terminal application and resize that new instance so you can see both terminal windows at once (that is, if you even *have* a GUI; if you don't, then . . . well, you're *almost* out of luck).

Imagine a similar (but much more typical) example: you're a software engineer and you're writing new (or modifying existing) tests for your application. Ideally, you want to be able to make changes to your tests and application code while getting immediate feedback as to whether any of those changes have broken your test suite. You want a fast feedback loop. This is where being able to split your terminal window becomes a very useful tool.

But just being able to split a window into one or more screens isn't the only problem that has to be solved. Software engineers open many different types of files during a typical workday, and sometimes, you may find yourself in a situation in which the files you have open would be easier to read and modify if they were placed in a different layout that just wasn't possible using a standard terminal emulator.

In Figure 1-1, we can see we're not just splitting a terminal window into equal-sized chunks. We have window C, which spans the full width of the screen, while windows D, E, and F are a third of the overall screen, and windows A, B, G, and H split 50% of the available screen dimensions.

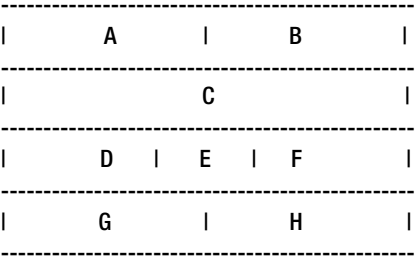


Figure 1-1. Example of a complex layout of terminal windows

We also want to be able to manipulate these windows very easily, through shorthand key bindings, and change their dimensions (and even change the layout of the windows, so that they are rearranged into a different format, to fit with the work we do later on in the day).

Humans are also creatures of habit, meaning that there will be a particular layout of windows that we find works best for us 90% of the time, and, so, being able to automate the creation of a particular layout is another feature that would be useful.

Finally, a tool that has the capability to allow me to share my screen seamlessly, so that I can pair program with another individual and have him/her take over the typing on my machine, is incredibly useful when you're a remote worker.

Having complete control over our terminal environment—how it looks and behaves—is a very powerful idea, and one that is possible through the use of an application called `tmux`. In the next section, I'll explain a little about what `tmux` is and means, as well as how to install and configure it.

■ **Note** Some readers may have heard of a recent terminal emulator called `iTerm2` (<http://iterm2.com> [for Mac OS X only]), which allows you to make split windows (among other features) but suffers from much less ubiquity than my tool of choice: `tmux`. For example, if I'm on a remote Linux server, I can quickly download and install `tmux` and be up and running. I can't do that with a program that is limited to a single operating system. As with `Vim`, a popular terminal based text editor, ubiquity is the key.

tmux

`tmux` is short for *[t]erminal [mu]ltiple[x]er*. A multiplexer is simply a fancy way of describing an application that lets you easily manage multiple terminal windows within one screen.

`tmux` runs a server/client architecture, meaning that when you start the application, it will fire up a single server, and every `tmux` instance you create on your machine will ultimately connect to that single `tmux` server. The benefit of this design is that while

your machine is running, you can *detach* a tmux “session” (i.e., close tmux but keep the details of that session open, as it’ll be stored on the tmux server running in a background process), so you can then *reattach* to the session at another time.

■ **Note** Visit <http://tmux.sourceforge.net/> for frequently asked questions and helpful information (such as documentation, IRC, and mailing list details), as well as to download binaries of the software.

tmux provides a lot of powerful features (most of which were described indirectly via the introduction of this chapter), which I’ve summarized into a few categories following:

- Ability to connect to existing local and remote sessions
- Advanced window and pane management
- Ability to move windows between different sessions
- Scripted automation

The usefulness of tmux truly reveals itself once you start utilizing it on a day-to-day basis and incorporating it firmly into your workflow. By the time we’re finished, you should have a much better understanding of the power and flexibility tmux provides and will wonder how you ever managed without it.

Terminology

Let’s take a brief detour to consider the terminology we’ll be using to describe tmux’s functionality. This will help to understand different tmux concepts as we move through the following chapters.

Prefix Command

The purpose of a multiplexer is to help you load multiple programs within a single window. Because you are effectively loading a program *within* a program (e.g., loading Vim inside a tmux window), tmux must avoid command conflicts with the subprograms being loaded. To do this, it introduces the concept of a prefix command, which helps to differentiate tmux commands from other programs you use.

■ **Note** The default prefix for tmux commands is <C-b>. You’ll notice that we shorten references to keyboard shortcuts. The principle is as follows: pressing the keys <Ctrl> and b at the same time can be represented as <C-b>. Similarly, pressing the keys <Esc> and 1 can be expressed as <Esc-1>. If we need to press any further keys, for example, let’s say we need to press <Ctrl> and b at the same time followed by d, then we express that as <C-b>d

Let's take a look at a quick example, to clarify what the prefix command does and why. I appreciate that we have yet to even open `tmux`, but the concept of a prefix key is fundamental to using `tmux` in the first place, and so I'm hoping you'll indulge me for just a moment longer, while I attempt to explain it.

Imagine we have `tmux` running, and for those who have never seen `tmux` before, you'll likely not notice much difference in your terminal's appearance (other than a bar at the foot of your screen, but I'll come back to this and describe what the bar is and what it means later), because visually, `tmux` should act as a container *around* your terminal.

Now, let's say we want to open a text file within the popular Vim text editor (e.g., `vim ~/foo.txt`) and modify the content by deleting a specific selection. Chances are you would open the file in Vim, find the content you want to delete, select it, and execute the `d` command (which is Vim's delete command). The problem with this process is that `tmux` assigns its own functionality to the `d` key (a command for detaching from a session; again, if this doesn't make sense, don't worry too much for now, as I'll cover sessions in due time). This is a perfect example of why `tmux` commands have a prefix: to avoid conflicts with other programs loaded within a `tmux` screen.

Due to `tmux` commands having a prefix, we can safely use Vim (or any other program) and not have to worry that executing a command within our subprograms will cause a side effect in `tmux`. In the previous scenario, we would detach from our `tmux` session using the command `<C-b>d` (where `<C-b>` is the prefix, followed by the `d` to indicate we wish to *detach* from the session).

Throughout the rest of the book, I'll refer to the prefix key `<C-b>` as just `<P>` (for *[P]refix*). This means the structure of all `tmux` commands in this book will take the form `<P>{key-binding}:command-prompt`. In the preceding example, in which we detached from the `tmux` session using the key binding `<C-b>d`, we would represent this using `<P>d` (see the following note regarding the "command prompt").

■ **Note** `tmux` provides a "command prompt" (similar in ways to Vim's `COMMAND-LINE/Ex` modes), which you can access by using `<P>:`, followed by a command. For example, `<P>:{command}`.

The reason for shortening the prefix command in the following chapters is, first, to make the commands shorter and easier to read, but more important, I'll be showing you how you can change the prefix key to be any key combination you like. So if you end up using this book as a reference, and you happen to have changed the prefix to be something else (let's say `<C-a>`), it would be easier to mentally replace `<C-b>` with your own prefix.

Help?

Unfortunately, if you need help with `tmux` commands, you don't have as rich a support feature as found in other programs, such as the Vim text editor (whose built-in `:help` documentation are very detailed and useful), but there are still a few options available to you, which are useful to know about.

Command and Key Binding References

tmux provides a quick reference list of all available key bindings, which you can access either via a key binding or the command-line prompt (or even from outside tmux itself).

To access the key binding reference via a key binding, you would use `<P>?`. To access this reference via the command prompt, you would use `<P>:list-keys`. Finally, you can also access this list from *outside* of tmux, using `tmux list-keys` (allowing you to utilize this information in some form of scripted automation, which I'll cover in more detail in Chapter 6).

The `list-keys` command will only display a list of available tmux key bindings. This does not include all commands that are executable within the tmux command prompt. For that list, you would use `<P>:list-commands` (or `tmux list-commands`, if you're outside of tmux); there is no key binding variation.

If you would like to see some extra information regarding each of the tmux sessions you have open, the following command will display this information for you: `<P>:info` (or `tmux info`, if you're outside of tmux).

Manual

Although the Internet has lots of useful information about how to do certain things in tmux, ultimately, the best resource of documentation is the official manual, which is linked to from the tmux web site and is directly accessible at www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man1/tmux.1.

Alternatively, and more usefully, you can access this documentation via your terminal, using the command `man tmux` (which also makes it much easier to filter and search through).

Message Feedback

One annoyance with tmux is when you execute a command incorrectly. What you'll notice happen is that tmux *tries* to be helpful by displaying a message telling you the correct format of the command it thinks you were trying to execute. But, unfortunately, that message only displays for a fraction of a second and then disappears, not leaving you enough time to see what the requirements of the command actually are.

Luckily, tmux provides a key binding that shows us the complete list of messages tmux has passed to us during our current session (the list of messages is displayed in ascending order, so the oldest messages are at the top, and the most recent at the bottom): `<P>~`. (You can also access this feature via the command prompt `<P>:show-messages`.)

Installation and Configuration

I mentioned earlier that one of the benefits of using tmux over other solutions is its ubiquity across different platforms. Installing tmux is remarkably simple for such a powerful and distributed piece of software, as we can see in the following options.