Practicing Rails

Learn Rails Without Being Overwhelmed

Justin Weiss

Version 1.0, built February 11, 2015.

Table of Contents

Introduction 6
Tiny Apps: The best way to study new Rails ideas 9
  Using Rails to learn Rails 13
  How to answer questions with your new app playground 28
  What do you do when you’re done with a tiny app? 32
  Putting it all together 34
How to build your own Rails app 36
  What to build first 37
  Turning your ideas into running Rails apps 46
  Which feature do you build next? 55
  Putting it all together 64
Testing your code efficiently and effectively 68
  How to write great tests quickly 69
  Better ways to structure your tests 79
  What’s the best way to learn Test-Driven Development? 90
  Putting it all together 93
What you need to learn, and what you can skip (for now) 96
  How deep you really need to go 97

Your Rails learning roadmap 100
"For Now" is a powerful phrase 111
Putting it all together 113
What to do when things go wrong 114
Errors: How do you fix them? 115
Debugging: A developer's most valuable skill 123
Research: When debugging fails 136
The most common Rails errors, and how to fix them 141
Putting it all together 143
Keeping up with the Rails community 145
When to give a new tool a chance 150
Staying up-to-date with your libraries 154
Putting it all together 158
How do you learn Rails when you don't have the time? 160
What to do when your motivation abandons you 166
Putting it all together 171
Epilogue 173
The best ways to continue mastering Rails 176

To my wife and daughters, for their patience, love, excitement, and patience.

Introduction

[This book] is meant to be practical. It is meant for use.

– Mindfulness in Plain English[1]

I'm addicted to learning.

Right next to me, I see books on code generation, agile processes, statistics, writing, JavaScript, Erlang, and a lot more. And that doesn't even count the Kindle.

I want to read them all. But if I'm not careful, reading them will be a total waste of time. I'll read one, I'll love it, I'll set it down, and a week later I'll have forgotten everything in it.

Books are my favorite way to learn something new. But even after buying and reading tons of books, you'll still get lost when you try things on your own. And at that point, have you learned? Or are you just transcribing?

1. http://www.urbandharma.org/udharma4/mpe1-4.html

It seems so easy when it's words on a page. But when you try to transform *your* ideas into code, out comes the stress, the frustration, and the worry. You don't have the time to study everything all over again, so did you spend that time learning for nothing?

This is exactly how I felt learning Rails. It's how I felt when I first started writing, and playing Go, and using Emacs. But it doesn't have to stop you.

It is possible to learn Rails without having the things you learn abandon you as soon as you try to grab ahold of them.

That's what this book is about. It's a second book of Rails. A companion. It'll show you how to learn the most in the least amount of time, using the resources and knowledge you already have. And in the process, I'll guide you through some of the toughest lessons I've learned during my programming and Rails career.

I have three rules for reading this book:

1. It's not you, it's me.

   If there's anything you don't understand here, ask me about it. This stuff gets complicated, so it's not your fault for misunderstanding it. It's my fault for not explaining it well enough! Send me an email (justin@justinweiss.com), and I'll do my best to clear it up.

2. Embrace struggle, failure, and reward.

   Learning to become a great Rails developer is hard work. And, being hard work, the only way to learn is to struggle. I've heard programming described as “Running into a brick wall, constantly.” So don’t worry if you feel that way – we all do.

If it came easily, it wouldn’t feel so great when you get it. Seriously, finally fixing a problem you’ve been fighting for the last hour is the best feeling in the world. So struggling should tell you that you’re on the brink of learning something really valuable.

3. Find the fun.

   Every developer I know has those areas of programming that they love to do more than anything else.

   For some people, one area is data modeling. Have them draw some boxes and lines on a whiteboard, and they can talk for hours. For others, it’s seeing their stuff running on other people’s phones and computers. The magic of deployment. For me, it’s refactoring. Transforming a chunk of terrible code into something clean and beautiful.

   You’ll have your own preferences. So while you follow this book, search for the things that really resonate with you, the things you get lost in, the things you just want to do for hours.

   Keep those in mind. Because when you get frustrated, you’ll have something you like to do to come back to. It’ll remind you why you’re doing this, and help you enjoy some of the fun before you dive back into the frustrating.

Learning isn’t just about reading. It’s also about action. And that is why you can’t learn Rails without practicing Rails.

Chapter 1
Tiny Apps: The best way to study new Rails ideas

Lose your first 50 games as quickly as possible.

– Go Proverb[2]

Build your first 50 Rails apps as quickly as possible.

– Go Proverb (paraphrased)

Books on software development are huge. If I really try, I can get through one in a week. That is, if I'm commuting by bus, and all the expressways are closed, and there's some kind of protest going on… Otherwise, it takes even longer.

2. http://senseis.xmp.net/?LoseYourFirst50GamesAsQuicklyAsPossible

But I don't read programming books on the bus. Because when I do, the information drains out of my mind before I get home. Instead, I read them in front of my computer, so I can try things out. Sure, it takes longer to get through each one. But what's the point of reading non-fiction if you're just going to forget what you learned when it matters?

The best way to learn new Rails ideas and techniques is to *use* them. Right away. Practice them, internalize them, and make those techniques yours.

Rails gives you two ways to try new ideas right away: generators and scaffolds. With three lines, you have an entire Rails app ready to use:

```
rails new test_app
cd test_app
bin/rails generate scaffold some_model name:string
```

You can try anything: routing, HTML, associations, filters, callbacks, validations, and everything else.

This isn't an app you'll actually ship. It's only there to help you learn. When you generate a tiny app like this, you go from book to playground in seconds. That's powerful.

I generate apps for all kinds of things[3]:

3. This command finds all the directories I have inside `~/Source/testapps`, and counts them. And if you're ever confused about a terminal command like this, try typing the command into Explain Shell[4].
4. http://explainshell.com/

```
ls ~/Source/testapps | wc -l
52
```

That is, I've generated *fifty-two* Rails apps, just to try out ideas I have. I can explore new Rails features, use tricks I learn from blog posts, do research for my articles and this book, and play with anything else I can think of.

(And those are from *after* I sort-of-accidentally wiped my hard drive three months ago).

What's the difference between `rails` and `bin/rails`?

Sometimes, you'll use `rails` or `bundle exec rails`, and other times you'll use `bin/rails`. Why is that?

When you're already inside a Rails app, you should use `bin/rails` and `bin/rake`. These commands are created when Rails generates your app, and the commands will use the right version of Rails for that app. In Rails 4.1 and higher, `bin/rails` will probably be faster than running `bundle exec rails`.

But when you don't have a Rails app, those files in `bin/` don't exist yet, so you can't use them.

So, when you're inside a Rails app, use `bin/rails`. When you're trying to create a new Rails app, use `rails`. Yep, I wish this was less confusing, too!

All of those are just Rails 4.1 or 4.2 apps, using Rails defaults, with one or two generated models. Nothing complicated, just a playground for learning and breaking things.

Learning from tiny apps

When you're working with such small apps, you focus on the single thing you want to learn. That way, you don't have to worry about learning that new idea inside your existing, more complicated apps.

For example, if you're trying to learn `ActiveModel::Serializers`, and your controller isn't serializing the model correctly, you can never be sure if the problem's a gap in your knowledge, a problem with Rails, or a problem with your app.

If you tried this out in a tiny test app, you could be pretty sure the problem was just a simple mistake or a misunderstanding. You can catch these mistakes on a small scale before you fight the bugs that appear when you use them into a larger app.

Soon, you'll go from "technique I just read about" to "this seems interesting" to "fully running Rails app using that technique" *without having to think about it*.

Wrapping it up

If you really want to learn something, you have to use it. This goes for Rails, too. And the fastest way to learn new Rails concepts is to scaffold new apps where you can try them out.

When you explore new concepts in tiny test apps, you'll be able to focus on the idea you're trying to learn. You can try a technique in different situations. You'll get practice using it. And you won't have to worry as much about integration pain or errors.

Using Rails to learn Rails

So, you're ready to try something new inside a tiny Rails app. You're still missing a few steps, though:

- How can you build a realistic-enough Rails app to use as your playground?
- How do you learn a new concept inside a real, tiny Rails app?

Coming up with quick ideas

Models called `Test1` aren't that helpful. I have a hard enough time understanding new Rails concepts without also having to fight against bad class names. So, you need realistic-looking models, resources, views, and controllers. You need a system that you know well, but can still build quickly. And that starts with an idea.

Coming up with ideas on the fly is hard. And if you come up with new app ideas every time you want to try something out, you're just wasting time. Instead, think of a few different app types you can reuse.

What do I mean by an app type?

An app type is an idea for an app that you'll use when you generate new tiny apps. It might be something you've built already, or something you've been thinking about building.

Here are some examples of app types I've used:

- A blog
- A link aggregator, like Reddit or Hacker News
- A forum, like Discourse
- A Q&A site, like Stack Overflow

These might seem big, but remember: you're not building a full-featured app, you're building a tiny app. You're building the bare minimum you need to learn something new about Rails. The app type is just meant to make naming your models and controllers easier, and to help you picture the relationships between your objects.

When I build a tiny app, I use the idea of a bug tracker. So when I generate a Rails app to try out something new, I'll start by generating a Bug model. If I'm learning something that needs associations, I'll also generate a User or Project model.

The details don't matter too much, so feel free to choose one of the examples I mentioned.

Why choose an app type now?

When you build a lot of tiny apps, you'll want to keep going back to the same object model, the same user interface (UI), the same resources. Things you understand, that you can rely on.

You’re already working hard on learning a new Rails feature. So you don’t need to spend time coming up with a whole new app, new metaphors, and a new data model every single time.

Plus, as you keep coming back to the same generic app, you’ll get faster at generating the scaffolds, models, views, and controllers you’ll need. So you’ll spend less time doing busywork and more time learning.

How to build a tiny app

Most of my sample apps start like this:

```
rails new test_polymorphic_association
cd test_polymorphic_association
bin/rails generate scaffold bug title:string description:text
bin/rake db:migrate
```

This gives you a simple app that you can try out in your browser. You can hack on it, break it, and play with it. You spent almost zero time building the app, so you don’t have to worry about screwing it up. You can do anything you want to it!

But… scaffolds?

You might hear that Rails developers don’t use scaffolds in the real world. I totally disagree with that. I use scaffolds all the time, even though I could write the code from scratch. Why?

In real apps, I can’t ship great features until I play with them first. With scaffolds, you can do that without taking much time. If you change your mind about how your feature’s designed, you can tear it out. No big deal.

And sure, you might replace the scaffold before you ship. But I prefer to spend as little time as possible on the code until I’m happy with how a feature works.

I care about getting the most knowledge in the least amount of time, and scaffolds and other Rails code generators are a great way to do just that.

Exploring inside a working Rails app

Next, take the idea you’re learning and translate it into a working Rails app. This takes practice, and usually goes like this:

- Take the thing you’re learning and put it into the right place in your app.
- If it doesn’t need to be accessed through the UI, try it out in the Rails console.
- If you touched the controllers or views, boot your Rails server and play with your code through the browser.
- Try a version of the idea that’s a little more complicated, and get it working again.

Where does the code go?

How do you go from “feature in a book” to “website I built that uses that feature?”

First, generate your scaffolds. If your go-to app type is a bug tracker, generate a scaffold for the `Bug` model. For larger features, generate a few other scaffolds.

Now you have some places to put validations, controller code, and view code. Do your best to imitate the code that you're reading. Just copy the code and put model code into the model, view code into the view, and so on. If you don't know where the code goes, you can try it in the Rails console, which I'll talk about in the next section.

When you've generated the scaffold and added code, map the `root` route to the `index` action of the new controller:

config/routes.rb

```
root 'bugs#index'
```

After that, start up a server:

```
bin/rails server
```

visit `http://localhost:3000` in your browser, and try your scaffold out!

Playing with these apps in your browser can be slow, though. You have to deal with controller and view code that you might not need for the thing you're trying to learn. There could also be new Rails concepts that are hard to play with through the browser. You'd rather just write the code, and see it run.

There are a few good ways to do just that.

Trying out new ideas, faster

You don't have to bring up a whole Rails server if you want to try something out. Instead, you can often take a lighter-weight approach. You can use the Rails console.

For example, I wrote an article about Rails' validation contexts[5]. With validation contexts, you could have an `Article` class that looked like this:

app/models/article.rb

```ruby
class Article < ActiveRecord::Base
  # validate in draft mode
  validates_presence_of :author_id
  # validate only when published
  validates_presence_of :body, on: :publish
end
```

And the body attribute would only get validated if you called `valid(:publish)` on the article.

I was curious whether calling `valid?` twice with different contexts would *add* to the `errors`, or if it would overwrite them. When I investigated it, I never had to start a Rails server. I used the console instead.

You start a rails console by running `bin/rails console` inside your app's main directory. Once you're in, you can call methods, define classes, do pretty much everything you'd do in code, and it'll just run:

5. http://www.justinweiss.com/blog/2014/09/15/a-lightweight-way-to-handle-different-validation-situations/

```
$ bin/rails console
Loading development environment (Rails 4.1.5)
irb(main):001:0> a = Article.new
=> #<Article id: nil, title: nil, body: nil, author_id: nil,
      created_at: nil, updated_at: nil>
irb(main):002:0> a.valid?(:publish)
=> false
irb(main):003:0> a.errors.messages
=> {:author_id=>["can't be blank"], :body=>["can't be blank"]}
irb(main):004:0> a.valid?
=> false
irb(main):005:0> a.errors.messages
=> {:author_id=>["can't be blank"]}
irb(main):006:0> exit
```

Yep, looks like the errors are overwritten.

Trying validation contexts this way was really easy. Imagine how much more work it would be to test this from the views!

(Note that here, I used `Article` instead of `Bug`, because I couldn't come up with a good example of validation contexts within a bug tracker. This is why having a few different app types in mind is handy).

Some tips for investigating ideas through the Rails console

Rails is designed for the web, so it might seem like there's a lot that you can't test through the console. But there are some ways you can get further than you might expect.

First, you can use a lot of Rails features through the `app` object. `app` is a special object that has some useful methods for experimenting with your Rails app[6].

For example, you can try out your routes:

```
irb(main):001:0> bug = Bug.create(:name => "Some bug name")
(0.3ms)  begin transaction
  SQL (2.9ms)  INSERT INTO "bugs" ("created_at", "title",
      "updated_at") VALUES (?, ?, ?)  [["created_at", "2014-11-12
      07:51:02.211664"], ["title", "Some bug name"], ["updated_at",
      "2014-11-12 07:51:02.211664"]]
   (1.2ms)  commit transaction
=> #<Bug id: 1, title: "Some bug name", description: nil,
      created_at: "2014-11-12 07:51:02", updated_at: "2014-11-12
      07:51:02">
irb(main):002:0> app.bug_path bug
=> "/bugs/1"
```

You can make web requests, so you can run controller and view code:

6. In case you're curious, `app` provides all of the Rails integration test methods: http://guides.rubyonrails.org/testing.html#helpers-available-for-integration-tests

```
irb(main):003:0> app.get "/bugs/1"
Started GET "/bugs/1" for 127.0.0.1 at 2014-07-09 06:16:21 -0700
  ActiveRecord::SchemaMigration Load (0.6ms)  SELECT
       "schema_migrations".* FROM "schema_migrations"
Processing by BugsController#show as HTML
  Parameters: {"id"=>"1"}
  Bug Load (0.3ms)  SELECT  "bugs".* FROM "bugs"  WHERE "bugs"."id"
       = ? LIMIT 1  [["id", 1]]
  Rendered bugs/show.html.erb within layouts/application (17.2ms)
Completed 200 OK in 228ms (Views: 180.7ms | ActiveRecord: 0.3ms)
=> 200
irb(main):003:0> puts app.response.body.first(200)
<!DOCTYPE html>
<html>
<head>
  <title>Bugsmash</title>
  <link data-turbolinks-track="true" href="/assets/bugs.css?body=1"
       media="all" rel="stylesheet" />
<link data-turbolinks-track="true" href="/as
=> nil
```

The console also gives you `helper`. The `helper` object provides all of your app's view and helper methods in the Rails console:

```
irb(main):005:0> helper.content_tag :h1, "Hey there"
=> "<h1>Hey there</h1>"
```

`helper` works with Rails view methods like `content_tag` and `form_for`, and any methods you define in `app/helpers`. It's an easy way to play with code you'll use inside a view.

Finally, there's _. The result of the last line you ran in the console is automatically saved in a variable named _. You can use it in the next line:

```
irb(main):001:0> Article.new
=> #<Article id: nil, title: nil, body: nil, author_id: nil,
      created_at: nil, updated_at: nil>
irb(main):002:0> _.valid?
=> false
```

Here, I didn't need to assign `Article.new` to a variable, because I could just use _ to reference it.

The Rails console is the fastest way you can try the things you're learning. It's the tool I reach for first when I need to try out something new.

Automating your experiments with tests

The Rails console is helpful, but you have to type the same things each time you start a new console session. You'll need to find or build your object, perform some actions, and only then can you do what you came into the console to try out.

But Rails already has a way of running code quickly and repeatedly: test files. Besides using your test files to test code, you can also use them to experiment with code. For example, going back to the Article example from earlier:

test/models/article_test.rb

```
require 'test_helper'
class ArticleTest < ActiveSupport::TestCase
  test "seeing if errors are overridden when valid? is called twice
       with different contexts" do
    article = Article.new
    article.valid?(:publish)
    puts
    puts "--After calling valid?(:publish)"
    puts article.errors.full_messages
    article.valid?
    puts "--After calling valid? again"
    puts article.errors.full_messages
  end
end
```

You can run the test with `bin/rake`:

```
bin/rake
Run options: --seed 895
# Running:
--After calling valid?(:publish)
Author can't be blank
Body can't be blank
--After calling valid? again
Author can't be blank
.F.....F
Finished in 0.458934s, 17.4317 runs/s, 21.7896 assertions/s.
```

This might look weird, because there aren't any assertions. And the output isn't that great to read. But remember, this is just about experimentation. I no longer have to worry about setting up my objects, because all the setup code is in my test file, which I can run whenever I want.

Focused tests

Ruby has a quick way to run a single test case (or set of test cases). When you run a single test, you get clearer, faster responses. Using the last example, from your Rails root, you could run just the test case we wrote:

```
ruby -Itest test/models/article_test.rb -n "test_seeing_if_errors_
      are_overridden_when_valid_is_called_twice_with_different_
      contexts"
```

Let's take a closer look at that command. The `-Itest` part tells Ruby to look in your Rails app's `test` directory when you use `require`. You need this, because your test files require `test/test_helper.rb`, which boots Rails so it can run your test.

`test/models/article_test.rb` is the name of the test file to run. In Ruby, you can run a test file as if it's just a Ruby script, and Ruby will automatically discover any test cases in them and run them.

Finally, the `-n "test_seeing_if_errors_are_overridden_when_valid?_is_called_twice_with_different_contexts"` tells Ruby which specific test to run.

The name of this method is a little bit different than what you defined earlier. When you use the `test "some friendly name"` syntax, Rails does two things to build the name that you use with `-n`:

1. Prefix the name with `test_`

2. Replace all spaces with _

So, if you said `test "my friendly name"`, you'd use `-n "test_my_friendly_name"` to run that test. A little annoying, but that's how it is.

The second thing you'll notice is that the test name is *really* long. If you don't feel like typing that much, you can use a different `-n`:

```
ruby -Itest test/models/article_test.rb -n "/is_called_twice/"
```

Instead of running only one test, this syntax will run *all* of the tests that have `is_called_twice` somewhere in the name. This is a lot shorter, but you might run a few tests instead of just one. In your sample apps, where you're just using tests to try things out, this probably won't matter.

Tests vs. the console

So, the console is great for messing with objects, but getting those objects set up can be hard. Tests are great for getting objects set up, but hard for messing with them. Can you get both?

If you use tests along with a gem called Pry[7], you can use your tests to set up the objects, then open a console with Pry. Once you're in the console, you can start experimenting with your code.

If you add Pry to your Gemfile:

7. http://pryrepl.org

Gemfile

```
gem 'pry', group: [:development, :test]
```

and install it:

```
bundle install
```

you can use `binding.pry` in your test, wherever you want to enter a console:

test/models/article_test.rb

```
test "seeing if errors are overridden when valid? is called twice
      with different contexts" do
  article = Article.new
  article.valid?(:publish)
  binding.pry
end
```

Then, when you run `bin/rake`, you'll see:

```
bin/rake
Run options: --seed 19674
# Running:
From: /Users/jweiss/Source/test_validation_contexts/test/models/
       article_test.rb @ line 7 ArticleTest#test_seeing_if_errors_are_
       overridden_when_valid?_is_called_twice_with_different_contexts:
    4: test "seeing if errors are overridden when valid? is called
       twice with different contexts" do
    5:   article = Article.new
    6:   article.valid?(:publish)
 => 7:   binding.pry
    8: end
[1] pry(#<ArticleTest>)> article.errors.messages
=> {:author_id=>["can't be blank"], :body=>["can't be blank"]}
```

It’s a console!

You can do most things here that you can do with the Rails console (but you won’t have the `app` and `helper` objects). Pry even has syntax highlighting and some really nice other features[8]. And when you combine Pry with tests, you can try out new ideas really quickly.

Wrapping it up

Tiny Rails apps are a great way to try new ideas out in a realistic way. When you use scaffolds and Rails generators, you can build these apps quickly.

So, when you’re introduced to a new idea, generate an app with a scaffold or two. Add the code you’re seeing to the right place in your app.

8. http://pryrepl.org

If you can access the code you added without going through the UI, use a Rails console to explore that code. Otherwise, start up your Rails server and try it out through the UI.

If you start to get annoyed with the amount of setup you have to do, put some of that setup code into a test. If you combine test cases with Pry, you can have your test case do all of the repeatable setup, and drop you off at the right place to experiment.

Exercise

Take the most recent Rails idea you heard about. (If you're having trouble thinking of one, try something like Active Record Enums[9]). Start a tiny Rails app using the process in this section, and get it built and working. Then, experiment with the tiny app. How can you break it? Look up the documentation. Are there any other features you could try?

How to answer questions with your new app playground

Copying ideas out of a book and into your tiny Rails apps will help you learn them faster. But in order to master them, you have to tweak the code you copy, until you discover all of their secrets and make them your own.

9. http://guides.rubyonrails.org/4_1_release_notes.html#active-record-enums

Testing the boundaries

Has this ever happened to you?

You read something in a Rails book, and start thinking about it. You get a sentence or two in, and questions start filling your head:

“Why does it work that way?”

“How could this possibly work?”

“What if I tried using it with this other idea I just learned about?”

All of a sudden, you’re ten pages past the last place you remember being, and you have to go back and re-read everything. Or, you assume that you learned the concept well enough, and get lost in the next chapter.

This is what happens to me when I read technical books. I just can’t let those questions go!

Most Rails books and videos are good at showing you what’s possible. But they can’t explain everything. Those gaps will raise questions, and you’ll naturally want to have those questions answered.

So, when I say “play with and modify the things you learn”, I mean “Answer the questions you have about the things you learn, using code.”

You can do this by searching for the documentation, and trying different configuration, different parameters, and methods that you didn’t know about before. By using things you just learned along with ideas you already know well, so you can see how they interact. And by trying to break things, which I’ll talk about next.

The searches, the error messages when things break, and the feeling of success when you discover something new, will all stick with you.

It's so much easier, and a lot more fun, than just reading a chapter in a book. Because that concept, those ideas, are now yours.

How can you break it?

One of my favorite ways to learn something new is to break it.

For example, say I'm exploring Rails 4.2's `config_for` method. That looks like this:

```
Rails.application.config_for(:redis)
```

That line will load `config/redis.yml`. If I'm in development mode, it'll look for configuration under the `development:` key inside that file, and so on. Then, it'll return the values under that key as a hash.

There's a lot you could play with here to answer some interesting questions. What if you passed a string instead of a symbol? What if you passed a path, instead of just a filename? Could you find a way to load a `.yml` in `lib/` instead? Could you pass an absolute path, like `/usr/local/etc/redis.yml`? What if your `.yml` file didn't have the Rails environments defined in it? What if it used yaml includes, like `&default`? Can you use `ERB` inside your config file?

Next, try some of these out, and see what happens.

When you use your curiosity to break things:

- You get to see different kinds of errors *when you're expecting them*. This is so helpful: It's less intimidating to see an error message when you already know you're going to get an error message, and you start to see certain messages really often. You'll develop intuition about what those error messages mean. You'll eventually start psychic debugging[10], where you can tell someone where the problem is in their code without even looking at it. (This is a really fun way to amaze and annoy your friends and coworkers).

- You'll start thinking of ways to break features without knowing how they're implemented. That's your first step to getting great at writing test cases. And it'll *really* help you once you start practicing Test-Driven Development[11].

- When you break things on purpose, you get more comfortable when things break. The way to deal with errors is by building curiosity, not fear. "Why is this thing breaking?", rather than "How can I make this problem go away!"

Wrapping it up

Once you have a good playground for trying something you just learned, use that playground to own that concept. Explore the boundaries of that concept until you feel like you really get it.

Start by brainstorming questions you have about the feature. Some of these questions will come to you naturally, and others you'll have to think a lot about.

10. http://blogs.msdn.com/b/oldnewthing/archive/2005/12/02/499389.aspx
11. Abbreviated TDD, test-driven development is a mode of testing where you write tests against code that you haven't written yet. You use those tests to tell you how to design your code.

If you're having trouble brainstorming questions, use "How can I break this feature?" to come up with some ideas.

Once you have some good questions, answer them on your own inside your tiny Rails app. You won't be able to answer them all, but you'll teach yourself a lot by trying.

What do you do when you're done with a tiny app?

When you're through experimenting with a tiny Rails app, you might be tempted to delete it. But I tend not to.

Storage is cheap, and Rails apps are pretty small. If you keep your playground apps alive after you're done with them, they can come in handy later:

- When the thing you learned comes up in a real app, your tiny app provides a great example of that idea being used.
- If you have more questions about that feature of Rails later on, you can avoid some work. Your playground is already there.
- It's rewarding to revisit your early work and see how much you've learned since then.

  We don't always feel like we're progressing as quickly as we want to. But when you look at your earlier code, you can really see how much you've grown. Your older code will seem primitive, and it'll take a lot of willpower to resist rewriting it right then.

So keep your old code around.

Don’t lose your apps!

But there’s no point keeping these apps around if can’t find them when you need them. I’m actually pretty bad at this. It’s just easier to type `rails new test8` than it is to come up with a good name. But this is the process I’ve been following:

Generate all of your test apps in their own folder. I’ve been using `~/Source/testapps`. That way, they don’t clog up whichever folder you’re using to work on real stuff.

Then, come up with a two or three word description of the specific feature you’re playing with. Some examples of things I’ve used are `validation_contexts`, `gemfile_groups`, `rc_files`, and `validation_mixins`. Use that description as the Rails app’s name.

If you can only think of one word to use, like `validations`, it usually means the thing you want to play with is too big. Break it apart into a few different apps, each testing one smaller idea.

Afterward, a quick `ls ~/Source/testapps` will give you a quick list of all the things you’ve explored. It’s your own personal reference guide to Rails!

Wrapping it up

So, when you’re done playing with your tiny Rails app, don’t delete it. Keep the app around to refer to later.

Picking good names for your tiny apps will help you find them when you need them. It’s hard to find out which of `test1`..`test57` is the one that taught you polymorphic associations. A two to three word name is best. It's short enough to scan, but long enough that each app is focused on one small idea.

Putting it all together

If you want to learn a new idea in Rails, you have to use it right away. If you follow the steps in this chapter, you'll build the kind of in-depth Rails knowledge that you've been hoping for:

1. When something you read looks interesting, or you have questions about it, generate a tiny Rails app using `rails new`. Use a subfolder for all of your test apps, and give them a good two or three word name focused on the idea you're exploring.
2. Build a simple example of the thing you're trying to learn inside your tiny app.
3. If you think it should be tested through your Rails app's UI, start up the server with `bin/rails server`, visit `http://localhost:3000` and try it out.
4. If it's something smaller, that you can play with without a browser, use `bin/rails console` to start up a console, where you can play with what you've built.
5. If setting it up in the console gets annoying, write a fake test for it, and run it with `bin/rake` or run the single test case.
6. If you want repeatable setup *and* a console, use Pry[12] inside your tests.

12. http://pryrepl.org

7. Brainstorm some questions you have about the idea you're exploring.
8. Discover the answers to those questions within your tiny Rails app by modifying the code you just wrote.

Chapter 2 How to build your own Rails app

You'll know it first by the weird, tingling feeling in your shoulders or legs. You'll get stressed out by the empty command line and skeleton Rails app on your screen, and have absolutely no idea where to start. It's just easier to hop onto Hacker News[13] and read a few articles instead. Inspiration has to come sometime, right?

This feeling is totally normal. Whenever I'm about to start a new Rails app, I *still* feel like I want to give up computers forever and run into the woods or something. But I have a process to share with you that will help you get past this, so you can turn your ideas into real, working apps.

Coder's block

An empty Rails app is frightening. It's our form of writer's block!

13. http://news.ycombinator.com

You might recognize where that fear is coming from. It's the same feeling you get when you hop onto reddit[14], see the ten new gems released that day, and worry about how you're going to learn all of those before the next versions come out and make everything you learned worthless.

You're just overwhelmed by the task in front of you.

And that's awesome, because there are some simple techniques you can use to fight this feeling of overwhelm.

A new Rails app is a huge, poorly-defined task. You have your dream of what you want your app to be floating in your head, and that makes it impossible to start. There's just too much to think about, and too much to build.

When you face a large, fuzzy, overwhelming task like this, the answer is *always* to break it up. Break your idea apart into small tasks that lead you to where you want to be. You'll have a path you can take that, no matter what, will get you closer to finishing your app. `

What to build first

Starting your app will help you past that first wave of stress. Right now, though, you don't have anything to start on. You just have a large, fuzzy idea that seemed easy to build when you thought about it. But now that you're about to build it for real, it feels impossible.

14. http://www.reddit.com/r/ruby

For example, say you decided to write an app to keep track of the bugs in the projects you work on. (Something like GitHub issues[15]). It's going to have milestones, and workflows, and tags, and it'll be the best bug tracker ever. But with all those features, where do you start?

You can't work on everything. So you have to work on *one thing*.

How do you choose the first thing to work on?

When you start a new project, try this short process to help decide which thing to work on first: Take a few minutes and think what you're trying to build. Write down every feature that comes to mind. Think of the different paths a user could take through your application, the different things they could do. Describe each one in a single sentence.

Then, focus on just the paths where, if you didn't have them, your app couldn't exist. Core paths.

Core paths are the things you'd talk about if someone asked you to describe your app in 30 seconds. If you're working on forum software, posting and replying to a topic would be core to the project. If you're working on a way to send notifications to an iPhone, registering a device and sending a notification to it would be a core path.

Core paths are good to start with, since you can play with your app without too much work in the beginning. You'll have a solid foundation to build new features off of. And that will make the next feature you work on even easier to start.

15. https://github.com/rails/rails/issues

If you can't decide on which core path to focus on first, it doesn't really matter which you choose. So pick one (at random, if you need to), and see how you like working on it.

A quick example

Let's break this down a little bit. To build a bug tracker, we might need to:

- Assign bugs to a user
- Assign bugs to a milestone
- Create bugs
- Update bugs
- View all bugs with a tag
- And lots more…

There are a lot of possible features.

But we can't really have a bug tracker without creating bug records. And that seems like something we could build without having to build much else in the system. That is, it stands alone. And a feature that stands alone is a good place to start. It's simpler.

So we'll start by creating bug records. But making a different decision is not a big deal, as long as it moves the app forward!

Writing a Rails app that can create a bug record is a lot less intimidating than writing a full clone of GitHub Issues. You might already know where to start building. But in case you don't, let's go into more detail. Some features will be harder to think about than others, and those need more planning.

Which part of the feature should you start with?

Once you're ready to build your first core path, you might feel like writing models for the objects you have in mind. Maybe you want to create migrations, add attributes, and connect everything together.

But soon, you'll have a ton of pieces that may not all fit. It'll take a long time to see how your app is used. And if you didn't design your models well, you'll just have to do that work over again when you build your UI and actually start playing with your app.

It's a lot easier to take the opposite approach. Why build something that nobody will ever see? If you *might* need a data model for a future feature, why build it before you build that feature?

When you write software, it's tempting to write code for consistency's sake, or just because you think you'll need it later.

But every line of code you write is an opportunity for bugs to sneak in. When you predict the future with your code and actually get it right, it's a real rush. But all those lines of code that haven't paid off yet are still sitting there. They're a magnet for bugs, because they're not being used. And you have to see and think about them whenever you run into that part of the codebase.

In general, less code is better code. And starting from the view and building toward the model is the best way I know to consistently write less code.

So, start with UI, and infer your data model from what you see in the UI.

There are lots of tools for making rough feature sketches, like OmniGraffle[16] and Balsamiq[17]. I've tried a lot of those tools, but I always come back to pen and paper. Rectangles, arrows, and text are usually enough to get started thinking about how people will use your feature. And that's all you're looking for at this point. You don't have to be a designer.

If you have to share your sketches with other people, pen and paper might not be that convenient. But even then, you might want to *start* on paper. And once you have a decent sketch, it's usually easy to move it into a tool that makes sharing easier.

When I sketch out a feature, I follow this process:

1. Take the small, core feature from earlier.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that person to be able to do that thing.
4. Describe the path through that action, as if you were telling someone what you were going to do.
5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

So, for example, here's this process applied to the bug tracker idea:

1. Take my small, core feature from earlier.

16. https://www.omnigroup.com/omnigraffle/
17. http://balsamiq.com

"Create a record of a bug."

2. Think of *one* simple thing someone could do with that feature.

What's the bare minimum someone needs to create a bug record? Let's say, title and description. Maybe we should be able to assign it to someone to work on. But could you have a useful bug tracker without an assignee? Well, maybe if it's just me using it for now. So let's go without it. Here's our action:

A user can create a record of a bug, with a title and description.

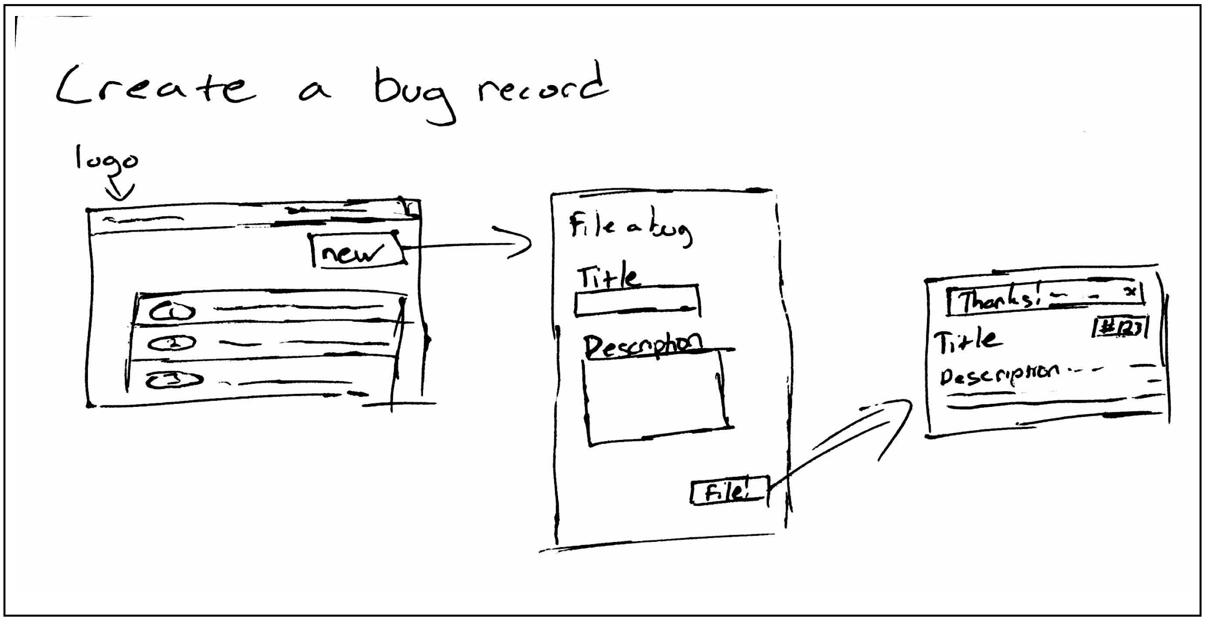
And maybe a closely related action that we could technically skip for now, but might get for free:

A user can see previously created bug records.

(Eventually, you'll naturally see extra features like this you could get without much extra effort. But for now, if you don't see these opportunities, it's totally OK. You'll get it with time and practice).

3. Draw *just enough* screens for that user to be able to do that thing.

This will prove without a doubt that I'm no artist. But I don't have to be! At this point, you're thinking about *interactions*, not visuals. So squiggly lines, boxes, and arrows are the way to go. Think of what your app would look like from 30 feet away.



4. Describe the path through that action, as if you were telling someone what you were going to do.

"Someone comes to this form. They can fill in the title and description, and hit the submit button. This takes them to a page where they can see the record of the bug they just entered."

5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

OK, let's think about what's in those sketches and what I just described. I see a bug record, with a title and description. And that's about it. Simple enough.

Obviously, this is a really basic feature, and just about exactly what Rails was designed to build. But the reason Rails was designed to build features like this is because a lot of web development is building features like this!

So if you look at your sketches and think it's too advanced, or if it'll take longer than a day to build, *think smaller*. Can you break that action down into a few steps that you can work on separately? Can you remove a little bit of functionality to make that action simpler? Can you start with a scaffold, and build the more complicated parts once you have the basics working?

Wrapping it up

It's hard to start Rails apps. And that's because until you start building them, the idea in your head is huge, vague, and complex.

So, break your app down into smaller features. Then, start with one simple core path. Once you start building your first feature, you'll create momentum for the next feature, and the next, and so on.

When you begin work on a single, small feature, start with the UI:

1. Take the small feature, or core path, from earlier.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.
4. Describe the path through that action, as if you were telling someone what you were going to do.
5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

Once you get that first little feature built, you'll be able to actually use your app. And that's when it starts getting really fun.

Exercises

This process doesn't come naturally. It takes practice. And luckily, you can practice this process almost anywhere. It starts with brainstorming features.

So, to begin, try coming up with some possible features for these app ideas. Try for 5-10 small features from each one:

- A youtube playlist downloader
- A TODO list
- A message board
- An appointment scheduler
- A Q&A forum

Then, pick a feature to start with, using the criteria I talked about: the feature you'd talk about if someone asked you to describe your app in 30 seconds. (If you're having trouble deciding, just pick a random one of your top few choices. Any decision is better than none).

For each feature you picked, follow the five-step process. Go as far as drawing actual sketches and planning which objects you'd need to start with. Don't worry if it seems really hard at first, just make some good enough decisions and keep going. (And that's "good enough" for you, not "good enough" for some imaginary expert online critic!)

Next, grab another feature and go through the process again, until you start to feel more comfortable with it.

If you're going to practice anything in this book, this should be it. Once you feel like you can start new Rails apps anytime, the rest of the process will come so much more easily to you.

Turning your ideas into running Rails apps

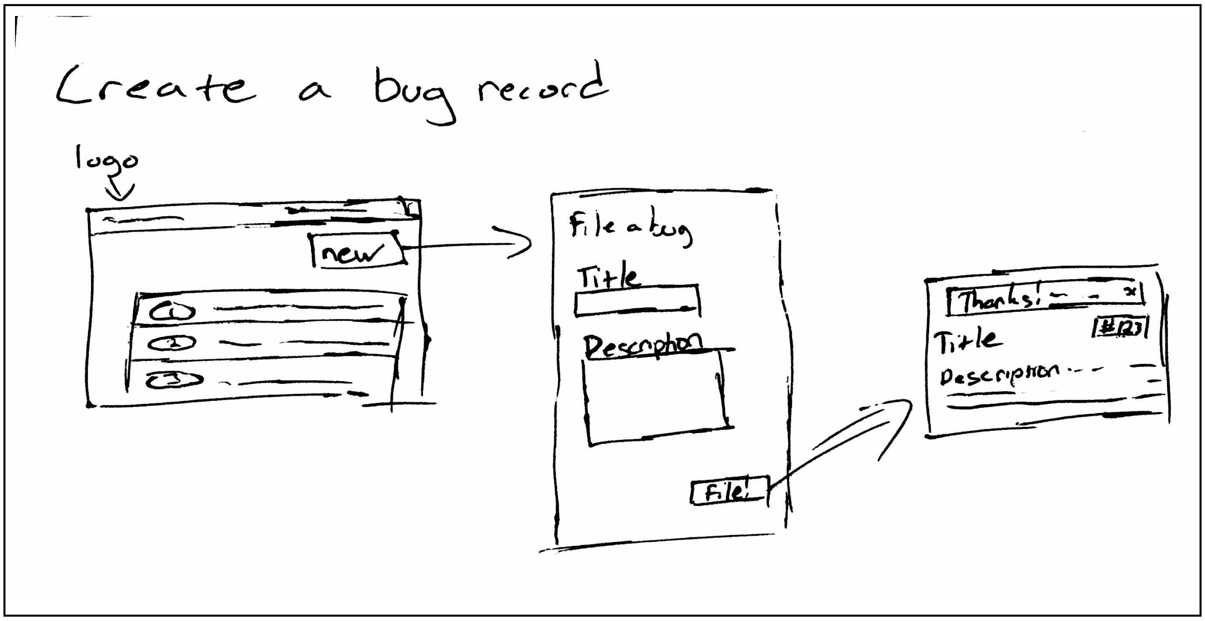
Now that you have a rough idea of what you're building, let's turn it into real code. And once again, we'll start from the UI and go down to the model from there.

Before you start writing code, you have to know which code to write. You can figure this out from the sketches you just drew.

The screens you drew correspond to views in your app, and the actions you take map to controller actions. So there are a few things your sketch can help you figure out:

- Which views do I need to write?
- Which controllers and controller actions do I need?
- What data does each controller need to manage?

This can be hard to do when you don't have a lot of practice with Rails. So let's walk through an example, using my sketches. Here it is again:



This path has three main screens, a few form elements, and a table. Starting from here, can we answer these questions?

- Which views do I need to write?

For the most part, a screen you sketched out represents a view. So, we need a view for the bug list, a view for the "add a bug record" form, and a view for showing a bug record. Editing isn't in our first sketch, so we won't think about that right now.

- Which controllers and controller actions do I need?

A single view will usually correspond to a single controller action. So, for our example, our "show a bug record" view will correspond to a "show" action on our controller.

And what's the controller? This can be a little trickier to discover. But you can usually figure out which controllers you need to build by looking at the actions you have and how they're related. If all of your actions work with the same kind of object, then the controller should be named after that object. In our example, that's pretty straightforward. All of our actions – show, new, and index – operate on a `Bug`, so we should have a Bug controller that all the actions can live on.

When you discover these names and relationships, it's a good idea to write them down, either on your diagram or on another sheet of paper. Trying to keep everything in your head is stressful. Better to capture it somewhere where you don't even have to think about it until you need it.

- What data does each controller need to manage?

Once you have your controllers, you can start thinking about what kind of data the controllers need. In our case, we have one controller, and three views. Two of the views operate on a single bug record. The bug record has an ID, a title, and a description. One object, three attributes.

The other view needs a list of bugs. Once we have a `Bug` model, though, a list of them should be pretty easy to get.

This is an iterative process

Associating sketches to objects is really hard at first. It takes some background knowledge, a little intuition, and some experience. You'll probably make some mistakes as you go through the process. I still do!

So instead of trying to get your answers to the three questions perfect, just come up with some answers that make sense to you. Use those answers to identify your data

Thinking in resources

Most of the time, there's an easier way to associate your controllers and views to your sketch.

Rails works well with "resources." You have a resource when you have a controller that has seven actions, all dealing with a single kind of thing: `index`, `new`, `create`, `show`, `edit`, `update`, and `destroy`.

Usually, your resource will be the model that's featured on a page. Ask, "What model is this view dealing with? Is it creating the model on this page?" In this example, we're dealing with the `Bug` model, which means we should have a `BugsController`, and the view is `app/views/bugs/new.html.erb`.

If you can figure out which resource your feature uses, you don't have to make as many decisions. Instead of having to think up different controller actions for every page, you can just ask, "Is this the `show` action for this resource? Is it the `edit` action?"

If you can't figure out what your resource is, don't worry about it for now. Later, look at what your controller actions all have in common. Do they all touch a single ActiveRecord class? Do they deal with a group of models that all belong to another model? Through those questions, you'll discover your resource.

models, which might become your resources. Then go back to your controllers, keeping those resources in mind.

Keep cycling through this process and refine your answers until you're happy with them. The more you do this, the more natural it will get.

Don't be too attached to your answers. If they never change, your first stab at it is taking too long.

Creating the Rails app

OK, now it's time to write some code! I know, it seems like it's taken forever to get here. But before this point, what would you have written?

Now, we have a plan: Some views, a model, and a controller. This puts us in great shape. Because this is *exactly* what Rails was designed for.

The closer you can get to actions related to a single resource, the more support you'll get from Rails. Since we've identified a resource for our "filing a bug" feature, Rails can help us a lot. And there's no better way to get help from Rails than with Rails generators:

```
bin/rails generate scaffold bug title:string description:text
```

Easy enough. Using scaffolds gives you a little more code than you need. They're useful for prototypes, but you'll probably throw away some of the code in real-world apps. In that case, you have two good options:

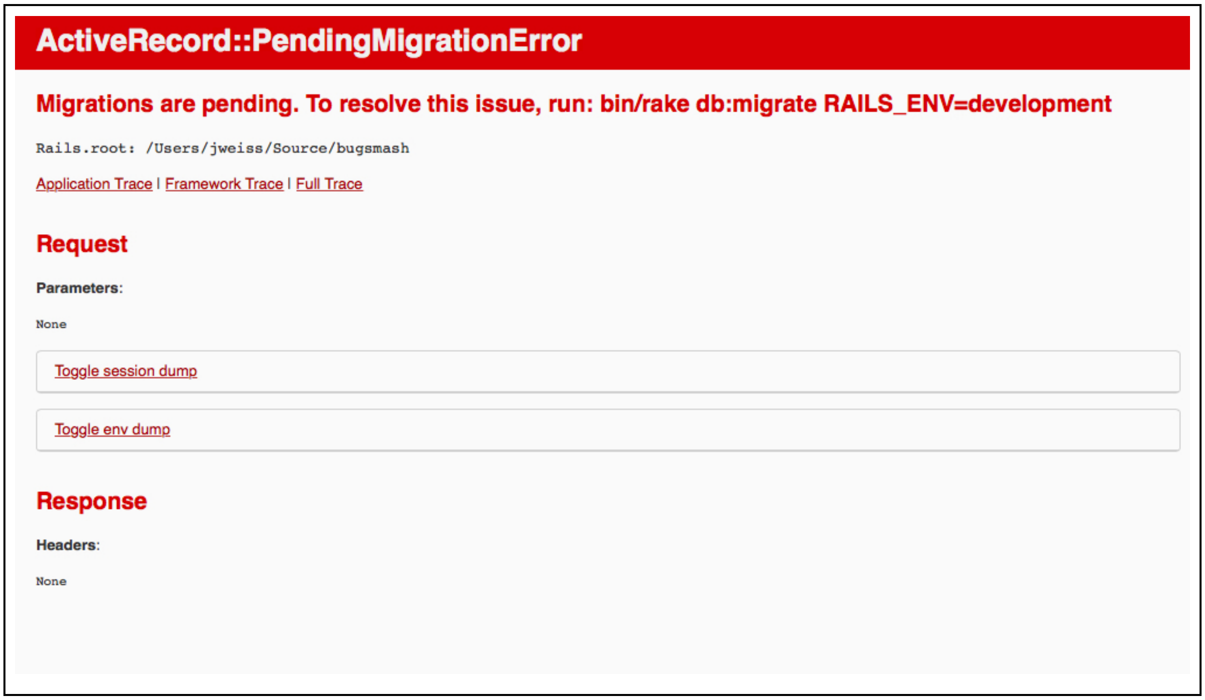
1. Generate the scaffold anyway, and just throw away the code you're not using.

2. Generate only the things you know you need. For me, these tend to be models, controllers, and integration tests, but not views.

For the first few features, it's really nice to have code already written that we can just tweak. So I typically just use full scaffolds for these.

Now, there's just some other tiny tasks we need to get our app up and running.

Since we generated a data model, we have to migrate our database. It's easy to tell when you forgot to do this, since you'll see this message when you start your server:



```
ActiveRecord::PendingMigrationError

Migrations are pending. To resolve this issue, run: bin/rake db:migrate RAILS_ENV=development

Rails.root: /Users/jweiss/Source/bugsmash

Application Trace | Framework Trace | Full Trace

Request

Parameters:

None

Toggle session dump

Toggle env dump

Response

Headers:

None
```

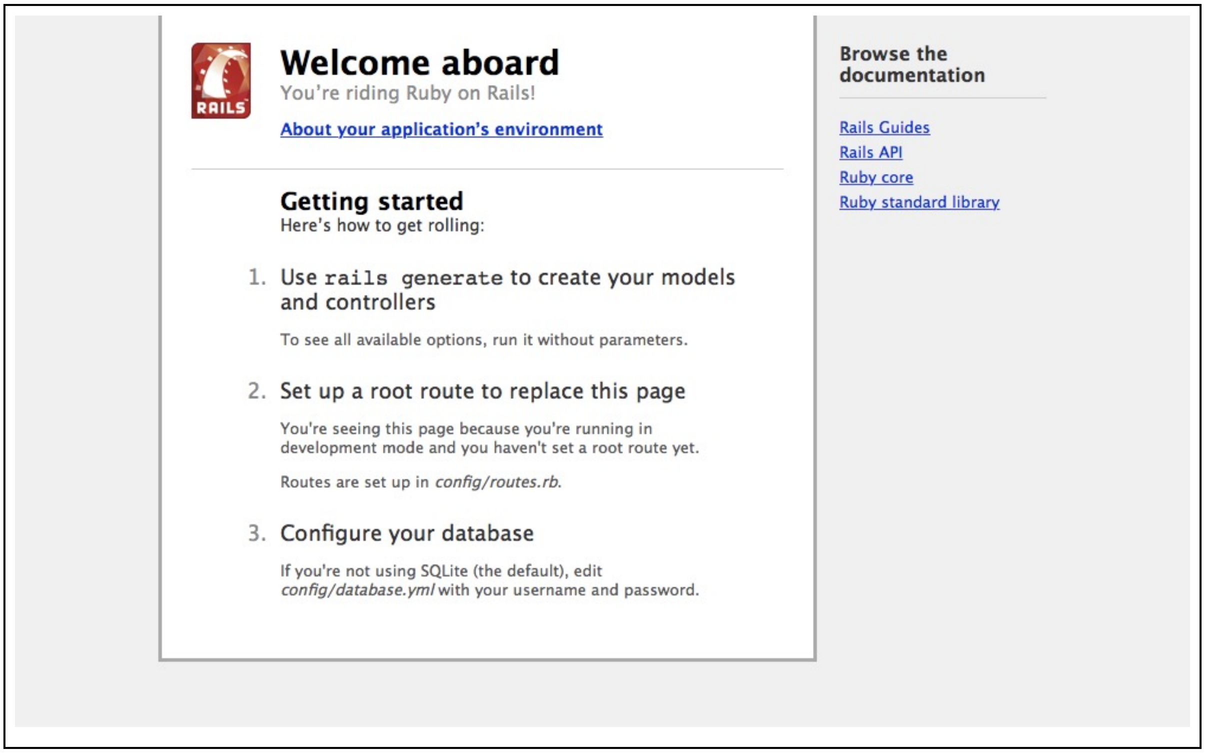
So, run this command to get your database working:

```
bin/rake db:migrate
```

OK, now we're almost good to go. If you run

```
bin/rails server
```

And visit your app at http://localhost:3000, you'll see the default Rails page. That's not what you want! At some point you'll have a real homepage. Early on, though, the `index` action of your most important resource is usually a good substitute. In our case, that's the `index` action of `bugs_controller.rb`.



Welcome aboard
You're riding Ruby on Rails!
About your application's environment

Getting started
Here's how to get rolling:

1. Use `rails generate` to create your models and controllers

   To see all available options, run it without parameters.

2. Set up a root route to replace this page

   You're seeing this page because you're running in development mode and you haven't set a root route yet.

   Routes are set up in *config/routes.rb*.

3. Configure your database

   If you're not using SQLite (the default), edit *config/database.yml* with your username and password.

Browse the documentation

Rails Guides
Rails API
Ruby core
Ruby standard library

The message Rails gives us here is less helpful. But step 2 at least points us to the file we need to change. So, let's hop into `config/routes.rb`, uncomment this line:

```
# root 'welcome#index'
```

and change it to[18]:

```
root 'bugs#index'
```

Now, run

```
bin/rails server
```

again, visit http://localhost:3000, and watch your new resource come up.

Play around with your new app! Try working through the features you came up with. Can you accomplish what you set out to do? How close is it to your sketch?

To be honest, it's not all that close. It doesn't look or feel as nice as you imagined it. But just like your original feature was a rough sketch on a piece of paper, this is a rough sketch with code. It may not look like much, yet, but you've accomplished a lot – you have a working app!

And just like you can refine a sketch to make it look professional, you can refine your app to make it feel professional. But it's a lot harder to go from "no code" to "a professional looking app" than it is to go from "no code" to "a rough sketch of code with

18. In case you're wondering, the part before the # refers to "bugs_controller" without the "_controller", and the second part refers to the action you want to call, which is `index`.

the right ideas” to “a professional looking app.” It’s so much harder that you won’t get much further than “no code” if you try to make it professional right away.

You’re much closer now than you were before. You beat your procrastination and got something started. So congratulate yourself and take a break.

Wrapping it up

Once you have a rough idea of where you’re going, you have to turn that idea into working code. But to do *that*, you have to know which code to write.

From your sketches, you can answer these questions:

- Which views do I need to write?
- Which controllers and controller actions do I need?
- What data does each controller need to manage? Views usually come from screens, models come from forms and the content on the page, and controllers come from collections of views related to the same model. If you start recognizing views and actions that are all related to the same object, you can think of that object as a resource. If you have resources, Rails can help you build your app with generators.

Generators aren’t always the right answer, but they can get you to working with actual code quickly, so you can iterate your way to a better solution. And once you build a quick sketch with code, future features become a lot easier to work on.

Exercises

Take three of the features you sketched out last chapter. Which views, controllers, models, and resources can you discover from your sketches?

Then, use the techniques in this chapter to build a tiny app that implements part of the feature you sketched out. It's OK if you get stuck. Just try to get familiar with that process of discovering the components you need to build, and turning your sketch into code.

Like the last exercises, this will take a few tries to get right. But every time you try to go through this process, you'll get better at it.

Which feature do you build next?

Now you have a real Rails app you can play with. But it's probably different from what you had in mind.

Maybe you want a better default font or style for the site.

Maybe you want the forms to be more interactive.

Maybe you need some images.

Maybe you need to validate your form input, or hook some models together.

There's still lots to do. And again, when you look at all these tasks as a big collection, it feels like you're never going to finish. But what if you followed the ideas from the last chapter, and just focused on one for now? You could commit and ship each task separately, and they'd each give you a little bit of value. Write the rest of them down, and save them for later.

Keep these tasks as small as possible. Each should be the kind of thing you could research and hack on for a little bit without having to learn everything about everything.

T-Shaped development

When I build apps, I practice **T-Shaped Development**: Getting a basic app working, and polishing one bit of it at a time.

Why T-Shaped?

I got the idea of "T-Shaped" from Valve's Employee Handbook (PDF)[19]:

> ***We value "T-shaped" people.*** *That is, people who are both generalists (highly skilled at a broad set of valuable things – the top of the T) and also experts (among the best in their field within a narrow discipline – the vertical leg of the T).*

That mental picture of a broad base with deep focus in specific areas stuck with me. And even though it doesn't perfectly apply here, it helps to imagine new features this way.

19. http://www.valvesoftware.com/company/Valve_Handbook_LowRes.pdf

So, when you build something, try to get something rough up as quickly as possible. As long as the core of the feature you want to build is there, it's fine.

From that core idea and your sketches from earlier, a lot of small, more polished tasks will naturally appear. Go deeper with each of these small tasks.

This way, you don't have to face the big, fuzzy, "build this page and make it look right" task that you started with. Instead, you have smaller, more focused tasks you can take one at a time.

And those smaller tasks also trigger just-in-time learning.

Just-in-time learning

Early on, there will be a lot of things that you don't know enough about. Maybe you want to add a layout to the page, but don't know how to do it. Or maybe you're having trouble figuring out how to find the right association options when the defaults don't work.

This is a good thing. It means you're not trying to learn everything at the beginning. That's my favorite way to procrastinate when I start something new!

When you wait to learn something until you need it:

- You can learn *current* best practices for the thing you want to do, instead of learning whatever the best practices were when you learned the rest of Rails.
- You have more motivation to learn it. Because if you don't, you can't finish your app.

- You have built-in examples in your own app.
- You can use the thing you learned right away. Which is the best way to make that learning stick.

Start off knowing a little about a lot. You only need to know enough to pick up the next thing you want to learn. Then, when you have a topic you need to study more closely, read *a ton* on it, until you've internalized the best practices, edge cases, and tricks.

When you learn more than the average developer, on *just the things* you'll be using all the time, it'll really pay off.

A quick break

How are you feeling about what you've done so far? How far did you get in the exercises? Which parts were hard for you?

While you're working, take some breaks to ask yourself those kind of questions. Things you're having trouble with are the things you should isolate, study, and practice. You'll get a lot more out of practicing the things you *don't* know well than the things you do.

A more complicated example

Once you finish your first feature, it's time to look at another.

For the first feature of our project, we focused on the core concept of the product. After you finish that first feature, you may have already built that core. So where do the next features come from?

- Ask some questions: What is the app missing? What did you postpone to get that first feature done?
- Try using what you have so far. What would make your life easier if it was built? What annoys you while you're using your app? What could you automate?
- If you're building this app for someone else, watch how they use the project. Where are they struggling? What do they complain about?

In the bug tracker, there's a missing feature that jumps out immediately: Assigning someone to investigate and fix a tracked bug.

Breaking apart fuzzy features

So, the next feature is "Assigning a user to fix a tracked bug." How would you build it? What does it do? Which parts of the system does it touch? Depending on how experienced you are, it might seem easy to build. Or it might seem so far beyond your skill level that you just want to hide under your desk until someone else builds it.

Again, when you feel this way, it means you haven't broken the task up enough. So let's think about the task, using a process like the one you used for the first feature:

1. Take the small feature from earlier.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.

4. Describe the path through that action, as if you were telling someone what you were going to do.
5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

Sometimes, you can use some of the stuff you've already built for the second feature, so this could go even more quickly than it did the first time.

For the feature we just identified:

1. Take the small feature from earlier.

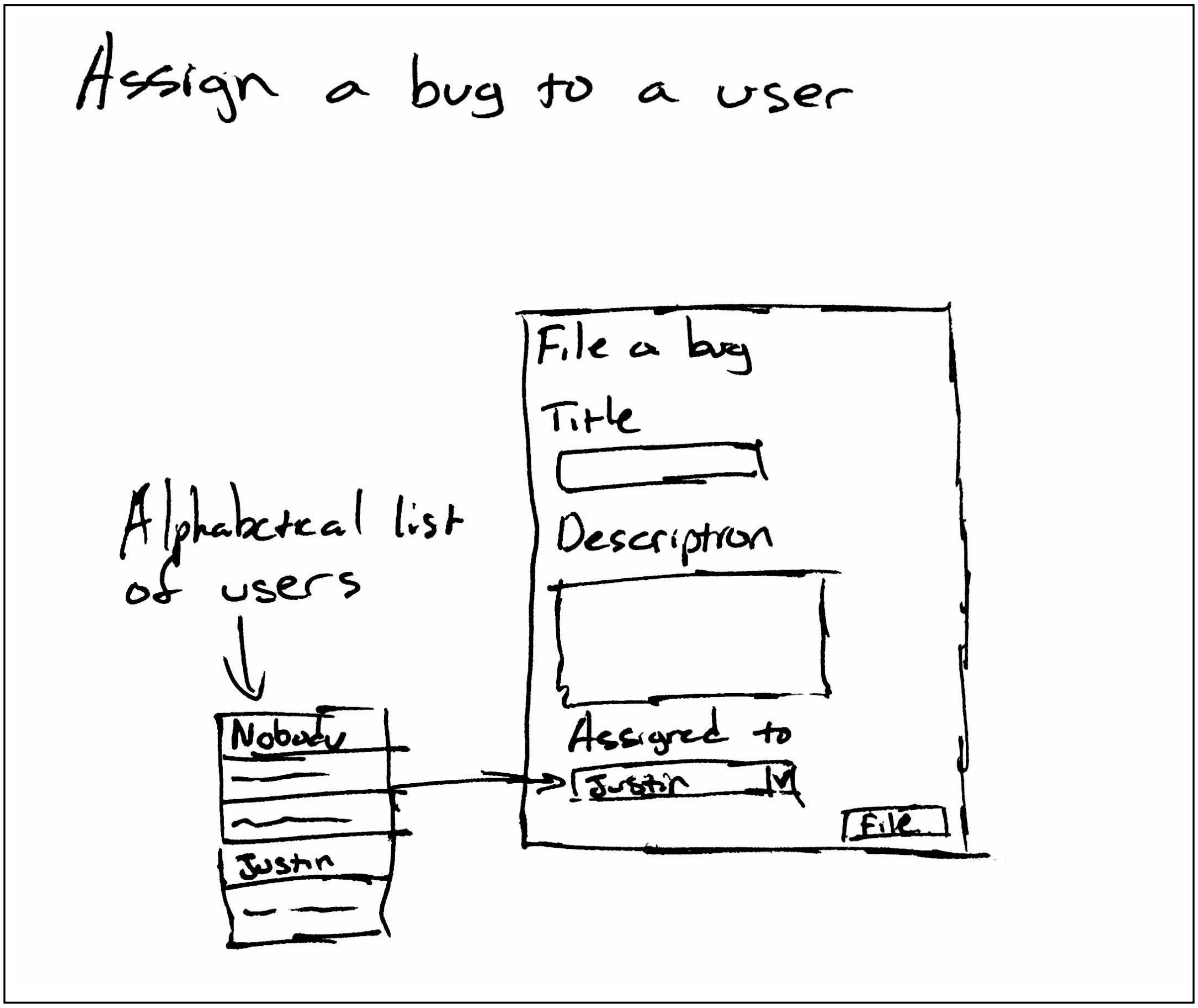
"I can assign a user to a bug record."

2. Think of *one* simple thing someone could do with that feature.

Assign a user to a bug record. But wait! If we can assign a bug to a user, those users need to come from somewhere. But does that need to be built right now? Can we fake the data until we actually need it? Maybe. For now, let's focus on the "assigning" part, and create some users by hand using the Rails console or `db/seeds.rb`[20].

3. Draw *just enough* screens for that user to be able to do that thing.

20. http://guides.rubyonrails.org/active_record_migrations.html#migrations-and-seed-data



4. Describe the path through that action, as if you were telling someone what you were going to do.

While creating a bug record, the person creating it will see a list of users on the system in a drop-down. They can pick a user from that list, and the bug will be assigned to that user.

5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

So, we have a User object, that at least needs a name. And that's it!

During this process, you'll identify a *ton* of features you're not going to build right now. Here, we have some stuff like finding a list of bugs by user, logging in, and signing up. That's fine! Write them down somewhere, and you can get to them later.

Baseline, polish

Once again, you create the simplest thing that slightly resembles what you want. Once you do that, identify places that need extra work. Then, break those apart into smaller tasks, and knock them out one by one.

What you consider "good enough" will change depending on your level of experience. As you master things like testing, TDD, HTML, CSS, Bootstrap, and ActiveRecord, you'll be able to produce a higher quality base from the beginning.

But if you haven't yet mastered those things, don't take an app from nothing to professional in one step. It'll take a lot out of you and set you up for failure.

Instead, create checkpoints after every bit of value you add to your app. Take a break, dump everything you're not using out of your working memory, and prepare for the next small task.

Wrapping it up

Once you build a Minimum Viable Feature, you'll notice a lot of improvements you want to make right away. But you shouldn't take them on all at once. Instead, record them, and work on one at a time. Pick them based on how important or interesting they are to you.

When you start from basic functionality and polish one piece at a time, you can build a great app in many small steps. Building an app in small iterations is great for learning, because it gives you an excuse and built-in motivation to learn a new topic.

When you're ready to build your second feature, you'll either know what you want to build, or you won't. If you don't, these techniques will help you draw out some ideas:

- Ask some questions: What is the app missing? What did you postpone to get that first feature done?
- Try using what you have so far. What would make your life easier if it was built? What annoys you while you're using your app? What can you automate?
- If you're building this app for someone else, watch how they use the project. Where are they struggling? What do they complain about?

Then, use the process from your first feature to begin sketching out the second:

1. Take your small feature.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.

4. Describe the path through that action, as if you were telling someone what you were going to do.
5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

Exercises

Consider your tiny app from the last chapter. What's missing? What do you want to build next?

Can you come up with five ways to polish your first feature, and one major new feature to build next?

Once you do, sketch out the second feature using our feature development process, and build it into your tiny app. What did you need to learn in the process? Were you able to find that information?

Putting it all together

If you follow these steps, you will get through those first few instances of coder's block. And you'll finally get something built.

The key is avoiding large, fuzzy tasks. At every stage, you should be trying to break large tasks apart into smaller ones.

Specifically, when you're starting a new project, focus on building an important path through your app as your first feature. Build the thing you'd talk about if someone asked you to describe your app in 30 seconds.

Then, build from the UI down, using this process:

1. Take the small feature from earlier.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.
4. Describe the path through that action, as if you were telling someone what you were going to do.
5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

Once you've followed those steps, use your sketches to answer these questions:

- Which views do I need to write?
- Which controllers and controller actions do I need?
- What data does each controller need to manage?

Views usually come from screens, models come from forms and the content on the page, and controllers come from collections of views related to the same model. If you can identify resources, it's even easier.

For the first few features, use Rails generators and scaffolds to get your app up quickly. The sooner you can play with your app, the sooner you can figure out what's working and what's not. And you don't want to put a lot of investment into your app until you're sure it's the right thing to build.

Once you get that first feature done, it starts getting fun! But there's still lots of work to do. So, polish the first few things you've built, one task at a time.

When you're ready to move onto your next feature, use that same five-step sketching process to plan it out. If you're having trouble coming up with features:

- Ask some questions: What is the app missing? What did you postpone to get that first feature done?
- Try using what you have so far. What would make your life easier if it was built? What annoys you while you're using your app? What can you automate?
- If you're building this app for someone else, watch how they use the project. Where are they struggling? What do they complain about?

It might seem like a long, time-consuming process. But as you get more experienced, it'll go faster. And you'll always have this process to fall back on when you get stuck.

> *"Nobody tells this to people who are beginners, I wish someone told me. All of us who do creative work, we get into it because we have good taste. But there is this gap. For the first couple years you make stuff, it's just not that good. It's trying to be good, it has potential, but it's not. But your taste, the thing that got you into the game, is still killer. And your taste is why your work disappoints you. A lot of people never get past this phase, they quit. Most people I know who do interesting, creative work went through years of this. We know our work doesn't have this special thing that we want it to have. We all go through this. And if you are just starting out or you are still in this phase, you gotta know its normal and the most important thing you can do is do a lot of work. Put yourself on a deadline so that every week you will finish one story. It is only by going through a volume*

> *of work that you will close that gap, and your work will be as good as your ambitions. And I took longer to figure out how to do this than anyone I've ever met. It's gonna take awhile. It's normal to take awhile. You've just gotta fight your way through."*
>
> *– Ira Glass*[21]

This quote is one of my favorites. It's *so* accurate. So, when you get blocked, keep it in mind. Let your taste drag your work to where you want it to be.

And don't give up, because if you keep working, you will get there.

21. https://www.youtube.com/watch?v=BI23U7U2aUY

Chapter 3
Testing your code efficiently and effectively

A lot of the developers that came to Ruby early, arrived through (or wrote) books like The Pragmatic Programmer[22] and philosophies like Extreme Programming[23]. These developers tried to push automated testing in languages like Java, but Ruby is where they created a culture of testing from the beginning. So as long as I've been working in Ruby, testing has been a big deal.

This culture has been great to have, because it leads to better, more reliable code. When you're *expected* to write tests, you write more tests.

On the other hand, you'd be surprised to know just how much useful code is hidden away (or not written at all!) because the authors are too embarrassed to release them

22. https://pragprog.com/book/tpp/the-pragmatic-programmer
23. https://en.wikipedia.org/wiki/Extreme_programming

without tests. What if the whole world knew that they didn't test-drive all of their code? This is the dark side of this testing culture. And it's a shame that it happens.

Testing doesn't have to be intimidating. It turns out that with a few tips about how to write and organize your tests, testing your code isn't so much different than writing your code. So as you build up your coding skills, you can improve your testing skills at the same time.

How to write great tests quickly

Your tests are most efficient when you have the right amount of tests, and they test the right code. Not too many, not too few. Of course, it's not that easy. It'll take some practice before you'll know how and what to test. But I have a few tips that'll get you started.

What do you test, and what can you skip?

If you're using Test-Driven Development, you don't really have a choice what to test: you're testing the next thing you're going to build.

But if you're writing tests *after* you write your code, you'll have a little more work ahead of you. You'll have to decide what to test, and what you can skip.

When I'm testing something new, my tests fall into a few categories:

1. Happy path tests.

Assume that everything worked. What should have happened? What should the system look like afterward? For most features, I'll usually only have one or two of these.

2. Sad path tests.

   There are a lot more ways to fail than to succeed, and you should test a good sample of them. Same thing: what should have happened? Did it happen? Because there are more ways to fail, I'll normally test more sad paths than happy paths.

3. What if? tests.

   What happens to your code in strange situations? For example, what if you're using objects as keys to a hash, and you modify one of those objects out from under the hash? If your code is processing some data, but the data isn't formatted right, should your app fix it, or ignore it?

   These might not be happy path *or* sad path tests. They're there to satisfy your own curiosity, or to lock down your app's behavior in weird situations.

   You won't always have "What if?" tests. But when you have them, they can be an important way to document how your app works.

Your tests will be most efficient if you focus on the places where happy path tests turn into sad path tests. These places are called edge cases[24], or boundaries.

24. https://en.wikipedia.org/wiki/Edge_case

For example, if you're testing that you can't save a bug with a name longer than 100 characters, you should test at 100 and 101 characters. If you have tests for those two situations, there's not much point in *also* testing 150, or 5, or 1000 characters.

When aren't you testing enough?

If you run into a bug in your app that your tests didn't catch, it means you're probably missing a test. When you discover a bug:

1. Write a test that fails while the bug exists.
2. Fix the bug.
3. Make sure the test passes now.
4. Check in both your test and fix, so you don't run into the problem again.

If you're having trouble writing a test that only fails if the bug exists, you can try flipping it around. Write a test as if your bug is supposed to happen. Then, reverse the assertions.

For example, if you were able to save a bug without a description, but the description should have been required, you could write a test like this:

```
test "can create a bug record without a description" do
  bug = Bug.new(title: "My new bug record",
    description: "")
  assert bug.save, "Bug couldn't be saved:
      #{bug.errors.full_messages}"
end
```

This test should pass. Then, reverse the assertion:

```
  test "cannot create a bug record without a description" do
    bug = Bug.new(title: "My new bug record",
      description: "")
    refute bug.save, "Bug should not have been saved."
  end
```

And this test should fail, but only because you haven't fixed the bug yet. As soon as you fix the bug, the test should pass. Remember to fix the test name, too! And once you fix the bug, you can tweak your assertions to be a little better. For this example, testing that the right error messages were added to the bug record would be better than just testing that it couldn't be saved.

These tests are useful as "regression tests," and will catch you if you break something the same way later on. Regression tests are some of the most useful tests I have. If I broke something once, it turns out I'm pretty likely to break it again.

How to fight tester's block

Have you ever stared at an empty test case and had no idea what to write?

This is the same problem from Chapter 2! And it has the same solution. You can *add testing* to your feature development process. And just as your process helped you write code, it can also help you write your tests.

So, let's go back to that original example. I have a bug tracker, and the first thing I want to do is create bug records. We're still in the early stages, so there's no users, or milestones, or anything. All our bug record has is a title and description.

The steps were:

1. Take your small feature.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.
4. Describe the path through that action, as if you were telling someone what you were going to do.
5. As you describe that path, write out the objects, properties of those objects, and other actions you think you need to develop that path.

Now, we're going to replace steps 4 and 5 with tests:

1. Take your small feature.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.
4. Translate that sketch into a failing integration test.
5. Get the next part of that test to pass, by:
6. Writing a failing controller test.
7. Writing some failing unit tests.
8. Writing enough code to get the unit and controller tests to pass.
9. Repeat steps 5-8 until your integration test passes.

(If you're not using TDD, you can leave steps 6 and 7 until after 8. That is, you can write your controller and unit tests *after* your integration test passes. It's not a big deal either way right now).

So, once you've drawn your sketch, translate your sketch into an integration test as closely as possible.

I like Minitest and Rails' testing classes. But when I write integration tests, I like Capybara[25] even more. Capybara gives you tests that sound a lot like a description of a feature:

test/integration/create_bugs_test.rb

```
require 'test_helper'

class CreateBugsTest < CapybaraTestCase
  test "can create a new bug" do
    visit bugs_path
    click_link "File new bug"
    assert_equal current_path, new_bug_path
    fill_in "Title", with: "My new bug"
    fill_in "Description", with: "Short description."
    click_button "File bug"
    # ...and so on...
  end
end
```

This is what an integration test could look like. Can you see where it matches the sketch? The test reads exactly how you'd describe the feature to someone else over the phone: "First, you visit the bugs page. Then, you click 'File new bug.' Are you on the new bug form? Good. Fill in the title and description and click 'File bug', and then..."

25. https://github.com/jnicklas/capybara

I love starting with integration tests. It's like a list of instructions telling me how to build the feature.

Next, you'll get the first part of this test to pass. In order to do this, you need to build the page, which naturally leads you to the controller tests.

Actually, the first part of this specific integration test is boring, so let's pretend you already did that part. Instead, we'll focus on getting the second part of the integration test to pass – the `create`, instead of the `index`.

So the next step is to write a failing controller test. Which page, which controller action do you need to build to get this integration test to pass?

Here, it's `create`. So you'll write a controller action for that.

test/controllers/bugs_controller_test.rb

```
  test "should create a bug record" do
    assert_difference("Bug.count", 1) do
      post(:create, bug: {
        title: "My new bug record",
        description: "Short description."
      })
    end
    assert_redirected_to bug_path(assigns(:bug))
  end
```

While you write these controller tests, think about what kind of models and model methods you'd need to get these tests to pass.

Then, leave this failing for now, and write some failing unit tests. This is where you can go a little crazy. Test edge cases, happy paths, sad paths, and whatever else you think of.

This feels a little like brainstorming. You're just getting possible tests out of your head and into code:

test/models/bug_test.rb

```
test "can create a bug record" do
  bug = Bug.new(title: "My new bug record",
    description: "This bug record was just filed by a test.")
  assert bug.save, "Bug couldn't be saved:
       #{bug.errors.full_messages}"
end
test "should not create a bug without a description" do
  skip
end
test "should not create a bug without a title" do
  skip
end
test "should not create a bug with too long of a title" do
  skip
end
# ... and so on ...
```

Now, you can get your tests to pass by adding the code you've been putting off writing until now. Get your model tests to pass, then your controller tests. And once you're done with that, your integration test will be one step closer to passing.

Then, you'll move onto the next part of your integration test, and start step 5 all over again.

After you work through this process a few times, it'll go pretty fast. It starts to become habit. But it helps a lot, because every step leads naturally into the next. No step is all that big or overwhelming, and that keeps you motivated, in flow, and working.

Wrapping it up

Your Rails app should have just enough tests to test your code well. You'll get a better idea of what that means as you write more apps and tests.

When you add some steps to your feature development process from Chapter 2, your feature and your tests will be easier to write, and you won't get stuck as easily. The revised process looks like this:

1. Take your small feature.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.
4. Translate that sketch into a failing integration test.
5. Get the next part of that test to pass, by:
6. Writing a failing controller test.
7. Writing some failing unit tests.
8. Writing enough code to get the unit and controller tests to pass.
9. Repeat steps 5-8 until your integration test passes.

How many controller and unit tests do you need? There's really no number that works for everything. But you should have at least a few happy path and sad path tests. I tend to have more sad path tests than happy path tests.

You might also have some "What if?" tests. Those tell you and your future co-developers what's *supposed* to happen in a crazy situation.

No matter how good your tests are, you'll still run into bugs. Bugs imply that you missed a test somewhere. Otherwise, you would have caught it! It's OK, though. Write a regression test, to make sure the bug gets caught next time:

1. Write a test that fails while the bug exists.
2. Fix the bug.
3. Make sure the test passes now.
4. Check in both your test and fix, so you don't run into the problem again.

Testing is a huge topic, and deserves its own book(s). But these tips should get you started, so you can practice writing tests. With the extra steps in your feature development process, you'll write a lot more tests, and you'll get better at it. And as a bonus, you'll have an easier time building new features.

Exercises

Take a look at the list of methods in Capybara[26]. Using just those methods and Ruby, could you write short descriptions of how someone would walk through the two features you wrote in Chapter 2?

26. http://rubydoc.info/github/jnicklas/capybara/master#The_DSL

Better ways to structure your tests

When I write Rails tests, I usually use Minitest instead of RSpec, and I think you should try Minitest, too. Minitest fits my mind better for a few reasons:

1. Since Rails uses it for its own tests, and it's bundled with Rails, it tends to work correctly out-of-the-box with Rails.

2. Minitest test suites are just classes. Tests are just methods. These are concepts that you'll use all the time in Ruby or other object-oriented languages.

   This means you won't have to learn a separate language to write your tests, and you can use Ruby modules[27], methods, and inheritance to organize your tests.

Writing better tests with the three-phase pattern

So, how do you write Minitest tests?

Rails generators do a lot of the setup for you:

```
require 'test_helper'
class BugTest < ActiveSupport::TestCase
  # test "the truth" do
  #   assert true
  # end
end
```

27. http://www.ruby-doc.org/core-2.2.0/Module.html

You uncomment it, come up with a name, and you're ready to write your test, right? But then what? What do you write first? What should your test code look like?

With the three-phase test pattern, you'll turn those stubbed out lines of code into clear, well-structured test cases.

The three-phase test pattern

Your test cases should work in three phases[28]:

1. First, you set some stuff up ("Arrange")
2. Then, you do something ("Act")
3. Then, you make sure that what you expected to happen, actually happened. ("Assert")

For instance, imagine you were testing a method on an array in Ruby. Following this pattern, your test could look like:

```
test "Array#sort will sort an array of numbers" do
  # arrange
  unsorted_array = [7, 4, 2, 3]

  # act
  sorted_array = unsorted_array.sort
  # assert
  assert_equal [2, 3, 4, 7], sorted_array
end
```

28. http://www.c2.com/cgi/wiki?ArrangeActAssert

Simple enough. But every part of the test has a place to go, and each stage of the test almost tells you how to write it.

Sometimes, you won't need an Arrange phase, or the Act and Assert phases will be combined. But it still helps to think about all three phases as you write your tests.

The Assert phase gotcha

There's a trick to the Assert phase: you shouldn't use the same logic that the Act phase used in the Assert phase. You should always take two paths to the same answer. Otherwise, you won't notice bugs in the code you're calling, because it's just getting called again in the Assert phase.

For example, if you're doing some math:

```
test "average returns the average of a set of numbers" do
  # arrange
  numbers = [1, 2, 3, 4]

  # act
  average = numbers.average
  # assert

  # this is bad
  assert_equal [1, 2, 3, 4].average, average
  # this is better
  assert_equal 2.5, average
end
```

Calling `[1, 2, 3, 4].average` again in the Assert phase is bad, because `average` could return almost *anything* and that assertion would still pass.

Here, that's pretty clear. But even when things get more complicated, make sure you're not just running the same code twice. Otherwise you're only verifying that your method was *called*, not that it worked the way you expect it to.

Usually, the easiest way to take a second path to the answer is to find the answer by hand and hardcode it. It can be brittle, but it's better than your tests breaking without you realizing it.

Why three phases?

If you split your tests into those three phases, you have simpler questions to answer. Instead of "How should I write this test?", you can focus on each phase: "How should I set this test up?", "What am I testing?", "What should the answer look like?"

These questions still might not have easy answers, but the answers will be a lot easier than thinking about the entire test at once. And if you're lucky, you can even share phases between related tests, making your next test much less painful to write.

How to use object-oriented design to make your tests better

As your test suite grows, you'll repeat code in a few different tests.

With Minitest, all of your test suites are just classes. That means you can use your object oriented design skills to reorganize your tests. You'll learn more object oriented design as you learn Rails. But there are a few ways I often reorganize my tests:

Writing custom assertions with Extract Method[29]

There's nothing special about Minitest assertions, they're just Ruby methods. Look at the definition for `assert_equal`, for instance:

```ruby
def assert_equal exp, act, msg = nil
  msg = message(msg, E) { diff exp, act }
  assert exp == act, msg
end
```

Other than that `msg = message` line, it's pretty much exactly what you or I would have written. (That first line is just a helper for writing failure messages).

So, if you have an assertion that's used in a few different places:

```ruby
test "empty draft post is valid" do
  @post = Post.new(draft: true)
  assert @post.valid?, "Post is invalid: #{post.errors}"
end
```

you can extract the assertion into its own method:

test/test_helper.rb

```ruby
def assert_valid(obj)
  assert obj.valid?, "#{obj.class} is invalid: #{obj.errors}"
end
```

29. These capitalized phrases are names of common ways to reorganize your code. You can see a gigantic list of them at http://refactoring.com/catalog/.

```
  test "empty draft post is valid" do
    @post = Post.new(draft: true)
    assert_valid @post
  end
```

`assert_valid` is not only shorter to type, it's easier to understand.

Better mocks (even though you shouldn't)

Minitest makes mocks and stubs a little ugly to use. This is intentional – Minitest wants to discourage you from overusing them. Still, sometimes mocks and stubs are good ideas. And once again, you can use Extract Method to pull ugly mock code into a method that's easier to understand.

For example, for an app of mine[30], I wanted to mock some network communication with DGS[31], a server for playing Go that my app depends on. The mock code was ugly:

30. https://github.com/justinweiss/dgs_push_server
31. http://www.dragongoserver.net

```
test "When a player is queued, notifications are sent" do
  response = game_csv(1)
  dgs = MiniTest::Mock.new
  DGS::ConnectionPool.stub(:checkout, dgs) do
    DGS::ConnectionPool.stub(:checkin, nil) do
      dgs.expect(:get, response, [
        session, "/quick_status.php?version=2"])
      assert_difference "Notification.count", 1 do
        worker_class.new.perform(players(:justin).id)
      end
    end
  end
  dgs.verify
end
```

That is truly awful. What part of it is actually doing the work? And this is only one of about five places I used this exact same mock!

All that mocking stuff, though, can be pulled out into its own method:

test/test_helper.rb

```
def mock_dgs_with_response(response)
  dgs = MiniTest::Mock.new
  DGS::ConnectionPool.stub(:checkout, dgs) do
    DGS::ConnectionPool.stub(:checkin, nil) do
      dgs.expect(:get, response, [
        session, "/quick_status.php?version=2"])
      yield
    end
  end
  dgs.verify
end
```

Then, the test looks like:

```
mock_dgs_with_response(game_csv(1)) do
  assert_difference "Notification.count", 1 do
    worker_class.new.perform(players(:justin).id)
  end
end
```

Now, the test is a lot clearer, it's obvious what it's testing, and the setup code doesn't overpower the work being done.

Pulling tests up to a superclass (or module)

Sometimes, you'll have a few classes that all act in the same way. Maybe they have a common base class, maybe they include the same modules, or maybe they're just duck-typed[32].

For example, say we have a bunch of taggable objects.

You could test just the taggable functionality by including it into an empty class:

32. When a class doesn't have any real relationship to another class, other than responding to some of the same methods, they're duck-typed. For example, `String` and `Array` both have a `length` method. So if you only care about the length of something, you can consider these two classes duck-typed.

```
class EmptyTaggable
  include Taggable
end
class TaggableTest < ActiveSupport::TestCase
  setup do
    @object = EmptyTaggable.new
  end

  # Lots of taggable tests on EmptyTaggable...
end
```

Or you could test the taggable functions on just one or a few of the classes that use them. This seems like it would *probably* be ok.

But I'm surprised by how often the *integration* between a module and the class it's included in creates problems. And you're never going to find those problems without testing that module in every class that needs it.

You could copy and paste the taggable tests into every taggable class, but that seems like a bad idea. Instead, you can pull those test cases into their own module:

```
module TaggableTestCases
  extend ActiveSupport::Testing::Declarative
  test "A taggable object can be tagged" do
    taggable_object.tags = ["First", "Second"]
  end
  private
  def taggable_object
    raise NotImplementedError,
      "Override me in your test class!"
  end
end
```

and include it into each of your actual test classes:

test/models/post_test.rb

```
class PostTest < ActiveSupport::TestCase
  include TaggableTestCases

  ...
  private
  def taggable_object
    Post.new
  end
end
```

And that way, your tests will actually match the real world your code runs in.

So, there are lots of ways to apply object oriented principles to your tests. But there are a few guidelines to follow:

1. For tests, clarity is better than cleverness. You don't have anything testing your tests, so you have to be careful not to make things too abstract. Hardcoding values, copy and pasted code, all that kind of stuff is usually OK in tests, if they make the test easier to understand.

2. Organizing your tests is easiest if your test organization matches your code organization. `app/models` should be tested by `test/models`. `app/presenters` should be tested by `test/presenters`. If you have a model that includes a module, maybe its test should include a module with tests that test that module (like my third example above). If your objects all inherit from a parent, maybe your tests should inherit from a parent test class. But still, keep guideline #1 in mind as you organize your tests.

3. Wait until you feel real pain before reorganizing your tests. Because nothing is testing your tests, you should have a higher bar for refactoring your tests than you do refactoring your code.

Wrapping it up

If you aren't already settled on RSpec, you should give Minitest a try. It's easy to understand, it's small, and you can practice your object-oriented design skills while you write your tests.

When you start a new test, you should think of it in three phases:

1. Arrange
2. Act
3. Assert

Sometimes, you won't need an Arrange phase, or the Act and Assert phases will be combined. But thinking in terms of those smaller steps can be a lot easier than thinking about the test as a whole. And remember, when you verify your code, take a different path to the answer in your test than you took in the code!

As you get better at object-oriented design, you can use the skills you build to reorganize your tests. You can apply most of the design skills you learn to your tests, but some of my favorites are:

- Using Extract Method to write custom assertions.
- Using Extract Method to make mocks easier to write.
- Using modules to share tests between test suites.

Your test code should be simpler than the code it's testing. But sometimes, you'll feel enough pain in your tests that reorganizing them is still worth doing.

Exercises

Look at the test code for Rails' hash extensions at https://github.com/rails/rails/blob/master/activesupport/test/core_ext/hash_ext_test.rb. Which of the three phases can you find in those tests? Can you find places where some of the phases are combined?

Which patterns does the Rails team use to organize their tests? Could you use any of those patterns in your own tests?

What's the best way to learn Test-Driven Development?

If you don't have a lot of experience testing your code, TDD can be incredibly hard to learn. Think about all the things you need to know in order to do TDD right:

- You have to know which features you want to implement.
- You have to be able to identify which parts of each feature should be tested.
- You have to learn a testing tool.
- You have to know how to write some tests for each of those parts.
- You have to know how to write a test without seeing the code first.
- You have to know how to write the "simplest thing that could possibly work."
- You have to learn how to use your tests to grow the design of the code.

- You have to know enough about refactoring and object-oriented design to keep your TDD'd code from turning into a mess.

When it's written out like that, I'd be surprised if you didn't struggle. It's clear that you can't really go from zero to TDD in a single step. So, what do you do? How can you learn TDD without being overwhelmed?

Break it apart!

Remember, when you feel like you're facing too much or you don't know what to do next, break it apart into smaller steps. Is there something easier you could learn that would get you closer to where you want to be?

For example, when you learn TDD, you could split it up like this:

- Write some code for a feature, without tests. (So you can get some experience coming up with and writing features)

- Study some code and write down some ways that the code might break on different kinds of input. (So you can learn about what to test and how to test)

- Write tests for code that already exists, using the Minitest built into Ruby (So you can learn Minitest and testing, without having to learn TDD yet)

- Test-drive some code using TDD and Minitest (Since you already know how to write tests, you just have to learn to write them first)

Each of these is a much smaller skill that you can work on separately. You can focus on and practice the first step, then the next, and so on. Instead of jumping straight to TDD, you can *grow* toward it.

It seems like a lot more work. But you'll learn each of these skills faster than trying to learn TDD from scratch, and you'll have a better foundation to build on when the next helpful testing style comes along.

Focus on one thing at a time

The key is to focus and practice on the thing just beyond your comfort zone, without trying to leap straight to the last step. Each of these steps has some value in itself. Even if you stopped after the first step, you'd still have learned something valuable. But each of these steps builds on the last. You can concentrate on learning *one thing* really well, instead of having a bunch of things all thrown at you at once. You can learn without getting overwhelmed and giving up.

Wrapping it up

To do TDD right, you need to already have a lot of skills: coding, testing, refactoring, and a good object design sense. Building these all at once won't be easy. These are all skills that you'll refine over time.

So, instead of reaching for TDD, grow toward it. Work on the skills that will get you pointed in the right direction, but aren't too much of a stretch.

When you need to grow toward a new skill like TDD, you should reach one level beyond your comfort zone. Something like 75% old skills, 25% new skills. That way, you'll be able to reorient yourself when you get lost.

Which of those skills do you know right now? Which will get you closest to knowing TDD, while also being built on stuff you already know?

Putting it all together

Testing in Rails is a gigantic topic. There's tons of stuff you need to learn and practice to get really good at it. It's incredibly easy to get overwhelmed. So I'm never surprised when I hear that testing (and especially TDD) is the topic that finally made someone feel like they weren't smart enough to learn Rails.

Testing is important. But it's not worth giving up Rails for. So don't worry if your first tests aren't great. Even stub tests or halfway finished tests are better than no tests – they'll at least remind you what still needs to be written.

When you get into testing, it's hard to know exactly how many tests to write, and what to test. Test too much, and you'll get burned out and your code won't be any more reliable. Too little, and bugs will creep back in.

Finding this balance is easiest if you add testing to your feature development process. That turns the process from Chapter 2 into something like this:

1. Take your small feature.
2. Think of *one* simple thing someone could do with that feature.
3. Draw *just enough* screens for that user to be able to do that thing.
4. Translate that sketch into a failing integration test.
5. Get the next part of that test to pass, by:
6. Writing a failing controller test.
7. Writing some failing unit tests.
8. Writing enough code to get the unit and controller tests to pass.
9. Repeat steps 5-8 until your integration test passes.

For each thing you test, you should have at least one test that verifies that the feature works, and a few tests that verify that it fails in the way it should, when it should. You might also have some "What if?" tests that can test some more unexpected situations.

When you run into a bug, follow this process to fix it:

1. Write a test that fails while the bug exists.
2. Fix the bug.
3. Make sure the test passes now.
4. Check in both your test and fix, so you don't run into the problem again.

It'll be easier to investigate the bug when you have it reproducing in a test, and the same bug won't show up again.

The code in your tests should follow three phases:

1. Arrange
2. Act
3. Assert

Not every test will have all of these phases, but thinking of this structure will make it easier to know what code to write next in your test case.

Your tests are just code, which means you can reorganize them if they get painful or repetitive. Some of the approaches that have helped me out are:

- Using Extract Method to write custom assertions.

- Using Extract Method to make mocks easier to write.
- Using modules to share tests between test suites.

But make sure you don't refactor too early. Your test code should be simple, not abstract.

Test-Driven Development is hard, because you have to know how to test before you can TDD. You also need lots of other skills, like writing Rails code, refactoring, and object oriented design knowledge.

So instead of going straight for TDD, work toward it by building the other skills. Those skills will help you write Rails apps in general, and as you get better, TDD will come a lot more easily. It'll almost be natural at that point.

Finally, don't let testing intimidate you. There's a lot to learn. But if you keep it simple, do what you can with the skills you have, and grow toward more advanced skills like TDD, you'll get there.

Chapter 4 What you need to learn, and what you can skip (for now)

Web development used to be easy. A little bit of HTML and a visitor counter in perl you grabbed from some website, and you had a professional looking homepage[33]. There weren't any stylesheets to fight, no front-end javascript frameworks, no asset pipeline. Just some text and a few images.

Now, it's a lot more complicated. There's an enormous number of technologies and skills[34] to learn to build Rails apps professionally. Beyond basic programming skills, there's Rails, testing, HTML / CSS / Javascript, deployment, SQL, and git. And that's skipping a lot.

33. http://www.microsoft.com/misc/features/features_flshbk_hp1.htm
34. http://www.readysetrails.com/index.php/181/this-is-why-learning-rails-is-hard/

If you tried to learn them all, it would take years. You'd get bored and give up after a few months if you weren't building anything real before then.

But Rails gives you support. You can muddle through most of these technologies until you need to know them. You don't need to know HTML right away, you can use scaffolds and helpers. You don't need to know how to deploy an app right away, you can run it on your development machine. You probably won't need to know SQL for a while, because ActiveRecord hides it well enough that you can ignore it at first.

You can learn each topic once it's interesting to you. Once you *need* to know it. And you'll learn more quickly when you understand *why* you need to know these technologies from your own experience, instead of learning them because a book tells you to.

That doesn't mean it'll be easy. There's still a lot to learn! But for now, you don't have to learn it all.

How deep you really need to go

When you start something huge like Rails, you should learn just enough to get by. Build a baseline level of skill.

Once you have to know more about an area, study it – and go more in-depth than you think you need to.

How deep is baseline knowledge?

Baseline knowledge is where you start. It's what you need to know about a topic in order to understand and practice what you learn. It's bootstrapping your learning process.

For example, if you're going to learn how to write HTML, you don't need much baseline knowledge – you just have to know how to use a text editor, what a tag is, and how to open your file in a web browser. Once you know that, you can start to learn and practice on your own.

For Rails, you need more baseline knowledge. You have to know how to open a terminal, start a Rails server, how and when model code, controller code, and view code is run, and how to run Rake tasks. But if you can already build a tiny Rails app, you're already there. Everything you learn after that is just building on the knowledge you already have.

Why go further than you need to?

Kathy Sierra gave a great talk[35] about the learning process and the stages you go through as you learn something new. In the talk, she refers to three stages of mastering something:

1. Can't do, but need to
2. Can do, with effort

35. http://businessofsoftware.org/2014/05/building-the-minimum-badass-user-pt-2-unfinished-business-kathy-sierra/

3. Mastered (reliable, automatic)

When you get to the third stage, you've mastered the skill. These skills don't require thought, you rely on instinct and intuition.

But there's a problem in the *second* stage, when you're past the basics but not an expert. You're better at the skill, but it's not internalized yet. You still have to think about it. This takes time and energy.

And if all your skills are in the second stage, you'll have to spend energy thinking about every skill you use.

For example, someone learning Rails might wonder how to make some messy controller logic better. They would need to know things like refactoring, patterns, object oriented design, and Rails conventions.

If that person was in the second stage in each of these things, they'd have a lot to think about. They'd consider the tradeoffs of different approaches, file structure, testing, stubbing and mocking, and abstraction. And the more they knew about each of those topics, the more factors they'd have to think about to make the best decision.

That's a lot to keep in mind all at once. Making that many conscious decisions can be tiring. And it might not seem worth doing that refactoring after all, if it can't be done right.

But what if that person had a lot of experience in refactoring and patterns, but almost no experience in Rails conventions? They might make some bad instinctive decisions, but they would also make good instinctive ones. More importantly, they wouldn't be stuck thinking about their decisions forever. You can always improve bad decisions, but you can't improve code that doesn't exist.

So, stay in the first and third stages. You'll be more productive with 5 skills in the third stage and 5 skills in the first stage, instead of 10 in the second stage.

Your Rails learning roadmap

Once you've built some baseline knowledge, you can learn more specific topics more deeply. But there are so many things you could study next. And when you're just starting out:

- You haven't built enough to know what you need to learn.
- It's easy to get stuck, and not know what skill you're missing.
- It's hard to understand what's worth learning and what you can skip for now.

You'll explore many different areas when you're learning to write Rails apps: things like Ruby, Rails, HTML, CSS, Javascript, and UI design. Some things will be especially helpful to learn early, and others you can put off until much later.

For example, with HTML, you can't do much without knowing about links, headers, formatting, and forms. But you shouldn't have to know how to write to a `<canvas>` tag until a real app needs it.

So I'm going to give you a roadmap. A guide to the things you'll use on almost every project. What to learn, and where to learn more.

This isn't a list of requirements. It's here for guidance. It's meant to help you when you get stuck, or don't know what to focus on next. Where you can get the most value out of the time you spend.

If you can learn a lot about this small set of things, you'll be well on your way to feeling totally confident when you start your next Rails app.

What do you need to know about Ruby?

You've probably heard that you can't learn Rails without knowing Ruby.

You don't need to know everything about Ruby. But when I'm writing a Rails app, most of the code I have to write isn't Rails-specific. It's just Ruby code. Code that would work outside of Rails.

You can do a lot with scaffolds and the Rails API. But the time you spend on Ruby outside of Rails will help you build much more interesting apps.

When learning Ruby with Rails, these are the things I'd focus on:

- **Basic syntax.** You should know how to call methods, do math, write `if` statements, that kind of thing.
- **Blocks.** In Rails, you'll use methods like `form_for` and `each` that pass values to a block. You should understand what blocks are, how they're used, and how to write them.
- **Attributes.** You don't have to know how `attr_reader`, `attr_writer`, and `attr_accessor` work internally, but you should be able to use them.

- **Defining classes and instance variables.** Otherwise, you won't be able to understand, write, or modify your Rails code.

- **Core data structures.** You should know a lot about `Enumeration`, `Array`, `String`, `Hash`, `Date`, `Time`, and `DateTime`, and study their APIs closely. You'll spend a surprising amount of time with these classes in your apps. Learn these well, and you'll do amazing things with only a single, clear line of code.

Early on, learn how to search, read, and understand the Ruby and Rails API documentation. rdoc.info[36] will get you far. The Ruby documentation can be hard to find: http://rdoc.info/stdlib/core/frames is where you want to look. Type "Array" into the search box, for example, and you'll get tons of information about arrays in Ruby.

When you look up documentation on a method, you'll see other methods and classes that will expand your idea of what's possible with Ruby. Just like Wikipedia, it can be fun to follow interesting-looking links and see where they take you. The docs also include Ruby source code and examples, which will help you learn what good Ruby looks like.

So, if you want to get started with Ruby, where do you begin?

The quick start guide in Agile Web Development with Rails[37] will give you enough of the syntax to move forward with Rails. For more depth, Programming Ruby 2.0[38] will help. Eloquent Ruby[39] will teach you to write better-looking Ruby code. And as you learn Rails, you'll practice and learn more Ruby, and vice versa.

36. http://rdoc.info
37. https://pragprog.com/book/rails4/agile-web-development-with-rails-4
38. https://pragprog.com/book/ruby4/programming-ruby-1-9-2-0
39. http://www.amazon.com/Eloquent-Ruby-Addison-Wesley-Professional-Series/dp/0321584104

And Rails?

Like Ruby, there's so much stuff in Rails, but you'll use small parts of it most of the time. Learn about the common things, and you'll be able to build apps without having to stop every few minutes to study documentation.

In my Rails apps, I tend to touch the same few things:

- **Generators.** They're the fastest way to get started. Use them well and they'll speed you up. They're especially great for building tiny Rails apps.

- **Migrations.** It's rare that you'll write a Rails app without a database, so you should know how to generate and modify database tables to store data.

- **Validations.** You can't write a professional Rails app without validating data, so you have to learn these.

- **ActiveRecord Associations and Scopes.** Well-written associations can make the rest of your Rails code much simpler. A *lot* of your time in Rails will be spent using associations and scopes.

- **Form helpers.** Most Rails apps need to collect some data, so learning the form helpers well will help you out.

- **`link_to` and `routes`.** Knowing how URLs map to code is important, so you don't get lost in your app[40].

40. If you *do* get lost in your routes, Rails' `rake routes` command will display a lot of helpful information about the routes in your app.

- **How values make it from the controller to the view and back.** This can be confusing, and if you don't understand it you can run into weird bugs that'll take forever to fix up.

Generally, if you spend most of your time learning ActiveRecord well, you'll be fine. Most of your Rails code will be in ActiveRecord models or plain Ruby objects.

HTML / CSS?

When you build Rails views, you'll probably work in HTML. When you want to make your pages look good, you'll write CSS. If you don't already know these, you'll have to learn them to build Rails apps. But HTML and CSS are nice to learn, because you can start with easy and functional pages, and work in small steps up to beautiful and engaging.

Start by getting a basic HTML page rendered. You'll need to know:

- The html/head/body skeleton of an HTML page.
- What tags and elements are, what attributes are, and how they're used along with text on your page.

For HTML, I like to focus on the "Markdown Subset" of HTML – that is, the kind of HTML that Markdown[41] can write. That's mostly:

- Headers (`h1`, `h2`, `h3`).
- Links (`a` tags).

41. http://daringfireball.net/projects/markdown/syntax

- Images (`img` tags).
- Lists (`ul`, `ol`, and `li`).
- Formatting (`strong`, `em`).
- Paragraphs (`p` tags).

Until you use lots of CSS and JavaScript, you'll stick to those tags on most pages.

You'll also need to know how to use `forms` and `inputs`. Even if you use `form_for` in Rails instead of writing forms by hand, you'll need to know how the different kinds of input look and act. And you'll spend enough time working with forms that they're worth learning well.

Once you start adding CSS or bootstrap, you'll have to understand `div` and `span` tags.

If you've never built apps for the web before, skip CSS until you master a few other skills first. Just use HTML without CSS. Your pages might not look great, but they'll work well enough to learn other things.

When you learn CSS, start with Bootstrap[42]. CSS alone is really fiddly and annoying. Even after years working on webapps, I still find CSS to be the most frustrating part. And enough sites use Bootstrap that it's useful to know on its own.

Bootstrap is a set of CSS styles that make webapps look good without a lot of work. For Rails, I use the bootstrap-sass[43] gem.

42. http://getbootstrap.com
43. https://github.com/twbs/bootstrap-sass/

If you use Bootstrap, focus on learning the grid really well. You'll spend a lot of time laying out your pages, and that's what the grid is for. A little work with the Bootstrap grid, and the look of your pages will really improve.

I actually don't know of any great HTML / CSS books for beginners. But since HTML and CSS are visual and easy to run and reload, you'll pick it up quickly. Concentrate on the basics, and you should see how the code you write changes your page.

So, create a sample Rails app with a scaffold or two. Load it up in the browser while you load some Rails view code in your editor. Play with the HTML, and see how your page changes.

As you learn more, you might even be able to grab HTML from other sites that seem to do cool things. This is getting harder[44], but it's still possible.

JavaScript?

Once you know some HTML and CSS, you can learn a little bit of JavaScript. With snippets of javascript in `.js.erb` files rendered by Rails, you can make your apps much more interactive without a lot of work. Most sites still use jQuery[45], and it's still the default in Rails, so you can assume that you have it in your own app.

You should learn how to:

- Select HTML elements with CSS selectors (like `$('.user')` to select all elements with `class="user"`).

44. Because a lot of HTML, CSS, and JavaScript is processed into an unreadable mess.
45. http://jquery.com

- Show and hide elements after you select them. (like `$('.user').hide()`
- Add bits of HTML to the page.

Using the Rails helpers (like `link_to` with `:remote => true`) will also take you far.

You can ignore single-page app frameworks (like Backbone, Ember, or Angular) while you're starting out. You'll be able to learn much more quickly when you're not bouncing between Ruby and JavaScript. Besides, Rails still favors server-side code, and Rails is *so* much easier when you do what it wants you to do.

What else?

Even if you don't know a lot about design, it doesn't take much to get something looking decent. Focus on improving your typography, layout and grids, and whitespace. You'll see the biggest benefit for the time you spend.

Design for Hackers[46] is a great free email course that'll teach you the basics.

How do you use this list?

This may seem like a lot, but remember: you don't have to pick it all up at once. It's just a catalog of valuable things to start learning when you don't know what to study next.

You can explore and practice most of these things on their own. So when you get interested, find some documentation, a book, or API reference. Create a tiny app, and

46. http://designforhackers.com

experiment. Use the techniques from Chapter 1 to learn these things clearly. And with some practice, you'll be much closer to becoming a solid Rails web developer.

The final list

In Ruby:

- Basic Syntax.
- Blocks.
- Attributes.
- Classes and instance variables.
- Enumerations, strings, hashes, arrays, dates, and times.
- How to search and read the API documentation.

My favorite books for learning Ruby are Programming Ruby 2.0[47] and Eloquent Ruby[48]. Both are really good.

In Rails:

- Generators.
- Migrations.
- Validations.
- Associations and Scopes.

47. https://pragprog.com/book/ruby4/programming-ruby-1-9-2-0
48. http://www.amazon.com/Eloquent-Ruby-Addison-Wesley-Professional-Series/dp/0321584104

- Form helpers.
- `link_to` and Routes.
- How data gets from the view to the model, and back up.

I usually recommend Agile Web Development with Rails[49] to pick up the Rails basics. Learn Ruby on Rails[50] is good for absolute beginners – it's filled with definitions and explanations, and teaches basic Rails concepts in an inviting way. Ruby on Rails Tutorial[51] is also good, but can be overwhelming, because it covers everything. The new edition[52] uses the standard Rails stack, and should be easier to get into.

Besides those, the Railscasts[53] are a great way to learn some small Rails topics in more detail.

For HTML / CSS:

- General page structure (html, head, title, body).
- The "Markdown Subset" (headers, strong, em, lists, links, images).
- Forms and inputs.
- (A little later) divs and spans.
- Bootstrap[54].

49. https://pragprog.com/book/rails4/agile-web-development-with-rails-4
50. http://learn-rails.com/learn-ruby-on-rails.html
51. https://www.railstutorial.org
52. http://news.railstutorial.org/rails_tutorial_3rd_edition/
53. http://railscasts.com
54. http://getbootstrap.com

I don't know of any great beginning HTML and CSS books. If you know of any, please let me know! Get familiar with Mozilla Developer Network[55] – it's the best reference for HTML and CSS.

For JavaScript:

- Selecting elements by class name or id.
- Adding / removing elements from the page.

I can't vouch for any great JavaScript books either, but I've heard good things about Human Javascript[56]. The Codecademy Javascript[57] and jQuery[58] tracks would be good places to start. And the Mozilla Developer Network[59] and jQuery documentation[60] are your standard JavaScript API references.

For UI:

- Typography
- Layout and grids
- Whitespace

Design for Hackers[61] will teach you the basics. If you go through that course, you'll be able to design better-looking pages than a lot of developers I know.

55. https://developer.mozilla.org/en-US/
56. http://read.humanjavascript.com/
57. http://www.codecademy.com/tracks/javascript
58. http://www.codecademy.com/tracks/jquery
59. https://developer.mozilla.org/en-US/
60. http://api.jquery.com
61. http://designforhackers.com

If you learn these topics well, you won't have to reach for a reference book every five minutes. You'll stay in flow, and that's when development gets really fun!

Exercises

Do a quick self-evaluation. Look at the Ruby section: What do you know already? What are you missing? Which of the topics seem interesting to you?

Then, answer these questions:

- What are you going to learn next?
- Where are you going to learn it?
- How will you know when you have learned enough about it, for now?

"For Now" is a powerful phrase

I tried to keep that roadmap as short as possible. But even cut down, it seems like a lot to learn. And don't you eventually need to know everything? Won't you run into problems if you skip stuff?

The trick of it is, you can really only focus on one thing at a time. Choosing something to learn means you can't do the rest right now.

But that choice, saying "I'm going to pick this thing to do" is hard. It's stressful. The pressure of all the things you didn't choose weighs on you. And you won't feel confident that you made the right decision, when there's a universe of other decisions you didn't even consider.

Looking at the problem in reverse can help.

I come back to a two-word phrase:

"For now."

As in, "That's right, I'm *not* going to do these other things. But only for now."

In other words, make a "Not Right Now" list.

The Not Right Now list

It might seem like you need to master JavaScript before you can write a Rails app. You will master JavaScript, someday. But you won't get anywhere without an app you can get excited about, and you don't need JavaScript for that first stage. So cross JavaScript off the list. For now.

I'm not sure why. But calling out all the things you're *not* going to do is so much easier than picking the things you *are* going to do. It's refreshing to say no. Especially when you know your decisions aren't permanent.

You won't get stuck worrying about what you should be learning instead of what you're actually learning. And when you come back to your list later, you won't even care about half the stuff on there anymore.

"For now" is powerful. You're giving yourself permission to set other things aside so you can focus. You won't study things shallowly because you're trying to chase the next thing you feel like you're missing. So you'll learn a lot more in less time.

Putting it all together

As you grow past being a beginner at Rails, things get more confusing. It's not even the framework. (That, you'll figure out). It's which path you take, what you learn, what you should study. It's about what to focus on, facing the nearly infinite amount of stuff you *could* be learning.

It's about deciding which path to take when you don't know which one is right yet.

Here, I've talked about a few skills that I keep coming back to while I write Rails apps. And I'm confident the time you spend in learning them well will pay off.

But even then, you still have to figure out what to start with.

So, take a look at what you know and what you don't. What you want to learn, and what your app needs you to learn. Set aside some things that seem less important, and turn them into a "Not Right Now" list. Eventually you'll have a few things you just can't set aside – learn those thoroughly. And keep moving forward.

Intermediate Rails isn't about learning all the stuff you learned as a beginner in a little more depth. It's about the stem of the "T" in T-Shaped Learning. It's about gaining deep knowledge in a few different areas, one thing at a time. And it's about using that knowledge to build your own apps, the way you imagined them.

Chapter 5
What to do when things go wrong

Errors are the most frustrating part of getting started with anything new.

It's the worst in the beginning. Just when you start to get excited, you run into an error message that doesn't make any sense.

Is it your fault? Is it the book's fault? Or the framework's fault?

How do you debug an error when you don't even know what's *supposed* to happen?

If anything's going to make you completely give up, it's this. Getting stuck when you're already short on time will stress you out like nothing else.

But once you learn a few techniques, and recognize a few patterns, you can start solving these errors once and for all.

Errors: How do you fix them?

So, you saw a bunch of words spew out onto the screen. Did you just break everything?

Ruby errors can be intimidating. You miss a character somewhere, and all of a sudden your screen is full of lines of text, talking about methods and files you've never seen before:

```
ActiveModel::ForbiddenAttributesError
      (ActiveModel::ForbiddenAttributesError):
  activemodel (4.1.1) lib/active_model/
      forbidden_attributes_protection.rb:21:in
      `sanitize_for_mass_assignment'
  activerecord (4.1.1) lib/active_record/
      attribute_assignment.rb:24:in `assign_attributes'
  activerecord (4.1.1) lib/active_record/core.rb:452:in
      `init_attributes'
  activerecord (4.1.1) lib/active_record/core.rb:198:in `initialize'
  activerecord (4.1.1) lib/active_record/inheritance.rb:30:in `new'
  activerecord (4.1.1) lib/active_record/inheritance.rb:30:in `new'
  app/controllers/bugs_controller.rb:27:in `create'
  actionpack (4.1.1) lib/action_controller/metal/
      implicit_render.rb:4:in `send_action'
  actionpack (4.1.1) lib/abstract_controller/base.rb:189:in
      `process_action'
  actionpack (4.1.1) lib/action_controller/metal/rendering.rb:10:in
      `process_action'
```

What just happened? A Ruby `Exception` did, and you're looking at a backtrace (or stacktrace). Backtraces are awesome, because they can show you exactly where your error took place. All you have to do is read the backtrace and you'll have a good chance of finding out where your bug's coming from.

How do you read a backtrace?

An exception has three parts: the exception type (or class), a message, and a backtrace. Rails exception messages tend to be pretty good:

```
ActiveRecord::RecordNotFound (Couldn't find Bug with 'id'=1)
```

They'll usually tell you exactly what the problem is. So it's a good idea to look at the exception message first.

But sometimes the message won't be all that helpful:

```
ActiveModel::ForbiddenAttributesError
      (ActiveModel::ForbiddenAttributesError)
```

I mean, what?

When that happens, you'll have to read the backtrace.

A backtrace is a list of all the methods that you were inside when the error happened. For example, with this code:

```
def outer_method
  first_method
  second_method
end
```

If an exception was thrown in `second_method`, you'd see something like:

```
test.rb:9:in `second_method': Some kind of error happened!
      (RuntimeError)
    from test.rb:3:in `outer_method'
    from test.rb:12:in `<main>'
```

You read backtraces from top to bottom. So in this case, focusing on the method names (surrounded by ` and '), you can see that the error happened in `second_method`, which was called by `outer_method`, which was called by `<main>`.

You probably won't need to read all the way to the bottom, since errors usually happen in one of the top few methods in the backtrace.

Backtraces will tell you where a problem happened, down to the line number. The format you'll see is:

```
file name:line number: in `method name`
```

The first line will also tell you what kind of exception was thrown.

So, for the first error we saw:

```
app/controllers/bugs_controller.rb:27:in `create'
```

The error happened in `app/controllers/bugs_controller.rb`, line 27. In the second:

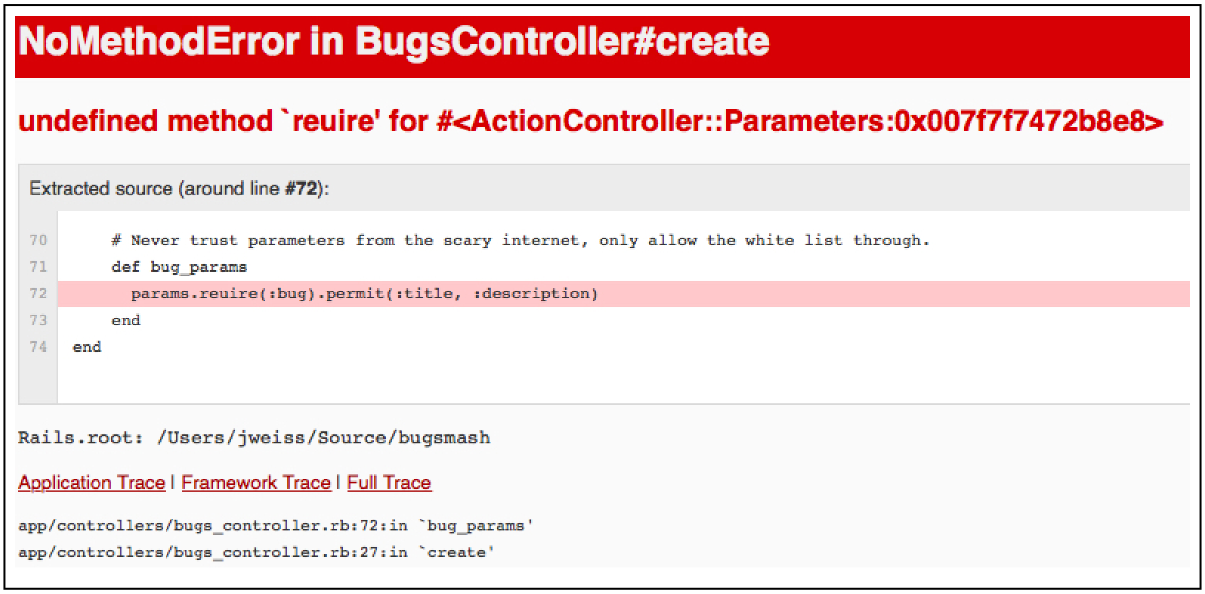
```
test.rb:9:in `second_method': Some kind of error happened!
      (RuntimeError)
```

The error happened in `test.rb`, line 9.

Reading exceptions and backtraces gets a lot easier with practice. These days, I can usually just look at an exception and know where the problem is right away. But it's taken me a while to get to that point. So don't worry if they're hard to read at first.

Finding problems *you* caused

Rails usually does a good job of showing you where an error happened:



NoMethodError in BugsController#create

undefined method `reuire' for #<ActionController::Parameters:0x007f7f7472b8e8>

Extracted source (around line **#72**):

```
    # Never trust parameters from the scary internet, only allow the white list through.
    def bug_params
      params.reuire(:bug).permit(:title, :description)
    end
end
```

```
Rails.root: /Users/jweiss/Source/bugsmash
```

Application Trace | Framework Trace | Full Trace

```
app/controllers/bugs_controller.rb:72:in `bug_params'
app/controllers/bugs_controller.rb:27:in `create'
```

But sometimes what Rails shows you won't make sense, or you won't recognize the code you're seeing. You might think it's a bug in Rails, or some other code that you didn't write. That's so tempting to believe, but it's rarely true.

In The Pragmatic Programmer[62], you'll find a phrase that will help fight that temptation: "`select` isn't broken[63]"

> *It is rare to find a bug in the OS or the compiler, or even a third-party product or library. The bug is most likely in the application.*

This goes for Rails, too. Usually, if you run into an error, it's in code you wrote. So, look for *your* code in the stack trace. You can usually just scan down the backtrace until you see a filename starting with `app/`.

Even if the error wasn't your fault, it's still a good idea to look for your code first. If you can find out where in your app an error happened, you'll probably be close to where you'll need to be to fix it.

To make backtraces a little friendlier, Rails hides a lot of the lines from you. To me, this is more annoying than helpful, because I always forget that Rails does this. I'll look in the wrong place for the error, and that just wastes time.

So, when I start a new Rails project, I'll usually turn backtrace hiding off. You can do this by uncommenting a line in `config/initializers/backtrace_silencers.rb`:

config/initializers/backtrace_silencers.rb

```
# You can also remove all the silencers if you're trying to debug a
      problem that might stem from framework code.
Rails.backtrace_cleaner.remove_silencers!
```

62. https://pragprog.com/book/tpp/the-pragmatic-programmer
63. https://pragprog.com/the-pragmatic-programmer/extracts/tips

Your backtraces will be noisier. But you'll get more practice reading them. You'll be able to walk all the way to the source of the problem, every time. And if it turns out to be too much information, it's easy to turn it back off.

Wrapping it up

Once you learn to read exceptions, you can often instantly find the code that caused an error.

If the exception has a good message, you might not have to read the exception's backtrace. Other times, you'll have to dig through the backtrace to find out where you should look for the problem.

When you read a backtrace, go from top to bottom. Usually, the code that needs fixing is one of the first few lines. And start from files in `app/` or `lib/`, because the problem is most likely in your own code.

Exercises

The more backtraces you see, the more quickly you'll use them to guide your fixes. So I have a few examples to walk through.

```
NoMethodError (undefined method `reuire' for
      #<ActionController::Parameters:0x007f7f7472b8e8>):
  app/controllers/bugs_controller.rb:72:in `bug_params'
  app/controllers/bugs_controller.rb:27:in `create'
  actionpack (4.1.1) lib/action_controller/metal/
      implicit_render.rb:4:in `send_action'
  actionpack (4.1.1) lib/abstract_controller/base.rb:189:in
      `process_action'
  actionpack (4.1.1) lib/action_controller/metal/rendering.rb:10:in
      `process_action'
  actionpack (4.1.1) lib/abstract_controller/callbacks.rb:20:in
      `block in process_action'
```

```
ActionView::Template::Error (undefined method `title' for
      nil:NilClass):
    2:
    3: <p>
    4:   <strong>Title:</strong>
    5:   <%= @bug.title %>
    6: </p>
    7:
    8: <p>
  app/views/bugs/show.html.erb:5:in
      `_app_views_bugs_show_html_erb__3646330667202147 10_70092694084120'
  actionview (4.1.1) lib/action_view/template.rb:145:in `block in
      render'
  activesupport (4.1.1) lib/active_support/notifications.rb:161:in
      `instrument'
  actionview (4.1.1) lib/action_view/template.rb:339:in `instrument'
  actionview (4.1.1) lib/action_view/template.rb:143:in `render'
  actionview (4.1.1) lib/action_view/renderer/
      template_renderer.rb:55:in `block (2 levels) in
      render_template'
```

```
ActiveModel::ForbiddenAttributesError
      (ActiveModel::ForbiddenAttributesError):
  activemodel (4.1.1) lib/active_model/
      forbidden_attributes_protection.rb:21:in
      `sanitize_for_mass_assignment'
  activerecord (4.1.1) lib/active_record/
      attribute_assignment.rb:24:in `assign_attributes'
  activerecord (4.1.1) lib/active_record/core.rb:452:in
      `init_attributes'
  activerecord (4.1.1) lib/active_record/core.rb:198:in `initialize'
  activerecord (4.1.1) lib/active_record/inheritance.rb:30:in `new'
  activerecord (4.1.1) lib/active_record/inheritance.rb:30:in `new'
  app/controllers/bugs_controller.rb:27:in `create'
  actionpack (4.1.1) lib/action_controller/metal/
      implicit_render.rb:4:in `send_action'
  actionpack (4.1.1) lib/abstract_controller/base.rb:189:in
      `process_action'
  actionpack (4.1.1) lib/action_controller/metal/rendering.rb:10:in
      `process_action'
```

For each of those examples, which of those lines refer to your code? Which file and line would you look at? Without seeing the code, what do you think happened?

Debugging: A developer's most valuable skill

Debugging code is one of those skills you just *have* to build. If you can't debug your code, the first problem you run into will stump you. And if you can't debug other people's code, you'll never know why a library isn't working in your app.

If you're coming from another programming language, this won't be a surprise. But you'll notice that many Rails developers don't use an IDE[64] to write Ruby code. That leaves you without a lot of the debugging support you might be used to.

And if you're not familiar with debugging at all, you might not know where to start.

How do you debug code easily?

Debugging is all about learning what your code is doing while it runs. And with Ruby and Rails, there are a few great ways to do just that.

`puts` is a programmer's best friend. It's quick and simple, but it will help you debug almost anything. In fact, the hardest bug I've ever run into was solved using nothing but printing values, when all other debuggers failed.

There are a few ways you can use `puts` to debug problems:

Printing out variables

This one is easy. Say you have a method:

```
def post_name(post_params)
  post_params[:name]
end
```

But when you run it, it returns `nil`! With `puts`, you can see what's going on:

64. An IDE, or Integrated Development Environment, is a source code editor that usually has debuggers, code navigation, and other neat stuff built-in. RubyMine[65] is the most common IDE you'll see in the Ruby world.
65. https://www.jetbrains.com/ruby/

```
def post_name(post_params)
  puts post_params.inspect
  post_params[:name]
end
```

Now, when you run it, you'll see this printed to the screen:

```
{"name"=>"My Sample Post"}
```

Ah, it looks like we used a string inside the hash, but tried to access it using a symbol. So we could fix it with:

```
def post_name(post_params)
  post_params["name"]
end
```

Printing variables is most helpful when you're pretty sure *where* the problem's happening, but you're not sure *why*. It helps you make sure that the code you're running is actually doing what you think it's doing.

Entry / exit of functions

Sometimes, you'll see a method, and you're not sure how or when it gets called. Or maybe a method you wrote isn't getting called when you think it should be.

When that happens, you can add `puts` statements inside other methods, and have them show you how your program flows:

```
def first_method
  puts "inside first_method"
  inner_method
end
def second_method
  puts "inside second_method"
end
def inner_method
  puts "inside inner_method"
  # ... do stuff ...
end
first_method
second_method
```

Here, you'd see:

```
inside first_method
inside inner_method
inside second_method
```

And that'll help you figure out exactly how the code actually runs. This is especially powerful when you combine it with `caller`[66], which returns a full backtrace up to that point:

```
def inner_method
  puts caller(0)
  # ... do stuff ...
end
```

66. http://www.ruby-doc.org/core-2.1.5/Kernel.html#method-i-caller

```
~/Source jweiss$ ruby test.rb
test.rb:14:in `inner_method'
test.rb:7:in `first_method'
test.rb:17:in `<main>'
```

This way, you can see the path that led to your method getting called.

Binary searching your code

Sometimes, your code will just break, and you'll have absolutely no idea where it happened. Maybe a variable became nil, but it *used* to be OK. Binary searching your code will quickly tell you where the problem started.

Did you ever play the "guess what number I'm thinking of" game as a kid?

I'd think of a number between 1 and 100, let's say 40. You guess 50. I say, "lower," and you now know you don't have to think about 50-100 anymore. You just eliminated half the numbers. So you pick 25. I say, "higher," which narrows it down even further. You pick 38, I say "higher" again, and so on.

You don't have to guess every number. You pick good numbers, around halfway between your other guesses. That helps you get close to the right number quickly.

Binary searching is exactly like this.

You can say, "Before this point in my program, everything looked good. After this other point in my program, there was a problem. So the error must have happened somewhere between these two points." And then you bring those two points closer, to shrink the number of places you have to look.

You stick a `puts` statement somewhere between the good point and the bad point, and see what your data looks like. If your `puts` statement shows that everything is OK, you know you don't have to check any of the code before it. If it shows that it's broken, you don't have to check anything after it. The amount of code you have to check shrinks quickly, and you can narrow down on the exact bit of code that solved the problem.

The process looks like this:

Say you had a variable that turned into `nil` somewhere, but you knew it was OK when you first saw it.

```
def a_long_method(value)
  puts value # => "Hello!"
  # ... code ...
  # ... more code ...
  # ... more code ...
  # ... more code ...
  # ... this is a super long method ...
  puts value # => nil
end
```

So you know the problem must have happened between those two points. Let's see if we can narrow down the problem to half that method:

```
def a_long_method(value)
  puts value # => "Hello!"
  # ... code ...
  # ... more code ...
  puts value # => "Hello!"
  # ... more code ...
  # ... more code ...
  # ... this is a super long method ...
  puts value # => nil
end
```

OK, so we know it happened somewhere in the bottom half. Let’s split that up:

```
def a_long_method(value)
  # ... code ...
  # ... more code ...
  puts value # => "Hello!"
  # ... more code ...
  # ... more code ...
  puts value # => nil
  # ... this is a super long method ...
  puts value # => nil
end
```

Looks like it’s one of those two lines in the middle:

```
def a_long_method(value)
  # ... code ...
  # ... more code ...
  puts value # => "Hello!"
  # ... more code ...
  puts value # => "Hello!"
  # ... more code ...
  puts value # => nil
  # ... this is a super long method ...
end
```

Now we know it broke somewhere in that second-to-last line, and it only took a few steps to discover that.

The binary search process isn't just for debugging code. It works in lots of other areas. It's so useful that git[67] and Minitest[68] both automate it.

Binary searching has helped me solve some of the craziest problems I've ever run into. If you're patient and careful and don't take shortcuts, it really can't fail.

`puts` isn't always the best way to debug problems you run into. But it's the simplest, and will work where other debuggers fail. So, spend the time to get to know how to use it well.

67. http://git-scm.com/book/en/Git-Tools-Debugging-with-Git
68. https://github.com/seattlerb/minitest-bisect

How can you use logs to figure out what went wrong?

Rails logs a ton of information into the files under `log/`. These can be used for a lot of the same things as `puts`, but a lot of the work of printing things is already done for you.

These log files are automatically shown when you run a Rails server in development mode, and look like this:

```
Started GET "/" for 127.0.0.1 at 2014-09-04 05:55:08 -0700
  ActiveRecord::SchemaMigration Load (0.7ms)  SELECT
       "schema_migrations".* FROM "schema_migrations"
Processing by BugsController#index as HTML
  Bug Load (0.9ms)  SELECT "bugs".* FROM "bugs"
  Rendered bugs/index.html.erb within layouts/application (18.6ms)
Completed 200 OK in 263ms (Views: 204.1ms | ActiveRecord: 1.6ms)
```

When you're running tests or Rake tasks, you won't see these files automatically. Instead, you'll have to open the log files yourself. So, if you're trying to find the logs for a test run you just did, look at `log/test.log`[69].

These log files will help you discover things like whether you're executing the wrong SQL statements or rendering the wrong partial. If you prefer to have all of your information in one place, you can use `Rails.logger.info` instead of `puts` to print things to the log file:

69. I usually open an extra Terminal window, where I run `tail -f log/test.log` from my Rails app's directory, to watch for these log messages.

app/controllers/bugs_controller.rb

```
def index
  @bugs = Bug.all
  Rails.logger.info "-----"
  Rails.logger.info "@bugs = #{@bugs.to_a}"
  Rails.logger.info "-----"
end
```

```
Started GET "/" for 127.0.0.1 at 2014-09-04 06:01:33 -0700
Processing by BugsController#index as HTML
-----
  Bug Load (0.7ms)  SELECT "bugs".* FROM "bugs"
@bugs = [#<Bug id: 1, title: "", description: "", created_at:
       "2014-09-03 13:33:48", updated_at: "2014-09-03 13:33:48">]
-----
  Rendered bugs/index.html.erb within layouts/application (16.3ms)
Completed 200 OK in 44ms (Views: 29.1ms | ActiveRecord: 1.4ms)
```

Neat! We can even see that the SQL isn't run until `#to_a` is called.

If you can use the logging that Rails already gives you, you won't have to use `puts` to find certain kinds of problems. It can save you a lot of work.

How can you walk through code to discover the source of a problem?

`puts` and logs are quick and helpful, but they won't always tell you what you need to know. Sometimes you'll want to see and modify variables. Or you might want to call methods yourself with different parameters, so you can see what would happen in different situations.

When you want to play with running code, try a debugger. Most programming languages have debuggers, and most of them are part of the editors you use to write code in those languages. But even though Rubyists have RubyMine[70], a really good Ruby/Rails editor with a built-in debugger, most people don't use it. (The debugger is good enough, though, that I know people that use RubyMine *just* for debugging).

If you're using a plain text editor like a lot of Rubyists do, you'll probably use a command line debugger. And the best command line debugger for Ruby 2.1 is Byebug[71].

If you're using Rails 4.2 with Ruby 2.0 or newer, `rails new` will include Byebug for you. If you aren't, Byebug is easy to install. Just add it to your Gemfile:

Gemfile

```
gem "byebug", group: [:development, :test]
```

and run `bundle install`.

So what does Byebug do for you?

Well, say you ran into a bug in your app. You have a set number of things available to sell, but you can only sell a fraction of them to any one person. Anytime you load the page, though, the maximum amount you can sell turns out to be 0. The method looks like this:

70. http://www.jetbrains.com/ruby/
71. https://github.com/deivid-rodriguez/byebug

app/models/item.rb

```
class Item < ActiveRecord::Base
  def sellable_count
    1/6 * available_count
  end
end
```

app/views/items/_item.html.erb

```
<td><%= @item.sellable_count %></td>
```

You could stick a `byebug` statement in the view:

app/views/items/_item.html.erb

```
<td><%= byebug;@item.sellable_count %></td>
```

and you'd see a `rails console`-like command line in your terminal when you hit the page:

```
[11, 20] in /Users/jweiss/Source/sales/app/views/items/_item.html.erb
   11:
   12:   <tbody>
   13:     <% @items do |item| %>
   14:       <tr>
   15:         <td><%= item.title %></td>
=> 16:         <td><%= byebug;item.sellable_count %></td>
   17:         <td><%= link_to 'Buy', new_cart_item_path(item)
      %></td>
   18:       </tr>
   19:     <% end %>
   20:   </tbody>
(byebug) _
```

From this debugging console, you can do all kinds of things. You can step through your code line by line, you can call methods, you can print the values of variables – everything you can do with `rails console` and more. Byebug includes a great walk-through, which you can find at https://github.com/deivid-rodriguez/byebug/blob/master/GUIDE.md.

Besides fixing bugs, Byebug can help you explore code that you don't know well. Instead of having to read the code and remember what each variable is used for, you can step through it and see the real values that show up in actual situations.

A debugger is one of the most valuable tools you have. Spend the time to learn one well.

Wrapping it up

There are many, many tools to help find and fix problems in your code. For me, the most valuable have been `puts`, logging, and debuggers.

`puts` is simple and works everywhere, but can be the most work. You can use `puts` for lots of things, but I find it most useful for:

- Printing out the values of variables
- Keeping track of when methods were called
- Binary searching large chunks of code to narrow down where to look for bugs

Rails uses something like `puts` to log tons of stuff to a file, from the SQL it executes to the partials it renders. You can see what Rails prints by looking in the files under `log/`, and you can even print your own stuff to this log with `Rails.logger.info`.

When you need to play with the code to understand what it's doing, use a debugger like Byebug. A lot of the time, it'll be faster and cleaner than all the `puts` statements you'd have to sprinkle everywhere, and you can explore unfamiliar code instead of reading it.

Most of debugging is about seeing where code in the real world differs from the code as you imagined it. If you have a way to see the values of variables and a way to run code, you can solve just about any problem you run into.

Research: When debugging fails

Sometimes a problem will occur deep inside Ruby or Rails, and you won't be able to get to it with a debugger. Or maybe debugging the problem is just beyond your skill level right now.

When that happens, you're going to have to do some research to figure out the cause. And you might have to get help.

Why Google and StackOverflow may not be the answer

When some people run into a problem, they'll jump straight to Google, search on the error message, and see if someone's already solved it.

This can be the fastest way to get your problem solved, and is a lot easier than investigating it yourself. But you lose the opportunity to investigate it, and you miss the chance to build experience debugging and solving your own problems. This robs you of a chance to get to know your code, the language, and the framework better.

Every problem you run into is an opportunity to learn. StackOverflow and Google and other sites can help you with that. But if you use them to solve a problem, make sure you read the entire answer, follow references, and immerse yourself in the knowledge the solution brings you. Don't just copy and paste the solution into your own code.

If you use other developers' solutions to fix your problems, your app will become a mess of inconsistent code that probably only works coincidentally.

When you're short on time, sure, see if you can find an answer online. But try debugging a little bit on your own before you search. The skills you build will help you when StackOverflow and Google don't have the answer.

The internet turned up a single thread from four years ago, with no comments[72]. What do I do now?

Sometimes you'll be stuck, you'll do a search, and nothing useful comes up. What do you do then?

You're going to have to seek out your own answers, or find help.

Dig into unfamiliar code

Great developers know how to read and understand code. When you read code, you'll understand much more than what the documentation tells you. After spending enough time with the code, you'll know as much about it as the author does!

72. http://xkcd.com/979/

But reading code isn't like reading a book. You can't just scan a bunch of files and immediately understand what they do.

Instead, experiment with the code you're trying to read. Write down your thoughts and discoveries as you have them. Run the code and its tests in a debugger, print messages showing you where you are, and modify code to see how it affects its surroundings. Break it as much as you want, if it helps you understand it better. Once you pull the code to your machine, it's yours to play with.

Learn from others

Lots of projects have mailing lists or IRC chatrooms where people talk and ask questions about the project. On these lists and chats, you can learn from the author of the project you're having trouble with. The project's homepage will usually tell you about these, where they are, and how to access them.

When you have a question about a project, bring together all the details you've gathered so far, write up a short description of the problem and what you've tried, and use it to make your question as clear as possible.

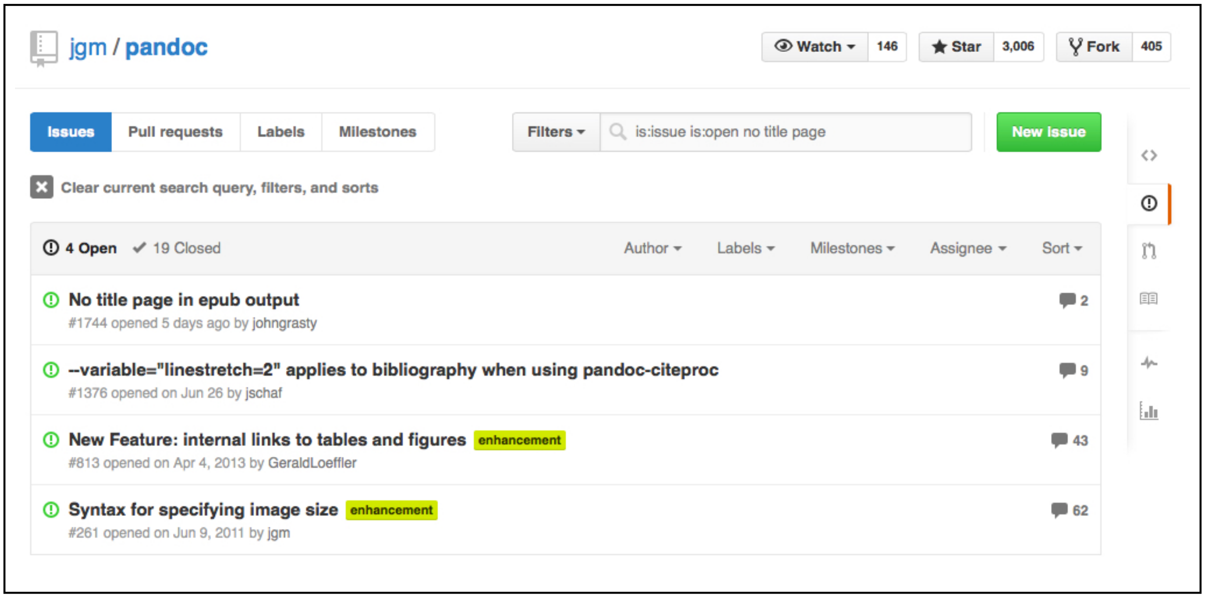
If you have a sample project that shows the problem, it's incredibly helpful for the maintainer. They'll almost always be able to fix a problem if they can see it happening themselves.

It can be disappointing if you don't get a response. But don't let waiting keep you from investigating. Learn more about the problem on your own, and ask again later.

The Ruby community is really good about helping people out. I have a few open source projects, and I love hearing from the people that use them. Often, the problems they run into are bugs I didn't even know about, which helps me fix them for everyone.

How do you get help from a GitHub page?

Most projects on GitHub have an "Issues" page, where people file bugs. If you have a problem, a quick search can show you who else has run into it:



is:issue is:open no title page

Clear current search query, filters, and sorts

4 Open ✓ 19 Closed

No title page in epub output 2
#1744 opened 5 days ago by johngrasty

--variable="linestretch=2" applies to bibliography when using pandoc-citeproc 9
#1376 opened on Jun 26 by jschaf

New Feature: internal links to tables and figures enhancement 43
#813 opened on Apr 4, 2013 by GeraldLoeffler

Syntax for specifying image size enhancement 62
#261 opened on Jun 9, 2011 by jgm

If you're lucky, someone will have a workaround. If you're really lucky, your problem will already be fixed in a later version of the project!

Wrapping it up

When you can't debug a problem, you'll have to do some research.

If you find an answer on a blog or StackOverflow, don't accept it right away. Use it as a chance to understand what caused the problem and why the solution works. Copying and pasting without understanding leads to messy, buggy, inconsistent code.

You can learn more from the project maintainers directly. Find the GitHub page, find the IRC channels, and find the mailing lists. There's nothing like getting your questions answered by the person who wrote it.

Finally, practice reading code. Learning to read and explore code will teach you things you won't learn anywhere else. And when you get good enough at reading code, you'll be able to solve problems you might have thought were impossible before.

But remember, reading code isn't like reading anything else. It's about debugging and exploration. Don't just scan it, play with it.

Exercises

When you're learning Rails, it's natural that you'll run into some problems that you won't be able to solve on your own. So can you figure out how to get into the Rails IRC chatroom so you can ask questions? Or how to read the Rails mailing lists?

Say you had an ActiveRecord model that looked like this:

```
class Relationship < ActiveRecord::Base
  enum type: {
    sibling:     0,
    child:       1,
    parent:      2,
  }
end
```

But when you call `Relationship.new`, you get this error:

```
SystemStackError: stack level too deep
```

This is an error in Rails. But has it been fixed yet? Can you figure out what caused it? (Hint: good keywords for this problem would be "enum" and "stack" or "SystemStackError"

The most common Rails errors, and how to fix them

When you work in Rails, there are some errors you'll see all the time. Here are a few of the most common, and how to fix them:

- `NoMethodError: NoMethodError: undefined method 'selcet' for [1, 2]:Array`

This means that the method you tried to call on an object doesn't actually exist on that object. Usually, it means you forgot to define the method, or typo'd it, or forgot to `require` a file. In this specific example, someone seems to have typo'd `select` as `selcet`.

- `NoMethodError: undefined method `method' for nil:NilClass`

  This is a special case of the first error. This means that you called a method on something you expected to be an object, but it was `nil` instead. These are some of the most annoying errors to fix, because it can be really hard to figure out *why* something became nil. Sometimes they're easy, but other times they need a ton of investigation.

- `SystemStackError: stack level too deep`

  This means a method of yours kept calling itself. So, for instance, you'd see this if you had a method `find` that called `find` somewhere inside it. Usually, these are either typos, copying and pasting code from somewhere else and forgetting to change the name, or bugs in recursive functions.

- `ActiveRecord::RecordNotFound: Couldn't find Bug with 'id'=102`

  These will usually tell you exactly *why* the record wasn't found, so they're usually easy to debug.

- `ActiveModel::ForbiddenAttributesError (ActiveModel::ForbiddenAttributesError)`

  This means that you either had a typo or a missing key in your `params.permit` call in your controller. You were trying to set a parameter, but your controller wasn't allowing it to be set.

Putting it all together

Running into errors can be stressful. But it's a necessary part of writing software. And there's nothing like running into a problem with something to help you understand and remember it.

In order to solve a problem in your code, you have to figure out where the problem came from. Exceptions and backtraces will usually show you where you had a problem, down to the specific line number. Usually, you'll want to:

1. Look at the exception message, if there is one, to understand the problem.
2. Scan the backtrace from top to bottom.
3. As you scan, look for your code (code in `app/` or `lib/`).
4. Use those lines to navigate to the place in your code where the error happened.

Once you find out *where* the problem happened, you have to dig into *why*. This is where you can start debugging.

Debugging is all about understanding your code, and figuring out where your code differs from how you imagined it. That usually involves looking at the values of variables and running code.

You can use `puts`, logs, and debuggers to discover problems. With a little setup, a good debugger like the `byebug` gem can help you find issues quickly. But don't underestimate `puts`. It can succeed where debuggers fail. It's a lot more work, though, so I usually use a debugger if I can.

You can use `puts` to do things like:

- Printing out the values of variables
- Keeping track of when methods were called
- Binary searching to narrow down where a bug could be hiding (like playing the “guess a number” game)

All three of these have helped me figure out some truly disastrous bugs.

You don’t have to `puts` everything. You can discover a lot from the Rails logs, too! If you’re having problems with handling parameters, loading things from the database, or rendering views, the Rails logs will help out a lot. And you can print your own stuff to this log using `Rails.logger.info`.

When you still can’t figure it out, you’ll have to do some research. Dig into the code, search for other people with the same problem online, or try to find out where the project maintainers hang out and talk to them.

There’s no process for fixing errors that works every time. So you should attack problems with a few of these techniques at once.

Once you get really good at this, you won’t fear errors anymore. And you’ll build more interesting things, because you’ll know what to do if they break.

Chapter 6
Keeping up with the Rails community

Keeping up with the Rails community feels like running on a treadmill. Except this treadmill makes you gain weight instead of losing it.

There are so many articles to read and videos to watch, and they all seem to tell you that the last thing you learned is wrong.

It could be a video about how your controllers are organized badly. A new version of a gem that breaks all of your code. A series of posts about tearing out all your views and using Angular to make your apps more fun to use. (Wait, I have to be a Rails *and* Javascript *and* Angular expert before I can write apps now? And how many build systems do I really need, anyway?)

If it feels like it never ends, that's because it doesn't.

But I've found some good ways to skip the stress of keeping up, without missing the important stuff when you need to see it.

Avoiding the temptation of the new

When many people learn something like Ruby, they'll dive right into the community. They'll subscribe to a bunch of subreddits[73] and check them a few times a day. They'll crawl Hacker News[74] for new and interesting projects. They might subscribe to Ruby-Weekly[75], and the Thoughtbot[76], Ruby Rogues[77], and Ruby on Rails[78] podcasts.

This isn't a bad thing. These are all great places to get Ruby news. But finding the news isn't the problem. It's filtering it, focusing on the right things, and keeping yourself from getting overwhelmed by the crazy amount of stuff that comes out every single day.

New stuff is inherently interesting. It's like candy. But most of the new things you see won't help you right away. Exploring it all isn't progress. It's just motion.

So you'll have to separate the things that are interesting because they're new, from the things that are interesting because they'll help you get work done.

Building a good mental filter

How do you find the useful things in a pile of new things?

73. http://www.reddit.com/r/rails
74. http://news.ycombinator.com
75. http://rubyweekly.com
76. http://giantrobots.fm
77. http://devchat.tv/ruby-rogues/
78. http://5by5.tv/rubyonrails

When I see an article that looks interesting, I ask myself a few questions. These let me know if I should read more:

1. Is this something I need to know right now?

   Sometimes, you'll be lucky enough to see a blog post on reddit that solves the exact problem you're thinking about. This one is obvious – if it'll make your life easier right now, it's worth reading more.

2. If I knew this a year ago, would it have made my life easier today?

   It seems like I see the same problems over and over, way more often than I'd expect. If you ran into a problem on your last project, chances are you'll run into it again. And if there's a new, better way to solve it, it's probably worth learning.

   This doesn't mean you have to update all your old code with your new knowledge. Just knowing it for the next time will make it worthwhile.

You can still explore tutorials and videos outside of these two categories. I certainly do. But you'll learn much more efficiently if you use these ideas to help you decide what's worth the time.

Push vs. Pull

There's still a problem, though. News sites become an addiction.

This still happens to me: I'm working on a project, and I get stuck. It might only be for ten seconds. But before I know it, I've typed `news.ycombinator.com` into my address bar. And for the next hour I'm jumping between it, reddit, and twitter, hoping every time that something new and interesting will show up, so I don't have to face the fact that I don't know what to do next.

"Oh, hey, apparently someone wrote an article about semantic versioning! I wonder if anyone's commented on it on Hacker News. Oh man, there's so many people arguing about that here. I should follow this URL in the comments and read more about it. Huh, that post talks about a conversation I think I missed. I'm gonna track it down and see what's up."

And then it's somehow midnight. But it's OK. I'll just work on my project tomorrow.

Going to sites like Hacker News and reddit is the "pull" model of getting tech news. You're the one digging it up. But these days, I've been using the "push" model more and more. I've been subscribing to email newsletters, feeds, and podcasts. Things that get delivered to *you* instead of you looking for *them*.

Why is this better?

Tech news sites are addicting because of the Fear of Missing Out[79]. They move so quickly, that if you're not checking them, you might not see that one article that would finally teach you everything you need to know! And everyone will know it except for you, and you'll look bad when you talk about `will_paginate` while everyone else is using `kaminari`.

And every once in a while, you *do* discover that thing, and it feels great, and it feeds a whole new cycle of checking and reading.

79. https://en.wikipedia.org/wiki/Fear_of_missing_out

When information gets sent to you, though, you'll still get the most important and the most popular articles. But you can take it in at a slower pace. You'll get all the same information, but it'll come with more time and reflection mixed into it.

Besides, if you miss a few articles, you're not falling behind. There's been over ten years of Rails news that you've missed already, but you don't have ten years of catch-up to do. You only have to learn the things that stuck around.

Wrapping it up

Chasing Rails news is a quick way to get overwhelmed without actually learning the important things. If you try to read everything that looks interesting, you'll quickly get frustrated and give up.

Instead, filter the information that you get and focus on just the things that'll be important to learn. But how do you know what will be important before you read it?

There are a few questions that help me decide:

1. Is this something I need to know *right now*?
2. If I knew this a year ago, would it have made my life easier today?

If you start reading something and you can't answer "yes" to those questions, it means that you can skip it for now.

But once you start skipping interesting-looking stories, or try to stop reading tech news entirely, it'll cause its own kind of stress. You'll feel like you're falling behind.

So avoiding all news isn’t the answer, either. Instead, let the best stories come to you. Subscribe to newsletters and podcasts. As a bonus, those stories will come with some extra time and reflection, which will help you avoid a lot of the Rails drama that explodes and quickly disappears.

If you find a few good newsletters and podcasts, keep a steady learning and practice schedule, and study things as they become important to you, you’ll make much more progress than those who constantly chase the news sites.

Challenges

For the next week, make a note of every time you realize you’re on a social news site chasing Ruby or Rails news. Count them. Then, try to bring that number down.

Subscribe to RubyWeekly and, for each story, use the two questions from this section to focus on just the stories that are important to you. That focus, combined with deep learning and practice, will be much more effective than constantly chasing the new.

When to give a new tool a chance

Follow the tips in the last section, and you’ll handle Rails news at a much better pace. But because you’re focused on just the information that’s useful to you right now, it’ll be hard to pick up new, unexpected knowledge.

Sometimes, you’ll want to ignore those guidelines, and read about the new stuff. But it’s hard to decide where to spend your limited time.

Something that's totally fascinating right now could be unusable a year from now. And when it doesn't get updated with Rails 5, you'll have to spend just as much time tearing it out as it saved you to begin with.

When you see new tools and techniques, how can you decide whether they're worth using *before* you learn them?

What indicates quality tools?

If a project doesn't have a good readme, examples, or even tests, you shouldn't depend on it until it does. Having a readme and tests are the basic measure of a maintainer who cares. And they'll come in handy when you get confused, or if something isn't working quite right.

Second, take a look at the API and examples. I'm happiest using a library when the API matches the way I would have written the code.

Finally, look at how popular the library is and how often it's updated. ruby-toolbox[80] is the best place to get those stats.

You'll usually run into two problems when you integrate a gem into your app: upgrades will break it, or you'll run into things it can't solve yet. Both can be helped by an active development team.

If a gem stops making sense in your app, don't worry about ripping it out. Do it as soon as you can. The code you write changes based on the gems you use. So if you're

80. http://ruby-toolbox.org

unhappy with the way a gem works, it's better to remove it before it warps the rest of the code you write.

What about new techniques?

Techniques are a lot easier to try than libraries. But they can be more of a problem when they *don't* work, since they'll affect a lot more of your code. That makes it harder to fix if they make your code worse.

So, when I see a new technique that looks interesting, I try it out on a new small project. Then, I take the technique as far as it'll go. For instance, after reading Practical Object-Oriented Design in Ruby[81], I tried following Sandi Metz' Rules[82] for an entire app. Same thing with Avdi Grimm's Confident Ruby[83]. It'll give you a taste of how the technique would work in real code.

Unfortunately, a lot of good programming techniques won't be as helpful in small projects.

For these, try refactoring one of your existing projects.

First, create a new branch for your experiments, using `git checkout -b some_branch_name`[84], so you don't wreck your code if you don't like the new look.

81. http://www.poodr.com
82. http://robots.thoughtbot.com/sandi-metz-rules-for-developers
83. http://www.confidentruby.com
84. You can learn more about Git and its commands from the excellent Pro Git[85].
85. http://www.git-scm.com/book/en/v2

Then, rewrite a class or a controller using the techniques you just learned, and decide which code you like better (like DHH Ping Pong[86]). Show them to other developers and see what they think. New techniques should improve your code, so a direct comparison is the best way to judge.

The more the technique affects your code, the more time you should spend thinking about it. But those are usually the techniques that will be the most valuable.

Wrapping it up

When you discover a new tool or technique, it can be hard to decide whether it's worth learning more deeply.

I've found some ways to decide whether tools are worth using in your own apps:

1. Does it have a readme? Examples? Tests?
2. What is the interface like? Is it something that would fit into your app?

Techniques can be harder to evaluate. Sometimes, it's easiest to try them in a small app. Otherwise, do a refactoring on one of your main apps, and compare it to the code you had before. To compare it:

1. Make the change.
2. Look at the old code next to the new. (If you don't have a better way, printing it out works well!)
3. Ask yourself, which code do you prefer?

86. http://www.dhh-ping-pong.com

4. Ask a few others, which code do they prefer?
5. If the new way is an improvement, go with it.

Exercises

Visit http://www.reddit.com/r/ruby. Find the first story that refers to a gem. Use the guidelines for trying a tool above. If you had the problem it fixes, would you use that gem to fix it? Why?

Read Sandi Metz' Rules[87]. Could you find a place in your app where you could try them out? Once you do, compare your old code with your new code. Which do you like better?

Staying up-to-date with your libraries

As your apps get bigger and you add more dependencies, keeping up with changes can be challenging. Especially if you depend on things as big as Rails. But you don't have to read ten "What's new in Rails 4.2" articles to keep up.

Start with the CHANGELOG

Most popular gems have CHANGELOG files in their git repository, like bundler: https://github.com/bundler/bundler/blob/master/CHANGELOG.md. These will help you catch up on the big changes from version to version. Usually, they're just a short

87. http://robots.thoughtbot.com/sandi-metz-rules-for-developers

summary of each major change. But they give you a starting point, so you can do more research on interesting changes.

What kind of research do I mean? Changelogs usually reference bug numbers on GitHub. If you see an entry in a changelog that has a bug number attached, you can find the bug in the project's Issues[88] to understand what changed, how it changed, and why it changed.

Usually, you can find these changelogs by searching google for `bundler github` (if I'm looking for the bundler changelog). They're usually on the first GitHub page you see.

If there isn't a changelog, you can also look at the project's README or the project's wiki on GitHub. But since those aren't designed to help you catch up from version to version, they aren't as helpful to go through.

Using up-to-date reference documentation

When you're working with gems, you'll also need to keep up-to-date reference documentation around. That way, you can look up method and class names and see examples while you're writing your own apps.

You can find API documentation for any version of any gem at rdoc.info[89]. But for even faster doc lookup, you should check out Dash[90] or Zeal[91].

88. https://github.com/bundler/bundler/issues
89. http://rdoc.info
90. http://kapeli.com/dash
91. http://zealdocs.org

I use Dash, so when I need to look up API docs, I hit option-Space, start typing, and all my gem documentation shows up instantly. It's a change in my workflow that's paid for itself many times over.

A few tips on Rails, specifically

Rails is a really big project, and the Rails contributors do a great job of maintaining changelogs and documentation.

The Rails guides[92] are good, and they're also built from the same git repository[93] as Rails, so they're always up to date.

If you just want to catch up to the newest version of Rails, the release notes are the best place to start. For example, here are the release notes for Rails 4.2: http://guides.rubyonrails.org/4_2_release_notes.html

I upgraded a library, and everything broke. What do I do now?

Version upgrades often mean breakages. This is not your fault, you didn't break it! But you'll still have to debug and fix it.

If you're lucky, the README or in the CHANGELOG will tell you about broken features or things that aren't supported anymore. Usually, you'll just have to change the type of object you're passing into a method or call a different method. If you're un-

92. http://guides.rubyonrails.org
93. https://github.com/rails/rails/tree/master/guides/source

lucky, you'll have to dig into the library and debug it, using the techniques in Chapter 5, to find out where the problem is.

Wrapping it up

When you use gems in your project, you have to pay attention to changes in those gems. The best way to keep up with changes is to track down the project's CHANGELOG file. You can usually find those files in the root of the project's git repository.

You should always keep docs on the newest version of your libraries close-at-hand. I use Dash[94], but rdoc.info[95] is also a good, if slower, choice.

Rails does a great job at keeping good lists of changes. The best place to look for the changes in each version of Rails are the Release Notes. The release notes are part of the Rails guides[96], and you can find the notes for each version there.

Exercises

Can you find the CHANGELOGs for Rake and Minitest? What changed in the newest version?

Find the documentation for Rake. What does the `desc` method do?

What is the most interesting change you can find in Rails 4.2?

94. http://kapeli.com/dash
95. http://rdoc.info
96. http://guides.rubyonrails.org

Putting it all together

The Rails community moves really fast. Keeping up is not only challenging, it's overwhelming. If you want to learn new things about Rails by following the community, you have to build good filters.

You can cut down on the noise and meaningless drama if you let the news come to you instead of seeking it out. Subscribing to RubyWeekly and Ruby podcasts can deliver the best news to you, and you can follow it on your own schedule.

When you see an article that looks interesting, ask these questions:

1. Is this something I need to know *right now*?
2. If I knew this a year ago, would it have made my life easier today?

If you can't answer "yes" to either one, you might want to skip it.

If you always skip stories that don't fit into those categories, though, you'll have trouble discovering brand new things. But you still need to make sane decisions about when to bring those new ideas into your own code.

To bring a tool (library, gem, framework) into your app, it should:

1. Have a readme, examples, and tests.
2. Have a reasonable API that smoothly fits into your app.

For techniques, directly compare your code using the technique to your code as it was before:

1. Make the change.

2. Look at the old code next to the new.
3. Ask yourself, which code do you prefer?
4. Ask a few other people, which code do they prefer?
5. If the new way is an improvement, go forward with that.

When you add a new gem into your project, you have to keep up with the changes to that gem. Otherwise, upgrading could badly break your app, and you'd have to figure out why. Besides, if you never pay attention to updates, how will you know which cool new features you can use?

You *could* read a bunch of blog posts about the updates to your gems. But it's usually better to go directly to the source, by reading the project's changelog. You can usually find a changelog in the root of the project's git repository. If you can't find one, you might have to dig through git commits and issues. Or just update the gem and hope for the best.

Finally, limit the amount of new shiny Rails news that you chase, and focus on really learning the few things that are important to you. You'll learn much more quickly, and you won't be overwhelmed by things that don't matter.

Chapter 7
How do you learn Rails when you don’t have the time?

Your TODO list is so long, you’re about to toss it into the fireplace. So where will your practice even fit in?

If you have a full-time job, you can probably only practice Rails in the early morning and late evening. Right now, though, you probably aren’t spending that time just staring at a wall. You’re spending time with your family, or running errands, or doing one of the many other things you need to do to keep yourself and the people around you happy.

So, you’re going to have to carve off some time you’re already using. Luckily, it doesn’t have to be much.

Some days, you might be so excited about a new idea that you can't wait to get home and try it out. Other days, the last thing you want to do is sit in front of the computer for a few more hours. Or you might be exhausted, and can't keep your eyes open for long enough to get through another day of practicing Rails.

Then you skip for just one day, and then two days, and all of a sudden it's been two weeks since you spent time growing your Rails skills.

You can't rely on motivation to help, because it won't always be there. And trying to learn Rails during random spare hours won't work. Those sessions are just too easy to miss.

To build a new skill, you have to build a schedule.

If you say, "Monday, Tuesday, and Thursday nights from 10-midnight is my time to work on learning Rails", it'll quickly become a habit. Something you do because it's *what you do*. You'll start to protect that time, because you know it's coming. And you'll have fewer excuses to skip it.

In the beginning, think short instead of long. Aim for 40 minutes a day, a few times a week, to start.

40 minutes isn't too hard to find. If you skip just one episode of whichever hour-long show you're marathoning right now (or maybe that's just me?), that's your time right there.

If you really can't find the time anywhere else, staying up 40 minutes later at night usually won't be too painful. Or you could try waking up a little earlier each day.

40 minutes has another interesting property. It's long enough that you can finish a complete task, like learning some new methods, building a sample app, or working

through an exercise in a Rails book. This is great, because you can really feel like you accomplished something.

But 40 minutes is also short enough that it'll surprise you when it's over. You'll start the next task and leave it unfinished.

When you leave something unfinished, it'll stay in the back of your mind. You'll unconsciously look for closure.

For example, say I ran out of time while I was learning the Rails form helpers. At some point during the rest of the day, I'll be browsing news sites, reading email, or working on work projects. And I'll accidentally come across information or ideas that teach me things about forms, or prompt new ideas about how I could better structure my form code, or how I could handle parameters in a better way in my controller.

It feels like magic when this happens. It's totally crazy. But it does happen! It's all information you were going to see anyway, you just might not have remembered it. But leaving projects unfinished means you'll pay more attention to the things that'll help you finish.

And that's not the only benefit.

Have you ever had to leave work right in the middle of what you're doing, and you think about it for the rest of the night and the next day, until it's done? Or investigated a bug for hours, and ran out of time before you could fix it? As much as I like to leave work at work, my wife can tell *every single time* I had to interrupt what I was doing because I had to head home. My head's just not all there.

Finishing is *so* satisfying. So if you're having a motivation problem, you can leave projects unfinished as a brain hack to get you started the next day. Don't find a stopping point. You can even walk away from your computer in the middle of a word.

Keeping your schedule consistent

When you first try to keep a schedule, it'll feel weird. Really weird. To me, it feels like I'm just pretending to be professional, or I'm just acting like someone who has it all together. That feeling starts to really hit around the fourth or fifth day, and it goes away after about three weeks. It's totally normal, but it can be dangerous.

Don't listen to that voice that says, "You're just pretending. You're not actually accomplishing what you think you're doing, so what does it matter if you skip today? You'll just get the same amount of nothing done tomorrow." If you start taking it seriously, it'll ruin all the time you've put into building your habit so far.

Change your routine is rough, and you'll naturally resist it. Our daily routines seem normal to us, that's what makes them routines. So it takes a lot to change those routines, since you're breaking habits that have taken *years* to form, in just a few weeks.

I'm serious, the first few weeks of keeping a consistent learning schedule are *really* hard. It's the hardest part of this entire learning process.

So for those first weeks, focus on consistently spending some amount of time on learning Rails, and focus less on how quickly you're learning or what you're getting done. Even if you don't feel like you're making any progress, if you were able to make the time you scheduled, you've already won.

Morning or evening?

If you have a day job, and you’re trying to set aside some time to work, you usually have two options: Work in the morning before your job, or work in the evening after your job. Neither’s perfect, and it mostly comes down to your preference.

By the time evening rolls around, you might be excited to start. You could have been thinking about getting to work on Rails all day, and can’t wait to get home to try something you just learned. And you don’t really have a set deadline (except sleep), so you can let your motivation carry you beyond the time you set aside.

But you might also be drained. If you’re tired, it’s easy to convince yourself to skip it, “just this once.” And after a frustrating day at work, you’ll start to tell yourself that you’ve had a rough day, you *deserve* to just get a good night’s sleep, because you can always catch up tomorrow.

It’s also easy to push until later. “If I watch one more episode, I’ll start as soon as it’s done. It’ll only be 10 minutes late.” But before you know it, you’re an hour late, you’ve destroyed your sleeping schedule and you’ll pay for it tomorrow, when you’re drained again and you’ll skip again.

Mornings can be quieter. It’s easier to start on time, especially if you turn off the snooze feature on your alarm.

But you might feel brain-dead and uncreative in the morning, which can be killer if you’re learning and practicing creative work like learning Rails. And it’s hard to wake up early until you get used to it.

I used to try to write and code in the evening, but I skipped it way too often. I doubted I’d be able to make the mornings work, since I’ve always thought of myself as a night person. But I heard a lot of people I trust and respect suggest trying to wake up a little earlier for a week or so. I did, and it was hard, and I felt totally unproductive.

But somehow, when I *measured* my productivity, I found out I was twice as productive in the mornings as the evenings. This is crazy, because it felt like the exact opposite!

Still, rather than saying "Mornings are better!", I'd suggest trying out a week or so of scheduled evening time, and a week or so of scheduled morning time, and measure what you actually get out of it.

Wrapping it up

All of us are busy, and we don't have a ton of free time to spend on things like learning Rails. But learning Rails will take time, and it has to come from somewhere.

Whenever you try to build a new skill, consistency is much more important than the amount of time you spend. Your biggest problem will be skipping the time you set aside, "but just for today," and I'll tell you from experience that that quickly turns into skipping weeks, and months.

Once you're consistent enough you form a habit, motivation won't be as much of an issue. For example, I don't even have to get motivated to write my articles anymore. It's just become something I do, so I do it.

At first, all you need to do is find 40 minutes, a few set days a week, to spend. 40 minutes is long enough to finish one task, but short enough to leave other tasks unfinished, which is another way to keep you motivated to return.

Forming this routine is going to be tough, and will feel weird at first. But keep at it, and soon you won't even notice you're doing it.

Exercises

What specific time on which days are you going to spend this week? Monday, Thursday, and Friday from 10:30–11:10pm? Sunday, Monday, and Thursday from 6–6:40am? Block it out on your calendar, and set a reminder. And hit a few of them in a row!

What to do when your motivation abandons you

I've always been a dabbler.

Before I really learned Emacs, I gave up on it three times. justinweiss.com[97] is my fourth attempt at a blog, and it's the first one that lasted longer than a month or two.

When I first start something, I'm *so* excited about it. I read everything. I subscribe to everything. I find people I can chat with about it, and I put all of my time and focus into it.

But once the newness wears off, I hit a wall. I stop putting the time into it. I start to get frustrated because I'm not seeing progress, and I start to wonder if this is something I really want to do anymore.

I just recently realized how important this was to figure out, so I spent the past year focused on increasing my motivation[98]. And with a few really minor changes, I've become much better at handling those motivation swings.

97. http://www.justinweiss.com
98. This is kind of meta – I used motivation to learn motivation, before my motivation ran out!

So how do you make sure you can get stuff done when you need to?

I've found a few tricks that will help.

Habit

The first trick is what I talked about in the last section: doing something consistently enough that it becomes a habit. Once you get that consistency down, motivation won't enter into it, because it'll just be something you do.

Of course, answering "How do you get motivated to study Rails consistently?" with "Study Rails consistently!" is less than helpful. You'll have to use some of the other tips to get these habits started. But this is still good to remember, since it'll help you see where you're trying to get to – creating a solid habit.

Soon, it'll be easier to keep going than it will be to stop. And every time you show up as scheduled, you'll be one step closer to that point. So if you're not motivated by the work one day, you might be motivated by *creating a habit of work* instead.

Pre-prepare

Some days you won't feel like starting something because the task isn't ready yet. For example, I might not feel like writing an article, because I'm not sure what I'm going to write about.

To fight this, do a little bit of preparation in advance. This preparation can be its own task!

When you plan your next step, it's motivating on a few totally different levels:

- By separating the decision about *where to start* from the decision about *what to do*, you end up with two smaller, clearer tasks. Each of these tasks is easier to start than your bigger, fuzzier task.

- It creates that unfinished loop in your mind. I mentioned how powerful unfinished loops can be earlier, and they're just as helpful here.

  In the time between preparing for a task and starting it, you'll feel like something's missing. And once it's time to finish that puzzle, you'll feel determined to actually do it.

Processes

Having processes to follow, like those I've included at the ends of Chapter 1, Chapter 2, and Chapter 3, can also help you keep motivated.

When you see the next step right in front of you, it's often smaller than you imagined it was. It's easier to remember where you are, where you're going, and what you need to do to get there. You'll spend less time wondering what to do next, so you can start on it right away.

The Seinfeld method, and where it breaks down

If you talk to software developers who've tried to build a skill, whether it's contributing to open source, studying a new language, or learning an instrument, you'll hear about the Seinfeld method[99]:

99. http://lifehacker.com/281626/jerry-seinfelds-productivity-secret

> *[Jerry Seinfeld] told me to get a big wall calendar that has a whole year on one page and hang it on a prominent wall. The next step was to get a big red magic marker.*
>
> *He said for each day that I do my task of writing, I get to put a big red X over that day. "After a few days you'll have a chain. Just keep at it and the chain will grow longer every day. You'll like seeing that chain, especially when you get a few weeks under your belt. Your only job next is to not break the chain."*
>
> *"Don't break the chain," he said again for emphasis.*

The Seinfeld method has helped me build a lot of habits, and I credit it for changing my life in a lot of great ways. You really should try it.

There's one problem with it, though. What happens when you *do* break the chain?

When you miss a habit and break a chain, you lose all that built-up motivation that helped you keep the streak going. And you lose it at the exact time you need *extra* motivation to build the streak back up.

Once I break a chain, I fumble around at the 5-7 day mark for a while, and never again touch the record I had before. Five days in a row is still good, but it should be better.

Recently, I've been using Beeminder[100] instead. I still have a goal, like "Write 1000 words per day," or "Contribute to open source 7 times this week." You can see my writing goal for this book here[101]. But once I break my habit, it charges my credit card

100. https://www.beeminder.com
101. https://www.beeminder.com/justinweiss/goals/write

an amount that increases every time I fail. This keeps me on track to start another long streak. (Imagine how expensive failing every five days would get!)

The early motivation Beeminder gives me is usually enough to keep me moving forward, and will give me a little extra kick when I start to fall off track.

Putting them together

You can't overdo these tricks. Not all of them will work all the time. So combine them, and use them often.

For example, if you're trying to finish a tutorial series on Rails, start a streak. Try to watch a half of a video per day, and keep the streak going. An hour before you start the video, read the description, get out your notes, and prepare for what it's going to teach you. Use the process from Chapter 1 to build a tiny Rails app exploring what you just learned. And keep doing it until your learning becomes a habit.

Wrapping it up

Motivation is one of the hardest things you'll have to manage as you pick up any new skill. But handling your motivation swings is something you have to do. Most of your successes and failures have more to do with motivation than they do with your intelligence, your tools, and the time you have to spend.

I have a few tricks you can use to keep motivated, even when you're just not feeling it.

First, build a practice schedule. Then, plan what you're going to do during that time. Use processes so you don't get lost and distracted. And finally, have a way you can see your streak of showing up, and don't break it. If you combine all of these, your streak will turn your schedule into a habit, and you won't need these tools as much anymore.

So, what does your schedule for learning Rails look like today? What are you going to do tomorrow?

Exercises

Sign up for either a Seinfeld calendar app (I use Habit List[102] for the iPhone, but you could just buy a paper calendar and use that instead) or Beeminder (it's free unless you fail). Keep track of your Practicing Rails streak, and don't break the chain. It'll get hardest around the one or two-week mark, so expect that and try not to get discouraged. After three weeks to a month, it'll be easier to do it than it'll be to skip it.

Putting it all together

Time and motivation are two things you have to manage if you want to successfully master Rails. But they're also really hard to get the hang of.

Most of us are pressed for time. But small bits of time are easier to carve out of an already full day than big chunks. And small bits add up, when you spend them con-

102. https://itunes.apple.com/us/app/id525102168

sistently. So the fastest way to put the hours into learning Rails is to schedule small chunks of time that you can hit consistently.

To start being consistent, you need motivation. And when you don't feel motivated, you'll need some hacks. You can:

- Build a practice schedule
- Plan what you're going to do, an hour or two in advance of doing it
- Use processes, so you don't have to decide what to do next
- Keep consistent with a habit tracker, calendar, or Beeminder

Using one of these will help out. Using all of them will help more. But the most important thing is to use these hacks to keep your schedule consistent. That's how you'll learn the most in the shortest amount of calendar time.

Once you're consistent for a while, your schedule will become a habit. And at that point, you won't need much motivation, because you'll just do what you need to do.

Because it is *what you do*.

Epilogue

You’ve seen a lot of ideas in this book. Here are some of the most important:

- As soon as you want to learn something, try it out.

  When you’re just reading about code, you have no real reason to remember it, so you don’t. But when you build real apps, you’ll understand much more about it, and you’ll have a hope of actually remembering it.

- Start backwards.

  When you write an app, start from the UI down. Allow your vision of the feature to guide your development. It’s easy to know if you’re building the right thing when you start from the end and trace back to the beginning. And UI sketches and HTML views are a lot easier to think about than abstract data models.

- Keep it simple, and add complexity later.

  The most frustrating struggles come from running into problems you don't know how to solve. You can skip these problems by avoiding new things until you understand the old things. This goes for everything from gems and libraries to patterns and object-oriented design principles.

- Systems, not motivation.

  If it takes consistent practice to get good at Rails, not being motivated to practice is going to be your biggest problem. You can't rely on motivation every day. Instead, set up systems, habits, and processes, so you don't *have* to motivate yourself to work.

- When you feel yourself procrastinating or stressed about something, break it apart.

  Large, fuzzy tasks are killer. If the next thing you want to do is tiny, and you can start on it in the next five minutes, you probably will. If it's big and vague, you'll put it off until you know how to start it. Which will probably be never.

  This goes for skills, too. Master tiny skills until you don't have to think about them, and you'll have more focus to put toward the things you don't know as well. If you try to learn large fuzzy skills, it'll take longer, and you'll be worse at all parts of it until it's mastered.

Ruby and Rails can be tough to learn. There's a lot there. Some of it's complicated. But you'll get it, I know you will. And as you face all the difficulty, the studying, the practicing, and the learning, you'll start to realize how worth it it all is.

Once you master Rails, you'll have a go-to language and framework for sketching out your ideas. You'll turn your thoughts into working code, and that's the most powerful way to build your development skills. It's a lot of fun, too!

So keep experimenting, keep breaking things, and stay curious.

Thank you so much for reading my book. I really hope you enjoyed it, and that you learn to love writing new things in Ruby and Rails as much as I do.

Before you finish, I have one more favor to ask: While you learn Rails, email me (my address is justin@justinweiss.com). Let me know how you're doing, and keep me updated on your progress. Seriously, I love talking about this stuff, so it'll be great to hear from you.

The best ways to continue mastering Rails

I read *a lot*. I don't expect you to read as much as I do. But I have some favorite books and websites, the ones I recommend all the time.

These are my most frequent recommendations in one place.

If there's an area you'd like to spend more time with, go after these books first.

Ruby

Eloquent Ruby[103], by Russ Olsen

Eloquent Ruby won't just teach you Ruby, it'll teach you to write great Ruby that looks like a Rubyist wrote it. If you want to write idiomatic Ruby, this is your book.

103. **http://eloquentruby.com**

Programming Ruby 1.9 and 2.0[104], by Dave Thomas, with Chad Fowler and Andy Hunt

This is another great intro to Ruby, and I actually learned Ruby from an earlier edition of this book. The second half of the book is an amazing reference guide to Ruby's core libraries. It's so good that it makes this book worth owning all on its own.

Practical Object-Oriented Design with Ruby[105], By Sandi Metz

Once you start to notice that your code is growing hard to work with, or you don't quite understand it, or you break your tests every time you change something, you have to read this book. It's full of extremely good advice on how to organize your code, using practical, real-world examples.

Confident Ruby[106], by Avdi Grimm

The more code you have to keep in your mind at once, the more stressful and frustrating it is to read it.

Confident Ruby focuses on writing simpler, more straightforward code, code that's not always thinking about edge cases and `if`s. This book is an absolute joy to read, and has had an enormous impact on how I write Ruby code.

104. https://pragprog.com/book/ruby4/programming-ruby-1-9-2-0
105. http://www.poodr.com
106. http://www.confidentruby.com

Rails

Learn Ruby on Rails[107], by Daniel Kehoe

If you're a total beginner, this is a friendly, gentle introduction to Rails. Daniel explains terms and patterns clearly, and does a great job making a complicated framework as easy to understand as it can be.

It doesn't go into as much detail as the next two books, but it's a great way to get prepared to read your next Rails book.

Agile Web Development with Rails 4[108], by Sam Ruby, with Dave Thomas and David Heinemeier Hansson

I learned Rails with the first edition of this book, and I still buy each new edition as it comes out.

I love the structure of this book. The first half introduces you to Rails by walking through a sample app, and the second half solidifies that knowledge by looking at each part of Rails in depth.

It also covers Rails as it comes out-of-the-box, so you don't have to worry about getting other gems set up.

107. http://learn-rails.com/learn-ruby-on-rails.html
108. https://pragprog.com/book/rails4/agile-web-development-with-rails-4

Rails Tutorial, 3rd Edition[109], by Michael Hartl

The Big One. A lot of people have a love-hate relationship with this book – it covers **everything** you need to build a real Rails app, but that also means it can be hard to get through.

Still, it's a great companion to this book: Rails Tutorial will go into a lot of different parts of Rails and web development, and Practicing Rails will help you take what you learn from Rails Tutorial and use it in your own apps.

With the techniques you learn from Practicing Rails, you'll avoid a lot of the overwhelm you'd otherwise get from Rails Tutorial, because you'll be confident you can study anything complicated on your own.

The Rails Guides[110]

The Rails Guides are detailed, efficient, and always up-to-date. They can be a little hard for brand new developers to get into, but they're the place I go back to when I have questions about Rails.

109. https://www.railstutorial.org
110. http://guides.rubyonrails.org

Testing

What Do I Test?[111], by Eric Steele

Eric and I share a lot of the same philosophy when it comes to testing. And he's a really funny guy.

So if you want a good, short intro on how to *really* start testing your Rails apps, check this one out.

Working Effectively with Legacy Code[112], by Michael Feathers

Testing gets easier the more often you do it, so learning enough to get started will get you far. But eventually, you'll find some areas you have no idea how to test. Or you'll get dropped into really bad code without any tests, and won't know how to get it tested enough to refactor.

Working Effectively with Legacy Code is the best book I've found for testing what otherwise seems untestable. When I was at Microsoft, I used it to figure out how to test printer driver installation. There's not much you'll see in Ruby and Rails that'll be harder to test than that.

111. https://whatdoitest.com
112. http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052

General Programming and Open Source

The Pragmatic Programmer[113], by Dave Thomas and Andy Hunt

The Pragmatic Programmer is still my favorite book on software development. A lot of what's in there now seems obvious, but that's only because of the impact it's had on the industry.

It's full of information about a lot of different programming topics, and it'll teach you to think about how you can improve your work. And that's the fastest way to make yourself a better programmer.

Pro Git[114], by Scott Chacon and Ben Straub

I didn't get much into source control in this book, but you'll soon want to learn it. Pro Git is the book I recommend to developers new to Git. It's comprehensive, has great examples, and you can read the entire thing online.

The last chapter in the back teaches you to build Git from first principles, and if you read and understand that, you'll know Git better than most other developers.

Build a Ruby Gem[115], by Brandon Hilkert

Build a Ruby Gem is about writing your own gems, but it's really about creating your own open-source projects.

113. https://pragprog.com/book/tpp/the-pragmatic-programmer
114. http://www.git-scm.com/book/en/v2
115. http://brandonhilkert.com/books/build-a-ruby-gem/

Learning how to start is one of the toughest part of creating open-source projects, and a gem is a great way to get there. Build a Ruby Gem walks you through the complicated parts of creating gems, so you can enjoy the coding and start building your open-source résumé.

Smalltalk Best Practice Patterns[116], by Kent Beck

When you organize your code, you'll often go back to the same few patterns all the time. This book is the best I've found to teach you the patterns to organize your code that you'll use 90% of the time.

It's not about Ruby, so the code samples might be tough to understand. But the information in it is so great that it's worth whatever extra work you'll have to do.

Bonus non-software related book

If I had to pick a single book that's had the largest effect on me in the past year, it's this one.

The Power of Habit[117], by Charles Duhigg

If there's one thing that took me from sitting on the couch playing videogames and watching TV to posting a new article every week and writing an entire book, it's setting up and building good habits. If you want to become a great Rails developer in your spare time, you're going to have to do the same thing.

116. http://www.amazon.com/Smalltalk-Best-Practice-Patterns-Kent/dp/013476904X
117. http://www.amazon.com/The-Power-Habit-What-Business/dp/081298160X

The Power of Habit is a look into how habits can be formed and changed, with tons of practical examples. Like this book, it’s a multiplier – it’ll make everything else you learn go faster and easier.