Painless Tmux

A Sane Person's Guide to Command Line Happiness

by Nate Dickson

Painless Tmux

A Sane Person's Guide to Command Line Happiness

Nate Dickson

This book is for sale at http://leanpub.com/painless_tmux

This version was published on 2015-06-16

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Nate Dickson by spreading the word about this book on Twitter!

The suggested hashtag for this book is #painlessTmux.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#painlessTmux

Also By Nate Dickson

Painless Vim

Contents

About this Beta Book

Painless Tmux is in **beta** status, meaning all the chapters and sections that are going to be in the book are now in the book. Entire sections are no longer likely to be replaced with pictures of bunnies, but the text is still in a pretty rough and ready state.

The upside is that you can be an active participant in the development of *Painless Tmux*. Reader feedback will be taken seriously and will help me focus on what makes this book best for *you*.

Thank you for supporting this book, even in these early stages. You're incredible, I hope you know that!

Preface

Why This Book?

I wasn't going to write a tmux book. I never meant to start writing technical books in the first place. But a number of readers of *Painless Vim* asked if I would be writing another tech book, and if I did could I please make it about tmux?

Which felt like a weird request at first. But it makes sense. If you're using vim you're already (probably) in a terminal, and if you're editing text or code with all these keyboard shortcuts why don't we take it one step further and make it so you can do *everything* with awesome keyboard shortcuts?

So here we are. A book about tmux. And honestly, I'm pretty excited. Because *Painless Vim* was hard to write, for a couple of reasons:

- I was new to vim
- I was new to writing tech books
- Vim is a challenging topic

Painless Tmux is completely different: I've been using Tmux for over a year, and consider it a mainstay of my daily routine. I know exactly what I want to show you, and I'm confident that I can get you to a stable, productive and honestly amazing tmux workflow without too much hassle. So, let's get right into it!

Introduction

What is Tmux and Why Do I Need It?

In a day and age where monitors are huge and processors are powerful it doesn't make sense that we're limited to a single command line interface per terminal session. Tmux is a way to fix that. When you're using tmux you're in charge of an endless army of terminals that you can use to monitor and command all aspects of your work from a single window. So you can monitor output from your server while you write new code, run your compilation scripts, and do it all while looking like the world's most über häcker. If you spend more than a minute or two a day on the command line you need to be using tmux, and the best part is that it's pretty simple to get set up and comfortable.

What This Book Is

This book aims to be:

Short

You've got better things to do than read books about terminal multiplexers[1]. Tmux isn't that hard, once you get into it, and I'll keep this brief so you can get on with the important stuff you're doing.

Cheap

tmux is free. While I don't want to sell my work for nothing, I'm coming pretty darn close. This is a beginner's book, and my goal is to lower the bar to entry. To start with, I'll price it low and hope that helps you along.

Easy to understand

My main goal here is to help you build confidence and get your feet under you. If you're reading this during the beta period and you've got suggestions, please send them on over to me! I'm more than willing to listen and change.

[1] at least, I *hope* you do. Then again, apparently I don't have anything better to do than *write* books about terminal multiplexers...

Friendly!

My high horse died a while back, and I never bothered to buy myself a soapbox, so I won't be doing any preaching. I think tmux is an awesome way to be productive and look good doing it, and I want to help you have that same level of fun and power.

Opinionated (but in a good way)

I'll keep the tone friendly, but I got *opinions* people. There are things I think are good and things I think aren't so good. But I'll let you know *why* I think some things are good or not so good and let you make up your mind on your own.

Lighthearted, but serious

The *tone* of the book is meant to be light and approachable, but the *advice* is meant to be rock solid. I'm not going to have you do things to your configuration files that I wouldn't do, even temporarily, or to prove a point. The goal is to make it easy for you to get things *right* [2] the first time.

What This Book Isn't

This book definitely is *not*:

The end-all-be-all reference on tmux

I'm here to help you get started, not to be the only book you'll ever read on the subject. Tmux is actively changing, and once you get your feet under you it will be easy for you to adapt to those changes on your own. I'm here to help you spread your wings and fly, free and on your own. You can do it!

A list of tips

A lot of the stuff in the later chapters won't make a ton of sense if you haven't been through the earlier chapters. The chapters are laid out iteratively, starting with small changes that build up to a final configuration that should work well. Then you'll have a good jumping-off point for your further experiments.

A deep dive

The goal here is to give you enough information to be comfortable, but not all the underlying technologies and line editor commands and other such things. Again, these are available online and in many other reference guides all over the place. When you decide that you want to know more about something you'll be able to find it. I have faith in you.

[2] According to me, see the whole "Opinionated" thing up above.

Who am I?

I'm Nate Dickson, it says so right on the cover.

I'm also the author of *Painless Vim*[3], a book that seeks to make learning vim less terrible. Beyond that I'm a professional programmer by day, author, blogger, and daddy by night. You can check out a fairly large amount of my writing on my personal blog[4], my sporadically maintained tech blog[5], or my less-maintained Apple blog[6]. You can also check me out on Twitter[7], Github[8] and App.net[9].

References

I will refer you to a lot of other books, posters, blogs, wikis, etc. etc. over the course of this book. For the most part these are things I've used extensively myself, and can heartily recommend. Again, I'll try to provide reasons for my references and let you decide if it sounds like something you'd like to investigate on your own.

The Obligatory "Conventions Used in This Book" Section

First off, regarding the actual word "tmux". In general I will write it all lowercase, because that's how most people think of it, like "perl" or "ruby". That's the command you enter to start the editor, and that's really the name of the program.

However, when it's in a book, chapter, or section title I will capitalize it, just as I would any other word in a title: *Painless Tmux*.

Occasionally I'll put in chapters called "Interludes". These are chapters that are less intense than the regular ol' chapters, and more focused on taking a look back over what we've already covered and crystalizing what we've learned. Think of them as nice places to sit down and reflect on how wise you are becoming.

Speaking of Opinions

There are times when I will be speaking not out of absolute fact, but out of my personal preferences or opinions. When such is the case, I will set these discussions off with this cool little "dialog" icon. I will seek to give good reasons for the opinions I state in these sections, but I won't be hurt if you decide I'm wrong, and ignoring the opinion sections shouldn't break mess up your final configuration.

[3] https://leanpub.com/painless_vim
[4] http://natedickson.com
[5] Coals 2 Newcastle
[6] http://crazyapplenews.com
[7] @poginate
[8] http://github.com/PogiNate
[9] @poginate. Yeah, I see the pattern too.

Other than that, I use the same conventions everyone else uses. Code appears in `a monospaced font`, and longer blocks of code will appear

```
in a large block
of monospaced text.
```

Special keys like `<shift>`, `<enter>`, `<esc>` will have angle brackets around the name to make it real obvious that you shouldn't be typing that word, you should be pressing that key. If you need to press `<ctrl>` and another key at the same time (and you *will* need to do this, quite frequently) I'll just write them like so: `<ctrl>b`.

If you need to press keys in sequence I will put a space between the keys. So if you need to press `g` and then `t` I would type it like this:

```
g t
```

This will usually happen when you start off a command made up of two keys, and then press a third key:

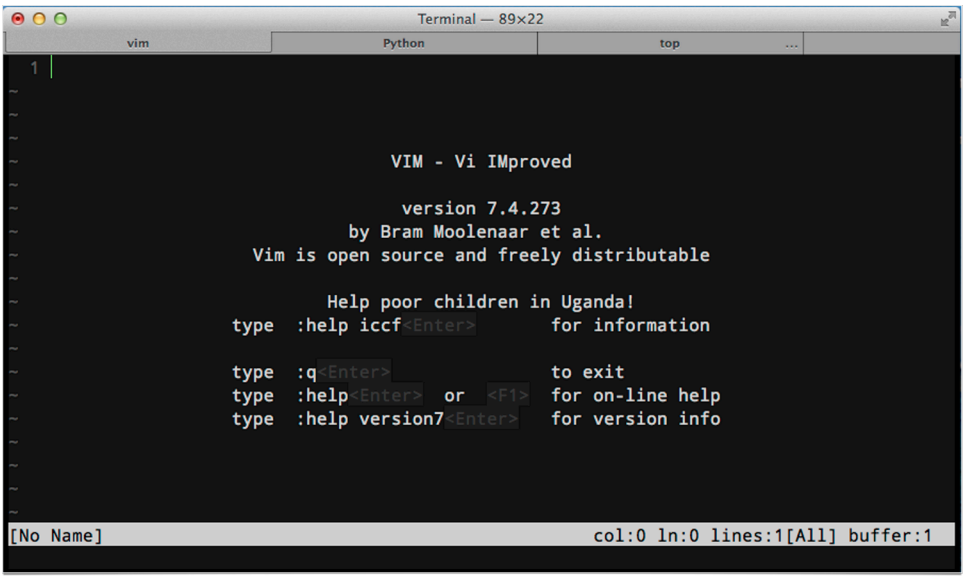
```
<ctrl>b x
```

In this case you would press `<ctrl>` and `b` at the same time, then release them and press `x`.

Again, this will all make more sense in a bit. For now let's get moving.

1. Say Hi to Tmux!

Let's say you're a typical developer working on a website. You've got one terminal window open for developing[1], another window open for your live rebuild/reload script, and a third open to keep an eye on the server logs. And you're pretty cool so you've got these all in tabs on your terminal app. Something like this:

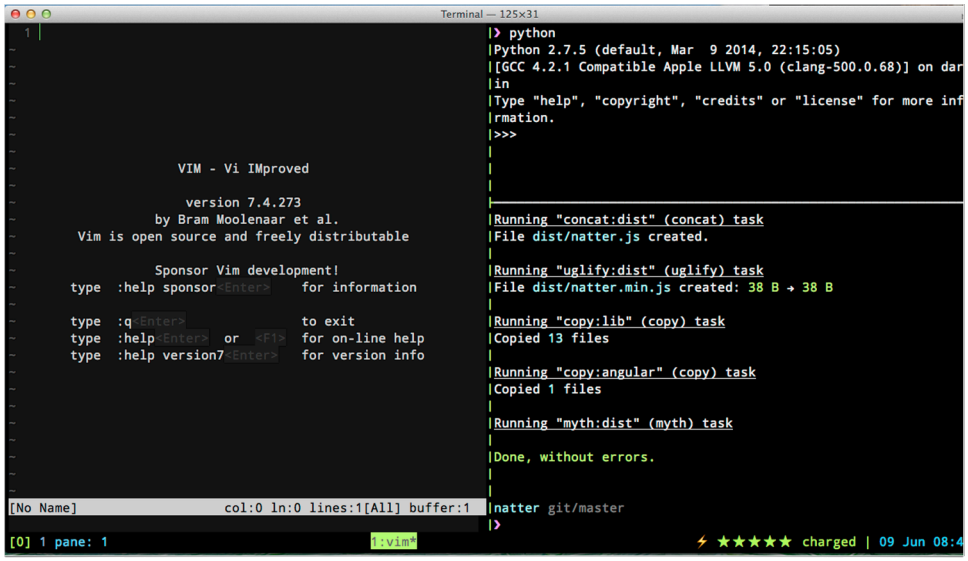


Sure, you could do it that way...

And that's fine. But we can do so much more! Right now you're stuck with an either/or situation: either you can have all your terminals in one window, or you can see them all at the same time. If they're all in one window you can move and resize them easily. If they're all in different windows you can see what's going on in multiple processes at once, but you've got multiple windows to move around if you need to rearrange your workspace.

And that's the first and most obvious problem that tmux solves. Let's get all your windows in a single terminal session, and have them all visible at the same time:

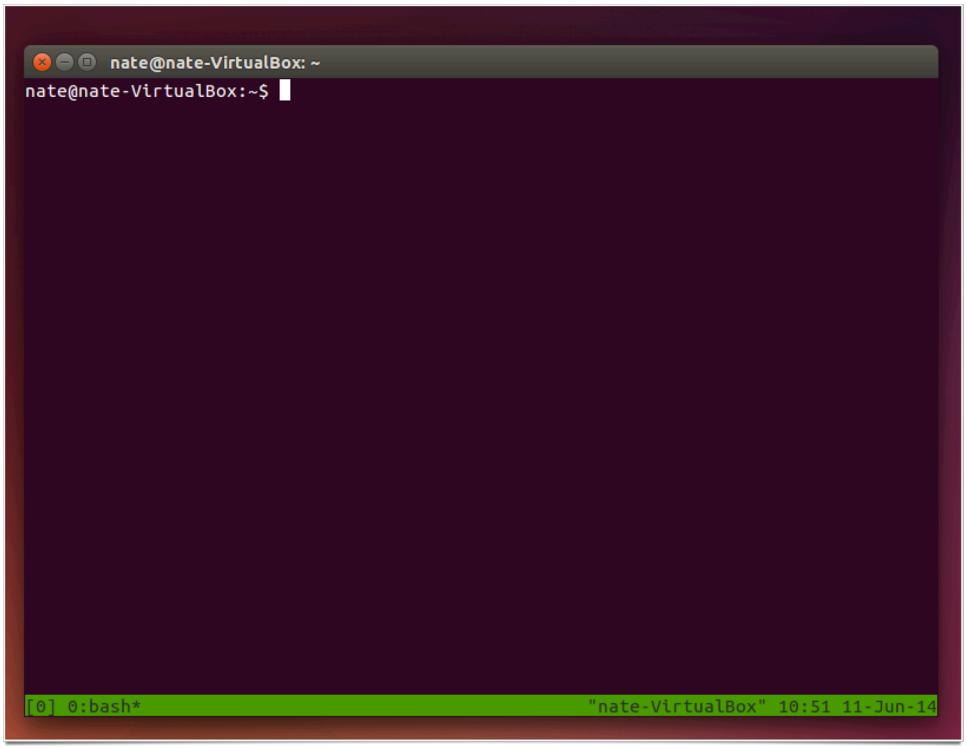
[1] For the purposes of this illustration we're going to take it as read that you're using vim. But if you're using something else I won't judge you.



Or you could do it this way!

But that's just the beginning. Tmux is all about putting your workspace into context, grouping related screens, and making it easy to switch between contexts as needed. For people who work with remote servers, tmux allows you to save your contextual setup between sessions, so when you come back to your server after a few hours (or even a few weeks) everything is where you left it. And the *even better* news is that all this power is *stupidly* simple to use.

Unfortunately, tmux doesn't look *quite* that awesome out of the box. We're going to have to do a few things to it first. A brand new install of tmux looks more like this:



I'm brand new! And a little underwhelming!

The rest of this book is dedicated to getting you from that brand new tmux to the fully customized tmux of your dreams.

2. The Prefix: For Before We Begin

Tmux is designed to let you work fast and efficiently, without getting in your way. To that end, it will ignore anything you type until you hit the `<prefix>` key combination, which by default is `<ctrl>b`.

Vim Says `<leader>`, tmux Says `<prefix>`, Let's Call the Whole Thing "Trigger".

Here's the thing: vim has it's `<leader>` key combo, tmux has its `<prefix>` keys, if you have a KVM switch it's probably got a magic key combination as well. Internally I just call all of them the "trigger" keys. They're the keystrokes that put your keyboard into magic command mode, regardless of what they're called by the application in question. Don't worry if you find yourself thinking "`<leader> x` when thinking about tmux commands. It's okay. Just make sure you hit the right keys.

The default `<prefix>` is kind of a problem. `<ctrl>b` is a weird key combination to hit with either hand. Even if you have already mapped `<Caps Lock>` to be an additional `<ctrl>` key[1], it's still taking your hands away from home row.

So, we're going to jump right into setting up a tmux config file that will make your life better. We'll be talking *much* more about your config file when we get to the Configuration is Fun chapter, but for now we're going to make a couple of small changes to get you started. Tmux looks for configuration commands in a file called `.tmux.conf` in your user directory[2]. Using your favorite editor open this file, or create it if it doesn't exist. We're going to add two lines:

```
set -g prefix C-a
unbind C-b
```

Save that, and start (or re-start) tmux. What we've just done is tell tmux that we want to use `<ctrl>a` for our prefix command, and we want to stop using `<ctrl>b` for anything. If you're using `<caps lock>` as an extra `<ctrl>` key the advantages of this new prefix are obvious: the `a` key is right next to the `<caps lock>` key[3]

Again, we'll get more into the syntax of config file stuff in a bit, but let's take our new tmux config for a test drive.

---

[1] If you haven't done this already, please check out the Appendix on why Caps Lock Must Die.

[2] on OSX this is `/Users/yourUserName` and on linux is probably `/users/yourUserName`, but in either case `cd ~` will get you there.

[3] On QWERTY, Dvorak and Colemak keyboards, at any rate. If you're using an alternate keyboard layout you can use a different key, but you'll want to make sure you're not overwriting an existing command. For now, you can do that using `<ctrl>b,?` from inside tmux, which will give you a list of all currently setup keyboard commands. Press `q` to get back out of this list.

A Tour of the Tmux

Don't Memorize These Keystrokes!

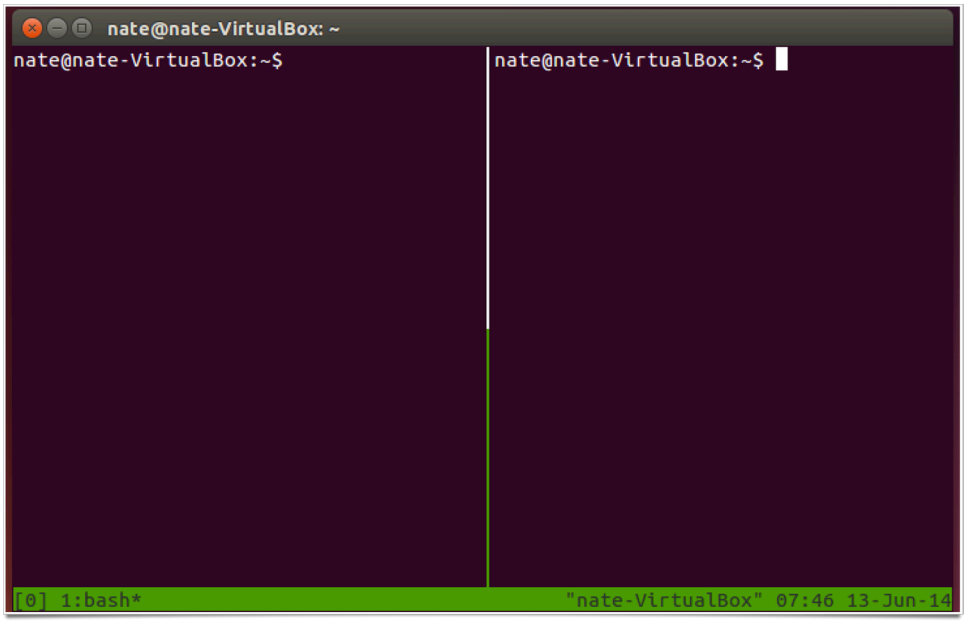
We are going to do a quick tour with a lot of the default commands, but as you'll see they're a bit…weird. In the Configuration chapter we will be changing a *lot* of these commands to more ergonomic versions. So, get used to the *concepts*, but don't memorize the keystrokes. You won't be using them long.

Let's start with a quick dash into and out of tmux. I'm going to take it as read that you've already installed tmux, and you're ready to go.

Start tmux with the `tmux` command in any terminal window. As we saw in the last chapter, the only visual clue that you're in tmux is the bar across the bottom of the screen, which is nice and all, but doesn't actually help us do anything. This is where we start using our `<prefix>` key. For starters, let's split the screen in half. Enter the following command:

`<prefix>` `%`

Your screen should now look a little something like this:



Look! Two Command Lines!

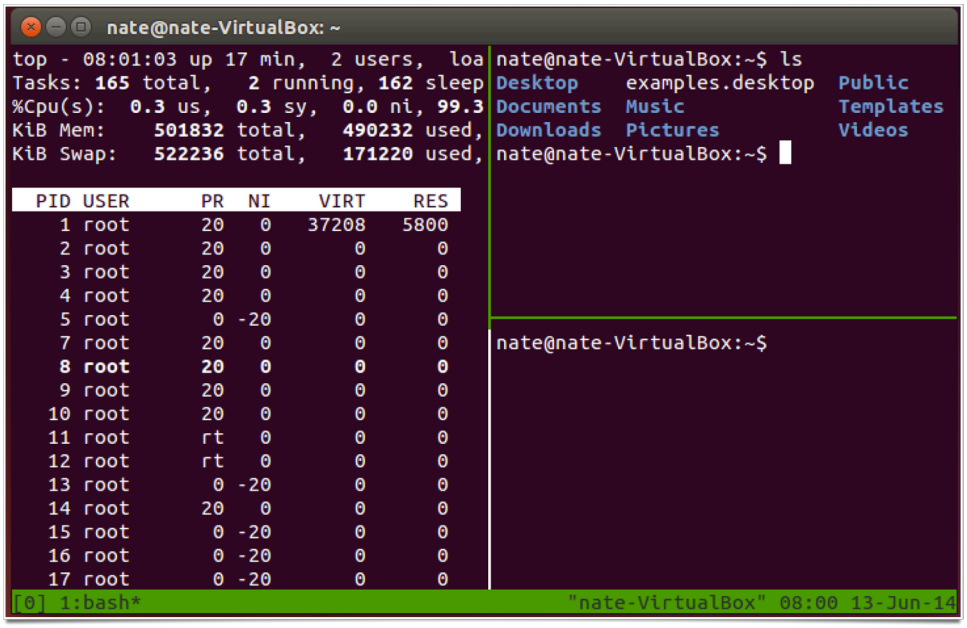
And even better, they are separate processes. Anything going on in the first *pane* is completely independent of anything going in the second pane. But first you need to be able to switch between the two. By default you do this by hitting `<prefix>` and then an arrow button. When you open a

second pane your focus jumps to that pane, so press `<prefix> <left arrow>` to jump to the first pane. Just for fun run the `top` command in this window, then hit `<prefix> <right arrow>` to jump back to the second pane and run something there. Maybe just hit "ls", it doesn't matter. You're on your way to tmux awesomeness.

But you can do more than two panes! While you're in your second pane hit

`<prefix>` `"`

And now you've got this:



Whoa! Three Command Lines!

And we can keep doing this theoretically forever. But that trick is getting old, we want to move on. Let's say you want to leave your tmux *session* for a while. Maybe you're using vim to write fan fiction at work. It doesn't matter. To exit a session simply hit

`<prefix>` `d`

And you're back to the regular command line. The "d" in this case stands for "detach", because that's what you're doing: detaching yourself from tmux. This is a handy trick, especially if you're using tmux on a remote computer because you can detach, close your ssh session, then reconnect to the server and reattach to that session and everything will be right where you left it. But we'll get into that more in the chapter on Sessions. For now we can see what sessions we have running by typing

```
tmux ls
```

At the command line. Unless you've been doing some exploring without me[4] you should only have one session on that list:

```
0:  1 windows (created Fri Jun 13 07:43:25 2014) [80x23]
```

I love this because it gives you all kinds of information you don't really need,[5] but it does tell you the one thing you need to know right at the beginning: the name of the session. In this case it's named "0". Not the most inventive name, but what do you expect from tmux? Let's re-attach to session `0` by entering the command

```
tmux attach -t 0
```

Again, don't worry too much about what that means yet, we'll talk about all this session stuff later. For now you should just see that you're right back in the session we created earlier.

Okay, let's end this chapter by closing down all these panes we created. For good or ill, this is a slightly key press intensive process by default. To kill the pane that currently has focus you press

<prefix> `x`

And tmux will prompt you to make sure you actually want to do this. Press `y` to confirm and that pane will go away. Do this three times and when you kill the last pane you also kill the entire tmux session, and get dumped back out the plain ol' command line.

Confirm the entire session is dead by typing "tmux ls" again. You should get the message

```
Failed to connect to server
```

Meaning "tmux isn't running right now, friend."

Okay, we've done some good work here. Before we dive back in let's define some terms.

[4] I promise I'm not hurt. I'm just so happy to see you spread your wings and fly on your own. You're a beautiful new tmux butterfly!

[5] yet. Some of it will become relevant later on.

3. Interlude: Defining the Terms

There are some words that we use over and over when talking about tmux, to the point that they start to lose their meaning and I have to go for a walk and think about something else for a while. Then I come back to my desk and look up the name for that situation where a word loses all meaning[1]. These words are *tmux*, *Session*, *Window*, and *Pane*.

Obviously, these aren't *complicated* topics, but let's take a few pages to nail down what they mean in the context of tmux, going from the outside in.

Tmux

Okay, this part gets a little technical, but you're a smart person and I know you can handle it.

In order to provide all the functionality you get from it, tmux utilizes a *client/server* pattern. This means that when you type `tmux` for the first time on the command line a lot of things happen very quickly:

- If the tmux program isn't running, it is started.
- A new session is started, containing exactly one window, which contains one pane.
- A new instance of your shell[2] is started for that pane.
- That new session is *attached* to the server, takes over your terminal, and shows you what is displayed in that session.

We will go over what all this means, don't worry. That's what this book is for. For now it's important to remember: tmux is a client/server program. That lets it do cool things. Okay. As you were.

Sessions

When you type `tmux` at the command line, what you are actually doing is creating a brand new tmux *session*. A session is a container for all your tmux goings-on. You can create multiple sessions by detaching from one and starting another, and you can switch between them with a simple keystroke.

---

[1] Semantic Satiation

[2] You probably already know this, but your *shell* is the program that is running inside a terminal window, and it acts a lot like an operating system. It's job is to provide you a command line on which to type commands, and to do things in response to those commands. Popular ones (in alphabetical order) are bash, csh, and zsh. Honorable mention goes to fish, which is an up-and-comer that deserves some attention. If you don't know what shell you're using it's *probably* bash, which has become a de facto standard in recent years. Wikipedia is more (oh so much more) information on what a shell actually is, and, of course, a list of shells you could be using.

From your point of view, a single session should hold everything that relates to a specific task. For example, let's say that you're writing a book in vim, and also working on a new website for your sister. You could start two sessions, one called "Writing" and one called "Website", and switch between them as you move from project to project.

A Session contains one or more *windows*.

Windows

This is the term that leads to the most confusion. When you say something like "open a Terminal window, start tmux in that window and then create a new window" people get sad. So from here on out we're not going to talk about the terminal window. We don't care about the terminal window. The terminal window is dead to us.

The only windows you or I care about are the ones *inside* tmux.

A *window* is everything you can see at one time in tmux. When we split the tmux view into three pieces in the first chapter we had created three panes in a single window. We will get into creating multiple windows soon.

Let's go back to the example of having a Writing session and a Website session. In the writing session you might have two windows: the first would be taken up entirely by your text editor, so you can live distraction free. The second window could be split in half, one side showing your code examples for the book and the other side showing the folder where your book lives. Or something like that.

In your Website session I would have my first window split into two panes: the first, largest pane would have my editor in it. The second, smaller pane would have a command line available for doing quick little things like checking `git status` or running a build script. My second window would have as many panes as needed to show the logs from all the various servers and watchers I have running when I'm making websites.

As you can see, a window contains one or more *panes*.

Panes

A pane is a single command line inside a window. When you create a brand new tmux session you have a single session holding a single window with a single pane. And this, sadly, is how all the people who don't use tmux live *all the time*. Pity them. And then suggest to them that they buy this book. But that is not the point.

Panes are really the heart of tmux. When you create a new pane you actually create a brand new shell with its own local variables and everything. You can have multiple panes on screen at once, all doing different things, completely independent of one another. A chapter all about how to use them to best effect is zooming your way, but for now it's time for us to dive back in and talk about how to configure tmux for fun and profit.

4. Configuration is Fun

So, now that we're all clear on what we call everything in tmux it's time to get around to using it all, right?

Well, no. Because here's the thing. Tmux has a *lot* of less-than-stellar defaults, and I would be doing you a disservice by letting you practice them and then telling you to change them all right when you were starting to master them. So instead I'm going to help you set them all to something more sensible, and then help you get into using them. Along the way we'll get used to tmux's config syntax. So, yeah, it'll look a little backwards, but trust me, this is the way to do it. Cool? Cool. Let's do it.

Syntax

We already reset the biggie: the `<prefix>` key. Let's take a look at how that command is built:

```
set -g prefix C-a
```

`set` is obvious enough: we're trying to set a new keystroke. The `-g` means "global", which also makes sense. In tmux-land, "C" means `<ctrl>`, and `-a` means "also press a". Okay, we've dissected the crap out of that command. Now you know what you're doing, and you should feel powerful.

The other command we used is even easier: `unbind C-b`. This just frees up the default `<prefix>` command, so if you decide you want to use it later you can, but until then if you hit it on accident it won't screw you up. Let's add a few more common-sense defaults.

Counting

The first thing we're going to do is change tmux's style of counting. Tmux is one of those "created by and for programmers" apps that thinks you start counting at zero and go up from there. And when I'm writing code I'm okay with that, but it leads to problems when you're using it as a mnemonic[ˆcantSpell] for your windows or panes, it messes you up, because window `0` will be all the way to the *left*, meanwhile the zero key is all the way to the *right*. So we're going to tell tmux to count like people instead of programmers. Add this to `.tmux.conf`:

```
set -g base-index 1
```

And while we're at it, let's do one more:

```
setw -g pane-base-index 1
```

What's that you say? You've been through the dictionary three times and you've never seen the word `setw`? Tmux uses it as a shortcut for

```
set-window-option
```

I think you'll agree `setw` is much easier to type. In this case we're telling every tmux window to number its child panes starting at one. You're probably not going to be referring to panes by number all *that* often, but we'll set it for consistency.

Binding keys

Tmux tries to stay out of your way as much as possible. In general you can press any key on the keyboard, or even any *set* of keys and tmux will diligently pass those keystrokes on to the program you're working with, unless you hit the `<prefix>` key combination. But we've been over this. Where we're going now is the wild world of what happens *after* you hit the `<prefix>` and what you can make do.

Let's start with a simple one: We're going to be making a lot of edits to our `.tmux.conf` file over the next little while, so let's make easy to reload our config file between changes. Add this line:

```
bind r source-file ~/.tmux.conf \; display "config file loaded."
```

This is a good introduction to one more tmux idiom: When you `bind` a key like `r` all by itself, what you're *actually* binding is `<prefix> r`. In this case we're telling tmux to reload your config file and then tell us that it succeeded. Save your tmux config, close tmux, then re-enter tmux. Okay! From here on out you can use `<prefix> r` to reload and check your changes.

Kill it! Kill it with Fire! I mean, kill it with X!

Okay, now let's make it easy to kill things. By default, if you press `<prefix> x` tmux will ask if you're *sure* you want to kill a pane before it does it. That's nice and all, but what if you'd rather just kill it?

Let's do that right now. The first thing we need to do is unbind "x" from it's current job of "ask and then close" and then rebind it to just "close"

```
unbind x
bind x kill-pane
```

And while we're at it, let's create a key combo for killing the entire session:

```
bind X kill-session
```

Note the capital x, so this keystroke is actually `<prefix> <shift>x` which is still a lot easier than closing every single pane and window.

We're doing good so far. We have our basic commands in place, but we have a little more work to do. For one thing, we have thus far neglected the single most important part of any terminal based application: *colors*.

Once again, we'll fix this by adding a single line to our `.tmux.conf` file:

```
set -g default-terminal "screen-256color"
```

This tells any program running inside tmux that you've got a full 256 colors to play with, so it can hit you with all the glory of a VGA monitor from 1991, command line style.

Okay, two more defaults that we really need to get rid of: Splitting a window horizontally and vertically. These are currently bound to `<prefix> %` and `<prefix> ">`, which are really hard to remember. Let's make them more intuitive:

```
bind | split-window -h
bind - split-window -v
```

The "-h" and "-v" for "horizontal" and "vertical" are somewhat unintuitive, but this tells tmux that if you press `<prefix> |` you'll have two panes side by side, and when you press `<prefix> -` you'll end up with two panes piled on top of one another.

You Can Take the Longcuts, Too.

It's worth noting that all of these shortcuts are just a way to save typing. If you don't like remembering things like `<prefix> |` to split a window you can type `<prefix> :` to bring up tmux's command line, and type `split-window -h` from there. Any command we can bind we can also use directly.

Why Didn't you Unbind the Original?

You can if you want. I leave the original options bound for two reasons: First: I don't have any other use for those weird key commands. Second: every once in a while I have to work on a machine that doesn't have my updated preferences on it and if I occasionally practice using the weird defaults I'll waste less time trying to remember how to make more panes happen.

Nice work! You have successfully changed a bunch of defaults into much more usable forms, and set the stage for the rest of this book. We're going to get into the joys of using panes, windows, and sessions in the next few chapters, and we'll be making more changes to `.tmux.conf` along the way, but you've laid a good groundwork here.

The last thing we're going to do is make it so we can send the `<ctrl>a` command to a program running inside tmux. Tmux tries to be nice, and there are apps that respond to `<ctrl>a`, like vim[1]. If you are running vim in tmux and desperately need to increment a variable you'll be able to type `<ctrl>a,<ctrl>a` and boom! That number is incremented, son. There are probably better reasons for this command, I don't know.

ANYWAY

To pass the prefix keystrokes through to a program inside vim we need to create a keybinding, and then remove the original keybinding:

```
bind C-a send-prefix
```

Coincidentally, this is the reason for the `unbind C-b` thing we did clear back in the Prefix chapter. The prefix key is now `<ctrl>a`, so if you didn't unbind this you could type `<ctrl>a,<ctrl>b` and the program running inside tmux would receive `<ctrl>a` and nobody would be happy. Let's make sure to unbind that.

Handy Chart of the Changes You Just Made

"BaseConfiguration"

Original	Suggested	What it does	`.tmux.conf` line to add
`<ctrl>b`	`<ctrl>a`	`<prefix>`	`set -g prefix C-a`
`<prefix> %`	`<prefix>|`	Split the window into two horizontal panes	`bind | split-window -h`
`<prefix> "`	`<prefix> -`	Split the window into two stacked panes	`bind - split-window -v`
`<prefix> x`	`<prefix> x`	Close a pane	`bind x kill-pane`[2]
N/A	`<prefix> X`	Close the session	`bind X kill-session`

[1] Albeit not very frequently. `<ctrl>a` in vim means "increment the number under the cursor", and it does it fairly intelligently. If you have a line with "456" and you put the cursor on the 4 it will update the number to "457" instead of "556". I have never done this, but hey, if you need it, there you go.

[2] Remember that you'll have to add `unbind x` *Before* entering this command to remove the original binding.

"BaseConfiguration"

Original	Suggested	What it does	`.tmux.conf` line to add
`<prefix>` `<ctrl>b`	`<prefix>` `<ctrl>a`	Sends `<ctrl>a` to an application running inside tmux	'bind C-a send-prefix
`<prefix> :`	N/A	Enter command mode.	N/A

5. Painless Panes

Tmux's panes are the part of tmux that most people will recognize immediately. Look at all those panes! They're everywhere! And we've already gone over some basic commands for using them; you can split them horizontally, split them vertically, kill them, what more do you need?

Well, we're about to find out. There's quite a bit you can do with panes to make your terminal much more responsive to your needs.

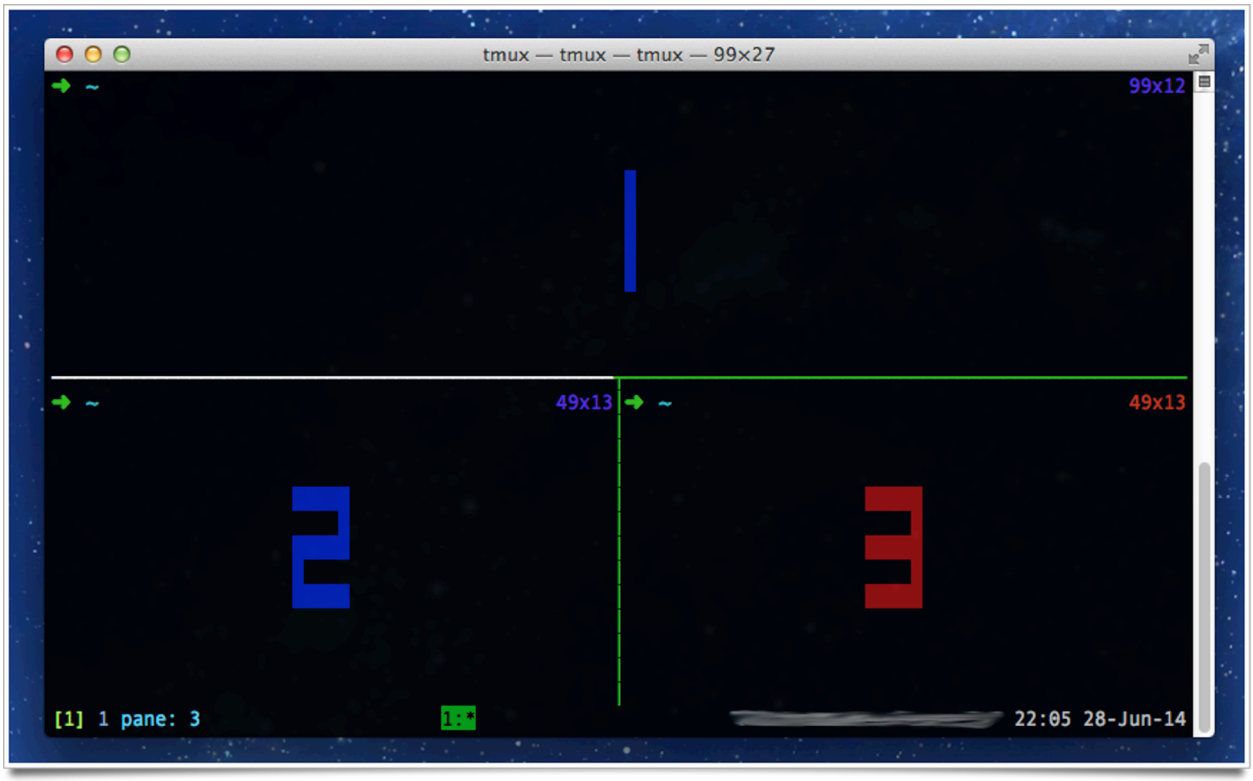
Moving

Let's start with the simplest stuff first. Moving panes around. Say you've set up a tmux window with the following commands:

Split Some Panes!

```
tmux
<prefix>  -
<prefix>  |
```

This should leave you with a layout like the following:



*You have **no idea** how difficult it was to get this screenshot.*

Hey, Cool! Giant Numbers in Every Pane!

Pressing `<prefix> q` will make those appear for about a second. Which is a short amount of time in which to press the oddly tangled OSX "snap a screenshot" keystroke.

Now let's say that you want to keep this general layout: big window on top with two little windows underneath. But you want the all the stuff from pane 2 in the big window, with 1 and 3 down below[1]. You can pull this off by rotating the panes using `<prefix> <ctrl>o` which I think is the most complex key command we've seen so far. This will rotate each pane clockwise.

If you don't want to rotate the entire thing you've got commands for that as well. `<prefix> {` or `<prefix> }` swap your current pane with the *previous* and *next* pane, respectively. The pane numbers stay where they are, but the contents move around.

A more common scenario is that you created a bunch of panes, but they're all the same size and you can't focus on the important one. Sure, you could just lay them all out manually, but that's no fun.

Fortunately tmux has a set of *default layouts* that you can apply quickly. For example, a very common layout is to have one "main" pane where you're doing all your work and a second pane that's monitoring output or whatever. The default layouts are:

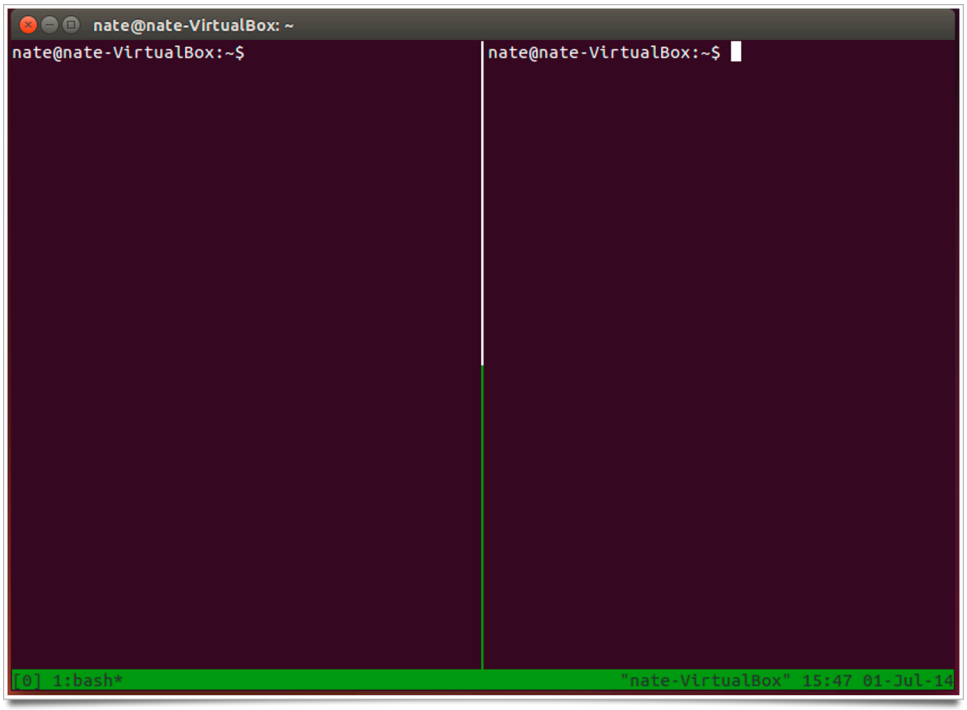
[1] I don't know why either. In real life I have done this maybe five or six times.

"DefaultLayouts"

Keystroke	What it does
`<prefix> <alt>1`	Switch to Even Horizontal Layout
`<prefix> <alt>2`	Switch to Even Vertical Layout
`<prefix> <alt>3`	Switch to Main Vertical Layout
`<prefix> <alt>4`	Switch to Main Horizontal Layout
`<prefix> <alt>5`	Switch to Tiled Layout
`<prefix> <space>`	Rotate through the default layouts

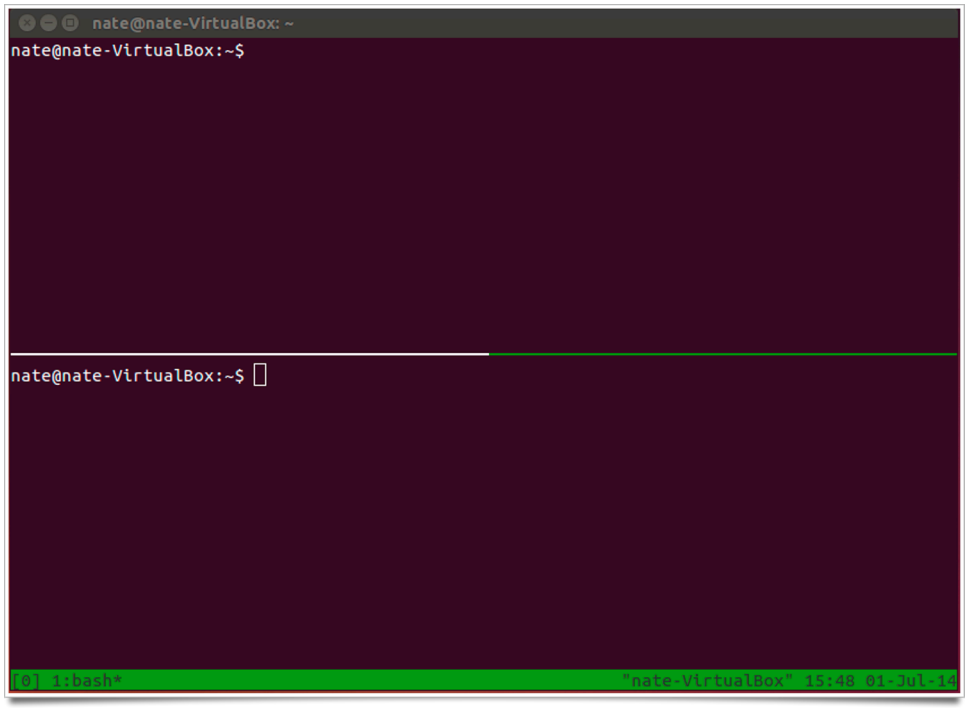
These are all basically what they sound like:

Equality for all:



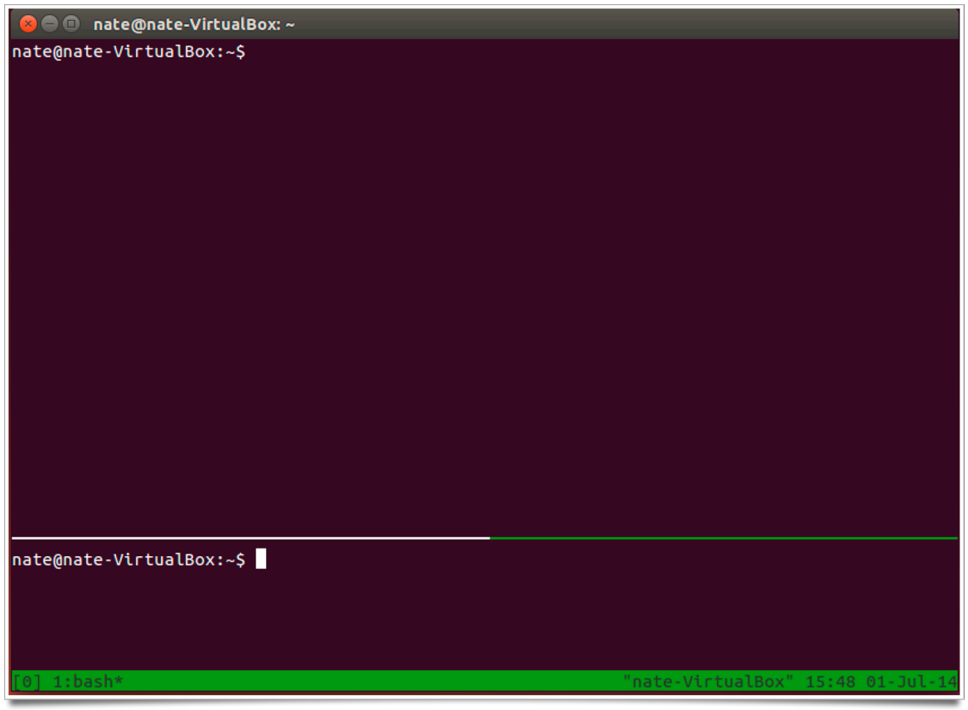
Even Horizontal

Equality in the other direction:



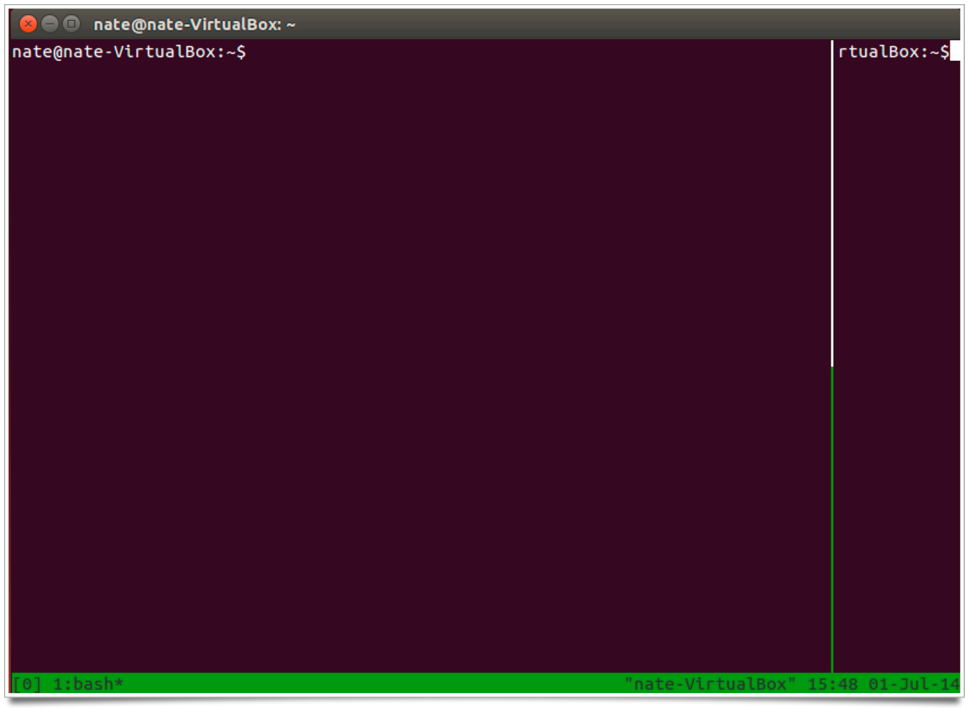
Even Vertical

This often gets used with an editor on top and a server monitor underneath:



Main Vertical

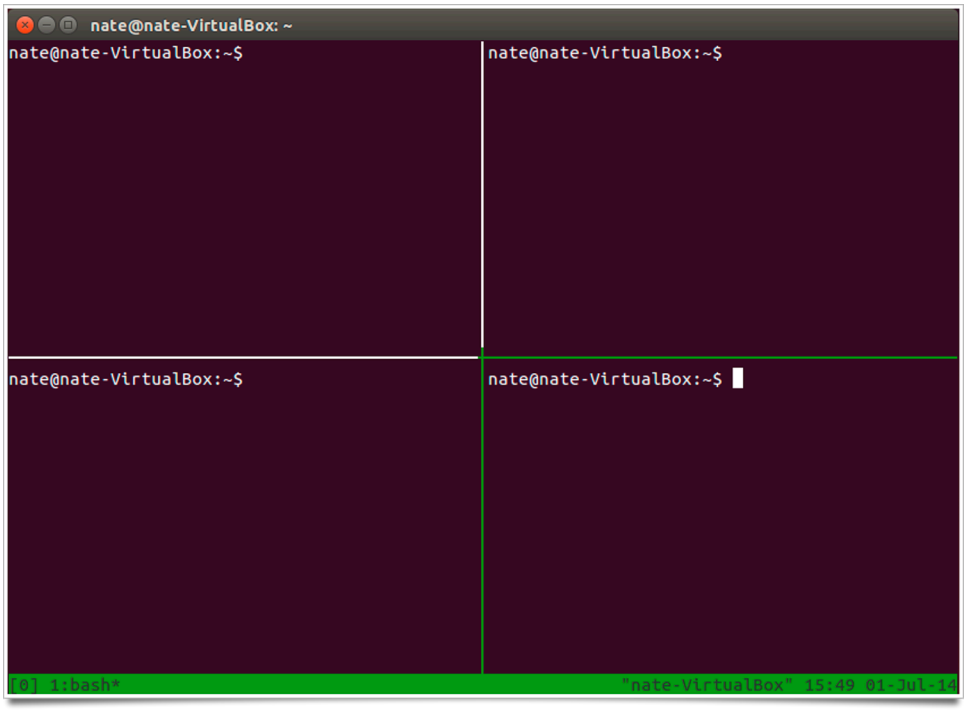
I've seen this one with an awesome command line chat client in the small window:



Main Horizontal

Brady Bunch[2] style:

[2]Like the super-timely reference to a TV show that was cancelled even before *I* was born?



Tiled

Hit `<prefix> <space>` to rotate through these standard layouts. While you're at it, create a bunch of extra windows, see how each default layout looks with a bunch of windows. Then maybe start killing windows off with `<prefix> x` and keep playing until you've got a handle on it. Are we good? Okay. Then let's move on to customizing your layouts.

Resizing

So, as awesome as the default layouts are, you can do more. You can make new layouts, make them fit exactly the way you want them. The default tmux shortcuts for resizing windows are based on the arrow keys:

"ResizingPanes"

Keystroke	What it does
`<prefix> <ctrl>up`	increase the height of this pane by one row
`<prefix> <ctrl>down`	decrease the height of this pane by one row
`<prefix> <ctrl>left`	increase the width of this pane by one column
`<prefix> <ctrl>right`	decrease the width of this pane by one column
`<prefix> <alt>up`	increase the height of this pane by five rows
`<prefix> <alt>down`	decrease the height of this pane by five rows
`<prefix> <alt>left`	increase the width of this pane by five columns
`<prefix> <alt>right`	decrease the width of this pane by five columns

So, the left and right functions are actually a little more subtle: if your pane is on the *right* side of the window you the right commands will shrink the window, if it's on the left side it will grow it a little. If you're familiar with how vim windows work it's quite similar. But we can make it even **more** similar.

Resizing, Vim Style

One of my main reasons for using tmux is because it's keyboard centric and plays well with vim, my other favorite keyboard centric tool. And after spending (wasting) all that time getting used to moving things around using vim's default key strokes, it made sense to me to be able to resize tmux windows using the vim key bindings. This is how I would suggest doing just that:

"ResizingPanes

Original	Suggested	What it does	`.tmux.conf` line to add
`<prefix> <ctrl>up`	`<prefix> K`	increase the height of this pane by one row	`bind-key -r K resize-pane -U`
`<prefix> <ctrl>down`	`<prefix> J`	decrease the height of this pane by one row	`bind-key -r J resize-pane -D`
`<prefix> <ctrl>left`	`<prefix> H`	increase the width of this pane by one column	`bind-key -r H resize-pane -L`
`<prefix> <ctrl>right`	`<prefix> L`	decrease the width of this pane by one column	`bind-key -r L resize-pane -R`

Now, even if you decide not to use these keystrokes, there's a few things you can learn from this. The first is the `bind-key -r` syntax. The `-r` makes the command *repeatable*, meaning that if you are somewhat quick you can press the prefix key once, and then push the letter key multiple times to repeat the command. It will time out after a short period, but this is handy for quickly changing the size of the pane you're playing with.

The second thing is that this kind of doesn't save you any keystrokes. You're still pressing four keys to make this happen. The difference is that all the keys you're pressing are right where your hands should be resting naturally, instead of having to move clear over to the arrow keys.

And there's one more thing. You'll notice that, with the arrow key commands, you can either resize by a single space with `<prefix> <ctrl>` and a direction, or with *five* spaces if you hit `<prefix> <alt>` and a direction. If you decide that you want your vim-style keys to change the size of the windows by five spaces (or any other number of your choosing) you can accomplish this by adding a *count* to the end of the line in `.tmux.conf`:

```
bind-key -r K resize-pane -U 10
```

Will resize your window by ten spaces. Play around with the numbers in `.tmux.conf` and decide what works for you. I go back and forth between 3 and 5 spaces.

Zooming

Sometimes, however, you don't want to mess with resizing all the panes on screen, you just want to focus on *one pane* and ignore all the others for a while. Pressing `<prefix> z` will *zoom* your current pane, filling the entire window. To let you know it's done this, tmux will put the letter `z` down by the name of the window, letting you know that you don't have a window with a *single* pane, you have a window with a *zoomed* pane. Pressing `<prefix> z` again will return the pane to its regularly scheduled shape and size.

This is often a very handy "distraction free" writing technique. When you need all your supporting panes you can get them back easily, but when you need to force yourself to (say) finish a chapter about tmux panes you can block everything out, fire up vim, and just *write*.

Oh yeah, I guess it works well for coding as well.

Breaking Window Panes (no, it's okay!)

The time will come when you realize you have too many panes in a single window. You will realize that you don't need to monitor all seven servers in a single window; you can spread those puppies out a little bit. That's what the "break" command is for. This one is super simple: Just hit `<prefix> !` [3] and whatever pane you're currently in will be moved it a brand new window of its very own. From there you're free to split it up in to as many pieces as you want.

Using Panes

In real world practice, I've found that three panes is about the limit of what I can use effectively. I will usually have one pane with an editor that takes up most of the window, a second pane that is parked at a command prompt so I can run any one-off commands I need to, like doing special compiles or playing with git. If I use a third pane it's often running some long term process, like a watcher that re-compiles my code when it detects changes, or a server that lets me monitor requests and responses. More than that and the panes tend to be too small to be of any real informative value. You can always zoom in on a pane if you need to see what it says, of course. If you find yourself zooming one pane frequently you might consider just breaking it out into its own window. As we are about to see, switching windows is exactly as easy as switching panes.

[3] I like to imagine the exclamation point is meant to represent the sound of breaking glass.

6. Tmux Windows Work a Lot Like Tabs

One step larger than panes are windows. We've seen quite a bit of them in the previous chapters, and we've seen how they interact with panes as well. In this chapter let's take a look at how to get the most out of windows[1].

Windows are for Contexts

When you're working in software there are a lot of moving parts that all need their own personal space. You'll usually want to have an editor up and running (or more than one), a number of servers that are all reporting their status back to you, and just for the fun of it, let's throw a file browser in the mix as well. The thing is, you don't need to see all of these things all the time. So we put them into separate windows. I tend to think of each window supporting a particular task or context, and I only put things that are related to that task in that window. Tmux, it just so happens, supports this idea with window names.

By default, tmux will label each window with whatever command last ran in that window. So if you have a window with three panes and you run a ruby script in pane two, Tmux will display "ruby" down in the status bar for that window. This is similar to what many terminal emulators do. But it's not really all that helpful. For example, with this default behavior, if you run the OSX package manager homebrew[2] you'll see the name flash wildly between ruby, bash, clang, llvm, make, and any number of other programs as homebrew does its magic and installs all that's best and brightest in open source software on your machine. This can be entertaining, sure, but it's not all that *useful*. So let's give the window a specific name, that explains what it's doing. Do this by hitting `<prefix>` `,` (that's "Prefix, then comma"). You'll get a notification bar across the bottom of the screen saying "rename-window:" which should give you a pretty good idea of what to do next. Type a name for that window and hit `<enter>` and your window will keep its new name even if you run commands that would normally change the window name.

[1] You have *no idea* how hard I am working to keep the Microsoft jokes in check right now.

[2] http://brew.sh/

Useful Windows for Coders, Useful Windows for Writers

I typically have three windows: Editor, Servers, and Scratch. What's interesting is that after quite a bit of trial and error, I find that I use this same setup of three windows whether I'm writing code or writing text. The Editor window has vim and a command prompt, the servers window automatically starts up any relevant servers (we'll get there soon), and the Scratch window is just a single pane with a command prompt.

When I'm writing code, the Servers window will have my watch program that updates whenever code changes as well as the output from node/apache/nginix/ and MySQL/redis/MongoDB, depending on the project. When I'm writing blogs, the Servers window will have the watch server as well, and will be where I will run commands to upload my posts when they're finished.

So, once you have multiple windows, you need to be able to move between them. Time for a table!

"SwitchingWindows"

Keystroke	What it does
`<prefix> 0-9`	Switch to window 0-9
`<prefix> w`	Choose a window from a menu
`<prefix> c`	Create a new window
`<prefix> ,`	Rename current window
`<prefix> n`	Cycle to next window
`<prefix> p`	Cycle to previous window

In general the `<prefix> #` commands will be the ones you use the most often. The window numbers are displayed by the window name, and it's easy to just hit the number for the window you want. If you have more than ten windows the `<prefix> w` option gives you a nice view of all your windows, and you can arrow up and down through the list to the one you want, then hit `<enter>` to go there. Again, easy.

Monitoring Panes

So here's the situation: you're a tmux master and you're using tmux to track ten different servers, all doing their own thing. You go to lunch and when you get back you have to see what's changed while you were gone. Sure, you could go through every single window and check for status updates, or you could let tmux take the guesswork out of it for you and just show you the windows where something has changed. The monitoring commands are `<prefix> <alt>p` for the previous window with activity and `<prefix> <alt>n` for the next window with activity. This way, if you've only got changes in windows 5, 7 and 10 you can just jump to those windows and see what's new. Granted,

this isn't a situation that comes up all that often for solo developers, but is quite useful for network and system admins.

7. Sessions

At first glance, it doesn't look like there's all that much to say about sessions. You create one, full it with windows and fill those with panes…and that's about it. But sessions are the cornerstone of tmux's power features, and for some people they're the reason they got into tmux in the first place.

A tmux session's main strength is its *persistence*. When you create a new session it will keep running until you either restart the machine or remove all the panes and windows from the session. This becomes especially important when you're running tmux on a remote machine, but we'll get into that more in the Remote Tmux chapter. For now let's look at how to make sessions work for you locally.

Sessions are for Projects

Earlier I said that a window is a way to logically group tasks by *context*. In that vein, a session is a way to group *contexts* by *project*. I would strongly suggest you keep each session tied to a specific long-term activity, and don't worry about "wasting" a session by only having one window open in that session. Tmux sessions are not that heavy in the grand scheme of things, and if you only have one thing going on in a particular project (like, say, hosting a Minecraft server or similar) then put it in a session all to itself. It'll be okay.

For most projects, of course, you'll have several windows full of contextual information, and it will be a pain to set them up every time you want to start work. In later chapters we'll get into ways to set up your session quickly and efficiently, ready at a moment's notice to work the way you want it to.

Session Setup

Whenever you start tmux, you're starting a new session. Here's how easy it is: type `tmux` at the command line and BOOM, new session, right there. If you close the last pane in a session the entire session closes as well. You can tell if you have any sessions running by typing `tmux ls` on the command line.

Create Aliases for Common Tmux Tasks

There are a number of command-line tmux utilities that can be invoked outside of a running tmux session. Many of them are long and convoluted, but you can easily shorten them. If you're interested in doing so I would suggest you check out the Useful Aliases appendix. Go ahead. I'll wait.

Tmux is not super imaginative in session naming, however. If you've followed along up to this point your first session will be named "1". This is less than helpful. To change the name of a session you use the `<leader> $` command. This will open a prompt asking you for a new name for the session. Give it something more useful, like the name of the project you're working on in this session.

Naming.convention

I generally give my sessions names that start with a capital letter and my windows names that start with a lowercase letter, but that's not super important. It makes things look right to my programmer's brain when I'm creating automated layouts and the commands are in the form of `Session.window` but we'll get to that in a bit.

Detaching and Attaching

As I mentioned, a session will persist as long as the computer has power and it has at least one pane open. But just because the session exists doesn't mean you have to stay connected to it. In fact, we played with detaching and attaching to sessions in the Prefix chapter, but you can be forgiven if you don't remember the arcane commands we used back then.

Fortunately the detach command is anything but arcane. Just hit `<prefix> d` to be detached from your tmux session and dumped back out to the non-tmux command prompt. From here you could create a new tmux session, or even do command line stuff, if you've got a program that doesn't play nicely with tmux[1]. You can also see a list of all the tmux sessions you have open by typing `tmux ls`[2].

When you want to attach to a session you need to tell tmux *which* session it should attach to. The command for this is `tmux attach -t sessionName`[3].

These three commands make it easy to detach from a session, find out what sessions are active, and attach to one of them. But you can switch between sessions without jumping out to the non-tmux shell.

Swapping Sessions

Once you've created a couple of sessions you might want to flip back and forth between them as you work on different projects. The good news here is that the tmux server controls all the sessions and can attach you to any of them at a keystroke. There are a few ways you can do this:

[1] This is a definitely not-painless topic, but the gist of it is that there are a few programs that depend on the state of the shell in which they were opened, but provide global functionality. OSX's `launchctl` is famously one of them. If you are doing anything that uses `launchctl` I would strongly recommend getting out of tmux beforehand.

[2] or `tls` if you're using the recommended aliases from the appendix.

[3] or `tat sessionName` if you're using aliases.

- If you know all the sessions you have open you can use `<prefix> (` and `<prefix> )` to cycle to the previous and next session, respectively.
- If you want to pick from a list of all open session you can hit `<prefix> s` and arrow up and down through the list of available open sessions.

Let's throw these in a table to make them easy to remember.

Session Commands

Keystroke	What it does
`<prefix> (`	Switch to previous session
`<prefix> )`	Switch to next session
`<prefix> s`	Choose from a list of sessions
`<prefix> $`	Rename the session
`<prefix> d`	Detach from your current session

The Why of Sessions

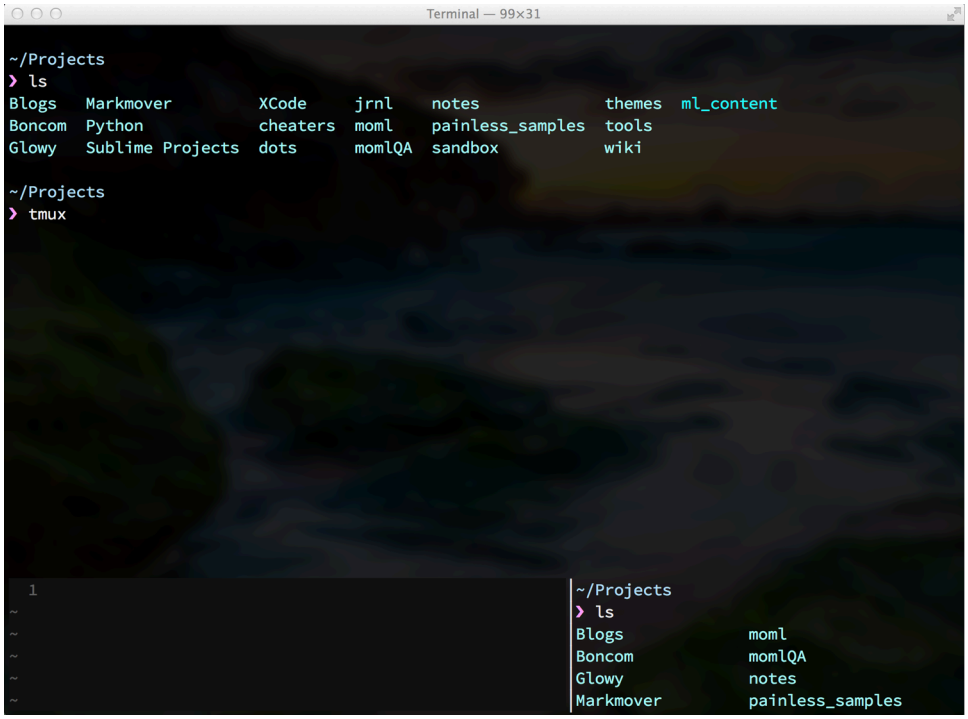
The value of persistent sessions is in keeping your workspace constant. Whenever you attach to a session it's laid out just as it was when you last detached, which is an obvious win for people who are working on servers or other machines that stay up and running all the time. We will get into that more in the Remote Tmux chapter, but for now just keep that in your mind. You can start a server program in tmux, detach from the session and use your terminal to do anything else you want to do for a while, then come back to that session and pick up right where you left off.

This is less of a draw on a laptop, where you're frequently shutting the machine all the way down, because tmux loses its sessions when the computer loses power. However, we will mitigate against that in the Automation chapter.

So don't be discouraged if this chapter feels a little sparse. We are doing the groundwork, and you have to admit that using multiple sessions is helpful on a day-to-day basis. Just remember that it will be in the later chapters are where sessions truly shine.

8. Copy Mode

Before we get into the meat of this chapter, let's try a little experiment: in a terminal window where tmux is running grab the mouse and scroll up. Wooo! We're suddenly off the edge of the world, back into the commands you ran before you started running tmux.



Pre-tmux on top, tmux down below.

Tmux takes over the command line screen using a magic little program called "ncurses" that lets it re-draw the terminal window over and over instead of line-by-line, the way terminals were *meant* to work[1]. Which means that the regular terminal scroll-back functionality is broken. But tmux would never leave you stranded without a good way to scroll back to the content you need and want.

Each pane in your tmux session keeps its own history, complete with all the commands you've entered and all the responses the system has issued. You probably use this history all the time: let's say you're trying to remember how to issue some `tar` command and you keep getting the switches wrong. You try it once, read the error message, press the up arrow to re-populate the command line and change a few things, try it again, and so on until you finally admit defeat and us a GUI tool as nature intended. Tar is not meant for human use.

[1]This is the same underlying technology used by vim and top and pretty much any other program that hijacks your screen to show you a persistent view instead of a you-type-something-I-show-something view. We're really really not going to worry about how it works. It's probably voodoo, but it's been doing it for so well so long we're going to leave it alone.

But Tmux is. Tmux gives us access not only to all our previous commands, but also to all the output from those commands. From copy mode we can copy and paste all this text goodness between buffers, or even sessions. Granted, most of the time you won't be doing that, you just want to scroll back over error outputs, but *someday* you'll want to copy something from a pane's buffer and then you'll thank me. For now, though, let's start out just scrolling back over your command history.

Entering Copy Mode

Press `<prefix> [` to enter copy mode. You won't notice a huge difference, except there is suddenly a little box in the corner of your pane telling you what line you're on, in the least intuitive way possible. We'll get to that. Now you can move your cursor around with the arrow keys, scrolling back through your history in that pane. If you press `<enter>` you're back out of copy mode, your cursor is back on the current line, no harm done. So far we've done just enough to demonstrate the copy mode exists and you can use it to see things you've already done. But we can do better. For starters, we can justify the name "copy mode" really quick.

Seriously, What is That Box Telling Me In Copy Mode?

It's telling you what command you're on in the history, with the most recent command as "0" and on back through the log. If you have a relatively young pane it may only have a few commands in it, so it'll say something like "0/6", and as you scroll back it will change to "1/6", "2/6" and so on. It's an admittedly odd way to keep your place in a buffer, but there you go.

Get back into copy mode by hitting `<prefix> [` again and then scroll through your output for a while. When you find some fascinating text hit `<space>` and then start moving around with the arrow keys again. You'll see that you're now highlighting text, and who doesn't love that? When you press `<enter>` to leave copy mode this time the text that is highlighted will be copied to tmux own internal clipboard[2] and you can then type `<prefix> ]` to paste that text anywhere else in your tmux session.

A Stack of Snips

When you copy text in copy mode it's placed into a copy buffer for you to retrieve at your leisure. But there's more! Tmux places that copy buffer into a *stack* of buffers, meaning it's a "last-in-first-out" arrangement. Let's say you copy the text "Everything is amazing" from your history. You now have

[2] Yes, you can integrate this clipboard with the system clipboard, but it's a pain, and its a quite different pain depending on your OS. I will point you to some resources in due course, but for now, know that you can detach from tmux (without quitting it) and type `tmux show-buffer` at the command line and whatever text you copied will be available right there. While I have indeed set up tmux to share its copy buffer with OSX I find that I have never actually *used* that integration, so it feels like a less-than-painless topic to cover.

a single buffer with that string in it. You can see this by typing `<leader> : show-buffer<enter>`. Now let's say you copy the announcement "I love it all!" and run `:show-buffer` again. You'll see the second string (I love it all!) and the first will appear to be gone and you'll call me a liar. But I'm not, I promise I'm not. To prove it, type `<leader> :choose-buffer<enter>` and see what happens: you get a list of all the current buffers, and you can choose which one you want to paste by highlighting it with the arrow keys and pressing `<enter>` to paste your selected buffer back to the current command line in your current pane.

Navigating in Copy Mode

Painlessly Turning Tmux into Vim

I've said it before, and I'll say it again: I'm a bit biased towards vim. And this bias is in full effect in this section, as we learn how to use vim's movement keys in tmux's copy mode. But vim's movement commands are intuitive and a natural way to move around a body of text. If you're a vim user, this section is a no-brainer, you should absolutely do it. If you don't use vim I would still consider it. You've probably run into enough programs that use the vim navigation keys to make them at least a little familiar, and if they aren't yet, then this might not be a bad time to get used to them. If you don't want to use the vim style keys you can skip to the Awesome ASCII Screenshots section, which is kinda fun.

Let's face it, moving around a screen with the arrow keys, one space at a time, is a drag. It's an especially huge drag if you're scrolling back through miles of errors trying to find a specific line where the program tells you what actual file you should look at to find the offending code that broke your system so magnificently. To alleviate this burden, tmux makes it easy to adopt pretty much all of vim's motion keys, including searches, by adding one line to your `.tmux.conf` file:

```
setw -g mode-keys vi
```

This is not the most intuitive syntax ever, but you get the point: we want to enable a set of "mode keys" that emulate "vi" behavior. Once you've done this copy mode becomes a lot more navigable, and it enables vi(m) style searching as well. Here's a quick refresher on the most useful vim style navigation commands for tmux:

Keystroke	What it does
h j k l	Move one space left, up, down, right, respectively
`<ctrl>b`	Scroll up one "page"
`<ctrl>f`	Scroll down one "page"
g	Jump to the top of the buffer
G	Jump to the end of the buffer
w	Jump to the beginning of the next word
b	Jump back one word
f*n*	Jump forward to the next occurrence of *n* on that line
F*n*	Jump backward to the previous occurrence of *n* on that line
`?*search*<enter>`	Search backwards through the buffer for *search*
`/*search*<enter>`	Search forwards through the buffer for *search*
n	Repeat the last `/` or `?` search in the same direction
N	Repeat the last `/` or `?` search in the opposite direction

Of particular note are the `?` and `/` searches, which will save you a lot of headaches. Let's say you're trying to find the name of the file that caused your compiler to explode so impressively. You're pretty sure It's a `.cpp` file, and you just want to scan back through all the references to `.cpp` files in the output. You would type

```
? .cpp
```

And hit enter. Tmux would then highlight the closest occurrence to your cursor, starting from your cursor and going up, which is to say, backwards in the buffer. If that's not hit hit `n` to jump to the next occurrence, and again, and again…until you find it.

The `/` search works exactly the same way, but backwards. Most of the time in tmux copy mode you'll be starting with your cursor at the end of the buffer, so you'll use the `?` search to roll backwards through the file. But if you've already scrolled up to the top of the buffer and want to search your way back down then `/` is there to help you out.

Awesome ASCII Screenshots

This is a feature that rarely comes in handy, but when it does it's *very* handy indeed. Tmux has the ability to take a snapshot of everything you can currently see in a pane and copy that all to a clip buffer. Now, obviously, you're not going to want to paste an entire pane's worth of text at a command line, but you might want to save it to a text file that you can send to someone else. You can accomplish this with a pair of commands: `capture-pane` and `save-buffer`

By default, neither of these are mapped to a specific keystroke, most likely because they're not seen as high-priority targets. This means that we have to enter command mode to perform these commands. Remember that the shortcut for command mode is `<prefix> :` and you'll know you're in command mode when your cursor jumps to the command bar at the bottom of the screen.

Let's play with capturing a pane and saving it out, and while we're at it let's create a shortcut that lets us do this automatically.

The slowest way to do this is to take four steps:

1. Enter command mode
2. Type `capture-pane<enter>` to capture the pane.
3. Enter command mode again
4. Type `save-buffer ~/buffer.txt<enter>` to save the buffer out to a file

We can do this in two steps:

1. Enter command mode
2. Type `capture-pane; save-buffer ~/buffer.txt<enter>`

Or we could do all of this in a single step by adding a new shortcut to our .tmux.conf file. When you're running multiple commands from a single shortcut you should use tmux's shell syntax. You see, you can call pretty much every tmux command from the *regular* command line by passing that command as an argument to the tmux program. So if you wanted to run "show-buffer" without entering command mode you could type `tmux show-buffer` at your regular prompt and it would behave exactly the same as if you had typed `<prefix> :` and then entered the `show-buffer` command at the tmux command prompt. An additional benefit to the shell syntax is that it's accessible from outside of tmux. Try it now!

1. In a tmux pane start the `top` process so you have a nice solid block o' text.
2. Type `<prefix> : capture-pane<enter>`
3. Type `<prefix> d`
4. From the command line type `tmux show-buffer<enter>`

And you'll get a static representation of what was on `top` when you took the snapshot all those seconds ago.

So, all of this is prelude to our command we're going create in our `.tmux.conf` file. Here's the line to add:

```
bind C-s run "tmux capture-pane && save-buffer ~/buffer.txt"
```

The syntax we used here binds `<prefix> <ctrl>s` to our new command. The extra `<ctrl>` keeps this rarely-used command from taking up a spot that could be used by something more, you know, *useful*, and ensures that when we tell tmux to save a screen shot we *mean* it.

As an aside, we will be seeing a lot more of the shell syntax in the chapter on Custom Sessions, but for now just remember that just about anything you can do from command mode you can also do from the command line.

Keystroke	What it does
`<prefix> [`	Enter Copy Mode
`<prefix> ]`	Paste current buffer
`<prefix> =`	List all buffers and choose one from which to paste
`<space>`	(in copy mode) Start selection
`<enter>`	(in copy mode) Exit copy mode, copy selection to buffer

Original	Suggested	What it does	`.tmux.conf` line to add
N/A	`<prefix> <ctrl>c`	Choose a buffer and paste the contents	bind C-c choose-buffer
N/A	`<prefix> <ctrl>s`	Save a copy of the current pane to `~/buffer.txt`	bind C-s capture-pane; save-buffer ~/buffer.txt

9. Interlude: Finesse

Okay, you've done some good work with getting used to how tmux does its stuff, and you should be feeling pretty good about using tmux on a daily basis. So I'm going to take a moment and talk about how to give your tmux setup a little bit of sparkle and shine, how to keep things looking clean and easy to update. Finesse is all about doing things well and making it look easy. This is going to come in handy in the next chapter particularly, but let's be honest, it's important all the time.

Finesse is about doing things *right*, not just *good enough*. When you do things right, you stop worrying that your setup is going to go *poink*[1] suddenly because you didn't actually know what you were doing when you put it together. You're confident, and if you have to fix something every once in a while you can do so without worry, because you know where and what everything is.

But too often configuration files get treated as if "good enough" was, well, *good enough*. I mean, if it gets the job done, right?

Well, no. Sorry. Because while your computer doesn't care how your config files look, you really should. When you want to add something to your config you should know where to look to make sure you're not overriding something you already set up. So, let's start by adding some finesse to your `.tmux.conf` file.

Comments are Free, and Priceless

Computers don't mind skipping over comments, and will happily ignore as many lines as you want them to ignore in your setup files. Humans, on the other hand, are forgetful and often judgmental of anything they don't remember writing themselves, and will therefore delete things if they don't understand them. To stop your future self from deleting a bunch of lines out of your configuration file, make it explicitly obvious what each section is doing and why. In my mind a "section" of `.tmux.conf` consists of all the commands that apply to a certain type of operation. So all the entries that change how copy mode work would go into a section. All the lines where you define how to split windows and panes go into a section. All the lines where you change your prefix settings go into a section. And you should start each section with a comment, saying exactly what that section is doing. A sample `.tmux.conf` file might look a little something like this:

[1] That's the internationally recognized sound of something breaking without warning because the person who built it didn't take the time to do it right.

```
# prefix settings
set -g prefix C-a
bind C-a send-prefix
unbind C-b
unbind x

# Color Handling
set -g default-terminal "screen-256color"

# Base index and escape time
set -g base-index 1
setw -g pane-base-index 1
set -s escape-time 1
```

This way it's obvious what everything is doing and why.

Check it in

Okay, thing two: Make sure your `.tmux.conf` file is checked into a source code control repository of some sort. GitHub famously Loves ~/[2], and is an excellent place to check all your "dot files" in, ensuring that you don't lose them, and that you can easily set up a new machine to use the settings you have so painstakingly crafted to fit your way of working. Heck, you can check out my dot files on GitHub[3] and see what I'm doing right now. If you don't want your dot files out where everyone can see them you can check them into BitBucket[4] and keep them private. It doesn't matter what service you're using, as long as your config files are backed up somewhere catastrophe-resistant.

Finesse Starts With You

So, if you do all this will tmux start up faster and run smoother? No and maybe, in that order. If your config file is clean and tidy you're less likely to tell tmux to map the same key to two different functions, or map it and then unbind it two lines later. But these tips will ensure that your tmux configuration files will be legible and useable under just about any circumstances, and will help you sleep easy and work quickly. And as we get into the fun stuff involved with customizing what tmux looks like it's important to have a clean, well organized config file at the outset. So take a few minutes now and organize your config file, and check it into source control before moving on to the next chapter. You'll be glad you did.

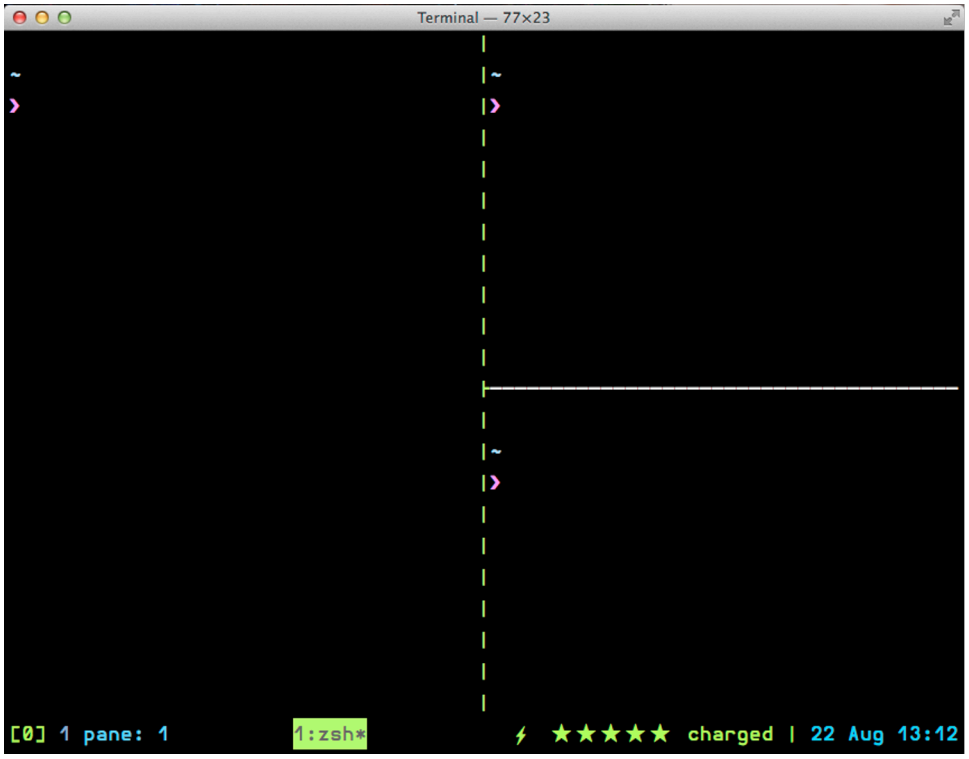
[2] http://dotfiles.github.io
[3] https://github.com/poginate/dots
[4] https://bitbucket.org

10. The Anatomy of Tmux

When you're deep in the guts of tmux things like Sessions, Windows, and Panes, and you're busy learning (and defining) the keystrokes to control all of the above, it can be easy to think that tmux looks basically as good as it ever will. But this is not the case! We're going to step back and take a look at tmux's actual *looks*, the on-screen chrome that separates the panes, tells us what's happening in each window, and gives us some insight into the state of our session. Once we've got a handle on how tmux builds up the screen it shows us we can dive into customizing tmux to look the way you want it to.

Dividers

Remember that as much as it tries to look like a GUI, tmux is a text-based program, and as such can only use text characters to build up a screen. Tmux uses pipes (|) and hyphens (-) to draw vertical and horizontal lines, like the ones it draws between the panes in a window. Why am I telling you this? Because these characters being part of a regular ASCII set means they are subject to being styled by your terminal's default font. Observe:



In this font, there is a fair amount of space between the repeated characters.

Using this font the bars are solid, but you'll notice that a slight amount of "flair" at the ends of the hyphen character give the horizontal bar some odd bumps.

So, step one of understanding your tmux style is understanding your fonts. We will look at ways to change how the dividers are represented in the the next chapter, but for my money sticking to the pipes and hyphens gives the cleanest appearance. But don't let that stop you from using something else. As I said, we'll get into that soon enough.

The Status Bar

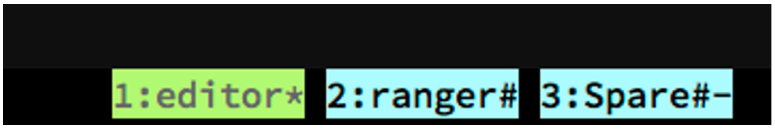
Unsung and uncommented-upon, the status bar relays a wealth of information in a small amount of space, and as we will see, there's a ton of room to customize it as well. Let's take a look at the default status bar, which will give us a starting point for our customization ventures later.

The default status bar, in all it's green glory.

From left to right, this status bar is showing us the following:

- The name of the session, in this case " `0` "
- The number and name of each window, what is running in that window, and which window is active.
- The hostname of the system you're on. This becomes useful if you're often connecting to remote tmux sessions, but if you're configuring tmux for your local machine you might not want this, as it takes up a lot of space.
- The current date and time.

The list of active windows is actually quite useful and informative, if you know what it's trying to tell you. You'll notice the asterisk next to the word "bash" in the status bar up above. That is telling us that window 1 is active, and bash is the last command run in that window. If you have more than one window that little marker becomes more useful, and is joined by a couple of friends:



This status bar has three windows open, and each is in a different state.

In this status bar[1] each of the three windows has a different status marker:

- The first window, which is named "editor" has the "this is the current window" asterisk
- The second window, named "ranger" has the "this inactive window has had some activity in it since you last looked at it" octothorp (#)

[1]Which has been customized somewhat already. Consider this a teaser of things to come.

- The third window, named “Spare” has the “this was the last window you looked at before you switched to the current window” dash (-) as well as the ‘this inactive window has had some activity in it since you last looked at it” octothorp (#).

As you can see, these little symbols convey a lot of information. The official `man` page for tmux gives a more complete list:

```
Symbol    Meaning
*         Denotes the current window.
-         Marks the last window (previously selected).
#         Window is monitored and activity has been detected.
!         A bell has occurred in the window.
+         Window is monitored for content and it has appeared.
~         The window has been silent for the monitor-silence
                    interval.
Z         The window's active pane is zoomed.
```

The status bar is divided into three sections. When we start customizing how our status bar looks, I strongly recommend you look back at the chapter on fluency and keep the instructions for customizing each section of the status bar separate and well documented. Nothing can screw up a good tmux configuration faster or more thoroughly than a mis-configured status bar. On the other hand, tmux’s status bar has the power to convey a lot of useful information in a very compact space, and getting it right is a lot of fun.

So why don’t we dive in?

11. Customizing Tmux

Okay, we're ready to dive in, start changing up the status bar and the dividers and making tmux look exactly the way we want, right?

Well, almost. Before we start let's talk for a moment about colour.

A Word About Colour in Tmux

Yes, that's right, tmux is a non-American product, which means it spell "colour" the way all the *other* English speaking countries spell it. My fellow Americans, we just have to get used to it. It's really not that big a deal, but it will probably screw you up at least once. If one of your "color" settings doesn't work, make sure you spelled it correctly and try again.

To be able to access the full range of colo(u)rs available to tmux we need to make sure the following line is in our `.tmux.conf` files:

```
set -g default-terminal "screen-256-color"
```

This line tells any program running inside tmux that tmux can handle the crushing requirements of a VGA monitor and show 256 different colors, and also tells your terminal program to watch out, because tmux will be hitting it with post-1987-level color requirements. The vast majority of terminal emulators will, of course, have no problem with this. Even OSX's built-in Terminal.app can handle it. But if you want to check your system I recommend you take the following steps:

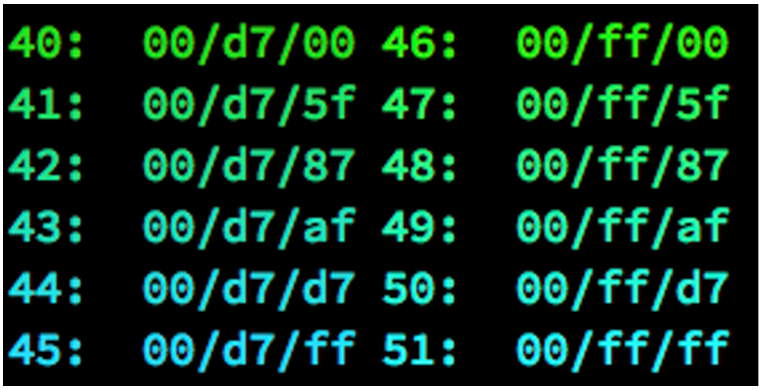
1. Open a tmux session, because we're here to test tmux's color capabilities
2. Grab the the `color test` script from vim.org: http://www.vim.org/scripts/download_script.php?src_id=4568
3. Put that script somewhere nice. I have a directory called `~/bin` for things like this
4. Run `perl ~/colortest -w`
5. Look at the pretty output

If the output is less than pretty check your `.tmux.conf` file, and your terminal emulator's settings. On some linux machines you may need to add a line to your `.bashrc` or `.zshrc` or similar to do a check for tmux and set the color mode appropriately:

```
[ -z "$TMUX" ] && export TERM=xterm-256color
```

Which is basically telling your shell to check if tmux is running and to set the xterm-256color environment variable if it isn't. Tmux will handle it's own TERM variable.

Once you have your 256 color abilities squared away make sure you can find this script again, because the output is going to come in handy when you're actually configuring tmux. You see, in that output it gives you three pieces of information in a remarkably concise way: the first is the actual *color*, as each string is colored the color it represents, if you see what I mean. The second is what tmux calls the *colour name* of the color, and the third is the hex value of that specific color. Altogether it looks like this:



Colour40 is kind of a lime green, colour45 is more of a sea blue.

So, okay, it's time to start configuring tmux now, right? It's time to dive in, right?

Well, almost.

You see, there's one other thing, and this gets messy. We need to talk about the interaction between your *shell* and your *terminal emulator*, because there are some color-management oddities hiding in that gap. Okay, quick history lesson, again:

Back in the day your shell was running on a mainframe and it would compose a screen full of text and send it to your *terminal*, which was a physical box that had a huge clicky keyboard and a monochrome monitor. This is because computers were expensive and huge and frankly it was a pretty good idea. It works well. Eventually some of the terminals got the ability to show four, and then eight, colors, and even sixteen, although in most cases you had eight regular colors and eight "bright"[1] variations on those colors. This also works well, because you can send four bits[2] to tell the terminal what color a section of text should be, which is half as much as it would take to tell the terminal to print a single letter. These colors were standardized and are usually called "ANSI" colors now.

[1] Many people would like to point out that the "bright" variants are not *technically* ANSI colors, they are *AIX* colors. Duly noted.

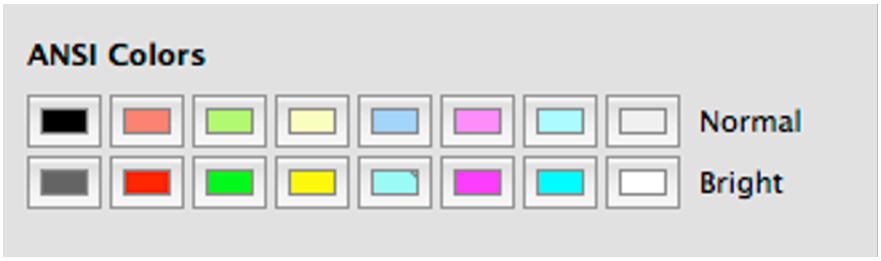
[2] Computer textbooks of my youth were eager to tell you that four bits is called a nibble because it's half of a byte. This is just a thing that gets put in textbooks; nobody actually calls it that. However, this is the second book I've included that little factoid in, because apparently I just can't let it go.

If You're Curious...

Each *normal* ANSI color is represented by an integer from 0-7, and then you add eight for the *bright* color:

- Black (0) & Bright Black (8)
- Red (1) & Bright Red (9)
- Green (2) & Bright Green (10)
- Yellow (3) & Bright Yellow (11)
- Blue(4) & Bright Blue (12)
- Magenta (5) & Bright Magenta (13)
- Cyan (6) & Bright Cyan (14)
- White (7) & Bright White (15)

While this was happening we were getting better at making computers, and the shell and the terminal moved into the same box. Instead of an actual hardware terminal you now have an application that *acts like* (or *emulates*) a terminal, and your shell sends screens full of text to that, instead of sending it over a wire to an actual hardware terminal. And one of the things your terminal emulator emulates is the concept of representing things as one of eight colors (and eight "bright" colors). The problem is that you can set these colors to be any of the *trillions* of colors your computer can display.



"Bright Black" is my favorite color name.

Over time, most shells and terminal programs have come to accept 256-color information, using an entire *byte* to send color data. But a lot of programs still use the ANSI colors, and we're talking programs you use every day like, `ls`, `file`, `cat`, and `man`. The upshot is that when you start playing with color schemes you have to be aware that you need to consider the color settings in two places: your Terminal emulator's ANSI color settings and your tmux config file.

Well, three places. Your actual shell has a color scheme as well, believe it or not. Setting up a fancy prompt is lots of fun, but we're wandering far afield of our topic of "customizing tmux" into "customizing all the things". So I'm just going to say that there are wonderful projects like bash-it[3]

[3] https://github.com/revans/bash-it

or oh my zsh[4] that make it easy to customize your shell to look exactly the way you want it. Let's move on to actually, really, customizing tmux. For real this time!

Actually Really Customizing Tmux for Real This Time

So, if you look at tmux there are really only two bits of "chrome" that you can style: the dividers between panes and the status bar along the bottom. There is a secret third bit as well: the tmux command line, but that's a secret. Let's start with the dividers.

Dividers

When you style an element in tmux, you can give it two styles: a default style and an active style. For each style, you can give tmux a foreground color and a background color. The borders around panes are usually represented by simple, thin lines around each pane. But if you set a background color you can have the pane borders take up a *lot* more space, visually[5]. But let's change that, shall we? Let's say we want our active pane to be delineated by a red line in a white space, while our default panes are surrounded by blue lines on a green background. This sounds terrible and ugly. Let's do it.

Maybe Don't Follow Along Here

Remember back in the beginning where I said I would never have you put anything in your config file that I wouldn't put in, not even for example purposes? This is the kind of thing I was talking about. The colors here are hideous and you should not use them. The principle being taught, however, is useful. Once you've seen how to do this you can try it yourself, but with *nice* colors.

Okay, here is where all that stuff we talked about with the "colours" and 256 color space and ANSI colors comes into play. Tmux is incredibly accommodating when it comes to telling it how to color something. From the tmux[6] man page:

> The colour is one of: **black**, **red**, **green**, **yellow**, **blue**, **magenta**, **cyan**, **white**, aixterm bright variants (if supported: **brightred**, **brightgreen**, and so on), **colour0** to **colour255** from the 256-colour set, **default**, or a hexadecimal RGB string such as '#ffffff', which chooses the closest match from the default 256-colour set.

[4] https://github.com/robbyrussell/oh-my-zsh

[5] Of course, it's taking up the same amount of space either way: one character. This is why we use monospaced fonts in our terminal emulators. But normally the background is the same color as your terminal's background, so the line *looks* smaller.

[6] I am using version 1.9a. You can check your version by running `tmux -V` in any version newer than 1.4. If that command doesn't work, you are using a version older than 1.4 and should *definitely* upgrade immediately.

RGB Colors: Just Don't.

It's tempting, but ultimately disappointing. I know that you have your favorite colors memorized and can type out the RGB hex string for each one in seconds, but remember that tmux will take the color you give it, which is one of literally millions of colors, and map it back to one of the 256 that it understands. This means that a color like `#856363` (dusty rose) gets turned into, basically, red. Just use the 256 color set if you're going to go beyond the 16 named colors.

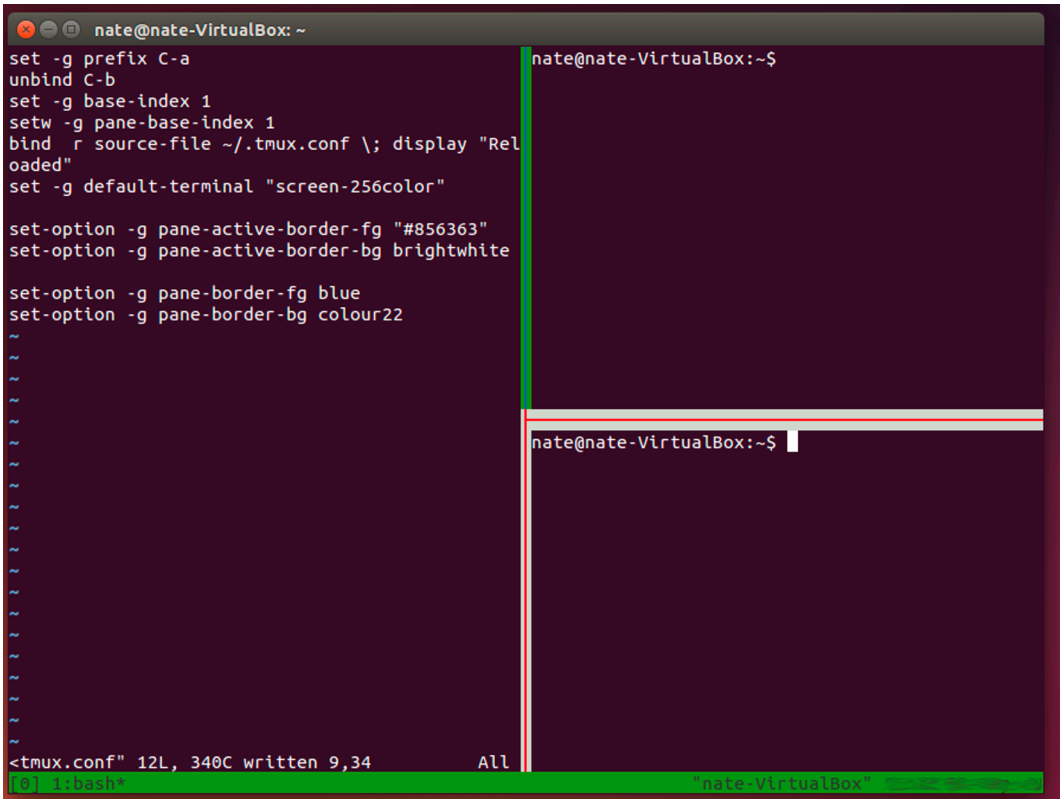
So, let's use this knowledge to tell tmux how ugly we want it to be for a little while. We'll start with the "default" colors:

```
set-option -g pane-border-fg blue
set-option -g pane-border-bg colour22
```

So far so good. We have a thin blue line running through a dark green stripe. Now let's get to the active pane:

```
set-option -g pane-active-border-fg "#856363"
set-option -g pane-border-bg brightwhite
```

And the the result:



These are some truly awesome borders.

This terrible color scheme was chosen to point out a few weirdnesses. The terminal I was using (in an Ubuntu virtual machine) decided that `brightwhite` means "kinda dingy gray". And the aforementioned "dusty rose" RGB color `"#856363"` turned into the aforementioned "red". Only `colour22` turned out the way I was expecting.

There's one option listed in the `man` page that I didn't use: `default`. This is actually quite helpful, as it lets you make sure that things get reset when you're testing out a bunch of color schemes, and it serves as a good placeholder in your file, saying "I considered changing this, but the default is how I like it.". In most config files, you would want the not-active pane section to look like this:

```
set-option -g pane-border-fg default
set-option -g pane-border-bg default
```

At least while you were testing. You see, sometimes, while I was writing this chapter I would change the background color of the active pane to, say, purple, and then reload my .tmux.conf file. Then I would change the foreground but remove the line changing the background. But it would stick around, because it's been told to be purple and hasn't been told to *stop* being purple. So instead of deleting or commenting out the `pane-border-bg` line I just changed it to `default` to reset it, and life was good again.

Status Bar. This is the big one

Okay, on to the status bar. This is the fun part, and also the freaking huge part. I've considered breaking this out into its own chapter, and still might. We'll see. As we've already seen, the status bar is crammed full of useful information, and we can easily add more or even change what it's showing us. But we have another job for it: we need it to look cool. And balancing those two objectives is a mighty task.

As previously mentioned, the status bar is broken up into three pieces. What I didn't say earlier is that you only have control over what appears in two of them. I also didn't mention that they use a "printf" style language to let you include various bits of information that tmux packages up for you. Let's talk about that now.

The left and right sides of the status bar are yours to play with. You can include all kinds of pre-packaged data from tmux in them, and even include the output of shell scripts. The center part of the status bar is used to show you the window list, with all it's glorious symbols about what's going in in which window. We have the least control over what's in that center section, so let's cover it first.

Front and Center: The Window List

Again quoting from the tmux man page, by default the center part of the status bar shows[7]:

> By default, the window list shows the index, name and (if any) flag of the windows present in the current session in ascending numerical order. It may be customised with the **window-status-format** and **window-status-current-format** options.

We can change the information showed there, but for now we're going to leave the data alone and change the colors, because priorities. Also, I've given it a lot of thought, and I can't think of a better format than the default one. It's freakishly compact and informative. If you want to change it I'll show you how, but colors first!

So, in the examples with the dividers I showed you how to specify each part of what you wanted to change, with each change on its own line. And that format still works. We can write something like this:

```
setw -g window-status-current-fg black
setw -g window-status-current-bg green
```

[7] Any quotes from the man page will have British spelling instead of American. We've been over this, please don't send me correction requests. Well, about that one issue. But please send me any *other* corrections you may notice, because it makes this book better for everyone. Also, you're looking really good today. Did you do something new with your hair?

And tmux will do it, no problem. But there's more! The status bar gets access to a whole new level of customization that the divider lines don't have: attributes. Attributes are things like (again, from the `man` page):

> The attributes *[sic]* is either none or a comma-delimited list of one or more of: bright (or bold), dim, underscore, blink, reverse, hidden, or italics, to turn an attribute on, or an attribute prefixed with 'no' to turn one off.

So if we want our current window to have an underline we can add another line, like so:

setw -g window-status-current-fg black setw -g window-status-current-bg green setw -g window-status-current-attr bright

And we'll have a bright green box around black text. But we can also use a more compact format, where we include all the changes we want to make in a single string. To use this new notation you put `-style` where we've been putting `-fg` and `-bg`, so instead of `window-status-current-fg` we would write `window-status-current-style` followed by a quoted string with *all* the styles you want to apply to that element. Tmux calls these "style strings" and they save a lot of space.

So now we can tell tmux not only that we want our current window to be black text on a green background, but we also want it to be blinking and underscored:

```
setw -g window-status-current-style "fg=black,blinking,underscore, bg=green"
```

Please, please don't do that for your current window. We can do the same thing for non-current windows:

```
setw -g window-status-style "fg=cyan,bg=default,dim"
```

And we're good. But wait, there's more! We can actually style windows based on their status! We've got a bunch of styles here, so it's table time.

State	Name of style	What it means
Default	`window-status-style`	Any window that isn't doing anything special.
Current	`window-status-current-style`	This is the window you are looking at right now.
Bell	`window-status-bell-style`	This window has had a UNIX "bell" sent, usually from an error of some sort
Activity	`window-status-activity-style`	This window has had something happen in it since last you looked.
Content	`window-status-content-style`	You set a "content" alert on this window, and your content has been added.

If you're using tmux to monitor some long-running process the `blink` attribute could be one very obvious way to call your attention to a window that has hit a `bell` or `activity` state, so you can jump to it quickly. In my experience, the "Activity" state is fairly common, so I would recommend only using something as irritating as `blink` on bell states, because UNIX doesn't like using the bell unless it has to.

Now, tmux isn't assuming, and it won't highlight activity without you asking it to. To turn on activity monitoring globally add the following lines to your configuration file:

```
Setw -g monitor-activity on
Set -g visual-activity on
```

(Quick refresher: `setw` sets window options, `set` sets tmux options). This way tmux knows you want it to give you activity notices.

But what about the "Content" option? I would definitely put a style for this in my config file, and I'd make it obvious. Something like:

```
setw -g window-status-content-style "fg=yellow,bright,blink,bg=default"
```

So that when I decide I want to monitor a window for content that window stands out like *crazy* when it hits the content I want.

But I wouldn't put a specific content watch in my config file. Instead you define those on the fly. For example, let's say you've got `tail` running in a window, looking at server logs, and you want to know if a certain ip address is connecting to your server. Maybe it's a bad guy maybe it's another server and you're looking at connection patterns. Whatever. We can tell tmux to monitor all the text in a window and notify us when it comes up. To do this all you have to do is use the tmux command line and type

```
Setw monitor-content "192.168.0.138"
```

And when that content shows up the window name will be styled the way you told it to. Remember, in this case we used `setw` instead of `setw -g` because we only want our *current* window to watch for that IP address, not *every single* window.

There's one more line you might want to add to your config file that concerns the center area of the status bar. As we're about to see, the left and right sides can change sizes frequently, even mid-session, and in my opinion, it looks best when the central section stays, well, *central*. If you want it to be centered regardless of what happens on the sides just add this line:

```
set -g status-justify centre
```

With that your status area will justify itself to always be in the center (or centre) of the status bar. Once that's done, let's move on to the left and right sides.

Left (and Right) of Center: Quick Information About Your Environment

So, as much fun as the center of the status bar was, we're about to get into the real goods. The center area is pretty much only used for showing you window information, but the sides can show you *anything*. But, as always, with great power comes great ability to make a mess. Before we start configuring the status bar, consider what you actually want to see.

This is going to change based on the system you're working on. If you're setting up the configuration file for your local system you probably don't need to know its IP address or hostname. If you're setting up a server or desktop you don't need a battery gauge. Let's start with the base configuration and work up from there. The `man` page tells us that

> By default, the status line is enabled (it may be disabled with the status session option) and contains, from left-to-right: the name of the current session in square brackets; the window list; the title of the active pane in double quotes; and the time and date.

Okay, so far so good. This is actually pretty useful. But we're not stopping there. To define what shows up on each side of the status bar we are going to create a format string for that side of the bar, much like we did for the middle section. The syntax is a little different, however. Tmux has a vast number of values you can put into this string. You can see them in the tmux `man` page under the `FORMATS` section, and I'll include the most popular ones in a table here. Don't thank me, it's the least I can steal from the official docs. I mean do.

Let's start by re-creating the default status line. By default, the left side has "the name of the current session in square brackets." To re-create this our entry in `.tmux.conf` looks like this:

```
set -g status-left "[#S]"
```

Okay, that's boring. You've probably guessed that `#S` is the shortcut for the name of the current session. The square brackets are, well, square brackets. Now let's do the right side. We need the title of the current pane, no problem, and the time and date. Easy, right? Well, yes, if you know `strftime` formatting. If you don't then it's a little more work. Let's create this string and then talk about it:

```
set -g status-left "\"#T\" %R %D"
```

No big surprises here: we needed to escape the double quotes inside the string with `\` characters, `#T` is our shortcut for "Name of the current pane" and then we have some `%letter` combinations. Those are `strftime` things. You can go to this site[8] to build up a useful date-time format if you want one in your status bar.

[8] http://strftime.net

We've learned a pattern here. Most characters inside the string are just printed directly, unless they are preceded by a `#` or a `%` in which case tmux will try to make them into tmux commands for the pound sign or strftime formats for the percent sign. I've already given you a link to learn more about the time formatting; let's focus on the commands.

There are four types of commands we can put in the format string:

Directive	What it does
`#(command)`	Insert the first line of the results from that command
`#[attributes]`	Change the attributes for the rest of the string.
`##`	Print the # Character.
`#{Format}`	Expand the Format variable.

"But wait," you're saying, "We just used `#S` and `#T` and those aren't in your convenient little list *anywhere*! You're a liar, Nate Dickson, if that is your real name! A dirty rotten liar!"

Okay, calm down. The `#S` and `#T` commands are just *shortcuts* for `#{session_name}` and `#{pane_title}` respectively. Tmux has a long list of format variables, and they created shortcuts for a bunch of them to make it easier for you to create custom status bar strings. If you want the whole list you can find it in the `FORMATS` section of the tmux man page[9].

Okay, I can see that you're still upset, I'm gonna give you a whole list of the shortcuts here, and then we'll talk when you feel better, okay? Okay.

Shortcut	Long version	What it does	Sample Output
`#H`	`host`	Hostname of computer	dev.natedickson.com
`#h`	`host_short`	Hostname without domain name	dev
`#D`	`pane_id`	Unique Pane ID	15 (if it's pane 3 of window 5?)
`#P`	`pane_index`	Index of current pane	3
`#T`	`pane_title`	Title of current pane	vim
`#S`	`session_name`	Name of the session	Coding
`#W`	`window_name`	Name of Window	Editor

Most of these are very obvious, and you can see where you might use those. Your local server probably doesn't need to tell you the name of your computer, but if you're connecting to a bunch of other servers putting `#H` or `#h` in their status bars isn't a bad plan. You will rarely need to know the unique pane id, and I have even less need to know the "title" of the current pane, which is usually just the name of the last-run process, but the pane index can be helpful. The session name is rarely shown anywhere else, but the window name is in the center section of the status bar, so I usually leave it out.

Tmux provides a lot of other format variables, but they usually aren't useful in the status bar. I

[9] http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man1/tmux.1?query=tmux&sec=1#x464f524d415453

mean, you can include `saved_cursor_x` if you want, but most people don't find it terribly useful information. But if you want it just wrap it in a pound sign and curly brace combo and you're good to go: `#{saved_cursor_x}` if that's your idea of a good time.

Changing Attributes

Once we've got all the information we want in the string, it's time to start playing with presentation. Remember all those attribute things we played with in the center section? You can use them all in the ends as well, by wrapping them in `#[]` marks. Whenever you set attributes they persist throughout the rest of the string, or until you set new attributes. This means you can write cool looking, if clunky to write, strings like this:

```
set -g status-left "#[fg=colour155][#[fg=colour110]#S#[fg=colour155]]"
```

Which will wrap green square brackets around a purplish session name. It kinda looks like grapes!

You have to really really want it to look like grapes.

The point is that you can change anything about the presentation of any element of your status bar. If you want your session name to appear on a white background and blink you can do that. Please don't, but you can. Here, go ahead and try it:

```
set -g status-left "#[fg=colour155][#[fg=colour110,bg=white,blink]#S#[fg=colour1\
55]]"
```

Now, what did we just learn? That's right, when you set an attribute it stays set all the way through the rest of the string. If you wanted JUST the name to blink you need to *un*set the `blink` thing, and change the background color back:

```
set -g status-left "#[fg=colour155][#[fg=colour110,bg=white,blink]#S#[fg=colour1\
55,bg=default,noblink]]"
```

Timing this screenshot just right was fun.

For the record, at time of writing my `status-left` looks like this:

```
set -g status-left "#[fg=colour155][#S] #[fg=colour110]#I #[fg=colour81]pane: #P"
```

Using System Commands

Tmux gives you the ability to include the output of system commands in your status bar, but again, with great power comes great ability to bork yourself. There are a few things to think about here.

Consider Your System

We are used to very powerful computers that can handle huge amounts of processes all at once seemingly by magic. And that's great! But remember that there will probably be times where you want to use tmux on systems that aren't all that powerful, like a raspberry pi. Or on a system that really shouldn't be wasting cycles on formatting, like an image rendering server. Before you make your status line too fancy with external commands think about the cost.

As mentioned previously, tmux will print the *first line* of your output into the status bar. What this means is that you can create a script that prints a single line and make it easy for tmux to run that script, then get the output into your status bar. Since this can be just about anything, let's start simple. Well, kinda simple. Let's have tmux tell us the SSID of the wireless network we're using.

We've got three steps here:

1. Find a command that returns the SSID
2. Process the data so that it *only* shows the SSID
3. Call our new script from tmux

Step one is the step that will have the biggest differences between operating systems. Linux users can use the `iwconfig`[10] suite of tools, and OSX users have the deeply-buried and likely-to-be-deprecated airport[11] command. Since this is just an example I'm not going to worry too much about how you get the data. The OSX Airport thing returns data that looks like this:

[10] http://en.wikipedia.org/wiki/Wireless_tools_for_Linux

[11] http://osxdaily.com/2007/01/18/airport-the-little-known-command-line-wireless-utility/

```
     agrCtlRSSI: -60
     agrExtRSSI: 0
    agrCtlNoise: -82
    agrExtNoise: 0
          state: running
        op mode: station
     lastTxRate: 300
        maxRate: 300
lastAssocStatus: 0
    802.11 auth: open
      link auth: wpa
           BSSID: f4:40:61:ef:42:ac
            SSID: great_wifi
             MCS: 15
        channel: 40,-1
```

And all we need is the stuff after the `SSID:` line. This takes us to step 2: processing the data. You can do this any way you want, but just to show off a bit I'm going to use a perl one-liner:

```
airport | perl -nle 'print $1 if /\bSSID: (\w*)/'
```

If you're not big into regular expressions this won't make a ton of sense, but basically we're asking perl to look for a line that has `SSID` on it, without a letter *before* SSID, and then returning whatever comes next. If we run this on the pretend network above it will return

```
great_wifi
```

All by itself, which is exactly what we want for tmux. So, how do we get this *into* tmux?

That's step 3. A short command like this could just be shoved into your command line format string, except the parentheses will throw it off. Try it:

```
set -g status-right "#(airport | perl -nle 'print $1 if /\bSSID: (\w*)/')"
```

And what you'll actually get is:

```
/')
```

In your status line. Basically tmux looked for everything up to the closing parenthesis and tried to run that, got nothing back (because that command doesn't work) and returned the rest as is. So we'll need to wrap this little utility in a file that we can run.

I have a directory off of my user directory for little scripts like this. Create a file in your favorite text editor and paste the line `airport | perl -nle 'print $1 if /\bSSID: (\w*)/'` right in there. Save it, and now let's make it executable:

```
chmod a+x ~/bin/wifi.sh
```

Test it once yourself, make sure you're getting the right data back, and now we can add that command to our tmux configuration:

```
Set -g status-right "#(~/bin/wifi.sh)"
```

Once again, the sky is the limit here, as long as it returns text in a single line. There are a few caveats to be aware of when writing your script for your tmux configuration:

1. Whatever script you write will be running every time tmux refreshes the status line. You can control this by creating a line in your config file that says `set -g status-interval X` where X is the number of seconds between refreshes. I recommend 15 or 30.
2. If you're running this script two to four times per minute you need to be sure it isn't creating files, even temporary files.
3. Again, be aware of the resources you're using here. Do a few profile runs if you're worried, make sure you're not shooting yourself in the foot.

For another example you can check out my battery meter for tmux[12]. Unicode makes that one line of information far more fun to play with.

Super Secret! The Command Line

Okay, it's time to come clean. The Tmux command line isn't super secret. I was just saying that to give it an air of mystery. One that, frankly, it really doesn't deserve. The fact is, there's not a lot you can do with this one. You can style the background color, foreground color, and maybe make it be underlined and stuff, but that's about it. To set the style on your command line you enter a command that should look super familiar by now:

```
set -g message-command-style "fg=default,bg=default"
```

You could give them colors if you wanted to, I guess. You can also use the attributes `bright`,`dim`, `underscore`, `blink`, `reverse`, `hidden`, and `italics`" although some of those (`hidden` and `blink` spring readily to mind) would make the command line nigh-unusable.

When tmux wants to tell you something it uses the "message" bar. To style the message bar, unsurprisingly, you just enter:

[12] http://natedickson.com/blog/2013/04/30/battery-status-in-tmux/

```
set -g message-style "fg=cyan,bg=purple,blink"
```

Or whatever you want for your styling. Although you you could resist blinking cyan and purple is beyond me.

Okay, we've covered a *ton* of ground in this chapter, and what we've covered here is probably 9/10ths of the reason you bought this book. But don't quit now, because the next chapter will save you bucketloads of time, and once again make you look awesome in the process. Because that's where we start talking about automated setups. Oh yeah. This just got *real*.

12. Automating your Tmux Setup

Up to this point we've been doing all our tmux setup by hand, and on the fly. If we wanted a session with three windows, each with two panes, we had to create all of those at the moment we wanted to use them. And there are tons of times where you will absolutely want to do things this way. Knowing how to split windows into panes, create new windows etc. Will serve you well for as long as you use tmux.

But what about when you want to use the same tmux setup day in and day out? It gets very tedious creating each one of those panes and windows by hand every time, and when you want a specific pane to run a specific program that just adds to the tedium. But we're better than that, you and I. We want to reduce tedium wherever we find it, and we reduce tedium through automation!

In this specific case we will reduce tedium by setting up our favorite working environment *once* and simply issuing a command whenever we want to use that environment. You would expect a tool like tmux, one that's designed to boost developer productivity, to have facilities in place that would make it easy to do something like this. You would be right.

There Is More Than One Way To Do It

There are several ways to automate the creation of your tmux sessions, and they all have their pros and cons. I'm going to focus on using the functionality built into tmux and based on stupidly simple shell scripting, as this method is the one I've found to be the most portable and the most reliable.

Start Simple

We're going to build our first session script iteratively, making sure it works at every step along the way. So, first decision: What do we want to call this session? As we discussed earlier, I generally name my sessions after the project I'm working on in that session. Since I don't know what *you* are working on, I'll just name this session **Painless**. It's a nice word. When we are done, we want to be able to type `painless` at the command line and have our pre-built Painless session pop up like magic. So create a file called `painless` somewhere permanent. I have all my session starting scripts in `~/tmux/` which is under version control, so I can keep all these scripts and move them between my computers, but it's up to you.

Okay, first things first. Before we start a new session named "Painless" we want to ensure that there isn't one already running. If there is we'll just attach to that session. Tmux makes this a very easy check. From the command line run

```
tmux has-session -t Painless
```

This command follows the standard unix convention of returning zero on success (meaning, if there is a session named "Painless" it will put the number 0 in a magic register that we're not going to worry about right now. Unix interprocess communication is not painless. The point is that in our shell script we can run this command and test the result:

```
tmux has-session -t Painless
if [[ $? != 0 ]]
then

fi
tmux attach -t Painless
```

This Bash Script Isn't Optimized

Those of you who are good at shell scripting are going to hate this section. There is going to be a ton of repetition, many things that could be streamlined and optimized and generally made more awesome. This is intentional. I want this section to be perfectly clear even to people who don't write a lot of shell scripts. Just think of it as your chance to improve on what I've done here, and then say to yourself "the student has become the master" in an impressive voice.

For people who don't speak bash[1] the script above runs the `tmux has-session` command and then checks the result. If it got anything other than zero it does nothing (for now). Then regardless of what happened it tries to attach to a session called Painless. The `-t` flag after both tmux commands stands for "target", and as we progress we'll see how fancy that target argument can get, but for now it's our way of telling tmux which session we're talking about.

This script isn't going to do a whole lot right now, but let's make it executable anyway. On the command line run `chmod a+x ~/tmux/painless` to make it a program instead of just a text file.[2] Now test the script by typing `~/tmux/painless` and…you get an error. That's fine, at least it's trying to run the script. We'll call that a success!

The next step is to create the Painless session if it's not already running. But first, let's make sure we're not trying to start tmux from within tmux, like so:

[1] Throughout this section I'm going to use "bash" as a code word for "shell scripting in general". I can verify that these scripts work in both bash and zsh. If you're using csh or fish or what have you it's possible you'll have to make some modifications. But the general principles are the same.

[2] I have also put my `~/tmux/` directory on my path, so I can run any of these scripts without including a full path to the script.

```
tmux has-session -t Painless
if [[ $? != 0 ]]
then
        tmux detach
        tmux new-session -s Painless -n Editor -d
fi
tmux attach -t Painless
```

So the first line just says "detach from any session you may be attached to". This is a good safety precaution, because running a tmux session from within a tmux session tends to be...bad. Yes it's possible, but it's outside the scope of this book.

The next line uses even more fun switches! The `-s` switch (according to the `man` page) lets you "select a new session for the attached client interactively". We are going to say it means "session" and go from there. The `-n` switch means "next window". In this case we are creating the first window and naming it "Editor" because that's generally what I run in my first window. The `-d` switch detaches from the session. We do this so that we can keep sending the session commands from this script. For now save the script and run it, and you should find yourself in a tmux session named Painless, with a single window named Editor. Good work!

Multiple Windows

But we can do so much more. And we will. A tmux window with a single pane feels like a waste. Let's split the Editor window into two panes, and while we're at it, let's actually start up an editor in one.

```
tmux has-session -t Painless
if [[ $? != 0 ]]
then
        tmux detach
        tmux new-session -s Painless -n Editor -d
        tmux split-window -h -p 40 -t Painless
        tmux send-keys -t Painless:1.1 'vim' C-m
fi
tmux attach -t Painless
```

Okay, more fun switches! The `split-window` command accepts `-h` meaning "horizontal". In this case "horizontal" means the windows are next to each other instead of on top of one another. The `-p` switch lets you give a *percentage* of the total screen, and the new window will take up that percentage. In this example we want it to be 40% of our total real estate. And then the trusty `-t` switch lets us tell tmux that we're still talking to the "Painless" session.

But now we've got a problem. What if we want to send commands to individual windows and panes inside that session? Most of you have already looked at the next line and figured it out, haven't you? Nice work. The `-t` switch takes paths like this:

```
Session:WindowIndex.PaneIndex
```

So in this line (`Paneless:1.1`) we want the first pane of the first window in the Painless session. Remember clear back when we told our indices for panes and windows to start with 1 instead of 0? This is why.

But the `send-keys` line is doing more than just showing us how to handle addressing from the command line. You can see that we sent the `vim` command in single quotes, followed by `C-m` which seems cryptic. But it's not. It's just tmux-language for `<enter>`. Go ahead and run this latest version and you should have vim in a pane that is slightly larger than it's empty neighbor.

Where Am I?

We have been designing these scripts to be runnable from anywhere, setting up a perfect session for some project you are working on. More often than not you will want all the shells in your tmux session to be based in your project's directory. By default, when tmux creates a new pane or window the shell is rooted in the current working directory. Let's add a simple line to make sure that all our windows are where we want them to be:

```
tmux has-session -t Painless
if [[ $? != 0 ]]
then
        cd ~/painless
        tmux detach
        tmux new-session -s Painless -n Editor -d
        tmux split-window -h -p 40 -t Painless
        tmux send-keys -t Painless:1.1 'vim' C-m
fi
tmux attach -t Painless
```

Man, that was easy! Way easier than changing the directory for each individual pane after we create it! Not that I have ever done that! Please don't look at the history on my dot files in github!

Building Blocks

You now have all the commands needed to create your own awesome tmux configurations, but let's flesh this one out a bit more. I have found that for most projects I have three base windows:

- Editor
- Servers
- Scratch

All of which are pretty self explanatory: I run vim in the first one (with a side window for quick system commands), I keep an eye on any servers I need for the project (database, web server, whatever) in the second one, and I use the third one for just doing one-off commands here and there.

Now let's implement this in our Painless session:

```
tmux has-session -t Painless
if [[ $? != 0 ]]
then
        cd ~/painless
        tmux detach
        tmux new-session -s Painless -n Editor -d
        tmux split-window -h -p 40 -t Painless
        tmux send-keys -t Painless:1.1 'vim' C-m
        tmux new-window -n Servers -t Painless
        tmux split-window -v -t Painless:2
        tmux send-keys -t Painless:2.1 'ranger' C-m
        tmux send-keys -t Painless:2.2 'ml start' C-m
        tmux new-window -n Scratch -t Painless
        tmux send-keys -t Painless:3.1 'git status' C-m
fi
tmux attach -t Painless
```

And now we've got three windows, all ready to go.

What is "Ranger"?

Good eye! Ranger is a terminal based file browser that I often leave running in my tmux sessions, partially because it's useful to get the lay of the land, especially on projects where I'm working with a lot of other people who are adding files all the time, partially because it looks cool. You can find out more about it here[3], and install it via your favorite package manager.

[3] https://github.com/hut/ranger

Shutting Down

One word of caution: that which you create you must also destroy. Sometimes. When you start a bunch of processes like this in tmux they will *usually* be shut down when you end the session, as they are children of that session. But not always. For example, in the code above I ran `ml start` which is a script designed to start a server.[4] In this case the *script* is a child of my tmux session, but the actual server *isn't*. So if I start `ml` in tmux I can close the session and still have access to the `ml` server. If that's not what you want then you need to specifically run a script like `ml stop` before you close your tmux session.

One more word of caution: It's tempting to leave `top` running in a pane somewhere so you can keep an eye on your processes while you're coding. But I wouldn't recommend it. On OS X at any rate `top` tends to get resource-hungry in and of itself, and you've got better ways to keep an eye on how things are going under the hood. But that's just my opinion.

Conclusion

Scripted tmux sessions are a huge productivity enhancement, making it dead simple to set up a session exactly the way you want it without thinking about it. Naturally you should have all your tmux setup scripts in source control, and don't be afraid to update them over time as a project changes or you learn more. As usual, the tmux man page has scores of commands you can run in your setup scripts, which is one of the greatest things about tmux.

[4] I don't know of an actual server called `ml`, I just put this in there to demonstrate how I would use it.

13. Remote Tmux

As a developer, my main use of tmux has been to make my shell environment massively functional. I can use my customized-to-the-hilt vim install alongside a pane dedicated to git, have a window full of server output and tails on log files, and basically I can see all of the moving parts of my project under development in one window. And that's awesome. But tmux was built on a client-server model for a reason, and we can get a lot of milage out of this design decision. The time has come for us to look on how you can use tmux on remote machines.

And there are really two sides to this equation. The first is using git on a headless or remote resource, like a server. When I have done tmux training classes this has been how probably 90% of the attendees plan to use tmux. They do a lot of work keeping servers up and running while we developer divas think of new and better ways to break the servers with our amazing code.

But there's another use of remote tmux: collaborative development. Often called "Pair Programming", tmux makes it easy for two people to work on a single code base without having to look over each others shoulders, and gives them the freedom to work together even if they're not on the same continent.

Both of these scenarios are simple to set up, but require some planning and forethought to set up correctly. Let's take a look at how to use tmux in remote environments without setting yourself up for disaster.

Servers

When working on a remote server your main consideration should be keeping the server happy. No fancy scripts to update your tmux status bar, for example. Tmux itself is fairly lightweight, but remember that it creates a new instance of your shell for every pane you have open, so think hard about how many panes you actually *need* to get the job done. In most cases three or four panes in two windows (in one session) should do the trick.

But let's actually get into an example. Let's say you have a server named `SOL` and you're named, I don't know, "Joel". The first thing you do is connect to `SOL` like you always would:

```
Joel@local> ssh joel@sol.deep13.com
```

Just Go With It

I'm going to gloss over ssh best practices as they fall outside the scope of this book. For now I'm going to do things the simplest possible way and let you work out how you want to do things like key generation on your own.

And then you're signed in! Now start tmux on the remote machine:

```
Joel@SOL> tmux
```

So far we've done exactly nothing interesting. From here you can set up your tmux configuration like you always do. Again, we're not doing anything we haven't done a million times already in this book. If you're on the ball you've got all your dot files on the server so your tmux sessions are styled up nice and your editors behave the way you want them to.

Branching Your Dotfiles

This isn't directly related to tmux, but it fits here. When you have spent a lot of time getting vim to run just so and tmux to look exactly right it can be a terrible feeling to sit down at a new computer and have to start all over again. This is one reason why your dotfiles should live in a source code repository.

But what's good for OS X may not be good for Debian. There may be things in your desktop `.tmux.conf` file that are too resource intensive to run on your raspberry pi.

This is why git is awesome. Create a branch for your new system and update the configurations for each machine at will. When you re-style or update your "main" configuration you can then easily copy the changes into your other configurations without having to re-create the wheel every time.

So here's where we get some bang for our tmux buck. If you were to detach from your ssh session immediately you would lose your setup. Everything that was running under your username would end, and when you logged back in you would have to do it all over again. But we can avoid that. Before logging out of your ssh session all you have to do is detach from your tmux session. Then when you kill your ssh connection the tmux *server* is still happily running on your remote machine, keeping your session alive and well until you come back, with everything right where you left it.

Now, depending on the server you're accessing leaving four shells and a couple of apps running could be a big no-no. If you're working on your home server that gets a few hits from family members that's one thing. If you're editing config files on one of Amazon's load balancers that's something else entirely. Be aware of the context of your actions.

Keep It Simple

Your server has one main job, and it's not running an instance of tmux. Don't do anything that will interfere with what your server is supposed to do. I realize that I have said that four times already, but it's worth repeating once more, with a level three heading and everything. *Keep It Simple.*

So, what does simple look like? For the most part we're talking about the status bar setup here. No matter how crazy you get making things blink you're not going to overload the server with graphical

effects. However, something like the "how much battery power do I have left?" script is definitely a bad idea, because 1.) servers don't run on batteries and 2.) it's running a lot of scripts to draw a few stars. In it's place I would recommend just putting the name of the server (which is `#H` in short code language) so your right hand status string would look like this:

```
set -g status-right "#[fg=colour155]#I | #[fg=colour45]%d %b %R"
```

In other words, the current host and the current date and time. See? Simple!

Team Tmux

A frequently touted feature of tmux is the ability to have more than one person connect to a tmux session, so they can view the same files at the same time. Used properly, this can be a huge productivity boost: two team members from different sides of the world can work together, see exactly what the other person is seeing, and code socially. Technologically this is very easy. All you need is:

1. A computer that can run tmux
2. Tmux
3. Two or more people.
4. At least two computers that can run ssh

Let's return to our example above. Joel has his local machine, and a server called `sol.deep13.com`. If you look back at the beginning of the chapter, Joel has been leaving tmux running on this server so he can sign in and have all his settings right where he wants them. So far so good. But what if Joel wants to work with a new guy? Let's say his company hires someone named Mike, who is also assigned to work on the `SOL` system. Joel could tell mike to ssh into the SOL box and connect to the same session that Joel already set up. So he tells Mike that the session is called "ScreamingSkull" and Mike, armed with this information connects to SOL and types

```
tmux attach -t ScreamingSkull
```

And he's able to pick up right where Joel left off. In fact, they can work together...kind of. They can both connect to that tmux session and see the same stuff, which is pretty cool. But there are some dangerous downsides. Let's see if Joel and Mike will demonstrate some of those for us.[1]

[1] Yes, I'm going to keep this up for a while. You should really just relax.

Don’t Let People Inside Your Shell

So, up to this point Joel and Mike have been doing a very good job of using tmux on a remote system as a way to collaborate on projects and files. But let’s say that one day Joel gets lazy, and doesn’t want to push all his new changes up to the SOL to have Mike see them. So he opens the ssh port on his local machine, and tells Mike to just connect to his computer directly. They use tmux just as they always have, but something just isn’t quite right. And, to make a long story short, they both end up dead and dishonored.

Inviting someone into your local machine is a recipe for trouble. OS X and Linux both benefit from decades of UNIX wisdom in terms of user permissions and roles and whatnot, but that doesn’t mean that your personal laptop should be a multi-user sandbox. *That’s not what it’s for*. Tmux makes the problem even worse, because as far as the system is concerned the “user” is whoever started the session, so if two people are sharing a session they are, as far as the computer knows, one person. So if you start a tmux session on your local computer and let someone else connect to it they are, for all intents and purposes, you. They can delete everything in your home folder, mess around with your settings, edit all your dot files and the system won’t bat an eye[2] . Worse, if you happen to `sudo` something in a shared session that person now has sudo rights for as long as your system lets you do things without re-entering a password.

So don’t do it. It may seem like a pain to have to set up a third machine just to get someone else’s opinion on some code you’re refactoring, but it’s worth it. Even if someone is your best friend that doesn’t mean you give them unfettered access to your home folder. This is one place where a little bit of healthy paranoia goes a long way.

But let’s move on from this grim spectacle of death and carnage before we start wearing tinfoil hats. Let’s talk about how you can work together in a sane and happy way.

Sane Standards

Okay, now we come to a sensitive topic, and we’ll need to talk about it openly, like adults. This may lead to fights, but if we’re all mature we can get through it without anybody getting injured. We’re going to talk about editor settings. Worse, we’re going to talk about shared editor settings.

Quick example. Both Joel and Mike are vim users. Good for them! But wait! One day when Joel is signed into the SOL while he reads an amazing and lighthearted book all about vim[3]. He realizes that he’s been an idiot all these years, using ‘ \ ‘ as the leader key in vim, and sets the leader to `,` like a sane person. But in his excitement he forgets to tell Mike that he made this change.

The next day Mike signs into SOL and none of his leader shortcuts work any more. Joel isn’t at his desk, so he’s baffled[4]. But still, he can’t use a few specialized shortcuts, what’s the worst that could happen?

[2] This is mostly because computers don’t *have* eyes, but you get the picture.

[3] http://painlessvim.com

[4] And apparently not all that clever.

They both end up dead and dishonored.

Okay maybe not. Maybe it's not actually that big a deal. But what we learn from this example is that you need a set of sane, agreed upon standards for a shared tmux server.

There are a few ways to do this. One is to have everyone who will be using the shared server sit down and talk about what standards they want in the config files on the server. This is a good way to lose friends and alienate people. The tabs-vs-spaces fight alone can lead to bloodshed.[5] And once you get pas this, you have to talk about the number of spaces per tab. This battle will never end.

So it's clear that you and your dysfunctional teammates will lose literally years of productivity if you try to make your own set of standards. That's no way to live. Instead you should borrow someone else's standards. This is the value of the internet: you can let someone else have the fight and reap the benefits.

If you're all using vim, for example, you should grab Tim Pope's Sensible Settings[6] and just agree that on the shared server you will do what Sir Tim of Pope says is best.

> ## What are "Dotfiles"?
>
> this is something that you're already familiar with, even if you don't know the name. "Dotfiles" generally refers to configuration files stored in a user's home directory. Your `.tmux.conf` file is a dotfile. On unix-based machines putting a dot in front of a filename hides it, and makes your home directory look cleaner.

But there's so much more. Github has done yeoman work in collecting people's dotfiles[7] and making them available to you. Many of the best and brightest in the programming world have put their dot files up for everyone to find. So did I! You can find mine here[8]. I suggest you find a name you all trust and steal their collection of dotfiles, clone them onto the shared server.

Agree on the Size of Your Terminal Window

Tmux does a lot of things right, and window geometry is one of them. The reason for this is that tmux geometry is based on percentages instead of absolute rows and columns. If you start a session in a window that's 80 columns by 200 rows tmux will do all the math to make your panes all look right in that window. If you then resize to 180 by 450 it'll resize everything for you without batting an eye.

[5] the correct answer is spaces, by the way. Don't even get me started.

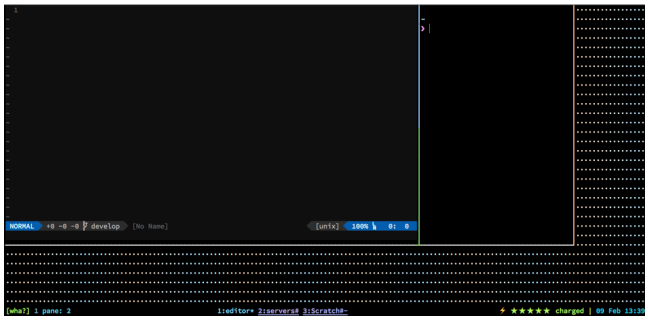
[6] https://github.com/tpope/vim-sensible

[7] https://dotfiles.github.io/

[8] https://github.com/PogiNate/dots

> ## My screen has *Waaaaaay* more than 400 by 80 Pixels!
>
> Yep! Remember that programs like tmux are pixel agnostic, meaning they have *no idea* how your screen actually looks. All they care about is characters. When you connect to tmux your terminal program tells tmux "I'm 60 characters wide and 80 characters tall". Tmux sends those characters to the terminal and lets the terminal figure out if those are 14 point characters or 48 point characters.

But what if two people are connected to the same session? And what if one of them has their terminal set to display at 175 by 40 and one is set to 300 by 150? Well, then the person using the bigger screen gets this:



Look at all those dots!

You see, tmux has to make a decision. If two people are connected to the same session which client sets the size of the windows and panes in that session? Tmux has decided that, given the fact that terminal windows don't have scroll bars[9] it will resize everything based on the smallest terminal window connected to the session. If you are using a larger terminal then you get the world o' dots treatment. It's not perfect, but it works pretty well.

The way to avoid the march of the dots is to just agree ahead of time how big your terminal windows are going to be when you connect to a session. Or you could just re-size you terminal until the dots went away. It's up to you, really.

Automated Sessions Are Your Best Friend

When you're working with others on a shared server it becomes imperative that you set up some automated sessions. If you've built your automated sessions correctly, this makes it painless to connect to a known session without having to run `tls` to see what sessions are already running.

The nice thing about the automated session scripts we created earlier is that they're singleton style: if there's already a "bob" session running you're attached to that session. If there isn't already a

[9] *Usually* don't have scroll bars. If someone is connecting to tmux via an old school IBM green screen terminal there is no way for them to scroll. I know all the fancy new terminals emulators have scrollbars.

session named "bob" then it creates a new one, but you really don't have to worry about it. Either way you just type `bob` at the command line and you're golden.

There's another reason to use automated sessions when you're working remotely, and that's screen size. When you're working locally you more or less always have your terminal window at the size you like it. But what if you are connecting via your tablet or phone? On a smaller screen you'd probably want each window to be focused on one or two panes instead of three or four.

It's not as crazy as it might sound. Panic makes Prompt[10], an excellent SSH client for the iPhone and iPad, Android has apps like Juice[11], and Microsoft Surface tablets run just about any program you can think of. I know a couple of system admins who have finally stopped carrying their laptop everywhere in favor of a lighter and more portable tablet+keyboard combo, and if you need to restart a server in a hurry being able to do so from your phone could be a lifesaver.

To this end, I would recommend having special automated sessions for smaller form factors. Let's look at how these three sessions would look.

First, the desktop session. I know there's a lot of talk about "mobile first!" but that's websites, not command line interfaces.

So, let's look at the setup script for a standard desktop session;

```
#! /usr/bin/env sh
tmux has-session -t MST

if [[ $? != 0 ]]
then
  cd ~/Projects/sandbox
  tmux detach
  tmux new-session -s MST -n editor -d
  tmux send-keys -t MST 'vim' C-m
  tmux split-window -h -t MST -p 30
  tmux new-window -n servers -t MST
  tmux split-window -v -t MST:2 -p 35
  tmux split-window -h -t MST:2.1 -p 30
  tmux send-keys -t MST:2.1 'ranger' C-m
  tmux send-keys -t MST:2.3 'git status' C-m
  tmux send-keys -t MST:2.2 'node' C-m
  tmux new-window -n Scratch -t MST
  tmux send-keys -t MST:3 'brewup' C-m
  tmux select-window -t MST:1
fi
tmux attach -t MST
```

[10] https://panic.com/prompt/

[11] https://play.google.com/store/apps/details?id=com.sonelli.juicessh

This should be very familiar. The first window is running vim and has a 30% column on the side for entering commands, the second window has the ranger file browser, the latest from git, and node running in interactive mode, and the third window is for doing anything else that might come up, but I've got it running my `brewup` alias[12] so that I know my homebrew[13] recipes are always up to date. This is basically my template from the Automation chapter.

So now let's modify it for the iPad[^OrYourTablet]. The first thing we need to do is cut down the number of windows and modify where they are going to show up.

If you're using tmux on a tablet you're probably going to want to do two things:

1. Use the tablet in landscape mode
2. Use a bluetooth keyboard. Onscreen keyboards take up a ton of real estate.

So let's lay out your session for a widescreen. Fortunately at this level you don't have to make too many changes. I generally move the sidebar on the editor window to be a bottom bar so that I can get to a full 80 characters across.[^Why]

For the "Servers" window I would recommend dropping the small pane that shares the top row, and I leave the "Scratch" window completely alone. It's fine.

The config file for our new setup looks like this:

```
#! /usr/bin/env sh
tmux has-session -t MST

if [[ $? != 0 ]]
then
  cd ~/Projects/sandbox
  tmux detach
  tmux new-session -s MST -n editor -d
  tmux send-keys -t MST 'vim' C-m
  tmux split-window -v -t MST -p 30
  tmux new-window -n servers -t MST
  tmux split-window -v -t MST:2 -p 35
  tmux send-keys -t MST:2.1 'ranger' C-m
  tmux send-keys -t MST:2.2 'node' C-m
  tmux new-window -n Scratch -t MST
  tmux send-keys -t MST:3 'brewup' C-m
  tmux select-window -t MST:1
fi
tmux attach -t MST
```

[12] `brew update; brew upgrade --all`

[13] http://brew.sh/

And finally we have the Phone version. On a phone you really don't have a lot of room for fanciness, and I've found I go down to a single pane on my "Editor" window, and the same two panes on the "Servers" window:

```
#! /usr/bin/env sh
tmux has-session -t MST

if [[ $? != 0 ]]
then
  cd ~/Projects/sandbox
  tmux detach
  tmux new-session -s MST -n editor -d
  tmux send-keys -t MST 'vim' C-m
  tmux new-window -n servers -t MST
  tmux split-window -v -t MST:2
  tmux send-keys -t MST:2.2 'git status' C-m
  tmux send-keys -t MST:2.1 'node' C-m
  tmux new-window -n Scratch -t MST
  tmux send-keys -t MST:3 'brewup' C-m
  tmux select-window -t MST:1
fi
tmux attach -t MST

~~~~~~

The sharp-eyed among you will notice that I gave all three of these the same ses\
sion name, `MST`. This was on purpose. All three scripts will check to see if th\
ere's already a running MST session and connect to it instead of creating a new \
one. I've gone back and forth on this a number of times, but overall I feel that\
 this works better for me, as it means I've only got one copy of vim running at \
a time, one node instance, etc. If you're working on a multi-developer machine i\
t might be better to have your scripts set up different sessions so you don't st\
art playing in someone else's sandbox.

D> ## Tmux's "Zoom" Feature is Made for Mobile
D>
D> One of the best features in tmux is the "zoom" feature that lets you focus on\
 a single pane for a while. We've covered it before, but here's a quick refreshe\
r. When you want to make one pane take up the whole window temporarily you type \
`<prefix> z` and the currently selected pane becomes a window for a while. When \
you're ready to let the other panes back onto the screen just type `<prefix> z` \
again and  you're back to normal. The fact that this feature exists is why I sti\
ll have a two-pane window in my iPhone script. I can quickly see if there's any \
```

```
tiny red letters indicating a problem and zoom in on them to get the details.

[^OrYourTablet]: Or your tablet of choice. I'm currently rocking a Kindle Fire!

[^Why]: I've never figured out why we still maintain this 80 character width, bu\
t whatever.




{bump-link-number}

# Fancy Tmux Tricks {#Tricks}

This section is all about little things you can do that don't really fit anywher\
e else. These are the odds and ends; things that I wanted to include but that do\
n't really require a full chapter of their own. You've made it through a whole b\
ook full of heavy stuff, now let's try some of the fun stuff.


## Clock!

Honestly, this chapter exists almost entirely because of this command. It's a bu\
ilt-in that tmux includes, and, I suppose, given the right circumstances, it cou\
ld be useful. Here's the scoop: press `Prefix-t` and you get a clock that fills \
the current pane. This is less useful in a huge single-pane window, more useful \
as the third little pane off on the side of your session. I'm not sure how often\
 anyone would really use this, since you probably also have a clock on your OS's\
 menu bar or task bar or whatever, and if you're me you also have a clock on the\
 status bar. But if you want a bigger, fancier, ASCII-er clock, this is the comm\
and for you.

## Programmatically Check for an Active Tmux Session

This one is a bit more useful, and a bit more involved.

When you're using tmux you may occasionally want to check if  you're actually in\
 a tmux session or not. this is fortunately very easy. When you are in a session\
 the `$TMUX` variable is set, with the ID of the current tmux session. If you're\
 not in a session it's not set. So if you run
```

```
        echo $TMUX

From the command line outside of a session it will return nothing. If you run it\
 from within a session it will return something like this:

     /private/var/folders/p4/x66czy2s5vj4_4l_3k9jftq40000gp/T/tmux-502/default,4\
416,0

Which gives you a very easy check. If the `$TMUX` variable is empty you aren't i\
n a tmux session[^ZeroLength]. If it's populated, you are. From the command line\
 you can do something like this:

     [ -z $TMUX ] && echo "Empty: Yes" || echo "Empty: No"

and from a script you would do something like

[^ZeroLength]: That's what the `-z` check is. In bash and zsh it means "is the f\
ollowing variable zero-length?"

{lang="bash"}
~~~~~~~

if [ -z $TMUX ]
then
     //do something
else
    //do something else
fi
```

So that gets you a nice boolean: Am I in a TMUX session or aren't I?

But what if you want to know *which* tmux session you're in from a shell script? We've accessed the session name before, when we were setting up the status line. You might remember that there's a nice shortcut (#S) for formatting strings. But we're not setting up a format, we're just asking for a string. You can ask tmux to display a message, and inside that message you can use all those formatting shortcuts we know and love. Try something like this from the command line inside a tmux session:

```
 tmux display-message -p '#S'
```

Good stuff, no? The problem is that this command also works from *outside* of tmux if a session is running, and will return the name of the most recently attached session. This means if you start a session called "Painless" and then disconnect the command above will return "Painless". So I would only suggest using this command if you have already checked the $TMUX variable. Of course, you can do both in a single line:

```
 [ ! -z $TMUX ] && tmux display-message -p 'Current Session: #S' || echo "No Act\
ive Session"
```

And you'll be in good shape. Now, what you actually *do* with this information is up to you.

IntelliTmux

Every once in a while you may have to actually step out of the terminal and use a full IDE. It happens! IDE's are amazing, because they can bring just absolutely everything into one container and, you know, occasionally bog your system down so bad you have to restart. But that's not the point.

The point is this: Just because you have to use an IDE doesn't mean you have to leave behind the power of tmux. Most IDE's worth the name have a terminal option, a small terminal that you can pop up somewhere in the IDE window. See where I'm going with this?

In essence, setting up a tmux session that you can use from within your IDE gives you a choice. If you want to do something the way the IDE does it you can. If you want to flex your command-line muscles it's just a keystroke away. We're finally living in a world where GUI and CLI don't have to be mutually exclusive.

Obviously, a tmux session you design to run from within IntelliJ won't have an "editor" tab, but it'll be close to the generic template we set up in the Automation chapter.

Well friends, we have reached the end of our little adventure into tmux. We had fun didn't we?

Hopefully by now you've got all you need to get in there and start using tmux to make you more productive on your local box as well as on any servers you happen to access on a regular basis.

There's still quite a bit to learn, but I think I've covered about 80% of what you really need to know.

And if I didn't cover something you think is important, or if you have questions, let me know! You can email me at `painlesstmux@natedickson.com` and let me know what's on your mind.

Appendices

Installing Tmux

On Linux

Your friendly linux package manager should have the latest tmux, if your distro doesn't have it by default. For example, on Ubuntu, you could just type

```
sudo apt-get install tmux
```

To get the latest version. The exact command may vary, but if you're a linux user you're used to that by now, and let's face it, you probably got it installed without even reading this section.

On OSX

I strongly recommend you use Homebrew[14] to install tmux. There are other methods available as well, of course, but Homebrew is actively maintained and is pretty slick and easy to use to boot. Once you have homebrew installed you can just type

```
brew install tmux
```

And you're good to go.

[14] http://brew.sh/

Should I Use iTerm2?

My opinion is that you should not. iTerm2[15] was created when the built-in Terminal.app was very lacking. As in, it wouldn't display 256 colors. Naturally power users wanted something better, and an open source project was born. And for a long time it was an indispensable part of an OSX programmer's tool belt.

But the thing is, Terminal.app has gotten much better. It handles a full color palette now, as well as most if not all of the power features that iTerm2 was built on. And iTerm has created a weird feature that I would strongly recommend you *not* use, which is their tmux integration.

On paper it looks like a great idea. You can issue tmux commands and manipulate tmux windows using the mouse! You can use menu commands to create tmux things! But it does some weird things, not least of which it requires you have *two* windows open, one that is running the special iTerm version of tmux, and one that is your actual tmux session. But even in that session you have some serious weird stuff going on. It changes all your shortcuts for things like creating new panes and windows. Beyond that, if you get really used to how iTerm does tmux, you'll be kinda lost if you ever have to use tmux on a Linux server or some other environment.

It's worth noting that you can use tmux normally in iTerm2, and it works just fine. If you love iTerm2 I would recommend using it for all it's awesomeness but forgetting about it's tmux mode.

On Windows

Yeah, listen windows bro: nope, sorry. You might, with a lot of work, have some success with Cygwin, but in general the results have been less than stellar. In general your best plan is to go to the next about Using Vagrant.

[15] http://www.iterm2.com/#/section/home

All the Command Tables!

Throughout the book I've been giving you tables full of useful commands, as well as suggested changes to those useful commands. In this appendix all of those are collected into a single place. The Commands are grouped by the chapter they came from, and are in the same order in which they appeared in the text.

Config

Base Configuration

Original	Suggested	What it does	`.tmux.conf` line to add
`<ctrl>b`	`<ctrl>a`	`<prefix>`	`set -g prefix C-a`
`<prefix> %`	`<prefix> |`	Split the window into two horizontal panes	`bind | split-window -h`
`<prefix> "`	`<prefix> -`	Split the window into two stacked panes	`bind - split-window -v`
`<prefix> x`	`<prefix> x`	Close a pane	`bind x kill-pane`[16]
N/A	`<prefix> X`	Close the session	`bind X kill-session`
`<prefix> <ctrl>b`	`<prefix> <ctrl>a`	Sends `<ctrl>a` to an application running inside tmux	'bind C-a send-prefix
`<prefix> :`	N/A	Enter command mode.	N/A

Panes

[16]Remember that you'll have to add `unbind x` *Before* entering this command to remove the original binding.

Default Layouts

Keystroke	What it does
`<prefix> <alt>1`	Switch to Even Horizontal Layout
`<prefix> <alt>2`	Switch to Even Vertical Layout
`<prefix> <alt>3`	Switch to Main Vertical Layout
`<prefix> <alt>4`	Switch to Main Horizontal Layout
`<prefix> <alt>5`	Switch to Tiled Layout
`<prefix> <space>`	Rotate through the default layouts

Resizing Panes

Keystroke	What it does
`<prefix> <ctrl>up`	increase the height of this pane by one row
`<prefix> <ctrl>down`	decrease the height of this pane by one row
`<prefix> <ctrl>left`	increase the width of this pane by one column
`<prefix> <ctrl>right`	decrease the width of this pane by one column
`<prefix> <alt>up`	increase the height of this pane by five rows
`<prefix> <alt>down`	decrease the height of this pane by five rows
`<prefix> <alt>left`	increase the width of this pane by five columns
`<prefix> <alt>right`	decrease the width of this pane by five columns

Resizing Panes, Vim Style

Original	Suggested	What it does	**`.tmux.conf`** line to add
`<prefix> <ctrl>up`	`<prefix> K`	increase the height of this pane by one row	`bind-key -r K resize-pane -U`
`<prefix> <ctrl>down`	`<prefix> J`	decrease the height of this pane by one row	`bind-key -r J resize-pane -D`
`<prefix> <ctrl>left`	`<prefix> H`	increase the width of this pane by one column	`bind-key -r H resize-pane -L`
`<prefix> <ctrl>right`	`<prefix> L`	decrease the width of this pane by one column	`bind-key -r L resize-pane -R`

Other Pane Commands

Keystroke	What it does
`<prefix> q`	Show the pane number in each pane briefly.
`<prefix> !`	“Break” a pane into a new window of its own.

Other Pane Commands

Keystroke	What it does
`<prefix> z`	"Zoom" a pane to fill the window, or zoom it back out.

Window Commands

Switching Windows

Keystroke	What it does
`<prefix> 0-9`	Switch to window 0-9
`<prefix> w`	Choose a window from a menu
`<prefix> c`	Create a new window
`<prefix> ,`	Rename current window
`<prefix> n`	Cycle to next window
`<prefix> p`	Cycle to previous window
`<prefix> <alt>p`	Previous window with activity
`<prefix> <alt>n`	Next window with activity

Session commands

Session Commands

Keystroke	What it does
`<prefix> (`	Switch to previous session
`<prefix> )`	Switch to next session
`<prefix> s`	Choose from a list of sessions
`<prefix> $`	Rename the session
`<prefix> d`	Detach from your current session

Copy Mode Commands

Keystroke	What it does
`<prefix> [`	Enter Copy Mode
`<prefix> ]`	Paste current buffer
`<prefix> =`	List all buffers and choose one from which to paste
`<space>`	(in copy mode) start selection
`<enter>`	(in copy mode) Exit copy mode, copy selection to buffer

Keystroke	What it does
h j k l	Move one space left, up, down, right, respectively
`<ctrl>b`	Scroll up one "page"
`<ctrl>f`	Scroll down one "page"
g	Jump to the top of the buffer
G	Jump to the end of the buffer
w	Jump to the beginning of the next word
b	Jump back one word
f*n*	Jump forward to the next occurrence of *n* on that line
F*n*	Jump backward to the previous occurrence of *n* on that line
`?*search*<enter>`	Search backwards through the buffer for *search*
`/*search*<enter>`	Search forwards through the buffer for *search*
n	Repeat the last `/` or `?` search in the same direction
N	Repeat the last `/` or `?` search in the opposite direction

Original	Suggested		What it does	`.tmux.conf` line to add
N/A	`<prefix>`	`<ctrl>c`	Choose a buffer and paste the contents	bind C-c choose-buffer
N/A	`<prefix>`	`<ctrl>s`	Save a copy of the current pane to `~/buffer.txt`	bind C-s capture-pane; save-buffer ~/buffer.txt

Customizing Tmux

Directive	What it does
`#(command)`	Insert the first line of the results from that command
`#[attributes]`	Change the attributes for the rest of the string.
`##`	Print the # Character.
`#{Format}`	Expand the Format variable.

Shortcut	Long version	What it does	Sample Output
`#H`	`host`	Hostname of computer	dev.natedickson.com
`#h`	`host_short`	Hostname without domain name	dev
`#D`	`pane_id`	Unique Pane ID	15 (if it's pane 3 of window 5?)
`#P`	`pane_index`	Index of current pane	3
`#T`	`pane_title`	Title of current pane	vim
`#S`	`session_name`	Name of the session	Coding
`#W`	`window_name`	Name of Window	Editor

Useful Aliases

There are a bunch of tmux commands that you issue from the command line, things that make it easier to see what tmux is up to, or to start new tmux sessions in special ways. The problem is that a lot of them are fairly long and tedious to type, which is why I recommend creating aliases for these commands.

A quick word about aliases: put them in a file called ~/.alias instead of keeping the list in your shell config. This makes it easy to try out new shells, it makes it easy to put your aliases in source control, and it makes it easy to edit them without worrying about messing up your main config file. Just add the line:

```
source ~/.alias
```

And you're golden. Okay, so, before you set up any tmux aliases, you're going to move all your other aliases to an alias file, right? Okay, I'll wait.

Using AKA for Alias Management

A while ago some *genius*, some *visionary*[a] created a little ruby command line application that makes alias management much simpler. Not that it's hard to add a few aliases to your `.bashrc` or `.zshrc` file, but you shouldn't have to. You can just create all your aliases in a single alias file and just source that in your shell's config file. And while you're doing that, why not have an easy way to manage those aliases, without having to edit the file directly? That's what `aka` is all about.

To install `aka` you just run `gem install aka` or `sudo gem install aka` depending on your setup. Once you've installed aka you'll need to add the file it creates and maintains (~/.alias) to your shell config by adding the line `source ~/.alias` somewhere in the file. All of these instructions are on the aka GitHub page[b]. I have found that aka is an incredibly fast and frictionless way to handle all the aliases that I use when working with tmux.

[a] Yeah, it was me. Are you surprised?
[b] https://github.com/PogiNate/A.K.A.

If you decide to use `aka` you can just type

```
aka -a alias "command"
```

To set up a new command, otherwise you'll just add a line to your `.alias` file. In keeping with the rest of the book, I'm going to put all of these into a nice pretty table.

Command	Suggested Alias	What it does
tmux ls	tls	Returns a list of all active tmux sessions.
tmux attach -t	tat	Attach to a named session. Type “tat foo” to attach to a session named “foo”.
tmux new-session -s	tns	Create a new named session. Type “tns foo” to create a session named “foo”.
tmux kill-session -t	tks	Kill a named session. Type “tks foo” to kill session “foo”.

Of all these commands, the `tls` one is most likely to see a lot of use, since it’s a pure status command, and you’ll probably be using customized setups anyway. If there are no sessions open tmux will return

```
failed to connect to server: Connection refused
```

Or something similar. Basically it just means “there aren’t any sessions running, so the server is shut down too.”

Multiplayer Options

Earlier I wrote about how to use tmux to handle pair programming situations by using a server that both you and your counterpart can access. At the time I was flippant and glib about what kind of server you'd actually use, but now it's time to get serious. You may or may not have a server laying around with your name in the `shudders` file. But this is a bright and wonderful day for us hacker types, and there are several options available. I'm going to list three.

Raspberry Pi

The Raspberry Pi[17] is a fabulously inexpensive little machine that makes it easy to get a dedicated Linux box up and running with almost no effort. For $25 you get a board. Add in an SD card, a power cable, and a network cable and you have a perfect little pair programming environment. Raspbian[18], the semi-official distro of the Raspberry Pi, has ssh configured out of the box and tmux is in the apt repository, all ready to go.

There are limitations, of course. A Raspberry Pi needs a public IP address to be of any use if you and your counterpart aren't working on the same subnet. This limitation can be overcome through the use of a VPN, but that brings with it its own host of networking issues. But these are the same issues you would have to overcome in any network sharing scenario, and we have a whole internet of tools to fix them.

Digital Ocean

If you don't want to mess with getting a static IP or working out the firewall rules to let someone else access a machine on your local subnet you can hire a super inexpensive external server to do the job. Digital Ocean[19] provides small, raspberry-pi-sized servers for around $5 a month if you leave them on all the time, or .7¢ an hour if you turn it off when you're not using it.[20] For that money you get an SSD-based virtual server with your choice of operating systems installed, and from there you have a linux machine of your very own out in the cloud. Several people are realizing that they can do the majority of their work on such a setup and as a result can work comfortably from anywhere without worrying that losing their laptop means losing all their work.

The advantage here is that the digital ocean machine you set up has a public IP Address and both players can hit it equally, no NAT tunneling required. As long as you're good about turning the

[17] https://www.raspberrypi.org/

[18] http://www.raspbian.org/

[19] https://www.digitalocean.com

[20] Yes, seven-tenths of one cent USD per hour.

lights out when you're done working together you will probably only pay a dollar or less every month for the privilege.

Vagrant

Vagrant is probably the least acceptable option for sharing your tmux session but I'm including it because a lot of people think it's the way to go. Don't get me wrong, Vagrant is an excellent tool, in fact the next section of the Appendix is all about how you probably *should* be using Vagrant for your individual development work. But it's not well suited for network usage.

The reason for this is simple: It's a virtual machine. To connect to the wider internet Vagrant needs to have a bridged network connection, meaning it gets an IP address of its own on your local network, and from there you have all the issues of getting your counterpart access to your subnet. Additionally, at time of writing this bridged network can only be DHCP, meaning it won't have a stable IP address unless you can do some fancy stuff with the router closest to you. Considering how inexpensive the other options are I would strongly suggest using one of them instead.

Setting Up Your Multiplayer Server

Once you've chosen a server it's time to get it configured for work. First things first: install a version of Linux you like. I don't care what Distro you prefer.

Once you've done that make sure that git is installed (it usually is by default these days) and, of course, tmux. Clone your code base to the server and set up some sane standards as discussed in the Remote Tmux chapter and you're in business.

Vagrant

In the section on installing tmux I mentioned that Windows users are kinda out of luck. Which used to be true. These days there is a very good option that pretty much everyone should take advantage of: Vagrant[21].

A whole tutorial on how to use Vagrant is beyond the scope of this book, but let me lay out the scene for you and then point you at some resources if the scenario I describe sounds good.

Okay, let's say you're a Windows user and you're working on a website for your company. Like a good front-end developer you're using all kinds of fancy javascript build tools[22] and those tools are ever so slightly nasty about running on windows, because they were all written by people who fear that their skin will be burned off if they ever touch a PC. Also, you've picked up this pricelessly witty book about tmux and you'd like to try it, but again, no windows love.

Fortunately you know about Vagrant. You set up a file in your project directory called `.Vagrant-file` and in this file you configure a few things about your dream server. It should have your favorite javascript build tools installed, it should have tmux installed, and most importantly it should have all your source code synced so that you can run all your builds on a linux environment instead of Windows, so the snooty code will stop whining.

Once you get your vagrant configuration worked out you simply type `vagrant up` at a command line and in a few seconds you are presented with a clean, shiny new virtual machine. By default Vagrant runs it's VM's "headless", meaning it doesn't actually give you a little VM window to play with a virtual GUI. Instead you just type `vagrant ssh` and you are given a zero-config ssh session right into that machine. Now you can run `tmux` and like magic you have harnessed the power of multiple sessions running in a single window.

But vagrant gives you even more magic. Because it's a virtual machine you can suspend it, turn your computer off, come back to work the next day, start up again and your tmux session is still intact from the previous day. Congratulations. You have now out-awesomed the snooty Mac/Linux guys on your team. They can't do that on their local environments. The only way they can be as cool as you is to set up their own vagrant instances.

But here's where you get to look even more awesome. Since your vagrant file is in your source code directory you can commit it to the repository. Tell them "oh, don't worry guys, I got it all set up. Just pull my changes and run `vagrant up` and you're set." Nice work! Now your entire team has the full power of vagrant at their fingertips, and it's all because of you.

---

[21] https://www.vagrantup.com/

[22] Notice how I avoided using the actual *names* of any of those tools? That's because they'll all be different in two months.

Seriously, Vagrant strives to give you the best of both worlds. Your source code remains local, so you can use whatever IDE you prefer, but you can move all your server and build stuff to a virtual machine, so it doesn't monkey around with your configuration settings. And you can share a single vagrant file with an entire team, meaning everyone on the team has an **identical** dev environment. It won't get rid of "it works on my box!" *entirely*, but it's a huge step in the right direction.

If this sounds like a great idea, check out the Vagrant website and follow their tutorials. For the most part Vagrant is quite easy to set up and was made for teams. Give it a shot!

Caps Lock Must Die!

The QWERTY keyboard has a lot to answer for. Supposedly the keys are laid out to slow down typists[23], and when the keyboard made the leap from typewriter to computer we had to tack on a bunch of special keys down around the spacebar. But somehow Caps Lock, that most useless of keys, kept its position right on home row. This, friends, is a travesty. Two travesties in fact. But we can fix them both in one fell swoop.

The first travesty is the modifier key ghetto, down around the edges of the keyboard. Look at those poor suckers. `<alt>`, `<ctrl>`, either `<windows>` or `<command>`, based on your keyboard's default OS. Sometimes `<fn>` sneaks in there, sometimes it's a `<menu>` key that not a single person has ever once used. But there all of them are, clear down there, far away from home row, forcing you to do some really awkward things to use them, and as power users we use them *frequently*. Especially `<ctrl>` and `<alt>`.

Which leads us to travesty number two: the `<caps lock>` key, right there, right on home row. A key that serves no purpose in a day and age where we no longer type out telegraph messages to each other:

> ATTENTION FRIENDS STOP REALIZED YOU ARE STILL PUTTING CAPS LOCK KEYS ON KEYBOARDS STOP THIS IS STUPID STOP PLEASE PLEASE PLEASE STOP

At long last, steps are being taken. Google Chromebooks don't have a `<caps lock>` key any more, it's been replaced with a "search" key. Expensive custom keyboards these days include a dip switch that lets you permanently change `<caps lock>` into an additional `<ctrl>` key. And this, friends, is what you need to do.

Even without an expensive keyboard, you can remap the `<caps lock>` key to do something useful with it's life. I strongly recommend mapping it to `<ctrl>`, as this will make tmux commands, not to mention vim commands, much easier to type, by placing all the needed keys right on home row.

So do it today. Take a moment and remap your caps lock. Do it **today**. Thank you.

[23] I realize that the truth of that statement is hotly contested, but stay with me. That's not the point.

About This Book

About the Text

Painless Tmux is by Nathanial Ellsworth Dickson, but you can call me Nate.

It was written using an unwieldy combination of (in alphabetical order):

- Acorn[24] for editing screenshots
- Marked[25] for previewing
- MultiMarkdown Composer[26] for writing
- Scapple[27] for brainstorming
- Scrivener[28] for *everything*
- vim[29] (of course)
- VirtualBox[30] for giving me a Linux environment to play with

It was written on an unlikely number of macs.

About the cover

The cover features an illustration by RetroClipArt[31]. Used with permission. The typefaces used on the cover are:

- **Matchbook Serif** (*title*)
- **Aleo Light** (*subtitle*)
- **Caviar Dreams** (*by line*)

The cover was designed by Nate Dickson using iDraw[32] and Pixelmator[33]

[24] http://flyingmeat.com/acorn/
[25] http://markedapp.com/
[26] http://multimarkdown.com/
[27] http://www.literatureandlatte.com/scapple.php
[28] http://www.literatureandlatte.com/scrivener.php
[29] http://www.vim.org/
[30] https://www.virtualbox.org/
[31] http://www.shutterstock.com/pic-145767344/stock-vector-man-relaxing-with-newspaper-retro-clip-art-illustration.html
[32] http://www.indeeo.com/idraw/
[33] http://www.pixelmator.com/

Special Thanks

I would like to thank, first and foremost, my dear Libbie, for her support, kind words, encouragement, proofreading, spell checking, questions, and love. I'm the luckiest husband ever.