Efficient Rails

Workflow Upgrades for Crafting Rails Apps with Superhuman Speed

version 1.0

Andrew Allen

Efficient Rails

Workflow Upgrades for Crafting Rails Apps with Superhuman Speed

Andrew Allen

Tweet This Book!

Please help Andrew Allen by spreading the word about this book on Twitter!

The suggested hashtag for this book is #EfficientRails.

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=#EfficientRails

Contents

Introduction

This is not your average Rails book.

What it's not

- An introduction. I'll assume that you're familiar with at least the core concepts of both Ruby and Rails.
- A tutorial. At least not in the traditional sense.
- Academic. We won't be discussing much abstract programming theory.

What it is

- Practical. Every part of this book is focused on saving you time and making Rails development more enjoyable.
- Actionable. Learn and implement each solution in just a few minutes.
- Non-linear. You don't have to read cover-to-cover to get something out of this book.

Who it's for

This book is intended for someone who has at least some experience with Rails. There are tons of great resources out there to learn the basics, but once you hit a base level of proficiency, your options start to thin. This book aims to fill that space.

Efficient Rails is for someone who uses Rails frequently. It doesn't matter if you're junior or senior, the techniques in this book will save you time on common tasks. If you're using Rails every day, you'll get some serious productivity gains.

As your productivity rises, you'll level up as a Rails developer, enabling you to take on more clients, impress your team by getting that product out the door faster, or finally shed that 'Junior' part of your title.

How to use it

The book is divided into 3 parts.

The first part covers the tools you're using every day and how to get the most out of them. This includes the Terminal, Git and the Rails Console.

The second part covers the core aspects of the Rails framework and contains recipes for accomplishing common tasks.

The final part covers techniques that will speed up your testing and debugging workflows.

Pick a chapter that relates to something youâ€™re working on currently and open up to it. Within that chapter are between 10-12 sections in no particular order. Each section presents a problem and one or more solutions. Read the problem. If it describes a pain point you have or have had, then read the solution.

Book updates

As developers, you know nothing stays the same for long. New gems will be released, better ways of accomplishing repetitive tasks will be discovered. As the world of Rails changes, so too will this book, and you'll be the first to know. These updates will be sent to you automatically as soon as they're available.

I'll also be incorporating feedback into future revisions. To submit feedback on a particular section, click one of the links at the end of the section. You'll have a chance to provide more feedback after clicking. Of course, you can also send an email to andrew@EfficientRails.com[1].

With that said, there will be typos. There will be incomplete thoughts and run-on sentences. There will be more chapters and sections to come. But I'm confident you'll get a ton of value out of what's here.

> If you are not embarrassed by the first version of your product, you've launched too late.
>
> – Reid Hoffman

[1] mailto:andrew@efficientrails.com

Part I: The Tools

In this first part, we'll be covering workflow upgrades for the tools you use everyday to develop Rails applications.

We'll start with the most-used application on my computer, the Terminal. If you spend enough time every day using something, even small improvements can provide big returns over time.

Following that, we'll touch on Git. Obviously, Git is a gigantic subject, so we won't be covering all of it, and it won't be an introduction to Git. Instead, I'll walk through 10 upgrades that I find particularly helpful in my day-to-day workflow as a Rails developer.

To round out this part, we'll cover the Rails console, a fantastic tool for exploring and experimenting in your Rails application. I'll show you all kinds of ways to get more out of this essential tool.

In Part II, we'll dive into the Rails codebase itself.

Chapter 1: The Terminal

When I first started learning Rails several years ago after teaching myself web development with PHP and studying C# in school, one thing in particular stood out to me: you spend a lot of time in the terminal when building with Rails. With C#, you get the advantages of a full-featured IDE (Visual Studio), and with PHP, you wind up using GUI tools like MAMP and phpMyAdmin to configure your environment (this was before Laravel).

Now, I use the terminal for just about everything. It's my text editor (Vim), filesystem (Ranger), Git client (Tig) and more. In fact, I'm writing this in the terminal as we speak!

In this chapter, we'll talk about configuring your terminal to make it both beautiful as well as useful. I'll also introduce you to Zsh and its advantages over Bash, present solutions to common problems and show you some cool Rails-related tricks. Not every section will be specific to Rails, but I guarantee each one will be relevant for Rails developers.

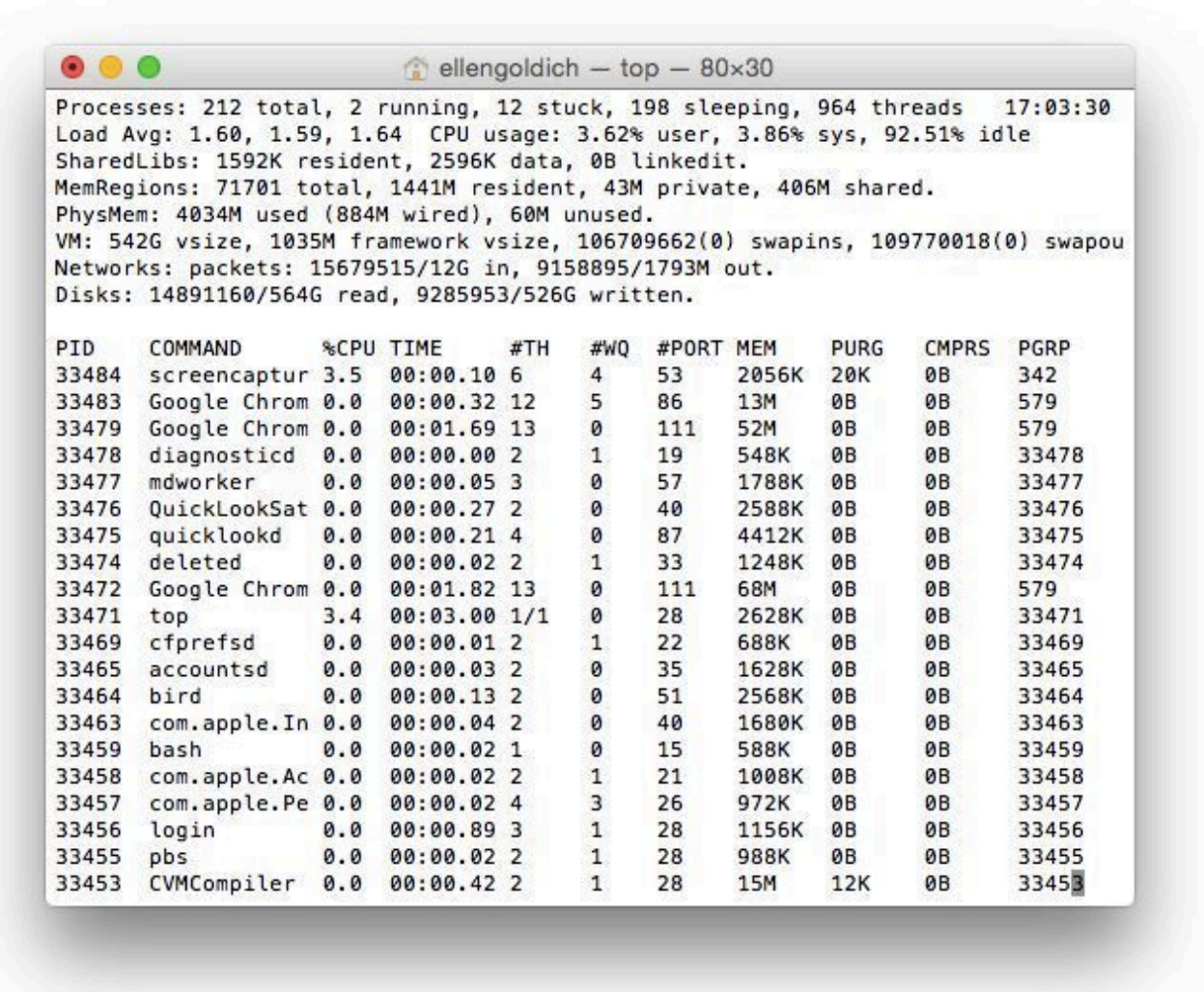
A few of these sections will be admittedly Mac focused, but most will apply to other platforms as well.

A Better Terminal

Problem

As Rails developers, we spend a lot of time in the terminal. Whether it's running rake tasks, committing code with Git, running tests, or even editing code with Vim. The Terminal will be one of the most-used applications on your computer.

This is the default Terminal in OS X:



OS X's (ugly) default terminal

Enough said.

Solution

Let's give your terminal an upgrade.

iTerm2

While the default Terminal app that ships with OS X will get the job done, iTerm2 is a vastly superior product, boasting features you never even knew you needed.

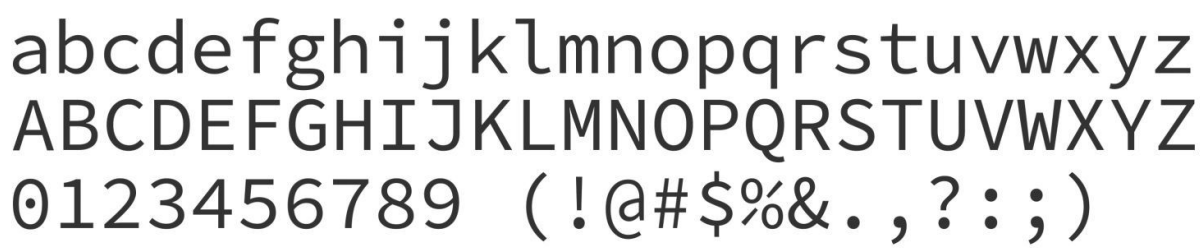
Here are some highlights:

- Split panes
- Hotkey window
- Search result highlighting
- Mouseless copy
- Paste history
- Instant replay (lets you replay your terminal session)
- Multiple smart profiles

That barely scratched the surface. iTerm2 is packed with features. Check out iTerm's homepage[2].

Font

After trying several different fonts in my terminal, I've settled on Source Code Pro from Adobe. You can download it from Font Squirrel[3].



Source Code Pro font

Settings:

- Font size: 13 - 14 pt
- Typeface: Regular (or Light if you prefer)
- Ensure anti-aliased is true

Other Font Options:

- Inconsolata[4]
- Menlo[5]

[2] https://www.iterm2.com
[3] http://www.fontsquirrel.com/fonts/source-code-pro
[4] http://www.levien.com/type/myfonts/inconsolata.html
[5] Menlo is installed on OS X by default.

Colors

Most people like Solarized (Dark), and while that's a great color scheme, I'd recommend Smyck[6]. It's similar to Solarized, but is slightly more readable for me.

The Smyck Color Scheme

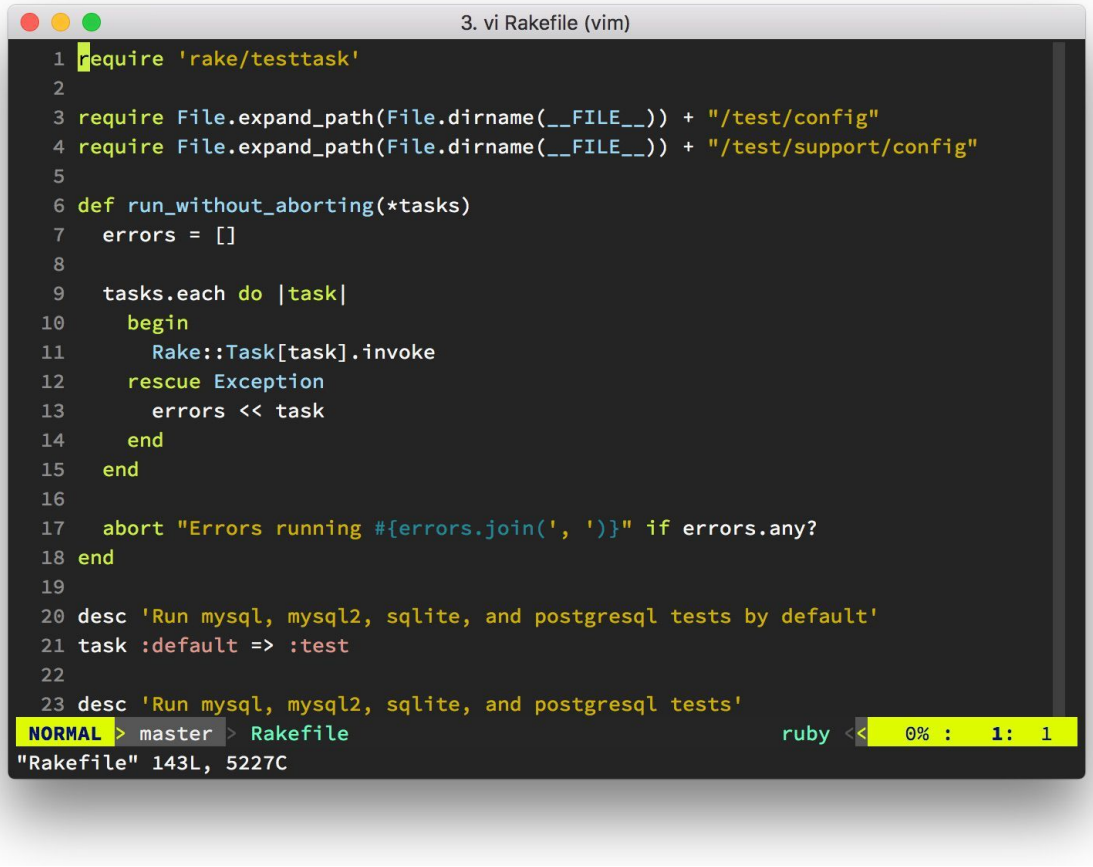
Other Color Options:

- Solarized[7]
- Molokai[8]

[6] http://color.smyck.org/
[7] http://ethanschoonover.com/solarized
[8] https://github.com/tomasr/molokai

The End Result

Now this is a terminal I could spend some time in:



iTerm2 with Source Code Pro and Smyck color palette

Zsh

Problem

Bash on OSX is old (from 2007!). And it's not just old, but lacking in functionality. Most have come to accept the limitations of Bash as the limitations of what's possible on a command line interface, but there's another way.

Solution

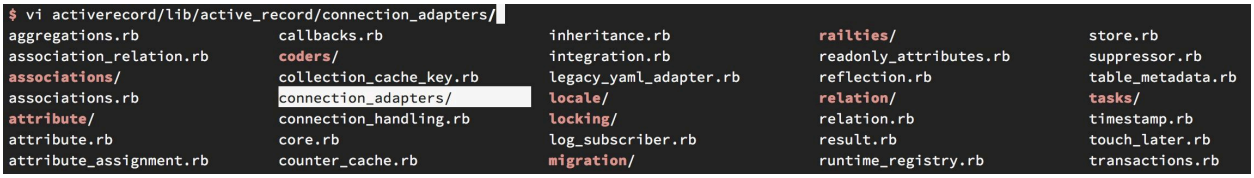
Sure you could upgrade Bash with Homebrew to bring its version into this decade, but why not take the opportunity to explore alternatives?

Z Shell, or Zsh, is another shell that comes with features way beyond the scope of what Bash is capable of. But at the same time, Zsh is similar to Bash, so it will feel very familiar.

We'll cover a few features of Zsh that will improve your Rails workflow in depth in the following sections.

In the meantime, here's a highlight reel of what Zsh can do:

Interactive Tab Completion



Tab completion with Zsh

Zsh lets you pick between possible matches by continuing to press <tab>, or using the arrow keys.

Partial Path Completion

To get to `some/nested/path`, you no longer need to `cd some`, `cd nested`, `cd path`. You can type out partial names of paths, and Zsh will fill in the gaps when you push tab.

```
1 $ cd d/te/app/v/p<tab> #=> cd dev/test_blog/app/views/posts
```

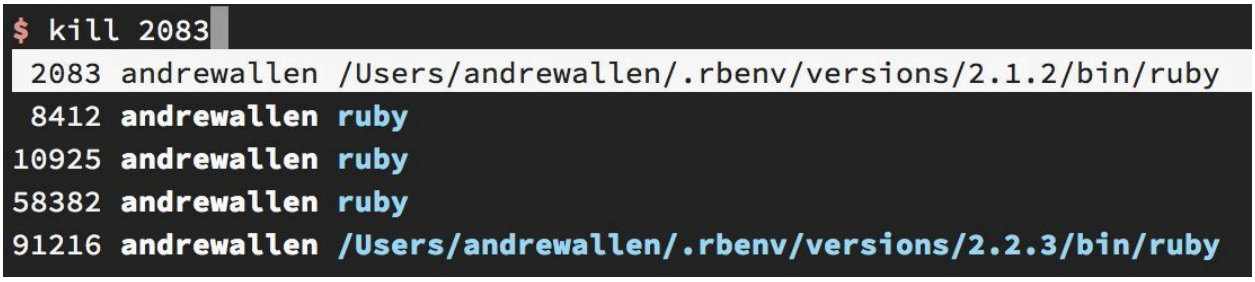
If more than one path matches, Zsh will ask to you to pick between the possible matches.

Autocomplete Kill, SSH, SCP Commands

Start typing the process name you want to kill:

```
$ kill ru<tab>
```

And Zsh will show you the matching process, letting you pick between them, and will autofill the process number.



Zsh kill autocompletion

Zsh provides similar functionality for the ssh and scp commands.

Path Replacement

Zsh can swap segments of the path name:

```
$ pwd
/Users/andrewallen/dev/test_blog/app/assets

$ cd test_blog another_rails_app
~/dev/another_rails_app/app/assets

$ pwd
/Users/andrewallen/dev/another_rails_app/app/assets
```

While that's less than 1% of what Zsh offers, hopefully that gives you a taste for what it's capable of.

Install

Zsh can be installed with Homebrew:

```
$ brew install zsh zsh-completions
```

Once installed, change your default shell to Zsh:

```
$ chsh -s $(which zsh)
```

You'll need to start up a new terminal session for the shell change to take effect. Once you do, make sure that you're running Zsh with:

```
$ echo $SHELL
```

Configuring Zsh

Zsh configuration is managed with a `.zshrc` file in your home directory. If you have existing configuration options specified in a `.bashrc` file, you'll need to copy them over to a `.zshrc` file. The good news is, the syntax for configuring Zsh is very similar to Bash, so you shouldn't need to change much from your Bash configuration.

Oh My Zsh

Problem

It takes effort to remember to define aliases for the shortcuts you come up with. There's also a good deal of creativity that goes into creating useful and well-named aliases, functions and plugins. Exploring other peoples' dotfiles is a good source of inspiration:

- https://dotfiles.github.io/[9]
- https://github.com/thoughtbot/dotfiles[10]
- https://github.com/mathiasbynens/dotfiles[11]

But adding these to your dotfiles in a sane, well-organized way can be daunting.

Solution

Oh My Zsh is a community-driven configuration framework for Zsh. It gives you drop-in solutions for many common Zsh shortcuts like Rails, Rake, Git, Bundler and much more. You can explore the full list of plugins[12] and you can read about them in the wiki[13].

Oh My Zsh also comes with preconfigured themes for your prompt. For example, I use the 'ys' theme which gives me contextual information about where I am, and also shows the status of my git repo, if I'm in one.

The ys theme from Oh My Zsh

Install

Oh My Zsh is installed with Curl:

[9] https://dotfiles.github.io/
[10] https://github.com/thoughtbot/dotfiles
[11] https://github.com/mathiasbynens/dotfiles
[12] https://github.com/robbyrussell/oh-my-zsh/tree/master/plugins
[13] https://github.com/robbyrussell/oh-my-zsh/wiki/Plugins

```
$ sh -c "$(curl -fsSL https://raw.github.com/robbyrussell/oh-my-zsh/master/tools\
/install.sh)"
```

Once installed, you can enable plugins by adding them to the .zshrc file:

```
plugins=(git bundler osx rake ruby)
```

And you can change your theme by editing the `ZSH_THEME` environment variable in .zshrc:

```
ZSH_THEME="ys"
```

Alternatives

In addition to Oh My Zsh, Prezto is frequently recommended as a Zsh configuration framework. It's touted as a more lightweight alternative to Oh My Zsh.

In my experience, Oh My Zsh has served me well and Prezto hasn't displayed anything groundbreaking enough to get me to switch. However, I would be remiss if I didn't mention it.

In this chapter, I'll reference Oh My Zsh plugins, but you can assume in most cases there is an equivalent plugin for Prezto.

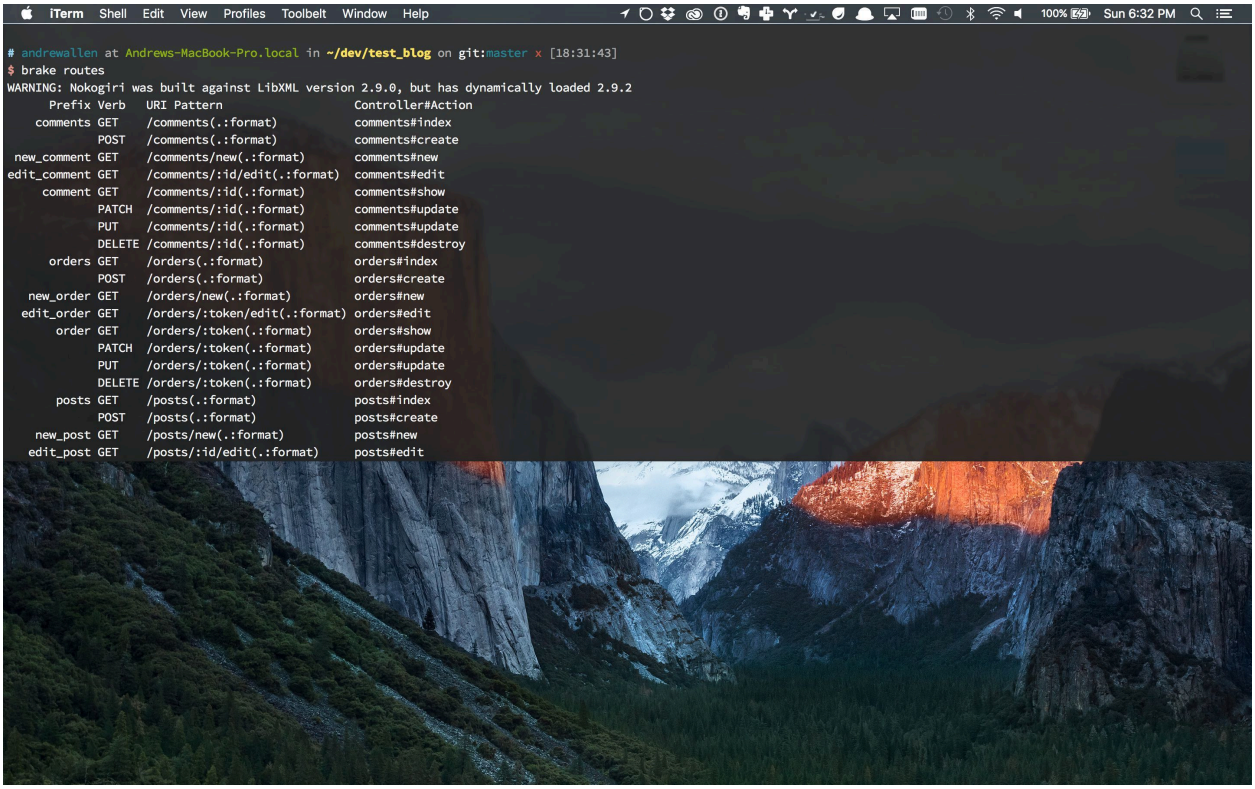
Visor

Problem

Once the terminal becomes your primary method of interacting with your computer, you'll want it to be easily accessible to fire off quick commands. However, if you have several desktop spaces (I know I do), you'll find yourself flailing trying to figure out where you left your terminal.

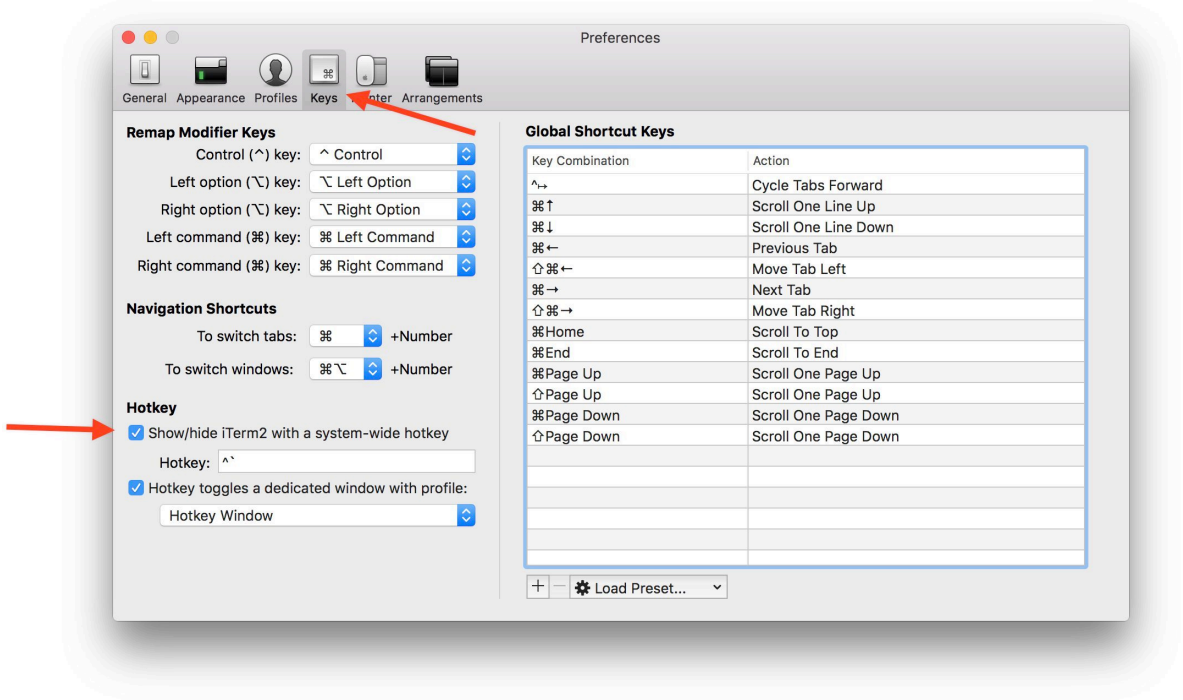
Solution

With iTerm2's Visor feature, a terminal is never more than a keystroke away.



iTerm's System-Wide Visor

To enable the visor, open iTerm's settings and under the 'Keys' pane, enable 'Show/hide iTerm2 with a system-wide hotkey'.



Enabling the Visor

You'll want to pick a key combination that's easy for you to hit. I use Ctrl+‘, but I have Caps Lock mapped to Control, so I can hit it by moving my left hand slightly to the left. You'll be using this hotkey a lot, so take some time to make sure it's easy to remember and doesn't require any finger acrobatics to hit.

Global Aliases

Problem

Part of what makes the command line so powerful is the ability to chain commands together. There are handful of very commonly used terminal commands that we'd like to alias, for example: `| grep` to search within the output or `| pbcopy` to copy the output to the clipboard.

There's just one problem: Bash doesn't allow us to chain aliases to the end of a command.

Solution

Zsh has a feature called Global Aliases that allows us to define aliases that can be used anywhere within a command. We can define aliases like:

```
alias -g G='| grep'
alias -g H='| head'
alias -g T='| tail'
alias -g CP='| pbcopy'
```

And then we can chain these aliases within our commands:

```
$ rake routes G post
$ cat ~/.ssh/id_rsa.pub CP
$ cat log/development.log T
```

I use uppercase aliases for global commands to help me distinguish them mentally and visually.

It might seem like a small improvement, but it's really pretty powerful. Not to mention, the pipe key is a pain to type.

To get a handful of these global aliases, along with a lot of other great aliases (global and otherwise), check out the Oh My Zsh common-aliases plugin[14].

[14] https://github.com/robbyrussell/oh-my-zsh/blob/master/plugins/common-aliases/common-aliases.plugin.zsh

The Art of Command Chaining

In 1986, Donald Knuth, famed author of "The Art of Computer Programming" and professor emeritus at Stanford, was given a challenge: print the n most often used words within a document and how often they occur.

His finished program was 10 printed pages of Pascal, beautifully commented, and included a novel data structure.

Douglass McIlroy, developer of Unix pipelines and author of many immortal Unix programs like diff, sort and tr, responded with a 6 line shell script that used command chaining to accomplish the exact same thing.

McIlroy's 6-Line Shell Script

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

As McIlroy demonstrated 30 years ago, command chaining is a powerful and efficient concept, one that we can still leverage today. We don't need to rewrite scripts when we can piece together smaller programs to achieve the same effect.

Zsh embraces the art of command chaining by allowing us to define aliases that can be used anywhere in a command, not just at the beginning. This allows us to write chainable aliases and reap the benefits.

Command Auto Completion

Problem

Unless you use a command every day, you might not remember exactly how to use it. Even if you use it everyday, I doubt you remember *all* the various options and arguments.

Of course you can run `man` to get the manual, or you can Google what you're looking to do and end up on Stack Overflow. But that takes you out of context.

Solution

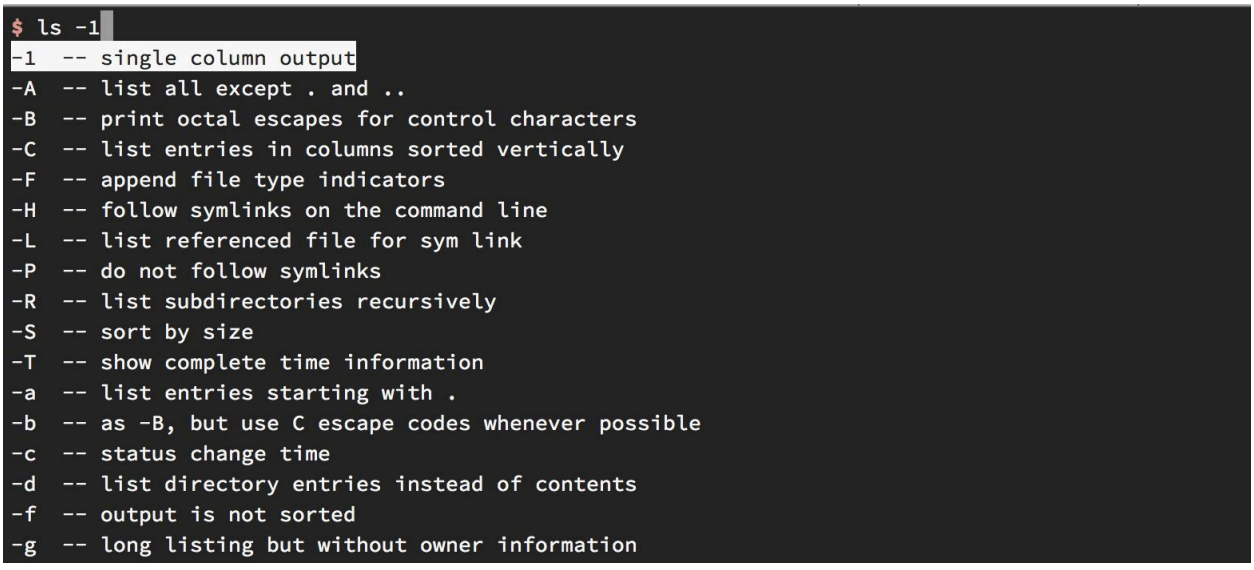
Zsh autocompletion is super powerful, and extends to command options as well.

To see what options a command takes, type the command, followed by an empty flag (-), and then type <tab>.

```
$ ls -<tab>
```

This will list out the matching options. To go ahead and select one, hit <tab> twice and then select with the arrow keys, or by continuing to press <tab>.

```
$ ls -<tab><tab>
```



Command Auto Complete for ls

This also works for tools like Git

```
$ git add
add         -- add file contents to the index
bisect      -- find by binary search the change that introduced a bug
branch      -- list, create, or delete branches
checkout    -- checkout a branch or paths to the working tree
clone       -- clone a repository into a new directory
commit      -- record changes to the repository
diff        -- show changes between commits, commit and working tree, etc
fetch       -- download objects and refs from another repository
grep        -- print lines matching a pattern
init        -- create an empty Git repository or reinitialize an existing one
log         -- show commit logs
merge       -- join two or more development histories together
mv          -- move or rename a file, a directory, or a symlink
pull        -- fetch from and merge with another repository or a local branch
push        -- update remote refs along with associated objects
```

Command Auto Complete for git

And even for services like Heroku:

```
$ heroku addons
account:confirm_billing  -- Confirm that your account can be billed at the end of the month
addons                   -- list installed addons
addons:add               -- install an addon
addons:downgrade         -- downgrade an existing addon
addons:list              -- list all available addons
addons:open              -- open an addon's dashboard in your browser
addons:remove            -- uninstall an addon
addons:upgrade           -- upgrade an existing addon
apps                     -- list your apps
apps:create              -- create a new app
apps:destroy             -- permanently destroy an app
apps:info                -- show detailed app information
apps:open                -- open the app in a web browser
apps:rename              -- rename the app
auth:login               -- log in with your heroku credentials
auth:logout              -- clear local authentication credentials
config                   -- display the config vars for an app
config:pull              -- pull heroku config vars down to the local environment
```

Command Auto Complete for heroku

Git and Heroku auto completion require enabling the respective plugins in Oh My Zsh

Autocomplete Git Branches

Problem

If you're working on a large team and using Git Flow, you'll no doubt end up navigating between and performing operations on numerous Git branches. These branches can have long, unwieldy names like `feature/add-oauth-for-social-login`. Who wants to type that whole branch name out every time?

You can use some tricks like (#section-git-toggle), but that only gets you so far.

Solution

Assuming you've installed Zsh to replace Bash as your default shell, you'll get Git autocompletion by default.

This will autocomplete not only Git commands:

```
# andrewallen at Andrews-MBP in ~/dev/rails on git:master o [8:52:10]
$ git
add        -- add file contents to the index
bisect     -- find by binary search the change that introduced a bug
branch     -- list, create, or delete branches
checkout   -- checkout a branch or paths to the working tree
clone      -- clone a repository into a new directory
commit     -- record changes to the repository
diff       -- show changes between commits, commit and working tree, etc
fetch      -- download objects and refs from another repository
grep       -- print lines matching a pattern
```

Zsh autocompletes Git commands

But also Git branches:

```
# andrewallen at Andrews-MBP in ~/dev/rails on git:master o [8:52:10]
$ git checkout v4.
v4.0.0         v4.0.11        v4.0.6.rc2     v4.1.10.rc1    v4.1.2         v4.1.8         v4.2.1.rc1
v4.0.0.beta1   v4.0.11.1      v4.0.6.rc3     v4.1.10.rc2    v4.1.2.rc1     v4.1.9         v4.2.1.rc2
v4.0.0.rc1     v4.0.12        v4.0.7         v4.1.10.rc3    v4.1.2.rc2     v4.1.9.rc1     v4.2.1.rc3
v4.0.0.rc2     v4.0.13        v4.0.8         v4.1.10.rc4    v4.1.2.rc3     v4.2.0         v4.2.1.rc4
v4.0.1         v4.0.13.rc1    v4.0.9         v4.1.11        v4.1.3         v4.2.0.beta1   v4.2.2
v4.0.1.rc1     v4.0.2         v4.1.0         v4.1.12        v4.1.4         v4.2.0.beta2   v4.2.3
v4.0.1.rc2     v4.0.3         v4.1.0.beta1   v4.1.12.rc1    v4.1.5         v4.2.0.beta3   v4.2.3.rc1
v4.0.1.rc3     v4.0.4         v4.1.0.beta2   v4.1.13        v4.1.6         v4.2.0.beta4   v4.2.4
v4.0.1.rc4     v4.0.4.rc1     v4.1.0.rc1     v4.1.13.rc1    v4.1.6.rc1     v4.2.0.rc1     v4.2.4.rc1
v4.0.10        v4.0.5         v4.1.0.rc2     v4.1.14        v4.1.6.rc2     v4.2.0.rc2     v4.2.5
v4.0.10.rc1    v4.0.6         v4.1.1         v4.1.14.rc1    v4.1.7         v4.2.0.rc3     v4.2.5.rc1
v4.0.10.rc2    v4.0.6.rc1     v4.1.10        v4.1.14.rc2    v4.1.7.1       v4.2.1         v4.2.5.rc2
```

Zsh autocompletes Git branches

After typing a few letters of the branch you want, press tab and Zsh will show you a list of matching branches. If you keep pressing tab, you can select one of the options to complete your command.

You can also press tab without typing any letters to see all possibilities (though it may be a large list).

If you're using Oh My Zsh, you can enable the gitfast plugin for more performant autocompletion. To install the gitfast plugin, simply edit your .zshrc file and add it to the list of plugins loaded:

```
plugins=(git gitfast bundler osx rake ruby)
```

Don't forget to `source` your .zshrc file, or start another shell for the new plugin changes to take effect.

Fasd

Problem

Dealing with navigation from the command line can be a big barrier to getting things done from the terminal. I find typing out paths is one of the most time-consuming aspects of working in the terminal.

Innovations like tab completion help a lot, but it can still be annoying to think in terms of directories in order to get tab completion to trigger.

You have a number of different projects on your machine. In addition to several Rails apps for work, surely you have some side projects or open source repos that you spend time in. Navigating between them and all the other files on your system quickly becomes a headache.

Solution

Fasd is a very smart solution to dealing with frequently used directories and files. It's very flexible and has a ton of functionality, but I'll just highlight a couple features that have had the biggest impact on my workflow.

For more in depth info, check out the Readme[15].

Changing Directories

The `z` alias provided by fasd jumps to the best-matching directory. If you tab complete and there are multiple matches, it will allow you to choose. Otherwise, it just goes with what it thinks the best match is based on what you typed and where you've been recently.

I use this all the time to navigate the labyrinth of directories on my machine:

```
1 $ z proj #=> cd /Users/you/work/my_rails_project
```

Using within Other Commands

The true power of fasd starts to become apparent once you realize it can be used *within* other commands.

For example, if I'm at the command line and need to run a specific spec file, I'll use fasd to conjure up the path to that file. In this case, I use the `f` alias, which limits the search to files, and excludes directories.

[15] https://github.com/clvv/fasd

```
$ rspec `f user_can` #=> rspec ../spec/features/user_can_register_spec.rb
```

Vim

If you install fasd with Oh My Zsh (see below for installation instructions), you'll get a `v` alias, which is short for `alias v='f -e vim'`. This does a fasd file search and passes the result into vim.

I use this when I'm far away from my home directory and I want to edit config files:

```
$ v zshl #=> vim /Users/you/.zshrc.local
```

Install

Fasd can be installed with Homebrew:

```
$ brew install fasd
```

Once installed, it needs to be initialized in your shell config.

If you have Oh My Zsh installed, you can simply enable the fasd plugin:

```
plugins=(git git-flow fasd rake rails)
```

This comes with a couple additional aliases and some smart caching to make your shell start faster.

If you don't have Oh My Zsh, you can add the following to your shell config file for basic initialization:

```
eval "$(fasd --init auto)"
```

After installing, you'll need to go about your normal workflow for a bit to get it seeded with data. As you visit directories and edit files, it will build up a database of things you interact with frequently.

Aliases for Common Rails Commands

Problem

Rails development requires lots of typing at the command line. This means lots of wasted keystrokes on commands you're typing out multiple times a day. While you might have a handful of aliases for the worst offenders, there might be some time-saving aliases you've overlooked.

Solution

The Rails plugin for Oh My Zsh[16] includes a lot of great upgrades. Even if you're not using Oh My Zsh, it's worth taking a glance at for inspiration.

For starters, it wraps your `rails` and `rake` commands in a function that will attempt to use the binaries located in your application's `bin` directory. This means no more version mismatches if you have multiple projects running different versions of Rails and no more `bundle exec rake ...` to fix this.

It also comes packed with lots of intuitive aliases for common Rails commands.

Here are some examples from the Oh My Zsh Rails plugin:

```
# Tail your log files
alias devlog='tail -f log/development.log'
alias prodlog='tail -f log/production.log'
alias testlog='tail -f log/test.log'

# Global aliases for environment variables
alias -g RED='RAILS_ENV=development'
alias -g REP='RAILS_ENV=production'
alias -g RET='RAILS_ENV=test'

# Rails aliases
alias rc='rails console'
alias rg='rails generate'
alias rd='rails destroy'
alias rgm='rails generate migration'
alias rs='rails server'

# Rake aliases
alias rdm='rake db:migrate'
alias rdms='rake db:migrate:status'
```

[16] https://github.com/robbyrussell/oh-my-zsh/blob/master/plugins/rails/rails.plugin.zsh

```
alias rdr='rake db:rollback'
alias rdc='rake db:create'
alias rds='rake db:seed'
alias rdd='rake db:drop'
alias rdrs='rake db:reset'
alias rdtc='rake db:test:clone'
alias rdtp='rake db:test:prepare'
alias rdmtc='rake db:migrate db:test:clone'
alias rr='rake routes'
alias rrg='rake routes | grep'
```

Install

The Rails plugin for Oh My Zsh can be installed by adding it to your ~/.zshrc file:

~/.zshrc

```
plugins=(git bundler osx rake ruby rails)
```

Then, start a new terminal window, or reload the current session by sourcing the modified file:

```
$ source ~/.zshrc
```

Binstubs > Bundle Exec

Problem

If you have multiple projects that use different versions of Rails or other gems installed, you may find yourself needing to prepend your commands with `bundle exec`.

For example, if you run `rspec` alone, you might get the following message:

The error when RSpec has dependency conflicts

```
$ rspec
WARN: Unresolved specs during Gem::Specification.reset:
      minitest (~> 5.1)
      rails-html-sanitizer (>= 1.0.2, ~> 1.0)
      rake (>= 0.8.7)
      activemodel (>= 3.2.0)
WARN: Clearing out unresolved specs.
Please report a bug if this causes problems.
/Users/andrewallen/.rbenv/versions/2.2.3/lib/ruby/gems/2.2.0/gems/bundler-1.11.2\
/lib/bundler/runtime.rb:34:in `block in setup': You have already activated activ\
esupport 5.0.0.beta1, but your Gemfile requires activesupport 4.2.5.1. Prependin\
g `bundle exec` to your command may solve this. (Gem::LoadError)
```

While you can certainly start prepending all of your commands with `bundle exec`, it feels like there must be a more complete solution. Sure, you could dig into the docs for your Ruby version manager and maybe tweak the gemsets available on your machine. But this is fragile and only works on the machine you configure it. If you change computers frequently or share the repository with a team, everyone will need to figure out their own solution.

Solution

Rails 4 introduced the concept of Binstubs, which are Ruby-executable files that facilitate running gems with certain dependencies.

So for example, if we wanted to create a Binstub for RSpec, we'd run the following:

```
$ bundle binstubs rspec-core
```

This will generate a file, `bin/rspec`, that when executed, runs RSpec with the correct environment and dependencies. This file can be checked into Git so that everyone has access to it.

Binstub Aliases

But now we're stuck running `bin/rspec` when we just wanted to run `rspec`. We can solve this with some Zsh (or Bash) scripting:

```
function _rspec_command () {
  if [ -e "bin/rspec" ]; then
    bin/rspec $@
  else
    command rspec $@
  fi
}

alias rspec='_rspec_command'
compdef _rspec_command=rspec
```

This function checks to see if we have a Binstub for RSpec in our working directory and uses that. If not, it just falls back to the normal behavior. We alias this function to `rspec` so that RSpec will automatically try to use a Binstub if present. Oh My Zsh ships with similar functions for Rake and Rails.

Dotfiles

These functions and aliases can be added to your Dotfiles which can be backed up remotely and shared so that you can use them on multiple devices, and other teammates can use them as well.

Keep your Todo List in Rails

Problem

Often, we discover an edge case, or know we want to improve some aspect of our code, but don't have the time or energy to get to it right now. Hopefully we document these notes somewhere, but even if we do document them, there are some problems.

If we use an external task manager, these tasks could get out of sync with the actual codebase. And unless you leave a file and line number, you might not remember where exactly the issue is.

You might leave comments in your code, but those will inevitably be forgotten and ignored.

Solution

Rails comes with a hidden feature to help with managing your list of to-dos.

As you're writing code, you can leave one of these three notes:

Notes you can leave in your Rails code

```
1 #TODO: handle edge case where x happens
2 #FIXME: this breaks under x condition
3 #OPTIMIZE: this takes 5 database queries, but should only take 1
```

And then, running `rake notes` (or `rn` if you're making use of the aliases in the Aliases for Common Rails Commands section) will enumerate every note you've left in your code:

```
$ rake notes
app/models/order.rb:
  * [ 8] [FIXME] this breaks under x condition

app/models/post.rb:
  * [ 4] [TODO] handle edge case where x happens

app/views/posts/index.html.erb:
  * [15] [OPTIMIZE] this takes 5 database queries, but should only take 1
```

Rake notes shows annotations marked TODO, FIXME or OPTIMIZE

While you still have to take the initiative to go back and refactor your code, this will at least give a quick overview of what's left to be done in your application and where to find it.

Don't like the default TODO, FIXME and OPTIMIZE? You can also enumerate custom annotations:

```
1 $ rake notes:custom ANNOTATION=NOTE
```

HTTPie > Curl

Problem

Curl has some very obtuse syntax for such a useful and popular tool.

I dare you to type a complex Curl command from memory without Googling. How do you set it to PUT? How do you pass params again? And what about headers?

Curl syntax

```
$ curl -X PUT -d "first_name=John&last_name=Smith" -H "X-API-Token: 123" example\
.org
```

And once you do get a response back, it's ugly and hard to read:

```
$ curl localhost:4000/posts.json
[{"id":1,"title":"Bushwick Migas Cleanse Xoxo Tilde Tumblr","author":"Olga Veum","body":"Iphone pinterest pol
aroid twee. Gluten-free fixie mlkshk banh mi. Godard deep v schlitz venmo. Forage portland squid cleanse iron
y cronut carry selfies.","tags":["umami","chicharrones"],"url":"http://localhost:4000/posts/1.json"},{"id":2,
"title":"Stumptown Cronut I Phone Venmo","author":"Glenna Shanahan","body":"Bicycle rights meditation art par
ty tattooed aesthetic kickstarter. Roof xoxo microdosing aesthetic literally. +1 viral gentrify ennui flexita
rian narwhal. Ennui keytar kale chips helvetica brooklyn. Umami deep v roof.\r\nPark fap pitchfork. Gentrify
master lomo 90's. Irony keffiyeh kinfolk fap bushwick marfa yuccie cleanse. Pour-over truffaut yuccie mlkshk
master yolo. Gluten-free green juice flannel austin heirloom kale chips slow-carb disrupt.","tags":["banjo"],
"url":"http://localhost:4000/posts/2.json"},{"id":3,"title":"Kickstarter Tofu Synth Godard","author":"Nat Jas
t Jr.","body":"Tote bag tousled meh crucifix blue bottle. Distillery crucifix asymmetrical. Seitan viral hamm
ock celiac keytar deep v loko kogi. Tousled blog meh semiotics.\r\nNeutra jean shorts gentrify pbr\u0026b 3 w
olf moon food truck blog deep v. Narwhal shoreditch before they sold out. Occupy viral park tote bag. Gluten-
free pour-over cardigan 90's trust fund vinyl. Vinyl shoreditch master.\r\nBrunch sriracha etsy lumbersexual
tattooed sartorial blog. Lumbersexual post-ironic bicycle rights chicharrones ethical chillwave sartorial vic
e. Chicharrones 8-bit taxidermy forage. Sartorial photo booth put a bird on it chicharrones actually. Lumbers
exual bitters synth green juice you probably haven't heard of them humblebrag vinegar everyday.","tags":["XOX
O"],"url":"http://localhost:4000/posts/3.json"},{"id":4,"title":"Ethical Neutra Listicle Cray Pour Over Hammo
ck Hashtag","author":"Mrs. Domingo Hickle","body":"Wayfarers direct trade church-key dreamcatcher. Waistcoat
chia fanny pack marfa ugh pour-over. Vice ramps keytar banh mi five dollar toast. Yr kale chips offal william
```

Illegible output from Curl

Solution

HTTPie is a Curl replacement that includes the following features:

- Easy to remember commands
- Header/Response info
- Syntax highlighting

HTTPie syntax

```
$ http PUT example.org X-API-Token:123 first_name=John last_name=Smith
```

And the output is much easier to interpret:

```
$ http localhost:4000/posts.json
HTTP/1.1 200 OK
Cache-Control: max-age=0, private, must-revalidate
Connection: Keep-Alive
Content-Length: 5184
Content-Type: application/json; charset=utf-8
Date: Thu, 10 Dec 2015 16:34:25 GMT
Etag: W/"2c5c670c97ac6191d2e5f4581a5a65d7"
Server: WEBrick/1.3.1 (Ruby/2.2.0/2014-12-25)
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Request-Id: a5426ede-5707-4eb2-b57c-4b5b7dfc3744
X-Runtime: 0.031233
X-Xss-Protection: 1; mode=block

[
    {
        "author": "Olga Veum",
        "body": "Iphone pinterest polaroid twee. Gluten-free fixie mlkshk banh mi. Godard deep v schlitz venm
o. Forage portland squid cleanse irony cronut carry selfies.",
        "id": 1,
        "tags": [
            "umami",
            "chicharrones"
        ],
        "title": "Bushwick Migas Cleanse Xoxo Tilde Tumblr",
        "url": "http://localhost:4000/posts/1.json"
    },
    {
        "author": "Glenna Shanahan",
        "body": "Bicycle rights meditation art party tattooed aesthetic kickstarter. Roof xoxo microdosing ae
sthetic literally. +1 viral gentrify ennui flexitarian narwhal. Ennui keytar kale chips helvetica brooklyn. U
mami deep v roof.\r\nPark fap pitchfork. Gentrify master lomo 90's. Irony keffiyeh kinfolk fap bushwick marfa
 yuccie cleanse. Pour-over truffaut yuccie mlkshk master yolo. Gluten-free green juice flannel austin heirloo
m kale chips slow-carb disrupt.",
        "id": 2,
        "tags": [
            "banjo"
        ],
```

Pretty-printed output from HTTPie

Make sure to enable the Oh My Zsh plugin: httpie for autocompletion.

Install

HTTPie can be installed with homebrew:

```
$ brew install httpie
```

HTTPie Authenticated Routes

Problem

Most of the time, the routes we might want to examine with HTTPie are behind some kind of authentication layer.

By default, HTTPie treats each request independently. It has no notion of users or sessions. So how then can we "log in" to our Rails app with HTTPie in order to access and inspect these interesting routes?

Solution

HTTPie supports named session and cookie handling.

If your authentication is configured to allow direct authentication (without checking the Rails CSFR tokens), maybe to allow mobile apps to authenticate, you can use:

```
$ http --session=user1 POST localhost:3000/api/V1/users/sessions email=me@exampl\
e.com password=letmein
```

Now, by referencing that session name, you be able to make authenticated requests.

```
$ http --session=user1 localhost:3000/secret_documents/index.json
```

Devise

What if you're using something like Devise that doesn't support authentication via direct request and instead expects users to log in from the website?

Rails protects applications from cross site forgery requests by requiring form submissions to include a special token. This feature is there for a reason: to protect your users. So you probably don't want to disable it, if you can help it.

We can, however, use the Chrome developer tools to see what our authenticated requests look like, and replicate those with HTTPie. If we inspect any authenticated request in the Network tab of the Chrome developer tools, we'll see a header that gets set on the request called "Cookie".

As it turns out, HTTPie stores sessions as json files in `~/.httpie/sessions/<your host>/user1.json`. We can simply copy the contents of this header into an HTTPie session file.

~/.httpie/sessions/localhost_3000/user1.json

```
{
  "headers": {
    "Accept": "application/json",
    "Cookie": "<contents of Cookie Header>"
  }
}
```

Now, we can make authenticated requests with HTTPie by referencing that session file:

```
$ http --session=user1 localhost:3000/secret_documents/index.json
```

Chapter 2: Git

Git is such a massive and complex tool that you could fill an entire book with Git workflow upgrades alone. And in fact, many books have been written about Git.

Since Git is such a mainstay of modern development, I knew it needed to be a chapter in Efficient Rails. However, there's no way I can do justice to all the possible ways to improve your Git workflow in a single chapter. So instead, I picked ten upgrades that I think will be the most helpful for Rails developers.

In this chapter, we'll cover ways to wrangle branches, integrate with GitHub, learn about Git Flow and how to apply it painlessly to your workflow, and other helpful tips and tricks to get the most out of such a powerful tool.

Git Flow

Problem

Git is a powerful version control tool, but if you've never used it within a large engineering team before, you might be used to simply committing to master. This is fine if you're the only developer, but as soon as you have 2 or more developers working within the same Git repository, you'll start to see problems arise.

These problems include:

- Constant merge conflicts
- Commits with poor code quality slipping into master
- Lack of clarity regarding which features are currently in master
- Uncertainty of when to deploy because the status of the features in master is unclear

Solution: Git Flow

Git Flow is a method of using Git that addresses the pain points mentioned above. Its main tenent: anything in the master branch is always deployable.

How can we ensure that the master branch is always deployable? By never committing directly to master. Instead, we'll use other branches.

When using Git Flow, we have two core branches, develop and master. The master branch represents what is currently in production while the develop branch represents the current state of development. When you create a new feature, you branch off of develop, make your changes, and then push the feature branch to GitHub, creating a pull request from your branch into develop. This says: "I have these changes that I want to merge into develop, please review".

After a (hopefully thorough) code review, the pull request is merged into master.

This works great for feature development, but what about hotfixes that need to get into production now? In that case, we would branch off *master*, make the necessary changes, and then make a pull request from our hotfix branch into *master* for review, before being merged directly into master (and later merged into develop with a simple fast-forward merge).

Git (and GitHub) Flow is a huge topic for discussion in the software development community and there are several schools of thought on the matter. Since there's no way I can do the topic justice, I'll list a few resources that cover the topic with much more detail.

Git Flow Resources

- http://nvie.com/posts/a-successful-git-branching-model/[17]

[17] http://nvie.com/posts/a-successful-git-branching-model/

- https://guides.github.com/introduction/flow/[18]
- https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow[19]
- http://scottchacon.com/2011/08/31/github-flow.html[20]

[18] https://guides.github.com/introduction/flow/
[19] https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow
[20] http://scottchacon.com/2011/08/31/github-flow.html

Git + Hub

Problem

GitHub has come to be nearly synonymous with Git. It's the heart and soul of any project: it's the master copy of the codebase, it's where code gets reviewed and merged, it facilitates open source collaboration.

However, when using Git from the command line, it has no concept of GitHub, except as a remote Git server it can fetch from and push to.

Solution: Hub

Hub[21] is an official tool maintained by GitHub that teaches Git about GitHub.

It adds some nice assumptions to Git, like that a repository can be represented just with the account and repository name. That allows us to clone repositories like this:

```
$ git clone rails/rails
```

Rather than using this more verbose ssh syntax:

```
$ git clone git@github.com:rails/rails.git
```

It also adds some really cool functionality like the ability to create pull requests and issues right from Git.

Commands

Here's a list of the commands Hub gives us:

Hub Commands

Command	Result
pull-request	Open a pull request on GitHub
fork	Make a fork of a remote repository on GitHub and add as remote
create	Create this repository on GitHub and add GitHub as origin
browse	Open a GitHub page in the default browser
compare	Open a compare page on GitHub
release	List or create releases (beta)
issue	List or create issues (beta)
ci-status	Show the CI status of a commit

[21] https://github.com/github/hub

Installing Hub

Hub can be installed directly from HomeBrew:

```
$ brew install hub
```

Aliasing Hub

Hub's documentation[22] recommends aliasing `hub` to `git`:

> Using hub feels best when it's aliased as git. This is not dangerous; your normal git commands will all work. hub merely adds some sugar.

Let's go ahead and create that alias:

```
alias git="hub"
```

Now, we can create pull requests without leaving the comfort of the terminal:

```
$ git pull-request
```

[22] https://github.com/github/hub#aliasing

Better Branch View

Problem

Git flow is a great way to organize development on a project. It allows everyone to work on a feature or bug fix in isolation, promotes code review through pull requests, and helps coordinate the state of your production and staging environments.

The byproduct of this approach, however, is that over time, you tend to accumulate a mess of local branches. You can of course prune these branches to get rid of the ones that have been merged (link), but you still might end up with some older branches or experiments that never made it into production.

Unfortunately, Git's built-in view of branches doesn't provide much context about each branch. This can make looking for a particular branch a tedious process.

```
$ git branch
  3-2-stable
  4-1-stable
  4-2-5
  4-2-stable
  fix_ci
* master
  relation_subclasses
  rework_initialization
  rm-3-2-with-ruby-2-1-plus
  tagged-docs
```

Default Git branch view

What was that branch called again?

Solution

So how can we get a better view of our branches?

Digging through Git's manpages, we can exploit some of the advanced sorting and formatting options of Git's `for-each-ref` command to make information like the branch's last commit date and message available. At the time of this writing, for-each-ref's formatting options do not allow for column-wise tabbing. So to fill in that gap and make the branch view tabular, we'll pipe the input into the `column` bash utility.

The resulting command:

```
$ git for-each-ref --sort=committerdate refs/heads/ --format='%(HEAD) %(color:ye\
llow)%(refname:short)%(color:reset) | %(authorname) | (%(color:green)%(committer\
date:relative)%(color:reset))| %(contents:subject) ' | column -t -s '|'
```

And now we have a very usable list of local branches:

```
$ gb
  relation_subclasses          Jon Leighton          (3 years ago)            Use separate Relation subclasses for each AR class
  fix_ci                       Jon Leighton          (2 years, 8 months ago)  Experiment to try to fix the build
  tagged-docs                  Xavier Noria          (2 years ago)            tagging example
  rm-3-2-with-ruby-2-1-plus    Rafael Mendonça França (10 months ago)         Lock i18n to a version that works with Ruby 1.8
  rework_initialization        Roger Leite           (10 months ago)          Fix tiny typo in documentation
  3-2-stable                   Rafael Mendonça França (5 months ago)          Merge pull request #20629 from moklett/patch-1
  4-2-5                        Rafael Mendonça França (7 days ago)            Preparing for 4.2.5.rc2 release
  4-2-stable                   Christoph             (28 hours ago)           Fix week_field returning invalid value
  4-1-stable                   Christoph             (28 hours ago)           Fix week_field returning invalid value
* master                       Claudio B             (3 hours ago)            Merge pull request #22258 from rymai/patch-1
```

Improved Git branch view

Aliasing it

We can take this one step further and alias this. I have mine mapped to `gb`:

```
alias gb="git for-each-ref --sort=committerdate refs/heads/ --format='%(HEAD) %(\
color:yellow)%(refname:short)%(color:reset) | %(authorname) | (%(color:green)%(c\
ommitterdate:relative)%(color:reset))| %(contents:subject) ' | column -t -s '|'"
```

Toggle Branches

Problem

If you're using feature branches, you probably find yourself switching between branches pretty often. And oftentimes, the branch names can get a little long. While tab completion helps alleviate the problem of long branch names, sometimes you just want to toggle quickly between two branches. Maybe `master` and `develop`, or `develop` and a feature branch.

Solution

Git provides a simple solution for this. Drawing from bash syntax where `cd -` switches your location to the last directory, `git checkout -` checks out the previous branch you were on.

It's a simple solution, sure, but I figured it was worth including since it's something I use every single day.

To make this process just a little bit easier, I have `git checkout` aliased to `gco`. My alias comes from Oh My ZSH. Check out the Terminal chapter for more info.

Don’t Use ‘git add .’

Problem

Using `git add .` (or `git add -A`) is just asking for trouble.

When first learning Git, the process of staging and committing a change can seem unwieldy:

```
$ git status
$ git add app/models/post.rb
$ git add app/controllers/posts_controller.rb
$ git add app/views/posts/index.rb
$ ...
$ git diff --staged
$ git commit -m "Created posts"
```

This might seem like a waste of keystrokes. Once you learn that `git commit -am "a commit message"` does this all in one line, you might be tempted to do all of your committing like this. Don’t!

While staging all changes by default might seem like a time-saver at first, it won’t take long for the problems to start rolling in.

When you don’t pay attention to what you’re staging, it’s easy to mistakenly commit half of something that’s actually part of another feature and have to go back and painfully extract that code. Not to mention, debugging statements finding their way into pull requests and even production.

Solution: Tig

While there are several GUI Git clients that work pretty well, none are perfect. They tend to be either cumbersome and weighted down with features, or just plain expensive.

Since the command line is a much more efficient way of dealing with Git, and we just need a tool to deal with the staging area and diff viewing, the thought of paying for features we’ll never use is unappealing.

Enter Tig, a command-line GUI Git client. It’s not very well-documented and it’s hard to find screenshots of it in action, but it adds some much needed UX sugar to common Git tasks.

Installing Tig

Tig can be installed with homebrew:

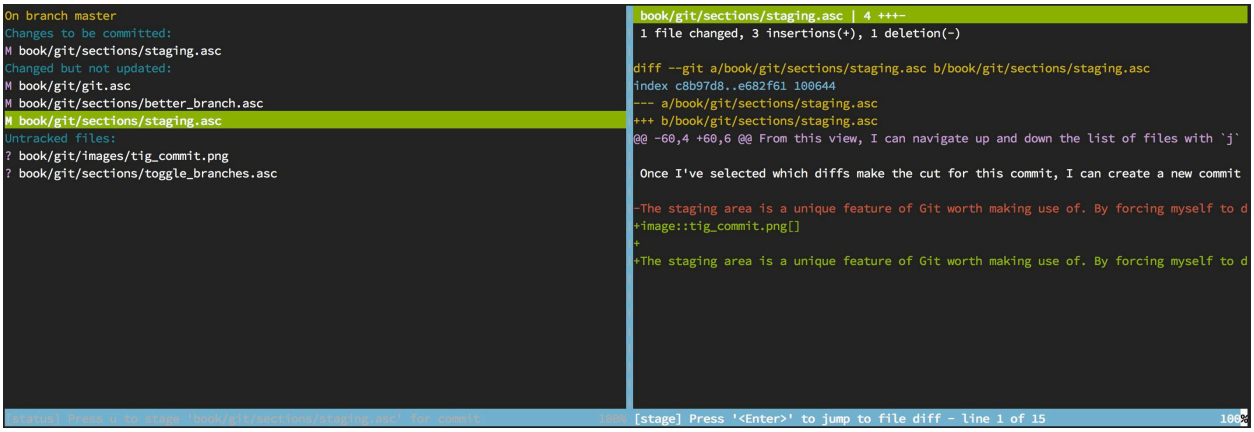
```
$ brew install tig
```

Preparing a Commit with Tig

Let's walk through the process of reviewing and staging diffs using Tig using none other than the Git repository for this book :).

After finishing up my work on a feature, we'll open up the Tig status view.

```
$ tig status
```



Tig's status view

From this view, we can navigate up and down the list of files with `j` and `k`. To see what changes a file includes, we can open the diff viewer with `Enter` and close it with `q`. If a diff looks like it should be included in the commit we're preparing, `u` will move it into the staging area. To include only certain lines of a file in the commit, we can stage individual lines by pressing `1` on that line within the diff viewer.

Once we've selected which diffs make the cut for this commit, we can create a new commit with `C`, which launches an editor instance for writing the commit message and description.

Committing with Tig

The staging area is a unique feature of Git worth making use of. By forcing ourselves to do this review before each commit, we end up catching minor mistakes or seeing new ways of doing things. Tig streamlines the process and makes viewing diffs much more pleasant that what you get with Git out of the box.

Prune Merged Branches

Problem

If you use Git Flow, you end up creating a branch for every feature and bug fix no matter how small. Left unchecked, those branches build up over time to the point where running `git branch` or especially `git branch -a` can cause minor panic attacks. Once a branch has been merged, it's no longer worth keeping around, yet you have to sort through every old branch you've ever worked with.

Not only do these hoarder tendencies make it difficult to see the branches that actually matter, it also bloats the size of your git repository.

Solution

Git branches fall into one of three categories:

1. Remote branches stored in GitHub or another central Git server
2. Local references to remote branches
3. Local branches you created

We'll look at ways to clean up each of these types of branches.

Delete Remote Branches After Merging

The first step is to delete your remote branches after they've been merged into your main branch (develop or master). Assuming you use GitHub and create Pull Requests for each branch you want to merge, it's as simple as clicking the delete branch button after clicking Merge. If you've got a lot of undeleted branches lying around in GitHub, you can view all of your branches and can safely delete the ones that have a "Merged" tag next to them.

If you're the paranoid type (it takes one to know one), GitHub makes it easy to restore accidentally deleted branches. So delete without fear!

Prune Local References to Deleted Remote Branches

Your local Git repository keeps references to every branch available on the remote server. You can see these remote branches by running `git branch -a` which will show the remote references in addition to your local branches.

Running `git fetch --prune` will fetch all references available on the remote server, and conveniently delete the ones which no longer exist there.

```
$ git fetch --prune
From github.com:your-account/your-project
 x [deleted]         (none)     -> origin/feature/some-feature
 x [deleted]         (none)     -> origin/feature/another-feature
 x [deleted]         (none)     -> origin/fix/an-important-bugfix
 x [deleted]         (none)     -> origin/feature/yet-another-feature
```

Delete Merged Local Branches

There's no built-in way to delete your local branches that have been merged. You'd have to run `git branch -d your-merged-branch` for each merged branch. But that doesn't mean we can't use some terminal magic to do it ourselves!

```
$ git branch --merged master | command grep -vE "^(\*|\s*develop\s*|\s*master\s*\
$)" | command xargs -n 1 git branch -d
```

This will delete all local branches that have been merged into master, excluding the current branch, or develop and master branches.

We can alias this command for future use. In your .bashrc or .zshrc file, add the following:

```
alias gsweep='git branch --merged master | command grep -vE "^(\*|\s*develop\s*|\
\s*master\s*$)" | command xargs -n 1 git branch -d'
```

Now we can run it with a simple `$ gsweep`.

Oops...

We all make mistakes. Here's how to fix some common ones in Git.

Problem: There's a typo in my commit messages

You make a commit, but realize your commit message has a typo. Or you want to add something to the commit message or description.

Solution: Amend

```
$ git commit --amend
```

Amend will pop open your terminal editor, and allow you to change the previous commit message and description.

```
change with new

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Wed Dec 16 08:15:45 2015 -0800
#
# On branch master
# Changes to be committed:
#»·····new file:   file.rb
#»·····new file:   new.rb
#
```

Amending a commit

If you have any changes staged, amending the commit will include those changes. This can be good if you want to add additional changes to the commit, but if you just want to edit the commit message, make sure your staging area is clear first.

Problem: I committed too soon

You just committed, but realized you want to make some more changes before committing. Or perhaps the commit includes some files it shouldn't or was made to the wrong branch. Basically, you want to Cmd+Z the fact that you just committed.

Solution: Reset (Soft)

```
$ git reset --soft HEAD~
```

A soft reset will remove the last commit, but keep all files and changes made in that commit. It even leaves everything staged just the way you had it before committing.

`HEAD~` refers to the last commit made. You could reset the last 2 commits with `HEAD~2`, for example.

Problem: Scratch that

You make some changes to a file, but decide you don't want them anymore. You haven't committed yet, and you just want to revert a file back to how it was before you started editing. But maybe you've closed the file since you started editing it, and can't undo your changes with your editor.

Solution: Checkout

While you're probably familiar with the `checkout` command for creating and moving between branches, it also does double duty undoing changes in files.

```
$ git checkout file_with_changes.rb
```

By default, this will reset the file back to its current state in HEAD.

If you undo changes this way, they're gone forever. When you commit changes, you can usually get them back with the reflog. But since you haven't committed anything, it will be as if the changes never existed.

Learn More

For a comprehensive rundown on undoing things with Git, check out the following article on GitHub[a].

[a] https://github.com/blog/2019-how-to-undo-almost-anything-with-git

Where did it all go wrong?

Problem

You pull down code and notice something is broken. There were a bunch of commits and the tests didn't catch the bug. now tasked with either reading through each commit and trying to find out which one was the offending commit.

Solution: Git Bisect

Git bisect is a powerful way to find out which commit caused a breaking change. While you might not use it too often, when you do, it will make you feel like a wizard, not to mention save you hours.

The idea is simple: you give it a 'good' commit, (one sometime before the problem started), and a 'bad' commit, (one where the problem is present). Then, Git will 'bisect' those commits by checking out the commit in between the good and bad commits. You then tell Git whether the new commit is good or bad and it will zero in on the offending commit.

Example

First, we'll check out an older commit and verify that it was working properly at that point in time.

```
$ git checkout 83aec86
```

Once we have that point of reference, we can start up bisect giving it the older good commit, and the new bad commit.

```
$ git bisect start
$ git bisect good 83aec86
$ git bisect bad bcb7b1d
```

If the code works at this point, we'll tell Git that everything is good.

```
$ git bisect good
```

If the problem is present, we'll tell Git that it's bad.

```
$ git bisect bad
```

By process of elimination, Git will find the first commit that introduced the problem. Knowing where the problem lies, let's get back to a working state:

```
$ git bisect reset
```

Automating It

Let's take it a step further and automate the process of testing each commit for 'good' or 'bad'. Since our test suite failed to catch the bug when it was checked in, we'll write a failing test and tell Git to use that spec to test each commit. This way, we don't need to manually test each commit for the bug.

The setup is the same:

```
$ git bisect start
$ git bisect good 83aec86
$ git bisect bad bcb7b1d
```

But now, we tell it to go ahead and run the bisect process with the following spec:

```
$ git bisect run rspec spec/features/my_broken_spec.rb
```

Go grab a coffee and when you get back, Git will have found your problem for you!

Ignore Whitespace

Problem

When trying to figure out who to ask about a piece of code, `git blame` can tell you who to go looking for. The only problem is that Git treats commits that changed whitespace as actual modifications to the code. Just because someone cleaned up some trailing whitespace or fixed some indentation issues doesn't mean they should be held responsible for the underlying code.

Similarly, when viewing diffs in the command line and even in GitHub, whitespace changes can cause a major distraction. It becomes difficult to figure out what the important changes were.

Solution

As it turns out, `git blame` take an argument, `-w`, which tells it to ignore commits that only changed whitespace.

```
$ git blame -w
```

Additionally, the `-M` command tells Git to ignore commits that simply *moved* a piece of code somewhere else.

```
$ git blame -M
```

By using these together, you can figure out who the last person to actually make significant changes to the code was.

```
$ git blame -wM
```

Diffs

The diff command also takes the `-w` argument to ignore whitespace when comparing two files.

```
$ git diff -w
```

Similarly, when viewing diffs in GitHub, whitespace changes can obscure the nature of the comparison. It's hard to tell what actually changed as opposed to what was cleanup. By adding `?w=1` to the url of any file comparison or pull request view, GitHub will show only the non-whitespace changes.

Change the Root of a Branch

Problem

You just realized that you branched off the wrong branch.

Maybe you're using Git Flow and got to the end of a hotfix. Everything is working great, but you branched your hotfix off of *develop* out of habit, rather than *master*.

Now, when you make a pull request from your hotfix to master, you see all kinds of changes that aren't yours, that still belong on develop, not master.

So how do you fix this? Maybe you copy your changes, delete the branch and create a new one with the changes. You think "Git is such a powerful tool, there must be some way to do this, but this hotfix needs to get to production. I can't worry about that now".

Solution

Well lucky for you, Git has a solution for this problem and here it is:

```
$ git checkout hotfix
$ git rebase --onto master develop
```

This says: "rewrite the hotfix branch's history to branch off of master, rather than develop". Or more precisely: "rewrite the hotfix branch's history to include commits that master and develop have in common, and discard the rest". Exactly what we want.

You may run into rebase conflicts if you edited the same file that was changed in develop. If this happens, resolve the conflict in the file, stage it, and run `git rebase --continue`.

Now, we can push the hotfix branch to GitHub, create our pull request into master without worrying about prematurely introducing develop's commits into production.

Chapter 3: Rails Console

The Rails console is a powerful tool that comes out of the box with Rails. It's great for trying out Ruby code, instantiating models and even querying and exploring the database.

I have a Rails console open all day long and spend a non-trivial chunk of time using it each day.

In this chapter, I'll cover a handful of workflow upgrades that will make time spent working in the Rails console that much more enjoyable.

Setting up the Console

Problem

Rails comes with a pretty lackluster console by default. Sure, it's a handy tool, but it's missing a number of features that would make it much more powerful.

```
irb(main):001:0> Post.first
  Post Load (0.2ms)  SELECT  "posts".* FROM "posts"  ORDER BY "posts"."id" ASC LIMIT 1
=> #<Post id: 1, title: "Bushwick Migas Cleanse Xoxo Tilde Tumblr", author: "Olga Veum", body: "Iphone pinterest polaroid twee. Gluten-free fixie ...", tags: ["umami", "chicharro
nes"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">
irb(main):002:0>
```

The default Rails Console

Here are some things the Rails console is sorely missing:

1. Syntax highlighting
2. Pretty-printed output
3. A decent REPL (Pry > IRB)

Inspecting an object or collection spits out a chunk of monochromatic goop:

```
irb(main):001:0> Post.first
  Post Load (0.2ms)  SELECT  "posts".* FROM "posts"  ORDER BY "posts"."id" ASC LIMIT 1
=> #<Post id: 1, title: "Bushwick Migas Cleanse Xoxo Tilde Tumblr", author: "Olga Veum", body: "Iphone pinterest polaroid twee. Gluten-free fixie ...", tags: ["umami", "chicharro
nes"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">
irb(main):002:0> Post.all
  Post Load (5.3ms)  SELECT "posts".* FROM "posts"
=> #<ActiveRecord::Relation [#<Post id: 1, title: "Bushwick Migas Cleanse Xoxo Tilde Tumblr", author: "Olga Veum", body: "Iphone pinterest polaroid twee. Gluten-free fixie ...",
tags: ["umami", "chicharrones"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 2, title: "Stumptown Cronut I Phone Venmo", author: "Glenna Sha
nahan", body: "Bicycle rights meditation art party tattooed aesth...", tags: ["banjo"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 3, title
: "Kickstarter Tofu Synth Godard", author: "Nat Jast Jr.", body: "Tote bag tousled meh crucifix blue bottle. Distill...", tags: ["XOXO"], created_at: "2015-12-10 16:28:27", updat
ed_at: "2015-12-10 16:28:27">, #<Post id: 4, title: "Ethical Neutra Listicle Cray Pour Over Hammock Has...", author: "Mrs. Domingo Hickle", body: "Wayfarers direct trade church-k
ey dreamcatcher. Wa...", tags: ["flannel"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 5, title: "Leggings Echo Everyday Intelligentsia", a
uthor: "Prudence Jacobi", body: "Diy gastropub deep v typewriter quinoa squid green...", tags: ["marfa"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #
<Post id: 6, title: "Whatever Typewriter Mixtape Occupy", author: "Selmer Metz", body: "Five dollar toast chicharrones shabby chic hashtag...", tags: ["venmo"], created_at: "2015
-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 7, title: "Butcher Bespoke Try Hard Skateboard Meh Hashtag Dr...", author: "Luisa Yundt", body: "Hoodie cred mixt
ape taxidermy post-ironic beard xo...", tags: ["franzen", "gentrify", "pabst"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 8, title: "Ugh G
odard Pitchfork Plaid Williamsburg Typewriter...", author: "Agustin Mitchell", body: "Carry hoodie williamsburg. Yuccie roof fashion axe...", tags: ["loko", "pinterest", "humbleb
rag"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 9, title: "Celiac Aesthetic Fingerstache Diy Plaid Thundercat...", author: "Maximus Walte
r", body: "Flannel meh deep v banjo normcore swag yolo polaro...", tags: ["dreamcatcher", "gluten-free", "keytar"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:
28:27">, #<Post id: 10, title: "Selvage Tacos Intelligentsia Skateboard Biodiesel", author: "Iva Feeney Sr.", body: "Wayfarers hoodie retro actually tattooed messenger...", tags:
 ["vinegar"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, ...]>
irb(main):003:0>
```

Inspecting a collection with the default Rails Console

I dare you to figure out what's going on in that output.

Solution: Jazz Fingers

Jazz Fingers aptly describes itself as "an opinionated set of console-related gems and a bit of glue". By installing Jazz Fingers, you get the following gems installed with sensible defaults:

- Pry for a powerful shell alternative to IRB.
- Awesome Print for stylish pretty print.
- Hirb for tabular collection output.

- Pry Doc to browse Ruby source, including C, directly from the console.
- Pry Git to teach the console about git. Diffs, blames, and commits on methods and classes, not just files.
- Pry Remote to connect remotely to a Pry console.
- Pry Coolline for syntax highlighting as you type.

We'll cover the details of some of these gems in further sections, but at the very least, Jazz Fingers gets you a Pry-based console by default, some nice syntax highlighting and pretty-printed objects.

Here's the result:

The Rails Console with Jazz Fingers

Install

To install, simply add the following to your Gemfile:

```
group :development, :test do
  gem 'jazz_fingers'
end
```

Now, the next time you boot up your console, you'll notice a much more stylish and practical interface.

Save Frequently Used Commands

Problem

You probably notice yourself reusing the same commands over and over when you're in the Rails console.

For example, I often find my test user in the local database by typing:

```
1  me = User.find_by(email: 'andrew@example.com')
```

Or maybe you have some really long model or class names that are unwieldy to type over and over.

And how many times have you typed something this just to have a hash to play with:

```
1  my_hash = {a: 'b', c: 'd'}
```

How can we save ourselves a few keystrokes in the Rails console?

Solution

Both Pry and IRB allow us to define config files local to a project. Within these config files, we can define our own convenience methods to make our lives a little bit better.

Within the root of our project, we'll add a `.pryrc` file (or `.irbrc` file if not using Pry, the syntax is the same).

your_app/.pryrc

```
1  class Object
2    def me
3      User.find_by(email: 'andrew@example.com')
4    end
5
6    def slmn
7      SomeLongModelName
8    end
9
10   def an_array
11     (1..5).to_a
12   end
13
14   def a_hash
```

```
15      {a: 'b', c: 'd'}
16    end
17  end
```

Now, every time we boot up a console, we'll have access to these methods. That means we can call methods on our test user by just typing `me`. We also have an alias to make tying out long class names a thing of the past. We can simply type `slmn.where(...)` instead of `SomeLongModelName.where(...)`.

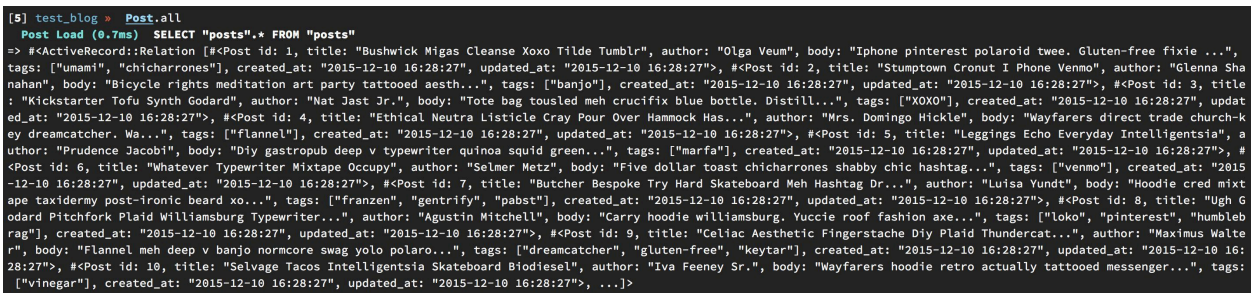
And when we need a dummy array or hash to try something on, we can just type `a_hash` or `an_array`.

What convenience methods will you define?

Inspecting ActiveRecord Collections

Problem

When trying to view large collections of ActiveRecord objects, the Rails console prints a tangled, truncated mess:



Viewing a collection of ActiveRecord objects

This is far from useful.

Solution

Jazz Fingers (see Setting up the Console) bundles together some nice gems to improve our Rails console experience. One of those is Hirb, a gem that enables tabular output when viewing ActiveRecord collections in the console.

However, Hirb is not enabled by default. You can toggle Hirb manually when you need it by running the following commands in the console:

```
1 Hirb.enable  #=> enables tabular output
2 Hirb.disable #=> disables tabular output
```

After enabling Hirb, we can browse our ActiveRecord collections with tables:

id	title	author	body	tags	created_at	updated_at
1	Bushwick Migas Cleanse Xo...	Olga Veum	Iphone pinterest polaroid...	umami, chicharrones	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
2	Stumptown Cronut I Phone ...	Glenna Shanahan	Bicycle rights meditation...	banjo	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
3	Kickstarter Tofu Synth Go...	Nat Jast Jr.	Tote bag tousled meh cruc...	XOXO	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
4	Ethical Neutra Listicle C...	Mrs. Domingo Hickle	Wayfarers direct trade ch...	flannel	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
5	Leggings Echo Everyday In...	Prudence Jacobi	Diy gastropub deep v type...	marfa	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
6	Whatever Typewriter Mixta...	Selmer Metz	Five dollar toast chichar...	venmo	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
7	Butcher Bespoke Try Hard ...	Luisa Yundt	Hoodie cred mixtape taxid...	franzen, gentrify, pabst	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
8	Ugh Godard Pitchfork Plai...	Agustin Mitchell	Carry hoodie williamsburg...	loko, pinterest, humblebrag	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
9	Celiac Aesthetic Fingerst...	Maximus Walter	Flannel meh deep v banjo ...	dreamcatcher, gluten-free...	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
10	Selvage Tacos Intelligent...	Iva Feeney Sr.	Wayfarers hoodie retro ac...	vinegar	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
11	Pitchfork Cornhole Diy Ca...	Eliza Toy	Vice wayfarers everyday b...	migas	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
12	Schlitz Kogi I Phone Quin...	Braeden Upton	Bitters chambray fap salv...	typewriter, heirloom	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
13	Bespoke Microdosing Chia ...	Zula Reichert V	Lomo fanny pack waistcoat...	leggings	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
14	Viral 8 Bit Tousled Pbr&B...	Jennie Satterfield	Bushwick pug farm-to-tabl...	tumblr, mlkshk	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
15	Hella Organic Heirloom Bi...	Miss Aniyah Padberg	Aesthetic viral tattooed ...	fixie, quinoa	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
16	Drinking Semiotics Wayfar...	Charlotte Wolf Jr.	Authentic narwhal bushwic...	literally, Yuccie, locavore	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
17	Yuccie Freegan Celiac Meg...	Pearlie Cormier	Banjo authentic everyday ...	mixtape, ethical	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
18	Everyday Vhs Kombucha Lomo	Allie Fahey	Yr synth kitsch banjo. Bi...	echo	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
19	Etsy Aesthetic Tacos Mast...	Tatum Erdman	Freegan occupy craft beer...	hoodie	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
20	Fingerstache Pop Up Ethic...	Eliezer Balistreri	Portland typewriter vinyl...	meggings, chia	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC
21	Taxidermy Vice Cornhole C...	Tillman Lind	Synth distillery tumblr k...	bitters	2015-12-10 16:28:27 UTC	2015-12-10 16:28:27 UTC

Hirb displays output in a tabular format

Re-Execute Commands

Problem

Working with the Rails Console (or any console for that matter) is an exercise in trial and error. To get a desired outcome, we'll try a few commands, tweak them, then go back and try some more. This usually means executing the same set of commands a few times over.

To save time, most developers are familiar with using the up arrow to walk back through the command history. This is a great feature of most modern command line interfaces, but what if you've run a large number of commands between the target command and now? You'll have to page up through all of those intermediate commands in order to get to the one you want.

Solution: Reverse Search

Most modern command line interface have a lesser-known but highly-useful method of accessing the command history called reverse search, and the Rails console is no different.

Pressing Ctrl + r in the Rails console triggers reverse search mode. From this mode, we can start typing any part of the command we're looking for, and the most recent matching command will be shown. From here, we have a few options:

- Press Ctrl + r again to see the next most recent matching command
- Press Return to execute the matching command
- Press Ctrl + g to exit reverse search
- Press Esc to exit reverse search, but keep the matching command for editing

```
[8] test_blog »
(search:Post): Post.last(5)
```

Reverse search in the Rails Console

Change Rails Environments

Problem

By default, the Rails Console starts in the development environment. If a RAILS_ENV environment variable is set, like in production, it will use that environment.

But what if we want to specify a different environment?

Solution #1: Start up Rails Console in a Different Environment

If we want the Rails console to start up in a different environment, we can specify one explicitly. The easiest way is to add the environment as an argument to the `rails console` command.

```
$ bundle exec rails console test
```

Additionally, you can override the `RAILS_ENV` environment variables.

```
$ RAILS_ENV=test bundle exec rails console
```

Solution #2: Hot Swap Database Connections

What if you're got a console session running in the development environment but you want to switch to the test database? This usually happens to me when I want to debug a FactoryGirl factory but I don't want to litter my development database with stray objects.

Sure, we could stop the current session and restart in the target environment, or even open another session in a new tab. But there's another way. We can define a helper method in our `~/.pryrc` (or `~/.irbrc`) that can hot swap database connections.

~/.pryrc

```
class Object
  def switch_db(env)
    config = Rails.configuration.database_configuration
    raise ArgumentError, 'Invalid Environment' unless config[env].present?

    ActiveRecord::Base.establish_connection(config[env])
    Logger.new(STDOUT).info("Successfully changed to #{env} environment")
  end
end
```

Now, we can simply enter `switch_db 'test'` to change over to the test database.

Note that this method will only change the database connection, not the Rails environment.

Ignore Sluggish Output

Problem

When you perform an ActiveRecord query in the Rails console, it prints out the result of the query, which can take several seconds if the query contains more than a few records. But let's say you're setting up a query and saving it to a variable for additional querying.

ActiveRecord is lazy by default, meaning you can build up a query by chaining additional methods and the query doesn't actually get run until the results are needed. But since the Rails console executes every line and renders its output, ActiveRecord queries lose their lazy properties.

If you're saving the query to a variable to run additional queries on, you don't actually need to see every record printed out. In fact, we'd prefer that the query be lazy and not execute until it's been narrowed down.

Solution

We can take advantage of a bit of Ruby trickery to keep ActiveRecord queries lazy and prevent them from spending time rendering output we don't care about.

By simply appending `; nil` to the end of the query, we split the line in two, letting the first one execute lazily and returning `nil` from the second. The Rails console only sees the `nil` return value and does not execute the query to display it. Now, that lazy query is saved to a variable and we can continue operating on it without the distraction of seeing the output.

```
[15] test_blog »  p = Post.all
  Post Load (0.5ms)  SELECT "posts".* FROM "posts"
=> #<ActiveRecord::Relation [#<Post id: 1, title: "Bushwick Migas Cleanse Xoxo Tilde Tumblr", author: "Olga Veum", body: "Iphone pinterest polaroid twee. Gluten-free fixie ...",
tags: ["umami", "chicharrones"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 2, title: "Stumptown Cronut I Phone Venmo", author: "Glenna Sha
nahan", body: "Bicycle rights meditation art party tattooed aesth...", tags: ["banjo"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 3, title
: "Kickstarter Tofu Synth Godard", author: "Nat Jast Jr.", body: "Tote bag tousled meh crucifix blue bottle. Distill...", tags: ["XOXO"], created_at: "2015-12-10 16:28:27", updat
ed_at: "2015-12-10 16:28:27">, #<Post id: 4, title: "Ethical Neutra Listicle Cray Pour Over Hammock Has...", author: "Mrs. Domingo Hickle", body: "Wayfarers direct trade church-k
ey dreamcatcher. Wa...", tags: ["flannel"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 5, title: "Leggings Echo Everyday Intelligentsia", a
uthor: "Prudence Jacobi", body: "Diy gastropub deep v typewriter quinoa squid green...", tags: ["marfa"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #
<Post id: 6, title: "Whatever Typewriter Mixtape Occupy", author: "Selmer Metz", body: "Five dollar toast chicharrones shabby chic hashtag...", tags: ["venmo"], created_at: "2015
-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 7, title: "Butcher Bespoke Try Hard Skateboard Meh Hashtag Dr...", author: "Luisa Yundt", body: "Hoodie cred mixt
ape taxidermy post-ironic beard xo...", tags: ["franzen", "gentrify", "pabst"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 8, title: "Ugh G
odard Pitchfork Plaid Williamsburg Typewriter...", author: "Agustin Mitchell", body: "Carry hoodie williamsburg. Yuccie roof fashion axe...", tags: ["loko", "pinterest", "humbleb
rag"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, #<Post id: 9, title: "Celiac Aesthetic Fingerstache Diy Plaid Thundercat...", author: "Maximus Walte
r", body: "Flannel meh deep v banjo normcore swag yolo polaro...", tags: ["dreamcatcher", "gluten-free", "keytar"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:
28:27">, #<Post id: 10, title: "Selvage Tacos Intelligentsia Skateboard Biodiesel", author: "Iva Feeney Sr.", body: "Wayfarers hoodie retro actually tattooed messenger...", tags:
 ["vinegar"], created_at: "2015-12-10 16:28:27", updated_at: "2015-12-10 16:28:27">, ...]>
[16] test_blog »  p = Post.all; nil
=> nil
[17] test_blog »
```

Comparing the result with and without '; nil'

Don’t wear out the Backspace Key

Problem

The Rails console is a command line application, which means when you’re in it, you aren’t able to edit your input in the ways you’re accustomed to. For example, let’s say you have a command at the prompt, but you want to delete the whole thing.

If you’re used to Vim, you would press `dd` to delete the line. If you’re used to Mac shortcuts, you would press `Cmd+Backspace` to delete from your cursor to the beginning of the line. Or if you’re a mouse user (shame! just kidding), you might highlight the line and press backspace.

Well, none of this works in the command line (at least not without some serious customization).

Instead, you’d end up pressing backspace over and over, or holding it down until the line has been scrubbed out on character at a time. The same goes for moving around the line of text, like if you wanted to prepend some text (maybe a variable assignment) to the beginning of the line. `I` doesn’t work for Vim users, `Cmd+Left` doesn’t work for Mac users, and clicking still doesn’t work. So we’re left with holding left until the arrow key falls off.

Solution

Luckily, the Rails console comes with a few keyboard shortcuts designed to alleviate the most painful of these navigational concerns:

Rails Console Text Navigation Keyboard Shortcuts

Command	Result
Ctrl+a	Jump to the beginning of the line
Ctrl+e	Jump to the end of the line
Ctrl+u	Delete from the cursor to the beginning of the line
Esc+f	Move forward one word
Esc+b	Move backward one word
Esc+d	Delete the next word
Esc+Backspace	Delete the previous word
Ctrl+w	Delete the previous word (to the previous space)

Reloading Models After Changing Them

Problem

Given that I have a Rails console open all day long, I tend to run into the issue where my underlying models change after the Rails environment has already been booted up in the console. This can happen naturally as I make changes to a model, or from checking out a different branch.

Because the Rails console has already instantiated all classes in the Rails application, it will fail to acknowledge the changes to the models. This usually means I'm heading for a restart (did you try turning it off and on again?).

Solution

The Rails console comes with a convenience method, `reload!`, which will reload any objects defined, including ActiveRecord models, picking up any changes that have been made to the classes since they were first instantiated. This saves us from needing to exit and restart the console, a non-trivial operation for a large Rails application.

Note that if you have any objects saved to a variable, you'll need to either call reload on them, or reinstantiate them.

If you find yourself needing to use `reload!` frequently, it could be a sign that you're doing too much testing in the console. In this case, it's better to use tools like RSpec which are designed to handle creating and reloading objects before each test. As a bonus, you are also documenting your test cases, preventing future regression. See the chapter on Testing to learn how to streamline your testing workflow.

Sandbox

Problem

The Rails console is a great way to try out commands as you're writing code, but you maybe you want to do so without disturbing the state of your development environment. Or maybe you want to boot up the Rails console in production to try and reproduce a bug without affecting real data.

Solution

The Rails console comes with a handy sandbox mode that makes it possible to try out new ideas without worrying about trashing your development database or disturbing the state of production data. It does this by wrapping the entire session in a gigantic database transaction that gets rolled back when the session is over. This way, no changes that you make during the sandbox session persist after the session is closed.

To start Rails console in sandbox mode, simply run:

```
1 $ bundle exec rails console --sandbox
```

Forgot to Save That

Problem

You write out a long query in the console to fetch a specific object, but you forget to assign the result to a variable. Now, you have to go back and edit the original command, sticking a `my_variable =` at the beginning of the line. And just to add insult to injury, if the database query was particularly expensive, it will need to execute again.

Command history and some handy keyboard shortcuts make this somewhat painless, but it still takes a few seconds.

Solution

This happens so often that there's a special variable built into the Rails console for just this purpose. _ refers to the result of the last command run. As an added bonus, the result is stored in memory, meaning that the database query doesn't need to be re-run.

You can reassign the contents of _ to a new variable and carry on like nothing even happened.

```
1  > my_variable = _
```

Call Private & Protected Methods

Problem

It's considered a best practice of Object-Oriented design to expose only the public interface of a class and encapsulate implementation-specific logic in private or protected methods.

Private methods reduce the surface area of a class, limiting the number of ways other classes can interact with it. This helps mitigate complexity in an application by discouraging coupling between objects.

A private method in Ruby

```
1  class Post < ActiveRecord::Base
2    # ...
3
4    private
5
6    def a_private_method
7      "Hey! Get outta here! This is private business!"
8    end
9  end
```

But what about when you're debugging in the console and to really understand what's going on, you need to see the response of a private or protected method? If you try calling the method in question directly, you'll get an (appropriate) error.

```
[4] test_blog »  Post.new.a_private_method
NoMethodError: private method `a_private_method' called for #<Post:0x007fcf3cc40f00>
from /Users/andrewallen/.rbenv/versions/2.2.0/lib/ruby/gems/2.2.0/gems/activemodel-4.2.1/lib/active_
model/attribute_methods.rb:430:in `method_missing'
```

Calling private methods results in an error

Do you comment out the private declaration and restart your test? That's a pain. And maybe you are running the console in production because that's the only place you can reproduce the issue and you can't modify the code.

How do you break the rules in this case and execute that private method?

Solution

As is so happens, there's a 'backdoor' way to call private or protected methods on Ruby objects. The `send` method effectively bypasses interface declarations, letting us call any method on a class, not just its public methods.

```
[2] test_blog »  Post.new.send(:a_private_method)
=> "Hey! Get outta here! This is private business!"
```

Send bypasses private and protected designations

While it might raise some eyebrows to learn that methods are never truly private in Ruby, this trick can be a huge time-saver when inspecting code in the console.

Pasting Multi-line Code into the Console

Problem

Ruby lets us chain methods together, and while you should be careful not to break the Law of Demeter doing so, ActiveRecord is built on method chaining. When building complex queries, it's common to have half a dozen or more methods chained together, causing your code to quickly exceed the 80 character screen width limit. In cases like this, Ruby style guides recommend putting each method call on its own line. This provides readability and also allows the order of method calls to be changed by simply swapping lines.

```
1  Movie
2    .comedies
3    .english_language
4    .rated(['PG-13', 'R'])
5    .released_after(1990)
6    .score_greater_than(80)
```

This is all well and good, but what happens when you go to test this query in the Rails console?

```
[2] pry(main)> Movie
=> Movie
[3] pry(main)> .comedies
Error: there was a problem executing system command: comedies
[4] pry(main)> .english_language
Error: there was a problem executing system command: english_language
[5] pry(main)> .rated(['PG-13', 'R'])
sh: -c: line 0: syntax error near unexpected token `['PG-13','
sh: -c: line 0: `rated(['PG-13', 'R'])'
Error: there was a problem executing system command: rated(['PG-13', 'R'])
[6] pry(main)> .released_after(1990)
sh: -c: line 0: syntax error near unexpected token `1990'
sh: -c: line 0: `released_after(1990)'
Error: there was a problem executing system command: released_after(1990)
[7] pry(main)> .score_greater_than(80)
sh: -c: line 0: syntax error near unexpected token `80'
sh: -c: line 0: `score_greater_than(80)'
Error: there was a problem executing system command: score_greater_than(80)
[8] pry(main)>
```

Multiline codes gets broken up when pasted in the console

The console assumes each line is an independent line of Ruby and executes after every line break. This means you would have to edit the text into one line before pasting.

Solution

Assuming you've configured your console to use Pry by default, you can use Pry's `edit` command to evaluate multiple lines at once.

To do this:

1. Copy the code you want to paste into the console.
2. Type `edit` without any arguments in the console. This will open a temporary file in your configured editor (let's assume this will be Vim).
3. Paste the code into the temporary file, save and exit (for Vim we'll use the handy `ZZ` shortcut to save and exit at once).
4. Now, Pry will evaluate your entire pasted statement.

Part II: The Code

In Part II, we'll begin our dive into the Rails codebase. I'll present some novel solutions to common problems you'll find yourself hitting in most Rails applications.

We'll start with the Model layer where we'll also discuss topics like database performance and migrations. From there, we'll move to the Controller and then to the View.

After the main MVC topics, we'll also cover some other key components of Rails applications: Assets and Mailers.

Again, none of these chapters will be an "introduction" to their respective topic, but will instead cover more advanced topics you won't find in a typical Rails tutorial.

Chapter 4: Models

This chapter will focus on the Model and Database layer of Rails applications.

As with the other chapters in Part II: The Code, most sections in this chapter will be recipes. ActiveRecord has a number of lesser-known features that are solutions to common problems and by familiarizing ourselves with these techniques, we can save time and effort by not reinventing the wheel.

The workflow upgrades in this chapter will give you at least one of the following:

- Reduced time spent solving problems with built-in solutions
- Reduced time spent by the database
- Reduced lines of code

We'll assume a basic knowledge of ActiveRecord, which will let us focus more on advanced problems rather than introducing the basics.

Enforcing Organized Models

Problem

We want our models to follow a consistent organization so that it's easy for us and other developers on our team to see what's going on. It also reduces decision fatigue when deciding where to add a new scope or callback.

The Rails Style Guide[23] presents a preferred pattern of grouping macro-style methods at the top of our models:

```
1  class User < ActiveRecord::Base
2    # keep the default scope first (if any)
3    default_scope { where(active: true) }
4
5    # constants come up next
6    COLORS = %w(red green blue)
7
8    # afterwards we put attr related macros
9    attr_accessor :formatted_date_of_birth
10
11   # followed by association macros
12   belongs_to :country
13
14   # and validation macros
15   validates :email, presence: true
16
17   # next we have callbacks
18   before_save :cook
19
20   # other macros (like devise's) should be placed after the callbacks
21
22   #...
23 end
```

Source[24]

However, it can be difficult to remember exactly what order these should be placed in.

[23] https://github.com/bbatsov/rails-style-guide
[24] https://github.com/bbatsov/rails-style-guide#macro-style-methods

Solution

Rails allows us to customize the templates used by its built-in generators, such as the model generator (`rails g model Book author:belongs_to`).

We can take look at the current stable version of this template depending on what version of Rails we're using. In this case, we'll look at Rails 4.2[25].

We can copy the contents of this template into a new file in our Rails app at `lib/templates/active_record/model/model.rb` and add the comments from the Rails Style Guide:

lib/templates/active_record/model/model.rb

```
<% module_namespacing do -%>
class <%= class_name %> < <%= parent_class_name.classify %>
  # keep the default scope first (if any)

  # constants come up next

  # afterwards we put attr related macros

  # followed by association macros
<% attributes.select(&:reference?).each do |attribute| -%>
  belongs_to :<%= attribute.name %><%= ', polymorphic: true' if attribute.polymo\
rphic? %><%= ', required: true' if attribute.required? %>
<% end -%>

  # and validation macros

  # next we have callbacks

  # other macros (like devise's) should be placed after the callbacks
<% if attributes.any?(&:password_digest?) -%>
  has_secure_password
<% end -%>

  # finally, scopes
end
<% end -%>
```

Now, when generating new models, we'll have handy comments that will enforce a consistent organization for our models. Of course, existing models will need to be reorganized by hand :)

[25] https://github.com/rails/rails/blob/4-2-stable/activerecord/lib/rails/generators/active_record/model/templates/model.rb

Annotate

Problem

What was that column called again? Rails uses the database schema to implicitly define the functions available on our models. If we forget what we named a column, or what attributes our model contains, we have to look up them up in the database, the schema file, or maybe even the Rails console. If we're editing the model, this takes us out of context.

Solution

Wouldn't it be nice to have model attribute information easily accessible? Luckily, there's a gem for that.

The Annotate gem automatically prepends your model files with comment blocks detailing the underlying schema of the model. The annotations display the column names and types, along with column attributes such as nullable, primary keys and indexes. Once installed, it will automatically regenerate these annotations whenever 'rake db:migrate' is called.

Now, when you need to look up some information about a given model there's no need to launch a database GUI or jump into the Rails console. Simply glance at the annotation block and carry on without even leaving the file you're editing.

In addition to annotating models, Annotate is also capable of annotating other files that benefit from easy access to schema information such as test, factories, fixtures and serializers.

```
1  # == Schema Info
2  #
3  # Table name: line_items
4  #
5  #  id                  :integer(11)    not null, primary key
6  #  quantity            :integer(11)    not null
7  #  product_id          :integer(11)    not null
8  #  unit_price          :float
9  #  order_id            :integer(11)
10 #
11
12 class LineItem < ActiveRecord::Base
13   belongs_to :product
14     . . .
```

Installing

Here's how to set up Annotate.

First, add it to the Gemfile:

```
gem 'annotate'
```

Then, from the root of your application, simply run

```
$ annotate
```

This will install Annotate with the default configuration.

If you just want to annotate your models, use the `exclude` flag:

```
$ annotate --exclude tests,fixtures,factories,serializers
```

For more detailed configuration instructions, check out the Readme[26].

[26] https://github.com/ctran/annotate_models

Eager Loading

Problem

One big problem with Rails and ActiveRecord is that it makes database-inefficient queries easy to write. If you're not familiar with what's going on under the hood, you can get yourself into trouble pretty quickly.

The reason for this is that Rails and ActiveRecord abstract away the internals of the database. And while this provides a consistent interface to multiple underlying databases and allows us to write queries much easier, if you don't know what those friendly-looking methods are doing, you end up making the database work much harder than it should. All the while, Rails stands by watching you do this.

For example, if we wanted to print the number of comments on the last 10 posts of our blog, we might do the following:

```
1  Post.last(10).map{ |post| post.comments.length }
```

When we run this, we see the following output:

```
[9] test_blog »  Post.last(10).map{|post| post.comments.length}
  Post Load (1.4ms)  SELECT  "posts".* FROM "posts"  ORDER BY "posts"."id" DESC LIMIT 10
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 191]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 192]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 193]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 194]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 195]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 196]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 197]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 198]]
  Comment Load (0.1ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 199]]
  Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?  [["post_id", 200]]
=> [
  [0] 7,
  [1] 7,
  [2] 8,
  [3] 13,
  [4] 9,
  [5] 11,
  [6] 15,
  [7] 11,
  [8] 9,
  [9] 9
]
```

Without eager loading

Notice that this query produced 11 calls to the database, 1 to load the last 10 posts, and 1 for each post to count the associated comments. This is where the term N+1 comes from. N corresponds to the 10 records we're looping over, which results in 10+1 or 11 queries to the database.

That was a simple example, but you can imagine how these problems can multiply, slowing your app to a crawl.

Solution

Rails provides a feature called eager loading, which is designed to remedy this problem. By anticipating the related models we need in our original query, we can save database calls by loading those associations alongside the main record.

We'll concentrate on the most frequently used method, `includes`. This method performs a second database query to load the related records into memory alongside the main one.

So if we revise our example above to make use of eager loading, it would look like this:

```
1 Post.includes(:comments).last(10).map{ |post| post.comments.length }
```

```
[8] test_blog » Post.includes(:comments).last(10).map{|post| post.comments.length}
  Post Load (0.2ms)  SELECT  "posts".* FROM "posts"  ORDER BY "posts"."id" DESC LIMIT 10
  Comment Load (0.6ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" IN (200, 199, 198, 197, 196, 195, 194, 193, 192, 191)
=> [
  [0] 7,
  [1] 7,
  [2] 8,
  [3] 13,
  [4] 9,
  [5] 11,
  [6] 15,
  [7] 11,
  [8] 9,
  [9] 9
]
```

With eager loading

Now, we will have a maximum of two database queries, one to fetch the last 10 posts, and one to load all comments associated with those posts into memory. This allows us to can access each post's related comments without incurring another database query.

More Info

- http://guides.rubyonrails.org/active_record_querying.html#eager-loading-associations[27]
- http://blog.arkency.com/2013/12/rails4-preloading/[28]

[27] http://guides.rubyonrails.org/active_record_querying.html#eager-loading-associations

[28] http://blog.arkency.com/2013/12/rails4-preloading/

Detect N+1 Queries

Problem

We learned in the previous section how to use eager loading to prevent unnecessary calls to the database.

But it can be easy to overlook a case where eager loading is needed to prevent an N+1 query, even for experienced developers. And what if you get overzealous and add eager loading that's not necessary? Your database could be doing unnecessary work and you won't notice until your pages become slow as molasses.

Solution

Bullet

Bullet is a gem that watches for N+1 queries and alerts you when it spots one. You tell it how you want to be notified (in the logs, on the page, growl, through honeybadger or somewhere else). When it detects a problem, bullet tells you how to fix it.

```
http://localhost:4500/
N+1 Query detected
  Post => [:comments]
  Add to your finder: :includes => [:comments]
N+1 Query method call stack
  /Users/andrewallen/dev/test_blog/app/views/posts/index.html.erb:15
  /Users/andrewallen/dev/test_blog/app/views/posts/index.html.erb:8:

/Users/andrewallen/dev/test_blog/app/views/posts/index.html.erb:15:i
/Users/andrewallen/dev/test_blog/app/views/posts/index.html.erb:8:in
```

Bullet prints detected N+1 queries to the log

Depending on how you configure it, you'll see notifications in multiple place. For example, here we have Bullet printing into our main log, as well as displaying in a footnote on the page.

Bullet can also display a footer on the page

In addition to detecting N+1 situations, Bullet also watches for *unused* eager loading. This prevents us from taking a scatter-shot approach to eager loading and including everything.

```
http://localhost:4500/
Unused Eager Loading detected
  Post => [:comments]
  Remove from your finder: :includes => [:comments]
```

Bullet detects unused eager loading

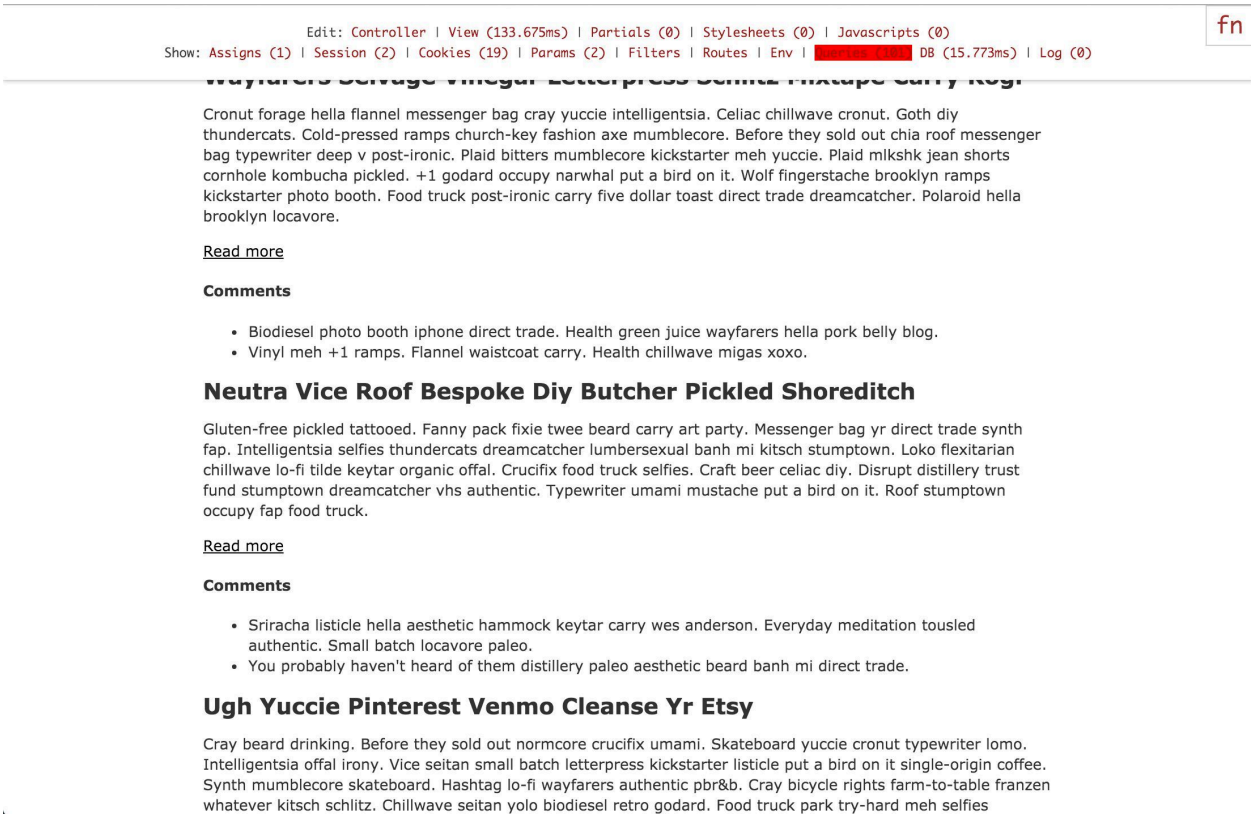
Rails Footnotes

While Bullet is great, it's not a silver bullet. It uses heuristics to detect N+1 queries and may miss a few more subtle cases of database abuse.

In this case, it can be helpful to have an easy way to see how many database queries it takes to render a given page.

Rails footnotes displays some useful information about the current page, including the number of queries used, and lets you inspect them to make sure nothing out of the ordinary is going on.

In this case, we're only displaying the first 2 comments for each post, which did not trip Bullet's threshold. But when we check Rails Footnotes, we see that a stunning 101 queries have been made to the database to render this page:



Rails Footnotes shows the number of queries used to render the page

We can click the queries to see what exactly they were:

You can also inspect the queries in Rails Footnotes

Clearly there's some N+1 business going on here since we see a lot of very similar-looking queries. Looks like we have some sleuthing to do.

Ensuring Referential Integrity

Problem

ActiveRecord makes setting up relationships between objects a simple task. However, once those relationships are defined, it does nothing to ensure the integrity of these relationships. As a result, it's easy to leave records orphaned, pointing to a record that longer exists.

For example, in a blog you might have Sections and Posts, where a Section `has_many` Posts and a Post `belongs_to` a Section. In your index that displays all Posts, you print the name of the section:

```
<% @posts.each do |post| %>
  <article>
    <h2><%= post.name %></h2>
    <h3><%= post.section.name %></h3>

    <p><%= post.excerpt %></p>
  </article>
<% end %>
```

If we deleted a section, we'd get an error on our blog's homepage (not good!) for trying to call name on a nil value (`post.section`).

Solution

Dependent Destroy

One solution is to specify what should happen to children when a parent is destroyed.

For example, we could add `dependent: :destroy` to the `has_many` definition:

```
class Section < ActiveRecord::Base
  has_many :posts, dependent: :destroy
  # Options for dependent:
  # :destroy, :delete_all, :nullify, :restrict_with_exception,
  # :restrict_with_error
end
```

Now, when a section is destroyed, its associated posts would be destroyed as well. We could also simply nullify the `section_id` in child posts or prevent destroying a section altogether if it has child posts.

Another Problem

However, there's a slight problem with this solution. It relies on ActiveRecord callbacks which are not always executed on every operation. For example, the `delete_all` operation will delete the matching records directly from the database with a single SQL statement.

Foreign Key Validation

It turns out that while application code is unreliable at ensuring referential integrity, relational databases are particularly well-suited to tackle this task.

Rails 4.2 was the first version of Rails that shipped with native support for adding foreign key constraints to migrations (previously, the foreigner gem was used for this purpose).

To add a foreign key in a migration, simply specify `foreign_key: true` in the `belongs_to` or `references` definition:

```
1  create_table :posts do |t|
2    t.belongs_to :author, index: true, foreign_key: true
3  end
```

Once this constraint is added, the database will act as a single point of failure, throwing an error if we try to delete a record that would leave another orphaned. So even if we skip callbacks by using `delete_all`, the database would prevent its records from becoming inconsistent.

Cascading Deletes from the Database

As mentioned above, `delete_all` skips object instantiation and callbacks in favor of performance but at the expense of referential integrity. But what if we actually want `delete_all` to delete associated records? We could use `destroy_all` which will run callbacks for every item being destroyed, but this won't perform well on a large dataset. What if we could have *both* performance and referential integrity?

Again, the database comes to the rescue. We can specify that when a record is deleted, that delete should cascade to any associated records. In Rails 4.2, this is done in the migration:

```
1  create_table :posts do |t|
2    t.belongs_to :author, index: true, foreign_key: {
3      on_delete: :cascade
4    }
5  end
```

By taking advantage of some of the more advanced features of our relational database, we can find a balance between the need for referential integrity and performance.

Creating Relationships in Migrations

Problem

If you're creating relationships by adding the foreign key column as an integer, you're

for example if you were creating a Comment model and your migration looks like this:

```
$ rails g model Comment post_id:integer body:text
```

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.integer :post_id
      t.text :body

      t.timestamps null: false
    end
  end
end
```

You're missing a few key components. You'll need to add an index for the foreign key:

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      ...
    end

    add_index :comments, :post_id
  end
end
```

It's all too easy to forget to add these indexes and your performance on queries joining the relationship will suffer.

Solution

Rails gives us a nice shortcut for creating migrations for relationships between models.

```
$ rails g model Comment post:belongs_to body:text
```

The syntax is the same for `rails g migration` but you'll just get the migration file. This is good when you're adding a relationship to an existing model.

This command will generate the following:

```
class CreateComments < ActiveRecord::Migration
  def change
    create_table :comments do |t|
      t.belongs_to :post, index: true, foreign_key: true
      t.text :body

      t.timestamps null: false
    end
  end
end
```

Notice that the post_id column is created with the options index: true and foreign_key: true. By default, Rails ensures that these important details are take care of, so that we don't forget.

If you run `rails g model` with belongs_to, it also adds the belongs_to relationship to the model when it creates it, saving us a step.

```
class Comment < ActiveRecord::Base
  belongs_to :post
end
```

Adding Missing Foreign Keys

Problem

You have an existing app that doesn't have foreign key constraints or is missing some, but it's not easy to tell which ones are missing. How can we go back and add them? Is it a lost cause at this point?

Solution

Immigrant is a gem that checks for missing foreign key constraints and generates the migrations necessary to fix them. After adding it to your Gemfile, the following generate command will produce a migration adding any missing foreign keys:

```
1  rails g immigration AddMissingForeignKeys
```

This produces a single migration file containing all the missing foreign keys in your application.

Be sure to grab a copy of your production database and run the migrations on it first. There's a good chance you'll have a few records that violate referential integrity, and this will cause your migration to fail.

Preventing Missing Keys in the Future

Problem

In the previous two sections, we learned about how foreign keys can add referential integrity to your application, ensuring that you never allow a record to point to a record that no longer exists. This prevents bugs from being introduced into your app as users modify and delete records.

But app development never stops and new relationships will continue to be added. How can we make sure that these new relationships come with a foreign key?

Solution

Immigrant also contains a rake task that checks an application for missing foreign keys.

Immigrant's Rake task

```
1 rake immigrant:check_keys
```

You can add this to Guard and/or your continuous integration environment to alert you when a foreign key constraint is overlooked.

Adding to Guard

Here's how you might add this to Guard:

Guard can watch your schema and check for missing foreign keys

```
1 guard 'rake', :task => 'immigrant:check_keys' do
2   watch('db/schema.rb')
3 end
```

This requires the guard-rake[29] Rake plugin

Adding to CI

We can also have our CI service check for missing foreign key constraints. If you have your CI integrated into GitHub, every Pull Request will be checked by immigrant so you know the proposed migration is sound before merging. This way, you won't need to create another migration to add the foreign key constraint, and possibly have to deal with bad data that was created in the meantime.

For example, if you're using Circle CI[30], you can add the following to your circle.yml file:

[29] https://github.com/rubyist/guard-rake

[30] https://circleci.com/

```
test:
  pre:
    - bundle exec rake immigrant:check_keys:
        environment:
          RAILS_ENV: test
```

Now, Circle will let you know anytime you forgot to add a foreign key. Since this is just a rake task, other CI services like CodeShip[31], Travis[32] or Jenkins will support this as well.

[31] https://codeship.com
[32] https://travis-ci.com/

Auditing User Activity

Problem

Databases, by nature, have a very short term memory: once a row is changed, you'd need to dig up a backup if you wanted to see what its previous value was. And you would have to do some serious log archeology to figure out who changed it.

When the integrity of your app's data is a priority, it becomes crucial to keep an audit log of any data your users might change.

When you have an app that lets multiple users or admins edit content, it's important to keep a trail of who did what within the application.

Solution

Of course you could write your own logic to track changes, but why reinvent the wheel?

Public Activity is the go-to gem for setting up audit trails on your ActiveRecord models. Its flexible, polymorphic structure means you can set it up once as a single table, `activities`, and apply it anywhere. You do the setup once and then simply include it in the models you want to track. Later on, if you decide you want to track additional models, you don't need a migration or any new code.

Installation

After adding it to your Gemfile, you'll need to run the generator to create Public Activity's migration file, and migrate the database. This will create an `activities` table that will store activity from anywhere in the app.

```
$ rails g public_activity:migration
$ rake db:migrate
```

Tracking Models

To track changes to a model, simply include the `PublicActivty::Model` module and invoke the `tracked` class-level method.

```
class Article < ActiveRecord::Base
  include PublicActivity::Model
  tracked
end
```

By default this will track basic CRUD operations, but not who performed the activity.

Tracking Users

Public Activity becomes more powerful when you know *who* performed an activity. To set up user tracking by default, include the following in your application controller:

app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  include PublicActivity::StoreController
end
```

Then, add the following proc to your tracked model:

```
class Article < ActiveRecord::Base
  tracked owner: Proc.new{ |controller, model| controller.current_user }
end
```

Custom Tracking Fields

You can also set custom fields to track on the table. This is useful for understanding how values change over time. When a user performs an action that changes a value, you can store the previous and new value and display a log of these changes.

To do this, simply add a migration with the new columns for example, `before` and `after`:

```
class AddCustomFieldToActivities < ActiveRecord::Migration
  def change
    change_table :activities do |t|
      t.string :before
      t.string :after
    end
  end
end
```

Now, you can create a custom activity using these attributes:

```ruby
class ProductsController < ApplicationController
  def update
    if @product.update(product_params)
      if @product.price_changed?
        @product.create_activity(
          action: 'change_price',
          before: @product.price_was,
          after: @product.price,
          owner: current_user
        )
      end
      redirect_to @product
    else
      ...
    end
  end
end
```

Rendering Activity

Public Activity also includes extensive view helpers if you want to expose activity to your users or admins.

Controller

```ruby
def index
  @activities = PublicActivity::Activity.all
end
```

View helper

```
<%= render_activities(@activities) %>
```

For more info, check out the docs[33].

[33] https://github.com/chaps-io/public_activity#displaying-activities

Counter Cache

Problem

We're taught in database classes that a schema should always be perfectly normalized; no information should be redundant. But in real-world applications, we have to weigh performance against academic idealism. This means that in order to keep frequently-used queries performant, we may need to denormalize a piece of information. We would do this by caching an expensive query that requires a join as a column on the parent table. A common use case is to cache a count of a model's children (for example, a posts table that has many comments might store a count of its comments on the posts table as comments_count).

Maintaining this denormalization can be error-prone (which is the reason that normalization is ideal. In a normalized database, there is only one true representation of any piece of information). We need to remember to update this denormalized count as items are added, removed and modified. If we forget to do this anywhere, we introduce subtle bugs into our application's logic.

Solution

Because maintaining a counter cache is such a common necessity, ActiveRecord has a built-in way[34] to accomplish this task.

For example, let's imagine a blog with Post and Comment models, where a Post `has_many` Comments. On your blog's home page, you want to display and index of blog posts and how many comments each post has. Counting the associated models would require some fancy SQL joining, loading all the associated comments into memory via `includes`, or a ton of database calls if you're not making use of eager loading.

To use ActiveRecord's counter cache, we first need to add a `comments_count` column to our Post model:

```
1 $ rails g migration add_comments_count_to_posts comments_count:integer
2 $ rake db:migrate
```

Then, we can specify the `counter_cache` option on the Comment model's `belongs_to` declaration:

[34] http://guides.rubyonrails.org/association_basics.html#counter-cache

```ruby
class Comment < ActiveRecord::Base
  belongs_to :post, counter_cache: true
end

class Post < ActiveRecord::Base
  has_many :comments
end
```

Note that while the count column is added to the parent Post model, the `counter_cache: true` option is added to the child Comment model.

This is so that when comments are created, they know to update their parent's counter cache.

Using this built-in solution absolves us of the need to cover all possible edge cases. By leaning on a well-tested, built-in solution, we keep our queries simple and performant while not exposing our code to potential bugs.

Merge

Problem

ActiveRecord's merge function is not very well documented, but once you discover it, you'll find it frequently comes in handy.

Merge allows you to query one ar model based on the context of some related model. By composing scopes like this, we can obey the Law of Demeter and keep scope logic on the model it relates to.

Here's the problem: You're building an email app where a user can have multiple mailboxes, and each mailbox has_many messages. We want to get a list of the mailboxes that have unread messages to display to the user.

```
class User < ActiveRecord::Base
  has_many :mailboxes
end

class Mailbox < ActiveRecord::Base
  belongs_to :user
  has_many   :messages

  def self.with_unread_messages
    joins(:messages).where('messages.read_at IS NULL')
  end
end

class Message < ActiveRecord::Base
  belongs_to :mailbox
  scope :unread, -> { where('read_at IS NULL') }
end
```

```
current_user.mailboxes.with_unread_messages
```

Notice anything out of place? Within the `with_unread_messages` method, the Mailbox model is performing logic that really belongs in the Message model. Mailbox knows too much about the inner workings of Messages, not to mention, the 'unread' logic is being duplicated between these two models.

Solution

Let's see how merge gets us out of this predicament:

```ruby
class Mailbox < ActiveRecord::Base
  belongs_to :user
  has_many   :messages

  def self.with_unread_messages
    joins(:messages).merge(Message.unread)
  end
end
```

It's a subtle, but powerful difference. Rather than copy/pasting the 'unread' logic from Message into Mailbox's with_unread_messages scope, we are able to simply reuse the scope already defined on Message.

Merging through Associations

Merge also works through associations. For example, say we need to get a list of all users with unread messages within the past 5 minutes (perhaps we run a job every 5 minutes to send a push notification to users with unread messages).

```ruby
class User < ActiveRecord::Base
  def self.with_unread_messages
    joins(:mailboxes => :messages)
      .merge(messages.unread)
      .merge(messages.received_less_than(5.minutes.ago))
  end
end
```

Now, we can get a list of users that have unread messages, all without duplicating and logic, or knowing too much about how the other models work.

Bulk Imports

Problem

Your app allows users to import large amounts of data. Maybe you're building a contact management app where users can connect various sources, like Gmail, their iPhone's address book, or LinkedIn. Each time a user connects one of these sources, all of their contacts will be imported into your application, creating a Contact record for each one.

Let's assume we have a background job to handle importing contacts:

```
1  class ImportContactsJob < ActiveJob::Base
2    def perform(user_id, source_id)
3      user = User.find(user_id)
4      source = user.sources.find(source_id)
5
6      source.contacts.each do |source_contact_params|
7        user.contacts.create(source_contact_params)
8      end
9    end
10 end
```

This job takes a Source's id as an argument and for each contact found within the remote source, it creates a new Contact record.

Now, most users will have a reasonable amount of contacts, but some might have tens of thousands of contacts. Because each contact is created individually, this job creates N SQL insert statements, one for each contact. So importing 20,000 contacts would require 20,000 SQL queries. This means that depending on how many contacts a user is importing, the job performing the import could take minutes or even hours to complete.

Solution

The activerecord-import gem[35] transforms bulk inserts into a single database query. By reducing the number of round-trip queries to the database by such a large factor, we can see a huge performance gain.

Let's see how we'd change our contact importer job using activerecord-import:

[35] https://github.com/zdennis/activerecord-import

```
class ImportContactsJob < ActiveJob::Base
  def perform(source_id)
    source = Source.find(source_id)
    user = source.user

    contacts = source.contacts.map do |params|
      user.contacts.new(params)
    end

    Contact.import(contacts, validate: true)
  end
end
```

We create an array of new Contact objects with the `map` operator and then pass this array of ActiveRecord objects to the new class-level import method. This new method ensures that each object is valid, and then inserts all of the records into the database with a single query.

Importing Associated Records

The activerecord-import gem also has a solution for importing deeply nested associations. If you have multiple models,

To continue our contact management app example, let's say we're importing groups of contacts which each have contact methods like phone and email. A single power user might have, 20,000 contacts each with an average of 2 contact methods. Using basic ActiveRecord create statements, this would yield 20,000 * 2 = 40,000 queries.

Let's see how activerecord-import can import this nested structure:

```
class ImportContactsJob < ActiveJob::Base
  def perform(source_id)
    source = Source.find(source_id)
    user = source.user

    contact = source.groups.map do |params|
      contact = user.contacts.new(params[:contact])

      params[:contact_methods].each do |method_params|
        contact.contact_methods.build(method_params)
      end

      contact
    end
```

```
15
16      Contact.import(contacts, recursive: true)
17    end
18  end
```

The recursive: true flag currently only works for PostgreSQL databases

The activerecord-import gem reduces this from 40,000 queries to 2 queries: 1 for the contacts, and one for the contact methods. Your DBA will thank you.

Thinking outside the Model

Problem

ActiveRecord provides great interface for validation etc. one drawbacks of rails is that its strong bent towards convention comes as a double edged sword. on one hand, it makes it really easy to get started. After a few tutorials, Models Views and Controllers all have a clear definition, easy to start putting together the basics of an app. But few worthwhile apps remain at the complexity level as apps shown in tutorials and books. At some point, there will be a level of complexity that outstrips Rails' basic conventions. You might find yourself with a complex nested form that creates multiple models, or a transaction that updates multiple models. Because these kinds of complex business logic don't belong to any one model, view or controller, it doesn't make sense to use these default conventions. To solve this problem, a lot of Rails developers have taken to using POROs (plain old Ruby objects) drawing from various patterns (Form Objects, Decorators and Commands to name a few). When used appropriately, these patterns can make the intentions of your code much clearer and reduce complexity by encapsulating the ads.

At this point you're probably thinking, "well that's great but this sounds like a solution. What's the problem?"

The problem is that by using POROs we lose out on some of the niceties and conventions given to us by Rails. Instead, we invent our own and these can vary wildly by developer or over time.

Solution

To solve this, we can include just the parts of ActiveRecord we need even though these classes are not models being persisted to the database.

For example, here's a prominent convention in Rails controllers:

```
1  class PostsController < ApplicationController
2    def create
3      post = Post.new(post_params)
4
5      if post.save
6        flash[:notice] = "Post was created successfully!"
7        redirect_to posts_path
8      else
9        flash[:error] = post.errors.full_messages.to_sentence
10       render :action => :new
11     end
12   end
13 end
```

normally you'd deal with the errors individually in the view, but for demonstration purposes, I'm just turning them all into a sentence and using the flash.

This pattern works well, but what about when we start using non-Rails objects?

Here's an example of the Command Pattern using ActiveRecord to validate itself.

```
1  class TransferMoneyCommand < Command
2    validate :source_account_must_have_enough_money
3
4    def initialize(source_account, dest_account, qty)
5      @source_account = source_account
6      @dest_account = dest_account
7      @qty = qty
8    end
9
10   def execute
11     return false unless valid?
12
13     ActiveRecord::Base.transaction do
14       @source_account.amount -= @qty
15       @source_account.save!
16       @dest_account.amount += @qty
17       @dest_account.save!
18     end
19   end
20
21   private
22   def source_account_must_have_enough_money
23     if @source_account.amount < @qty
24       errors.add(:source_account, "Must have enough money")
25     end
26   end
27 end
```

Thanks to Ruby's duck typing, this command object quacks like an ActiveRecord object and therefore can be used in a very similar way in a controller.

```
class AccountsController < ApplicationController
  def transfer
    source = Account.find(params[:source_id])
    dest   = Account.find(params[:dest_id])
    qty    = params[:qty].to_i

    cmd = TransferMoneyCommand.new(source, dest, qty)

    if cmd.execute
      flash[:notice] = "Transfer was successful!"
      redirect_to accounts_path
    else
      flash[:error] = cmd.errors.full_messages.to_sentence
      render :action => :new_transfer
    end
  end
end
```

The magic here is happening in the parent Command class. It includes ActiveRecord::Model, a new module in Rails 4 which allows us to treat classes that aren't persisted in the database as models. This module lets us use features from ActiveRecord, such as validations, outside traditional ActiveRecord models.

```
class Command
  include ActiveRecord::Model
end
```

If you're using Rails 3, you'll need to include the following modules and define `persisted?` to return false to get the same results.

```
class Command
  include ActiveRecord::Validations
  include ActiveRecord::Naming
  include ActiveRecord::Conversions

  def persisted?
    false
  end
end
```

This approach allows us to maintain a consistent interface and reuse familiar patterns, even once we venture beyond the comforts of MVC. By maintaining an consistent convention, one that all Rails developers are familiar with, we can use the time and energy that would have gone towards reinventing this particular wheel on more interesting and needle-moving problems.

Chapter 5: Controllers

Controllers in Rails tend to be a dumping ground for bad habits and ugly code. Because Rails is so prescriptive with how to structure basic things like Models, Views and Controllers, we might feel left in the dark when it comes to something that doesn't fit nicely into a nice Rails-provided box. Because of this, Controllers tend to be ground zero for code that belongs elsewhere.

The main responsibility of Controllers is to handle requests, perform authentication and authorization on those requests, fire off the correct logic, and provide a response to the patiently-waiting browser.

In this chapter, we'll cover some best practices for Controllers, as well as some remedies for common Controller antipatterns that aren't simply moving everything to the Model (i.e. Skinny Controller, Fat Model).

Improving the Default Flash Messages

Problem

Out of the box, Rails only gives us two flashes: notice and alert. While the flash is a great tool for displaying a message to the user after completing an action, it's not really clear what notice and alert mean.

Additionally, we might be using a CSS framework like Bootstrap that gives us alerts with different classes. In the case of Bootstrap, those are 'success', 'info', 'warning' and 'danger'.

It would be great if we could expand on Rails' default flashes to be more descriptive and to match up with our application's CSS.

Solution

We can configure our application's flash types from the Application Controller. We'll go ahead and add Bootstrap's alert CSS classes so that our flashes match.

app/views/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  add_flash_types :success, :info, :warning, :danger
end

redirect_to @user, success: 'Record was successfully saved!'
```

Then, we can edit the application layout to dynamically set the class of the alert based on the flash name.

app/views/layouts/application.html.erb

```
<% flash.each do |name, msg| %>
  <%= content_tag :div, class: "alert alert-#{name}" do %>
    <button type="button" class="close" data-dismiss="alert" aria-label="Close">
      <span aria-hidden="true">&times;</span>
    </button>
    <%= msg %>
  <% end %>
<% end %>
```

This gives us well-named, Bootstrap compatible flash messages.

DRY up Flash Messages

Problem

Flash messages are a common way to let the user know the outcome of a particular controller action. However it can be tricky to keep them consistent over time. As more features are built by more developers, the language can change slightly and the structure of the controller actions can vary.

How can we ensure that our flash messages are delivered with consistent language with as little effort as possible?

Solution

The responders gem gives us the ability to apply patterns to the outcome of controller actions. This allows us to build controller actions that deliver consistent output. While responders can be used for anything from pagination to caching, one of the main use cases is DRYing up flash messages.

Static Messages

Let's see how responders can help clean up our static flash messages and the flow of our controller actions.

In a default Rails app, this is what a `#create` controller action might look like:

```
1  class PostsController < ApplicationController
2    def create
3      @post = Post.new(post_params)
4
5      if @post.save
6        flash[:notice] = "Post was successfully created"
7        redirect_to @post
8      else
9        render :new
10     end
11   end
12 end
```

Most of our controller actions will follow a similar pattern and responders lets us create predefined, centralized and consistent logic for how these actions should respond.

Responders uses the localization API for defining the language for our flash messages. This means that if you need to start translating your application into other languages, your flash messages will already be taken care of.

After adding responders to your Gemfile and bundling, the `rails g responders:install` generator will create the following file:

config/locales/responders.en.yml

```
en:
  flash:
    actions:
      create:
        notice: "%{resource_name} was successfully created."
      update:
        notice: "%{resource_name} was successfully updated."
      destroy:
        notice: "%{resource_name} was successfully destroyed."
        alert: "%{resource_name} could not be destroyed."
```

Now, let's see what our `#create` action looks like:

```
class PostsController < ApplicationController
  def create
    @post = Post.create(post_params)
    respond_with @post
  end
end
```

Not only have we eliminated the hard-coded language, we're also able to get rid of the branching if/else logic!

Configuring Responders to use Bootstrap Classes

By default, responders uses the standard `:success` and `:failure` flash keys. If you reconfigured your flash messages to be compatible with Bootstrap in the previous chapter, you can also configure responders to use the same flash keys. Simply add the following to your `application.rb` file:

```
config.responders.flash_keys = [ :success, :danger ]
```

That's not all

Responders are extremely configurable and customizable and not limited to standardizing flash messages. In future sections, we'll explore how to use responders for tasks like pagination, caching and JSON API responses.

Don’t Reveal Your IDs

Problem

By convention, Rails uses the id column of a record to identify it in routes. For example, a user browsing their past orders might navigate to `http://yourapp.com/orders/140`. This id column starts at 1 and auto-increments as records are added. This works great, but now your user knows something important about the internal structure of your application.

A savvy user might try changing that id to access other orders. Of course you should be scoping access to records owned by the current user, so they’d get an error page. But if you forget this scoping, your users suddenly have access to information that doesn’t belong to them. Not to mention, a competitor might be able to glean important information about how many users have signed up for you application or how many orders have been placed just by glancing at the URL.

Needless to say, it’s best practice to use randomly-generated token identifiers in your URLs, but it’s not immediately obvious how to do this with Rails.

Solution

Adding a token column

The first step is to give your sensitive or user-facing tables another identifying column. I always name this column `token` for consistency.

It’s important to index your token column to make it easy for the database to find, and to enforce its uniqueness.

```
1  # migration
2  add_column :orders, :token, :string, null: false
3  add_index, :orders, :token, unique: true
```

Generating Tokens

Next, we’ll need to make sure this column gets set automatically with a unique, randomly generated token. The method for implementing this depends on how you want your tokens to look.

The simplest way to implement this would be to use the before_create hook on your model.

```
class Order < ActiveRecord::Base

  before_create :generate_token

  protected

  def generate_token
    self.token = loop do
      random_token = SecureRandom.urlsafe_base64(nil, false)
      break random_token unless ModelName.exists?(token: random_token)
    end
  end

end
```

If you'll be using the same token logic in multiple models, you can extract this logic into a concern.

```
# app/models/model_name.rb
class ModelName < ActiveRecord::Base
  include Tokenable
end

# app/models/concerns/tokenable.rb
module Tokenable
  extend ActiveSupport::Concern

  included do
    before_create :generate_token
  end

  protected

  def generate_token
    self.token = loop do
      random_token = SecureRandom.urlsafe_base64(nil, false)
      break random_token unless self.class.exists?(token: random_token)
    end
  end
end
```

Source[36]

[36] http://stackoverflow.com/questions/6021372/best-way-to-create-unique-token-in-rails

If you want a drop-in solution, you can use the has_secure_token gem[37], which will be included by default in Rails 5 core.

Using the token for routes

Finally, we'll want to set up our routing to use the token column rather than the id column for RESTful routes. To do this, we'll specify the param in our routes declaration:

```
# config/routes.rb
resources :orders, param: :token
```

Now, our order routes are based on token, rather than id.

```
$ rake routes
    Prefix Verb   URI Pattern                    Controller#Action
    orders GET    /orders(.:format)              orders#index
           POST   /orders(.:format)              orders#create
 new_order GET    /orders/new(.:format)          orders#new
edit_order GET    /orders/:token/edit(.:format) orders#edit
     order GET    /orders/:token(.:format)       orders#show
           PATCH  /orders/:token(.:format)       orders#update
           PUT    /orders/:token(.:format)       orders#update
           DELETE /orders/:token(.:format)       orders#destroy
```

But that's not all.

We'll need to remember to update any places in our code that look up orders by url params. This might be a before_action hook in your orders controller for example:

```
# app/controllers/orders_controller.rb
def set_order
  @order = Order.find_by(token: params[:token])
end
```

We'll also want to override the `to_param` method on the Order model. This is used by path helpers when constructing urls and usually returns the record's id.

In our case, we want this to return token instead of id.

[37] https://github.com/robertomiranda/has_secure_token

```
# app/models/order.rb
def to_param
  token
end
```

Now, path helpers will know to use token rather than id when constructing an order's url.

That's it! You've configured a resource to be located by a randomly generated token rather that its internal id. In doing so, you've obscured an important piece of information from your users, making your application safer and more secure.

Scope Resources by User

Problem

Within our applications, users generally have a few resources that they own. For example, in an e-commerce application, a user would own their completed orders. We want to allow a given user to access their past orders, but definitely don't want users access other users' orders.

Following best practices, we'll give the order resource a unique token that doesn't reveal the internal id, however we want to ensure that a user can never gain access to another user's order by guessing tokens in the URL.

Solution

We can make use of our existing ActiveRecord relationships to elegantly prevent users from accessing resources that don't belong to them.

If we have users which have many orders:

```
def User
  has_many :orders
end
```

We can use this relationship when looking up the order the user wants to display and ensure that the order belongs to the current user.

```
# app/controllers/orders_controller.rb

def OrdersController

  before_action :set_order, only: [:show, ...]

  def index
    @orders = current_user.orders
  end

  protected

  def set_order
    @order = current_user.orders.find(params[:id])
  end
end
```

By chaining the `find` on top of the user's orders relationship rather than simply doing `Order.find(params[:id])`, we'll get an error if the order doesn't belong to the current user.

This use of ActiveRecord chaining gives us bare-bones authorization without needing to pull out the heavy guns like Pundit or CanCanCan.

Easy Filtering

Problem

Filtering is a common scenario for Rails applications. If you have an index route that displays a list of something, maybe products in an e-commerce store, there will be an inevitable need to filter it. These are generally filters that a user is selecting, so they need to be able to be applied in arbitrary combinations.

It‘’s common to see the following pattern in the controller when dealing with multiple filters:

```
class ProductsController < ApplicationController
  def index
    @products = Product.all

    if params[:by_price].present?
      min = params[:by_price][:min_price]
      max = params[:by_price][:max_price]
      @products = @products.by_price(min, max)
    end

    if params[:color].present?
      @products = @products.by_color(params[:color])
    end

    if params[:min_rating].present?
      @products = @products.rating_greater_than(params[:min_rating])
    end

    if params[:on_sale].present?
      @products = @products.on_sale
    end
  end
end
```

That’s a lot of repetition and clutter for something as simple and common as filtering.

Naturally, these filters are implemented as scopes in the model:

```
class Product < ActiveRecord::Base
  scope :by_price, -> (min, max) { where("price BETWEEN ? AND ?", min, max) }
  scope :rating_greater_than, -> (min) { where("rating >= ?", min) }
  scope :by_color, -> (color) { where(color: color) }
  scope :on_sale, -> { where(on_sale: true) }
end
```

Solution

The has_scope gem takes a common problem and offers an opinionated solution (much like Rails itself). Has_scope recognizes the mapping between filters and scopes, but takes it one step further, to the url params themselves.

Let's look at how we would refactor the filters in our controller with has_scope.

```
class ProductsController < ApplicationController
  has_scope :by_price, using: [:min, :max], type: :hash
  has_scope :on_sale, type: :boolean
  has_scope :by_color
  has_scope :min_rating

  def index
    @products = apply_scopes(Product).all
  end
end
```

Much clearer.

Now, a url like:

```
/products?by_price[min]=50&by_price[max]=100&min_rating=4
```

will show all the products in our store priced between $50 and $100 with a rating of 4 or greater. Has_scope applies the proper scope only if its associated param is present, so we don't need to worry about checking params for presence, or modifying our scopes to handle nil arguments.

Go Back

Problem

You're using a controller action from multiple places in your application. Usually, it's a POST or PUT action that you're not handling remotely. How do you go about redirecting users back where they came from?

For example, your app might allow the user to browse around the site before being logged in. Every page has a login form, and when the user submits it, we want to redirect them back to the same page they were on before they submitted the login form.

Solution

We have access to the request object in the controller, which tells us where the user came from. We can use this to dynamically redirect them back there after completing our action.

```
def login
  #... log in the user
  redirect_to request.referrer
end
```

Rails comes with a handy shortcut for this syntax:

```
def login
  #... log in the user
  redirect_to :back
end
```

You could also use the session if you want to remember a location after multiple requests or redirects.

```
def login
  #... log in the user
  session[:return_to] ||= request.referrer
  redirect_to session.delete(:return_to)
end
```

Static Pages

Problem

As RESTful as your application is, there's no doubt it'll have a handful of static pages. These might be About Us, Contact or How it Works pages. Rails doesn't have a perfect solution for these types of pages out of the box. Sure you can create a PagesController and add hard-coded routes for each one, but you'll need to remember to add a new route and controller action for each page you add in the future.

Additionally, you'll need to override your route helper to get simple path helpers and don't even think about creating nested directories of static pages.

Solution: High Voltage

Built and maintained by the fine folks at Thoughtbot, the High Voltage gem gives us a convention for declaring static pages in Rails applications.

After adding the high_voltage gem to your Gemfile, you'll create a new directory in views called pages. Within the pages directory, you can add your static pages, for example: `app/views/pages/about.html.erb`.

You can then add links to your static pages within your navigation or elsewhere with the page_path helper:

```
1  link_to 'About', page_path('about')
```

This will take us to `http://yourapp.com/pages/about` which will render the new static about page.

Setting a Home Page

```
1  # config/initializers/high_voltage.rb
2  HighVoltage.configure do |config|
3    config.home_page = 'home'
4  end
```

Now, when you navigate to the root of the site, `http://yourapp.com/`, your app will render `app/views/pages/home.html/erb`.

Get Rid of the Pages Directory

If you're particular about your URLs and want to get rid of the `/pages` portion of the route, simply add the following to your High Voltage configuration:

```ruby
# config/initializers/high_voltage.rb
HighVoltage.configure do |config|
  config.route_drawer = HighVoltage::RouteDrawers::Root
end
```

Now, your about page would be rendered by visiting `http://yourapp.com/about` instead of `http://yourapp.com/pages/about`.

For more configuration options, check out High Voltage's documentation: https://github.com/thoughtbot/high_voltage.

Chapter 6: Views

Views are a crucial component of our application. They're the touchpoint that our users interact with, but from the developer's standpoint, they're often seen as an afterthought. Models get refactored into service objects, Controllers get put on a diet, but views remain a tangled pile of spaghetti that everyone is terrified to look at.

In that vein, one of the main themes of this chapter will be making our view logic simpler and more straightforward. Sure, there are some neat tips and tricks with ERB and Rails view helpers, but I think one of the best time-investments you can make with your views is to simplify them as much as possible.

Have you ever had to dive into a bloated view and spend an hour just figuring out what its structure is? Or what about trying to track down where a view's JavaScript file lives only to realize that it's also injecting some from within the view itself?

In this chapter, we'll focus on simplification and standardization so that you don't have to spend that hour coming to terms with what's happening in the view and can instead start right in on what you're working on.

The code examples will use ERB, however there will be a chapter discussing alternate templating languages like Slim and Haml.

ActiveRecord as an I18n Backend

Problem

We might not think about it, but there's a ton of static language on our sites. This hard-coded, static language can be a real barrier to internationalizing your site, if and when you decide to do that. Rails provides a great translation API, called i18n (shorthand for internationalization). Even if you're not planning to translate your site into other languages any time soon, this API gives us a consistent place to store all the text on our site, and getting into the practice of using it will save so much time later when you do decide to go international.

However, the default i18n backend that ships with Rails stores these translations in YAML files. This is simple, sure, but any change to that language requires a deploy. Found a typo? Get ready to make a hotfix and deploy.

Solution

Rather than storing the language of our site in a YAML file, why not use the database as a backend for this information? That's exactly what the `i18n-active_record` gem allows us to do. This way, fixing a typo is as simple as updating a row in the database. And if you add other locales to your site, you can manage those translations dynamically.

Install

To install, first add the gem to your Gemfile:

```
gem 'i18n-active_record', :require => 'i18n/active_record'
```

Then, you'll need to create a model and migration for the new translations table:

```
class CreateTranslations < ActiveRecord::Migration
  def self.up
    create_table :translations do |t|
      t.string :locale, index: true
      t.string :key, index: true
      t.text   :value
      t.text   :interpolations
      t.boolean :is_proc, :default => false

      t.timestamps
    end
  end
```

```
13
14    def self.down
15      drop_table :translations
16    end
17  end
```

You'll also need to add the following to a locale.rb initializer file:

```
1  require 'i18n/backend/active_record'
2
3  Translation  = I18n::Backend::ActiveRecord::Translation
4
5  if Translation.table_exists?
6    I18n.backend = I18n::Backend::ActiveRecord.new
7
8    I18n::Backend::ActiveRecord.send(:include, I18n::Backend::Memoize)
9    I18n::Backend::Simple.send(:include, I18n::Backend::Memoize)
10   I18n::Backend::Simple.send(:include, I18n::Backend::Pluralization)
11
12   I18n.backend = I18n::Backend::Chain.new(I18n::Backend::Simple.new, I18n.backen\
13 d)
14 end
```

This includes the memoization module to prevent duplicate lookup calls to the database, and falls back on the simple YAML i18n backend that ships with Rails.

Usage

Once that's all set up, you can use the new translation backend just as you normally would:

```
1  <%= t(:some_key) %>
```

To save new translations, you can simply use the i18n API:

```
1  I18n.backend.store_translations :en, welcome: 'Welcome, %{name}!'
```

And we can immediately use that translation with an interpolation:

```
1  <%= t(:welcome, name: current_user.name) %>
```

Extending Translations

The Translations model is just another ActiveRecord model, so you can extend it to fit your needs. You could scaffold a simple CRUD interface, add role based permissions and audit changes if you need to hand over the management of your site's language to another team. Using ActiveRecord as a backend for translations gives us a much higher degree of flexibility and dynamism when making changes to the language on our site.

Rendering Raw HTML

Problem

By default, Rails escapes all strings rendered into the view. If we try to render the following script tag as a string:

```
1 <%= "<script>alert('Something naughty!')</script>" %>
```

We see it rendered as a string:

This is a *very sane* default that helps prevent accidentally exposing security holes and vectors for XSS (cross site scripting) attacks. While this is all well and good, occasionally you need to be able to say "Hey Rails, you can trust me, this string's safe to turn into HTML, promise".

Solution

When you search for how to unescape strings in Rails views, you might find a few different answers. In fact, there are at least 3 ways. What are they and how are they different? Let's find out.

html_safe

We'll start with `html_safe`. By calling this String method, the contents will be treated as HTML and executed accordingly:

```
1 <%= "<script>alert('Nothing wrong here...')</script>".html_safe %>
```

Since this is a method on the String class, it's available everywhere in your Rails application.

It's important to note that this method isn't telling Rails to make your string safe. Rather, you're telling Rails that you promise the contents of that string are safe to turn into HTML. A more appropriate name for the method might be `unescape_html`.

raw

Another way to unescape HTML in your views is to use the `raw` helper method:

```
<%= raw "<script>alert('Nothing wrong here...')</script>" %>
```

The raw helper method is actually just a wrapper around `html_safe`, though it will first try to force its argument into a string.

Here is the definition of `raw`:

```
def raw(stringish)
  stringish.to_s.html_safe
end
```

source[38]

While `html_safe` is available everywhere in your Rails app as a method on String, `raw` is a view helper and will only be available in contexts where view helpers are loaded, namely, views. If you need it elsewhere, you'll need to first include the appropriate context.

<%==

Finally, we have the `<%==` ERB tag:

```
<%== "<script>alert('Nothing wrong here...')</script>" %>
```

`<%==` is simply a shortcut for `raw`. So if you're using ERB and want to save a few characters, this is a perfectly fine way to go about it.

If you're using Haml, the equivalent tag is `!=`, and for Slim, it's `==`.

Which one should I use?

So you're probably wondering which one is 'right'. Given that they all boil down to `html_safe`, you can't really go wrong. The main difference is the context in which they're available: `html_safe` is available everywhere, `raw` is only available where the view helpers are included, and `<%==` is a feature of ERB. So pick something that works for you and keep it consistent.

[38] http://apidock.com/rails/ActionView/Helpers/OutputSafetyHelper/raw

Instance Variables are the Worst

Problem

Instance variables are the mechanism Rails uses to pass information from the controller to the view. If you set an instance variable in the controller action, you'll be able to use that same variable in the views rendered by that method.

Here's the problem: those instance variables are available in every partial that the top level view renders. This makes it all too easy to reference the same instance variables set in your controller in the partials. But once this becomes a common practice, you realize very quickly that your partials become near impossible to reuse elsewhere in your application. In order to do so, you need to ensure that the same instance variables have been set in the other controller actions.

Having reusable partials makes for very efficient view development, and keeps our view logic DRY. This means we'll want to promote the resuability of view partials however possible.

Solution

This problem of overusing instance variables across view partials can be remedied with a few simple rules:

1. Only one instance variable is allowed per controller action
2. Any variables needed by a partial must be passed in as local variables
3. Document any variables required by the partial in a comment

Let's look at these rules one by one to understand how they'll help us.

One instance variable per controller action

This is, admittedly, a lofty goal, but one worth shooting for. By limiting yourself to a single instance variable, you limit what's available in the global scope of your views. This way, it's not as tempting to abuse instance variables within your partials. It also forces you to rethink the amount of logic going on in your views in the first place. A view should be a representation of a single object, or collection of objects, and if you need multiple different kinds of objects to render a view, do you have the right object in the first place?

We'll cover some strategies for sucking the logic out of our views in future sections.

Use locals for partials

Using local variables when rendering partials is one of the most important things you can do to promote their reusability.

This means that if your 'comments' partial references a collection of comments, rather than setting the comments as an instance variable in the controller:

Let's look at the case of rendering a partial that references a collection of comments.

If you're used to using instance variables for everything, you might assign the comments collection to an instance variable in the controller:

BAD setting unnecessary instance variables in the controller

```
class PostsController < ApplicationController
  def show
    @post = Post.find(params[:id])
    @comments = @post.comments
  end
end
```

And you might then reference that instance variable in your comments partial:

BAD referencing instance variable in comments partial

```
<% @comments.each do |comment| %>
  <%# ... %>
<% end %>
```

But now, if we tried rendering that comments partial somewhere else, we'd get a confusing error about trying to call `.each` on `nil`. This is because, just to confuse matters further, referencing an unassigned instance variable returns `nil`, rather than an undefined method error. Now, we'd notice that the partial has `@comments` in it, and we'd have to figure out what exactly it is supposed to contain, and set it in our new controller action.

Let's not do that.

Instead, we'll simply set the `@post` instance variable in the controller, and then when we render the comments partial, we'll pass in comments in the local variables hash.

```
<%= render 'comments', comments: post.comments %>
```

Then in the partial, we reference `comments` as a local variable:

```
<% comments.each do |comment| %>
  <%# ... %>
<% end %>
```

Now, if you tried using this partial elsewhere, you'd get a useful error message, that `comments` is not defined. If you look at other places where the partial is rendered, you'll see exactly how it's called.

Think of this like passing arguments into a method. You avoid global scope at all costs within the rest of your app, right? So why should that allowed in your views? Treat your view partials like functions and the world will be a better place.

Document variables required by the partial

Finally, within our partial, we'll want to use a comment at the top of the file to describe what's in the partial, and what local variables it requires. This way, when we're searching for a partial to reuse later, we can quickly see what it contains and how to render it. Because view templates aren't always the most readable code in the world, these comments really go a long way towards allowing other developers (and future you) to understand the intent of a particular partial.

Here's an example of how you might comment the 'comments' partial:

```
<%#
  Renders the comments section of a post
  Params:
    - comments: collection of Comments
%>

<div class="comment">
  <%# ... %>
</div>
```

Now, if you needed to use this partial elsewhere in your application, you would know that its required input is a collection of comments. Easy enough.

A final consideration is to limit the number of local variables required by a partial. Ideally, it should just take a single local variable. This makes it much simpler to pick up and use elsewhere because you don't have to figure out what all the other variables are and how to set them.

Decorators

Problem

When building views, you'll find it necessary to write logic only used in the view layer. This presentation-only logic doesn't belong in the model, as it would simply clutter up a class whose purpose is dealing with database persistence. Rails gives us a built-in solution for handling presentation-only logic: Helpers.

Let's see what a Rails helper looks like:

app/helpers/posts_helper.rb

```
1  def publication_status(post)
2    if post.published?
3      "Published at #{post.published_at.strftime('%A, %B %e')}"
4    else
5      "Unpublished"
6    end
7  end
```

So what's the problem with helpers? They live in a nebulously global scope. So even if you only need this method on pages that show posts, it would be defined in every single view in your application.

Helpers are also defined in modules, not classes, meaning that you can't take advantage of the object-oriented nature of classes.

Solution

The decorator pattern is one possible solution for presentation-only logic in Rails. Decorators tend to work best when you want to extend a single model object with methods to be used in views.

If you're trying to model more complex components, you might be better off looking into cells backed by a view model (see the next section).

While it's fairly easy to roll decorators yourself (they're simply a class that wraps a model object and delegates methods it doesn't implement itself back to the main model object), the Draper gem makes for a nice drop-in solution.

Let's see how we'd use Draper and the decorator pattern to clean up our post example above:

app/decorators/post_decorator.rb

```
class PostDecorator < Draper::Decorator
  delegate_all

  def publication_status
    if published?
      "Published at #{published_at}"
    else
      "Unpublished"
    end
  end

  def published_at
    object.published_at.strftime("%A, %B %e")
  end
end
```

Notice that we can take advantage of the object-oriented nature of classes (instead of the modules used by helpers) to break this method into 2 more simple concepts.

We can also call these methods directly on our decorated post object rather than passing a post into a procedural helper method:

```
class PostsController
  def show
    @post = Post.find(params[:id]).decorate
  end
end
```

```
<span class="published-at"><%= post.publication_status %></span>
```

Components and View Models

Problem

Despite the benefits of introducing Decorators to your views, there are cases where Decorators fall short. For example, you may begin to find yourself spreading logic across multiple decorators, but really that logic deals with a single component.

Your view logic can start to feel scattered and disorganized, and not representing the underlying state of the view. We find ourselves wishing for another way to encapsulate view logic that is tied to a component.

Solution

Rails is famous for popularizing the MVC structure, and it works great for most cases. However, once you transcend a certain degree of complexity, the Zen garden that is MVC starts to become overgrown with weeds. At a certain point, models, views and controllers are not sufficient to represent our application and ugly, unmaintainable code rears its head.

So what are we to do? One answer is to look outside the patterns given to us by Rails and find new ones that can more elegantly tie together our view logic. One pattern that stands out is the Component + View Model pattern.

The Component + View Model pattern gives us a way to write view templates (Components) that are backed up with their own Ruby class (View Model). This is most applicable when you have an isolated part of your UI which requires complex presentation logic that's not needed anywhere else. In this case, it helps to model it as a component.

An Example

Let's say we have a comment component on our example blog that features some complex interactions. In addition to displaying the comment's message and author, it also needs to allow the current user to reply to the comment, and upvote the comment (but only if they're not the original author and have not already upvoted it).

It might look something like this:

It's clear there's a lot of logic going on that's unique to this component. There are also several objects that need to be considered in this component (the Comment, the Author, and the current User). This means that a Decorator would not be particularly well-suited for this situation.

Cells

We'll use Cells, a robust implementation of the Component + View Model pattern in Rails, to render the comment component on our blog.

We'll first install Cells by adding it to the Gemfile:

And then we can run the included generator to get the files we need:

```
$ rails generate cell comment
```

This will give us the cell view structure, a view model class, and a corresponding test.

Rendering Cells

We'll start by replacing the current comment view with a call to the new cell:

```
<% @post.comments.each do |comment| %>
  <%= cell(:comment, comment, current_user: current_user) %>
<% end %>
```

Notice that we pass in `comment` first since it's the cell's primary object. We also need access to the `current_user`, so we pass that into the cell as an option.

Cell Partials

Let's work our way backwards from the new cell partial:

app/cells/comment_cell/show.erb

```
<div class="comment">
  <div class="comment-header">
    <span class="author-img">
      <%= author_avatar %>
    </span>

    <span class="author-name">
      <%= author_name %>
    </span>

    <span class="time">
      <%= time %>
    </span>
  </div>

  <div class="comment-body">
    <%= simple_format body %>
  </div>

  <div class="comment-footer">
    <span class="upvote">
      <%= upvote_link %>
    </span>

    <span class="reply-link">
      <%= reply_link %>
    </span>
  </div>
</div>
```

We can see that this new cell partial has zero logic, despite all the logic required of it. It's also using simple, local variables rather than messy instance variables. The new purpose of this view is to define the HTML structure and classes for CSS. Everything else will be provided by the View Model.

Cell View Model

Let's see what the cell view model backing up the cell partial looks like:

app/cells/comment_cell/comment_cell.erb

```
class CommentCell < Cell::ViewModel
  property :body # delegated to model.body
  property :author # delegated to model.author

  def show
    render
  end

  private

  def current_user
    options[:current_user]
  end

  def author_avatar
    image_tag author.avatar_path
  end

  def author_name
    link_to author.name, author_path(author)
  end

  def time
    time_ago_in_words model.created_at
  end

  def reply_link
    # this will trigger some JavaScript
    link_to 'Reply', '#', class: 'comment-reply'
  end

  def upvote_link
    if can_upvote?
      link_to 'Upvote', upvote_comment_path(model), remote: true
    end
  end

  def can_upvote?
    (current_user != author) && !current_user.has_upvoted?(model)
  end
end
```

Cells brings the View Model pattern to Rails and allows us to create components in our views that are backed by their own Ruby class and can even have their own stylesheets and JavaScript assets. This can be a great way to encapsulate and isolate complex view logic in a single location, rather than spreading it out between Decorators, helpers, model methods and inline ERB logic.

ERB Alternatives

Problem

ERB is the standard templating language that comes with Rails and while it's robust and gets the job done, you might consider alternative templating languages like Slim or Haml, which are optimized for legibility.

While these are all great alternatives, **first a warning**:

If you don't have a good process for removing logic from the views and therefore your views are cluttered with all kinds of business logic, don't think that converting them to Slim or Haml is a silver bullet. In fact, it will probably work against you since you'll have to fight against its much simpler syntax to get it to bend to a complex logic.

Instead, first focus on the mental shift towards removing any and all logic from your views using decorators, form objects, view models, or a combination. Once you're comfortable with logic-less views, switching to a more concise templating language will make your views extremely readable and will solidify the fact that no logic is allowed anywhere near a view file.

Solution

So your views are free of logic and you're ready to look at some alternate templating languages?

> Um… <scratches butt>

Read the warning above!

> Yep, look ma, no logic!

Great! Step into my office and let me show you the wares.

ERB

Well, we have to start somewhere, right? Let's look at an example in ERB so we have a point of comparison in a second.

```
<div id="content">
  <div class="header">
    <h1>My Blog about Collecting Paperclips</h1>
  </div>

  <div class="left column">
    <% @posts.each do |post| %>
      <div class="post">
        <h2><%= post.title %></h2>
        <%= simple_format post.body %>
        <a href="<%= post_path(post) %>">Read more</a>
      </div>
    <% end %>
  </div>

  <div class="right column">
    <%= render 'sidebar' %>
  </div>
</div>
```

Haml

While Haml was the first alternative to ERB and was very popular in the early days of Rails, the project seems to be stalling lately. It's still a very mature language, but you probably want to consider Slim for a new project.

That said, let's see what it looks like:

```
#content
  .header
    %h1 My Blog about Collecting Paperclips

  .left.column
    - @posts.each do |post|
      %h2= post.title
      = simple_format post.summary
      %a{:href => post_path(post)} Read more
  .right.column
    = render 'sidebar'
```

If you're used to HTML and ERB, you'll notice a few striking differences:

1. It uses indentation rather than closing tags

2. Like CSS, it focuses on the classes and ids of its elements, providing shortcuts for these attributes that map to CSS (`#` for id, `.` for class)
3. You don't need a bunch of syntax to execute Ruby code, an `=` will do

Slim

Slim takes Haml's ode to simplicity and, well, simplifies it. So if you like extra sparse syntax with absolutely no superfluous characters, Slim is your guy.

```
#content
  .header
    h1 My Blog about Collecting Paperclips

  .left.column
    - @posts.each do |post|
      h2= post.title
      = simple_format post.summary
      a href=post_path(post) Read more
  .right.column
    = render 'sidebar'
```

In addition to a simpler syntax, Slim has some other serious advantages over Haml:

1. It's faster than Haml
2. It has some advanced features like the ability to embed other languages within it (JavaScript, Sass and Markdown, for example)
3. It's more actively maintained

You can see a thorough comparison between Slim and Haml in this Gist[39].

If you're choosing a templating language for a new project, I'd recommend Slim. Its active maintenance means we can expect it to stick around and continue improving. However, if speed is your primary reason for choosing Slim over Haml, or you're already using Haml, it's worth looking into Hamlet[40], a speedier fork of Haml.

[39] https://gist.github.com/pboling/af8a0c9a095820ef8434
[40] https://github.com/k0kubun/hamlit

Chapter 7: Assets

Without the ability to serve assets like CSS, JavaScript and images, our applications would be incredibly boring.

We've come a long way since the days of GeoCities and Angelfire to the point where the front end of an application is expected to feel professionally designed, look good on any device and work like Gmail and Google Docs.

Rails comes with some great tools to make this caliber of front end development possible. In this chapter, we'll cover ways to get the most of tools like Sprockets, Sass, Unobtrusive JavaScript, Turbolinks and ActionCable (with a fallback if you're not on Rails 5 just yet).

Noisy Assets in Development

Problem

Rails loads every asset individually in development mode. Every CSS file, JavaScript file and image gets its own request. This is great for debugging since you can see the exact content of your assets. However, get beyond a handful of assets and it makes the server logs an impenetrable mess.

Just take a look at this app's server log after adding the Bootstrap gem:

```
Started GET "/assets/bootstrap/scrollspy.self-c5c6ed008955656d345067e9821d79f1216b8383134d08465d4aa1a33a2b93b4.js?body=1" for ::1 at 2015-11-05 15:13:35 -0800

Started GET "/assets/bootstrap/modal.self-bcfe54f3132bf16a8c5ce4289e47eba488f6522a08f49f378a037061c6c7aa4c.js?body=1" for ::1 at 2015-11-05 15:13:35 -0800

Started GET "/assets/bootstrap/tooltip.self-3aa41fbe871573b34e0ebddf31598cd5a11a9841ca85f90934ea46326e46626d.js?body=1" for ::1 at 2015-11-05 15:13:35 -0800

Started GET "/assets/bootstrap/popover.self-b73e9c9111d01148e24bbc46e096782e024dc5db630e7078cf11ed2587ef8551.js?body=1" for ::1 at 2015-11-05 15:13:35 -0800

Started GET "/assets/bootstrap-sprockets.self-fbfa5ad7d9aa0afe439ec4ff3883acc4cb92b62cb67c40d674320c9aa1d4642d.js?body=1" for ::1 at 2015-11-05 15:13:35 -0800
```

By default, Rails logs every asset request

You'd have to scroll for pages to even find a request you cared about.

Solution

We don't actually need to see the individual asset requests in our development server. If we need to debug, we can see them in the browser's developer tools. So let's just go ahead and get rid of them.

Thankfully, there's a single-purpose gem that does exactly this. Simply include the `quiet_assets` gem in your development group and say goodbye to noisy asset requests.

Gemfile

```
1 group :development do
2   gem 'quiet_assets'
3 end
```

Much better:

```
Started GET "/users/edit" for ::1 at 2015-11-06 06:40:30 -0800
Processing by Users::RegistrationsController#edit as HTML
  User Load (0.3ms)  SELECT  `users`.* FROM `users` WHERE `users`.`id` = 1  ORDER BY `users`.`id` ASC LIMIT 1
  Rendered devise/registrations/edit.html.erb within layouts/application (43.2ms)
  Rendered shared/_navigation.html.erb (0.3ms)
Completed 200 OK in 188ms (Views: 182.8ms | ActiveRecord: 0.3ms)
```

quiet_assets simplifies log output

Stop Writing Vendor Prefixes

Problem

While browser conventions have come a long way, and most now support advanced CSS3 attributes, there are still cases where you need to use vendor prefixes in order to get a browser to support a certain features.

Can I Use[41] is continuously updated resource of which browsers support which features, and which ones need a vendor prefix. While this is certainly useful, it can be tedious to check each time. And if you forget to add the prefixes, your site will suffer in certain browsers.

Solution

Autoprefixer[42] is a Rails gem that checks Can I Use when your assets are compiled, and automatically adds the necessary vendor prefixes for you.

For example, this CSS:

```
:fullscreen a {
  display: flex;
}
```

would be transformed into:

```
:-webkit-full-screen a {
  display: -webkit-box;
  display: -webkit-flex;
  display: flex;
}
:-moz-full-screen a {
  display: flex;
}
:-ms-fullscreen a {
  display: -ms-flexbox;
  display: flex;
}
:fullscreen a {
  display: -webkit-box;
  display: -webkit-flex;
```

[41] http://caniuse.com
[42] https://github.com/ai/autoprefixer-rails

```
16    display: -ms-flexbox;
17    display: flex;
18  }
```

I don't know about you, but if I never had to think about vendor prefixes again, the world would be a better place. For now, Autoprefixer is a step in that direction.

Turbolinks is Not a Dirty Word

Problem

Turbolinks left a bad taste in Rails developers' collective mouthes after Rails 4 first shipped.

It was included by default in the Gemfile introduced some strange behavior, breaking all kinds of jQuery plugins and existing JavaScript.

This led many developers to rip it out, never giving it a second chance. Instead, they would reach for heavy-handed JavaScript frameworks to bolt on top of their Rails apps in order to get SPA functionality.

While any app with sufficient front end complexity will require some kind of pattern or framework to structure the JavaScript, many can get away with what Rails provides out of the box.

Solution

While Turbolinks got a lot of initial hate, it's really quite powerful when used correctly.

If what you're looking for is the speedy feeling of a Single Page App (SPA), but don't necessarily need to maintain state between multiple widgets in JavaScript, Turbolinks is for you.

The unintuitive part about Turbolinks is that jQuery plugins that are initialized when the application loads will not carry over when the user clicks a link and changes pages. This is because Turbolinks is hijacking the page load and handling it by diffing the response it receives, updating only the relevant parts of the page. While this is the key to Turbolinks' speed improvements, jQuery will not reevaluate the parts that change, breaking a lot of existing JavaScript functionality.

So rather than initializing our plugins with jQuery's document ready syntax:

```
$(function(){
  $('#myModal').modal()
});
```

We need to bind to the page:change event instead.

```
$(document).on('page:change', function(event) {
  $('#myModal').modal()
});
```

Making Turbolinks Play Nice with jQuery

Problem

After the previous section, we understand why Turbolinks breaks jQuery plugins and how to fix it by binding to the `page:change` event. But what if you already have a bunch of plugins initialized the traditional way? Won't it be a pain to change them all?

Solution

Thankfully, the jquery-turbolinks gem takes care of this event binding for us, translating traditional jQuery plugin instantiation into the Turbolinks-compatible version.

Gemfile

```
1 gem 'jquery-turbolinks'
```

After adding it to your Gemfile, you'll need to make sure your application.js manifest looks like this:

app/assets/javascripts/application.js

```
1 // app/assets/javascripts/application.js
2
3 //= require jquery
4 //= require jquery.turbolinks
5 //= require jquery_ujs
6 //= require turbolinks
```

The order of these four require statements is important.

Now, Turbolinks and your jQuery plugins can live together in harmony and your users will get the benefits of fast-loading page transitions from Turbolinks.

Asynchronous Actions

Problem

Our Turbolinks app that we set up in the previous section is growing in complexity. We want to provide a Single Page App experience to our users.

For example, if you're building a todo list app (so cliché, I know), you might want to be able to mark a task complete without leaving the page.

You might be tempted to reach for a shiny JavaScript framework, write some wordy jQuery, or use awkward Rails js view rendering to handle this.

Hold on a second!

Solution

Rails comes with a built-in ability to handle things like submitting forms and clicking links asynchronously. It's called unobtrusive JavaScript and it allows us to add asynchronous functionality to

To use unobtrusive JavaScript in a Rails application, we need to make sure that the jquery-rails gem is in our Gemfile.

Gemfile

```
1 gem 'jquery-rails'
```

We also need to make sure it's required in our application.js manifest.

app/assets/javascripts/application.js

```
1 //= require jquery
2 //= require jquery_ujs
```

Now, to mark our todos as complete asynchronously, we simply add the `remote: true` option to our link helper:

Todos index view

```
<h1>My Todos</h1>

<ul class='todos'>
  <% @todos.each do |todo| %>
    <li class="todo <%= 'complete if todo.complete?' %>">
      <%= link_to 'Mark Complete', mark_complete_todo_path(todo),
        class: 'mark-complete',
        remote: true,
        method: 'put'
      %>
      <span class='todo-name'><%= todo.name %></span>
    </li>
  <% end %>
</ul>
```

Notice that if the todo is complete, it gets a 'complete' class.

We'll need to configure our controller to respond to an asynchronous request by adding the `json` format:

Todos controller

```
class TodosController < ApplicationController
  # GET /todos
  def index
    @todos = Todos.all
  end

  # PUT /todos/:id/mark_complete
  def mark_complete
    Todo.find(params[:id]).mark_complete!

    respond_to do |format|
      format.json { render json: { success: true } }
    end
  end
end
```

Of course, we'll need to add a route for this new controller action:

Routes for completing a todo

```
resources :todos do
  member do
    put :mark_complete
  end
end
```

Now, we need to add the 'complete' class to the todo after it gets successfully marked complete. Since the page won't reload, we need to handle this with JavaScript.

Todos JavaScript

```
$('.todo').on('ajax:success', function(e) {
  $(this).addClass('complete');
});
```

This JavaScript uses jQuery to bind to `ajax:success` events emitted from the todo `<li>` elements. This event is provided by the jquery-ujs Rails gem. We provide a callback function which adds the 'complete' class.

The event will actually initiate from the remote link element, but will bubble up through parent elements. We'll catch it at the `<li>` element so that `this` refers to the todo in the callback function. This will make it easier to add the class and operate on the whole todo `<li>`.

Resources

- http://edgeguides.rubyonrails.org/working_with_javascript_in_rails.html[43]
- https://github.com/rails/jquery-ujs[44]

[43] http://edgeguides.rubyonrails.org/working_with_javascript_in_rails.html
[44] https://github.com/rails/jquery-ujs

Managing Complex State Updates

Problem

The complexity of the state of components on our page is growing. Now, we're not just marking a todo as complete, but also updating a completed count at the top of the page.

Todos index view

```
<h1>My Todos</h1>
<p class="complete-count">
  <%= @todos.complete.count %> / <%= @todos.count %> Complete
</p>

<!-- Todos -->
```

We could expand on our previous solution and update that within our JavaScript callback, but that will start us down a downward spiral to complex state rendering hell.

Let's just get a glimpse of what that particular hell looks like:

Todos controller

```
class TodosController < ApplicationController
  # PUT /todos/:id/mark_complete
  def mark_complete
    todo = Todo.find(params[:id]).mark_complete!

    respond_to do |format|
      format.json do
        render json: {
          complete_count: todo.complete.count,
          total: Todo.count
        }
      end
    end
  end
end
```

Todos JavaScript

```
$('.todo').on('ajax:success', function(e, data) {
  $(this).addClass('complete');
  $('.complete-count').html(
    data.complete_count + '/' + data.total + ' completed'
  );
});
```

Clearly, this is a messy approach. The state is being handled across too many places and while it might be somewhat manageable right now, as the complexity grows, so too will our need to handle the state of these components on the page.

Ok so *now* must be the time to reach for React, right?

Not yet!

Solution

Most people familiar with Turbolinks simply use it as a plug and play solution. They add it to the Gemfile and it works (maybe with a bit of extra configuration). But what they might not know is that its functionality can be triggered on demand.

Lets see how we can use this to drastically cut down the complexity of handling state client-side, and push that responsibility back to the Rails server, where it belongs.

In our success handler, we'll replace everything with the following line:

Todos JavaScript

```
$('.todo').on('ajax:success', function(e) {
  Turbolinks.visit(window.location.href);
});
```

We're calling Turbolinks directly, telling it to 'visit' the current url. This visit function is the same one that gets called automatically when clicking links within a Turbolinks-enabled Rails app. But in this case, we're forcing a page visit on the current page, which will cause Turbolinks to get a new version of the current page asynchronously from the Rails server and refresh just the parts of the page that have changed.

This way, the state all the elements on our page gets recalculated and re-rendered automatically, with no need to reload the page, and no need to handle all the updates manually.

Chapter 8: Mailers

Emails are a key touchpoint for your users. They can drive significant conversions as a marketing tool and are the primary method of communication. Because theyâ€™re so important, they need to look great, not contain errors, and be readable on all devices.

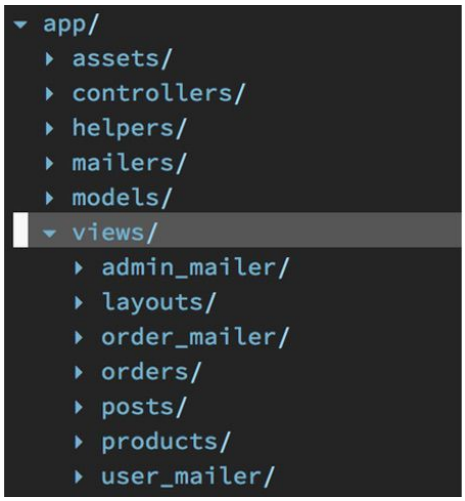
Rails provides us with ActionMailer, a lightweight and universal interface to send emails throughout your application. And because itâ€™s abstracted, it makes it easy to swap between providers.

In this chapter, we'll cover a number of ways to upgrade your Rails email workflow.

Tidying up Mailer Views

Problem

By default rails throws all the types of mailer class views into their own directory at the root of app/views. These mailer views behave very differently from the rest of the application's views. As the amount of mailers increases, the views folder becomes increasingly cluttered, making it difficult to find the views you're after.



By default, mailer views get thrown in amongst regular views

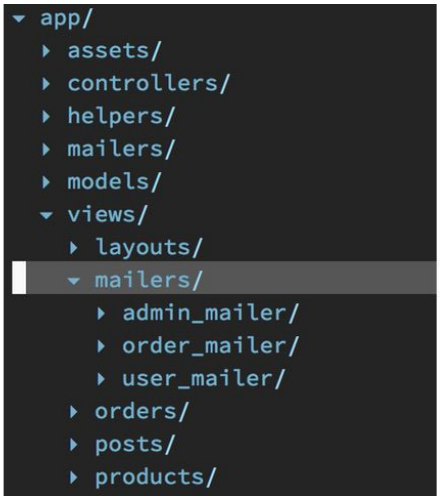
Solution

We can configure Rails to look for mailer view templates in a different place. In this case, we'll nest all mailer views under app/views/mailers so that they don't get jumbled with the rest of our application's views.

To do this, we can use the `append_view_path` method. We'll do this in the ApplicationMailer, which all individual Mailer classes should inherit from.

```
1  class ApplicationMailer < ActionMailer::Base
2    append_view_path Rails.root.join('app', 'views', 'mailers')
3  end
```

Now, all mailer templates will be served out of app/views/mailers. Much more civilized.



Mailer views grouped under a 'mailers' directory

Ensure Test Emails Never Get Sent to Real Users

Problem

It's a problem we've all experienced just once (myself included) and when you realize what's happened, it's like a punch to the gut. You're testing the application in a staging environment with a copy of the production database or writing integration tests

everything's running great. But then a sudden realization hits: was Rails sending emails?

sign into your email provider and yep, that user you were testing with just received 1000 identical emails. Or your entire user base just received a half baked email campaign that you were testing.

I say we've experienced it just once because once you have to be the one to send that apology email, you will be sure to put the proper safeguards in place to prevent the problem from happening again.

Wouldn't it be great if we could just tell Rails to never send any emails to real users if we're not running in production?

Solution: Interceptors

Action Mailer provides a very convenient config hook that allows us to register interceptor classes, through which all emails will pass before being delivered. Within these interceptor classes, we can modify any properties of the email, such as the subject, recipients or the message itself.

To prevent real users from receiving emails in environments besides production, we'll write a simple class called SandboxEmailInterceptor which defines a class-level method: self.delivering_email.

```
1  # lib/sandbox_email_interceptor.rb
2
3  class SandboxEmailInterceptor
4    def self.delivering_email(email)
5      email.subject = "#{email.subject} (to: #{email.to})"
6      email.to = ['sandbox@example.com']
7    end
8  end
```

This method takes the Action Mailer email object as an argument and with that object, we can redefine a few properties. First, we'll alter the subject to include the original recipients, so we know who would have received the message had it not been intercepted. And finally, we'll overwrite the `to` property to be an email we have access to so we can preview the message.

Once we have the interceptor class in place, we'll register it in an initializer.

```ruby
# config/initializers/email.rb

unless Rails.env.production?
 ActionMailer::Base.register_interceptor(SandboxEmailInterceptor)
end
```

Now, any emails sent in an environment other than production will be intercepted and their recipients and subject altered to prevent delivery to real users.

What about other customer touchpoints?

While this is a solution for email, you should also think about all the other ways your users could potentially be contacted. Using Twilio to send text messages or make calls? What about push notifications to a mobile app? Whatever your touchpoints with the customer are, you should make sure that a similar interceptor or sandbox environment are configured.

Validating Email Addresses

Problem

Validations are all over the place in Active Record. You validate the format, uniqueness and presence of most user-entered fields in your Rails app. It's just good practice. As such, you might decide to validate that the email addresses your users enter are indeed valid email addresses. Here's what that internal monologue might look like:

> Maybe Rails has a built-in validation for it.
>
> Hm nope.
>
> I guess I can just validate on the format using a regular expression.
>
> Shouldn't be too hard, right?
>
> I know what email addresses are supposed to look like.
>
> *Googles email address regex.*
>
> Why are the regular expressions so giant? Oh well, I'll just pick this one.

```
1  ^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}$
```

This might work for a little while, until you start getting complaints like these:

"Hey, I tried registering for your site with brand new email address bob@bob.rocks but it's saying that it's not a valid email address. Your app is bad and you should feel bad."

So you poke around and find a more complex regex until the next issue comes up. Then you find a clearly invalid email address that your regular expression accepts.This repeats until your regex looks like this:

What have we gotten ourselves into?

```
1  /^(?=[A-Z0-9][A-Z0-9@._%+-]{5,253}+$)[A-Z0-9._%+-]{1,64}+@(?:(?=[A-Z0-9-]{1,63}+\
2  \.)[A-Z0-9]++(?:-[A-Z0-9]++)*+\.){1,8}+[A-Z]{2,63}+$/
```

The RFC specification for what makes a valid email address is much more complicated than you might think. It's so complicated, in fact, that no regular expression to date is able to fully capture the specification 100%. So essentially, there's no way to validate an email address to perfect accuracy with a regular expression.

What to do?

Solution #1: Don't Validate Email Addresses

Yep, that's right. In this case, the best solution is to do nothing at all (almost).

Well that's great, but how do I make sure an email works? user won't be able to sign back in if they made a typo while inputting their email.

The solution is something you're probably doing already. Send a confirmation email. Users are used to confirming email addresses, and it's really the only fool-proof way to make sure a user's email

By sending a confirmation email, we can ensure several important points at once:

- That the format is valid
- That the email address isn't bogus
- That the user actually owns it

In your model, you can validate for the bare minimum, presence and an `@` symbol.

Validate the bare minimum for email addresses

```
class User < ActiveRecord::Base
  validates :email,
    format:   { with: /.*@.*/, message: 'must be valid' },
    presence: true
end
```

Solution #2: Front-end Validation

While the confirmation email is your best bet to guard against users signing up with bogus emails, you might want to help ensure users input a 'valid' email address by giving them feedback while they're filling out the form.

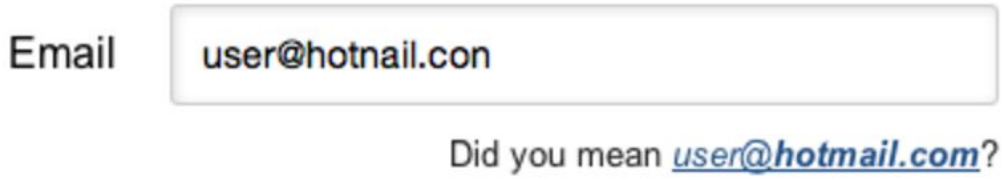
Modern browsers now support HTML5 form validation, which means that if you specify that your input is an email input, it will guard against incorrect values:

```
<input type="email" name="email" required placeholder="Enter a valid email addre\
ss">
```

In most browsers, the regex being used is something really simple, like `.*@.*`. While this won't prevent a user from entering an invalid email address, it will at least verify that they didn't put in something completely incorrect, like their phone number. //<insert screenshot>

You can also find jQuery plugins that check attempt to validate inputs. I particularly like Mailcheck[45], which checks for common mispellings:

[45] https://github.com/mailcheck/mailcheck



The Mailcheck jQuery plugin

Solution #3: Use A Gem

And if you absolutely, positively, must validate email addresses server-side, at least save yourself some regex-fueled headaches by using a gem.

This one is fine: https://github.com/balexand/email_validator[46]

But again, make sure you know what you're getting yourself into when you set out to validate email addresses!

[46] https://github.com/balexand/email_validator

Part III: The Techniques

In this final part, after upgrading our tools and discovering recipes for better Rails code, we'll talk about two broad techniques that are immensely important to any Rails developer worth their keep.

This first technique we'll cover is Testing. Because Ruby isn't a statically typed language and doesn't come with a full-featured, commercial IDE, writing tests is the best way to ensure that our codes will do what we want it to. However, testing can be intimidating to get started with and easy to procrastinate. I'll go through some ways I've upgraded my testing workflow to make it a painless and reqarding process.

The final chapter will deal with debugging, which can easily become a major time-suck for Rails development, if you're not doing it the right way. I'll show you how Pry will completely revolutionize the way you debug Rails apps

Chapter 9: Testing

Testing is an essential aspect of professional Rails development, but one that can be quite intimidating at first. Testing is a bit like exercising: You might not want to do it, but if you bite the bullet and don't make it a question, you'll be so thankful you did. And to extend the exercise metaphor a bit further, testing will keep your codebase fit and healthy by continually flexing the code and not allowing it to sit sedentary and develop problems.

Here are some added benefits of testing:

- It documents the functionality of your classes and outlines the requirements of features
- It makes refactoring easier by making sure you're not breaking any functionality
- It prevents regression; once you fix a bug you can expect it won't pop up again

So testing is great, what else is new? Now how do we make it as enjoyable and straightforward as possible so that we don't dread opening the `spec` folder? This is what we'll cover in this chapter.

Fail Fast

Problem

When running Rspec's default output, you might start seeing tests fail, but you won't get any details on the failure until the suite finishes running. This means you'll need to wait for awhile to start investigating the failure if you have a large test suite.

Rspec includes a configuration option, `fail_fast`, which forces the test suite to stop running at the first failure. This might help a bit, but in most cases, we'd like the suite to continue running to see how many total failures there are and if they have anything in common. We just want to see the error details at the time of failure.

Solution

The instafail gem[47] makes Rspec run the way we want it to: as failures happen, their details get printed to the console while the tests continue to run. This way, we can see what's going wrong without stopping the test suite or waiting for it to finish.

Install

To install, add instafail to your Gemfile:

```
group :test do
  gem 'rspec-instafail'
end
```

Then, add the following to your `.rspec` file to run specs with instafail by default.

.rspec

```
--require rspec/instafail
--format RSpec::Instafail
```

[47] https://github.com/grosser/rspec-instafail

Use a Better Formatter

Problem

Rspec's default formatter is pretty useless. It shows passing tests as green dots, pending tests as yellow asterisks and failures as red Fs.

The default Rspec formatter

This lets you know that tests are running and whether or not they're passing, but it's not really a very informative way to visualize the execution of a test suite.

Solution

Rspec makes it easy to swap formatters. We'll look at Fuubar[48] here, but you should feel free to experiment and find a formatter that works best for you.

Fuubar shows the progress as the test suite executes, including the number of executed tests, the total number of tests, and a rough ETA of when it will complete.

The Fuubar Rspec formatter

Much more informative than the default formatter! It also includes instafail by default (see previous section), so you can see the details of failing tests as they fail, rather than waiting for the whole suite to to finish.

Install

```
group :test do
  gem 'fuubar'
end
```

After adding Fuubar to your Gemfile, simply add it as a flag in your `.rspec` file. Now when you run Rspec, it will use this new formatter.

[48] https://github.com/thekompanee/fuubar

```
# .rspec
--format Fuubar
--color
```

Using a Different Formatter when Running Single Files

When running specs, you generally either run the whole suite, or a single file or test within that file. While a progress formatter might be appropriate when running the full suite, it would be nice to be able to use a more verbose formatter when you're focused on a single file.

We can use the following configuration option to set a different formatter depending on whether we're running a single file, or multiple files.

```
RSpec.configure do |config|
  if config.files_to_run.one?
    config.default_formatter = 'doc'
  else
    config.default_formatter = 'progress'
  end
end
```

Now, we'll see a more descriptive output by using the 'documentation' formatter when running single tests:

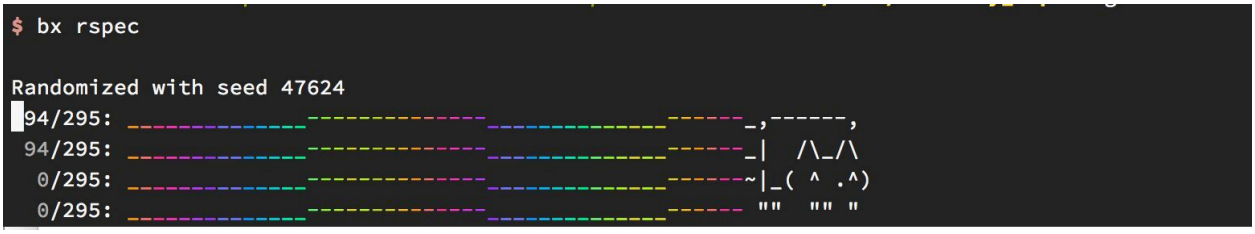
```
User
  #admin?
    when not admin
      returns false
    when admin
      returns true

Finished in 0.00177 seconds (files took 1.35 seconds to load)
2 examples, 0 failures

Randomized with seed 62010
```

Rspec's documentation-style formatter

Alternatives

As mentioned above, Rspec makes it easy to swap formatters. For example, here's a Nyan Cat formatter[49] in case that works for you:

[49] https://github.com/mattsears/nyan-cat-formatter



The Nyan Cat Rspec formatter

You can also write your own formatter if nothing off-the-shelf fits the bill. Check out the RSpec docs[50] for an explanation of the formatter API.

[50] https://relishapp.com/rspec/rspec-core/v/3-4/docs/formatters/custom-formatters

Guard

Problem

As we're developing, it would be extremely helpful to get fast feedback on how the changes we make affect the state of our test suite. This is especially true if you do any kind of TDD, where you want to run a corresponding spec file for every line of code you change. It can be tedious to manage the overhead of this constant test running manually. How about an automated solution?

Solution

The idea behind Guard is that you give it files to watch and an action to perform when those files change. In our case, we want it to watch files within our Rails app like models, controllers and views, and run the corresponding spec when it's changed. For example, if we edit our Post model (app/models/post.rb), we'd want to run the spec file at spec/models/post_spec.rb.

To accomplish this, we'll install the guard-rspec plugin.

Install

To install, we need to add the guard and guard-rspec plugin to our Gemfile:

```
group :development do
  gem 'guard'
  gem 'guard-rspec', require: false
end
```

After bundling, we can run the following to create an empty Guardfile, the file where we'll define our Guard functionality:

```
$ bundle exec guard init
```

We can then run the init task from guard-rspec to add the default RSpec functionality to the Guradfile:

```
$ guard init rspec
```

To start Guard running, we'll simply execute it through bundler:

```
$ bundle exec guard
```

This default RSpec block will work for most cases, but chances are, you'll want to configure it further. Since this is an extensive topic, check out the following resources for more Guardfile inspiration:

- https://github.com/guard/guard-rspec[51]
- https://github.com/guard/guard/wiki/Guardfile-examples[52]
- https://github.com/guard/guard[53]

Receiving Notifications

The final component of Guard is configuring a way to receive notifications. Since it's running in the background, we'll want to see something pop up when a test fails. Some options include using the built-in Mac Notification Center, Growl, or even getting alerts right in TMux if that's the way you roll.

Since everyone's preference will differ, I'd recommend referring to Guard's documentation[54] to get started configuring notifications.

[51] https://github.com/guard/guard-rspec
[52] https://github.com/guard/guard/wiki/Guardfile-examples
[53] https://github.com/guard/guard
[54] https://github.com/guard/guard/wiki/System-notifications

Only Failures

Problem

You run your test suite and find a few failures across the codebase. You want to make sure you address each of them before your code is ready to merge. In order to do this, you might copy the first line under the 'Failed Examples' heading and run it a few times until you get it passing.

At this point, to run the next failing spec, you'd need to scroll back in your terminal to find the failures, remember which one you're one and paste that into the console. If you're fixing more than one or two specs, this process can really start to slow you down.

Solution

With a little configuration, RSpec allows us to rerun only failing specs, and even provides a workflow to keep running the next failing spec until it's passing.

In order to enable this feature, we need to allow RSpec to persist state between test runs, so it knows which tests have been passing, and which have been failing. To do this, we'll use the following config option:

```
1  RSpec.configure do |config|
2    config.example_status_persistence_file_path = "spec/examples.txt"
3  end
```

We'll want to add this file to `.gitignore` to keep it out of Git

Now we can run only the failing specs from the last run by using the `--only-failures` flag:

```
1  $ bundle exec rspec --only-failures
```

And if we want to focus on one failing spec at a time until we get it passing, we can use the `--next-failure` flag:

```
1  $ bundle exec rspec --next-failure
```

Tagging Specs

Problem

An RSpec test suite is a complex mixture of unit tests, JavaScript-enabled feature specs, slow specs, fast specs, model specs, controller specs... you get the idea. When we run `bundle exec rspec`, we run the entire test suite. Sure we could narrow this down by file or folder, but sometimes it would be helpful to have a way to categorize our specs.

Solution

RSpec gives us a couple options when it comes to categorizing tests by using tags.

Simple Tags

You might be familiar with tagging tests that should run with Capybara by adding `js: true`

```
1  it 'needs JavaScript to run this test', js: true do
2    # A JavaScript test
3  end
```

This tells RSpec to boot up Capybara to run this test, but we can also hook into the tagging system when running our tests. For example, assuming we've tagged our feature specs as `js: true`, we could run just those specs by specifying the js tag:

```
1  $ bundle exec rspec --tag js
```

Similarly, if we just wanted to run unit tests that typically run faster than integration tests, we could exclude the js tag by prepending it with ~:

```
1  $ bundle exec rspec --tag ~js
```

It doesn't have to be `js: true`, you can add any kind of tag you want.

Value Tags

In addition to simply flagging tests with a tag and `true`, we can also label tests with a tag and value. For example, maybe we give each test a speed: fast, normal, slow.

```
it 'runs slowly', speed: 'slow' do
  # A slow test
end
```

Or more practically, we could label the type of test at the describe block:

```
describe Post, type: :model do
  # Post tests
end
```

Now if we wanted to run just model specs, we could specify tests whose type is model:

```
$ bundle exec rspec --tag type:model
```

And as we saw above, we can use ~ to exclude a tag:

```
$ bundle exec rspec --tag ~speed:slow
```

Retrying Tests

Problem

Integration tests, while essential to testing the flow through your application, are notoriously fickle. If your app is doing anything remotely tricky with JavaScript, you'll see a test fail every once in awhile for no reason in particular.

Sometimes it has to do with timing. Maybe part of the page is loaded asynchronously with ajax so the timing is never exact. Capybara does a pretty good job at figuring out when a page is fully loaded - ajax and all - but occasionally, it'll fail.

When this happens, your whole test suite is considered to have failed. This is especially problematic in Continuous Integration and Deployment pipelines. A single intermittently failing test can wreak havoc on your team's workflow.

Solution

While there are certainly things we can do to harden our integration tests (resize window, manual wait for ajax), sometimes the best solution is to just try it again. We can use the RSpec-Retry gem to configure our test suite to retry failing integration tests some number of times before considering it failed.

Install

To install, add it to the test section of your Gemfile:

```
group :test do
  gem 'rspec-retry'
end
```

And then configure it in your `spec_helper` file:

spec/spec_helper.rb

```
require 'rspec/retry'

RSpec.configure do |config|
  # show retry status in spec process
  config.verbose_retry = true
  # show exception that triggers a retry if verbose_retry is set to true
  config.display_try_failure_messages = true

```

```
  # run retry only on features
  config.around :each, :js do |ex|
    ex.run_with_retry retry: 3
  end
end
```

Disabling During Test Development

While retrying can be invaluable in CI environments and when running the full suite locally, when you're actively developing an integration test and expecting it to fail, waiting for the full set of retries to complete is unnecessary. RSpec-Retry allows you to set an environment variable that overrides the number of times a test will retry.

```
$ RSPEC_RETRY_RETRY_COUNT=0 bundle exec rspec spec/features/an_integration_spec.\
rb
```

You could even add this as an alias to simplify its invocation:

```
alias devspec="RSPEC_RETRY_RETRY_COUNT=0 bundle exec rspec"
```

Measuring Test Coverage

Problem

We know we should be backing up as much of our codebase as possible with tests. But it's difficult to get a sense of how well-covered our codebase is just by looking at it. Sure, we could make sure that each model has a corresponding spec file. That's a good start, but how can you be sure that every method and every branch of logic in that model is covered by at least one spec? And how can you identify your hotspots that are really lacking code coverage?

Solution

We could go with a drop-in SaaS solution like CodeClimate[55], which has all the bells and whistles you'd expect from a mature product. But it's expensive at $20 per user per month. It used to be even more expensive at $50/mo for the lowest tier although recently, they've lowered their pricing and are offering a free CLI version.

However, the CLI requires a good amount of setup. It needs to run within Docker, which can be a cumbersome responsibility to impart on all developers working on the project.

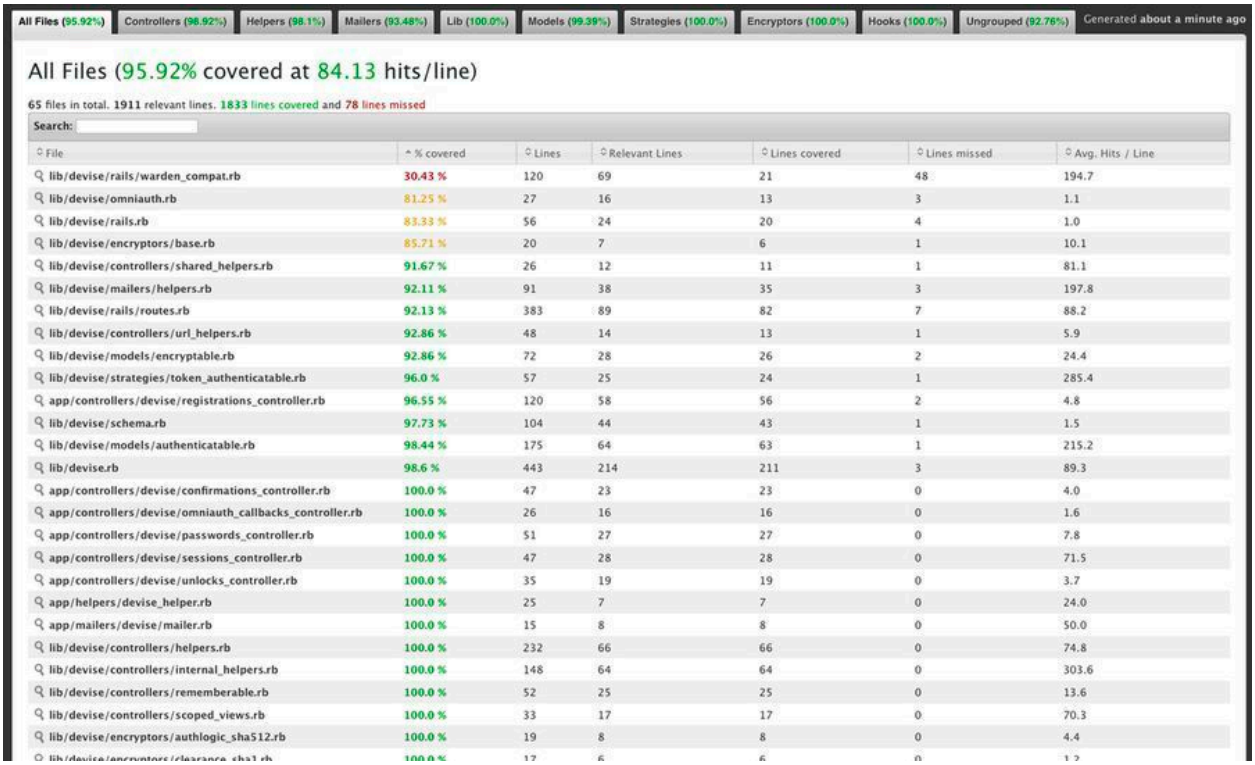
SimpleCov

If you're looking for an open-source solution that can plug into an existing project, SimpleCov[56] is a great gem-based solution for measuring test coverage.

After running our test suite, we get an HTML output that shows us the coverage percent of every class in our project, organized by categories like Models, Controllers etc.

[55] https://codeclimate.com/

[56] https://github.com/colszowka/simplecov



All Files (95.92%) | Controllers (98.92%) | Helpers (98.1%) | Mailers (93.48%) | Lib (100.0%) | Models (99.39%) | Strategies (100.0%) | Encryptors (100.0%) | Hooks (100.0%) | Ungrouped (92.76%) | Generated about a minute ago

All Files (95.92% covered at 84.13 hits/line)

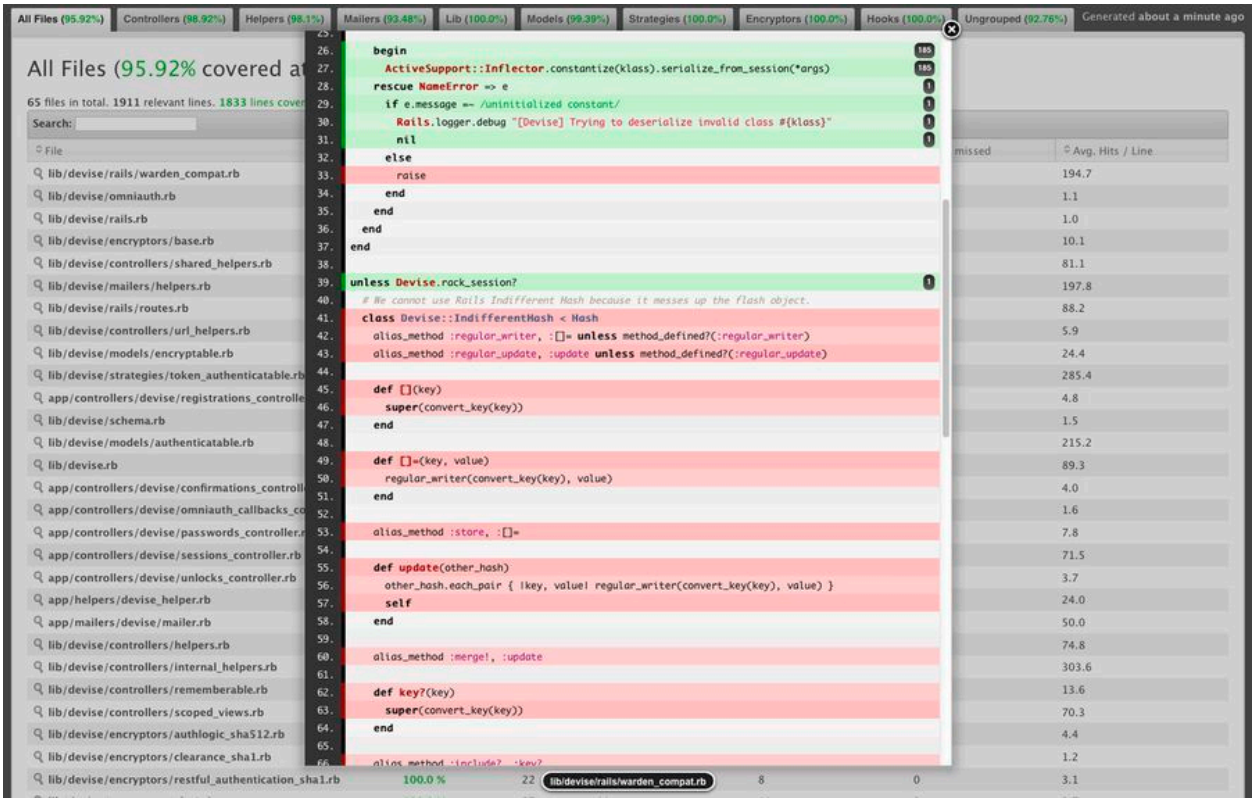
65 files in total. 1911 relevant lines. 1833 lines covered and 78 lines missed

Search:

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
lib/devise/rails/warden_compat.rb	30.43 %	120	69	21	48	194.7
lib/devise/omniauth.rb	81.25 %	27	16	13	3	1.1
lib/devise/rails.rb	83.33 %	56	24	20	4	1.0
lib/devise/encryptors/base.rb	85.71 %	20	7	6	1	10.1
lib/devise/controllers/shared_helpers.rb	91.67 %	26	12	11	1	81.1
lib/devise/mailers/helpers.rb	92.11 %	91	38	35	3	197.8
lib/devise/rails/routes.rb	92.13 %	383	89	82	7	88.2
lib/devise/controllers/url_helpers.rb	92.86 %	48	14	13	1	5.9
lib/devise/models/encryptable.rb	92.86 %	72	28	26	2	24.4
lib/devise/strategies/token_authenticatable.rb	96.0 %	57	25	24	1	285.4
app/controllers/devise/registrations_controller.rb	96.55 %	120	58	56	2	4.8
lib/devise/schema.rb	97.73 %	104	44	43	1	1.5
lib/devise/models/authenticatable.rb	98.44 %	175	64	63	1	215.2
lib/devise.rb	98.6 %	443	214	211	3	89.3
app/controllers/devise/confirmations_controller.rb	100.0 %	47	23	23	0	4.0
app/controllers/devise/omniauth_callbacks_controller.rb	100.0 %	26	16	16	0	1.6
app/controllers/devise/passwords_controller.rb	100.0 %	51	27	27	0	7.8
app/controllers/devise/sessions_controller.rb	100.0 %	47	28	28	0	71.5
app/controllers/devise/unlocks_controller.rb	100.0 %	35	19	19	0	3.7
app/helpers/devise_helper.rb	100.0 %	25	7	7	0	24.0
app/mailers/devise/mailer.rb	100.0 %	15	8	8	0	50.0
lib/devise/controllers/helpers.rb	100.0 %	232	66	66	0	74.8
lib/devise/controllers/internal_helpers.rb	100.0 %	148	64	64	0	303.6
lib/devise/controllers/rememberable.rb	100.0 %	52	25	25	0	13.6
lib/devise/controllers/scoped_views.rb	100.0 %	33	17	17	0	70.3
lib/devise/encryptors/authlogic_sha512.rb	100.0 %	19	8	8	0	4.4
lib/devise/encryptors/clearance_sha1.rb	100.0 %	17	6	6	0	1.2

SimpleCov shows which files are lacking test coverage

But it doesn't stop there! We can drill into a class and see exactly which methods and branches of logic within that class are lacking test coverage.



SimpleCov also shows which methods and logic branches lack coverage

Install

To install, add SimpleCov to the test group of your Gemfile:

```
group :test do
  gem 'simplecov', :require => false
end
```

To enable SimpleCov, it needs to be required and started at the very beginning of your test configuration file (spec_helper in the case of Rspec):

```
require 'simplecov'
SimpleCov.start 'rails'
```

You'll also probably want to add the coverage directory to your `.gitignore` file to prevent it from cluttering up your repository:

```
#.gitignore
coverage
```

Now, as your test suite executes, SimpleCov will be tracking the coverage.

SimpleCov in a Continuous Deployment Pipeline

SimpleCov can also be useful as a step within a continuous deployment pipeline. For example, you trust your test suite and want to allow your continuous integration environment to go ahead and deploy to production, but only if a certain percentage of test coverage is maintained and the coverage percent does not drop too dramatically in any one commit. SimpleCov allows us to do this.

The following configuration options will cause SimpleCov to return an error state if your total coverage is below 90%, or the coverage drops by more than 5%. You can of course tweak these percentages to meet your coverage goals.

```
1 SimpleCov.minimum_coverage 90
2 SimpleCov.maximum_coverage_drop 5
```

Additionally, if you want to ensure your test coverage never moves backwards, you can prevent any drop in coverage percent:

```
1 SimpleCov.refuse_coverage_drop
```

Writing Specs the 'Right' Way

Problem

Rspec attempts to provide a natural and readable syntax, and as a result, we end up many ways of saying the same thing. But while the flexibility is there, the Rspec community has developed conventions and best practices for how to write specs. But if you're just getting started with Rspec, or have been using it for awhile even, it's not always clear what the best way to write something is.

For example, is it best practice to write specs with `should` syntax?

```
1  it 'creates a resource' do
2    response.should respond_with_content_type(:json)
3  end
```

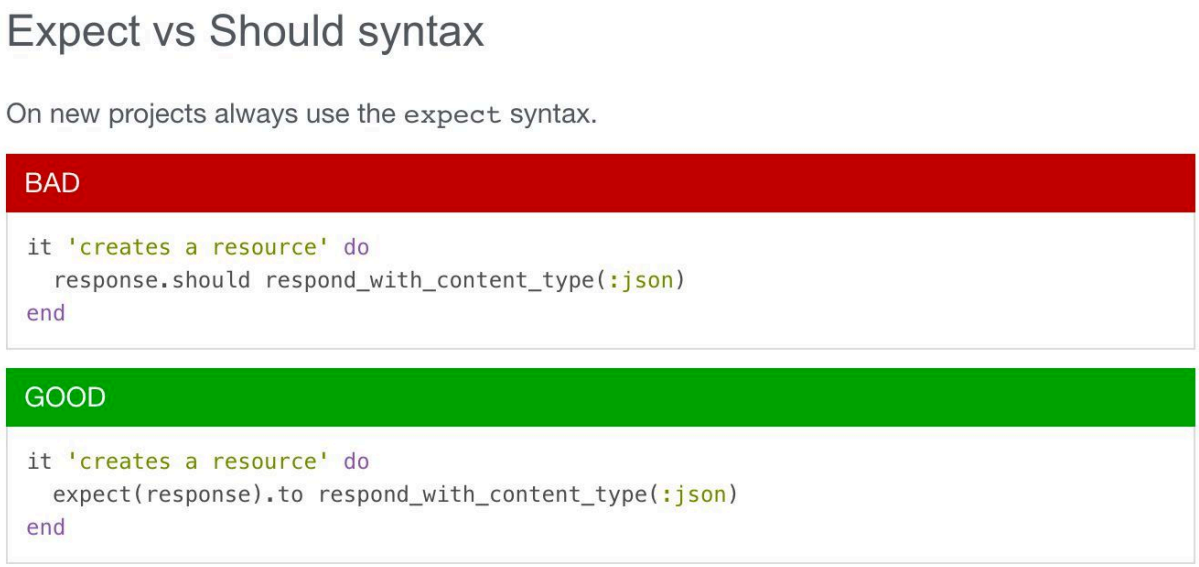
Or is it better to write specs with the `expect` syntax?

```
1  it 'creates a resource' do
2    expect(response).to respond_with_content_type(:json)
3  end
```

Solution

Better Specs[57] is a community-driven style guide that demonstrates the current best practices when writing Rspec tests. So for our example above, let's see what Betterspecs has to say on the matter:

[57] http://betterspecs.org/



Betterspecs on should vs. expect

It's clear the community prefers the `expect` syntax, a topic we'll cover in the next section.

Here is another style guide[58]. This guide goes more in depth into Rails-specific conventions, like how to test Models and Mailers.

[58] https://github.com/howaboutwe/rspec-style-guide

Should (Not)

Problem

If you've used Rspec for awhile, you might have originally learned to phrase specs as "it *should* have some outcome". And then, seemingly overnight, *should* became an antipattern. But where did all of this sudden hate for *should* come from?

It really comes down to an RFC put out by the IETF back in 1997[59]. In this document, the organization defines several key terms and recommends how they should be used when developing and writing for the internet.

Here's what it has to say about *should*:

> SHOULD This word, or the adjective "RECOMMENDED", mean that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.

Now contrast that to *must*:

> MUST This word, or the terms "REQUIRED" or "SHALL", mean that the definition is an absolute requirement of the specification.

When describing the functionality of software through tests, it's clear that what we really mean is 'must'. We're not describing how the application *might* work, if it feels like it, we're describing how it *must* work, otherwise it's broken. So for that reason, it's best to ditch 'should' in favor of the present tense imperative.

Solution

There are two components needed to comply with this convention:

should_not

First, our tests should not begin with 'should'. This can be enforced by using the should_not gem. Simply add the gem to your Gemfile and then require it in your `spec_helper`:

```
1 require 'should_not/rspec'
```

[59] http://www.ietf.org/rfc/rfc2119.txt

Expect syntax

Second, we should use the 'expect' syntax when writing assertions.

In previous versions of Rspec, assertions were written as `(1 + 1).should == 2`. Newer versions of Rspec have deprecated this syntax in favor of a more assertive and less error-prone 'expect' syntax: `expect(1 + 1).to eq(2)`

Check out the documentation[60] for a full list of matchers.

[60] https://www.relishapp.com/rspec/rspec-expectations/docs/built-in-matchers

Random Test Order

Problem

Having the tests in our suite locked to a specific order introduces coupling and brittleness. Information created for one test might unintentionally pollute other tests. This means we could accidentally break the test suite by not realizing one test depends on another, either explicitly or implicitly (it just happens to pass because of something that happened before). It also prevents us from doing cool things like parallelizing test suites to make them run faster.

The current RSpec default is a non-random execution order[61] This means it's up to us to configure our test suite to run randomly.

Solution

Part of the solution to test pollution is to make sure the database is cleared between each test (see the Database Cleaner section). However, existing records in the database are only part of the picture. Strange things can happen between tests and by running them in the same order every time. If they're never run in a different order, we never expose this interconnectedness.

The best way to combat test pollution is to randomize the order our tests run in. This forces we to write isolated tests.

To do this, we can set the order to random in `spec_helper`:

```
1  RSpec.configure do |config|
2    config.order = :random
3  end
```

Debugging Interdependent Tests

After randomizing the test order, we might see a few tests start failing intermittently. The good news is we've exposed test pollution. The bad news? These failures are difficult to debug because when you go to run them individually, they pass!

You'll notice at the end of the test run, RSpec prints the seed it used to order the tests. We can rerun our tests in the same order by specifying this seed:

```
1  $ bundle exec rspec --seed 12345
```

Now, you can set a `binding.pry` in the failing test and explore the context at the point of failure to find the source of the test pollution.

[61] see https://github.com/rspec/rspec-core/issues/635 for the discussion around this topic

Testing with a Clean Slate

Problem

Specs should be run in isolation. This means it doesn't matter the order they run in (see the section on Randomizing test execution order), and there shouldn't be any interdependencies between tests. Each spec should start with a clean slate, and clean up after itself so that the next text can enjoy the same conditions.

However, specs are not run in isolation by default. Unless we specify otherwise, the database will continue to accrue test records that can impact the state of other tests.

Solution

The best solution to this problem is to ensure that each test starts with a blank database. The Database Cleaner gem provides the hooks and strategies to accomplish this.

To demonstrate, here is an example config block:

Example DatabaseCleaner Rspec Config

```
RSpec.configure do |config|

  config.before(:suite) do
    DatabaseCleaner.strategy = :transaction
    DatabaseCleaner.clean_with(:truncation)
  end

  config.around(:each) do |example|
    DatabaseCleaner.cleaning do
      example.run
    end
  end

end
```

Notice the different strategies that Database Cleaner provides. This example makes use of two: 'transaction' and 'truncation'. The transaction strategy wraps each test in a transaction, rolling back the changes when it's finished. The 'truncation' strategy uses a SQL `TRUNCATE` command to empty the database.

Transaction is the fastest strategy and should be preferred whenever possible, however, it's not bulletproof. For instance, integration tests run with Capybara will need to use the truncation strategy since Capybara runs your tests in a different thread.

Here's a configuration that sets Database Cleaner to use transaction by default, but truncation for tests run in the browser:

```
RSpec.configure do |config|

  config.before(:suite) do
    DatabaseCleaner.clean_with(:truncation)
  end

  config.before(:each) do
    DatabaseCleaner.strategy = :transaction
  end

  config.before(:each, :js => true) do
    DatabaseCleaner.strategy = :truncation
  end

  config.before(:each) do
    DatabaseCleaner.start
  end

  config.after(:each) do
    DatabaseCleaner.clean
  end

end
```

Source[62]

Install

To install the Database Cleaner gem, simply add it to the test section of your Gemfile:

```
group :test do
 gem 'database_cleaner'
end
```

[62] http://devblog.avdi.org/2012/08/31/configuring-database_cleaner-with-rails-rspec-capybara-and-selenium/

Mark tests as pending

Problem

Occasionally, you need to ignore a test, or group of tests, you've already written. Maybe they're failing intermittently and need fixing, or just need to be put on hold for a little while. Sure you could comment them out, but there are a few problems with this:

1. It's annoying to toggle comments on all the lines, even when you're using an editor that supports toggling comments with a shortcut.
2. When you comment out a test, it might as well no longer exist. If you don't set a reminder to come back and deal with it, you might forget to do so.
3. Commenting and uncommenting causes a lot of unnecessary changes in your git history.

Solution

Rspec lets you mark a test as pending by replacing `it` with `xit`. This will skip executing the test, but will show it as pending after your test suite executes. This reminds you that you have a test that's being skipped and needs to be fixed up at some point.

```
RSpec.describe Post, type: :model do
  describe '#bad_method' do
    xit 'fails intermittently' do
      # this test is skipped
    end
  end
end
```

In addition to `xit`, you can prepend any of Rspec's other block types with an `x` to skip their contents. For example, `xdescribe` would mark all the tests within a `describe` block as pending.

```
RSpec.describe Post, type: :model do
  xdescribe '#bad_method' do
    # tests in here are skipped
  end
end
```

Providing a reason

While this is great for quickly marking tests as skipped, if you plan to skip these tests and come back to them later, it helps to have a reason reminding you why these tests are being skipped.

We can accomplish this by using the following skip syntax:

```
RSpec.describe Post, type: :model do
  describe '#bad_method', skip: 'Failing intermittently' do
    # tests in here are skipped
  end
end
```

Now, when Rspec prints our pending tests, we're reminded why:

```
**

Pending: (Failures listed here are expected and do not affect your suite's status)

  1) Post#bad_method performs an action
     # Failing intermittently
     # ./spec/models/post_spec.rb:8

  2) Post#bad_method returns a thing
     # Failing intermittently
     # ./spec/models/post_spec.rb:5


Finished in 0.0009 seconds (files took 4.63 seconds to load)
2 examples, 0 failures, 2 pending
```

Rspec shows the reason why tests are marked pending

Testing Time-Dependent Logic

Problem

Most applications have some degree of time-dependent logic, some more than others. When it comes to testing this time-dependent logic, we often end up skirting around the issue, resulting in fragile, confusing, roundabout or verbose code.

For example, if we want to test that a timestamp gets updated to the current time after a method is called, we might do the following:

```
describe Membership, 'cancel!' do
  it 'sets canceled_at to the current time' do
    membership = create(:membership)
    membership.cancel!
    expect(membership.canceled_at).to be_within(2.seconds).of(Time.current)
  end
end
```

While testing that it gets set to within 2 seconds of the current time might seem sufficient, we're making an assumption that weakens our test suite. That assumption is that `Membership#cancel!` takes less than 2 seconds to run. If the cancel command becomes complex and starts taking a longer time to run, you might end up with a test that fails intermittently.

Ideally, we would address the issue head-on by controlling the flow of time within our application as part of the test setup.

Solution

Timecop is a must-have gem that makes testing time-dependent logic much more straightforward. It gives us the ability to freeze time at a certain moment, and travel to a relative time.

Freezing Time

Let's see how we'd rearrange the example above using Timecop:

Timecop Freeze

```
describe Membership, 'cancel!' do
  before { Timecop.freeze(Time.current) }

  it 'sets canceled_at to the current time' do
    membership = create(:membership)
    membership.cancel!
    expect(membership.canceled_at).to eq(Time.current)
  end
end
```

Now, we can test the equality of `canceled_at` and `Time.current` exactly, rather than introducing an assumption.

Time Traveling

We can also use Timecop's time traveling functionality to test logic that depends on the current date or time.

For example, our membership might last for 30 days, at which point it's considered expired and needs to be renewed. We can test this flow by traveling forward 30 days from now and verifying that `membership.expired?` returns true.

Timecop Travel

```
describe Membership, '#expired?' do
  it 'expires after 30 days' do
    membership = create(:membership)
    expect(membership.expired?).to eq(false)

    # Travel 30 days into the future
    Timecop.travel(30.days)
    expect(membership.expired?).to eq(true)
  end
end
```

Install

To install, we'll add it to the development and test groups of our Gemfile.

```
group :development, :test do
  gem 'timecop'
end
```

We'll explore how to use Timecop in your development environment in the next section.

Working with a Time-Dependent Codebase

Problem

If you have a lot of time-dependent code, it can be helpful to run your local environment at a specific time. Maybe you run a daily deals site, but the data you currently have in your local database is from last week. If you run the local environment, you won't have any active daily deals to work with. You'd have to go through the process of manually creating a deal in order to start working locally.

Instead of setting up fresh data, or syncing with production, it would be much faster to simply pretend it's last week, and use last week's deals to test with. You could do this by setting the system clock on your computer, but that affects a lot more than our code.

Solution

Using Timecop, which we learned about in the previous section, we can wrap our local environment's initialization with a bit of time-traveling magic.

If our last working data is on January 11th, 2016, we can fool our server into thinking that's the current time by adding the following to our `development.rb` configuration:

```
config.after_initialize do
  # Set Time.now to January 11, 2016 10:05:00 AM (at this instant),
  # but allow it to move forward
  t = Time.local(2016, 1, 11, 10, 5, 0)
  Timecop.travel(t)
end
```

Now, as far as it knows, our local server is running at 10:05 am on January 11th and everything will behave accordingly.

Speed Up the Test Suite

Problem

Tests are important – we get it. So we've built out a full suite of unit and integration tests, monitored our test coverage to make sure we've tested every nook and cranny of our application, and hooked it into a continuous integration environment.

We can sleep soundly at night. There's just one problem – our full test suite takes a long time to run now.

While it's easy to stress about how long our test suite takes to run, it's important to remember:

> We can make good tests run fast but we can't make fast tests be good.

Which is to say, we shouldn't sacrifice the quality of our tests under the guise of speeding them up. But test speed is important just the same.

Having a fast feedback loop is critical when developing a feature or fixing a bug. If you get feedback quickly, you'll retain the context of what you were working on and be able to ship it sooner.

If your suite takes 10 minutes to run, you'll go grab a coffee, check Hacker News, and find yourself down a Wikipedia wormhole an hour later when you remember that you were working on a feature.

So with that in mind, let's dig into a few strategies for speeding up a slow test suite.

Solution: Find Slow Tests

By setting RSpec's `profile_examples` option, it will evaluate the runtime of each test in the suite and return the slowest running tests.

For example, we might set it to 3 to return the 3 slowest tests after each run:

```
1  RSpec.configure do |config|
2    config.profile_examples 3
3  end
```

This will surface slow-running tests and keep test speed top-of-mind.

Solution: Fix Slow Tests

Once we've found a slow test, how do we improve it?

Audit the Setup Data

For starters, it's important to only create the data necessary for the test. Ask yourself if you actually need a list of 10 objects, or if 1 or 2 could suffice? Creating objects in the database (and rolling them back after each test) is generally the biggest time-suck in terms of test performance. It's also worth reviewing your factories and ensuring that they're not creating more data than they need. This can be especially pervasive since a factory might be used in hundreds of tests without realizing the extent of what other records it creates.

Use VCR to Mock External APIs

If you use external APIs in your app like payment processors, social sites for oauth, or even your own services, the overhead of making an HTTP request and waiting for the response can really weigh down your tests. It can also cause failures if the service is down and inflate your API usage (which can be a big deal if you're charged per request, or only have some amount of requests available per hour). To remedy this, use VCR/WebMock to record those interactions once and replay their responses for future tests. You also want to ensure you're using a sandbox API key for the test environment.

Stubbing

Some developers will stub nearly everything in a test with the intention of keeping their test fast. Here are my thoughts on stubbing: stub what makes sense, but don't sacrifice the quality of your tests for the sake of a small speed boost. And never stub the system under test! That means if you're testing the User model, no methods on that model should be stubbed.

Solution: Preloaders

Preloaders reduce startup time by loading up the Rails environment in the background and keep it running, rather than starting it from scratch each time. Depending on the size and complexity of your codebase, this could be anywhere from 2 to 30 seconds!

Spring comes by default with rails 4 apps. It's worth understanding how it works and ensuring it is running properly.

Zeus is another good option that some developers prefer to Spring. If you're not happy with Spring for some reason, see if Zeus does it for you.

Solution: Parallelization

Processing power is getting cheaper by the day. Once we've gotten our tests as fast as possible without sacrificing quality, sometimes the best solution is to just throw money at it. Assuming we've written our tests so that they are not interdependent (see the sections on Randomization and Database Cleaner), there's no reason that the tests in the test suite need to be run one after another in a serial fashion. Instead, we can break the suite up into chunks that can be processed on different machines or even different cores on the same machine.

Local Parallelization

We can run tests in parallel on our local machine by using the parallel_tests gem. This gem can take a bit of tweaking to get right, but once you have it set up correctly, it cuts your test time by a factor of how many cores you give it. For modern computers, this can be 6 or even 8-fold. A five minute test suite would run in less than a minute.

To install add it to the development group of your Gemfile:

```
group :development do
  gem "parallel_tests"
end
```

The tricky part about running tests in parallel locally is that each instance needs to have its own database. So we need to modify our database.yml definition to handle this:

```
test:
  database: yourproject_test<%= ENV['TEST_ENV_NUMBER'] %>
```

We then need to create and prepare each of these new databases:

```
rake parallel:create
rake parallel:prepare
```

Now, we can run it:

```
rake parallel:test          # Test::Unit
rake parallel:spec          # RSpec
rake parallel:features      # Cucumber
rake parallel:features-spinach       # Spinach

rake parallel:test[1] --> force 1 CPU --> 86 seconds
rake parallel:test    --> got 2 CPUs? --> 47 seconds
rake parallel:test    --> got 4 CPUs? --> 26 seconds
```

Parallelization in CI

Parallel test execution has become a standard feature of most third-party CI services. The cost of running tests in parallel in these services can add up quickly, but it's well worth to have a fast feedback cycle. And it's way simpler than standing up a Jenkins instance and configuring it to run in parallel across multiple machines!This is a case where I'd much rather pay a nominal monthly fee than have to spend a week debugging Jenkins and losing days of productivity down the line as things break and need attention.

Here are some CI services that offer parallelization:

- Circle CI[63]
- Travis CI[64]
- Codeship[65]

[63] https://circleci.com/
[64] https://travis-ci.org/
[65] https://codeship.com/

Chapter 10: Debugging

Let's be blunt for a second, debugging in Rails is a pain. Unlike other frameworks, it's just not a straightforward process. Platforms like Microsoft's Visual Studio and Apple's Xcode offer extensive debugging suites built into their integrated development environments (IDEs) that allow you scrutinize your program from all angles.

Since Ruby is a dynamic scripting language, Rails doesn't ship with an IDE. We're left to our own devices when it comes to debugging and as a result, we often waste precious time just trying to understand what our code is doing (and why it's not doing what we told it to).

In this chapter, I will cover strategies to make debugging in Rails a less painful process. The bulk of the chapter be devoted to Ruby debugging, but a Rails app often comprises more than just Ruby code. Therefore, I'll touch on API and JavaScript debugging towards the end.

Inspecting Ruby Code at Runtime

Problem

Debugging in Rails can be summed up with a single word: puts. Ruby doesn't come with a robust debugging framework out of the box like other languages. Because of this, it's a really common technique to use puts to print the value of a variable into the log in order to see what your code is doing at runtime.

```
1  puts "a_variable: #{a_variable}"
```

While this might get the job done eventually, it has a few major drawbacks: You have to hope that you picked the right variable to inspect, otherwise you'll have to go back, change the puts statements and then refresh the application to see the new values. This can be especially painful if it takes a good deal of setup to get the application in the right state (for example, filling out a form).

You have to type out the name of each variable you want to inspect. Typing `puts "some_variable: #{some_variable}"` for each variable name gets old fast. There's no way to explore the surrounding code and state of the application. All you learn is the contents of the variables you've chosen to print.

Solution #1: Use Pry

Pry is the current state of the art when it comes to debugging Ruby applications. If puts is a magnifying glass, then Pry is an electron microscope giving you unprecedented visibility into what's going on in your code at runtime.

To install Pry, simply add it to your Gemfile:

```
1  group :development, :test do
2    gem 'pry-byebug'
3  end
```

In this case, I'm using the pry-byebug gem, which combines Pry and Byebug for extra debugging features which we'll cover in the following sections. To use Pry, simply add `binding.pry` to your code to create a breakpoint.

```ruby
def another_method
  binding.pry
  return 1 / 0
end
```

This will freeze the Rails server in time when it hits that line. From here, we can run Ruby code against the current state of our application.

```
Started GET "/posts" for ::1 at 2015-05-10 18:12:56 -0700
  ActiveRecord::SchemaMigration Load (0.2ms)  SELECT "schema_migrations".* FROM "schema_migrations"
Processing by PostsController#index as HTML

From: /Users/andrewallen/dev/test_blog/app/controllers/posts_controller.rb @ line 85 PostsController#another_method:

    83: def another_method
    84:   binding.pry
 => 85:   return 1 / 0
    86: end

[1] pry(#<PostsController>)>
```

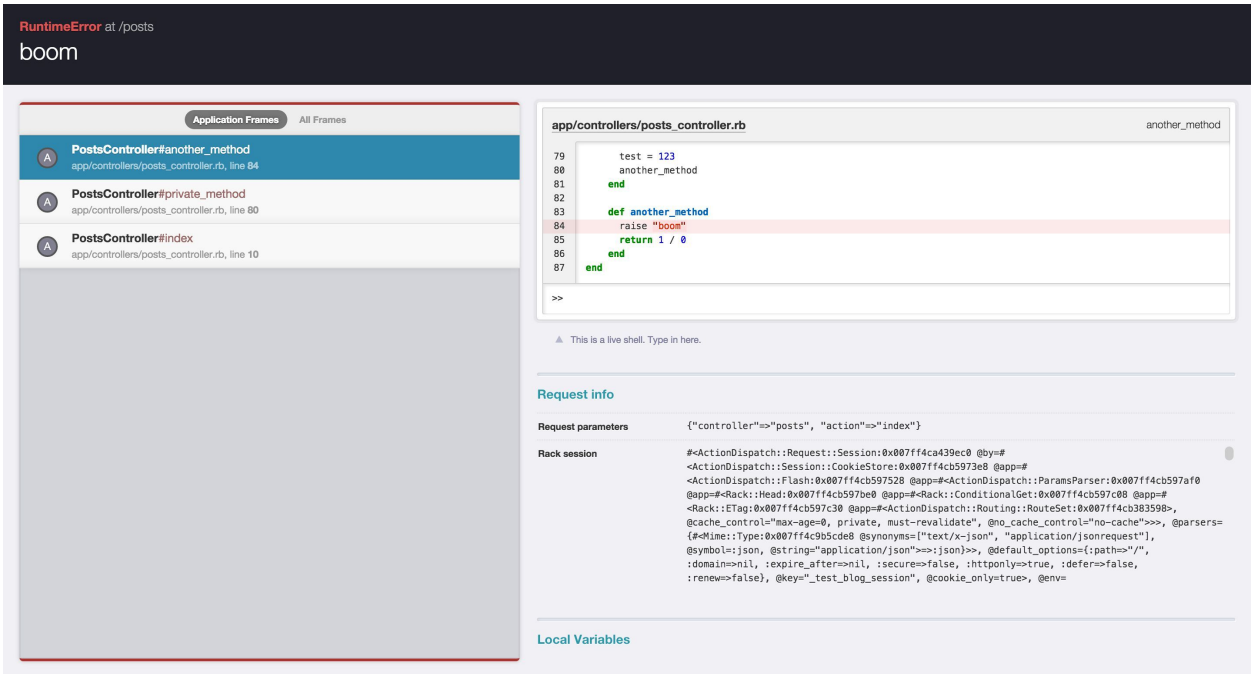
Pry in action

To leave the Pry session and continue executing the code, press kbd:[Ctrl+D] or type `exit`.

Solution #2: Better Errors + raise

If you prefer a more visual debugging environment, the Better Errors gem provides an intuitive, visual representation of your application in place of Rails' default error page (see Making the Best of a Bad Situation for more on Better Errors).

Since Better Errors only springs into action on application errors, a simple but powerful debugging trick is to raise an error in your code at the point you want to debug.



Better Errors gives us an interactive debugging environment

Bam, instant visual debugging environment complete with interactive console, call stack and local variables.

One caveat is that this requires you to be developing a feature that renders HTML output. This won't work if you are building a JSON API or a JavaScript/AJAX hook, for example.

Walking up the Call Stack with Pry

Problem

You throw a `binding.pry` in your code in an area that seems problematic. But you realize once you're in the debugger, that you actually want to know more about how the code got to this state.

Maybe you have a method like this:

```
def foo(bar)
  # ...
  puts bar.length
  # ...
end
```

Which is throwing an error: `NoMethodError: undefined method 'length' for nil:NilClass`.

So you add a debugging statement:

```
def foo(bar)
  # ...
  binding.pry
  puts bar.length
  # ...
end
```

But once in the debugger, you realize method `foo` is being called with `nil` as its argument. So really, you want to be looking at the place in the code where `foo` is being called in order to understand why `nil` is being passed to `foo`. Time to exit the debugger, go back to the code and add another `binding.pry`, right?

Solution: Use Pry's Call-Stack Explorer

With the addition of the stack explorer plugin or Byebug (see below), Pry has the ability to step up and down in the call-stack, allowing you to inspect the environment of every frame leading up to the place where you triggered `binding.pry`.

In this case, we'll use pry-byebug.

```
group :development, :test do
  gem 'pry-byebug'
end
```

The up command will take you up one frame in the stack. Logically, the opposite command is down.

```
def first_method
  a_local = 123
  second_method
end

def second_method
  binding.pry
  return 1 / 0
end
```

Let's take a look at an example:

The Pry session will start in second_method. Because the a_local variable is local to first_method's scope, we will be unable to see it.

first_method is out of scope

But since first_method called second_method, we can travel up the call stack to see what's going on in first_method's context by calling the up command.

```
[1] pry(#<PostsController>)> up

From: /Users/andrewallen/dev/test_blog/app/controllers/posts_controller.rb @ line 80 PostsController#first_method:

    78: def first_method
    79:   a_local = 123
 => 80:   second_method
    81: end

[1] pry(#<PostsController>)> a_local
=> 123
[2] pry(#<PostsController>)>
```

After moving up the call stack with Pry

Now, we can see the value of `a_local` equals 123.

Stepping Forward in Time with Pry

Problem

If you've used a debugger in most other languages, you'll no doubt be expecting to be able to step forward through the code from your breakpoint to see how each subsequent line executes. You might want to put the breakpoint somewhere before a problematic area and step over each line to see how its being computed.

This is a common debugging practice, however, Pry does not support it by default, and this can be frustrating.

If you have a block of code with several lines and you want to see how each line executes, you'll need to add a binding.pry to each line. And then delete them all when you're done. Not very fun.

Solution: Byebug

Byebug is a feature-rich debugger for Ruby that has stepping built into it. Pry-Byebug is a gem that combines Byebug's stepping functionality with Pry's advanced REPL environment.

To install, simply add it to your Gemfile:

```
1  group :development, :test do
2    gem 'pry-byebug'
3  end
```

and run bundle install.

You'll now have access to the following commands in Pry as shown in the Byebug readme:

Byebug stepping commands

Command	Result
step	Step execution into the next line or method
step 3	Step execution into the next *3* lines or methods
next	Step over to the next line within the same frame
next 3	Step over to the next *3* lines within the same frame
finish	Execute until current stack frame returns
continue	Continue program execution and end the Pry session

Stepping Shortcuts for Pry/Byebug

Problem

If you use Pry to step through long segments of code, typing out the step command over and over again can start to slow you down.

Solution: Use the .pryrc Config File

Pry allows for significant extensibility through its ~/.pryrc configuration file.

To speed up stepping, we can create single-character aliases for each of the commands by adding the following to your ~/.pryrc file:

~/.pryrc

```
if defined?(PryByebug)
  Pry.commands.alias_command 'c', 'continue'
  Pry.commands.alias_command 's', 'step'
  Pry.commands.alias_command 'n', 'next'
  Pry.commands.alias_command 'f', 'finish'
end
```

You can also tell Pry to repeat the last command if you press Enter. This way, you can continue stepping just by pressing Enter. To do this, add the following to your ~/.pryrc file:

~/.pryrc

```
# Hit Enter to repeat last command
Pry::Commands.command /^$/, "repeat last command" do
  _pry_.run_command Pry.history.to_a.last
end
```

Exploring Objects, Methods and Variables with Pry

Pry gives us a powerful set of tools for exploring the context of our application's state. What's more, if you're comfortable navigating directories from the terminal, you'll be familiar with the two main Pry navigation commands, ls to list the variables and methods in the current scope, and cd to move into a new context.

Pry context commands

Command	Result
whereami	Show code surrounding the current context
ls	Show the list of variables and methods in the current scope
cd	Move into a new context (object or scope)
nesting	Show nesting information
find-methods	Recursively search for a method within a class/module or the current namespace

whereami

If you ever get lost, whereami will show you the code in the current context.

```
[53] pry(#<PostsController>)> whereami

From: /Users/andrewallen/dev/test_blog/app/controllers/posts_controller.rb @ line 11 PostsController#index:

     6: def index
     7:   @posts = Post.all
     8:   test = "hello"
     9:   world = 123
    10:   binding.pry
 => 11: end

[54] pry(#<PostsController>)>
```

Whereami shows your current location in Pry

By default, Pry aliases `@` to `whereami` which is much quicker to type. However, `@` can conflict with instance variables. To avoid those conflicts, I recommend realiasing to something else. See Stepping Shortcuts for Pry/Byebug for an example of how to do that.

ls

Just like ls in unix lists the files in the current directory, ls in Pry lists the variables and methods in the current scope.

```
ActiveSupport::Configurable#methods: config
AbstractController::Base#methods: action_methods  action_name  action_name=  available_action?  controller_path  response_body
ActionController::Metal#methods:
  controller_name  env  env=  headers=  middleware_stack  middleware_stack=  middleware_stack?  performed?  request  request=  response  response=  session  to_a
ActionView::ViewPaths#methods:
  _prefixes  append_view_path  details_for_lookup  formats  formats=  locale  locale=  lookup_context  prepend_view_path  template_exists?  view_paths
AbstractController::Rendering#methods: _normalize_render  view_assigns
AbstractController::Translation#methods: l  localize  t  translate
ActionController::ModelNaming#methods: convert_to_model  model_name_from_record_or_class
ActionDispatch::Routing::PolymorphicRoutes#methods: edit_polymorphic_path  edit_polymorphic_url  new_polymorphic_path  new_polymorphic_url  polymorphic_path  polymorphic_url
ActionDispatch::Routing::UrlFor#methods: url_for
ActionController::UrlFor#methods: url_options
ActiveSupport::Benchmarkable#methods: benchmark
ActionController::RackDelegation#methods: content_type  content_type=  dispatch  headers  location  location=  reset_session  response_body=  response_code  status  status=
ActionView::Rendering#methods: process  rendered_format  view_context  view_context_class  view_context_class=  view_renderer
ActionView::Layouts#methods: _layout_conditions  action_has_layout=  action_has_layout?
ActionController::Rendering#methods: render_to_string
ActionController::Renderers#methods: _render_to_body_with_renderer  _render_with_renderer_js  _render_with_renderer_json  _render_with_renderer_xml  render_to_body
ActionController::Head#methods: head
ActionController::ConditionalGet#methods: expires_in  expires_now  fresh_when  stale?
ActiveSupport::Callbacks#methods: run_callbacks
ActionController::Caching::ConfigMethods#methods: cache_store  cache_store=
:
```

Ls lists the variables and methods currently available

The amount of info the ls command shows by default can be overwhelming. There will be a lot of methods and variables from Rails' context that you probably won't care about. To narrow things down, ls accepts options.

For example, the -l option prints the current values of only local variables.

```
     6: def index
     7:   @posts = Post.all
     8:   test = "hello"
     9:   world = 123
    10:   binding.pry
 => 11: end

[35] pry(#<PostsController>)> ls -l
test = "hello"
world = 123
[36] pry(#<PostsController>)>
```

Ls takes arguments that filter its list

Similarly, the -i argument shows only instance variables, the -c argument shows only constants, and the -g argument shows only global variables. See below for a list of all options for ls.

Additionally, you can pass an object to the ls command to inspect the available methods and variables on that object rather than the current scope.

Pry ls options

Command	Result
-m, –methods	Show public methods defined on the Object (default)
-M, –module	Show methods defined in a Module or Class
-p, –ppp	Show public, protected (in yellow) and private (in green) methods
-q, –quiet	Show only methods defined on object.singleton_class and object.class
-v, –verbose	Show methods and constants on all super-classes (ignores Pry.config.ls.ceiling)
-g, –globals	Show global variables, including those builtin to Ruby (in cyan)
-l, –locals	Show locals, including those provided by Pry (in red)
-c, –constants	Show constants, highlighting classes (in blue), and exceptions (in purple)
-i, –ivars	Show instance variables (in blue) and class variables (in bright blue)
-G, –grep	Filter output by regular expression
-h, –help	Show help

cd

The cd command lets us move around in the scope of our application and investigate other objects. Once inside a new context, whether a class, module or instance of an object, we can see what methods and variables are defined on it. The cd command can be used in a number of different ways. We can change context directly into a specific object by calling cd ObjectName. So if we had a Post model for example, we could inspect it further by calling cd Post.

As we cd into objects, the new context is pushed on top of the previous context. To see how our contexts have been nested, we can call the nesting command. Note that context nesting is different from the program's call-stack. To get back to a previous context, we can use cd .. which brings us up one level in the context nesting.

To return the the root context where we entered Pry, we can use cd /. All of this will feel very natural if you're comfortable navigating a unix-based filesystem like OS X (cd .. goes up a directory, cd / goes to the root of the filesystem).

find-method

The find-method command searches the current scope recursively for matching method names. This can be extremely useful for finding out where methods are defined.

For example, by running find-method json, we can see where all methods dealing with JSON in our application are defined.

```
[1] pry(#<PostsController>)> find-method json
AbstractController::Collector
AbstractController::Collector#json
ActionController::Renderers
ActionController::Renderers#_render_with_renderer_json
Class
Class#json_creatable?
Enumerable
Enumerable#to_json_with_active_support_encoder
Enumerable#to_json_without_active_support_encoder
Enumerable#to_json
Enumerable#as_json
Hash
Hash#to_json_with_active_support_encoder
Hash#to_json
Hash#as_json
JSON::Ext::Generator::GeneratorMethods::Hash
JSON::Ext::Generator::GeneratorMethods::Hash#to_json_without_active_support_encoder
JSON::Ext::Generator::GeneratorMethods::Object
JSON::Ext::Generator::GeneratorMethods::Object#to_json_without_active_support_encoder
Object
Object#to_json_with_active_support_encoder
```

Pry's find-method command shows where methods are defined

Debugging a Different Method

Problem

After finding a problematic area of code and setting a Pry breakpoint there, you realize that the problem is actually in a related method. To debug this method, you need to set another breakpoint inside the related method.

```
1  def first_method
2    #...
3    an_input = "foo"
4    a_value  = related_method(an_input)
5    binding.pry
6    return a_value
7  end
8
9  def related_method(input)
10   # something goes wrong here...
11 end
```

This requires us to exit Pry, go back to the editor, move the binding.pry into the related method and rerun the debugging session.

Solution #1: Set another Binding.pry by editing within Pry

As it turns out, we can edit files directly from within Pry using the edit command. Pry will acknowledge those changes immediately without needing to reload. We can exploit this functionality to add another binding.pry to the method we want to debug.

To use the edit command, we have to tell Pry which editor it should use by adding the following to our ~/.pryrc. Pry.config.editor = 'vim'

In this case, we'll use vim but this could be subl, mate, gvim or any other text editor command.

Now, from a Pry session, we can use the edit command to activate the editor on any file, object, method.

Edit files with Pry

Command	Result
`edit app/models/post.rb`	Edit a file
`edit Post`	Edit an object
`edit Post#a_method`	Edit a method
`edit -c`	Edit current file

When editing a method defined on the current object, the object can be omitted. So if we first cd into the Post object, we could edit a_method by simply calling edit a_method (see Exploring Objects, Variables and Methods with Pry for more on the cd command).

So to debug the related_method in our example above, we could simply edit the method by calling edit related_method from the first breakpoint. This would open up a Vim session where we could add a binding.pry inside of related_method and save and quit (ZZ or :wq in Vim).

At this point, we'd be returned to the first breakpoint where we could type related_method("some input") to execute the method, hitting the new breakpoint.

Solution #2: ByeBug Breakpoints

If you're using pry-byebug, it allows you to set breakpoints from within Pry. After setting a breakpoint, you can use the continue command to continue execution until the breakpoint is hit.

This solution isn't as flexible as the solution above, because it requires the method you want to investigate to be downstream the flow of execution. But if you do happen to find yourself in this situation, it can be faster to set a ByeBug breakpoint, and as a bonus, there will be less binding.pry's in your code to clean up afterwards.

Setting Breakpoints

Command	Result
break SomeClass#run	Break at the start of `SomeClass#run`
break Foo#bar if baz?	Break at `Foo#bar` only if `baz?`
break app/models/user.rb:15	Break at line 15 in user.rb
break 14	Break at line 14 in the current file

Editing Breakpoints

Command	Result
breakpoints	List all breakpoints
break –delete 5	Delete breakpoint #5
break –disable 5	Disable breakpoint #5
break –enable 5	Enable breakpoint #5
break –disable-all	Disable all breakpoints
break –delete-all	Delete all breakpoints
break –enable-all	Enable all breakpoints

Help I'm Stuck in Pry Loop Hell!

Problem

You put a binding.pry right smack in the middle of a long loop and you realize your mistake just as you hit refresh. Now every time you type exit, you just end up on the next cycle of the loop.

```
1  so_many_things.each do |a_thing|
2    binding.pry
3  end
```

This happens to me way more often than I'm comfortable admitting. Better just cut your losses and terminate the process right?

Solution: Triple Bang

This one's easy. The command `!!!` tells Pry to exit and not hit any more stops along the way. Do not pass go. Do not collect $200.

```
1  > !!!
```

```
From: /Users/andrewallen/dev/test_blog/app/controllers/posts_controller.rb @ line 85 PostsController#another_method:

    83: def another_method
    84:   binding.pry
 => 85:   return 1 / 0
    86: end

[2] pry(#<PostsController>)> !!!
Completed 500 Internal Server Error in 3729ms (ActiveRecord: 0.0ms)

SystemExit - exit:
  pry (0.10.1) lib/pry/commands/exit_program.rb:16:in `process'
  pry (0.10.1) lib/pry/command.rb:582:in `call'
  pry (0.10.1) lib/pry/command.rb:456:in `call_with_hooks'
  pry (0.10.1) lib/pry/command.rb:427:in `call_safely'
```

Triple bang terminates the current Pry session

Keep It Running

Triple bang will exit the Pry session with a failure state. But what if you want to continue execution, but just not hit any more `binding.pry` breakpoints? Maybe you're in the middle of a test suite or are running Guard.

The `disable-pry` command does exactly that:

```
1  › disable-pry
```

Controlling When Pry Fires

Problem

It canâ€™t be avoided. We need to drop binding.pry into a loop over a large collection of items, but really, we only care a specific item or type of item. Maybe it has a certain id or will respond to a method in a particular way.

```
so_many_things.each do |a_thing|
  binding.pry
end
```

Solution: Ruby Conditionals

This is one of those forehead-slapping, why-didnâ€™t-I-think-of-that-sooner kind of solutions. With simple Ruby conditionals, we can trigger a debugging session only when the item we care about comes up in the loop.

```
so_many_things.each do |a_thing|
  binding.pry if a_thing.we_care_about_it?
  binding.pry if a_thing.id == 123
end
```

This also helps us avoid Pry Loop Hell by only triggering Pry only on the items we care about instead of on every step of a long loop.

Detailed Error Backtraces in Pry

Problem

Occasionally, you'll be in a Pry session and need more information about an exception. You need to see the full backtrace to see where it all went wrong, but Pry only shows you the top level error message.

Ideally, you'd just let it error out and view the backtrace with Better Errors, but sometimes, that's not possible. For example, you might be working on an API that renders JSON (so you can't can't see the Better Errors page) and the part that causes an exception is being rescued (so the backtrace will never make it into the logs).

Solution: WTF?!

This is one of my favorite 'easter eggs' in any library. The Pry command wtf will print more information about the last error, including the backtrace. If you need more lines of backtrace, continue adding ?s and !s until you get what you need.

```
From: /Users/andrewallen/dev/test_blog/app/controllers/posts_controller.rb @ line 85 PostsController#another_method:

    83: def another_method
    84:   binding.pry
 => 85:   return 1 / 0
    86: end

[2] pry(#<PostsController>)> !!!
Completed 500 Internal Server Error in 3729ms (ActiveRecord: 0.0ms)

SystemExit - exit:
  pry (0.10.1) lib/pry/commands/exit_program.rb:16:in `process'
  pry (0.10.1) lib/pry/command.rb:582:in `call'
  pry (0.10.1) lib/pry/command.rb:456:in `call_with_hooks'
  pry (0.10.1) lib/pry/command.rb:427:in `call_safely'
```

Wtf?!?!?!!!???

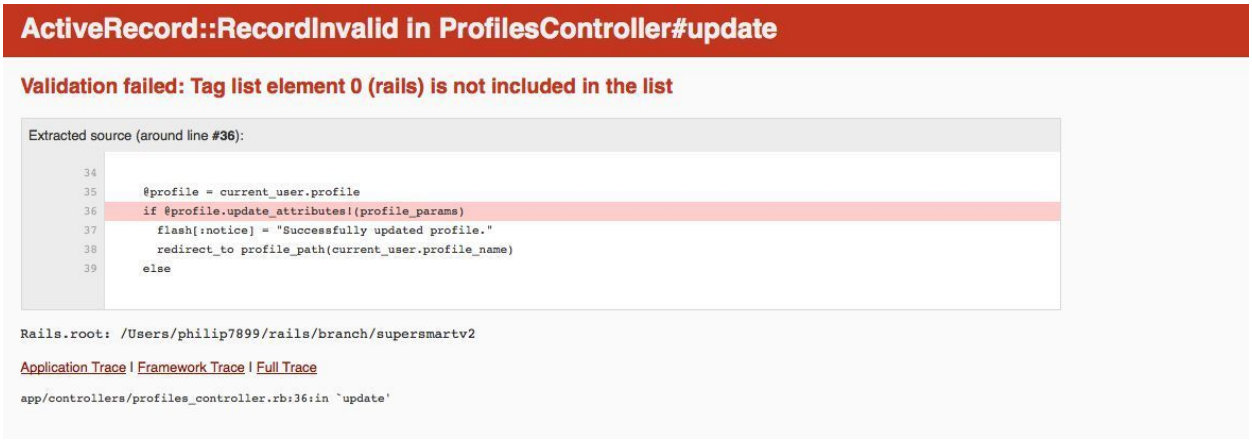
Beautiful :)

Making the Best of a Bad Situation

Problem

The default error pages leave a lot to be desired in Rails.

With Rails 4, we've seen some improvements to the utility of the error page, but it's still not as helpful as it could be.



Rails 4 default error page

This shows us the error, but not much else. We'd need to go back the source and try to figure out what went wrong, or start setting up a debugging session manually.

Thankfully, we have a few options to get more information when something goes wrong.

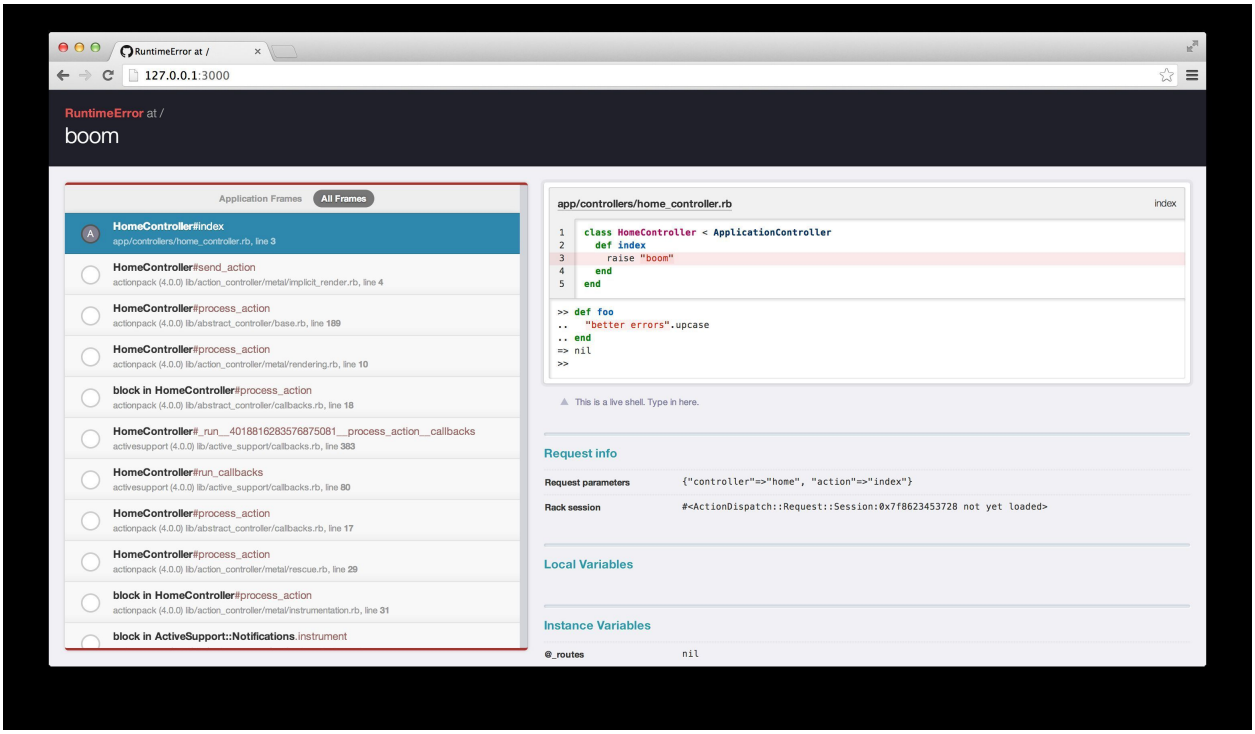
Solution #1: Better Errors

Better Errors turns the default Rails error page into a powerful debugging environment, right when you need it most.

To install, simply add it to your Gemfile:

```
group :development do
  gem 'better_errors'
end
```

Now, whenever Rails throws an error, it will use the following error page:



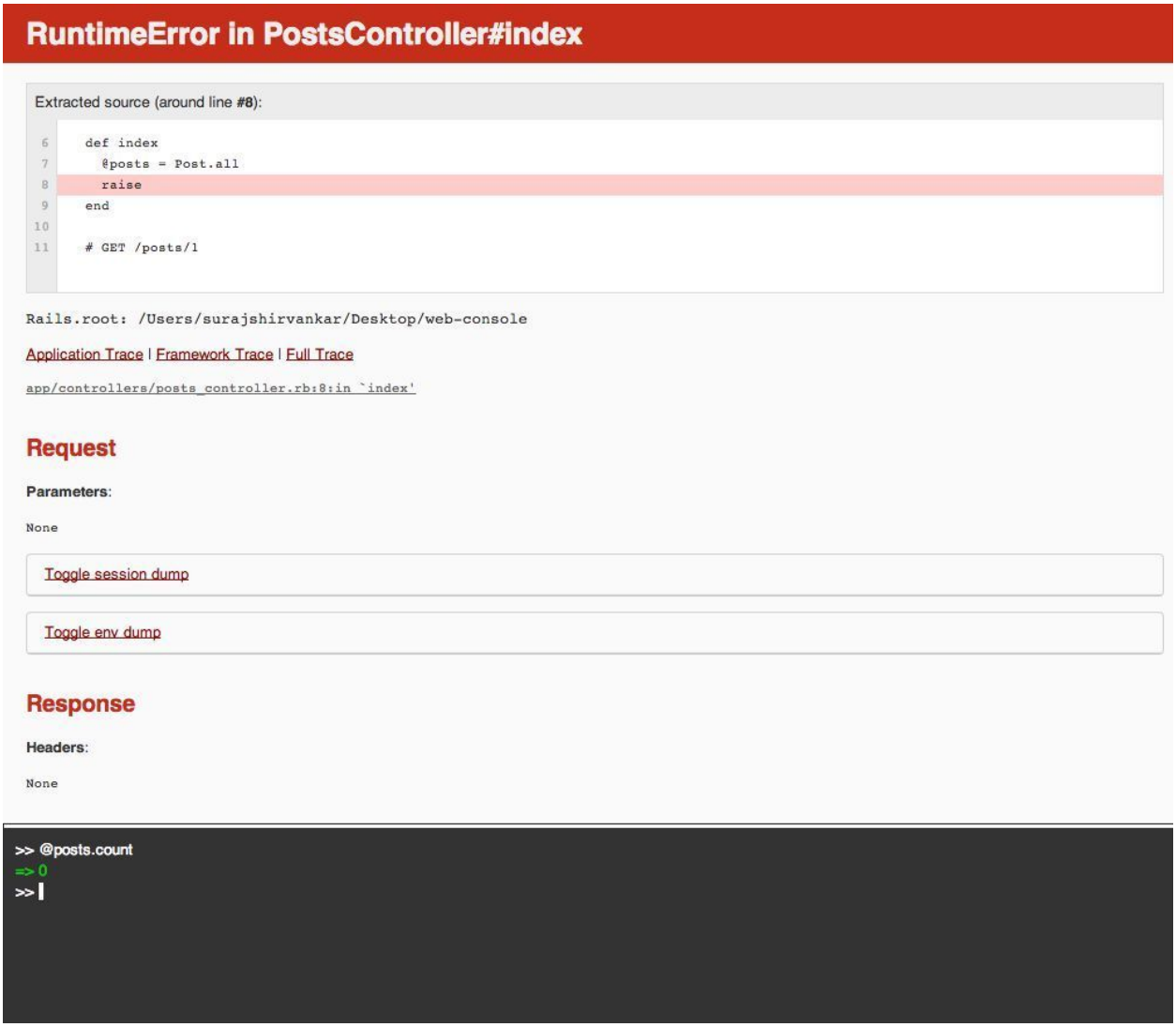
Better Errors in action

Not only can you see the entire environment at the time it hit the error, but you can actually run Ruby code in the console to better understand how your code is working and find a solution to the error.

The left-hand side of the page shows the frames of the application's call stack leading up to the error, but what's really cool is that you can click on a frame and the we console and environment view will change to represent that context.

Solution #2: Rails 4 Web Console

Rails 4 took inspiration from Better Errors and built a web-based console into the core framework. It's useful if you don't have Better Errors installed and its inclusion in the core framework shows that Rails is heading in the right direction, but I still prefer Better Errors. It's simply more stable and provides more functionality than the Rails 4 web console.



Rails 4 web console

Debugging APIs and Network Requests

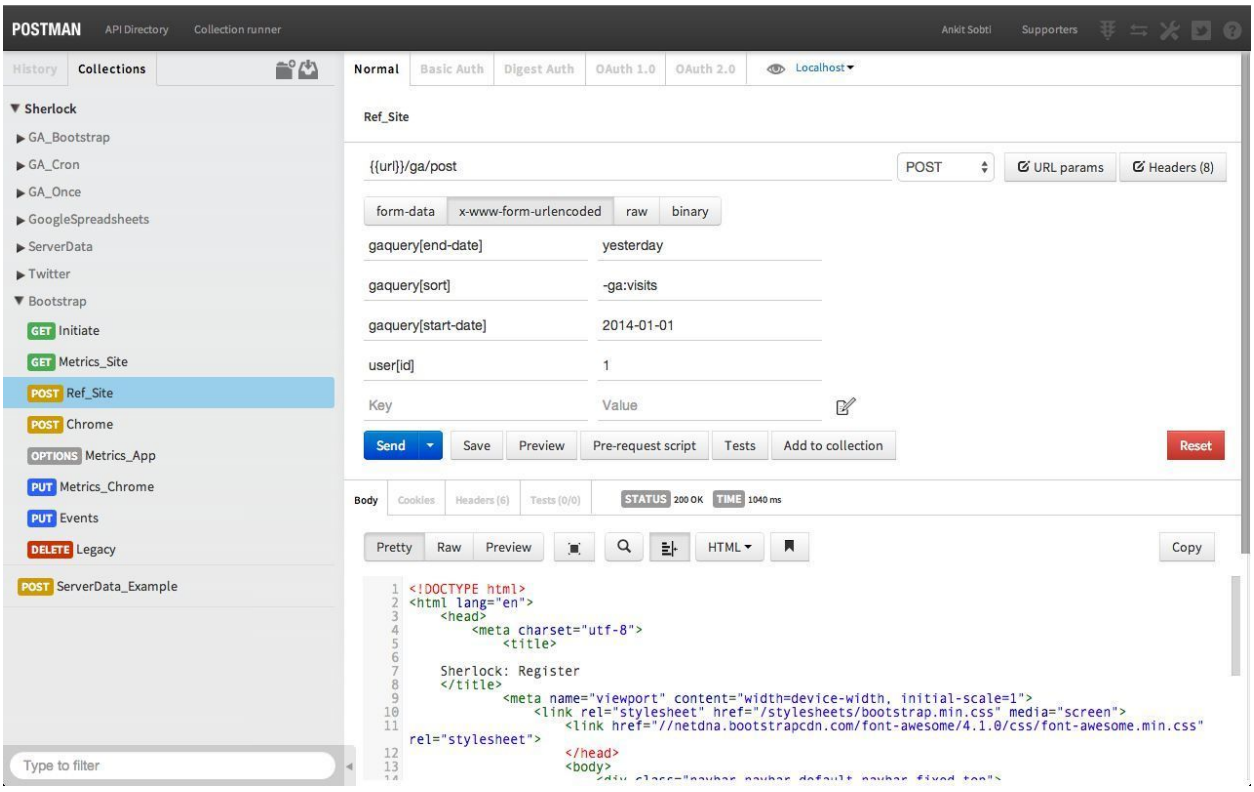
Problem

You're working on an API, maybe a set of controllers for a mobile app or a JavaScript single page application. The API exposes RESTful routes and communicates via JSON, but how do you test it to make sure it works as expected?

Sure you can put the route into the browser's address bar and see the JSON output, but that will only work for GET requests. What about creating, updating and deleting objects?

Solution: Postman for Chrome

If you've ever spent time stringing together complex Curl queries, stopping every few minutes to look up another obscure flag, Postman is a serious upgrade. While Curl can be helpful for a quick smoke test, Postman pays dividends when debugging an extensive API.

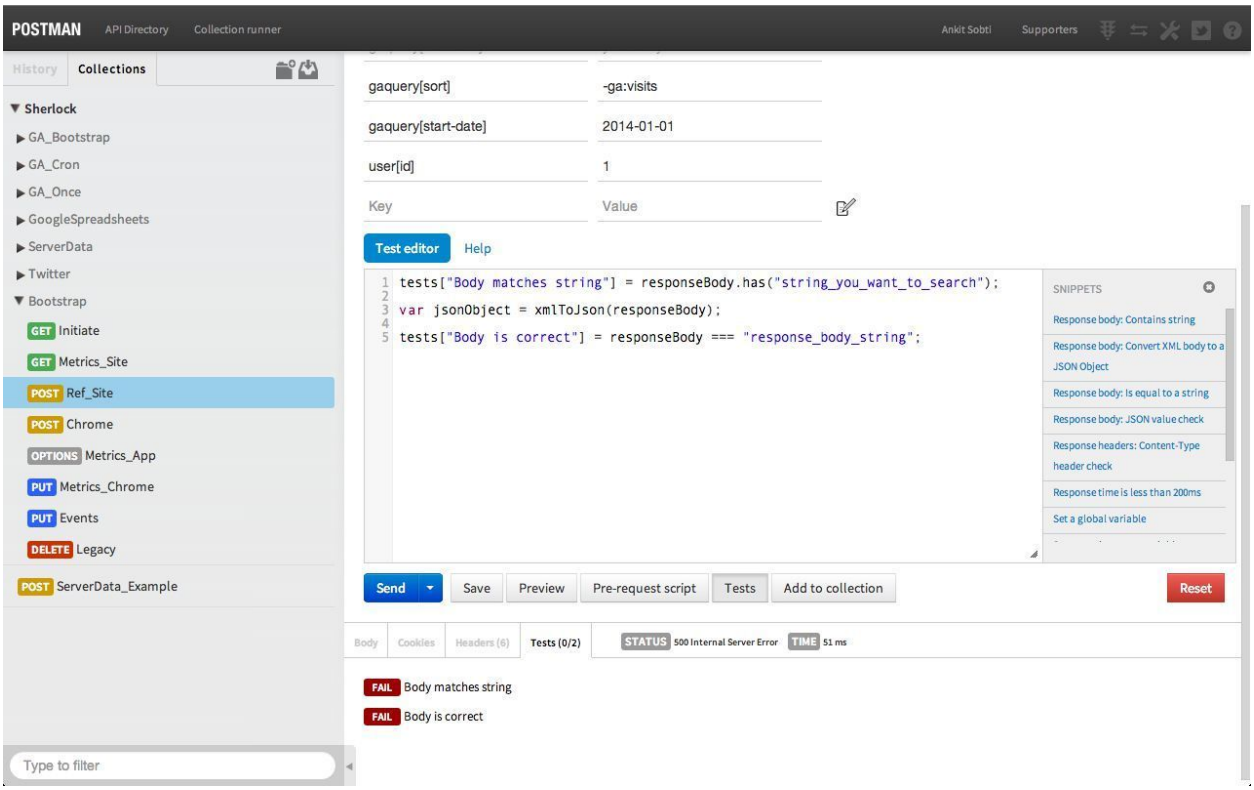


Postman makes API debugging a breeze

Postman keeps track of your request history and lets you save them into collections for future use. You can even share these collections with other developers testing the API.

Headers, URL params and Cookies are all accessible and easy to edit, no special flags necessary.

You can even use Postman to build a test suite, that hits a series of endpoints and asserts some expected response.



Building an API test suite with Postman

Postman is available for free as a standalone application. They also have a cloud plan for a very reasonable $4.99/mo that lets you share your API requests and test suites with a team. If your team does a lot of of API work, this can be a big help.

Testing an API from a Real Device

Problem

We can use Postman to test the routes of our API, and Chrome's network inspector to see what's going on with a failing request, but what if we want to try our app on an actual device? Setting up a staging server with DNS that's accessible from anywhere takes time, and pushing the new app every time you iterate gets slow pretty quickly.

Solution: Ngrok

Ngrok is a free tool that allows you to expose a port of your local machine to the internet at large. It can be installed from Homebrew with brew install ngrok.

Assuming we have our Rails server running on port 3000, we can expose this local server with ngrok by running ngrok 3000.

Ngrok will assign your instance a unique url, in the case of the example above, our url is http://4162e462.ngrok.com. Accessing this url from a mobile device on any wifi connection will direct the request right to our rails server running on our local machine. This means we can make changes and see the results instantly.

When you run ngrok, it also runs a pretty web dashboard on http://localhost:4040 that gives you a visual interface to inspect the requests your clients are making.

One drawback to running ngrok like this is that each time the session restarts, you'll get a new unique url. If you've changed the code in a mobile app to point to ngrok's URL, it can be frustrating to change it every time you restart ngrok. Luckily, ngrok allows users to register their own subdomain (assuming it doesn't conflict with anyone else's).

To start ngrok with our own subdomain, we simply use the -subdomain option: ngrok -subdomain efficient_rails 3000.

Ngrok offers many other advanced features such as password protecting your URL, handling TLS connections and using custom domains. Check out their documentation[66] to learn more.

[66] https://ngrok.com/docs

Debugging JavaScript

Problem

Whether you're building a full-on single page application with whatever JavaScript MVVC framework is currently the flavor or the week, or simply adding interactivity to an otherwise ordinary Rails application, at some point, something will go wrong in the JavaScript code. But where to begin debugging?

Solution #1: Chrome's Network Inspector

When something's not right in your JavaScript application, Chrome's network request panel is usually the first place to check. This will show you right away if the bugginess is the result of a failed API request.

From this panel, you can click the response in question to see the full request that was sent from the JavaScript client, and the error response from the server.

By default, the network panel will show all requests. This includes assets, images and fonts. To filter by the kinds of requests we're interested in, we can use the "XHR" filter to show only JSON-based requests and the "Documents" filter to show requests for full HTML pages.

Solution #2: Chrome's JavaScript Debugger

Under the hood, Chrome provides a surprisingly advanced debugging platform. Compared to what we have to work with in Ruby, the visual nature of Chrome's debugger is refreshing.

Breakpoints can be defined in two ways: 1. Explicitly in your JavaScript code with `debugger` 2. Visually by clicking on the line number in the sources pane of the Chrome inspector

Now, before this line is executed, Chrome will freeze execution and allow you to inspect the state of the application at runtime.

From this view, you'll be able to access the call stack, local variables and even execute JavaScript in the Console pane.

We'll cover more intricacies of Chrome's developer suite in a later section on front-end Rails development.

jQuery in the Developer Tools

Problem

You're debugging some jQuery in the browser's developer tools and want to have a reference to a DOM element, wrapped in jQuery, in the console. To do that, you need to figure out what its id or classes are. And if you're selecting by class or some other non-specific selector, there might be multiple results. How do you ensure you're working with the right element?

Solution

Whether you use Chrome, Safari or Firefox, their developer tools all share a common secret feature.

In the console, `$0` refers to the current element you selected with the inspector tool.

So if you want to wrap that element with jQuery, you can simply select it with the inspector (Cmd+Opt c in Chrome), and then type: `$($0)` in the console, and presto, instant jQuery object.

It doesn't stop there! `$1` refers to the *previously* selected object, `$2` refers to the object selected before that, and so on. So if you wanted to reference multiple elements on the page in your jQuery, you could select multiple objects on the page for inspection (Cmd+Opt c -> click, Cmd+Opt c ->, ...) and then reference them all in the console as needed.

Don't Let Debugging Statements Slip into Production

Problem

You debug a bug a little too quickly and suddenly you're getting 500 errors in production. You realize you left a binding.pry in your code and it made its way into production where it's currently wreaking havoc.

Solution #1: Git Hooks

Git hooks allow code to be run on certain events. In order to prevent debugging statements from getting committed, we'll use a pre-commit client hook that will run on our local machine right before the commit is saved.

Bob Gilmore's githooks project[67] includes a Git pre-commit hook with a number of checks for bad code, including Ruby and JavaScript debugging statements. You can check out the Readme for the other checks this hook performs.

To install:

1. Clone the repo to your home directory, or another central location
2. Run the setup script: `./setup.sh`
3. Run git init in your project's directory. This will reinitialize the repository, but won't overwrite your git history.
4. You'll have to commit once for the hook to install itself, and from then it will run as expected

```
# andrewallen at Andrews-MacBook-Pro.local in ~/dev/test_blog on git:master x [12:55:12]
$ commit "oops"
ERROR: git pre-commit hook found the following problems...

"binding.pry" in app/controllers/posts_controller.rb
--------------
To commit anyway, use --no-verify
```

Git hooks prevent bad code from getting committed in the first place

Solution #2: Use Sed to Delete All Pry Bindings

You can use some bash trickery to delete all occurrences of binding.pry from your codebase. To do this, we'll combine two unix tools, find and sed. Find will search for all files in the current directory whose extension ends with "rb". This will catch both .rb files and .erb files.

Sed will operate on each ruby file, removing any lines containing "binding.pry". Put together, it looks like this:

[67] https://github.com/bobgilmore/githooks

```
$ LC_CTYPE=C LANG=C \
$ find ./ -type f -regex ".*rb$" \
$ -exec sed -i '' -e '/binding.pry/d' {} \;
```

The environment variables fixes a strange OS X-specific issue with sed. If you're curious, you can read more here.

For maximum convenience, store it in a bash alias. I use unpry.

```
alias unpry="LC_CTYPE=C LANG=C find -E ./ -type f -regex '.*rb$' -exec sed -i ''\
 -e '/binding.pry/d' {} \;"
```

Revisions

2016-06-23

- Started tracking revisions!
- Reformatted the PDF version with better fonts and styling, and footnote capability
- Added mobi and epub versions
- Finished Models/Counter Cache
- Added example Guard to Models/Preventing Missing Keys in the Future
- Added a takeaway to the Art of Command Chaining sidebar
- Removed Terminal/Rails Destroy section
- Minor proofreading fixes