tao-of-tmux Documentation

Release v1.0.2

Tony Narlock

May 04, 2019

Contents

The Tao of tmux by Tony Narlock

This book is available for free to read on the web at <https://leanpub.com/the-tao-of-tmux/read>.

talen move it to sphinx.

Contents:

{frontmatter}

CHAPTER 1

Foreword

Nearly all my friends use tmux. I remember going out at night for drinks and the three of us would take a seat at a round table and take out our smart phones. This was back when phones still had physical “QWERTY” keyboards.

Despite our home computers being asleep or turned off, our usernames in the IRC channel we frequently visited persisted in the chatroom list. Our screens were lit by a kaleidoscope of colors on a black background. We ssh’d with ConnectBot into our cloud servers and reattached by running `screen(1)`. As it hit 2AM, our Turkish coffee arrived, the `|away` status indicator trailing our online nicknames disappeared.

It was funny noticing, even though we knew each other by our real names, we sometimes opted to call each other by our nicks. It’s something about how personal relationships, formed online, persist in real life.

It seemed as if it were orchestrated, but each of us fell into the same ebb and flow of living our lives. No one told us to do it, but bit by bit, we incrementally optimized our lifestyles, personally and professionally, to arrive at destinations seeming eerily alike.

Like many things in life, when we act on autopilot, we sometimes arrive at similar destinations. This is often unplanned.

So, when I write an educational book about a computer application, I hope to write it for human beings. Not to sell you on tmux, convince you to like it or hate it, but to tell you what it is and how some people use it. I’ll leave the rest to you.

1.1 About this book

I’ve helped thousands learn tmux through my free resource under the name The Tao of tmux, which I kept as part of the documentation for the tmuxp session manager. And now, it’s been expanded into a full-blown book with refined graphics, examples, and much more.

You do not need a book to use or understand tmux. If you want a technical manual, look at the manpage for tmux. Manpages, however, are rarely sufficient to wrap your brain around abstract concepts; they’re there for reference. This learning book is the culmination of years of explaining tmux to others online and in person.

In this book, we will break down tmux by its objects, from servers down to panes. It also includes a rehash of terminal facilities we use every day to keep us autodidacts up to speed with what is what. I've included numerous examples of projects, permissively licensed source code, and workflows designed for efficiency in the world of the terminal.

tmux is a tool I find useful. While I don't attach it to my personal identity, it's been part of my daily life for years. Besides the original resource, I've written a popular tmux starter configuration, a pythonic tmux library, and a tmux session manager.

I am writing this from vim running in a tmux pane, inside a window, in a session running on a tmux server, through a client.

A word to absolute beginners: Don't feel you need to grasp the concepts of the command line and terminal multiplexing in a single sitting. You have the choice of picking out concepts of tmux you like, according to your needs or interests. If you haven't installed tmux yet, please view the *Installation section* in the Appendix of the book.

Follow @TheTaoOfTmux for updates or share on Twitter!

1.2 Styles

Formatted text `like this` is source code.

Formatted text with a $ in front is a terminal command. `$ echo 'like this'`. The text can be typed into the console, without the dollar character. For more information on the meaning of the "dollar prompt", check out *What is the origin of the UNIX $ (dollar) prompt?* on Super User.

In tmux, shortcuts require a *prefix key* to be sent beforehand. For instance, `Prefix` + `d` will detach a tmux client from its session. This prefix, by default, is `<Ctrl-b>`, but users can override it. This is discussed in greater detail in *the prefix key* section and configuration.

1.3 How this book is structured

First, anything involving installation and hard technical details are in the Appendix. A lot of books use installation instructions as filler in the early chapters. For me, it's more of not wanting to confuse beginners.

For special circumstances, like *tmux on Windows 10*, I decided adding screenshots is best, since many readers may be more comfortable with a visual approach.

Thinking in tmux goes over what tmux does and how it relates to the GUI desktops on our computers. You'll understand the big picture of what tmux is and how it can make your life easier.

Terminal Fundamentals shows the text-based environments you'll be dealing with. It's great for those new to tmux, but also presents technical background for developers, who learned the ropes through examples and osmosis. At the end of this section, you'll be more confident and secure using the essential components underpinning a modern terminal environment.

Practical usage covers common bread-and-butter uses for you to use tmux immediately.

Server gives life to the unseen workhorse behind the scenes powering tmux. You'll think of tmux differently and may be impressed a client-server architecture could be presented to end users so seamlessly.

Sessions are the containers holding windows. You'll learn what sessions are and how they help organize your workspace in the terminal. You'll learn how to manipulate and rename and traverse sessions.

Windows are what you see when tmux is open in front of you. You'll learn how to rename and move windows.

Panes are a terminal in a terminal. This is where you get to work and do your magic! You'll learn how to create, delete, move between, and resize panes.

Configuration discusses customization of tmux and sets the foundation for how to think about `.tmux.conf` so you can customize your own.

Status bar and styling is devoted to the customization of the status line and colors in tmux. As a bonus, you'll even learn how to display system information like CPU and memory usage via the status line.

Scripting tmux goes into command *aliases* and the advanced and powerful *Targets* and *Formats* concepts.

Technical stuff is a glimpse at tmux source code and how it works under the hood. You may learn enough to impress colleagues who already use tmux. If you like programming on Unix-like systems, this one is for you.

Tips and tricks wraps up with a whirlwind of useful terminal tutorials you can use with tmux to improve day to day development and administration experience.

Cheatsheets are organized tables of commands, shortcuts, and formats grouped by section.

1.4 Donations

If you enjoy my learning material or my open source software projects, please consider donating. Donations go directly to me and my current and future open source projects and are not squandered. Visit http://www.git-pull.com/support.html for ways to contribute.

1.5 Formats

This book is available for sale on Leanpub and Amazon Kindle.

It's also available to read for free on the web.

1.6 Errata {#errata}

This is my first book. I am human and make mistakes.

If you find errors in this book, please submit them to me at tao.of.tmux nospam git-pull.com.

You can also submit a pull request via https://github.com/git-pull/tao-of-tmux.

I will update digital versions of the book with the changes where applicable.

1.7 Thanks

Thanks to the contributors for spotting errors in this book and submitting errata through GitHub. In addition, readers like Graziano Misuraca, who looked through the book closely, providing valuable feedback.

Some copy, particularly in *cheatsheets*, comes straight out of the manual of tmux, which is ISC-licensed.

1.8 Book Updates and tmux changes

This book was written for tmux 2.3, released September 2016.

As of January 2017, it's trivial to push out minor changes to Leanpub. Kindle is harder.

tmux does intermittently receive updates. I've accommodated many over the past 5 years on my personal configurations and software libraries set with continuous integration tests against multiple tmux versions. Sometimes, publishers overplay version numbers to make it seem as if it's worth striking a new edition of a book over it. It's effective for them, but I'd rather be honest to my readership.

If you're considering keeping up to date with new features and adjustments to tmux, the `CHANGES` file in the project source serves as a way to see what's updated between official releases.

{mainmatter}

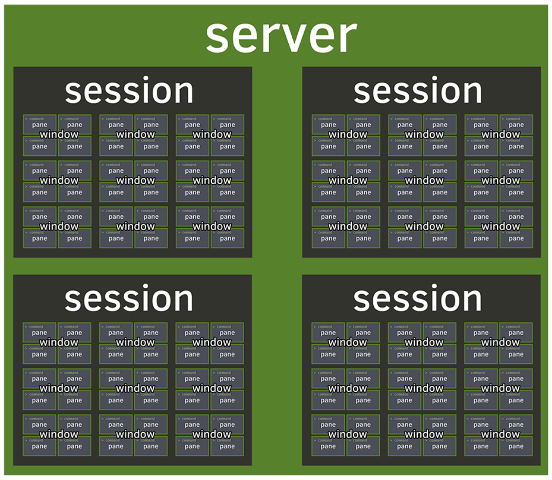
CHAPTER 2

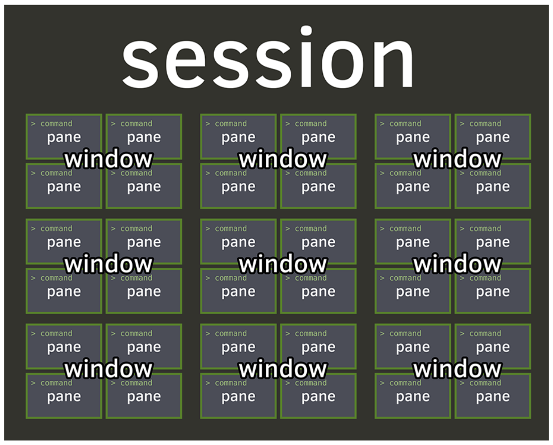
Thinking in tmux {#thinking-tmux}

In the world of modern computing, user interaction has 2 realms:

1. The text realm
2. The graphical realm

tmux lives in the graphical realm in which fixed-width fonts appear in a rectangular grid in a window, like in a terminal from the 1980s.



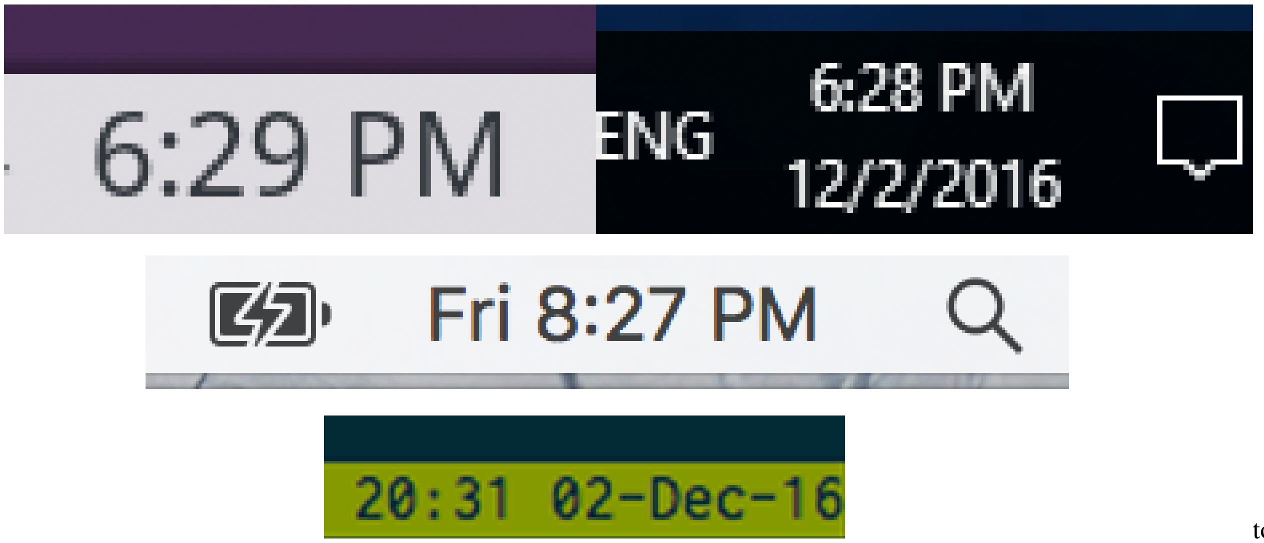


2.1 Window manager for the terminal

tmux is to the console what a desktop is to GUI apps. It's a world inside the text dimension. Inside tmux, you can:

- multitask inside the terminal, run multiple applications
- have multiple command lines (pane) in the same window
- have multiple windows (window) in the workspace (session)
- switch between multiple workspaces, like virtual desktops

Just like in a graphical desktop environment, they throw in a clock, too.



{width=75%} top-left: KDE. top-right: Windows 10. center: macOS Sierra. bottom: tmux 2.3 default status bar.

2.2 Multitasking

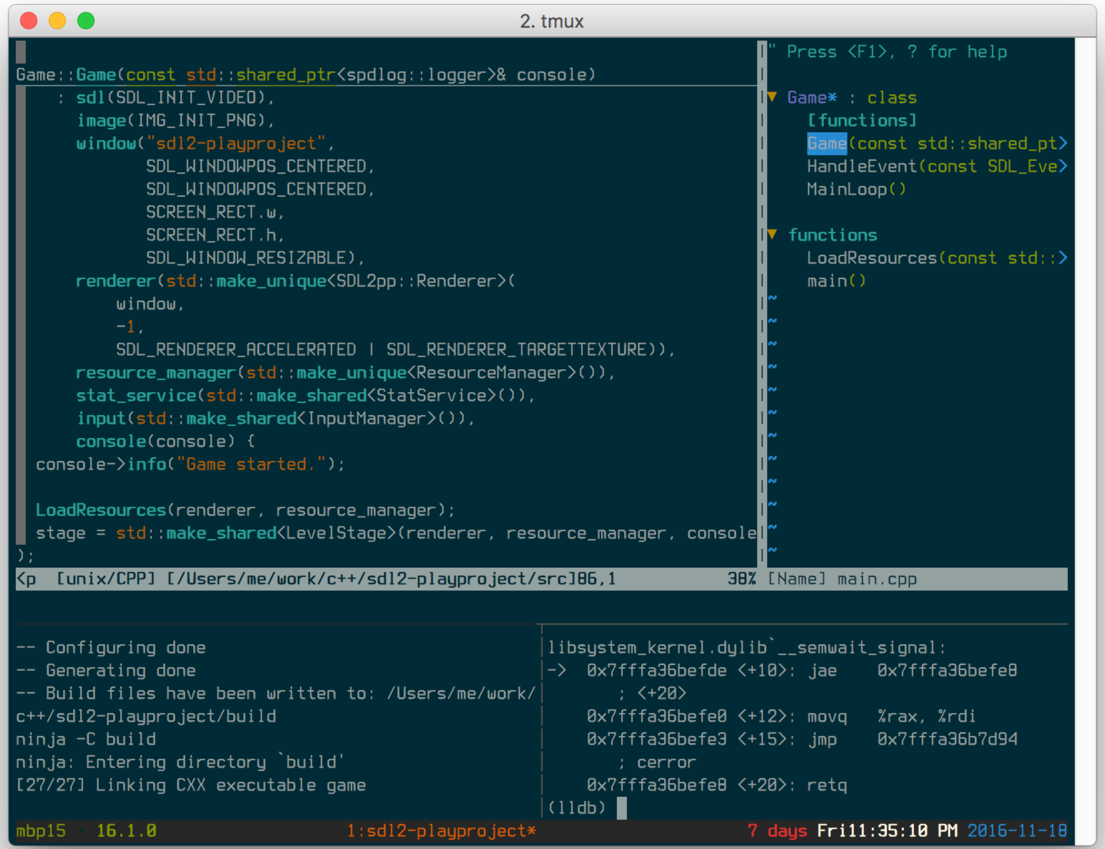
tmux allows you to keep multiple terminals running on the same screen. After all, the abbreviation "tmux" comes from - **T**erminal **Mu**ltiple**x**er.

In addition to multiple terminals on one screen, tmux allows you to create and link multiple "windows" within the confines of the tmux session you attached.

Even better, you can copy and paste and scroll. No requirement for graphics either, so you have full power, even if you're SSH'ing or on a system without a display server such as X.

Here are a few common scenarios:

- Running `$ tail -F /var/log/apache2/error.log` in a pane to get a live stream of the latest system events.
- Running a file watcher, like watchman, gulp-watch, grunt-watch, guard, or entr. On file change, you could do stuff like:
  - rebuild LESS or SASS files, minimize CSS and/or assets and static files
  - lint with linters, like cpplint, Cppcheck, rubocop, ESLint, or Flake8
  - rebuild with `make` or `ninja`
  - reload your Express server
  - run any other custom command of your liking
- Keeping a text editor, like vim, emacs, pico, nano, etc., open in a main pane, while leaving two others open for CLI commands and building via `make` or `ninja`.



```
2. tmux
```

```
Game::Game(const std::shared_ptr<spdlog::logger>& console)
    : sdl(SDL_INIT_VIDEO),
      image(IMG_INIT_PNG),
      window("sdl2-playproject",
             SDL_WINDOWPOS_CENTERED,
             SDL_WINDOWPOS_CENTERED,
             SCREEN_RECT.w,
             SCREEN_RECT.h,
             SDL_WINDOW_RESIZABLE),
      renderer(std::make_unique<SDL2pp::Renderer>(
          window,
          -1,
          SDL_RENDERER_ACCELERATED | SDL_RENDERER_TARGETTEXTURE)),
      resource_manager(std::make_unique<ResourceManager>()),
      stat_service(std::make_shared<StatService>()),
      input(std::make_shared<InputManager>()),
      console(console) {
  console->info("Game started.");

  LoadResources(renderer, resource_manager);
  stage = std::make_shared<LevelStage>(renderer, resource_manager, console
);
<p  [unix/CPP] [/Users/me/work/c++/sdl2-playproject/src]86,1          38%
```

```
" Press <F1>, ? for help

▼ Game* : class
    [functions]
    Game(const std::shared_pt>
    HandleEvent(const SDL_Eve>
    MainLoop()

▼ functions
    LoadResources(const std::>
    main()
~
~
~
~
~
~
~
~
~
~
~
~
[Name] main.cpp
```

```
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/me/work/
c++/sdl2-playproject/build
ninja -C build
ninja: Entering directory `build'
[27/27] Linking CXX executable game
```

```
libsystem_kernel.dylib`__semwait_signal:
->  0x7fffa36befde <+10>: jae    0x7fffa36befe8
      ; <+20>
    0x7fffa36befe0 <+12>: movq   %rax, %rdi
    0x7fffa36befe3 <+15>: jmp    0x7fffa36b7d94
      ; cerror
    0x7fffa36befe8 <+20>: retq
(lldb)
```

```
mbp15 · 16.1.0            1:sdl2-playproject*            7 days Fri11:35:10 PM 2016-11-18
```

vim + building a C++ project w/ CMake + Ninja using entr to rebuild on file changes, LLDB bottom right

With tmux, you quickly have the makings of an IDE! And on your terms.

2.3 Keep your applications running in the background

Sometimes, GUI applications will have an option to be sidelined to the system tray to run in the background. The application is out of sight, but events and notifications can still come in, and the app can be instantly brought to the foreground.

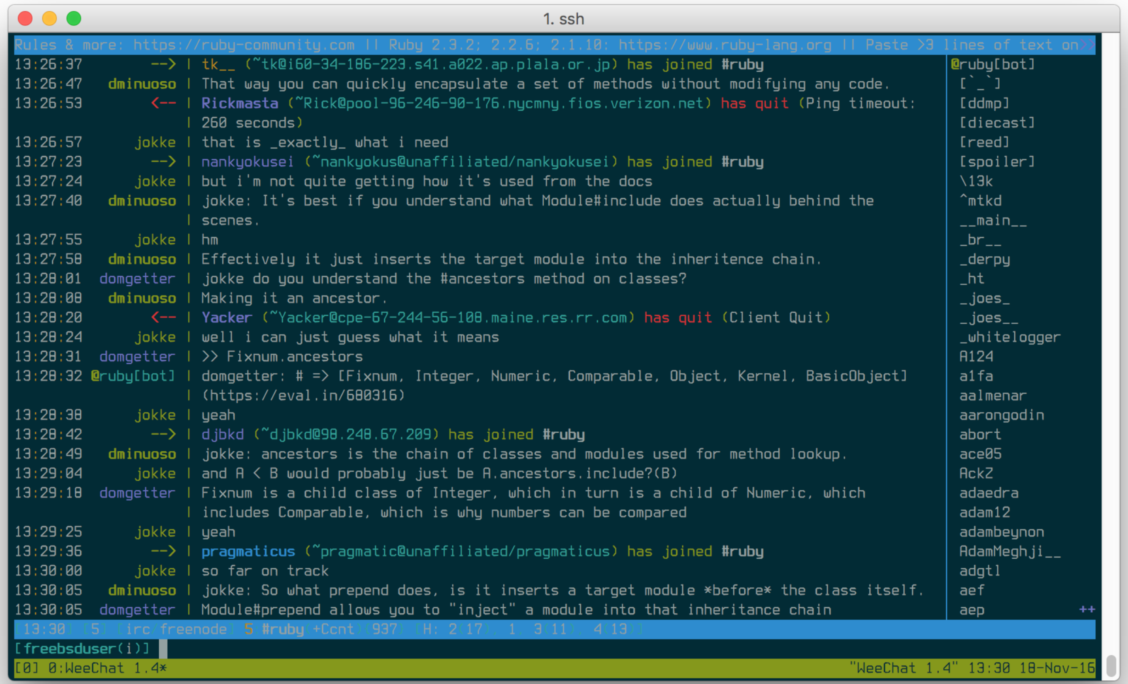
In tmux, a similar concept exists, where we can "detach" a tmux session.

Detaching can be especially useful on:

- Local machines. You start all your normal terminal applications within a tmux session, you restart X. Instead of losing your processes as you normally would if you were using an X terminal, like xterm or konsole, you'd be able to `tmux attach` after and find all the processes inside that were alive and kicking all along.
- Remote SSH applications and workspaces you run in tmux. You can detach your tmux workspace at work before you clock out, then the next morning, reattach your session. Ahhh. Refreshing. :)
- Those servers you rarely log into. Perhaps, a cloud instance you log into 9 months later, and as a reflex, `tmux attach` to see if there is anything on there. And boom, you're back in a session you've forgotten about, but still jogs your memory to what you were tweaking or fixing. It's like a hack to restore your memory.

2.4 Powerful combos

Chatting on irssi or weechat, one of the "classic combos", along with a bitlbee server to manage AIM, MSN, Google Talk, Jabber, ICQ, even Twitter. Then, you can detach your IRC and "idle" in your favorite channels, stay online on instant messengers, and get back to your messages when you return.



```
1. ssh
Rules & more: https://ruby-community.com || Ruby 2.3.2; 2.2.6; 2.1.10: https://www.ruby-lang.org || Paste >3 lines of text on>>
13:26:37        --> | tk__ (~tk@i60-34-186-223.s41.a022.ap.plala.or.jp) has joined #ruby              @ruby[bot]
13:26:47    dminuoso | That way you can quickly encapsulate a set of methods without modifying any code.   [`_`]
13:26:53        <-- | Rickmasta (~Rick@pool-96-246-90-176.nycmny.fios.verizon.net) has quit (Ping timeout:  [ddmp]
                    | 260 seconds)                                                                       [diecast]
13:26:57       jokke | that is _exactly_ what i need                                                      [reed]
13:27:23        --> | nankyokusei (~nankyokus@unaffiliated/nankyokusei) has joined #ruby                 [spoiler]
13:27:24       jokke | but i'm not quite getting how it's used from the docs                             \13k
13:27:40    dminuoso | jokke: It's best if you understand what Module#include does actually behind the     ^mtkd
                    | scenes.                                                                            __main__
13:27:55       jokke | hm                                                                                _br__
13:27:58    dminuoso | Effectively it just inserts the target module into the inheritence chain.         _derpy
13:28:01   domgetter | jokke do you understand the #ancestors method on classes?                         _ht
13:28:08    dminuoso | Making it an ancestor.                                                            _joes_
13:28:20        <-- | Yacker (~Yacker@cpe-67-244-56-108.maine.res.rr.com) has quit (Client Quit)        _joes__
13:28:24       jokke | well i can just guess what it means                                                _whitelogger
13:28:31   domgetter | >> Fixnum.ancestors                                                               A124
13:28:32 @ruby[bot] | domgetter: # => [Fixnum, Integer, Numeric, Comparable, Object, Kernel, BasicObject]  a1fa
                    | (https://eval.in/680316)                                                           aalmenar
13:28:38       jokke | yeah                                                                              aarongodin
13:28:42        --> | djbkd (~djbkd@98.248.67.209) has joined #ruby                                      abort
13:28:49    dminuoso | jokke: ancestors is the chain of classes and modules used for method lookup.       ace05
13:29:04       jokke | and A < B would probably just be A.ancestors.include?(B)                          AckZ
13:29:18   domgetter | Fixnum is a child class of Integer, which in turn is a child of Numeric, which    adaedra
                    | includes Comparable, which is why numbers can be compared                          adam12
13:29:25       jokke | yeah                                                                              adambeynon
13:29:36        --> | pragmaticus (~pragmatic@unaffiliated/pragmaticus) has joined #ruby                 AdamMeghji__
13:30:00       jokke | so far on track                                                                   adgtl
13:30:05    dminuoso | jokke: So what prepend does, is it inserts a target module *before* the class itself.  aef
13:30:05   domgetter | Module#prepend allows you to "inject" a module into that inheritance chain        aep        ++
[13:30] [5] [irc/freenode] 5 #ruby(+Ccnt){937} [H: 2(17), 1, 3(11), 4(13)]
[freebsduser(i)]
[0] 0:WeeChat 1.4*                                                             "WeeChat 1.4" 13:30 18-Nov-16
```

Chatting on weechat w/ tmux

Some keep development services running in a session. Hearty emphasis on *development*, you probably will want to daemonize and wrap your production web applications, using a tool like supervisor, with its own safe environmental settings.

You can also have multiple users attach their clients to the same sessions, which is great for pair programming. If you were in the same session, you and the other person would see the same thing, share the same input, and the same active window and pane.

The above are just examples; any general workspace you'd normally use in a terminal could work, especially projects or repetitive efforts you multitask on. The tips and tricks section will dive into specific flows you can use today.

Q> ### Do tmux sessions persist after a system restart? Q> Q> Unfortunately, no. A restart will kill the tmux server and any processes Q> running within it. Q> Q> Thankfully, the modern server can stay online for a long time. Even for Q> consumer laptops and PC's with a day or two uptime, having tmux persist Q> tasks for organizational purposes is satisfactory to run it. Q> Q> It comes as a disappointment, because some are interested in being able to Q> persist a tree of processes after restart. It goes out of the scope of what Q> tmux is meant to do. Q> Q> For tasks you repeat often, you can always use a tool, like Q> tmuxp, tmuxinator, Q> or teamocil, to resume common Q> sessions. Q> Q> Besides session managers, tmux-resurrect Q> attempts to preserve running programs, working directories, and Q> so on within tmux. The benefit with tmux-resurrect is there's no JSON/YAML Q> config needed.

2.5 Summary

tmux is a versatile addition to your terminal toolbelt. It helps you cover the gaps between multitasking and workspace organization you'd otherwise lose, since there's no GUI. In addition, it includes a nice ability to detach workspaces to the background and reattach later.

In the next chapter, we will touch on some terminal basics before diving deeper into tmux.

CHAPTER 3

Terminal fundamentals {#terminal-fundamentals}

Before getting into tmux, a few fundamentals of the command line should be reviewed. Often, we're so used to using these out of street smarts and muscle memory, a great deal of us never see the relation of where these tools stand next to each other.

Seasoned developers are familiar with Zsh, Bash, iTerm2, konsole, /dev/tty, shell scripting, and so on. If you use tmux, you'll be around these all the time, regardless whether you're in a GUI on a local machine or SSH'ing into a remote server.

If you want to learn more about how processes and TTYs work at the kernel level (data structures and all), the book *The Design and Implementation of the FreeBSD Operating System (2nd Edition)* by Marshall Kirk McKusick is nice, particularly, Chapter 4, *Process Management* and Section 8.6, *Terminal Handling*. *The TTY demystified* by Linus Åkesson (available online) dives into the TTY and is a good read.

Much more exists to glean off the history of Unix, 4.2 BSD, etc. I probably could have a coffee / tea with you discussing it for hours. You could look at it from multiple perspectives (The C Language, anything from the Unix/BSD lineage, etc.), and some clever fellow would likely chime in, mentioning Linux, GNU, and so on. It's like *Game of Thrones*; there's multiple story arcs you can follow, some of which intersect. A few good video resources would be *A Narrative History of BSD* by Marshall Kirk McKusick, *The UNIX Operating System* by AT&T, *Early days of Unix and design of sh* by Stephen R. Bourne.

3.1 POSIX standards

Operating systems like macOS (formerly OS X), Linux, and the BSDs, follow something similar to the POSIX specification in terms of how they square away various responsibilities and interfaces of the operating system. They're categorized as "Mostly POSIX-compliant".

In daily life, we often break compatibility with POSIX standards for reasons of sheer practicality. Operating systems, like macOS, will drop you right into Bash. `make(1)`, a POSIX standard, is GNU Make on macOS by default. Did you know, as of September 2016, POSIX Make has no conditionals?

I'm not saying this to take a run at purists. As someone who tries to remain compatible in my scripting, it gets hard to do simple things after a while. On FreeBSD, the default Make (PMake) uses dots between conditionals:

{line-numbers=off} .IF

```
.ENDIF
```

But on most Linux systems and macOS, GNU Make is the default, so they get to do:

{line-numbers=off} IF

```
ENDIF
```

This is one of the many tiny inconsistencies that span operating systems, their userlands, their binary / library / include paths, and adherence / interpretation of the Filesystem Hierarchy Standard or whether they follow their own.

I> **Find your path** I> I> Most operating systems inspired by Unix (BSD's, macOS, Linux) will allow you I> to get the info of your systems' filesystem hierarchy via `hier(7)`. I> I> {language=shell, line-numbers=off} I> $ man hier

These differences add up. A good deal of software infrastructure out there exists solely to abstract the differences across them. For example: CMake, Autotools, SFML, SDL2, interpreted programming languages, and their standard libraries are dedicated to normalizing the banal differences across BSD-derivatives and Linux distributions. Many, many `#ifdef` preprocessor directives in your C and C++ applications. You want open source, you get choice, but be aware; there's a lot of upkeep cost in keeping these upstream projects (and even your personal ones) compatible. But I digress, back to terminal stuff.

Why does it matter? Why bring it up? You'll see this stuff everywhere. So, let's separate the usual suspects into their respective categories.

3.2 Terminal interface

The terminal interface can be best introduced by citing official specification, laying out its technical properties, interfaces, and responsibilities. This can be viewed in its POSIX specification.

This includes TTYs, including text terminals and X sessions within them. On Linux / BSD systems, you can switch between sessions via `<ctrl-alt-F1>` through `<ctrl-alt-F12>`.

3.3 Terminal emulators

GUI Terminals: Terminal.app, iterm, iterm2, konsole, lxterm, xfce4-terminal, rxvt-unicode, xterm, roxterm, gnome terminal, cmd.exe + bash.exe

3.4 Shell languages {#shell-languages}

Shell languages are programming languages. You may not compile the code into binaries with `gcc` or `clang`, or have shiny npm package manager for them, but a language is a language.

Each shell interpreter has its own language features. Like with shells, many will resemble the POSIX shell language and strive to be compatible with it. Zsh and Bash should be able to understand POSIX shell scripts you write, but not the other way around (we will cover this in *shell interpreters*).

The first line of shell file is the shebang statement, which points to the interpreter to run the script in. They normally use the `.sh` extension, but they can also be `.zsh`, `.csh` and so on if they're for a specific interpreter.

Zsh scripts are implemented by the Zsh shell interpreter, Bash scripts by Bash. But the languages are not as closely regulated and standardized as, say, C++'s standards committee workgroups or python's PEPs. Bash and Zsh take features from Korn and C Shell's languages, but without all the ceremony and bureaucracy other languages espouse.

3.5 Shell interpreters (Shells) {#shells}

Examples: POSIX sh, Bash, Zsh, csh, tcsh, ksh, fish

Shell interpreters *implement* the shell language. They are a layer on top of the kernel and are what allow you, interactively, to run commands and applications inside them.

As of October 2016, the latest POSIX specification covers in technical detail the responsibilities of the shell.

For shells and operating systems: each distro or group does their own darn thing. On most Linux distributions and macOS, you'll typically be dropped into Bash.

On FreeBSD, you may default to a plain vanilla `sh` unless you specify otherwise during the installation process. In Ubuntu, `/bin/sh` used to be `bash` (Bourne Again Shell) but was replaced with `dash` (Debian Almquist Shell). So, here, you are thinking "hmm, `/bin/sh`, probably just a plain old POSIX shell"; however, system startup scripts on Ubuntu used to allow non-POSIX scripting via Bash. This is because specialty *shell languages*, such as Bash and Zsh, add helpful and practical features, but they're not portable. For instance, you would need to install the Zsh interpreter across all your systems if you rely on Zsh-specialized scripting. If you conformed with POSIX shell scripting, your scripting would have the highest level of compatibility at the cost of being more verbose.

Recent versions of macOS include Zsh by default. Linux distributions typically require you to install it via package manager and install it to `/usr/bin/zsh`. BSD systems build it via the port system, `pkg(8)` on FreeBSD, or `pkg_add(1)` on OpenBSD, and it will install to `/usr/local/bin/zsh`.

It's fun to experiment with different shells. On many systems, you can use `chsh -s` to update the default shell for a user.

The other thing to mention is, for `chsh -s` to work, you typically need to have it added to `/etc/shells`.

3.6 Summary

To wrap it up, you will hear people talking about shells all the time. Context is key. It could be:

- A generic way to refer to any terminal you have open. "Type `$ top` into your shell and see what happens." (Press q to quit.)
- A server they have to log into. Before the era of the cloud, it would be popular for small hosts to sell "C Shells" with root access.
- A shell within a tmux *pane*.
- If scripting is mentioned, it is likely either the script file, an issue related to the scripts' behavior, or something about the shell language.

But overall, after this overview, go back to doing what you're doing. If shell is what people say and they understand it, use it. The backing you have here should make you more confident in yourself. These days, it's an ongoing battle catching our street smarts up with book smarts.

In the next chapter, we will touch some terminal basics before diving deeper into tmux.

CHAPTER 4

Practical usage {#practical-usage}

This is the easiest part; open your terminal and type `tmux`, hit enter.

{language=shell, line-numbers=off} $ tmux

You're in tmux.

4.1 The prefix key {#prefix-key}

The *prefix* is how we send commands into tmux. With this, we can split windows, move windows, switch windows, switch sessions, send in custom commands, you name it.

And it's a hump we have to get over.

It's kind of like *Street Fighter*. In this video game, the player inputs a combination of buttons in sequence to perform flying spinning kicks and shoot fireballs; sweet. As the player grows more accustomed with the combos, they repeat moves by intuition, since they develop muscle memory.

Without understanding how to send *command sequences* to tmux via the prefix key, you'll be dead in the water.

Key sequences will come up later if you use Vim, Emacs, or other TUI (Terminal User Interface) applications. If you haven't internalized the concept, let's do it now. Prior experience command sequences in TUI/GUI applications will come in handy.

When you memorize a key combo, it's one less time you'll be moving your hand away from the keyboard to grab your mouse. You can focus your short-term memory on getting stuff done, resulting in fewer mistakes.

Q> ### Coming from `GNU Screen`? Q> Q> Your tmux prefix key can be set via your tmux configuration later! In Q> your `~/.tmux.conf` file, set the `prefix` option: Q> Q> {language=shell, line-numbers=off} Q> set-option -g prefix C-a Q> Q> This will set the prefix key to `screen(1)`'s (another terminal Q> multiplexer's) prefix key.

The default leader prefix is `<Ctrl-b>`. While holding down the `control` key, press `b`.

X> ### Sending tmux commands X> X> Practice: X> X> 1. Press `control` key down and *hold it*. X> 2. Press `b` and hold it. X> 3. Release both keys at the same time. X> X> Try it a few times. It may feel unnatural until you've done it a couple X> times, which is normal when memorizing shortcuts. X> X> Now, let's try something: X> X>

`<Ctrl-b> d`. So, X> X> 1. Press `control` key down and *hold it*. X> 2. Press `b` and hold it. X> 3. Release both keys at the same time. X> 4. Hit `d`! X> X> You've sent tmux your first command, and you're now outside of tmux!

You've detached the tmux session you were in. You can reattach via `$ tmux attach`.

4.1.1 Nested tmux sessions

You can also send the prefix key to *nested* tmux sessions. For instance, if you're inside a tmux client on a *local* machine and you SSH into a *remote* machine in one of your panes, on the remote machine, you can attach the client via `tmux attach` as you normally would. To send the prefix key to the machine's tmux client, not your local one, hit the prefix key again.

So, if your prefix key is the default, `<Ctrl-b>`, do `<Ctrl-b>` + `b` again, *then* hit the shortcut for what you want to do.

Example: If you wanted to create a window on the remote machine, which would normally be `<Ctrl-b>` + `c` locally, it'd be `<Ctrl-b>` + `b` + `c`.

Hereinafter, the book will refer to shortcuts by `Prefix`. Instead of `<Ctrl-b> + d`, you will see `Prefix` + `d`.

4.2 Session persistence and the server model

If you use Linux or a similar system, you've likely brushed through Job Control, such as `fg(1)`, `jobs(1)`. tmux behavior feels similar, like you ran `<Ctrl-z>` except, technically, you were in a "job" all along. You were just using a client to view it.

Another way of understanding it: `<Ctrl-b>` + `d` closed the client connection, therefore, 'detached' from the session.

Your tmux client disconnected from the server instance. The session, however, is still running in the background.

4.3 It's all commands

Multiple roads can lead you to the same behavior. *Commands* are what tmux uses to define instructions for setting options, resizing, renaming, traversing, switching modes, copying and pasting, and so forth.

- *Configs* are the same as automatically running commands via `$ tmux command`.
- Internal tmux commands via `Prefix` + `:` prompt.
- Settings defined in your configuration can also set shortcuts, which can execute commands via keybindings via `bind-key`.
- Commands called from CLI via `$ tmux cmd`
- To pull it all together, *source code* files are prefixed `cmd-`.

4.4 Summary

We've established tmux automatically creates a server upon starting it. The server allows you to detach and later reattach your work. The keyboard sequences you send to tmux require understanding how to send the prefix key.

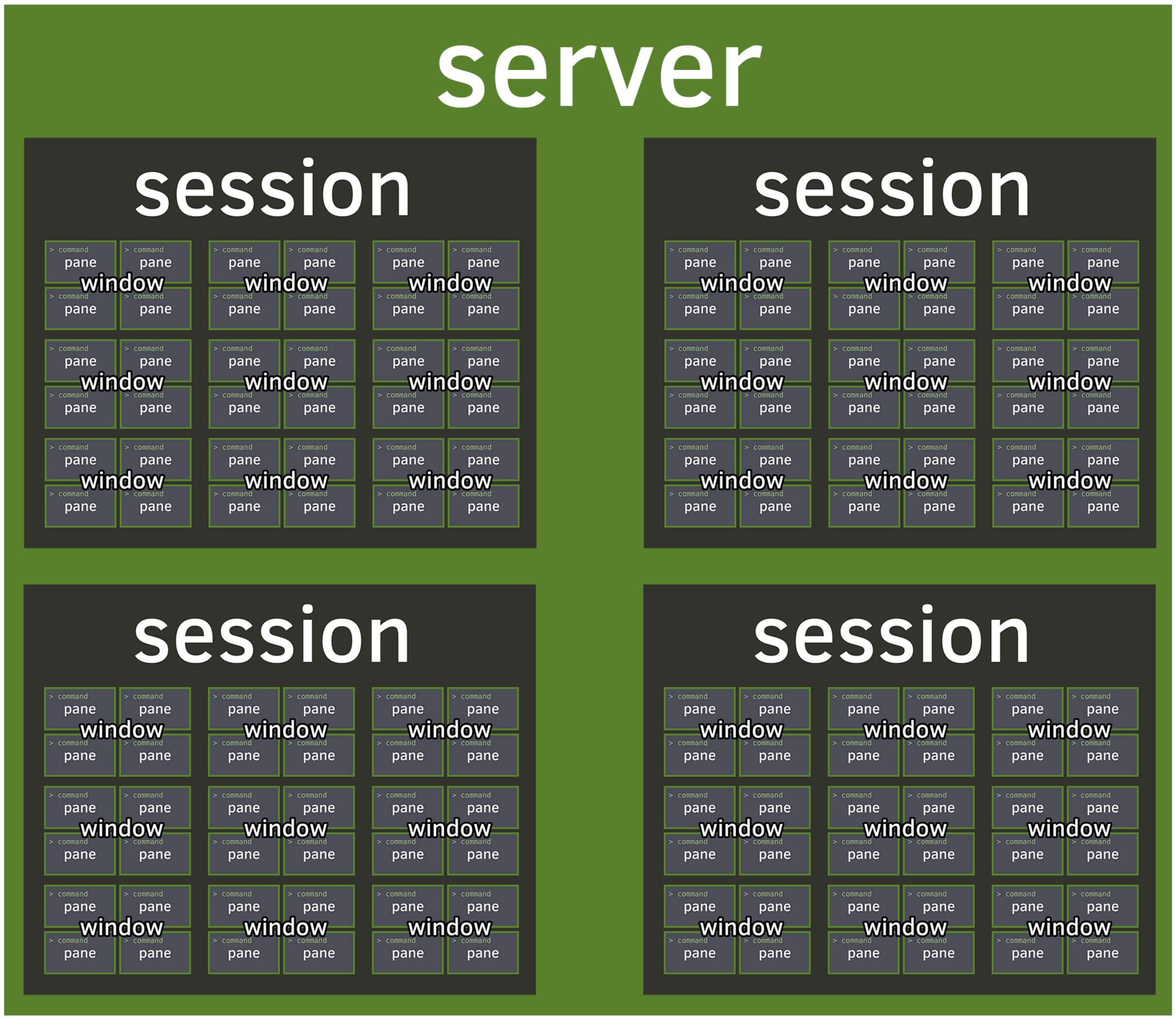
Keyboard sequences, configuration, and command line actions all boil down to the same core commands inside tmux. In our next chapter, we will cover the server.

CHAPTER 5

Server {#server}

The server holds *sessions* and the *windows* and *panes* within them.

When tmux starts, you are connected to a server via a socket connection. What you see presented in your shell is merely a client connection. In this chapter, we uncover the invisible engine enabling your terminal applications to persist for months or even years at a time.



{width=90%}

5.1 What? tmux is a server?

Often, when "server" is mentioned, what comes to mind for many may be rackmounted hardware; to others, it may be software running daemonized on a server and managed through a utility, like upstart, supervisor, and so on.

Unlike web or database software, tmux doesn't require specialized configuration settings or creating a service entry to start things.

tmux uses a client-server model, but the server is forked to the background for you.

5.2 Zero config needed

You don't notice it, but when you use tmux normally, a server is launched and being connected via a client.

tmux is so streamlined, the book could continue to explain usage and not even mention servers. But, I'd rather you have a true understanding of how it works on systems. The implementation feels like magic, while living up to the unix expectations of utilitarianism. One cannot deny it's exquisitely executed from a user experience standpoint.

How is it utilitarian? We'll go into it more in future chapters, where we dive into *Formats*, *Targets*, and tools, such as libtmux I made, which utilize these features.

It surprises some, because servers often beget a setup process. But servers being involved doesn't entail hours of configuration on each machine you run on. There's no setup.

When people think server, they think pain. It invokes an image of digging around `/etc/` for configuration files and flipping settings on and off just to get basic systems online. But not with tmux. It's a server, but in the good way.

5.3 Stayin' alive

The server part of tmux is how your sessions can stay alive, even after your client is detached.

You can detach a tmux session from an SSH server and reconnect later. You can detach a tmux session, stop your X server in Linux/BSD, and reattach your tmux session in a TTY or new X server.

The tmux server won't go away until all sessions are closed.

5.4 Servers hold sessions

One server can contain one or multiple *sessions*.

Starting tmux after a server already is running will create a new session inside the existing server.

W> ### Advanced: Multiple servers W> W> tmux is nimble. To use a separate server, pass in the `-L` flag to any W> command. W> W> `tmux -L moo` - connect to server under socket name "moo" and attach W> a new session. Create server if none already exists for socket. W> W> `tmux -L moo attach` will attempt to re-attach a session if one exists.

5.5 How servers are "named"

The default name for the server is `default`, which is stored as a socket in `/tmp`. The default directory for storing this can be overridden via setting the `TMUX_TMPDIR` environment variable.

So, something like:

{language=shell, line-numbers=off} $ export TMUX_TMPDIR=$HOME $ tmux

Will give you a tmux directory created within your `$HOME` folder. On OS X, your home folder will probably be something like `/Users/yourusername`. On other systems, it may be `/home/yourusername`. If you want to find out, type `$ echo $HOME`.

5.6 Clients

Servers will have clients (you) connecting to them.

When you connect to a session and see windows and panes, it's a client connection into tmux.

You can retrieve a list of active client connections via:

{language=shell, line-numbers=off} $ tmux list-clients

These commands and the other `list-` commands, in practice, are rare. But, they are part of tmux scriptability should you want to get creative. The *scripting tmux* chapter will cover this in greater detail.

5.7 Clipboard {#clipboard}

tmux clients wield a powerful clipboard feature to copy and paste across sessions, windows, and panes.

Much like vi, tmux handles copying as a mode in which a pane is temporarily placed. When inside this mode, text can be selected and copied to the *paste buffer*, tmux's clipboard.

The default key to enter copy mode is `Prefix` + `[`.

1. From within, use `[space]` to enter copy mode.
2. Use the arrow keys to adjust the text to be selected.
3. Press `[enter]` to copy the selected text.

The default key to paste the text copied is `Prefix` + `]`.

I> *Vi-like copy-paste* I> I> In your *config*, put this: I> I> {language=shell, line-numbers=off} I> # Vi copypaste mode I> set-window-option -g mode-keys vi I> bind-key -t vi-copy 'v' begin-selection I> bind-key -t vi-copy 'y' copy-selection

In addition to the "copy mode", tmux has advanced functionality to programmatically copy and paste. Later in the book, the *Capturing pane content* section in the *Scripting tmux* chapter goes into `$ tmux capture-pane` and how you can use *targets* to copy pane content into your paste buffer or files with `$ tmux save-buffer`.

5.8 Summary

The server is one of the fundamental underpinnings of tmux. Initialized automatically to the user, it persists by forking into the background. Running behind the scenes, it ensures sessions, windows, panes, and buffers are operating, even when the client is detached.

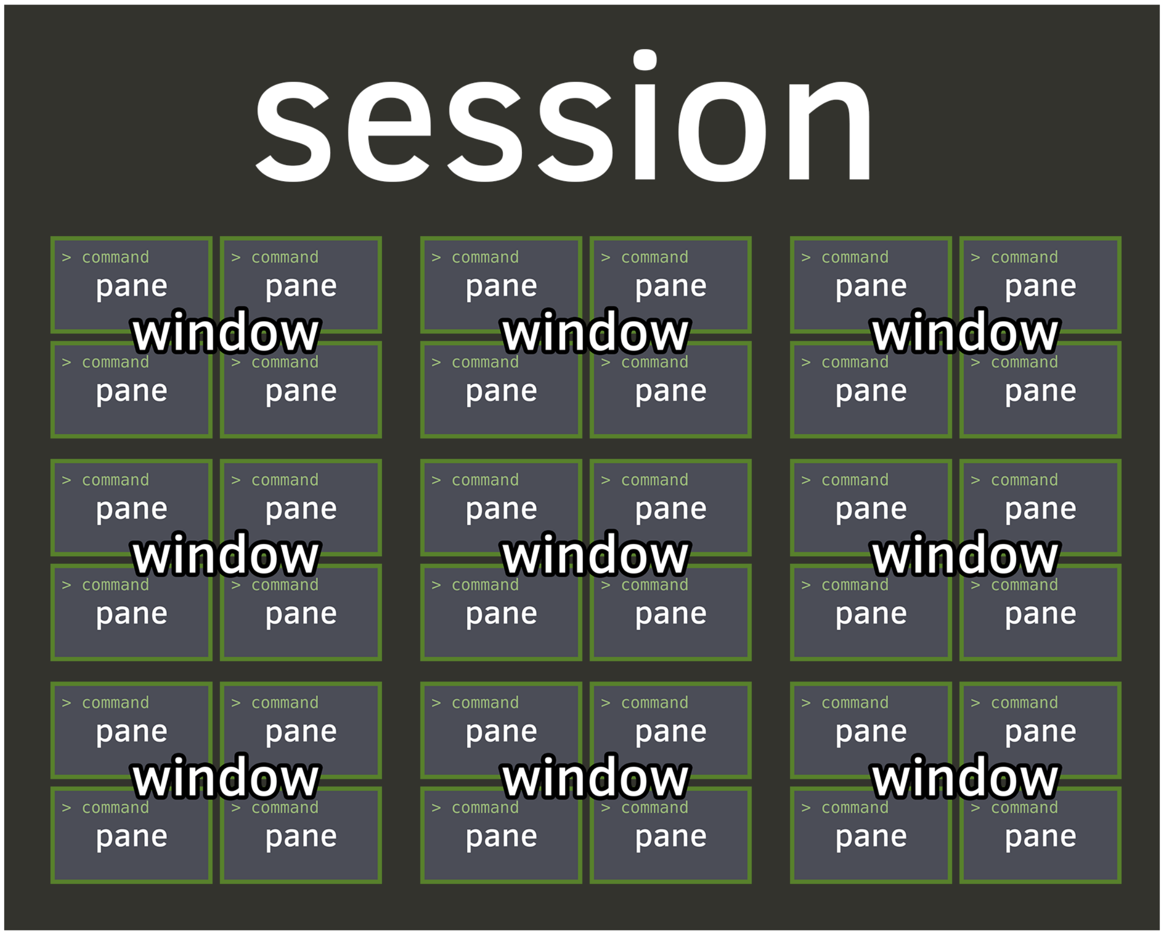
The server can hold one or more *sessions*. You can copy and paste between sessions via the clipboard. In the next chapter, we will go deeper into the role sessions play and how they help you organize and control your terminal workspace.

CHAPTER 6

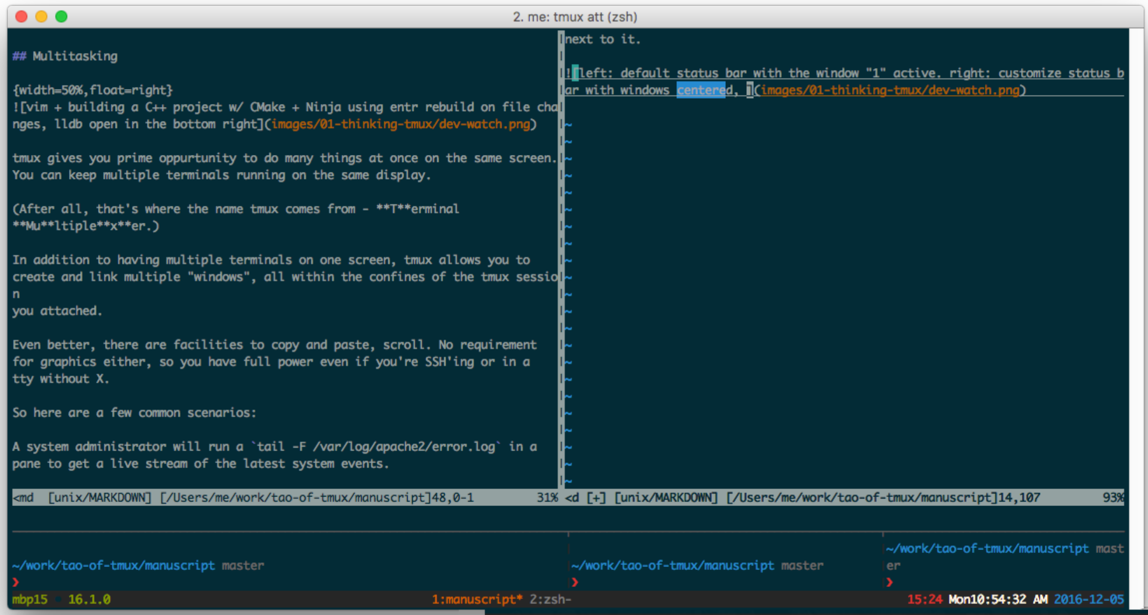
Sessions {#sessions}

Welcome to the session, the highest-level entity residing in the *server* instance. Server instances are forked to the background upon starting a fresh instance and reconnected to when reattaching sessions. Your interaction with tmux will have *at least* one session running.

A session holds one or more *windows*.



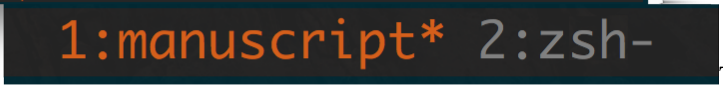
The active window will have a ⋆ symbol next to it.



```
2. me: tmux att (zsh)

## Multitasking                                                            |next to it.
                                                                           |
{width=50%,float=right}                                                    |![left: default status bar with the window "1" active. right: customize status b
![vim + building a C++ project w/ CMake + Ninja using entr rebuild on file cha|ar with windows centered, ](images/01-thinking-tmux/dev-watch.png)
nges, lldb open in the bottom right](images/01-thinking-tmux/dev-watch.png)  |~
                                                                           |~
tmux gives you prime oppurtunity to do many things at once on the same screen.|~
You can keep multiple terminals running on the same display.               |~
                                                                           |~
(After all, that's where the name tmux comes from - **T**erminal           |~
**Mu**ltiple**x**er.)                                                      |~
                                                                           |~
In addition to having multiple terminals on one screen, tmux allows you to |~
create and link multiple "windows", all within the confines of the tmux sessio|~
n                                                                          |~
you attached.                                                              |~
                                                                           |~
Even better, there are facilities to copy and paste, scroll. No requirement |~
for graphics either, so you have full power even if you're SSH'ing or in a |~
tty without X.                                                             |~
                                                                           |~
So here are a few common scenarios:                                        |~
                                                                           |~
A system administrator will run a `tail -F /var/log/apache2/error.log` in a |~
pane to get a live stream of the latest system events.                     |~
                                                                           |~
<md  [unix/MARKDOWN] [/Users/me/work/tao-of-tmux/manuscript]48,0-1     31% <d [+] [unix/MARKDOWN] [/Users/me/work/tao-of-tmux/manuscript]14,107     93%

                                                                                          ~/work/tao-of-tmux/manuscript mast
~/work/tao-of-tmux/manuscript master          ~/work/tao-of-tmux/manuscript master        er
❯                                             ❯                                           ❯
mbp15  16.1.0                     1:manuscript* 2:zsh-          15:24 Mon10:54:32 AM 2016-12-05
```



```
1:manuscript* 2:zsh-
```

The first window, ID 1, titled "manuscript" is active. The second window, ID 2, titled zsh.

6.1 Creating a session

The simplest command to create a new session is typing `tmux`:

{language=shell, line-numbers=off} $ tmux

The `$ tmux` application, with no commands is equivalent to `$ tmux new-session`. Nifty!

By default, your session name will be given a number, which isn't too descriptive. What would be better is:

{language=shell, line-numbers=off} $ tmux new-session -s'my rails project'

6.2 Switching sessions within tmux

Some acquire the habit of detaching their tmux client and reattaching via `tmux att -t session_name`. Thankfully, you can switch sessions from within tmux!

`Prefix` + `s` will allow you to switch between sessions within the same tmux client.

This command name can be confusing. `switch-client` will allow you to traverse between sessions in the *server*.

Example usage:

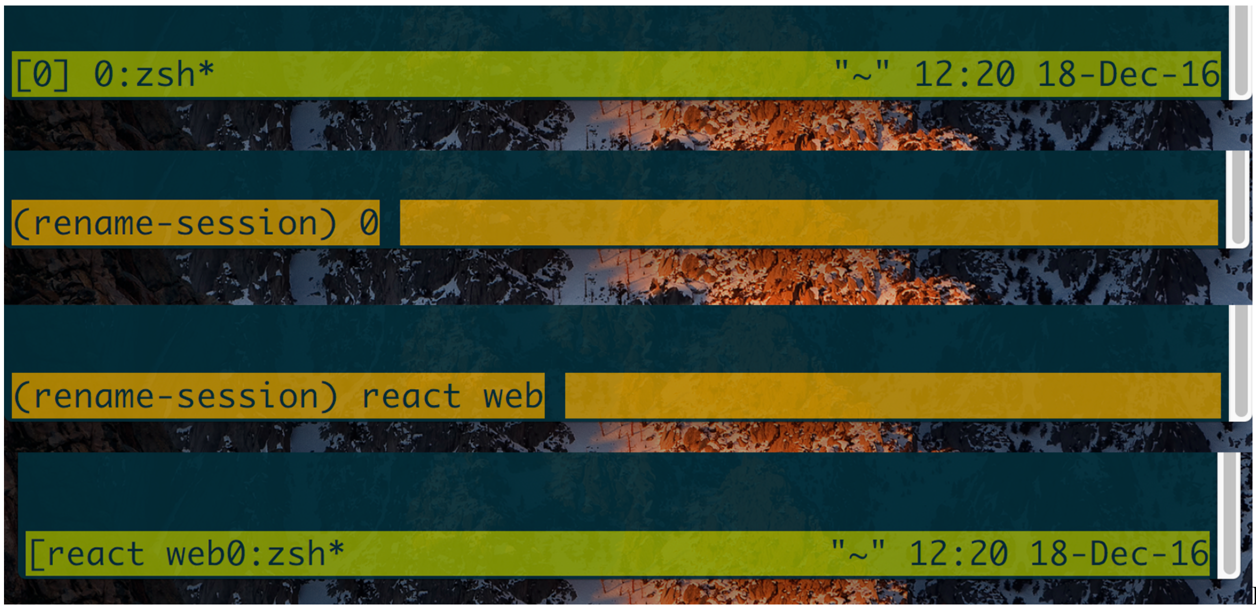
{language=shell, line-numbers=off} $ tmux switch-client -t dev

If already inside a client, this will switch to a session, named "dev", if it exists.

6.3 Naming sessions

Sometimes, the default session name given by tmux isn't descriptive enough. It only takes a few seconds to update it.

You can name it whatever you want. Typically, if I'm working on multiple web projects in one session, I'll name it "web". If I'm assigning one software project to a single session, I'll name it after the software project. You'll likely develop your own naming conventions, but anything is more descriptive than the default.



```
[0] 0:zsh*                                        "~" 12:20 18-Dec-16
(rename-session) 0
(rename-session) react web
[react web0:zsh*                                  "~" 12:20 18-Dec-16
```

Renaming a session '0' to 'react web'

If you don't name your sessions, it'll be difficult to keep track of what the session contains. Sometimes, you may forget you have a project opened, especially if your machine has been running for a few days, weeks, or months. You can save time by reattaching your session and avoid creating a duplicate.

You can rename sessions from within tmux with `Prefix` + . The status bar will be temporarily altered into a text field to allow altering the session name.

Through command line, you can try:

{language=shell, line-numbers=off} $ tmux rename-session -t 1 "my session"

6.4 Does my session exist?

If you're scripting tmux, you will want to see if a session exists. `has-session` will return a 0 exit code if the session exists, but will report a 1 exit code *and* print an error if a session does not exist.

{language=shell, line-numbers=off} $ tmux has-session -t 1

It assumes the session "1" exists; it'll just return 0 with no output.

But if it doesn't, you'll get something like this in a response:

{language=shell, line-numbers=off} $ tmux has-session -t 1 > can't find session 1

To try it in a shell script:

{language=shell, line-numbers=off} if tmux has-session -t 0 ; then echo "has session 0" fi

6.5 Summary

In this chapter, you learned how to rename sessions for organizational purposes and how to switch between them quickly.

You'll always be attached to a session when you're using a client in tmux. When the last remaining session is closed, the server will close also.

Think of sessions as workspaces designed to help organize a set of windows, analogous to virtual desktop spaces in GUI computing.

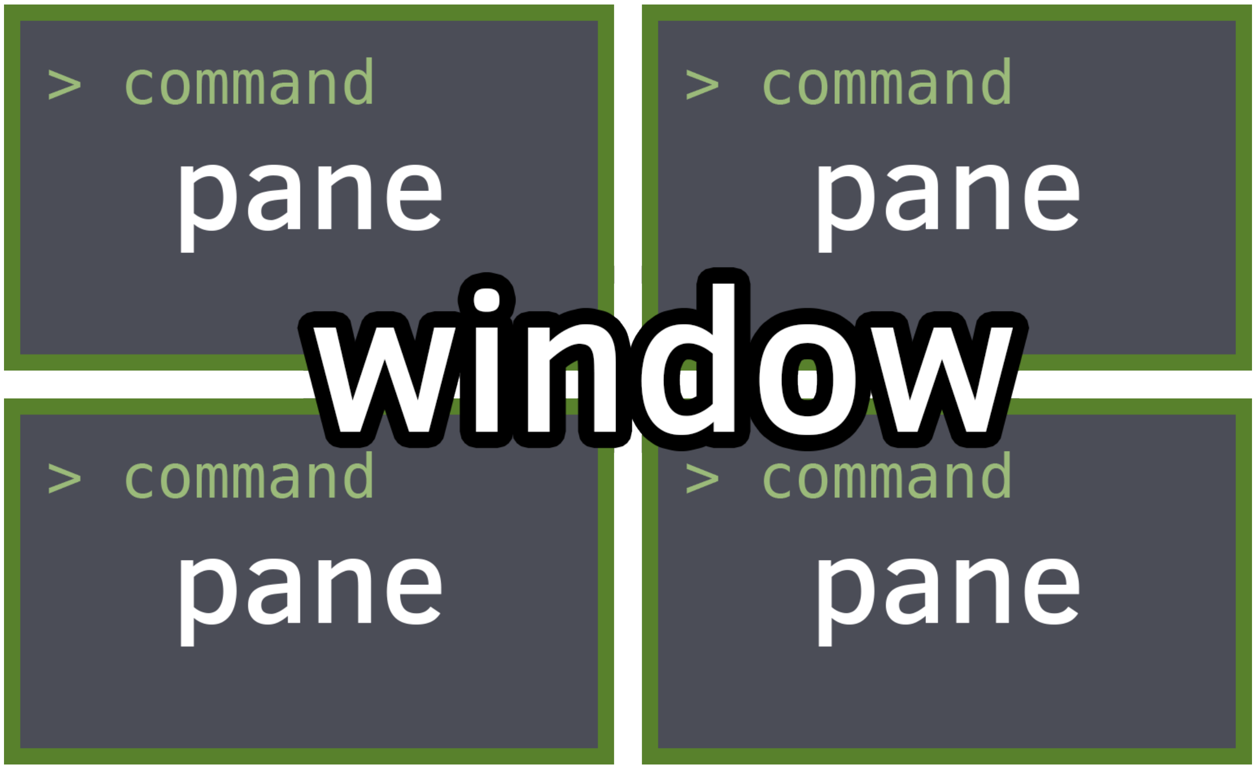
In the next chapter, we will go into *windows*, which, like sessions, are also nameable and let you switch between them.

CHAPTER 7

Windows {#windows}

Windows hold *panes*. They reside within a *session*.

They also have *layouts*, which can be one of many preset dimensions or a custom one done through *pane resizing*.



You can see the current windows through the *status bar* at the bottom of tmux.

7.1 Creating windows

All sessions start with at least one window open. From there, you can create and kill windows as you see fit.

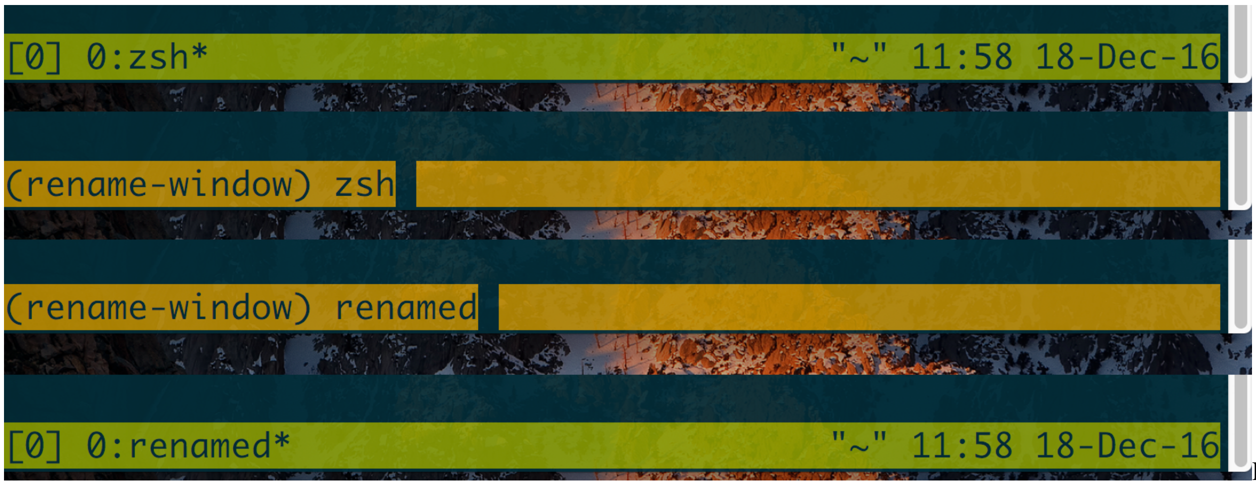
Window indexes are numbers tmux uses to determine ordering. The first window's index is 0, unless you set it via `base-index` in your *configuration*. I usually `set -g base-index 1` in my tmux configuration, since 0 is after 9 on the keyboard.

`Prefix` + `c` will create a new window at the first open index. So, if you're in the first window, and there is no second window created, it will create the second window. If the second window is already taken, and the third hasn't been created, it will create the third window.

If the `base_index` is 1 and there are 7 windows created, with the 5th window missing, creating a new window will fill the empty 5th index, since it's the next one in order and nothing is filling it. The next created window would be the eighth.

7.2 Naming windows

Just like with sessions, windows can have names. Labelling them helps keep track of what you're doing inside them.


```
[0] 0:zsh*                                   "~" 11:58 18-Dec-16

(rename-window) zsh

(rename-window) renamed

[0] 0:renamed*                               "~" 11:58 18-Dec-16
```
Renaming a window 'zsh' to 'renamed'

When inside tmux, the shortcut `Prefix` + `,` is most commonly used. It opens a prompt in the tmux status line, where you can alter the name of the current window.

The default numbers given to windows also become muscle memory after a while. But naming helps you when you're in a new tmux flow and want to organize yourself. Also, if you're sharing tmux with another user, it's good practice to give a hint what's inside the windows.

7.3 Traversing windows

Moving around windows is done in two ways, first, by iterating through via `Prefix` + `p` and `Prefix` + `n` and via the window index, which takes you directly to a specific window.

`Prefix` + `1`, `Prefix` + `2`, and so on… allows quickly navigating to windows via their index. Unlike window names, which change, indexes are consistent and only require a quick key combo for you to invoke.

Prompt for a window index (useful for indexes greater than 9) with `Prefix` + `'`. If the window index is 10 or above, this will help you a lot.

I> ### Tip: Search + Traverse Windows for Text I> I> You can forward to a window with a match of a text string by doing `Prefix` + I> `f`.

Bring up the last selected window with `Prefix` + `l`.

A list of current windows can be displayed with `Prefix` + `w`. This also gives some info on what's inside the window. Helpful when juggling a lot of things!

7.4 Moving windows

Windows can also be reordered one by one via `move-window` and its associated shortcut. This is helpful if a window is worth keeping open but not important or rarely looked at. After you move a window, you can continue to reorder them at any point in time after.

The command `$ tmux move-window` can be used to move windows.

The accepted arguments are `-s` (the window you are moving) and `-t`, where you are moving the window to.

You can also use `$ tmux movew` for short.

Example: move the current window to number 2:

{language=shell, line-numbers=off} $ tmux movew -t2

Example: move window 2 to window 1:

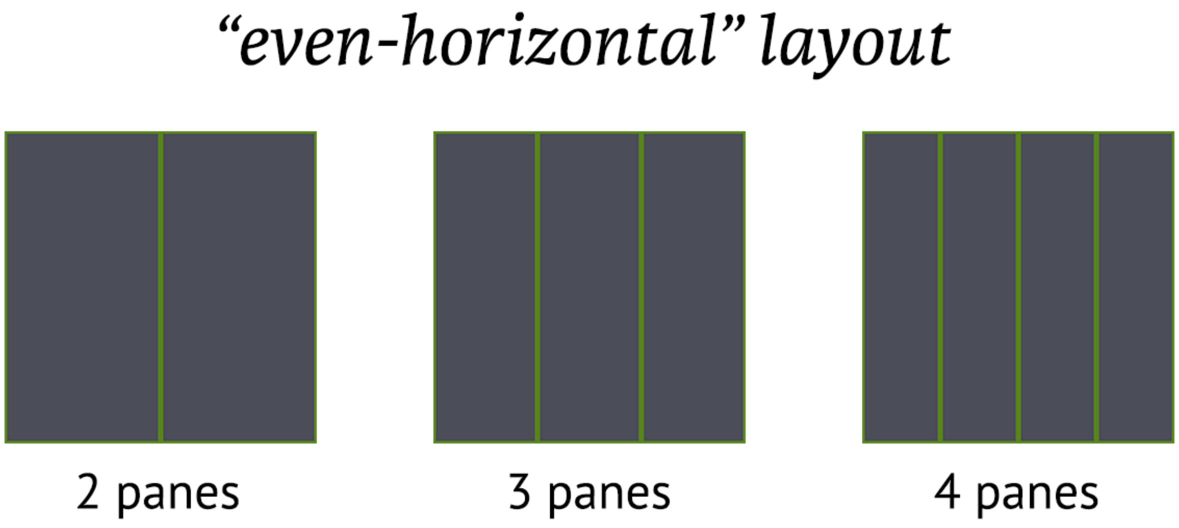
{language=shell, line-numbers=off} $ tmux movew -s2 -t1

The shortcut to prompt for an index to move the current window to is `Prefix` + `.`.

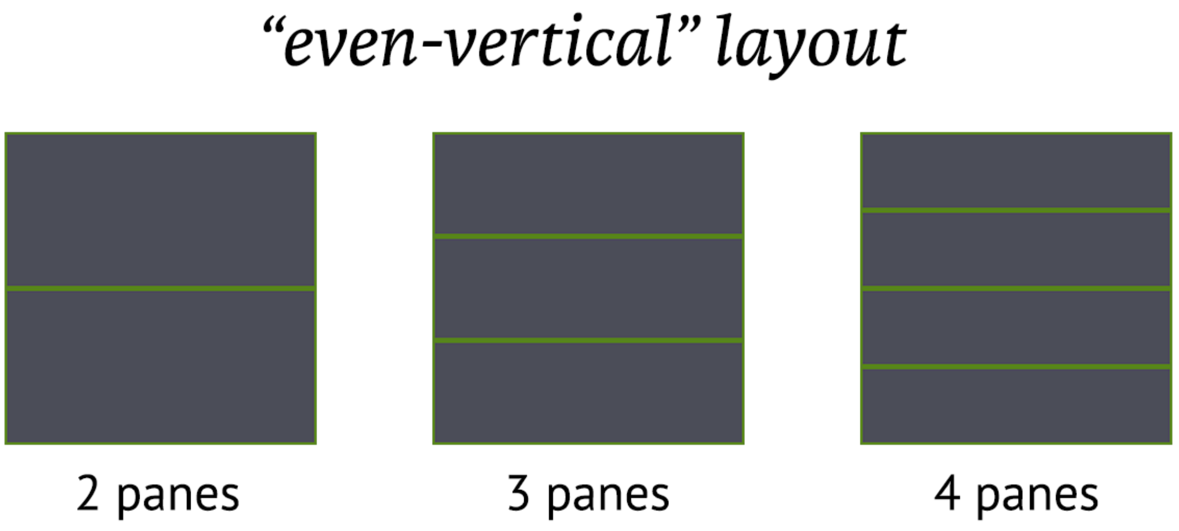
7.5 Layouts {#window-layouts}

`Prefix` + `space` switches window *layouts*. These are preset configurations automatically adjusting proportions of *panes*.

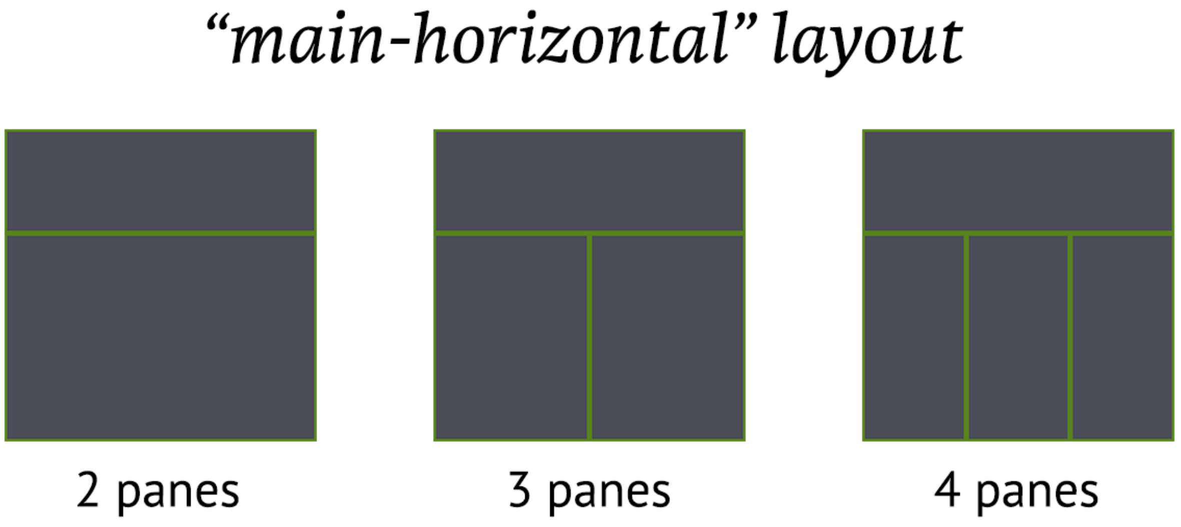
As of tmux 2.3, the supported layouts are:



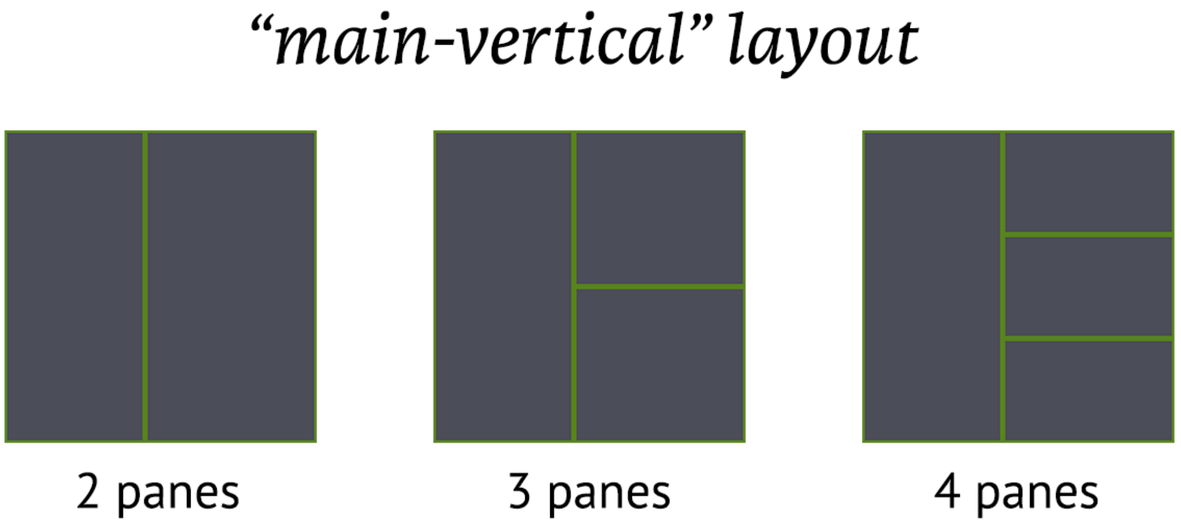
{width=75%}



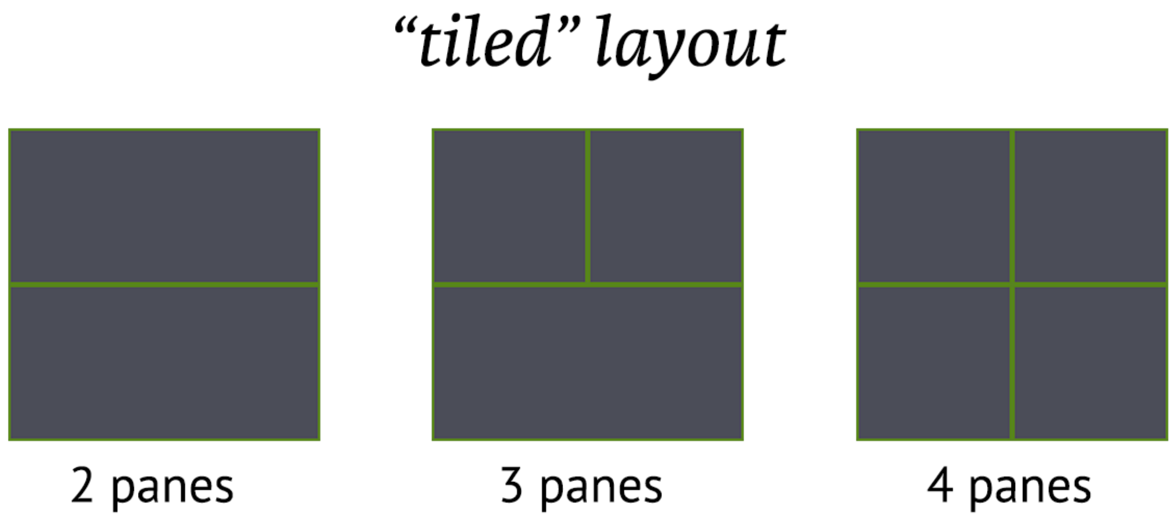
{width=75%}



{width=75%}



{width=75%}



{width=75%}

Specific touch-ups can be done via *resizing panes*.

To reset the proportions of the layout (such as after splitting or resizing panes), you have to run `$ tmux select-layout` again for the layout.

This is different behavior than some tiling window managers. *awesome* and *xmonad*, for instance, automatically handle proportions upon new items being added to their layouts.

To allow easy resetting to a sensible layout across machines and terminal dimensions, you can try this in your *config*:

{language=shell, line-numbers=off} bind m set-window-option main-pane-height 60; select-layout main-horizontal

This allows you to set a `main-horizontal` layout and automatically set the bottom panes proportionally on the bottom every time you do `Prefix` + `m`.

Layouts can also be custom. To get the custom layout snippet for your current window, try this:

{language=shell, line-numbers=off} $ tmux lsw -F "#{window_active} #{window_layout}" | grep "^1" | cut -d " " -f2

To apply this layout:

{language=shell, line-numbers=off} $ tmux lsw -F "#{window_active} #{window_layout}" | grep "^1" | cut -d " " -f2 > 5aed,176x79,0,0[176x59,0,0,0,176x19,0,60{87x19,0,60,1,88x19,88,60,2}]

```
# resize your panes or try doing this in another window to see the outcome
$ tmux select-layout "5aed,176x79,0,0[176x59,0,0,0,176x19,0,60{87x19,0,60,1,88x19,88,
↪60,2}]"
```

7.6 Closing windows

There are two ways to kill a window. First, exit or kill every pane in the window. Panes can be killed via `Prefix` + `x` or by `Ctrl` + `d` within the pane's shell. The second way, `Prefix` + `&`, prompts if you really want to delete the window. Warning: this will destroy all the window's panes, along with the processes within them.

From inside the current window, try this:

{language=shell, line-numbers=off} $ tmux kill-window

Another thing, when *scripting* or trying to kill the window from outside, use a *target* of the window index:

{language=shell, line-numbers=off} $ tmux kill-window -t2

If you're trying to find the target of the window to kill, they reside in the number in the middle section of the *status line* and via `$ tmux choose-window`. You can hit "return" after you're in choose-window to go back to where you were previously.

7.7 Summary

In this chapter, you learned how to manipulate windows via renaming and changing their layouts, a couple of ways to kill windows in a pinch or in when shell scripting tmux. In addition, this chapter demonstrated how to save any tmux layout by printing the `window_layout` template variable.

If you are in a tmux session, you'll always have at least one window open, and you'll be in it. And within the window will be "pane"; a shell within a shell. When a window closes all of its panes, the window closes too. In the next chapter, we'll go deeper into panes.

CHAPTER 8

Panes {#panes}

Panes are pseudoterminals encapsulating shells (e.g., Bash, Zsh). They reside within a *window*. A terminal within a terminal, they can run shell commands, scripts, and programs, like vim, emacs, top, htop, irssi, weechat, and so on within them.

8.1 Creating new panes

To create a new pane, you can `split-window` from within the current *window* and pane you are in.

You can continue to create panes until you've reached the limit of what the terminal can fit. This depends on the dimensions of your terminal. A normal window will usually have 1 to 5 panes open.

Example usage:

{language=shell, line-numbers=off} # Create pane horizontally, $HOME directory, 50% width of current pane $ tmux split-window -h -c $HOME -p 50 vim

{width=75%} ![](images/07-pane/splitw/-h -c $HOME -p 50 vim - 2 panes.png)

{language=shell, line-numbers=off} # create new pane, split vertically with 75% height tmux split-window -p 75

{width=75%} ![](images/07-pane/splitw/-p 75.png)

{pagebreak}

8.2 Traversing Panes {#pane-traversal}

I> *Moving around vimtuitively* I> I> If you like vim (hjkl) keybindings, add these to your *config*: I> I> {language=shell, line-numbers=off} I> # hjkl pane traversal I> bind h select-pane -L I> bind j select-pane -D I> bind k select-pane -U I> bind l select-pane -R

8.3 Zoom in {#zoom-pane}

To zoom in on a pane, navigate to it and do `Prefix` + `z`.

You can unzoom by pressing `Prefix` + `z` again.

In addition, you can unzoom and move to an adjacent pane at the same time using a *pane traversal* key.

Behind the scenes, the keybinding is a shortcut for `$ tmux resize-pane -Z`. So, if you ever wanted to script tmux to zoom/unzoom a pane or apply this functionality to a custom key binding, you can do that too, for instance:

{line-numbers=off} bind-key -T prefix y resize-pane -Z

This would have `Prefix` + `y` zoom and unzoom panes.

8.4 Resizing panes {#resizing-panes}

Pane size can be adjusted within *windows* via *window layouts* and `resize-pane`. Adjusting window layout switches the proportions and order of the panes. Resizing the panes targets a specific pane inside the window containing it, also shrinking or growing the size of the other columns or rows. It's like adjusting your car seat or reclining on a flight; if you take up more space, something else will have less space.

8.5 Outputting pane to a file

You can output the display of a pane to a file.

{language=shell, line-numbers=off} $ tmux pipe-pane -o 'cat >>~/output.#I-#P'

The `#I` and `#P` are *formats* for window index and pane index, so the file created is unique. Clever!

8.6 Summary

Panes are shells within a shell. You can keep adding panes to a tmux window until you run out of room on your screen. Within your shell, you can `tail -F` log files, write and run scripts, and run curses-powered applications, like vim, top, htop, ncmpcpp, irssi, weechat, mutt, and so on.

You will always have at least one pane open. Once you kill the last pane in the window, the window will close. Panes are also resizable; you can resize panes by targeting them specifically and changing the window layout.

In the next chapter, we will go into the ways you can customize your tmux shortcuts, status line, and behavior.

CHAPTER 9

Configuration {#config}

Most tmux users break away from the defaults by creating their own customized configurations. These configurations vary from the trivial, such as adding keybindings, and adjusting the prefix key, to complex things, such as decking out the *status bar* with system stats and fancy glyphs via powerlines.

Configuration of tmux is managed through `.tmux.conf` in your `$HOME` directory. The paths `~/.tmux.conf` and `$HOME/.tmux.conf` should work on OS X, Linux, and BSD.

Configuration is applied upon initially starting tmux. The contents of the configuration are tmux commands. The file can be reloaded later via `source-file`, which is discussed in this chapter.

For a sample config, I maintain a pretty decked out one at https://github.com/tony/tmux-config. It's permissively licensed, and you're free to copy and paste from it as you wish.

I> **Custom Configs** I> I> You can specify your config via the `-f` command. Like this: I> I> {language=shell, line-numbers=off} I> $ tmux -f path/to/config.conf I> I> Note: If a tmux server is running in the background and you want I> to test a fresh config, you must either shut down the rest of the I> tmux sessions or use a different socket name. Like this: I> I> {language=shell, line-numbers=off} I> $ tmux -f path/to/config.conf -Ltesting_tmux I> I> And you can treat everything like normal; just keep passing `-Ltesting_tmux` I> (or whatever socket name you feel like testing configs with) for reuse. I> I> {language=shell, line-numbers=off} I> $ tmux -Ltesting_tmux attach

9.1 Reloading configuration {#reload-config}

You can apply config files in live tmux sessions. Compare this to `source` or "dot" in the POSIX standard.

`Prefix` + `:` will open the tmux prompt, then type:

`:source /path/to/config.conf`

And hit return.

`$ tmux source-file /path/to/config.conf` can also achieve the same result via command line.

I> **Easy reloadin'** I> I> Even better, often, you will keep your default tmux config stored in I> `$HOME/.tmux.conf`. So, what can you do? You can `bind-key` to I> `source-file ~/.tmux.conf`: I> I> `bind r source ~/.tmux.conf` I> I> You can also have it give you a confirmation afterwards: I> I> `bind r source ~/.tmux.`

`conf\; display "~/.tmux.conf sourced!"` I> I> Now, you can type `Prefix` + `r` to get the config to reload.

Note that reloading the configuration only *re-runs* the configuration file. It will not reset keybindings or styling you apply to tmux.

9.2 How configs work

The tmux configuration is processed just like run commands in a `~/.zshrc` or `~/.bashrc` file. `bind r source ~/.tmux.conf` in the tmux configuration is the same as `$ tmux bind r source ~/.tmux.conf`.

You could always create a shell script prefixing `tmux` in front of commands and run it on fresh servers. The result is the same. Same goes if you manually type in `$ tmux set-option` and `$ tmux bind-key` commands into any terminal (in or outside tmux).

This in `.tmux.conf`:

{language=shell, line-numbers=off} bind-key a send-prefix

Is the same as having no `.tmux.conf` (or `$ tmux -f/dev/null`) and typing:

{language=shell, line-numbers=off} $ tmux bind-key a send-prefix

in a newly started tmux server.

The important thing to internalize is that a tmux configuration consists of setting server options (`set-option -s`), global session (`set-option -g`), and window options (`set-window-option -g`).

The rest of this chapter is going to proceed cookbook-style. You can pick out these tweaks and add them to your `.tmux.conf` and *reload*.

9.3 Server options

Server options are set with `set-option -s option value`.

9.3.1 Tweak timing between key sequences

{line-numbers=off} set -s escape-time 0

9.3.2 Terminal coloring

If you're having an issue with color detail in tmux, it may help to set `default-terminal` to `screen-256color`.

{line-numbers=off} set -g default-terminal "screen-256color"

This sets the `TERM` variable in new panes.

9.4 Session options

Aside from the *status bar*, covered in the next chapter, most user configuration will be custom keybindings. This section covers the few generic options, and the next section goes into snippets involving keybindings.

9.4.1 Base index

This was mentioned earlier in the book, but it's a favorite tweak of many tmux users, who find it more intuitive to start their window counting at *1*, rather than the default, *0*. To set the starting number (base index) for windows:

{line-numbers=off} set -g base-index 1

Setting `base-index` assures newly created windows start at 1 and count upwards.

9.5 Window options

Window options are set via `set-option -w` or `set-window-option`. They are the same thing.

9.5.1 Automatic window naming

Setting `automatic-rename` alters the name of the window based upon its active pane:

{line-numbers=off} set-window-option -g automatic-rename

Automatic renaming will be disabled for the window if you rename it manually.

9.6 Keybindings

9.6.1 Prefix key

The default *prefix key* in tmux is `<Ctrl-b>`. You can customize it by setting a new prefix and unsetting the default. To set the prefix to `<Ctrl-a>`, like GNU Screen, try this:

{line-numbers=off} set-option -g prefix C-a unbind-key C-b bind-key a send-prefix

9.6.2 New window with prompt

Prompt for window name upon creating a new window, `Prefix` + `C` (capital C):

{line-numbers=off} bind-key C command-prompt -p "Name of new window: " "new-window -n '%%'"

9.6.3 Vi copy-paste keys

This is comprised of two-parts: Setting the `mode-keys` window option to vi and setting the `vi-copy` bindings to use `v` to begin selection and `y` to yank.

{line-numbers=off} # Vi copypaste mode set-window-option -g mode-keys vi bind-key -t vi-copy 'v' begin-selection bind-key -t vi-copy 'y' copy-selection

9.6.4 hjkl / vi-like pane traversal

Another one for vi fans, this keeps your right hand on the home row when moving directionally across panes in a window.

{line-numbers=off} bind h select-pane -L bind j select-pane -D bind k select-pane -U bind l select-pane -R

9.6.5 Further inspiration

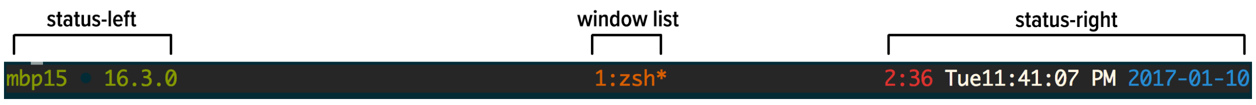
For more ideas, I have a `.tmux.conf` you can copy-paste from on the internet at https://github.com/tony/tmux-config/blob/master/.tmux.conf.

In the next chapter, we will go into configuring the *status line*.

CHAPTER 10

Status bar and styling {#status-bar}

The status bar, or *status line*, serves as a customizable taskbar in the bottom of tmux. It is comprised of 3 sections. The status fields on either side of the status line are customizable. The center field is a list of windows.



```
mbp15 ● 16.3.0                    1:zsh*                  2:36 Tue11:41:07 PM 2017-01-10
```

The `status-left` and `status-right` option can be configured with variables.

It's *configurable* through the `.tmux.conf` file and modifiable live through using `$ tmux set-option`.

I> *Finding your current status line settings* I> I> {language=shell, line-numbers=off} I> $ tmux show-options -g | grep status

10.1 Window status symbols

This window list is between the left and right status bar regions.

tmux indicates status of a window through symbols. See below:

Reminder: A pane can be *zoomed* via `Prefix` + `z`. To unzoom, press `Prefix` + `z` or move left / right / up / down panes.

10.2 Date and time

`status-left` and `status-right` accept variables for the date.

This happens via piping the status templates through `format_expand_time` in `format.c`, which routes right into `strftime(3)` from `time.h`.

A full list of variables can be found in the documentation for `strftime(3)`. This can be viewed through `$ man strftime` on Unix-like systems.

10.3 Shell command output

You can also call applications, such as tmux-mem-cpu-load, conky, and *powerline*.

For this example, we'll use `tmux-mem-cpu-load`. This works on Unix-like systems like FreeBSD, Linux distributions, and macOS.

To build from source, you must have CMake and `git`, which are available through your package manager. You must have a C++ compiler. On macOS, install Xcode CLI Utilities. You can do this by going to *Applications* -> *Utilities*, launching *Terminal.app* and typing `$ xcode-select --install`. macOS can use Homebrew to install the CMake and git package. Major Linux distributions package CMake, clang, and git.

Before this step, you can `cd` into any directory you're ok keeping code in.

{language=shell, line-numbers=off} $ git clone https://github.com/thewtex/tmux-mem-cpu-load.git $ cd tmux-mem-cpu-load $ mkdir ./build $ cd ./build $ cmake .. $ make

```
# macOS, no sudo required
$ make install

# Linux, BSD will require sudo / root to install
$ sudo make install
```

If successful, you should see the output below:

{language=shell, line-numbers=off} [100%] Built target tmux-mem-cpu-load Install the project… – Install configuration: "MinSizeRel" – Installing: /usr/local/bin/tmux-mem-cpu-load

You can remove the source code you cloned from the computer. The compiled application is installed.

You can now add `#(tmux-mem-cpu-load)` to your `status-left` or `status-right` option. In the "*Dressed up*" example below, I use `status-left` and also theme it to be green:

`#[fg=green,bg=default,bright]#(tmux-mem-cpu-load)`

So to apply it to your theme, you need to double check what you already have. You may have information on there you want to keep.

{language=shell, line-numbers=off} $ tmux show-option -g status-right status-right " "#{=21:pane_title}" %H:%M %d-%b-%y"

Copy what you had in response (or change, rearrange as you see fit) then add the `#(tmux-mem-cpu-load)` to it. You can apply the new status line in your current tmux session via `$ tmux set-option -g status-right:`

{language=shell, line-numbers=off} $ tmux set-option -g status-right '"#{=21:pane_title}" #(tmux-mem-cpu-load) %H:%M %d-%b-%y'

Also, note how I switched out the double quotes on either side of the option with single quotes. This is required, since there are double quotes inside.

You can do this with anything, for instance, try adding `uptime`. This could be done by adding `#(uptime)` to your status line. Typically the output is pretty long, so trim it down by doing something like this:

'#(uptime | cut -f 4-5 -d " " | cut -f 1 -d ",")''

In the next section, we go into how you can style (color) tmux.

10.4 Styling

The *colors* available to tmux are:

- `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, `white`.
- bright colors, such as `brightred`, `brightgreen`, `brightyellow`, `brightblue`, `brightmagenta`, `brightcyan`.
- `colour0` through `colour255` from the 256-color set.
- `default`
- hexadecimal RGB code like `#000000`, `#FFFFFF`, similar to HTML colors.

10.4.1 Status line

You can use `[bg=color]` and `[fg=color]` to adjust the text color and background within for status line text. This works on `status-left` and `status-right`.

Let's say you want to style the background:

Command: `$ tmux set-option status-style fg=white,bg=black`

In config: `status-style fg=white,bg=black`

In the examples at the end of the chapter, you will see complete examples of how colors can be used.

10.4.2 Clock styling

You can style the color of the tmux clock via:

{lang="text", line-numbers=off} set-option -g clock-mode-colour white

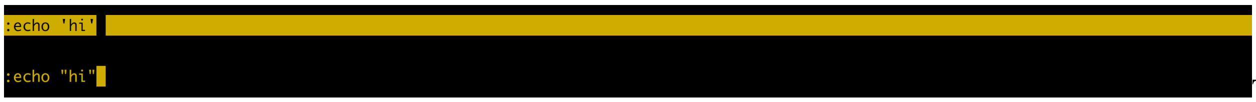
Reminder: Clock mode can be opened with `$ tmux clock-mode` or `Prefix` + `t`. Pressing any key will exit clock mode.

10.4.3 Prompt colors

The benefit of wrapping your brain around this styling is you will see it `message-command-style`, `message style` and so on.

Let's try this:

{lang="shell", line-numbers=off} $ tmux set-option -ag message-style fg=yellow,blink; set-option -ag message-style bg=black



```
:echo 'hi'

:echo "hi"
```

Top: default scheme for prompt. Bottom: newly-styled.

10.5 Styling while using tmux

So, you want to customize your tmux status line before you write the changes to your *config* file.

Start by grabbing your current status line section you want to edit, for instance:

{lang="text", line-numbers=off} $ tmux show-options -g status-left > status-left "[#S] " $ tmux show-options -g status-right > status-right " "#{=21:pane_title}" %H:%M %d-%b-%y"

Also, you can try to snip off the variable with `| cut -d' ' -f2-`:

{lang="text", line-numbers=off} $ tmux show-options -g status-left | cut -d' ' -f2- > "[#S] " $ tmux show-options -g status-right | cut -d' ' -f2- > " "#{=21:pane_title}" %H:%M %d-%b-%y"

Then, add the options to your *configuration*.

To be sure your configuration fully works, you can start it in a different server via `tmux -Lrandom`, verify the settings, and close it. This is helpful to make sure your config file isn't missing any styling info.

10.6 Toggling status line

The tmux status line can be hidden, as well. Turn it off:

{language=shell, line-numbers=off} $ tmux set-option status off

And, turn it on:

{language=shell, line-numbers=off} $ tmux set-option status on

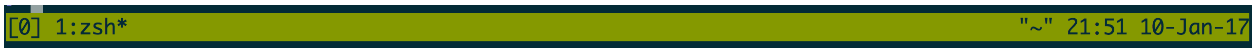
The above is best for scripting, but if you're binding it to a keyboard shortcut, *toggling*, or reversing the current option, it can be done via omitting the on/off value:

{language=shell, line-numbers=off} $ tmux set-option status

Bind toggling status line to `Prefix` + `q`:

{language=shell, line-numbers=off} $ tmux bind-key q set-option status

10.7 Example: Default config



```
[0] 1:zsh*                                                     "~" 21:51 10-Jan-17
```

This is an example of the default config you see if your tmux configuration has no status styling.

{line-numbers=off} status on status-interval 15 status-justify left status-keys vi status-left "[#S] " status-left-length 10 status-left-style default status-position bottom status-right " "#{=21:pane_title}" %H:%M %d-%b-%y" status-right-length 40 status-right-style default status-style fg=black,bg=green

10.8 Example: Dressed up {#status-bar-example-dressed-up}

![](images/09-status-bar/dressed up.png)

{line-numbers=off} status on status-interval 1 status-justify centre status-keys vi status-left "#[fg=green]#H #[fg=black]• #[fg=green,bright]#(uname -r | cut -c 1-6)#[default]" status-left-length 20 status-left-style default status-position bottom status-right "#[fg=green,bg=default,bright]#(tmux-mem-cpu-load) #[fg=red,dim,bg=default]#(uptime | cut -f 4-5 -d " " | cut -f 1 -d ",") #[fg=white,bg=default]%a%l:%M:%S %p#[default] #[fg=blue]%Y-%m-%d" status-right-length 140 status-right-style default status-style fg=colour136,bg=colour235

```
# default window title colors
set-window-option -g window-status-fg colour244  # base0
set-window-option -g window-status-bg default

# active window title colors
set-window-option -g window-status-current-fg colour166  # orange
set-window-option -g window-status-current-bg default
```

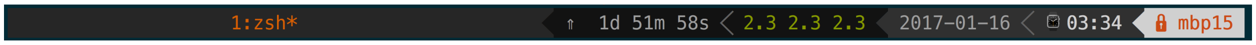
Configs can print the output of an application. In this example, tmux-mem-cpu-load is providing system statistics in the right-side section of the status line.

To build tmux-mem-cpu-load, you have to install CMake and have a C++ compiler, like clang or GCC.

On Ubuntu, Debian, and Mint machines, you can do this via `$ sudo apt-get install cmake build-essential`. On macOS w/ brew via `$ brew install cmake`.

Source: https://github.com/tony/tmux-config

10.9 Example: Powerline



```
1:zsh*        ⇑ 1d 51m 58s  2.3 2.3 2.3  2017-01-16  03:34  mbp15
```

The most full-featured solution available for tmux status lines is powerline, which heavily utilizes the shell command outputs, not only to give direct system statistics, but also to generate graphical-like styling.

To get the styling to work correctly, special fonts must be installed. The easiest way to use this is to install powerline fonts, a collection of fixed width coder fonts patched to support Wingdings-like symbols.

Installation instructions are on Read the Docs. For a better idea:

{language=shell, line-numbers=off} $ pip install –user powerline-status psutil

psutil, a required dependency of powerline, is a cross-platform tool to gather system information.

Assure you *properly configured python with your PATHs*, and try this:

{line-numbers=off} set -g status-interval 2 set -g status-right '#(powerline tmux right)'

10.10 Summary

Configuring the status line is optional. It can use the output of programs installed on your system to give you specialized information, such as CPU, ram, and I/O usage. By default, you'll at least have a window list and a clock.

In addition, you can customize the colors of the status line, clock, and prompt. By default, it's only a green bar with dark text, so take some time to customize yours, if you want, and save it to your *configuration*.

In the next chapter, we will go into the command line and scripting features of tmux.

CHAPTER 11

Scripting tmux {#scripting-tmux}

The command line shortcuts and options in tmux is an area often uncharted.

I will use tables in this chapter. Never get a feeling you have to commit a table to memory immediately. Not my intention, but every person's way of using tmux is slightly different. I want to cover points most likely to benefit people's flows. Full tables are in the *cheatsheets*.

11.1 Aliases {#aliases}

tmux supports a variety of alias commands. With aliases, instead of typing `$ tmux attach-session` to attach a session, `$ tmux attach` could do the trick.

Most aliases come to mind via intuition and are a lot friendlier than typing the full hyphenated commands.

{width="narrow"}

Command	Alias
attach-session	attach
break-pane	breakp
capture-pane	capturep
display-panes	displayp
find-window	findw
join-pane	joinp
kill-pane	killp
kill-window	killw
last-pane	lastp
last-window	last
link-window	linkw
list-panes	lsp
list-windows	lsw
move-pane	movep
move-window	movew
new-session	new
new-window	neww
next-layout	nextl
next-window	next
pipe-pane	pipep
previous-layout	prevl
previous-window	prev
rename-window	renamew
resize-pane	resizep
respawn-pane	respawnp
respawn-window	respawnw
rotate-window	rotatew
select-layout	selectl
select-pane	selectp
set-option	set
set-window-option	setw
show-options	show
show-window-options	showw
split-window	splitw
swap-pane	swapp
swap-window	swapw
unlink-window	unlinkw

If you know the full name of the command, if you were to chop the hyphen (-) from the command and add the first letter of the last word, you'd get the shortcut, e.g., **swap-w**indow is swapw, **split-w**indow is splitw.

11.2 Pattern matching {#fnmatch}

In addition to aliases, tmux commands and arguments may all be accessed via `fnmatch(3)` patterns.

For instance, you need not type `$ tmux attach-session` every time. First, there's the *alias* of `$ tmux attach`, but additionally, more concise commands can be used if they partially match the name of the command or the target. tmux's pattern matching allows `$ tmux attac`, `$ tmux att`, `$ tmux at` and `$ tmux a` to reach `$ tmux attach`.

Every tmux command has shorthands; let's try this for `$ tmux new-session`:

{language=shell, line-numbers=off} $ tmux new-session

```
$ tmux new-sessio

# ...

$ tmux new-s
```

and so on, until:

{language=shell, line-numbers=off} $ tmux new- ambiguous command: new-, could be: new-session, new-window

The limitation, as seen above, is command matches can collide. Multiple commands begin with `new-`. So, if you wanted to use matches, `$ tmux new-s` for a new session or `$ tmux new-w` for a new window would be the most efficient way. But, the alias of `$ tmux new` for new session and `$ tmux neww` for new windows is even more concise than matching, since the special alias exists.

Patterns can also match *targets* with window and session names. For instance, a session named `mysession` can be matched via `mys`:

{language=shell, line-numbers=off} $ tmux attach -t mys

Matching targets will fail if a pattern matches more than one item. If 2 sessions exist, named `mysession` and `mysession2`, the above command would fail. To target either session, the complete target name must be specified.

11.3 Targets {#targets}

If a command allows target specification, it's usually done through `-t`.

Think of targets as tmux's way of specifying a unique key in a relational database.

What I use to help me remember:

So, sessions are represented by dollar signs ($) because they hold your projects (*ostensibly* where you make money or help someone else do it).

Windows are represented by the at sign (@). So, windows are like referencing / messaging a user on a social networking website.

Panes are the fun one, represented by the percent sign (%), like the default prompt for csh and tcsh. Hey, makes sense, since panes are pseudoterminals!

When scripting tmux, the symbols help denote the type of object, but also serve as a way to target something deeply, such as the pane, *directly*, without needing to know or specify its window or session.

Here are some examples of targets, assuming one session named `mysession` and a client at `/dev/ttys004`:

11.3.1 `attach-session [-t target-session]`

{language=shell, line-numbers=off} $ tmux attach-session -t mysession

11.3.2 `detach-client [-s target-session] [-t target-client]`

{language=shell, line-numbers=off} $ tmux detach-client -s mysession -t /dev/ttys004

```
# If within client, -t is assumed to be current client
$ tmux detach-client -s mysession
```

11.3.3 `has-session [-t target-session]`

{language=shell, line-numbers=off} $ tmux has-session -t mysession

```
# Pattern matching session name
$ tmux has-session -t mys
```

11.3.4 `$ tmux kill-session [-t target-session]`

{language=shell, line-numbers=off} $ tmux kill-session -t mysession

11.3.5 `$ tmux list-clients [-t target-session]`

{language=shell, line-numbers=off} $ tmux list-clients -t mysession

11.3.6 `$ tmux lock-client [-t target-client]`

{language=shell, line-numbers=off} $ tmux lock-clients -t /dev/ttys004

11.3.7 `$ tmux lock-session [-t target-session]`

{language=shell, line-numbers=off} $ tmux lock-session -t mysession

11.3.8 `$ tmux new-session [-t target-session]`

{language=shell, line-numbers=off} $ tmux new-session -t newsession

```
# Create new-session in the background
$ tmux new-session -t newsession -d
```

11.3.9 `$ tmux refresh-client [-t target-client]`

{language=shell, line-numbers=off} $ tmux refresh-client -t /dev/ttys004

11.3.10 `$ tmux rename-session [-t target-session]` session-name

{language=shell, line-numbers=off} $ tmux rename-session -t mysession renamedsession

```
# If within attached session, -t is assumed
$ tmux rename-session renamedsession
```

11.3.11 `$ tmux show-messages [-t target-client]`

{language=shell, line-numbers=off} $ tmux show-messages -t /dev/ttys004

11.3.12 `$ tmux suspend-client [-t target-client]`

{language=shell, line-numbers=off} $ tmux suspend-client -t /dev/ttys004

```
# If already in client
$ tmux suspend-client

# Bring client back to the foreground
$ fg
```

11.3.13 `$ tmux switch-client [-c target-client] [-t target-session]`

{language=shell, line-numbers=off} $ tmux suspend-client -c /dev/ttys004 -t othersession

```
# Within current client, -c is assumed
$ tmux suspend-client -t othersession
```

11.4 Formats {#formats}

tmux provides a minimal template language and set of variables to access information about your tmux environment.

Formats are specified via the `-F` flag.

You know how template engines, such as mustache, handlebars ERB in ruby, jinja2 in python, twig in PHP, and JSP in Java, allow template variables? Formats are a similar concept.

The `FORMATS` (variables) provided by tmux have expanded greatly since version 1.8. Some of the most commonly used formats as of tmux 2.3 are listed below. See the *appendix section on formats* for a complete list.

Let's try to output it:

{language=shell, line-numbers=off} $ tmux list-windows -F "#{window_id} #{window_name}" > @0 zsh

Here's a cool trick to list all panes with the x and y coordinates of the cursor position:

{language=shell, line-numbers=off} $ tmux list-panes -F "#{pane_id} #{pane_current_command} #{pane_current_path} #{cursor_x},#{cursor_y}" > %0 vim /Users/me/work/tao-of-tmux/manuscript 0,34 %1 tmux /Users/me/work/tao-of-tmux/manuscript 0,17 %2 man /Users/me/work/tao-of-tmux/manuscript 0,0

Variables are specific to the objects being listed. For instance:

Server-wide variables: `host`, `host_short` (no domain name), `socket_path`, `start_time` and `pid`.

Session-wide variables: `session_attached`, `session_activity`, `session_created`, `session_height`, `session_id`, `session_name`, `session_width`, `session_windows` and all server-wide variables.

Window variables: `window_activity`, `window_active`, `window_height`, `window_id`, `window_index`, `window_layout`, `window_name`, `window_panes`, `window_width` and all session and server variables.

Pane variables: `cursor_x`, `cursor_y`, `pane_active`, `pane_current_command`, `pane_current_path`, `pane_height`, `pane_id`, `pane_index`, `pane_width`, `pane_pid` and all window, session and server variables.

This book focuses on separating the concept of server, sessions, windows, and panes. With the knowledge of targets and formats, this separation takes shape in tmux's internal attributes. If you `list-panes` all variables up the ladder, including window, session and server variables are available for the panes being listed. Try this:

{language=shell, line-numbers=off} $ tmux list-panes -F "pane: #{pane_id}, window: #{window_id}, session: #{session_id}, server: #{socket_path}" > pane: %35, window: @13, session: $6, server: /private/tmp/tmux-501/default pane: %38, window: @13, session: $6, server: /private/tmp/tmux-501/default pane: %36, window: @13, session: $6, server: /private/tmp/tmux-501/default

Listing windows isn't designed to display variables for pane-specific properties. Since a window is a collection of panes, it can have 1 or more panes open at any time.

{language=shell, line-numbers=off} $ tmux list-windows -F "window: #{window_id}, panes: #{window_panes} pane_id: #{pane_id}" > window: @15, panes: 1 pane_id: %40 window: @13, panes: 3 pane_id: %36 window: @25, panes: 1 pane_id: %50

This will show the window ID, prefixed by an `@` symbol, and the number of panes inside the window.

Surprisingly, `pane_id` shows up via `list-windows`, as of tmux 2.3. While this output occurs in this version of tmux, it's undefined behavior. It's advised to keep use of `-F` scoped to the objects being listing when scripting to avoid breakage. For instance, if you want the active pane, use `#{pane_active}` via `$ tmux list-panes -F "#{pane_active}"`.

By default, `list-panes` will only show panes in a window, unless you specify `-a` to output all on a server or `-s [-t session-name]` for all panes in a session:

{language=shell, line-numbers=off} $ tmux list-panes -s -t mysession > 1.0: [176x29] [history 87/2000, 21033 bytes] %0 1.1: [87x6] [history 1814/2000, 408479 bytes] %1 (active) 1.2: [88x6] [history 1916/2000, 464932 bytes] %2 2.0: [176x24] [history 9/2000, 2262 bytes] %13 2.1: [55x11] [history 55/2000, 7395 bytes] %14

And the `-t` flag lists all panes in a window:

{language=shell, line-numbers=off} $ tmux list-panes -t @0 > 0: [176x29] [history 87/2000, 21033 bytes] %0 1: [176x36] [history 1790/2000, 407807 bytes] %1 (active) 2: [88x6] [history 1916/2000, 464932 bytes] %2

The same concept applies to `list-windows`. By default, The `-a` flag will list all windows on a server, `-t` lists windows within a session, and omitting `-t` will only list windows within the current session inside tmux.

{language=shell, line-numbers=off} $ tmux list-windows > 1: zsh* (3 panes) [176x36] [layout f9a4,176x36,0,0[176x29,0,0,0,176x6,0,30{87x6,0,30,1,88x6,88,30,2}]] @0 (active) 2: zsh- (5 panes) [176x36] [layout 55ef,176x36,0,0[176x24,0,0,13,176x11,0,25{55x11,0,25,14,58x11,56,25[58x7,56,25,16,58x3,56,33,17],61x11,115,25,15}]] @6

11.5 Controlling tmux {#send-keys}

tmux allows sending keys, including Ctrl via `C-` or `^`, alt (Meta) via `M-`, and special key names. Here's a list of special keys straight from the manual:

`Up`, `Down`, `Left`, `Right`, `BSpace`, `BTab`, `DC` (Delete), `End`, `Enter`, `Escape`, `F1` to `F12`, `Home`, `IC` (Insert), `NPage/PageDown/PgDn`, `PPage/PageUp/PgUp`, `Space`, and `Tab`.

If special keys are not matched, the defined behavior is to send it as a string to the pane, character by character.

For this example, we will use `send-keys` through tmux prompt, because omitting target (`-t`) will direct the command to the current pane, but the keys sent will sometimes print before the prompt.

Open tmux command prompt via `Prefix` + `:` and type this after the `:`:

`send-keys echo 'hi'`

Hit enter. This inserted *hi* into the current active pane. You can also use targets to specify which pane to send it to.

Let's now try to send keys to another pane in our current window. Create a second pane via splitting the window if one doesn't exist. You can also do this exercise outside of tmux or inside a scripting file and running it.

Grab a pane ID from the output of `list-panes`:

{language=shell, line-numbers=off} $ tmux list-panes > 0: [180x57] [history 87/2000, 21033 bytes] %0 1: [89x14] [history 1884/2000, 509864 bytes] %1 (active) 2: [90x14] [history 1853/2000, 465297 bytes] %2

`%2` looks good. Replace `%2` with the pane you want to target. This sends `cal` to the input:

{language=shell, line-numbers=off} $ tmux send-keys -t %2 'cal'

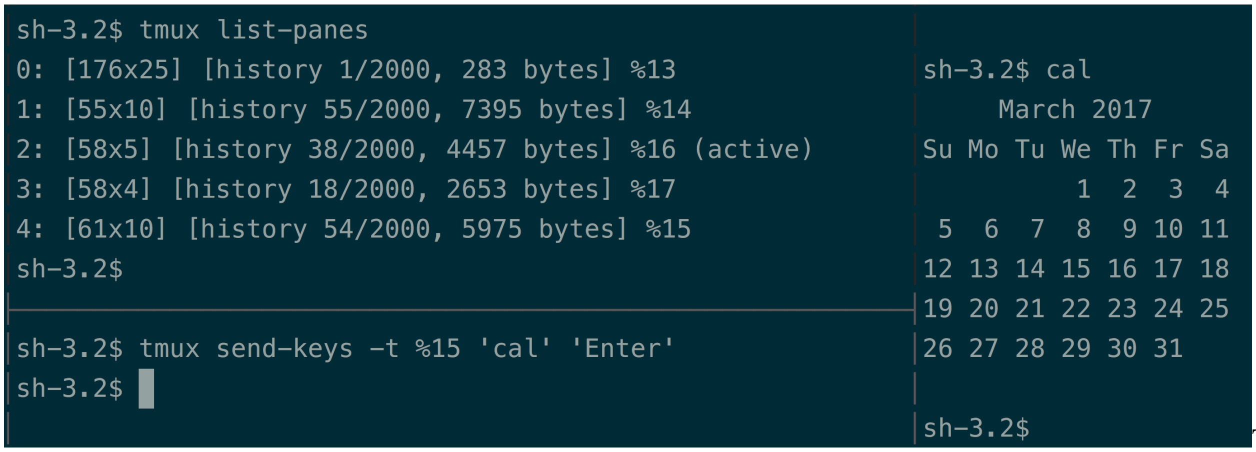
Nice, let's cancel that out by sending a `SIGINT`:

{language=shell, line-numbers=off} $ tmux send-keys -t %2 'C-c'

This cancelled the command and brought up a fresh input. This time, let's send an Enter keypress to run `cal(1)`.

{language=shell, line-numbers=off} $ tmux send-keys -t %2 'cal' 'Enter'

This outputs in the adjacent pane.



```
sh-3.2$ tmux list-panes
0: [176x25] [history 1/2000, 283 bytes] %13
1: [55x10] [history 55/2000, 7395 bytes] %14
2: [58x5] [history 38/2000, 4457 bytes] %16 (active)
3: [58x4] [history 18/2000, 2653 bytes] %17
4: [61x10] [history 54/2000, 5975 bytes] %15
sh-3.2$

sh-3.2$ tmux send-keys -t %15 'cal' 'Enter'
sh-3.2$
```

```
sh-3.2$ cal
     March 2017
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31

sh-3.2$
```

Top-left: Listing panes, Bottom-left: Sending keys to right pane, Right: Output of cal(1).

11.6 Capturing pane content {#capture-pane}

`$ tmux capture-pane` will copy a panes' contents.

By default, the contents will be saved to tmux's internal clipboard, the *paste buffer*. You can run `capture-pane` within any pane, then navigate to an editor, paste the contents (don't forget to `:set paste` and go into insert mode with `i` in vim), and save it to a file. To *paste*, use `Prefix` + `]` inside the pane you're pasting into.

You can also add the `-p` flag to print it to stdout. From there, you could use redirection to place the output into a file. Let's do `>>` so we don't accidentally truncate a file:

{language=shell, line-numbers=off} $ tmux capture-pane -p >> ./test

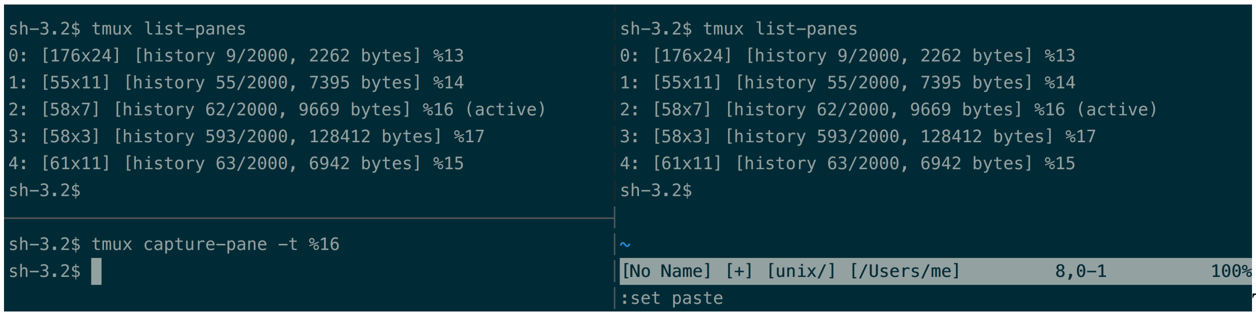
As an alternative to redirection, you can also use `save-buffer`. The `-a` flag will get you the same behavior as appended output direction.

{language=shell, line-numbers=off} $ tmux save-buffer -a ./test

To check what's inside:

{language=shell, line-numbers=off} $ cat ./test

Like with `send-keys`, *targets* can be specified with `-t`. Let's copy a pane into tmux's clipboard ("paste buffer") and paste it into a text editor in a third pane:



```
sh-3.2$ tmux list-panes
0: [176x24] [history 9/2000, 2262 bytes] %13
1: [55x11] [history 55/2000, 7395 bytes] %14
2: [58x7] [history 62/2000, 9669 bytes] %16 (active)
3: [58x3] [history 593/2000, 128412 bytes] %17
4: [61x11] [history 63/2000, 6942 bytes] %15
sh-3.2$
```

```
sh-3.2$ tmux capture-pane -t %16
sh-3.2$
```

```
sh-3.2$ tmux list-panes
0: [176x24] [history 9/2000, 2262 bytes] %13
1: [55x11] [history 55/2000, 7395 bytes] %14
2: [58x7] [history 62/2000, 9669 bytes] %16 (active)
3: [58x3] [history 593/2000, 128412 bytes] %17
4: [61x11] [history 63/2000, 6942 bytes] %15
sh-3.2$

~
[No Name] [+] [unix/] [/Users/me]          8,0-1          100%
:set paste
```

Top-left: Listing panes, Bottom-left: Capturing pane output of top-left pane, Right: Pasting buffer into vim.

Remember, you can also copy, paste, and send-keys to other windows and sessions also. Targets are server-wide.

11.7 Summary

tmux has a well-devised and intuitive command system, enabling the user to access bread and butter functionality quickly. At the same time, tmux provides a powerful way of retrieving information on its objects between `list-panes`, `list-windows` and `list-sessions` and formats. This makes tmux not only accessible and configurable, but also scriptable.

The ability to retrieve explicitly and reliably, from a session down to a pane. All it takes is a pane's ID to capture its contents or even send it keys. Used by the skilled programmer, scripting tmux can facilitate orchestrating terminals in ways previously deemed unrealistic; anything from niche shell scripts to monitor and react to behavior on systems to high-level, intelligent and structured control via object oriented libraries, like libtmux.

In the next chapter, we delve into optimizations that showcase the latest generation of unix tools that build upon old, time-tested concepts, like man pages and piping, while maintaining portability across differences in platforms and graceful degradation to ensure development tooling works on machines missing optional tools. Also, the chapter will introduce *session managers*, a powerful, high-level tool leveraging tmux's scripting capabilities to consistently load workspace via a declarative configuration.

CHAPTER 12

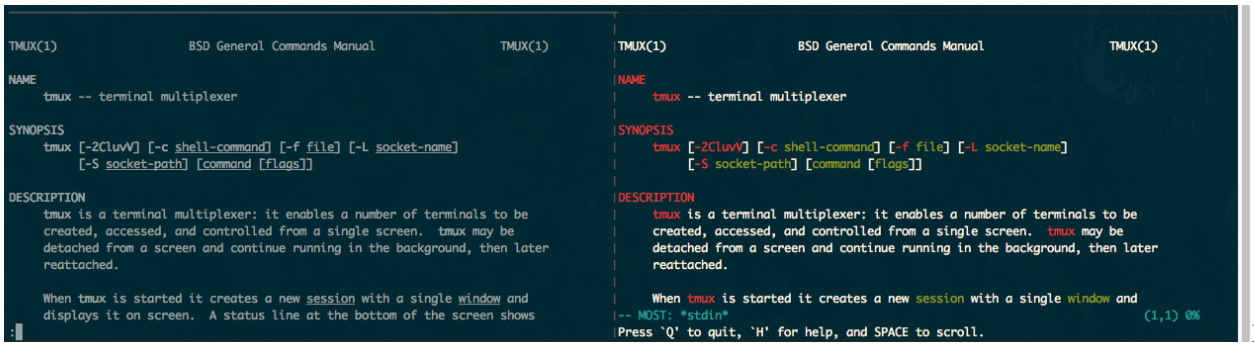
Tips and tricks {#tips-and-tricks}

12.1 Read the tmux manual in style

`$ man tmux` is the command to load up the man page for tmux. You can do the same to find instructions for any command or entity with a manpage entry; here are some fun ones:

{language=shell, line-numbers=off} $ man less $ man man $ man strftime

most(1), a solid `PAGER`, drastically improves readability of manual pages by acting as a syntax highlighter.



```
TMUX(1)                 BSD General Commands Manual                TMUX(1)

NAME
     tmux -- terminal multiplexer

SYNOPSIS
     tmux [-2CluvV] [-c shell-command] [-f file] [-L socket-name]
          [-S socket-path] [command [flags]]

DESCRIPTION
     tmux is a terminal multiplexer: it enables a number of terminals to be
     created, accessed, and controlled from a single screen.  tmux may be
     detached from a screen and continue running in the background, then later
     reattached.

     When tmux is started it creates a new session with a single window and
     displays it on screen.  A status line at the bottom of the screen shows
:
```

```
TMUX(1)                 BSD General Commands Manual                TMUX(1)

NAME
     tmux -- terminal multiplexer

SYNOPSIS
     tmux [-2CluvV] [-c shell-command] [-f file] [-L socket-name]
          [-S socket-path] [command [flags]]

DESCRIPTION
     tmux is a terminal multiplexer: it enables a number of terminals to be
     created, accessed, and controlled from a single screen.  tmux may be
     detached from a screen and continue running in the background, then later
     reattached.

     When tmux is started it creates a new session with a single window and
-- MOST: *stdin*                                                  (1,1) 0%
Press `Q' to quit, `H' for help, and SPACE to scroll.
```

left: man, version 1.6c on macOS Sierra. right: MOST v5.0.0

To get this working, you need to set your `PAGER` environmental variable to point to the MOST binary. You can test it like this:

{language=shell, line-numbers=off} $ PAGER=most man ls

If you found you like `most`, you'll probably want to make it your default manpage reader. You can do this by setting an environmental variable in your "rc" (Run Commands) for your shell. The location of the file depends on your shell. You can use `$ echo $SHELL` to find it on most shells). In Bash and zsh, these are kept in `~/.bashrc` or `~/.zshrc`, respectively:

{language=shell, line-numbers=off} export PAGER="most"

I often reuse my configurations across machines, and some of them may not have `most` installed, so I will have my scripting only set `PAGER` if `most` is found:

{language=shell, line-numbers=off} #!/bin/sh

```
if command -v most > /dev/null 2>&1; then
    export PAGER="most"
fi
```

Save this in a file, for example, to `~/.dot-config/most.sh`.

Then you can `source` it in via your main rc file.

{language=shell, line-numbers=off} source $HOME/.dot-config/most.sh

Patterns like these help make your dot-configs portable, cross-platform, and modular. For inspiration, you can fork, copy, and paste from my permissively- licensed config at https://github.com/tony/.dot-config.

12.2 Log tailing

Not tmux specific, but powerful when used in tandem with it, you can run a follow (`-f`) using `tail(1)`. More modern versions of tail have the `-F` (capitalized), which checks for file renames and rotation.

On OS X, you can do:

{language=shell, line-numbers=off} $ tail -F /var/log/system.log

and keep it running in a pane while log messages come in. It's like Facebook newsfeed for your system, except for programmers and system administrators.

For monitoring logs, multitail provides a terminal-friendly solution. It'd be an *Inception* moment, because you'd be using a log multiplexer in a terminal multiplexer.

12.3 File watching {#file-watching}

In my never-ending conquest to get software projects working in symphony with code changes, I've come to test many file watching applications and patterns. Pursuing the holy grail feedback loop upon file changes, I've gradually become the internet's unofficial connoisseur on them.

File watcher applications wait for a file to be updated, then execute a custom command, such as restarting a server, rebuilding an application, running tests, linters, and so on. It gives you, as a developer, instant feedback in the terminal, empowering a tmux workspace to have IDE-like features, without the bloat, memory, and CPU fans roaring.

I eventually settled on `entr(1)`, which works superbly across Linux distros, BSDs and OS X / macOS.

The trick to make entr work is to pipe a list of files into it to watch.

Let's search for all `.go` files in a directory and run tests on file change:

{language=shell, line-numbers=off} $ ls -d *.go | entr -c go test ./...

Sometimes, we may want to watch files recursively, but we need it to run reliably across systems. We can't depend on `**` existing to grab files recursively, since it's not portable. Something more POSIX-friendly would be `find . -print | grep -i '.*[.]go'`:

{language=shell, line-numbers=off} $ find . -print | grep -i '.*[.]go' | entr -c go test ./...

To only run file watcher if entr is installed, let's wrap in a conditional `command -v` test:

{language=shell, line-numbers=off} $ if command -v entr > /dev/null; then find . -print | grep -i '.*[.]go' | entr -c go test ./. . . ; fi

And have it fallback to `go test` in the event `entr` isn't installed. This allows your command to degrade gracefully. You'll thank me when you use this snippet in conjunction with a *session manager*:

{language=shell, line-numbers=off} $ if command -v entr > /dev/null; then find . -print | grep -i '.*[.]go' | entr -c go test ./. . . ; else go test ./. . . ; fi

If the project is a team or open source project, where a user never used the command before and could be missing a required software package, we can give a helpful message. This shows a notice to the user to install entr if not installed on the system:

{language=shell, line-numbers=off} $ if command -v entr > /dev/null; then find . -print | grep -i '.*[.]go' | entr -c go test ./. . . ; else go test ./. . . ; echo "\nInstall entr(1) to " echo "run tasks when files change. \nSee http://entrproject.org/"; fi

Here's why you want patterns like above: You can put it into a `Makefile` and commit it to your project's VCS, so you and other developers can have access to this reusable command across different UNIX-like systems, with and without certain programs installed.

Note: You may have to convert the indentation within the `Makefiles` from spaces to tabs.

Let's see what a `Makefile` with this looks like:

{language=makefile, line-numbers=off} watch_test: if command -v entr > /dev/null; then find . -print | grep -i '.*[.]go' | entr -c go test ./. . . ; else go test ./. . . ; echo "\nInstall entr(1) to run tasks when files change. \nSee http://entrproject.org/"; fi

To run this, do `$ make watch_test` in the same directory as the `Makefile`.

But it's still a tad bloated and hard to read. We have a couple tricks at our disposal. One would be to add continuation to the next line with a trailing backslash (`\`):

{language=makefile, line-numbers=off} watch_test: if command -v entr > /dev/null; then find . -print | grep -i '.*[.]go' | entr -c go test ./. . . ; else go test ./. . . ; echo "\nInstall entr(1) to run tasks on file change. \n"; echo "See http://entrproject.org/"; fi

Another would be to break the command into variables and `make` subcommands. So:

{language=makefile, line-numbers=off} WATCH_FILES= find . -type f -not -path '/.' | grep -i '.*[.]go$$' 2> /dev/null

```
test:
        go test $(test) ./...

entr_warn:
        @echo "----------------------------------------------------"
        @echo " ! File watching functionality non-operational ! "
        @echo "                                                    "
        @echo " Install entr(1) to run tasks on file change.    "
        @echo " See http://entrproject.org/                        "
        @echo "----------------------------------------------------"

watch_test:
        if command -v entr > /dev/null; then ${WATCH_FILES} | \
        entr -c $(MAKE) test; else $(MAKE) test entr_warn; fi
```

`$(MAKE)` is used for portability. One reason is recursive calls, such as here. On BSD systems, you may try invoking `make` via `gmake` (to call GNU Make specifically). This happened to me, while building PDFs for the book AlgoXY. I had to write a patch to make it properly use `$(MAKE)` for recursive calls.

The `$(test)` after `go test` allows passing a shell variable with arguments in it. So, you could do `make watch_test test='-i'`. For examples of a similar `Makefile` in action, see the one in my tmuxp project. The project is licensed BSD (permissive), so you can grab code and use it in compliance with the LICENSE.

One more thing, let's say you're running a server, like Gin, Iris, or Echo. `entr -c` likely won't be restarting the server for you. Try entering the `-r` flag to send a `SIGTERM` to the process before restarting it. Combining the current `-c` flag with the new `-r` will give you `entr -rc`:

{language=makefile, line-numbers=off} run: go run main.go

```
watch_run:
        if command -v entr > /dev/null; then ${WATCH_FILES} | \
        entr -c $(MAKE) run; else $(MAKE) run entr_warn; fi
```

12.4 Session Managers {#session-manager}

For those who use tmux regularly to perform repetitive tasks, such as opening the same software project, viewing the same logs, etc., frequent tasks will often lead to the creation of tmux scripts.

A user can use plain shell scripting to build their tmux sessions. However, scripting is error prone, hard to debug, and requires tmux to split windows into panes in a certain order. In addition, there's the burden of assuring the shell scripts are portable.

A declarative configuration in YAML or JSON configuration abstracts out the commands, layout, and options of tmux. It prevents the mistakes and repetition scripting entails. These applications are called tmux *session managers*, and in different ways, they programmatically create tmux workspaces by running a series of commands based on a config.

Teamocil and Tmuxinator are the first ones I tried. By far, the most popular one is tmuxinator. They are both programmed in Ruby. There's also tmuxomatic, where you can "draw" your tmux sessions in text and have tmuxomatic build the layout.

I sort of have a home team advantage here, as I'm author of tmuxp. Already having used teamocil and tmuxinator, I wrote my own in python instead of ruby, with many more features. For one, it builds on top of libtmux, a library which abstracts tmux *server*, *sessions*, *windows* and *panes* to build the state of tmux sessions. In addition, it has a naive form of session freezing, support for JSON, more flexible configuration options, and it will even offer to attach exiting sessions, instead of redundantly running script commands against the session if it's already running.

So, in tmuxp, we'll hollow out a tmuxp config directory with `$ mkdir ~/.tmuxp` then create a YAML file at `~/.tmuxp/test.yaml`:

{language=yaml, line-numbers=off} session_name: 4-pane-split windows: - window_name: dev window layout: tiled shell_command_before: - cd ~/ # run as a first command in all panes panes: - shell_command: # pane no. 1 - cd /var/log # run multiple commands in this pane - ls -al | grep .log - echo second pane # pane no. 2 - echo third pane # pane no. 3 - echo forth pane # pane no. 4

gives a session titled *4-pane-split*, with one window titled *dev window* with 4 panes in it. 3 in the home directory; the other is in `/var/log` and is printing a list of all files ending with `.log`.

To launch it, install tmuxp and load the configuration:

{language=shell, line-numbers=off} $ pip install –user tmuxp $ tmuxp -V # verify tmuxp is installed, if not you need to fix your `PATH` # to point to your python bin folder. More help below. $ tmuxp load ~/.tmuxp/test.yaml

If tmuxp isn't found, there is a *troubleshooting entry on fixing your paths* in the appendix.

12.5 More code and examples {#example-projects}

I've dusted off a C++ space shooter and a new go webapp I've been playing with. They're licensed under MIT so, you can use them, copy and paste from them, etc:

- C++14 space shooter minigame - side scrolling shmup demo (sdl2, cmake, json resource manifests, Linux/BSD/OS X compatible)
- Go tmux web frontend - display current tmux session and window information via browser (gin, bower)

Both support `tmuxp load .` within the project directory to load up the project.

Make sure to install `entr(1)` beforehand!

12.6 tmux-plugins and tpm

tmux-plugins and tmux package manager are a suite of tools dedicated to enhancing the experience of tmux users.

- tmux-resurrect: Persists tmux environment across system restarts.
- tmux-continuum: Continuous saving of tmux environment. Automatic restore when tmux is started. Automatic tmux start when computer is turned on.
- tmux-yank: Tmux plugin for copying to system clipboard. Works on OSX, Linux and Cygwin.
- tmux-battery: Plug and play battery percentage and icon indicator for Tmux.

CHAPTER 13

Takeaway {#takeaway}

In this book, we've taken an organized approach to understanding tmux. As you use tmux more and more, continue to come back and use this resource to help wrap your brain around concepts. You do not have to understand the intricacies of tmux, let alone the terminal, in a single sitting. Acclimation happens over time.

tmux's userbase varies in skill level. Some readers of this book may have just learned how to use the `Prefix` key yesterday. Others are looking to tweak their configurations and host it in their "dot files" on github. There also exists a very clever hacker who utilizes the advanced scripting capabilities tmux offers to pilot the terminal in ways previously thought impossible.

We've covered the *server*, *session*, *window*, and *pane* concepts. Panes are shells, AKA pseudoterminals or PTYs. The command system. That *configuration* is basically a file filled with commands. An overview of the *target* system lets you specify objects to interact with tmux commands. A breeze through *formats*, a template system with variables to retrieve information on tmux's current state. How to *send keystrokes* and *copy from tmux panes* programmatically. A lot of *terminal tricks* that work across platforms and well with tmux, including a *file watching workflow* to run linting, testing, and build commands on file changes. *Two permissively licensed open source projects* for demonstration. A tmux configuration you can copy and paste from. An object oriented tmux API wrapper and a tmux session manager.

If you liked this book, please leave a review on Amazon and Goodreads. I would also appreciate you leaving something in my tip jar. I am an independent software developer and could use all the help I can get.

If you found an error or have a suggestion, please contact me at `tao.of.tmux@git-pull.com`. I want this book to be the best it can be. If you are having technical difficulties with Kindle, please send me your receipt and I will comp you a leanpub coupon.

{backmatter}

CHAPTER 14

Appendix: Cheatsheets {#appendix-cheatsheets}

These are taken directly from tmux's manual pages, tabled and organized by hand into sections for convenience.

14.1 Commands

14.1.1 Session

{width="wide"}

Command	Action
no command	Short-cut for `new-session`
`attach-session`	Attach or switch to a session
`choose-session`	Put a window into session choice mode
`has-session`	Check and report if a session exists on the server
`kill-session`	Destroy a given session
`list-sessions`	List sessions managed by server
`lock-session`	Lock all clients attached to a session
`new-session`	Create a new session
`rename-session`	Rename a session

14.1.2 Window

{width="wide"}

Command	Action
`choose-window`	Put a window into window choice
`find-window`	Search for a pattern in windows
`kill-window`	Destroy a given window
`last-window`	Select the previously selected
`link-window`	Link a window to another
`list-windows`	List windows of a session
`move-window`	Move a window to another
`new-window`	Create a new window
`next-window`	Move to the next window in a sesssion
`previous-window`	Move to the previous window in session
`rename-window`	Rename a window
`respawn-window`	Reuse a window in which a command has exited
`rotate-window`	Rotate positions of panes in a window
`select-window`	Select a window
`set-window-option`	Set a window option
`show-window-options`	Show window options
`split-window`	Splits a pane into two
`swap-window`	Swap two windows
`unlink-window`	Unlink a window

14.1.3 Pane

{width="wide"} | Command | Action | |————|——————————————| | `break-pane` | Break a pane from an existing into a new window | | `capture-pane` | Capture the contents of a pane to a buffer | | `display-panes` | Display an indicator for each visible pane | | `join-pane` | Split a pane and move an existing one into the new space | | `kill-pane` | Destroy a given pane | | `last-pane` | Select the previously selected pane | | `list-panes` | List panes of a window | | `move-pane` | Move a pane into a new space | | `pipe-pane` | Pipe output from a pane to a shell command | | `resize-pane` | Resize a pane | | `respawn-pane` | Reuse a pane in which a command has exited | | `select-pane` | Make a pane the active one in the window | | `swap-pane` | Swap two panes |

{pagebreak}

14.2 Keybindings

{width="wide"} | Shortcut | Action | |————|——————————————-| |`C-b` | Send the prefix key (C-b) through to the | | | application. |

14.2.1 Miscellaneous

{width="wide"} | Shortcut | Action | |————|——————————————-| |`C-z` | Suspend the tmux client. | |`r` | Force redraw of the attached client. | |`t` | Show the time. | |~ | Show previous messages from tmux, if any. | |`f` | Prompt to search for text in open windows. | |`d` | Detach the current client. | |`D` | Choose a client to detach. | |`?` | List all key bindings. | |`:` | Enter the tmux command prompt. |

14.2.2 Copy/Paste

{width="wide"} | Shortcut | Action | |————|——————————————-| |`#` | List all paste buffers. | |`[` | Enter copy mode to copy text or view the history. | |`]` | Paste the most recently copied buffer of text. | |`Page Up` | Enter copy mode and scroll one page up. | |= | Choose which buffer to paste interactively from a | | | list. | |`-` | Delete the most recently copied buffer of text. |

{pagebreak}

14.2.3 Session

{width="wide"} | Shortcut | Action | |————|——————————————-| | | Rename the current session. |

Session Traversal

{width="wide"} | Shortcut | Action | |————|——————————————-| |`L` | Switch the attached client back to the last | | | session. | |`s` | Select a new session for the attached client | | | interactively. |

{pagebreak}

14.2.4 Window

{width="wide"} | Shortcut | Action | |—————————|————————————————————-| |`c` | Create a new window. | |`&` | Kill the current window. | |`i` | Display some information about the current window. | |`,` | Rename the current window. |

Window Traversal

{width="wide"} | Shortcut | Action | |—————————|————————————————————-| |`0 to 9` | Select windows 0 to 9. | |`w` | Choose the current window interactively. | |`M-n` | Move to the next window with a bell or activity | | | marker. | |`M-p` | Move to the previous window with a bell or activity| | | marker. | |`p` | Change to the previous window. | |`n` | Change to the next window. | |`l` | Move to the previously selected window. | |`'` | Prompt for a window index to select. |

Window Moving

{width="wide"} | Shortcut | Action | |—————————|————————————————————-| |`.` | Prompt for an index to move the current window |

{pagebreak}

14.2.5 Pane

{width="wide"} | Shortcut | Action | |—————————|————————————————————-| |`x` | Kill the current pane. | |`q` | Briefly display pane indexes. | |`%` | Split the current pane into two, left and right. | |`"` | Split the current pane into two, top and bottom. |

Pane Traversal

{width="wide"} | Shortcut | Action | |—————————|————————————————————-| |`;` | Move to the previously active pane. | |`Up, Down` | Change to the pane above, below, to the left, or to| |`Left, Right` | the right of the current pane. | |`o` | Select the next pane in the current window. |

Pane Moving

{width="wide"} | Shortcut | Action | |—————————|————————————————————-| |`C-o` | Rotate the panes in the current window forwards. | |`M-o` | Rotate the panes in the current window backwards. | |`{` | Swap the current pane with the previous pane. | |`}` | Swap the current pane with the next pane. | |`!` | Break the current pane out of the window. |

Pane Resizing

{width="wide"} | Shortcut | Action | |—————————|————————————————————-| |`M-1 to M-5` | Arrange panes in one of the five preset layouts: | | | even-horizontal, even-vertical, main-horizontal, | | | main-vertical, or tiled. | |`C-Up, C-Down` | Resize the current pane in steps of one cell. | |`C-Left, C-Right` | | |`M-Up, M-Down` | Resize the current pane in steps of five cells. | |`M-Left, M-Right` | |

{pagebreak}

14.3 Formats {#appendix-formats}

14.3.1 Copy / paste

{width="wide"} | Variable name | Description | |——————————|————————————————————| | buffer_name |Name of buffer | | buffer_sample |Sample of start of buffer | | buffer_size |Size of the specified buffer in bytes |

14.3.2 Clients

{width="wide"} | Variable name | Description | |——————————|————————————————————| | client_activity |Integer time client last had activity | | client_created |Integer time client created | | client_control_mode |1 if client is in control mode | | client_height |Height of client | | client_key_table |Current key table | | client_last_session |Name of the client's last session | | client_pid |PID of client process | | client_prefix |1 if prefix key has been pressed | | client_readonly |1 if client is readonly | | client_session |Name of the client's session | | client_termname |Terminal name of client | | client_tty |Pseudo terminal of client | | client_utf8 |1 if client supports utf8 | | client_width |Width of client | | line |Line number in the list |

14.3.3 Panes

{width="wide"} | Variable name | Description | |——————————|————————————————————| | alternate_on |If pane is in alternate screen | | alternate_saved_x |Saved cursor X in alternate screen | | alternate_saved_y |Saved cursor Y in alternate screen | | cursor_flag |Pane cursor flag | | cursor_x |Cursor X position in pane | | cursor_y |Cursor Y position in pane | | insert_flag |Pane insert flag | | keypad_cursor_flag |Pane keypad cursor flag | | keypad_flag |Pane keypad flag | | mouse_any_flag |Pane mouse any flag | | mouse_button_flag |Pane mouse button flag | | mouse_standard_flag |Pane mouse standard flag | | pane_active |1 if active pane | | pane_bottom |Bottom of pane | | pane_current_command |Current command if available | | pane_current_path |Current path if available | | pane_dead |1 if pane is dead | | pane_dead_status |Exit status of process in dead pane | | pane_height |Height of pane | | pane_id |Unique pane ID (Alias: #D) | | pane_in_mode |If pane is in a mode | | pane_input_off |If input to pane is disabled | | pane_index |Index of pane (Alias: #P) | | pane_left |Left of pane | | pane_pid |PID of first process in pane | | pane_right |Right of pane | | pane_start_command |Command pane started with | | pane_synchronized |If pane is synchronized | | pane_tabs |Pane tab positions | | pane_title |Title of pane (Alias: #T) | | pane_top |Top of pane | | pane_tty |Pseudo terminal of pane | | pane_width |Width of pane | | scroll_region_lower |Bottom of scroll region in pane | | scroll_region_upper |Top of scroll region in pane | | scroll_position |Scroll position in copy mode | | wrap_flag |Pane wrap flag |

14.3.4 Sessions

{width="wide"} | Variable name | Description | |——————————|————————————————————| | session_alerts |List of window indexes with alerts | | session_attached |Number of clients session is attached to | | session_activity |Integer time of session last activity | | session_created |Integer time session created | | session_last_attached |Integer time session last attached | | session_group |Number of session group | | session_grouped |1 if session in a group | | session_height |Height of session | | session_id |Unique session ID | | session_many_attached |1 if multiple clients attached | | session_name |Name of session (Alias: #S) | | session_width |Width of session | | session_windows |Number of windows in session |

14.3.5 Windows

{width="wide"} | Variable name | Description | |——————————|————————————————————| | history_bytes |Number of bytes in window history | | history_limit |Maximum window history lines | | history_size |Size of history in bytes | | window_activity |Integer time of window last activity | | window_activity_flag |1 if window has activity | | window_active |1 if window active | | window_bell_flag |1 if window has bell | | window_find_matches |Matched data from the find-window | | window_flags |Window flags (Alias: #F) | | window_height |Height of window | | window_id |Unique window ID | | window_index |Index of window (Alias: #I) | | window_last_flag |1 if window is the last used | | window_layout |Window layout description, ignoring zoomed| | |window panes | | window_linked |1 if window is linked across sessions | | window_name |Name of window (Alias: #W) | | window_panes |Number of panes in window | | window_silence_flag |1 if window has silence alert | | window_visible_layout |Window layout description, respecting | | |zoomed window panes | | window_width |Width of window | | window_zoomed_flag |1 if window is zoomed |

14.3.6 Servers

{width="wide"} | Variable name | Description | |——————————|——————————————————| | host |Hostname of local host (alias: #H) | | host_short |Hostname of local host (no domain name) | | |(alias: #h) | | socket_path |Server socket path | | start_time |Server start time | | pid |Server PID |

14.3.7 Commands

For `$ tmux list-commands`.

{width="wide"} | Variable name | Description | |——————————|——————————————————| | command_hooked |Name of command hooked, if any | | command_name |Name of command in use, if any | | command_list_name |Command name if listing commands | | command_list_alias |Command alias if listing commands | | command_list_usage |Command usage if listing commands |

CHAPTER 15

Appendix: Installing tmux {#appendix-installation}

15.1 macOS / OS X

15.1.1 brew

{language=shell, line-numbers=off} $ brew install tmux

15.1.2 macports

{language=shell, line-numbers=off} $ sudo port install tmux

15.1.3 fink

{language=shell, line-numbers=off} $ fink install tmux

15.2 Linux

15.2.1 Ubuntu / Mint / Debian, etc.

{language=shell, line-numbers=off} $ sudo apt-get install tmux

15.2.2 CentOS / Fedora / Redhat, etc.

{language=shell, line-numbers=off} $ sudo yum install tmux

15.2.3 Arch Linux (pacman)

{language=shell, line-numbers=off} $ sudo pacman -S tmux

15.2.4 Gentoo (portage)

{language=shell, line-numbers=off} $ sudo emerge –ask app-misc/tmux

15.3 BSD

15.3.1 FreeBSD

pkg(1)

{line-numbers=off} # pkg install tmux

pkg_add(1)

{line-numbers=off} # pkg_add -r tmux

15.3.2 OpenBSD

As of OpenBSD 4.6, tmux is part of the base system.

If you are using an earlier version:

{line-numbers=off} # pkg_add tmux

15.3.3 NetBSD

{language=shell, line-numbers=off} $ make -C /usr/pkgsrc/misc/tmux install

15.4 Windows 10

Check out the *tmux on Windows 10 appendix section*.

CHAPTER 16

Appendix: tmux on Windows 10 {#appendix-windows-bash}

As of Windows 10 build 14361, you can run tmux via the Linux Subsystem feature.

Usage requires enabling **Developer mode** via the “For Developers” tab in the “Update & security” settings.

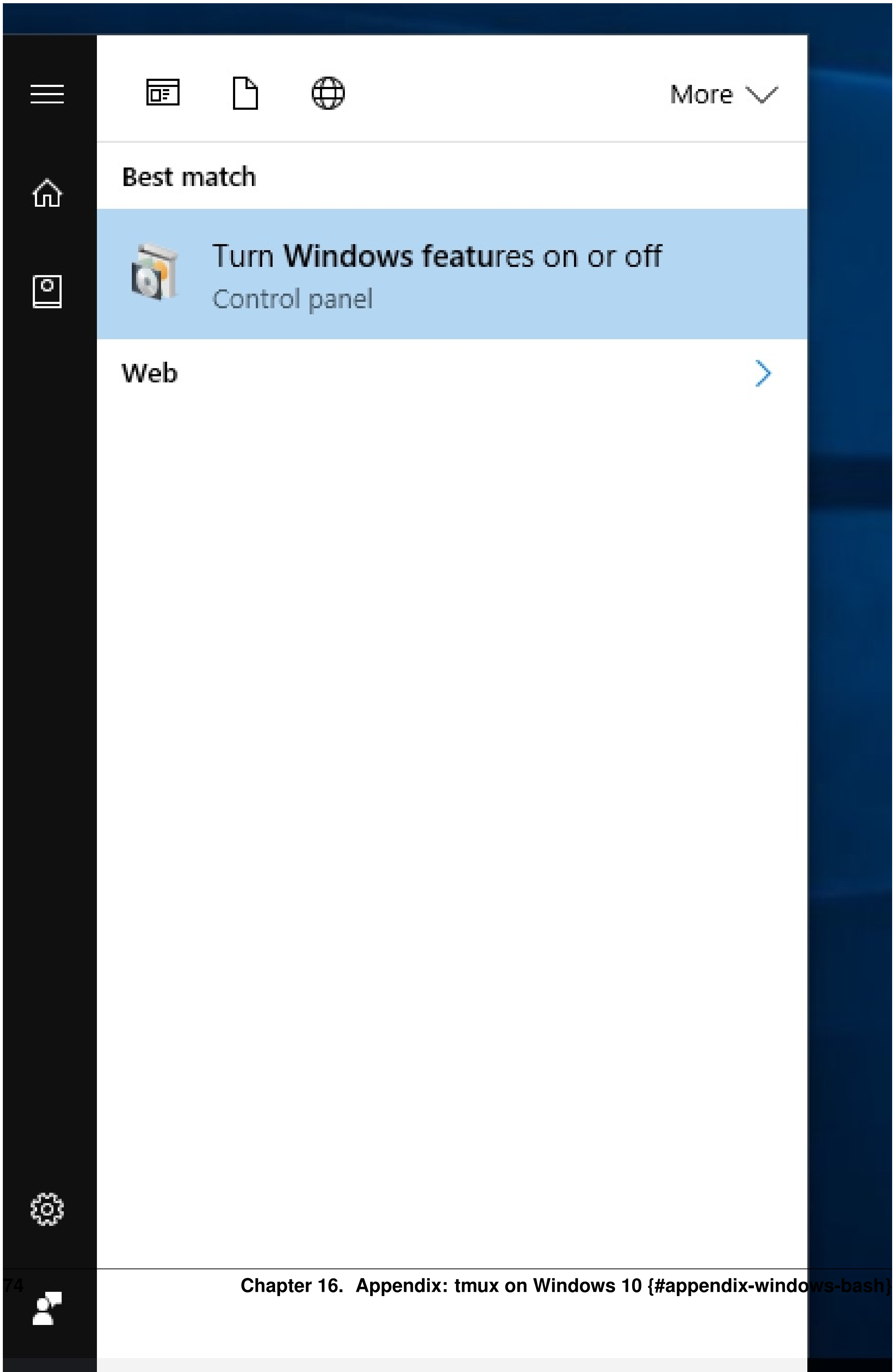
After enabling, open “Windows Features”. You can find it by searching for “Turn Windows features on or off”. Then check “Windows Subsystem for Linux (Beta)”.

You may be asked to restart.

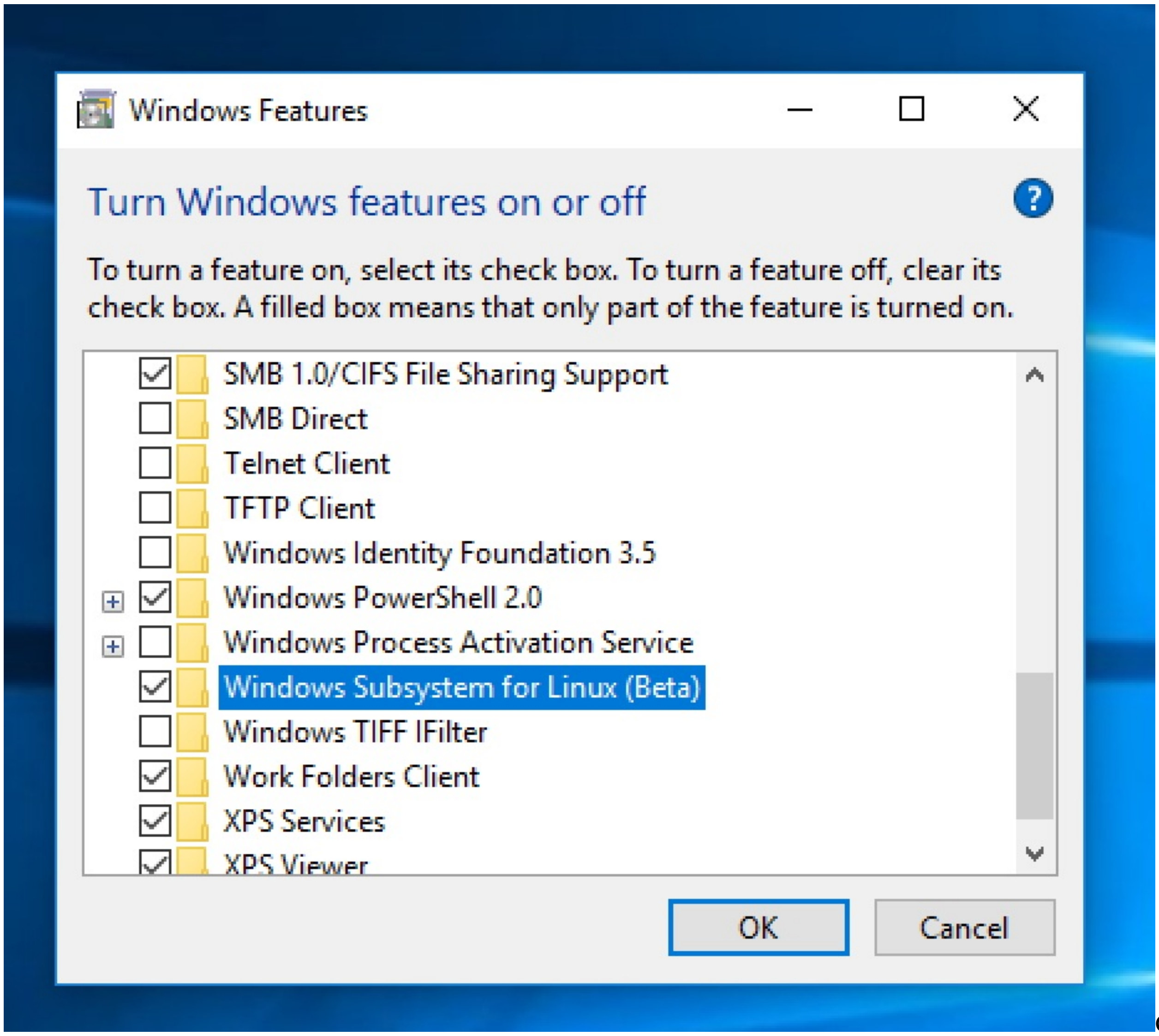
Then open Command Prompt as you normally would (Run cli.exe). Then type

```
C:\Users\tony> bash.exe
```

It will prompt you to agree to terms, create a user. In my build, tmux was already installed! But if it’s not, type `sudo apt-get install tmux`.


More
Best match
Turn Windows features on or off
Control panel
Web

Turn Windows Features on or off



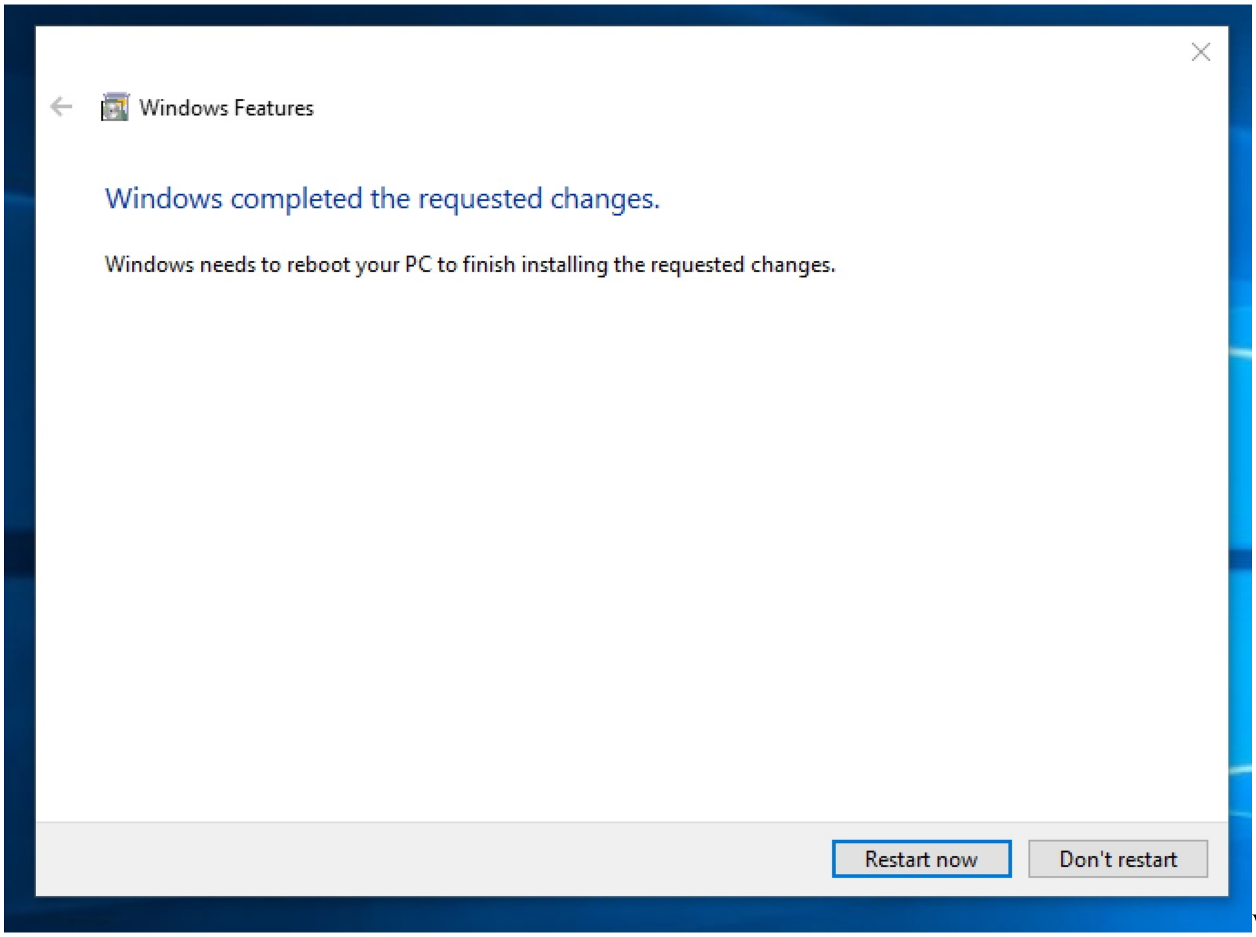
Windows Features

Turn Windows features on or off

To turn a feature on, select its check box. To turn a feature off, clear its check box. A filled box means that only part of the feature is turned on.

- [x] SMB 1.0/CIFS File Sharing Support
- [ ] SMB Direct
- [ ] Telnet Client
- [ ] TFTP Client
- [ ] Windows Identity Foundation 3.5
- [x] Windows PowerShell 2.0
- [ ] Windows Process Activation Service
- [x] Windows Subsystem for Linux (Beta)
- [ ] Windows TIFF IFilter
- [x] Work Folders Client
- [x] XPS Services
- [x] XPS Viewer

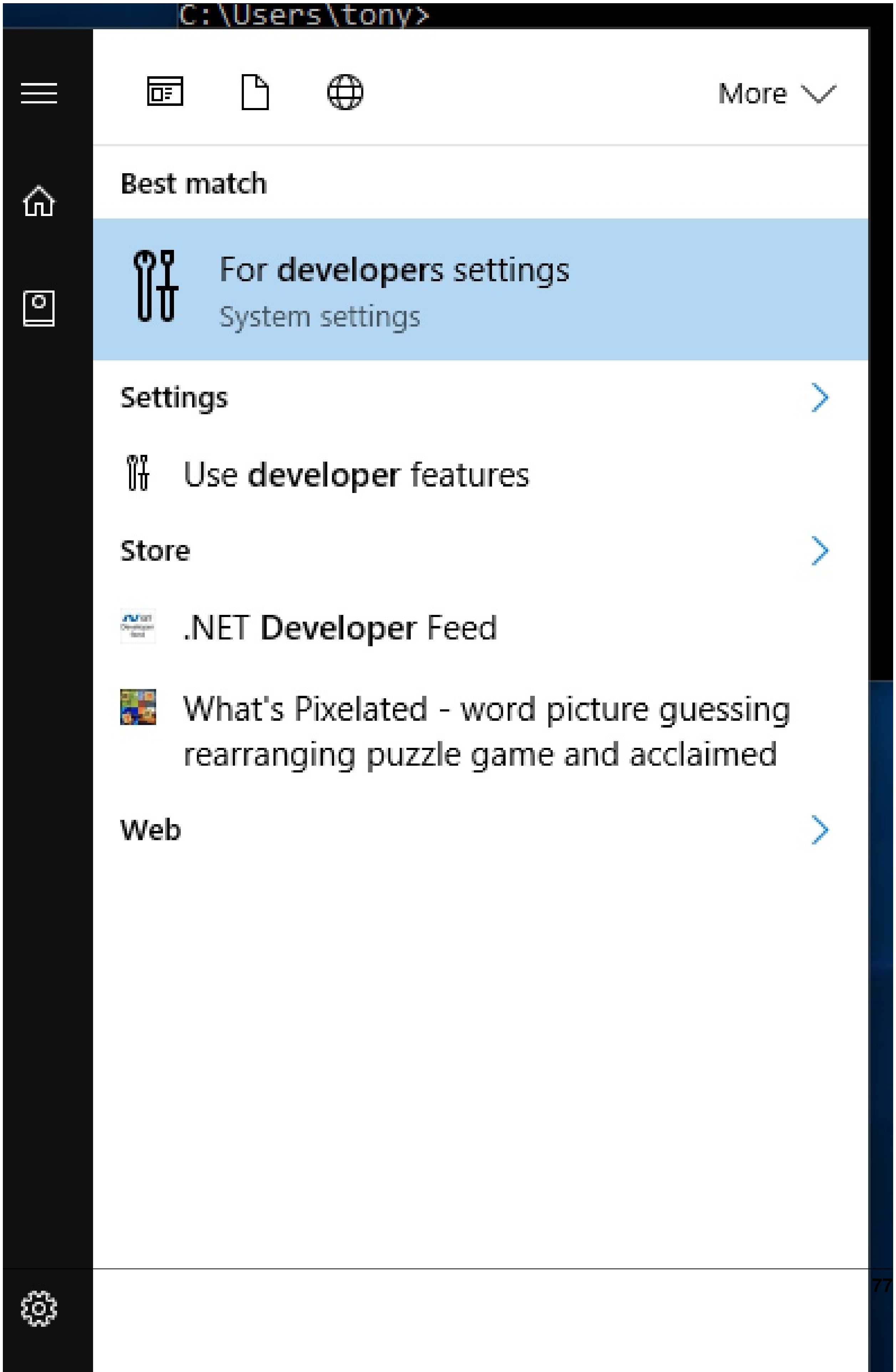
Check Windows Subsystem for Linux (Beta)



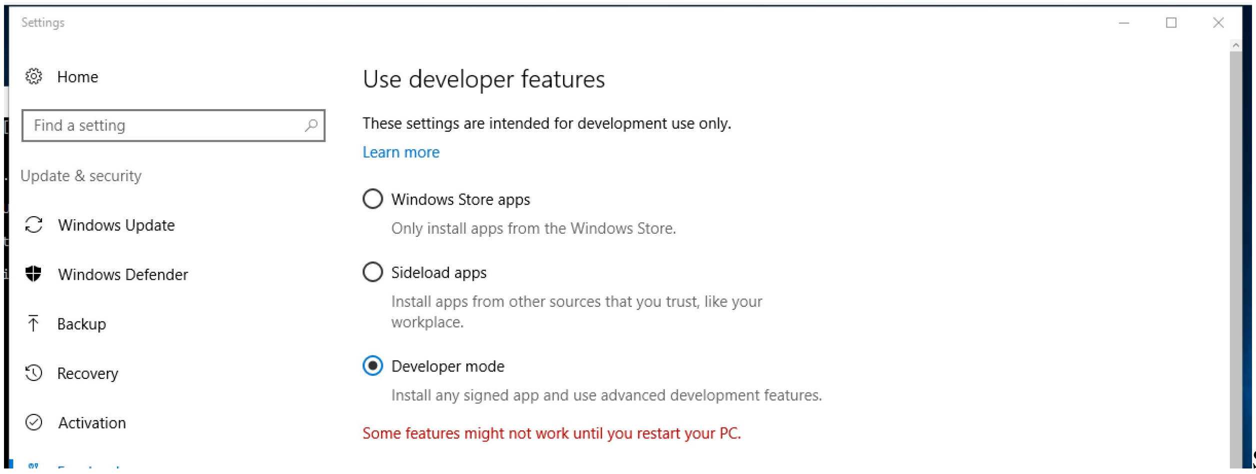
Windows completed the requested changes.

Windows needs to reboot your PC to finish installing the requested changes.

Windows completed the requested changes. Restart


C:\Users\tony>
More
Best match
For developers settings
System settings
Settings
Use developer features
Store
.NET Developer Feed
What's Pixelated - word picture guessing rearranging puzzle game and acclaimed
Web

Developer features


Settings

Home

Find a setting

Update & security

Windows Update

Windows Defender

Backup

Recovery

Activation

Use developer features

These settings are intended for development use only.

Learn more

Windows Store apps

Only install apps from the Windows Store.

Sideload apps

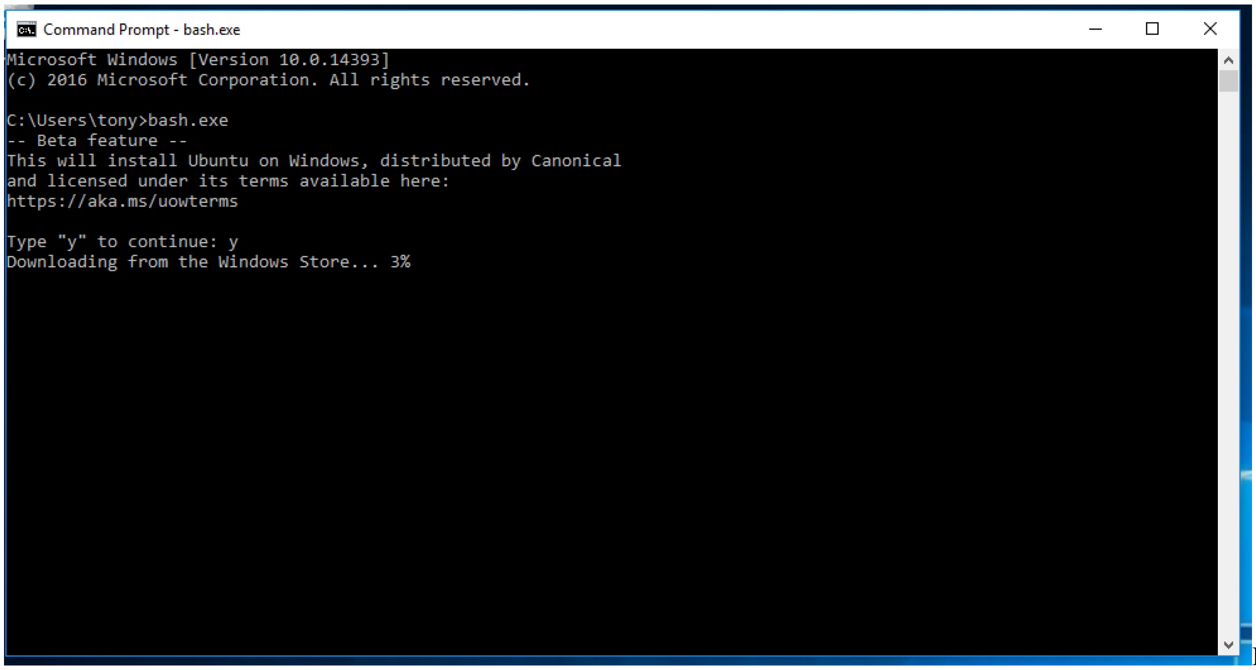
Install apps from other sources that you trust, like your workplace.

[✓ Developer mode]

Install any signed app and use advanced development features.

Some features might not work until you restart your PC.

Select Developer mode in Update & Security

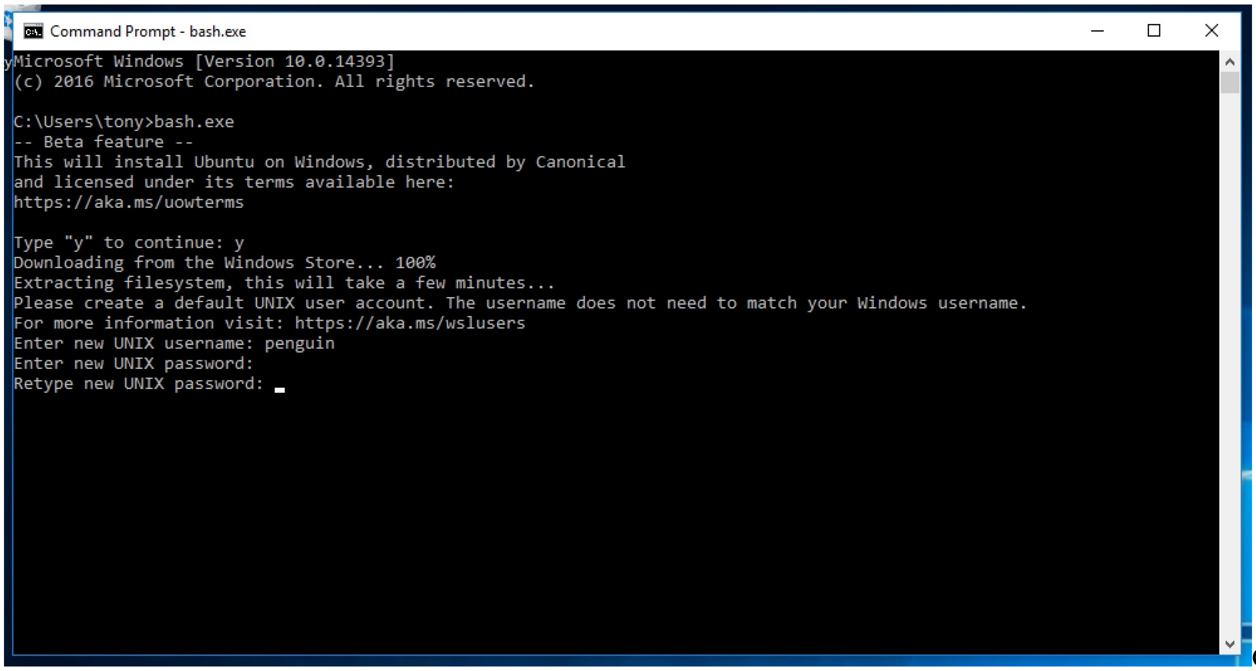

Command Prompt - bash.exe

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\tony>bash.exe
-- Beta feature --
This will install Ubuntu on Windows, distributed by Canonical
and licensed under its terms available here:
https://aka.ms/uowterms

Type "y" to continue: y
Downloading from the Windows Store... 3%
```

Installing Ubuntu from Windows Store

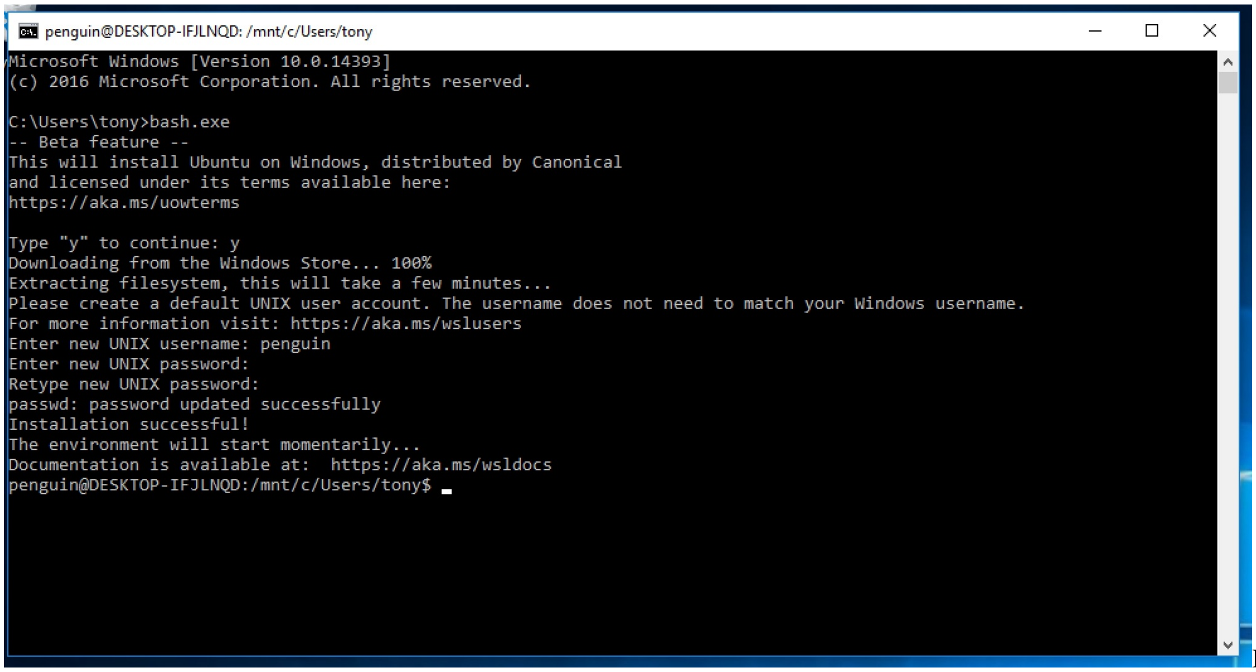


```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\tony>bash.exe
-- Beta feature --
This will install Ubuntu on Windows, distributed by Canonical
and licensed under its terms available here:
https://aka.ms/uowterms

Type "y" to continue: y
Downloading from the Windows Store... 100%
Extracting filesystem, this will take a few minutes...
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: penguin
Enter new UNIX password:
Retype new UNIX password:
```

Create Linux user



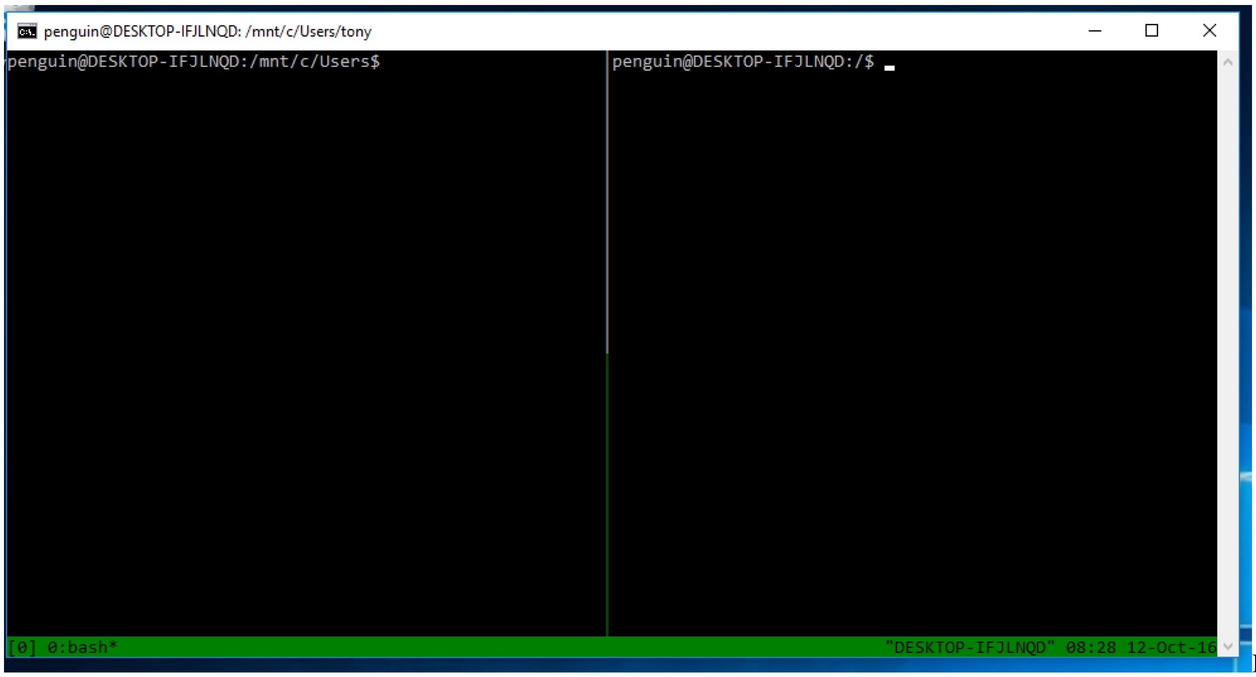
```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\Users\tony>bash.exe
-- Beta feature --
This will install Ubuntu on Windows, distributed by Canonical
and licensed under its terms available here:
https://aka.ms/uowterms

Type "y" to continue: y
Downloading from the Windows Store... 100%
Extracting filesystem, this will take a few minutes...
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: penguin
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Installation successful!
The environment will start momentarily...
Documentation is available at:  https://aka.ms/wsldocs
penguin@DESKTOP-IFJLNQD:/mnt/c/Users/tony$
```

In bash!

```
yourusername@COMPUTERNAME-ID321FJ:/mnt/c/Users/username$ tmux
```



```
penguin@DESKTOP-IFJLNQD:/mnt/c/Users$
```

```
penguin@DESKTOP-IFJLNQD:/$
```

```
[0] 0:bash*                                    "DESKTOP-IFJLNQD" 08:28 12-Oct-16
```

In tmux!

This should allow you to run tmux within bash.exe.

This is a real ubuntu installation, so you can continue to install packages via `sudo apt-get install **packagename**` and update packages via `sudo apt-get update && sudo apt-get upgrade`.

CHAPTER 17

Appendix: Troubleshooting {#appendix-troubleshooting}

17.1 `E353: Nothing in register *` when pasting on vim

If you are using macOS / OS X with vim inside tmux, you may get the error `E353: Nothing in register *` when trying to paste.

Try installing `reattach-to-user-namespace` via brew.

{language=shell, line-numbers=off} $ brew install reattach-to-user-namespace

17.2 `tmuxp: command not found` and `powerline: command not found` {#troubleshoot-site-paths}

This is due to your site package bin path (where application entry points are installed) not being in your paths. To find your user site packages base directory:

{language=shell, line-numbers=off} $ python -m site –user-base

This will get you something like `/Users/me/Library/Python/2.7` on macOS with Python 2.7 or `/home/me/.local` on Linux/BSD boxes.

The applications are in the `bin/` folder inside. So, concatenate the two and apply them to your `PATH`. This can be done automatically on every shell session by using one of these in your `~/.bashrc` or `~/.zshrc`:

{language=shell, line-numbers=off} export PATH=/Users/me/Library/Python/2.7/bin:$PATH # macOS w/ python 2.7 export PATH=$HOME/.local/bin:$PATH # Linux/BSD export PATH=”`python -m site --user-base`/bin”:$PATH # May work all-around

Then open a new terminal, or `. ~/.zshrc` / `. ~/.bashrc` in your current one. Then you can run `$ tmuxp -V`, `$ tmuxp load` and `$ powerline tmux right` commands.

CHAPTER 18

Indices and tables

- search