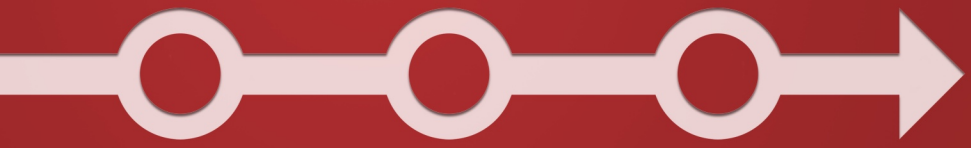

Practicing Rails

Learn Rails Without Being Overwhelmed

Justin Weiss

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 go over it all again, so did you spend that time learning for nothing?

This happens all the time. It's 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 being totally overwhelmed, and 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 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, it’s 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. Taking a chunk of terrible code and turning it into something clean and beautiful.

   You’ll have your own preferences. So as 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’s 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 it 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 two lines, you have an entire Rails app ready to use:

```
rails new 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]:

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

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

That is, I've generated *fifty-two* Rails apps, just for trying 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).

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.

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!

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 juggling it 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 it 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 it" *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 it in different situations. You'll get practice using it. And you won't have to worry as much about integration pain or errors.

How you can use Rails to learn Rails

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

- 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?

It's a concept for an app that you'll use to generate new tiny apps. It might be something you've built already, or something you've been thinking of 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 want to 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.

But 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 it, 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, but it'll usually go 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.
- Change it to be 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 it and put model code into the model, view code into the view, and so on. If you don't know where it 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]. 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.

It was really easy to try validation contexts out this way. Imagine how much more work it'd 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 it can be handy to have a few different app types in mind).

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`.

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 it. 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 it 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 it'll 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 it. 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, it probably won't matter.

Tests vs. the console

So, the console is great for messing with objects, but it can be hard to get those objects set up. 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). It even has syntax highlighting and some really nice other features[8]. And when you combine it 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 it. How can you break it? Look up the documentation. Are there any other features you could try?

Owning the things you learn

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://edgeguides.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.

You'll remember what you read about, because 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 so much 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 they 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.

A lot of the exploration comes from 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 you have about the feature, 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 it around as a reference 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

After you go through a few Rails tutorials, you'll know a lot about Rails. You might even have enough knowledge to rebuild the apps from the tutorial without looking at it.

But you didn't get into Rails to rebuild a tutorial's example blog. You want to build your own stuff! So why is it that when you try to build your *own* Rails app, it's always a lot harder than it should be?

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?

13. http://news.ycombinator.com

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.

Do you want to **build a new Rails app** from the ideas you have in your head?

Learn **step-by-step ways** to **test your code easily and efficiently**?

Know **what to do when your apps blow up**, and you have no idea how to fix them?

Know **which parts of Rails to learn *now***, and what you can put off until later?

Get just the *important* information out of the noisy Rails community?

And **keep your motivation fired up**, so you can turn mastering Rails into a habit you'll actually keep?

This is the end of the sample. **But you'll learn all these things (and more) in the full book of *Practicing Rails*.**

Get early access to *Practicing Rails* here.[14]

14. https://www.justinweiss.com/book