

[←Prev](#)

Submission

[Next→](#)

10 Rails: Basic Rails App



“Rails is the most well thought-out web development framework I’ve ever used. Nobody has done it like this before.”

— **James Duncan Davidson**

Overview and Purpose

In this checkpoint you'll start creating a new application using Ruby on Rails.

Objectives

After this checkpoint, you should be able to:

- Explain how Rails complements Ruby.
- Discuss the purpose of a README.

- Create a development database in Rails.
- Explain what the asset pipeline in Rails is.
- Deploy a Rails app to Heroku.
- Run a Rails server locally.

Bloccit

In this checkpoint, you'll start a new project, similar to [Reddit](#), named Bloccit.

Just like Reddit, Bloccit will be an app where people can post, vote on, share and save links and comments. Bloccit will have many features needed to make it a cool web app, but the first thing you need to do is design a basic user-interface (UI) as a foundation to build on.

Windows Users

If you are using a Windows machine you will need to log into your **Cloud9**. Once logged in, follow these steps to create a new workspace:

1. Click the "Create a new workspace" button.
2. Name your project "bloccit". Leave the Description field blank.
3. Choose "Public" for the git repository options. Leave the Clone from Git or Mercurial URL field blank.
4. Choose the "Rails/Ruby" template from the options below.
5. Click the "Create workspace" button.

Create Bloccit

The first step is to create a new Rails app. Run the `rails new` command in the `code` directory we created earlier:

Terminal

```
$ cd code  
$ rails new bloccit -T
```

The app name is `bloccit`. The `-T` option specifies that the app should not be created with standard test packages since we'll be testing our app with `RSpec`.

When we ran the `rails new` command, we should've seen a long output in your console. Among other things, `rails new` creates the Rails app structure. Open the project in the editor to explore the Rails app structure.

We should see a full Rails app structure. We'll explore the various directories as we progress through the Roadmap. We'll begin to make changes, but before you do, we'll want to establish the README, and update the database file and Git repositories

Create a New README

A `README` file should describe what the app or program does. It should also provide directions on how to install it, run tests, or anything else that another developer would need to know.

Open `README.md` and replace the existing content the following:

README.md

```
+ ## Bloccit: a Reddit replica to teach the fundamentals of web development and Rails.  
+  
+ Made with my mentor at [Bloc](http://bloc.io).
```

Feel free to change the style or content of `README.md` as you see fit.

Create the Development Database

Replace the contents of your `Gemfile` with the following:

Gemfile

```

+ source 'https://rubygems.org'

+
+ git_source(:github) do |repo_name|
+   repo_name = "#{repo_name}/#{repo_name}" unless repo_name.include?("/")
+   "https://github.com/#{repo_name}.git"
+ end

+
+ # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
+ gem 'rails', '~> 5.1.2'

+
# #1

+ group :production do
+   # Use pg as the production database for Active Record
+   gem 'pg'
+ end

+
# #2

+ group :development do
+   # Use sqlite3 as the development database for Active Record
+   gem 'sqlite3'
+ end

+
+ # Use Puma as the app server
+ gem 'puma', '~> 3.0'
+ # Use SCSS for stylesheets
+ gem 'sass-rails', '~> 5.0'
+ # Use Uglifier as compressor for JavaScript assets
+ gem 'uglifier', '>= 1.3.0'

+
+ # Use jquery as the JavaScript library
+ gem 'jquery-rails'
+ # Turbolinks makes navigating your web application faster. Read more: https://github.
+ gem 'turbolinks', '~> 5'
+ # Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
+ gem 'jbuilder', '~> 2.5'

+
+ gem 'thor', '0.19.1'

+
+ group :development do
+   gem 'listen', '~> 3.0.5'
+ end

+

```

At **#1** and **#2** we specify different databases for our Development and Production environments. We use `sqlite3` for our Development environment because it is an easy to use database perfect for rapid testing. Heroku only supports Postgres, so we use `pg` in our Production environment.

Because we changed your `Gemfile`, we must update our application with `bundle install --without production`. This command installs everything specified in the `Gemfile` and ensures that all of the gems work harmoniously. The `--without production` option ignores gems in `group :production`. These gems aren't needed or used in our Development environment. Our Production environment will automatically run `bundle install` when we deploy, and will account for gems declared in `group :production` at that point. On the command line, in the root `Blocxit` directory, type:

Terminal

```
$ bundle install --without production
```

Run the following command in your terminal to create the database:

Terminal

```
$ rails db:create
```

This creates a new local database for our app to use. We have to run this command after creating a new app, or after dropping an existing database.

The Asset Pipeline

As stated in the [Rails Guide](#):

The asset pipeline provides a framework to concatenate and minify, or compress, JavaScript and CSS assets. It also adds the ability to write these assets in other languages such as Sass and ERB.

The purpose of the asset pipeline is to make Rails apps fast by default while allowing developers to write "assets" (images, styles, and JavaScript, mostly) in a variety of languages.

Rails 4 requires some minor configuration changes to properly serve assets on Heroku:

Gemfile

```
...
group :production do
  # Use pg as the production database for Active Record
  gem 'pg'
+ gem 'rails_12factor'
end

group :development do
  # Use sqlite3 as the development database for Active Record
  gem 'sqlite3'
end
...
```

We added `rails_12factor` to the Gemfile; let's install it in our application:

Terminal

```
$ bundle install
```

Heroku provides a [detailed explanation](#) of the Rails 4 configuration changes.

Test Locally

Start the Rails server from your command line:

Terminal

```
$ rails s
```

If you're [using Cloud9](#), remember to start the Rails server with the `-p $PORT -b $IP` flags.

Navigate to [localhost:3000](#) to make sure the app is working locally.

Starting the web server with `rails s` will leave your terminal in an "open" state. That is, you won't see a command prompt until you stop the server. Open your app on localhost and view it next to the terminal where you started the server. Refresh the page on localhost, and you'll see the server logs update in your terminal. While you're running the Rails web server, the terminal logs all activity in your app.

Git and GitHub

Sign into your **GitHub** account and create a new repo named `bloccit`. You've already created a README, so make sure the "Initialize this repository with a README" is *unchecked*.

Commit and push your code up to your GitHub repo:

If your Rails server is still running, you can either stop it by pressing `CTRL-C` or leave it running and open a new Terminal tab. Either way, you'll need a Terminal prompt before moving forward.

Terminal

```
$ git init
$ git add .
$ git commit -m 'First commit and README update'
$ git remote add origin git@github.com:<user name>/<repo_name>.git
$ git push -u origin master
```

Use the URL from GitHub's instructions.

Reload the repo homepage on GitHub. It should display the content from `README.md` at the bottom of the page and you should see all of this repo's files.

Deploying to Heroku

It is time to deploy and share your app with the world. There are many choices for deploying and hosting Rails applications, and Bloc recommends the popular **Heroku** platform. Heroku makes it easy to manage and deploy Rails apps using the command line.

Sign up for a free Heroku account. Then install the **Heroku Toolbelt** for your OS. This toolbelt will allow you to run Heroku commands from the command line.

If you're using Cloud9, the Heroku toolbelt is already installed.

Log into your new Heroku account:

Terminal

```
$ heroku login
```

After you've logged in, create a new application in Heroku:

Terminal

```
$ heroku create
```

Because we did not specify a name with `heroku create`, Heroku created one for us.

We have a Production environment to `push` our application to. Type this command to push the code from the master branch of your Git repo to Heroku:

Terminal

```
$ git push heroku master
```

It may take a few minutes for the new application to propagate in Heroku. If you receive an error that says `Permission denied (publickey)`, [go here](#) to learn how to fix it.

Here is a video recap of how to deploy to Heroku:

You can see the web address for your application in Production by typing the following:

Terminal

```
$ heroku apps:info
```

Congratulations, you've deployed an application to your Production environment. For now, you'll receive an error message when you visit your Heroku URL. This is because the static index page **is not used in production**. We'll fix that soon.

Make sure you've added your **GitHub** account to your Bloc account page. Use the "Submit your work" tab to submit your first Bloccit commit for your mentor to review.

Recap

Concept	Description
<code>rails new</code>	<code>rails new</code> creates a new Rails application with the entire default Rails directory structure.
README	A <code>README</code> is a text file commonly distributed with a program. It contains information that describes what the program does, provides directions on how to install it, run tests, or anything else that another developer would need to know.
Asset Pipeline	The asset pipeline provides a framework to concatenate and minify, or compress JavaScript and CSS assets. It also adds the ability to write these assets in other languages such as Sass and ERB.
<code>rails server</code>	The <code>rails server</code> command launches a small web server named WEBrick, which comes bundled with Ruby.
Heroku	Heroku (pronounced her-OH-koo) is a platform-as-a-service (PaaS) that enables developers to build and run applications entirely in the cloud.
Gems	Gems are Ruby libraries that can be used to extend or modify functionality within a Ruby application.
	Rails ships with three environments: "Development", "Test", and

Rails Environments

"Production". These environments are used to tell your app to behave differently in different circumstances, primarily by setting different configuration options and variables.

How would you rate this checkpoint and assignment?



10. Rails: Basic Rails App

Assignment

Discussion

Submission

[←Prev](#)

Submission

[Next→](#)

11 Rails: Static Pages



“I make static art, not dynamic art. That's what I do.”

— **Michael Heizer**

Overview and Purpose

In this checkpoint you'll start creating a new application using Ruby on Rails.

Objectives

After this checkpoint, you should be able to:

- Explain MVC.
- Discuss the V in MVC.
- Discuss the C in MVC.
- Use Git branching.
- Explain routing in Rails.
- Explain what `localhost` means.
- Use `rails generate`.
- Manually create the same files as `rails generate`.

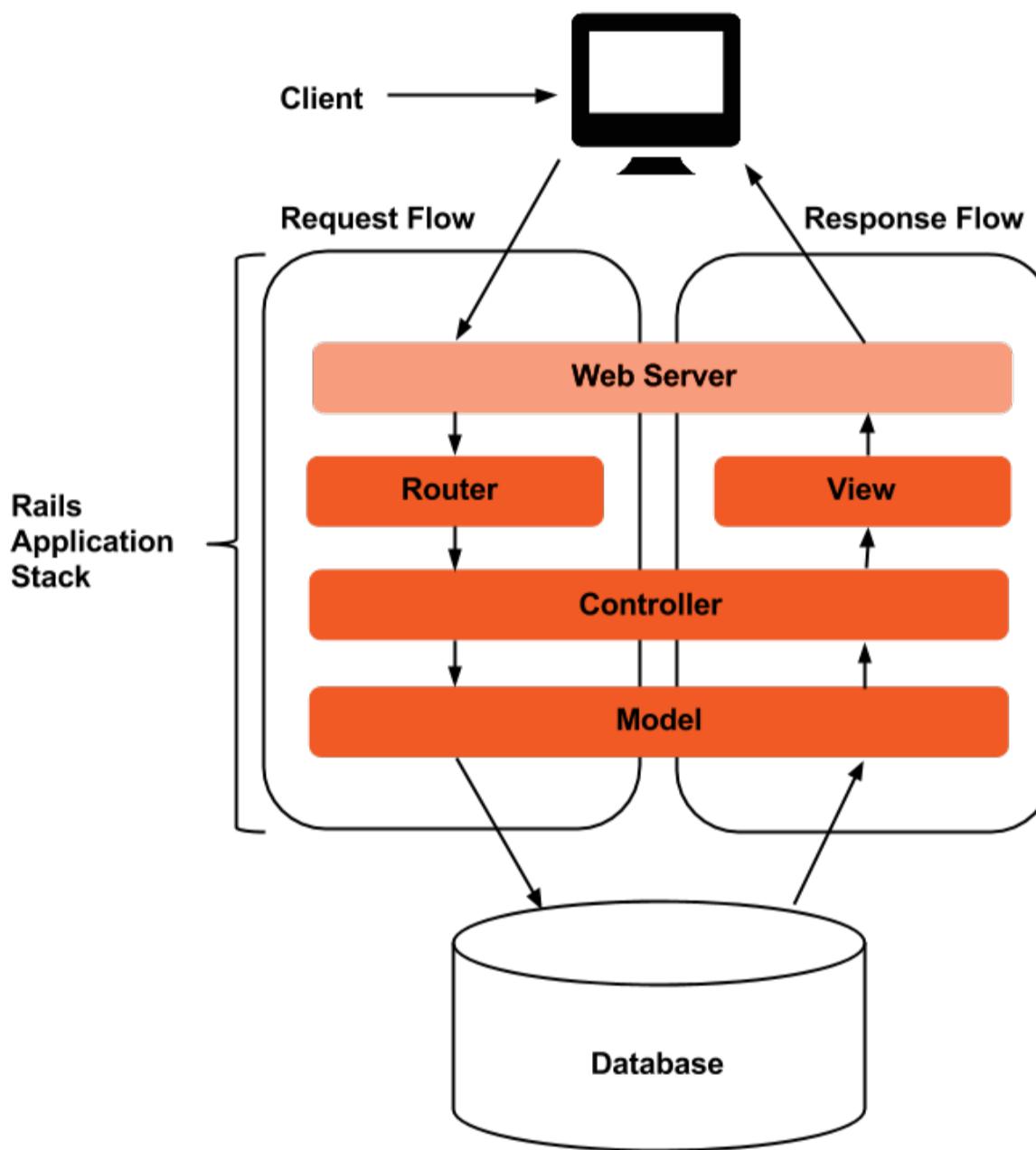
Static Views

We have a working Rails app, but other than the default index page there's not much to show. The purpose of this checkpoint is to build static views, and in the process we'll learn the fundamentals of MVC architecture.

MVC Architecture

MVC, which is an acronym for "Model View Controller", is the basic architectural pattern that guides the creation of all Rails applications. You worked with basic MVC when you built Address Bloc. In this checkpoint, we'll focus on views and controllers and learn about models later.

A view is equivalent to a web page, and a controller determines what view should be shown. Consider the diagram below and focus on the flow of the request and response, as they pertain to views and controllers.



When you visit a website, you initiate a chain of actions. In an MVC application, a request is handled by a controller, which receives information from the model layer, and then uses that information to display a view.

MVC architecture is analogous to the basic function in a restaurant:

1. A customer (user) places an order with the waiter (controller).
2. The waiter informs the kitchen (model) of the order.
3. After the kitchen makes the order, the waiter serves the dish (view) to a customer.

The waiter doesn't need to know how the order will be prepared, or how it will be consumed, and

that's just fine. Controllers, like waiters, should only be concerned with passing things to other parties.

We review MVC components and examples of their corresponding code in the next video:

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Generating a Controller and Views

The best way to understand the relationship between controllers and views is to create

them. We could create controller and view files manually, but Rails provides a handy generator which ensures that *all* necessary files are generated for a given controller. To generate a controller and its views, type the following on your command line in your project's directory:

Terminal

```
$ rails generate controller welcome index about
```

The output should look like this:

Terminal

```
create  app/controllers/welcome_controller.rb
route  get 'welcome/about'
route  get 'welcome/index'
invoke  erb
create    app/views/welcome
create    app/views/welcome/index.html.erb
create    app/views/welcome/about.html.erb
invoke  helper
create    app/helpers/welcome_helper.rb
invoke  assets
invoke  js
create      app/assets/javascripts/welcome.js
invoke  scss
create      app/assets/stylesheets/welcome.scss
```

We passed three arguments to the `rails generate` command. The first argument represents the controller name, which is `welcome`. The next two arguments (`index` and `about`) represent views corresponding with the `welcome` controller. We could've named the controller and views anything, but the names should correspond with their primary function, as a best practice.

Exploring Controllers and Views

Open your project in your text editor. You should see a file named `welcome_controller.rb` in `app/controllers/`. You should also see the two views you created in `app/views/welcome/`. The generator created some code:

```
app/controllers/welcome_controller.rb
```

```
class WelcomeController < ApplicationController
  def index
  end

  def about
  end
end
```

`WelcomeController` is a Ruby class, and contains two empty methods that correspond to view names. These identically named methods and views are an example of a Rails convention called **default rendering**. When a controller method's purpose is to invoke a view, *it must be named with respect to the view*. The `index` method in the `WelcomeController` will invoke the `index` view inside the `app/views/welcome` directory.

Open the `index` and `about` views and read the placeholder code:

app/views/welcome/index.html.erb

```
<h1>Welcome#index</h1>
<p>Find me in app/views/welcome/index.html.erb</p>
```

app/views/welcome/about.html.erb

```
<h1>Welcome#about</h1>
<p>Find me in app/views/welcome/about.html.erb</p>
```

Start the Rails server from your command line:

Terminal

```
$ rails s
```

Visit localhost:3000/welcome/index and localhost:3000/welcome/about to view the HTML code that was created by the controller generator.

Routing in Rails

The controller generator created the basic code needed for the `WelcomeController` and its views, and it also created code in the `config/routes.rb` file:

config/routes.rb

```
Rails.application.routes.draw do
  get "welcome/index"

  get "welcome/about"
  ...
end
```

This code creates HTTP `GET` routes for the `index` and `about` views. HTTP is the protocol that the Internet uses to communicate with websites. The `get` action corresponds to the HTTP `GET` verb. `GET` requests are used to retrieve information identified by the URL.

The HTTP protocol has other actions which we'll explore later.

If `routes.rb` doesn't specify a `GET` action, the view will not be served because the application won't know what to `get` when a user sends a request. Test this by commenting out these lines:

config/routes.rb

```
# get "welcome/index"

# get "welcome/about"
```

Restart the server and visit `localhost:3000/welcome/index`. We'll see a Rails "Routing Error" page. This error occurs when our app doesn't understand what we're requesting, because there is no corresponding `get` action.

Uncomment those two lines and **delete all the other commented lines in the file**. Add a `root` path to the `routes.rb` file:

config/routes.rb

```
Rails.application.routes.draw do
  get "welcome/index"

  get "welcome/about"

+   root 'welcome#index'
  ...
end
```

The `root` method allows us to declare the default page the app loads when we navigate to the home page URL. Test it by going to `localhost:3000`. You should see the welcome `index` view by default.

`root` is a method that takes a hash as an argument, here using the "implied hash" syntax. The line could be rewritten without using an implied hash as: `root({to: 'welcome#index'})`. You'll see implied hashes frequently in Rails because they enhance readability.

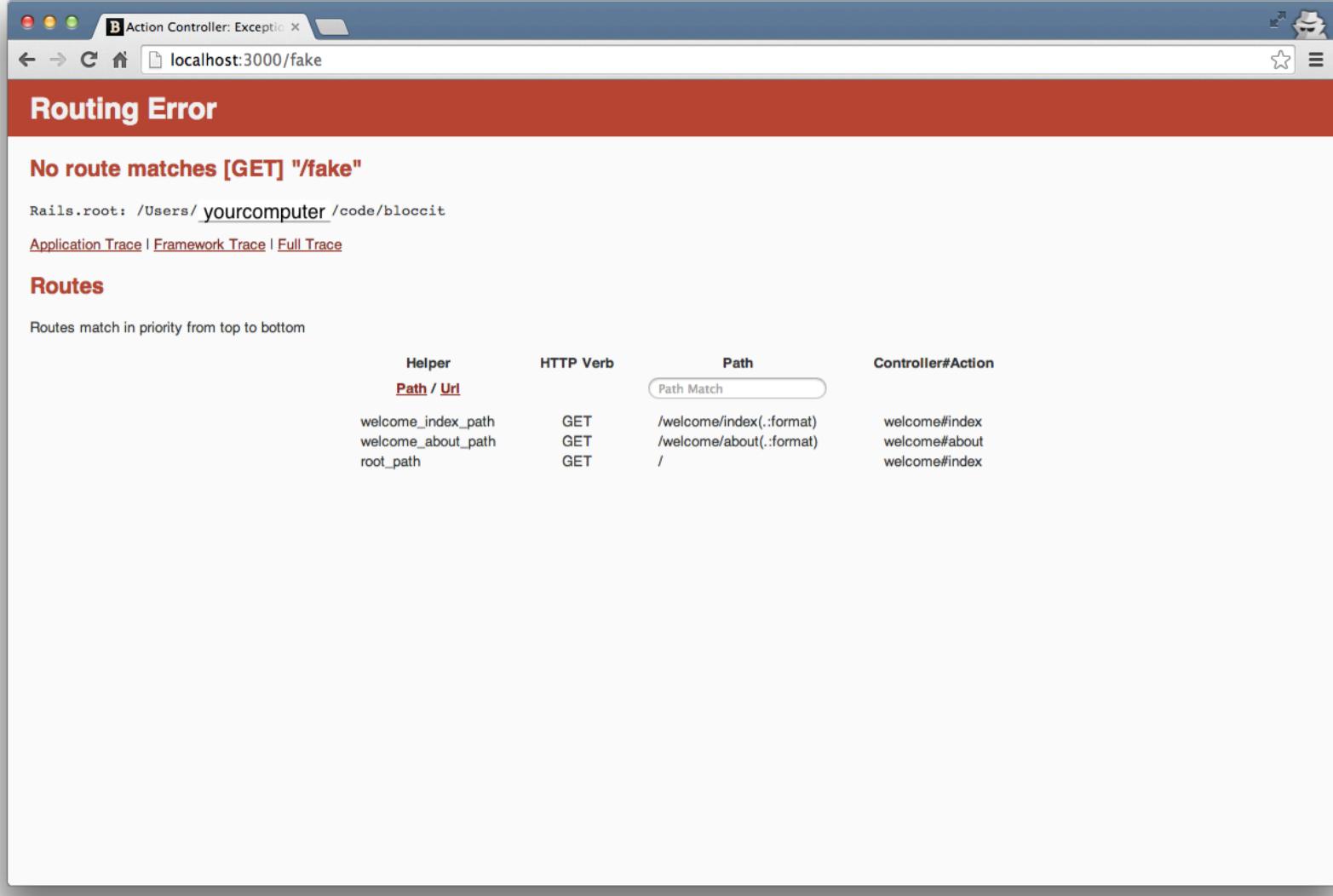
View your app's available routes by typing `rails routes` from the command line. Stop the Rails server (`CTRL+C`) and give it a try. You should see the following output:

Terminal

```
$ rails routes
      Prefix Verb URI Pattern          Controller#Action
welcome_index  GET  /welcome/index(.:format) welcome#index
welcome_about   GET  /welcome/about(.:format) welcome#about
root           GET  /                  welcome#index
```

- The first column represents the route name: `welcome_index`
- The second column represents the HTTP action associated with the route: `GET`
- The third column represents the URI pattern, which is the URL used to request the view: `/welcome/index`
- The fourth column represents the route destination, which translates to the controller and associated view: `welcome#index`

By default, Rails will present a searchable list of valid routes if an invalid route is requested. This is handy for troubleshooting large applications with many routes, and is also a nice fail-safe. Try it on `localhost`:



Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Recap

Concept	Description
MVC	MVC (Model–view–controller) is an architectural pattern that divides a given application into three interconnected parts with distinct responsibilities.
Git Branching	Diverges the master branch, so that you can work on new features without affecting the master branch. Git branches require little memory or disk space, making branching operations nearly instantaneous.
	The <code>rails generate</code> command creates controllers from

`rails generate`

templates. The `generate` command can also generate controller actions and their corresponding views.

Controller

Controllers are represented by the C in MVC. Controllers process requests and produce the appropriate output. Controllers communicate with the database and perform CRUD actions where necessary, via models.

Views

Views are responsible for rendering templates. View templates are written using embedded Ruby in tags and integrated with HTML.

`rails routes`

The `rails routes` command lists all routes, in the same order as `routes.rb`.

localhost

localhost is a hostname that represents "this computer".

How would you rate this checkpoint and assignment?



11. Rails: Static Pages

Assignment

Discussion

Submission

11. Rails: Static Pages

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

Use what you learned in this checkpoint to create a Contact page, do not use

`rails generate:`

1. Manually create `app/views/welcome/contact.html.erb`.
2. Manually create the route to your new page in `routes.rb`.
3. Add a `contact` action to `WelcomeController`.

Assignment

Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.

Solution

Do not watch this video until after you've attempted to complete the assignment. If you struggle to complete the assignment, submit your best effort to your mentor *before watching a solution video*.

[Static Pages Solution](#)

[←Prev](#)

Submission

[Next→](#)

12 Rails: Testing



“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”

— Martin Golding

Overview and Purpose

In this checkpoint you'll learn more about the power of TDD and RSpec.

Objectives

After this checkpoint, you should be able to:

- Create a test group in your Gemfile.
- Explain the red-green-refactor pattern.
- Explain the DRY philosophy.

Why We Test



BLOC

Intro: Bloccit - Testing

0:33



Test code is used to state expectations that are to be met when production code - the code that runs an application - is executed. Test code raises errors when its stated expectations are not met by production code. The two primary reasons to write test code are:

1. To ensure that production code does what it's intended to do; and
2. To ensure that production code doesn't break when you **refactor** it. **Refactoring** is "the process of restructuring existing computer code – changing the factoring – without changing its external behavior".

Like life, the one constant in programming is change. Code will evolve in ways you can't predict. If you have multiple dependencies, a small change to one file can create unexpected consequences in many files.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

RSpec

There are several testing frameworks for developing web applications with Rails, but we'll focus on **RSpec** because it is the canonical framework, and thus the most likely framework you will encounter as a professional Rails developer.

Add the `rspec-rails` and `rails-controller-testing` gems to your Gemfile:

Gemfile

```
...
+ group :development, :test do
+   gem 'rspec-rails', '~> 3.0'
+   gem 'rails-controller-testing'
+ end
```

We added the gems to the `:development` and `:test` groups because we want its tasks and generators to be available in both environments. We specified a version (`~> 3.0`) for `rspec-reails` to maintain predictable behavior despite new RSpec releases.

Run `bundle` from the command line to update your application with the installation of RSpec. Use the RSpec generator to configure Bloccit for testing:

Terminal

```
$ rails generate rspec:install
  create .rspec
  create spec
  create spec/spec_helper.rb
  create spec/rails_helper.rb
```

This generator creates a spec directory where we will write our tests.

RSpec will now automatically add test files for our models and controllers when we run `rails generate model...` or `rails generate controller...`.

The Test Database

Tests should be run in isolation because they can alter data stored in a database. That is, if we were to run tests in a Production environment, the tests could alter *production* data - that would be a very bad thing to do. Running tests in isolation is somewhat standard behavior for web development in general, so by default Rails designates a separate database for testing.

The test database is *completely empty* before you run your specs. Therefore, a spec must create the necessary data to test functionality. When the test is complete, the data is destroyed.

The Test database is isolated from the Development and Production databases. RSpec empties the Test database before running each spec. **Each test must create the data it needs.**

Test-Driven Development

Test-Driven Development is the process of writing tests *before* writing production code. Writing tests first might sound illogical, but it has many advantages:

- Only the production code needed to pass a test is written. This leads to a leaner and more efficient codebase because you only code what you need – nothing more and nothing less.
- Test-Driven Development allows developers to segment problems into small and testable steps.
- Testing early and often allows developers to catch bugs earlier, preventing more expensive problems later when the codebase is large and hard to navigate.
- Writing tests for code that doesn't exist can produce a **flow state**. Writing tests *before* writing production code can systematize your thought process, forcing you to be more explicit about how a function or page should behave.

Red, Green, Refactor

The TDD process involves three steps:

1. Write a failing test for production functionality that does not exist. (Red)
 - Ensure that the test *actually fails*. This verifies two aspects of the test: first, it demonstrates that the new spec does not pass with the existing code you've written, saving you from writing unnecessary code; second, it precludes the possibility that your test always passes, which could be an indication of a poorly-written test.

2. Create the production functionality such that the test passes. (Green)
3. Refactor the production code to make it cleaner and more sustainable. With a well-written test, you can refactor production code with the confidence that you will not break the application. If you refactor your code in such a way that would break the application, your test would fail and you would know to fix the problem that caused the failure.

We call this three-step process "Red, Green, Refactor", because of the colored command line output of running tests.

Basic Testing Principles

1. **Keep tests as low-level as possible:** Test models thoroughly (we'll learn about models soon), test controllers moderately, and test complete application flow lightly. If we know the foundation (models) of our application is solid, we can put more trust in higher functions like controller actions and application flow.
2. **Respect object limits:** When testing an object, try not to test any other objects, even if they're related. Narrow the scope of the test to be as small and self-contained as possible.
3. **Don't test "how", test "what":** We want to test what a method returns, not how it returns it. The internal implementation of a method is subjective, and while we believe in idioms and programming style, it is not the test's job to assess those things - only to assess what the code returns.
4. **Write DRY tests:** Wherever possible, avoid repetition in tests, just like production code.
5. **Test early and often:** Tests function as our safety net, but they can't help us if we don't use them. At a minimum, we'll want to run our specs before each commit. Running tests before each commit allows us to reduce bugs proactively before we add them to the codebase.

Our First Test

Let's create our first set of specs to test the `WelcomeController` actions. Generate a spec for `WelcomeController`:

Terminal

```
$ rails generate rspec:controller welcome
      create  spec/controllers/welcome_controller_spec.rb
```

This command generates a spec file - `welcome_controller_spec.rb` - and places it in the

`spec/controllers` directory. All specs will be written in the `spec` directory and are nested according to which part of the codebase they test. Open `welcome_controller_spec.rb` and add a test for the `index` action:

`spec/controllers/welcome_controller_spec.rb`

```
require 'rails_helper'

# #1
RSpec.describe WelcomeController, type: :controller do
  + describe "GET index" do
    + it "renders the index template" do
      # #2
      + get :index
      # #3
      + expect(response).to render_template("index")
    + end
  + end
end
```

- At #1, we describe the subject of the spec, `WelcomeController`.
- We use `get`, at #2, to call the `index` method of `WelcomeController`.
- At #3, we `expect` the controller's `response` to render the `index` template.

Run `welcome_controller_spec.rb` to confirm that the new test passes:

Terminal

```
$ rspec spec/controllers/welcome_controller_spec.rb
.

Finished in 0.0162 seconds (files took 2.75 seconds to load)
1 example, 0 failures
```

If you receive a message that says "...db/schema.rb doesn't exist yet" disregard it for now. We'll address it in a later checkpoint.

Let's add a similar test for the `about` method:

`spec/controllers/welcome_controller_spec.rb`

```
require 'rails_helper'

RSpec.describe WelcomeController, type: :controller do
  describe "GET index" do
    it "renders the index template" do
      get :index
      expect(response).to render_template("index")
    end
  end

  +
  + describe "GET about" do
  +   it "renders the about template" do
  +     get :about
  +     expect(response).to render_template("about")
  +   end
  + end
end
```

Run the spec again:

Terminal

```
$ rspec spec/controllers/welcome_controller_spec.rb
..
Finished in 0.0199 seconds (files took 2.96 seconds to load)
2 examples, 0 failures
```

Congratulations, your first two tests are a complete success! You'll find that seeing green results from your tests can be very satisfying and therapeutic.

The following video demonstrates the principles of TDD in more detail:

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Recap

Concept	Description
RSpec	RSpec is a test framework written in and for Ruby.
Rails Test	Rails' dedicated test database allows developers to initiate and

Database

interact with test data in isolation so that production data is not compromised.

Test-Driven Development

Test-Driven Development (TDD) is a software development process where test code is written prior to production code.

How would you rate this checkpoint and assignment?

**12. Rails: Testing**

A small orange icon of a document with a pencil.

A small grey icon of a speech bubble with a dot inside.

A small grey icon of a document with a checkmark.

12. Rails: Testing

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

Create an FAQ page using TDD, do not use `rails generate`, instead, create the necessary files manually:

1. Add a test to `welcome_controller_spec.rb` to test the `faq` action
2. Run test to see it fail. If you don't see a test fail in the expected way, it's hard to trust that it is testing what you intended.
3. Add the FAQ route to `routes.rb`
4. Run test again. You should see a new failure.
5. Add an `faq` action to `WelcomeController`
6. Run test again.
7. Create `app/views/welcome/faq.html.erb`
8. Run `welcome_controller_spec.rb` to confirm the new test passes

Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.

Solution

[←Prev](#)

Submission

[Next→](#)

13 Rails: HTML and CSS



“We never go out of style.”

— Taylor Swift

Overview and Purpose

In this checkpoint you'll add HTML and CSS to stylize your application.

Objectives

After this checkpoint, you should be able to:

- Incorporate HTML within a Rails application.
- Use CSS selectors within your Rails application.
- Install a CSS framework from a gem, such as Bootstrap.

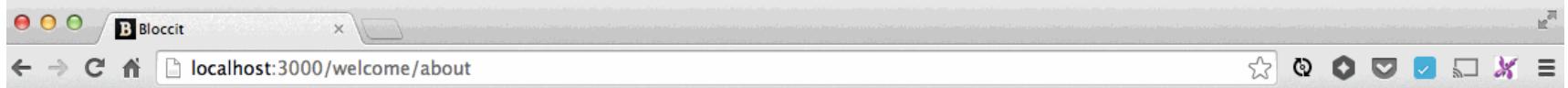
Structure and Style

The two basic building blocks of web development are HTML (Hypertext Markup Language) and CSS (Cascading Style Sheets). They provide the structure (HTML) and the style (CSS) for all web pages.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

HTML



- [Home](#)
- [About](#)

About Bloccit

Created by: Me

HTML is code that a browser can interpret and display as a web page. HTML by itself is not dynamic, but when used in conjunction with a language like Ruby and a framework like Rails, it can be *rendered* dynamically. Based on the type of request made by a user, an app will respond with different HTML.

HTML, Ruby, and `application.html.erb`

The **index** and **about** views have an `.html.erb` file extension. This file extension allows us to use HTML and Ruby (ERB stands for "embedded Ruby") in the same file. By integrating Ruby code with HTML, we can dynamically change the behavior of static HTML code, based on user input.

Nearly every view in a Rails application will have some unique HTML and Ruby code, but there is also common code that needs to be included in *all views*. Rather than repeating the same code in every view, we use `application.html.erb`. Each view (like **index** and **about**) is called from, and rendered inside, `application.html.erb`. In this way, you can think of `application.html.erb` as a container file that has HTML and Ruby code needed to run every view in a Rails app.

To help you understand the rendering process in `application.html.erb`, consider the following actions:

1. A user requests a view
2. The controller corresponding to the requested view invokes `application.html.erb`
3. `application.html.erb` inserts the appropriate view using `yield`

4. The complete web page is rendered and returned to the user

`yield` is used to invoke a block, which renders a given view inside the `application.html.erb` container. Code between `<%` and `%>` is interpreted as Ruby. If the `<% %>` contains an `=`, such as `<%= %>`, the result of the Ruby code is printed to the screen (i.e. rendered as HTML). If there is no `=` (only `<% %>`) then the Ruby code will be *executed* but not printed.

Modify the `application.html.erb` file to include additional HTML and Ruby that we need in every view in Bloccit:

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bloccit</title>
    <%= csrf_meta_tags %>

    <%= stylesheet_link_tag      'application', media: 'all', 'data-turbolinks-track': 'reload' %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
+   <ul>
+     <li><%= link_to "Home", welcome_index_path %></li>
+     <li><%= link_to "About", welcome_about_path %></li>
+   </ul>

  <%= yield %>

  </body>
</html>
```

The `` and `` tags render content as bulleted lists; (`ul`) stands for "unordered list"). Inside of the `` (list item) tags, you called Rails' `link_to` method. `link_to` is a helper method available in views, and returns a valid HTML hyperlink (called an **anchor tag**). For example:

```
<%= link_to "Home", welcome_index_path %>
```

Will render the following HTML:

```
<a href="/welcome/index">Home</a>
```

Rails Helper Methods

`link_to` is a Rails helper method that returns a string of HTML code. `link_to` takes two arguments, a string ("Home") which will be the display name of the hyperlink, and a path (`welcome_index_path`). `welcome_index_path` is a Rails method, generated by the `routes` file. Type `rake routes` on your command line again, and you'll see that the route name in the first column is `welcome_index`. When you add `_path` to the route name, it's recognized as a helper method that returns `"/welcome/index"`.

We could have typed `link_to "Home", "/welcome/index"` instead. The resulting link would have been identical, but Rails helpers are generally easier to use, and idiomatically correct.

Start the Rails server and make sure that you are directed to the correct pages when you click the "Home" and "About" links.

Terminal

```
$ rails s
```

Rather than starting and stopping your server frequently, you can open multiple tabs in your terminal. You can leave one open for Git and Rails commands and one for the Rails server. Remember, if you change a route or a config file, or add a new Ruby gem, you'll need to restart your server. For all other changes - like changes to controller, view, or model files, a restart is *not* necessary.

CSS Selectors in Rails

Let's use CSS to modify the font color of `<h1>` tags :

app/assets/stylesheets/welcome.scss

```
...
+h1 {
+  color: red;
+}
```

View the **index** and **about** pages. The content between the `<h1>` tags should be red now. There are a few important things to note about the code above:

1. The CSS we added was in the `welcome.scss` file. When we ran `rails generate controller`, `welcome.scss` was created. By Rails convention, each controller has a corresponding stylesheet and view.
2. Similar to the `.html.erb` extension, the `.scss` extension provides us with some

additional syntax options (known as **Sass**) to enhance default CSS capabilities.

Sass is not a core part of this program, but you'll have a chance to experiment with it in the projects phase with your mentor, if you choose to do so.

3. We wrote a CSS *selector* and *declaration*. The selector, `h1`, specifies which element to modify. The declaration, composed of a *property* (`color`) and *value* (`red`), specifies how to modify it. We turned all `<h1>` tags red by using the `h1 { }` selector.

CSS Frameworks and Bootstrap

We've added some simple styles above, but we have a long way to go in making Bloccit look presentable. While you could write comprehensive style sheets from scratch, it's much more efficient to use a CSS framework. A CSS framework comes with many different style and position classes that you can use directly or customize further.

One of the most popular CSS frameworks is **Bootstrap**, which we'll use for Bloccit. Bootstrap also provides layouts, forms, buttons, icons, Javascript functions, and more. Let's start by installing bootstrap and using a few basic style classes:

1. Stop the Rails server, open `Gemfile` and add the **bootstrap-sass** gem at the bottom of the `Gemfile` (make sure it is **outside** of a `group`):

Gemfile

```
...
+ gem 'bootstrap-sass'
```

2. Run `bundle install` to install the new gem.

3. Rename `application.css` to `application.scss`:

Terminal

```
$ mv app/assets/stylesheets/application.css app/assets/stylesheets/application.s
```

4. Add the following lines to the bottom of `application.scss`, to integrate the newly installed Bootstrap with our application:

app/assets/stylesheets/application.scss

```
+ @import "bootstrap-sprockets";
+ @import "bootstrap";
```

5. Include the Bootstrap file in `app/assets/javascripts/application.js`:

app/assets/javascripts/application.js

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
+ //= require bootstrap
//= require_tree .
```

To use Bootstrap's styles, add some of its layout classes to our container HTML:

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Bloccit</title>
    <%= csrf_meta_tags %>

    + <meta name="viewport" content="width=device-width, initial-scale=1">
      <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
-   <ul>
+   <div class="container">
+     <ul class="nav nav-tabs">
      <li><%= link_to "Home", welcome_index_path %></li>
      <li><%= link_to "About", welcome_about_path %></li>
    </ul>

    <%= yield %>
+   </div>

  </body>
</html>
```

The "viewport meta" tag added inside the `<head>` with a `content` attribute value of `width=device-width, initial-scale=1` instructs browsers on small, high-pixel density screens (such as retina iPhones) to display our pages at a regular, readable size. Without this tag, our pages won't scale properly.

`container`, `nav`, and `nav-tabs` are classes provided by Bootstrap. By assigning these classes to HTML elements like `<div>` and ``, you are styling them with default Bootstrap properties and values.

Remove the CSS rules you added to `welcome.scss` as well as the HTML changes you made in the `index` view (leaving the two `@import` lines). They were for demonstration

only.

Restart the Rails server and observe the changes.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Deploy to Heroku with `git push heroku master`.

How would you rate this checkpoint and assignment?



13. Rails: HTML and CSS

Assignment

Discussion

Submission

13. Rails: HTML and CSS

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

Use CSS and HTML to create a button in your `about.html.erb` page that links to one of your social media accounts:

1. Read about **CSS positioning** and the **Box Model** and use what you learn to center the text inside of the button.
2. Create a CSS class in `welcome.scss` called `.social-btn` and use the attributes you read about to shape and position it. Attributes like display, height, width, background, border, color, text-align and font will help you create a basic button.
3. Read our resource on Chrome's **Web Inspector**. Start Rails Server and experiment with paddings, margins, and other stylings in the inspector.
4. Use the `ActionView` helper method `link_to` to link your button to your favorite social media account. This link should be on the welcome **about** view.

Hint: You can pass an external link as the URL parameter to the `link_to` method in a string:

[←Prev](#)

Submission

[Next→](#)

14 Rails: Models



People come up to me all the time and say 'you should be a model', or 'you look just like a model,' or 'maybe you should try to be a man who models.' And I always have to laugh because I'm so good looking. Of course I'm a model.

— **Derek Zoolander**

Overview and Purpose

In this checkpoint you'll learn about models in a Ruby on Rails application.

Objectives

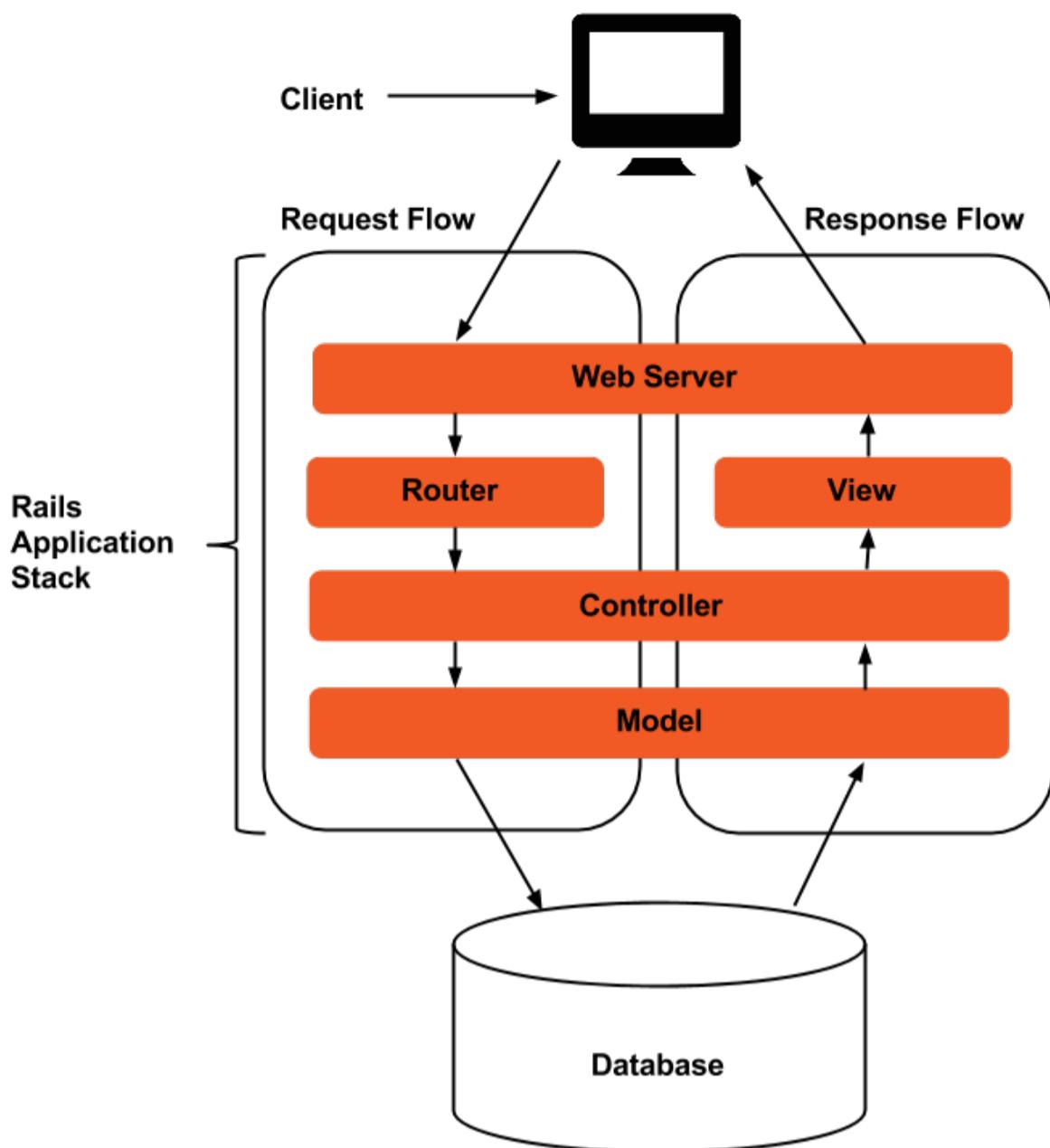
After this checkpoint, you should be able to:

- Create a model in a Rails application.

- Define `ActiveRecord` objects.
- Explain what `ActiveRecord` migrations are.
- Explain what a foreign key is on a table.

Models

Bloccit users will need to be able to *post* information and *comment* on those posts. Posts and comments will need to persist; that is, they'll need to be saved to a database so users can interact with them across sessions. When we need to persist data, we should immediately think about data models. Data models, or more simply "models", are the "M" in MVC architecture. Recall the diagram of MVC architecture to consider a model's place with respect to controllers and views:



Models are a programmatic representation of a table in a database. Models are also Ruby classes, similar to the classes we programmed earlier in the roadmap. In other words, the `Post` model will handle data pertaining to an instance of the `Post` class.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Post

The first model we'll create is `Post`. Users should have the ability to submit posts to Bloccit with titles and descriptions, so the `Post` model and its corresponding database table will need two attributes: `title` and `body`.

An attribute is synonymous with database table column. You can also imagine a database table as a tab on a spreadsheet. Attributes like "title" and "body" would be column headers on that spreadsheet tab.

Use a generator to create `Post` and its corresponding spec:

Terminal

```
$ rails generate model Post title:string body:text
  invoke  active_record
  identical db/migrate/20150606010447_create_posts.rb
  identical app/models/post.rb
  invoke    rspec
  create    spec/models/post_spec.rb
```

We used a generator to create a model named "Post" with two attributes: `title` and `body`.

- The `title` attribute is a string data type, because we expect it to be short. That is, we would probably not want to allow post titles with hundreds of characters.
- The `body` attribute is a text data type, because we expect a post's body to be verbose. It's possible that a user will need the ability to write hundreds of characters to provide context for a post.

The programmatic representation of the Post model was created by the generator above, in several files:

- `post.rb` is a Ruby class which represents the Post model. This class will handle the logic and define the behavior for posts.
- `post_spec.rb` is the test spec for the Post class.
- `20150606010447_create_posts.rb` is the database migration file. A migration file defines the action taken on the database for a given model. An application will have many migration files, and comprehensively they serve as a set of instructions for building a database. We'll explore migration files in detail later.

Use `cat` to see the contents of the spec file:

Terminal

```
$ cat spec/models/post_spec.rb
require 'rails_helper'

RSpec.describe Post, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

This is the template for a simple spec. We'll use TDD to define the behavior for `Post`. Add the following tests:

```
spec/models/post_spec.rb
```

```

require 'rails_helper'

RSpec.describe Post, type: :model do
- pending "add some examples to (or delete) #{__FILE__}"
# #1
+ let(:post) { Post.create!(title: "New Post Title", body: "New Post Body") }
+
# #2
+ describe "attributes" do
+   it "has title and body attributes" do
+     expect(post).to have_attributes(title: "New Post Title", body: "New Post Body")
+   end
+ end
end

```

- At #1, using the `let` method, we create a new instance of the Post class, and name it `post`. `let` dynamically defines a method (in this case, `post`), and, upon first call *within a spec* (the `it` block), computes and stores the returned value.
- At #2, we test whether `post` has attributes named `title` and `body`. This tests whether `post` will return a non-nil value when `post.title` and `post.body` is called.

Use `cat` to see the contents of `post.rb`, which was also created with a basic template:

Terminal

```
$ cat app/models/post.rb
class Post < ApplicationRecord
end
```

When the generator created this template, it made the Post class **inherit** from **ApplicationRecord**. Because we used a model generator, Rails assumed that we wanted our class to be used as a **model**. `ApplicationRecord` essentially handles interaction with the database and allows us to persist data through our class. Run the spec:

Terminal

```
$ rspec spec/models/post_spec.rb
```

You will see a verbose error, but focus on the first line:

Terminal

```
schema.rb doesn't exist yet. Run `rails db:migrate` to create it, then try again.
```

RSpec reported that `schema.rb` doesn't exist. `schema.rb` is a file located in the `db`

directory that represents an application's complete database architecture; the tables it uses and how those tables relate to each other. We don't have `schema.rb` because we have not yet created the database or any tables. The generator created the migration file, but we haven't executed that file yet. We'll do that now:

Terminal

```
$ rails db:migrate
== 20150606010447 CreatePosts: migrating =====
-- create_table(:posts)
  --> 0.0016s
== 20150606010447 CreatePosts: migrated (0.0017s) =====
```

Rake is a Ruby build command. It allows us to execute administrative tasks for our application. To see a complete list of rake tasks, type `rake --tasks` from the command line.

`rails db:migrate` created a new table named "posts". Let's review the migration file, which is the only file in the `db/migrate` directory (its name begins with a timestamp, and so will differ from the one below):

db/migrate/20150606010447_create_posts.rb

```
$ cat db/migrate/20150606010447_create_posts.rb
class CreatePosts < ActiveRecord::Migration[5.0]
  def change
    create_table :posts do |t|
      t.string :title
      t.text :body

      t.timestamps
    end
  end
end
```

The migration is written in Ruby. The migration file is actually a class named `CreatePosts`. When we run the migration, the `change` method calls the `create_table` method. `create_table` takes a block that specifies the attributes we want our table to possess.

Rails automatically adds timestamp attributes named `created_at` and `updated_at` to the migration. We'll discuss these attributes in depth later.

Run the tests in `post_spec.rb` again:

Terminal

```
$ rspec spec/models/post_spec.rb
..
Finished in 0.00817 seconds (files took 1.67 seconds to load)
1 examples, 0 failures
```

Our tests passed, so we know that the Post model has the attributes we expected.

Comment

The `Comment` model needs one attribute - `body` - and a reference to `Post`. Let's create the spec, model, and migration files with the model generator:

Terminal

```
$ rails generate model Comment body:text post:references
  invoke  active_record
  create    db/migrate/20150608215948_create_comments.rb
  create    app/models/comment.rb
  invoke    rspec
  create      spec/models/comment_spec.rb
```

Open `comment_spec.rb` and add the following test:

spec/models/comment_spec.rb

```
require 'rails_helper'

RSpec.describe Comment, type: :model do
-   pending "add some examples to (or delete) #{__FILE__}"
+   let(:post) { Post.create!(title: "New Post Title", body: "New Post Body") }
+   let(:comment) { Comment.create!(body: 'Comment Body', post: post) }

+   describe "attributes" do
+     it "has a body attribute" do
+       expect(comment).to have_attributes(body: "Comment Body")
+     end
+   end
end
```

Review `comment.rb`:

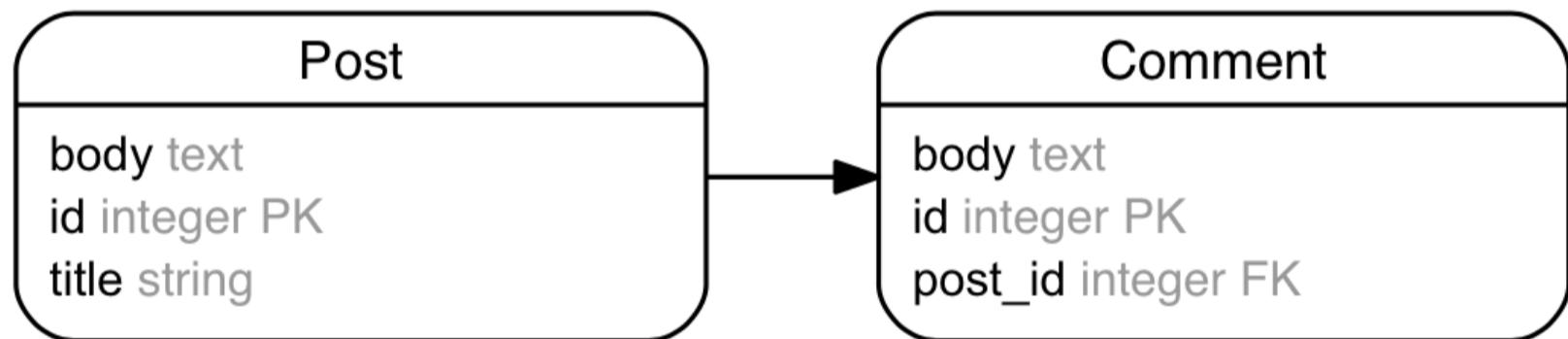
app/models/comment.rb

```
$ cat app/models/comment.rb
class Comment < ApplicationRecord
  belongs_to :post
end
```

We say that "*a comment belongs to a post*" or, conversely, "*a post has many comments*." Both of these phrases relate to a data concept known as "relationships" or "associations."

Each model instance in a Rails app automatically gets an `id` attribute to uniquely identify it. Each `post` will have a unique `id`, as will each `comment`. To make a comment belong to a post, we need to provide the post `id` to the `comment`. This is done using a **foreign key**.

A foreign key is the `id` of one model, used as an attribute in another model, in order to look up the relationship. In the Post/Comment example, this means that the `Comment` model needs to have an attribute named `post_id`. The `post_id` attribute exists so that a `comment` can belong to a `post` (the post specified by its `post_id`). To allow many comments to belong to one `post`, you'd have multiple comment records with the same `post_id`. The diagram below illustrates how the post's `id` attribute relates to a comment's `post_id` attribute:



Rails is an opinionated framework that enforces many conventions by design. The foreign key naming convention of `post_id` was enforced when you ran the model generator. The `post_id` attribute was automatically created in the `Comment` model when you generated it with the `post:references` argument.

Review the `create_comments` migration in the `db/migrate` directory and add a foreign key:

```
db/migrate/20140624203804_create_comments.rb
```

```
class CreateComments < ActiveRecord::Migration[5.0]
  def change
    create_table :comments do |t|
      t.text :body
      t.references :post, foreign_key: true

      t.timestamps
    end
  end
end
```

In reviewing the comment and post migrations, we see that the `create_table` method takes a `Symbol` argument which represents the table name, and a block argument that contains the details to be added to the table. This is one of the many reasons why Rails developers can code so efficiently - rather than creating the tables manually, and making sure all the attributes are set properly, we can rely on Rails' model generator to handle this mundane work.

Since we have a new migration file, we shall once again run the migrations, adding the comments table to the database:

Terminal

```
$ rails db:migrate
== 20150608215948 CreateComments: migrating =====
-- create_table(:comments)
  --> 0.0021s
== 20150608215948 CreateComments: migrated (0.0022s) =====
```

The results above tell us that the tables and attributes have been created successfully.

Run `comment_spec.rb`:

Terminal

```
$ rspec spec/models/comment_spec.rb
.

Finished in 0.01325 seconds (files took 1.78 seconds to load)
1 example, 0 failures
```

Git does not create empty directories by default. As a consequence, Rails automatically generates a blank `.keep` file in important directories that start as empty in a new application. Our `app/models` directory has one such file. Now that we've added two files to `app/models`, we should remove it:

```
$ rm app/models/.keep
```

You've successfully created two database tables and associated them using a foreign key. In the next section we'll update the models to reflect the attributes and associations we just created in the console.

Updating Post

Remember that when we use a model generator, the resulting model inherits from a class named `ApplicationRecord` by default. This inheritance pattern provides methods that the model will need in order to interact with tables in the database. Our comment class already relates to our post class, thanks to the model generator, but the post class does not yet relate to the comment class. Let's create that relation:

app/models/post.rb

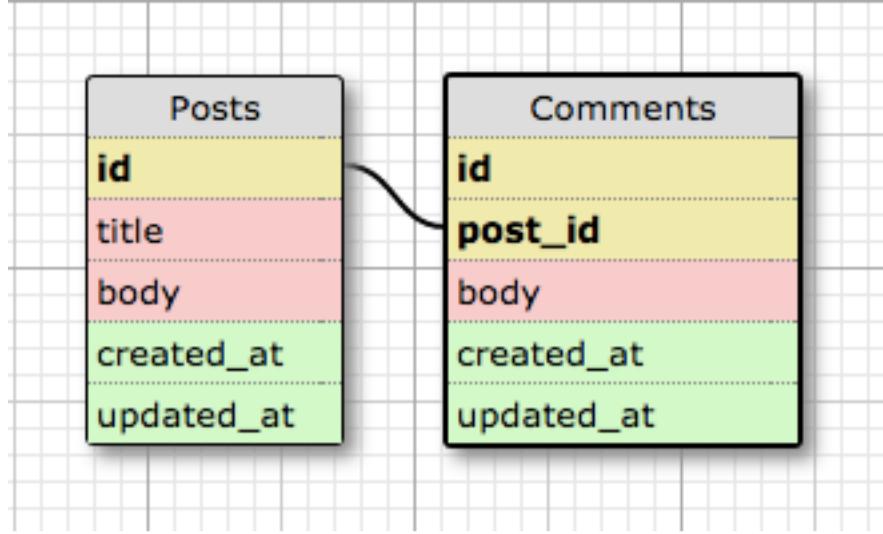
```
class Post < ApplicationRecord
  + has_many :comments
end
```

The `has_many` method allows a post instance to have many comments related to it, and also provides methods that allow us to retrieve comments that belong to a post.

This dynamic generation is similar to the way `attr_accessor` generates 'getter' and 'setter' methods for instance variables. We'll explore the precise methods created by `has_many` in the next checkpoint.

Visualizing the Database

It can be helpful to think about databases visually: What tables does the database have? What attributes do the tables have? How are the tables related? Here's a sample visualization for the two tables we've created in our application:



When we say that a comment belongs to a post, we mean that the comment stores that post's unique identifier in an attribute. The visualization makes this obvious.

In the proceeding video, we use a popular Ruby IDE, **RubyMine**, to visualize the database for demonstrative purposes only.

You do not need to download RubyMine during your Bloc course. Feel free to discuss it with your mentor and of course download it if you both feel like it will be helpful, but it is not a requirement for Bloc.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Recap

Concept	Description
Active Record Models	Active Record Models are the layer responsible for representing business data and logic. They facilitate the creation and use of objects whose data requires persistent storage to the database.

Generating Models

The `rails generate` command uses templates to create models, controllers, mailers, and more. When used to generate a model, it creates the Ruby class, test spec, and Active Record database migration.

Active Record Migrations

Active Record Migrations allow you to evolve your database schema over time. They use Ruby so that you don't have to write SQL by hand.

How would you rate this checkpoint and assignment?



14. Rails: Models

Assignment

Discussion

Submission

14. Rails: Models

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

Users may desire the ability to ask questions in Bloccit. The Questions model will be similar to the Posts model, but Questions should also have a `resolved` attribute that allows an administrator to mark the question as resolved.

1. Create a new model named `Question`. It should have `title:string`, `body:text`, and `resolved:boolean` attributes.
2. Create another new model named `Answer`. It should reference `Question` and have a `body` attribute.
3. Write the specs for the `Question` and `Answer` models. The specs should ensure that the model attributes can be called as methods, as we did in the checkpoint with `Post` and `Comment` specs.
4. Update `Question` so that it `has_many :answers`.
5. Confirm that `Answer` `belongs_to :question`, in the Rails console.

Assignment

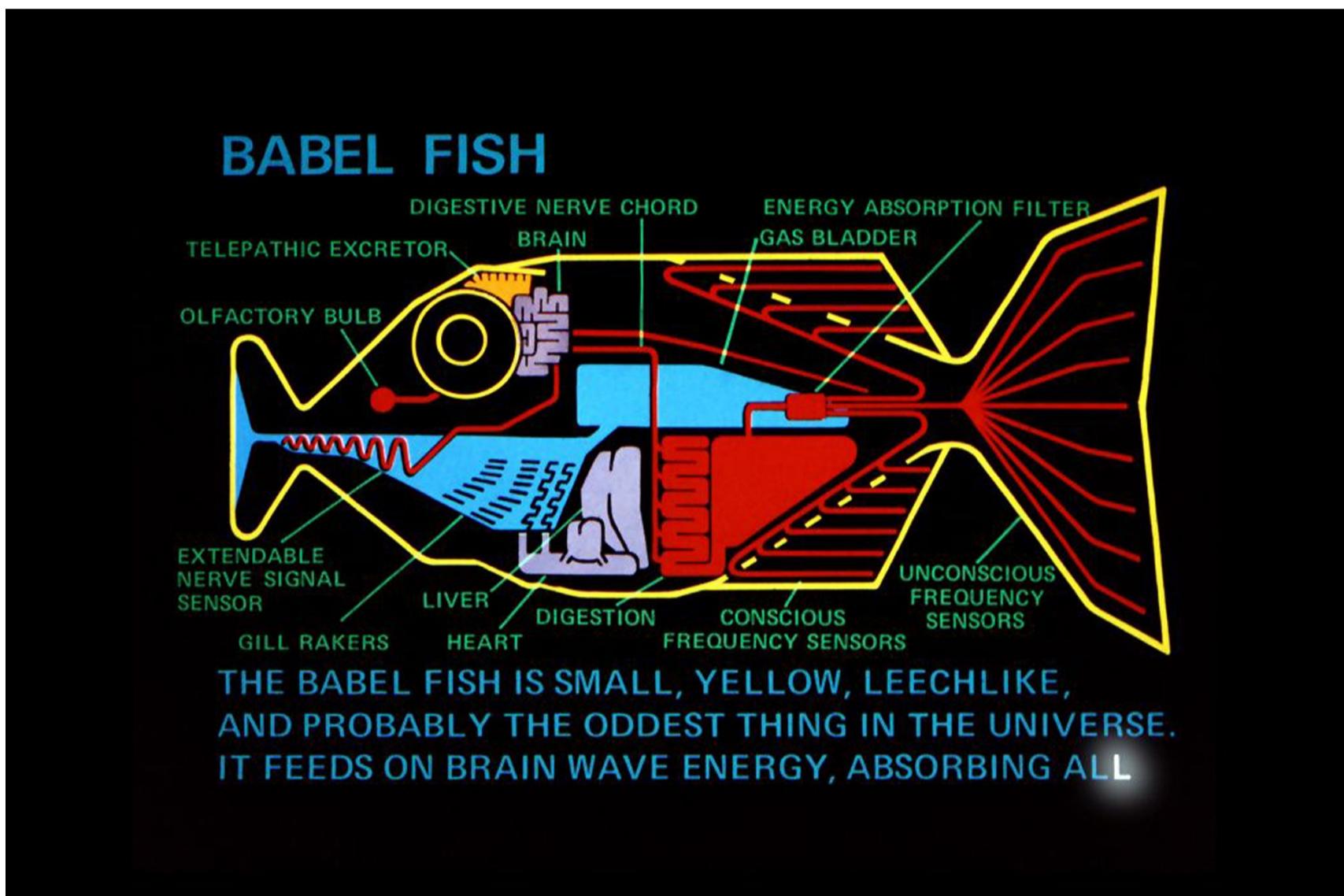
Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.

[←Prev](#)

Submission

[Next→](#)

15 Rails: Object Relational Mapping



“Meanwhile, the poor **Babel fish**, by effectively removing all barriers to communication between different races and cultures, has caused more and bloodier wars than anything else in the history of creation.”

— Douglas Adams

Overview and Purpose

In this checkpoint you'll learn more about the concept of an ORM, Object Relational Map.

Objectives

After this checkpoint, you should be able to:

- Use the Rails console to fetch data.
- Give the definition of an ORM.
- Discuss the benefits and pitfalls of an ORM.
- Use `pry-rails` to interact with your Rails application.

Object Relational Mapping

As we learned in the last checkpoint, a model is a Ruby class that must also be

represented as a database table. This implies that Rails must communicate with a database - and it does - but not without some complexity.

Communication between two systems which "speak different languages" is inherently complex because a translation service is required. As a Rails developer, you are essentially using two systems - Rails, which speaks Ruby, and a database, which speaks SQL.

Object Relational Mapping, or ORM is similar to a translation service, in that it provides a way for Rails developers to manipulate a database using Ruby, rather than writing SQL. Rails employs an ORM library named `ActiveRecord` to provide this translation service. To explore how Rails leverages ORM, we'll experiment with the Rails console. Let's watch a video introducing SQL and the `ActiveRecord` ORM:

Rails Console

The Rails console loads our application in a shell, and provides access to Rails methods, app-specific methods, persisted data, and Ruby. To launch the console from the command line, enter:

Terminal

```
$ rails c
```

And you should see the following message and prompt, or something very similar:

Console

```
Loading development environment (Rails 4.2.5)
2.2.1 :001 >
```

Because the console provides access to our application code, we can create posts and comments within the console, from the command line. Let's create a new post instance:

Console

```
> Post.create(title: "First Post", body: "This is the first post in our system")
```

Creating a post would not be possible in IRB because posts and comments are specific to our application. Ruby (which is the only language that IRB understands) would not know what a "Post" is, and would throw an error.

Here's what we did:

- Called the `create` method on `Post`. This created a new row in the `posts` table. The `create` method is not Ruby - it's part of the `ActiveRecord` class that `Post` inherits from. The first line in our `post.rb` file - `class Post < ActiveRecord::Base` - declares this inheritance and gives `Post` access to the `create` method.
- Passed a hash to the `create` method. The hash was comprised of two keys: `title` and `body`, and two values.

Earlier we stated that Rails and the database don't speak the same language - but we just created a new database row via a Rails method, in the Rails console. This is ORM at work. The **create method** is part of the `ActiveRecord`, which is Rails' ORM library. `create` translates this:

`Post.create(title: "First Post", body: "This is the first post in our system")` into SQL. We'll evaluate the resulting SQL in the next section.

SQL

Structure Query Language, or SQL, is the common language for all databases. Though some database technologies employ their own flavor of SQL, all SQL flavors are similar in syntax and usage. Active Record is a robust ORM library, and translates Rails code into a specific flavor of SQL, which it detects automatically from the database. This means that we don't need to worry about the nuanced SQL flavors. Let's review the SQL that was executed when we called `create` - it was printed in our Rails console:

Console

```
(0.1ms) begin transaction
# #1
SQL (0.8ms) INSERT INTO "posts" ("title", "body", "created_at", "updated_at") VALUES
# #2
(0.6ms) commit transaction
=> #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

At #1, we add a row to the posts table using the `INSERT INTO` SQL statement.

"`title`", "`body`", "`created_at`", "`updated_at`", are the column names (i.e. attributes) on the `posts` table. The list of values after `VALUES (?, ?, ?, ?)` in brackets (`[["title", "First Post"] ...]`) are values that correspond to the column names. The `created_at` and `updated_at` columns are default columns that Rails adds automatically, which is why we didn't need to specify them in the `create` call.

At #2, we commit the transaction which executes `INSERT INTO`. Commit statements end a SQL transaction and make all changes permanent. A transaction is one or more SQL statements that a database treats as a single unit.

We now have one row in the posts table.

Retrieving Information

It is important to remember that a row in a table corresponds to an instance of a class. Like a class instance, a row in a database table is unique. ORM allows us to retrieve information stored in a row and map it to a class instance that we create in our application. Let's retrieve a row from the posts table and map it to an instance of the `Post` class:

Console

```
> post = Post.first
# #3
Post Load (0.2ms) SELECT "posts".* FROM "posts" ORDER BY "posts"."id" ASC LIMIT 1
=> #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

At #3 `Post.first` executes a `SELECT` SQL statement and fetches the first row from the

posts table. `SELECT` is used to fetch a set of records from one or more tables.

After the first row is fetched, `ActiveRecord` converts the row's data into an instance of `Post`, or a post object. This post object is then assigned to the `post` variable. `ActiveRecord` makes this conversion from a database record to Ruby object possible.

Now that our instance is assigned, print it to view its value:

Console

```
> post  
=> #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

`post` is populated by the first row of data (currently the *only* row of data) in our posts database table.

Let's add a comment to the post we retrieved:

Console

```
> post.comments.create(body: "First comment!")  
  (0.1ms)  begin transaction  
 SQL (0.4ms)  INSERT INTO "comments" ("body", "post_id", "created_at", "updated_at")  
  (0.7ms)  commit transaction  
=> #<Comment id: 1, body: "First comment!", post_id: 1, created_at: "2015-06-10 19:50:00", updated_at: "2015-06-10 19:50:00"
```

Because we chained the method calls - `post.comments.create` - ActiveRecord interpreted this as "create a new comment for the first post". If we didn't specify the post to create a comment for, ActiveRecord would not have been able to update the `post_id`, which is critical because it defines the relationship between posts and comments. Inspect `post.comments`:

Console

```
> post.comments  
Comment Load (2.6ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = 1  
=> #<ActiveRecord::Associations::CollectionProxy:0x007f1d13e000> [#<Comment id: 1, body: "First comment!", post_id: 1, created_at: "2015-06-10 19:50:00", updated_at: "2015-06-10 19:50:00"]
```

`post.comments` returns an `ActiveRecord::Association` because a comment *depends* on a given post. We'll explore associations in the next section.

ActiveRecord Associations

We defined the relationships between posts and comments in their respective classes,

with `has_many` and `belongs_to`. These relationships are known as **associations**.

The `belongs_to :post` declaration in `Comment` generates a `post` method for each comment, giving us the ability to call `.post` on an instance of `Comment` and retrieve the associated post. The database stores this relationship, by keeping a `post_id` (foreign key) for each comment.

Retrieve the first comment in the comments table, and assign it to a `comment` variable:

Console

```
> comment = Comment.first
```

Fetch the post that is associated with `comment`:

Console

```
> comment.post
Post Load (0.4ms)  SELECT "posts".* FROM "posts" WHERE "posts"."id" = ? LIMIT 1
=> #<Post id: 1, title: "First Post", body: "This is the first post in our system",
```

Let's create another comment on `post`:

Console

```
> post.comments.create(body: "Second comment!")
```

The `has_many :comments` declaration in `Post` is the counterpart of `belongs_to :post`. The posts table makes no reference to comments. There's no `comment_id` column or array of `comment_ids` in the posts table. Instead, this relationship is stored in the comments table exclusively. A post retrieves its associated comments by fetching all the comments with a `post_id` that matches the `id` of the post. Storing the relationship in the comments table is a database strategy to allow data to be intersected or joined in an efficient manner.

Now that we have two comments associated with a single post, let's iterate over them using Ruby:

Console

```
# #4
> post.comments.each { |comment| p comment.body }
# #5
Comment Load (0.2ms)  SELECT "comments".* FROM "comments" WHERE "comments"."post_id" = ?
"First comment!"
"Second comment!"
=> [#<Comment id: 1, body: "First comment!", post_id: 1, created_at: "2015-06-10 19:15:00", updated_at: "2015-06-10 19:15:00">]
```

At #4, the `|comment|` block argument represents an instance of `Comment` with each iteration. We call `body` on each comment instance to retrieve the comment's body attribute from the database.

At #5, the `SELECT` statement fetches all the comments with the given `post_id`.

Recap

Concept	Description
Rails Console	Provides command line access to a Rails application and Ruby.
SQL	SQL is a language for communicating with a relational database.
Object Relational Mapping	Object-Relational Mapping (ORM) is a design pattern that connects the objects of an application to tables in a database. Using ORM, the properties and relationships of objects in an application can be connected to a database without the need to write SQL statements.
ActiveRecord	ActiveRecord is Rails' ORM library.

How would you rate this checkpoint and assignment?

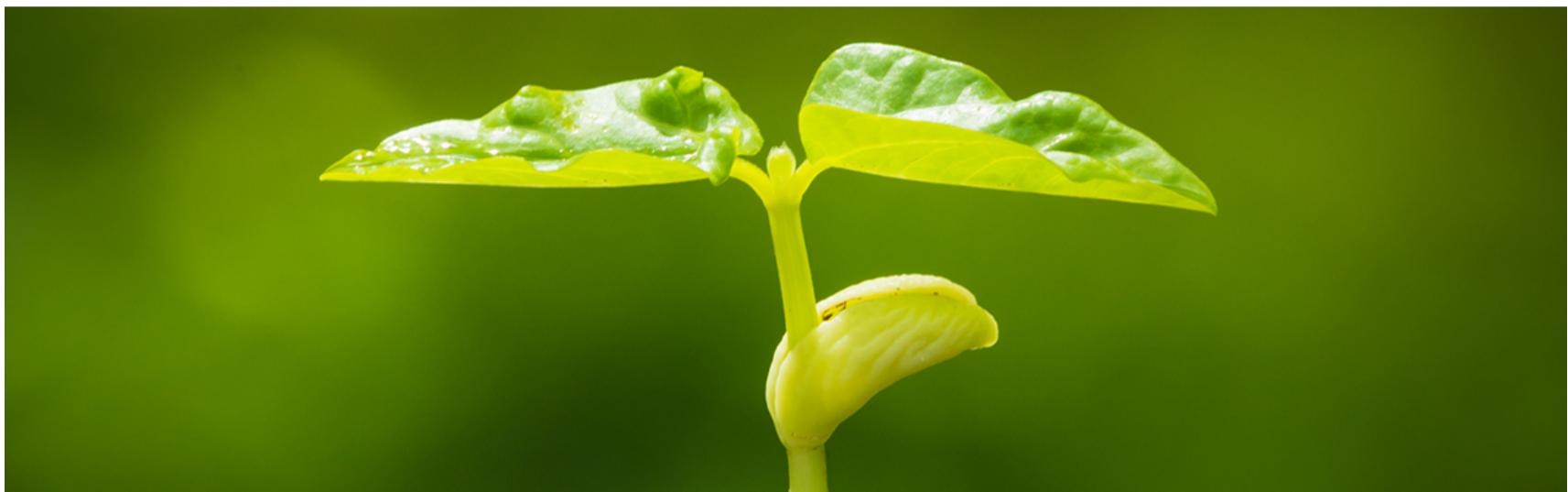


[←Prev](#)

Submission

[Next→](#)

16 Rails: Seeding Data



“If you want to grow a giant redwood, you need to make sure the seeds are ok, nurture the sapling, and work out what might potentially stop it from growing all the way along. Anything that breaks it at any point stops that growth.”

— **Elon Musk**

Overview and Purpose

Visually testing of applications is a great way to spot bugs. In this checkpoint, you'll learn how to populate your application with test data. This test data will consist of script-generated fake posts and comments, added to the development database only. This will allow you to see how the application will behave. Without fake data some of your HTML wouldn't render in your views. It is easier to spot issues with your application if you have some fake data to play around with.

Objectives

After this checkpoint, you should be able to:

- Seed a Ruby on Rails application with test data.

Seed Data

We've added database records manually through the Rails Console, but in a development environment it's helpful to have lots of data to work with. It would be monotonous to add many records manually, so we'll programmatically add fake data to Blocxit.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Generating Seed Data

Open `db/seeds.rb` and remove the commented lines. `seeds.rb` is a script (a small utility

program) we can run to seed the development database with test data. Add the following code:

db/seeds.rb

```
+ require 'random_data'

+ # Create Posts
+ 50.times do
# #1
+   Post.create!(
# #2
+     title: RandomData.random_sentence,
+     body: RandomData.random_paragraph
+   )
+ end
+ posts = Post.all
+
+ # Create Comments
# #3
+ 100.times do
+   Comment.create!(
# #4
+     post: posts.sample,
+     body: RandomData.random_paragraph
+   )
+ end
+
+ puts "Seed finished"
+ puts "#{Post.count} posts created"
+ puts "#{Comment.count} comments created"
```

At #1, we use `create!` with a *bang* (`!`). Adding a `!` instructs the method to raise an error if there's a problem with the data we're seeding. Using `create` without a *bang* could fail without warning, causing the error to surface later.

At #2, we use methods from a class that does not exist yet, `RandomData`, that will create random strings for `title` and `body`. Writing code for classes and methods that don't exist is known as "wishful coding" and can increase productivity because it allows you to stay focused on one problem at a time.

At #3, we call `times` on an `Integer` (a number object). This will run a given block the specified number of times, which is 100 in this case. The end result of calling `times` is similar to that of a loop, but in this use-case, it is easier to read and thus more idiomatic.

At #4, we call `sample` on the array returned by `Post.all`, in order to pick a random post to associate each comment with. `sample` returns a random element from the array every

time it's called.

RandomData

`RandomData` does not exist, so let's create it. Create a file named `random_data.rb` in the `lib` (short for "library") directory:

Terminal

```
$ touch lib/random_data.rb
```

Open `random_data.rb` and add the following code:

```
# #5
+ module RandomData
# #6
+   def self.random_paragraph
+     sentences = []
+     rand(4..6).times do
+       sentences << random_sentence
+     end
+
+     sentences.join(" ")
+   end
+
# #7
+   def self.random_sentence
+     strings = []
+     rand(3..8).times do
+       strings << random_word
+     end
+
+     sentence = strings.join(" ")
+     sentence.capitalize << "."
+   end
+
# #8
+   def self.random_word
+     letters = ('a'..'z').to_a
+     letters.shuffle!
+     letters[0,rand(3..8)].join
+   end
+ end
```

At #5, we define `RandomData` as a module because it is a standalone library with no dependencies or inheritance requirements. Modules help keep common functions organized and reusable throughout our application. Unlike classes, we can't instantiate or inherit from modules. Instead we use them as **mixins** to add functions to multiple classes.

At #6, we define `random_paragraph`. We set `sentences` to an array. We create four to six random sentences and append them to `sentences`. We call `join` on `sentences` to combine each sentence in the array, passing a space to separate each sentence. `join` combines each sentence into a full paragraph (as a string).

At #7, we follow the same pattern as we did in #6 to create a sentence with a random number of words. After we generate a sentence, we call `capitalize` on it and append a period. `capitalize` returns a copy of `sentence` with the first character converted to uppercase and the remainder of the sentence converted to lowercase.

At #8, we define `random_word`. We set `letters` to an array of the letters `a` through `z` using `to_a`. We call `shuffle!` on `letters` *in place*. If we were to call `shuffle` without the bang (!), then `shuffle` would return an array rather than shuffle in place. We `join` the zeroth through nth item in `letters`. The nth item is the result of `rand(3..8)` which returns a random number greater than or equal to three and less than eight.

Let's make `random_data.rb` accessible to all our specs going forward. Adding it to `application.rb` autoloads any references to the `lib` directory used by our code:

```
...
# Settings in config/environments/* take precedence over those specified here.
# Application configuration should go into files in config/initializers
# -- all .rb files in that directory are automatically loaded.
+   config.autoload_paths << File.join(config.root, "lib")
end
end
```

Drop the database and run `seeds.rb` by typing:

Terminal

```
$ rails db:reset
Seed finished
50 posts created
100 comments created
```

Open the Rails console to randomly check some results:

Console

```
$ rails c  
> p = Post.find(3)
```

We called another `ActiveRecord` class method, `find`, on `Post` and passed it a value which represents a unique post id. `find` will return the instance (row) of post data which corresponds to an id of 3. You should see an output with a funny looking `title` and `body` as that's what `RandomData` created in `seeds.rb`. Run the following methods to view how many comments the given post has:

Console

```
> p.comments.count
```

`count` is an `ActiveRecord` method that can be called on an `ActiveRecord` relation. `p.comments` returns an `ActiveRecord` relation, so `count` is a valid method to call on it.

Type `exit` to exit the console.

In the following video, we demonstrate seeding:



BLOC

Bloccit: Seeding the Database

1:31



Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Recap

Concept	Description
Modules	A module is independent code that contains everything necessary to implement only one feature.

Seeding Data

Rails' seeding feature allows the database to be populated with initial data based on the contents of `seeds.rb`.

How would you rate this checkpoint and assignment?



16. Rails: Seeding Data

Assignment

Discussion

Submission

16. Rails: Seeding Data

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

By running `rails db:seed`, new data is added to your database. Occasionally you'll seed unique data using `seeds.rb` without erasing or duplicating existing data. This concept is called **idempotence**.

Idempotent code can be run many times or one time with identical results. For example, `i = 4` is idempotent but `i = i + 2` is not.

When developing web apps, it's important to think about what code is idempotent because code might get run more than once, like when a user refreshes a page.

1. Add a post with a unique **title** and **body** to `seeds.rb`. Before creating a unique post, check whether it already exists in the database using the **title** and **body**. Only seed the post if it doesn't already exist. Use the `find_or_create_by!` method.
2. Run `rails db:seed` a couple of times, confirm that your unique post has only been seeded once using the Rails console.
3. Repeat steps one and two to create a comment with a unique **body**. Your

[←Prev](#)

Submission

[Next→](#)

17 Rails: Introduction to CRUD



“Physicists analyze systems. Web scientists, however, can create the systems.”

— **Tim Berners-Lee**

Overview and Purpose

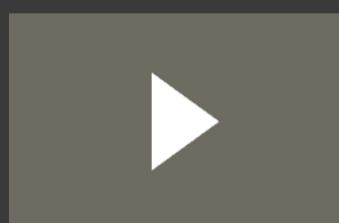
In this checkpoint you'll learn about CRUD.

Objectives

After this checkpoint, you should be able to:

- Define CRUD.
- Elaborate on resourceful routing in Rails.
- Explain how HTTP requests are used.
- Explain HTTP status codes and their purpose.

Viewing Posts



BLOC

Intro: Bloccit - CRUD Intro

0:26



Resources

"Resource" is a term used to describe an object that needs to be accessible by users and thus requires interactive capabilities.

It's an abstract term that encapsulates different parts of a Rails application. A resource is not one single entity, but rather, a collection of models, views, controllers, and routes.

For example, in Bloccit we have posts. We already created a model for our posts, and now we need to write the other pieces to complete the post resource. This includes the views, which visually represent posts within Bloccit; the controller, that will perform operations on a post such as creation or deletion; and the routes, which will map requests from clients (a web browser on a computer or mobile device) to the posts to perform the

actions the client requests.

In this checkpoint, we'll learn how to build resources, and how to interact with them through a concept known as CRUD. CRUD stands for **C**reate **R**ead **U**pdate **D**elete. Bloccit needs to be able to perform CRUD operations on our models (such as posts) so that users can perform these actions from the GUI we present to them (the views).

Bloccit also needs routes so that when a client makes a request to our application, such as requesting to delete a post, Bloccit knows where to look to perform that action. In other words, our routes act like a traffic controller. Like a traffic controller telling cars where to go, our routes tell the client requests where to go within Bloccit.

Let's say a user wants to delete a post. From a high level, the user clicks a button that says "Delete Post". The button sends a request to our server (application) across the Internet that points to a specific web URL. Our routing maps the URL to a controller action within Bloccit. In this example, the web URL will map to an action that deletes a post.

For example, imagine the web URL is `www.myapplication.com/posts/1/delete/`. When a user sends a request to that URL, our application looks for the route (using a table which we will explore later) and determines which controller action it should perform. This action in turn interacts with the model and ultimately the database, modifying the state of our application by deleting the post. These pieces work together to form the posts resource within our application.

This [Stack Overflow post](#) provides further explanation of resources in Rails.

Git

Create a new Git branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Generating a Resource

A resource has three components which align to MVC architecture: a model, view(s), and a controller. We'll build a resource for posts first, and because we already have a post model, we'll create a controller next:

Terminal

```
$ rails generate controller Posts index show new edit
```

We passed the controller generator five arguments, including the resource name, `Posts`. Review the following output:

Terminal

```
create  app/controllers/posts_controller.rb
  route  get 'posts/edit'
  route  get 'posts/new'
  route  get 'posts/show'
  route  get 'posts/index'
invoke  erb
create  app/views/posts
create  app/views/posts/index.html.erb
create  app/views/posts/show.html.erb
create  app/views/posts/new.html.erb
create  app/views/posts/edit.html.erb
invoke  rspec
create  spec/controllers/posts_controller_spec.rb
create  spec/views/posts
create  spec/views/posts/index.html.erb_spec.rb
create  spec/views/posts/show.html.erb_spec.rb
create  spec/views/posts/new.html.erb_spec.rb
create  spec/views/posts/edit.html.erb_spec.rb
invoke  helper
create  app/helpers/posts_helper.rb
invoke  rspec
create  spec/helpers/posts_helper_spec.rb
invoke  assets
invoke  js
create  app/assets/javascripts/posts.js
invoke  scss
create  app/assets/stylesheets/posts.scss
```

Open `routes.rb` and view the `get` method calls the controller generator added:

config/routes.rb

```
Rails.application.routes.draw do
  get "posts/index"

  get "posts/show"

  get "posts/new"

  get "posts/edit"

  get "welcome/index"
  get "welcome/about"

  root 'welcome#index'
end
```

The generated `get` method calls create routes for the post resource, but Rails offers a more succinct syntax. Let's refactor `routes.rb` with the `resources` method:

config/routes.rb

```
Rails.application.routes.draw do
-   get "posts/index"
-
-   get "posts/show"
-
-   get "posts/new"
-
-   get "posts/edit"

# #1
+   resources :posts

# #2
-   get "welcome/index"
-
-   get "welcome/about"
+   get 'about' => 'welcome#about'

  root 'welcome#index'
end
```

At **#1**, we call the `resources` method and pass it a `Symbol`. This instructs Rails to create post routes for creating, updating, viewing, and deleting instances of `Post`. We'll review the precise URIs created in a moment.

At **#2**, we remove `get "welcome/index"` because we've declared the index view as the root

view. We also modify the `about` route to allow users to visit `/about`, rather than `/welcome/about`.

The Rails router uses `routes.rb`. Let's watch a video exploring the important role the Rails router plays:

Run `rails routes` from the command line:

Terminal

```
$ rails routes
# #3
Prefix Verb URI Pattern          Controller#Action
# #4
  posts GET  /posts(.:format)      posts#index
         POST /posts(.:format)      posts#create
 new_post GET  /posts/new(.:format) posts#new
edit_post GET  /posts/:id/edit(.:format) posts#edit
  post GET  /posts/:id(.:format)    posts#show
# #5
    PATCH /posts/:id(.:format)    posts#update
      PUT /posts/:id(.:format)    posts#update
    DELETE /posts/:id(.:format)   posts#destroy
 about GET  /about(.:format)       welcome#about
  root GET  /                      welcome#index
```

At #3, we see a header with Prefix, Verb, URI Pattern, and Controller#Action. The verbs correspond to HTTP Request Methods. They specify the action to be done on the specified resource. For example, a GET asks for data, a POST creates data, a PATCH or PUT updates data, and a DELETE deletes data. Standard HTTP verbs make it simpler for different systems to interact. By using a well-documented and well-known system like HTTP, the behavior of different operations is clear and reliable.

At #4, Rails created a route to `/posts` which requires a GET. The route maps to the `index` method in `PostsController`.

At #5, we see PATCH and PUT verbs, which are similar. PUT updates data by sending the complete resource, whereas PATCH sends just the changes.

CRUD

CRUD stands for Create Read Update Delete. CRUD actions align with controller HTTP verbs and controller actions in a Rails app.

CRUD Action	HTTP Verb	Rails Action(s)
Create	POST	create
Read	GET	new/show/index/edit
Update	PUT/PATCH	update

[Delete](#)[DELETE](#)[destroy](#)

Start your Rails server and visit <http://localhost:3000/posts>. We see a `NameError` like the one below:

The screenshot shows a browser window with the URL `localhost:3000/` in the address bar. The title bar says "NameError in Posts#index". The main content area displays the error message: "Showing /Users/mike/code/bloccit/app/views/layouts/application.html.erb where line #13 raised: undefined local variable or method `welcome_index_path' for #<#<Class:0x007fa8002155e8>:0x007fa800e21f30>". Below this, a "Extracted source (around line #13):" block shows the following code:

```
11 <div class="container">
12   <ul class="nav nav-tabs">
13     <li><%= link_to "Home", welcome_index_path %></li>
14     <li><%= link_to "About", welcome_about_path %></li>
15   </ul>
16 </div>
```

Line 13 is highlighted in red. At the bottom of the error page, there are links for "Rails.root: /Users/mike/code/bloccit", "Application Trace", "Framework Trace", and "Full Trace". A file path "app/views/layouts/application.html.erb:13:in `app_views_layouts_application_html_erb_3160815371151985725_70179773060140'" is also shown.

Request

Parameters:

None

[Toggle session dump](#)

[Toggle env dump](#)

Response

Headers:

None

This error happens because we're using a generated path helper that no longer exists. When we changed our routes, we changed our path helpers. In particular, we changed `welcome_index_path` to `root_path` and `welcome_about_path` to `about_path`.

Update the application layout to reflect the new paths:

app/views/layouts/application.html.erb

```
...
-      <li><%= link_to "Home", welcome_index_path %></li>
-      <li><%= link_to "About", welcome_about_path %></li>
+      <li><%= link_to "Bloccit", root_path %></li>
+      <li><%= link_to "About", about_path %></li>
...
```

Save the changes and refresh the page. We see the default HTML created by the controller generator at <http://localhost:3000/posts> – this view is the index page. We can deduce this from the `rails routes` output above, specifically this line:

Output

posts	GET	/posts(.:format)	posts#index
-------	-----	------------------	-------------

We can see that the `/posts` route (column three) is associated with the `posts#index` controller action (column four).

Index Action

Let's use TDD to write the `index` action in `PostsController`. When we generated our controller, Rails created a basic spec for `PostsController`:

```
spec/controllers/posts_controller_spec.rb
```

```

require 'rails_helper'

# #6
RSpec.describe PostsController, type: :controller do

  describe "GET index" do
    it "returns http success" do
      # #7
      get :index
      expect(response).to have_http_status(:success)
    end
  end

  describe "GET show" do
    it "returns http success" do
      get :show
      expect(response).to have_http_status(:success)
    end
  end

  describe "GET new" do
    it "returns http success" do
      get :new
      expect(response).to have_http_status(:success)
    end
  end

  describe "GET edit" do
    it "returns http success" do
      get :edit
      expect(response).to have_http_status(:success)
    end
  end

end

```

At #6, RSpec created a test for `PostsController`. `type: :controller` tells RSpec to treat the test as a `controller` test. This allows us to simulate controller actions such as HTTP requests.

At #7, the test performs a `GET` on the index view and expects the response to be successful.

`have_http_status` is an RSpec **matcher** which encapsulates this logic.

`have_http_status(:success)` checks for a response code of **200**, which is the standard HTTP response code for success.

The remaining tests follow the same pattern.

Let's add another test to `posts_controller_spec.rb` to define the expected behavior of the

`PostsController#index` :

spec/controllers/posts_controller_spec.rb

```
require 'rails_helper'

RSpec.describe PostsController, type: :controller do
  # #8
  + let(:my_post) { Post.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)

    describe "GET index" do
      it "returns http success" do
        get :index
        expect(response).to have_http_status(:success)
      end

      + it "assigns [my_post] to @posts" do
      +   get :index
      # #9
      +   expect(assigns(:posts)).to eq([my_post])
      + end
    end

    # #10
    - describe "GET show" do
      - it "returns http success" do
      -   get :show
      -   expect(response).to have_http_status(:success)
      - end
    - end

    - describe "GET new" do
      - it "returns http success" do
      -   get :new
      -   expect(response).to have_http_status(:success)
      - end
    - end

    - describe "GET edit" do
      - it "returns http success" do
      -   get :edit
      -   expect(response).to have_http_status(:success)
      - end
    - end
  end
end
```

```

+ # describe "GET show" do
+   # it "returns http success" do
+     # get :show
+     # expect(response).to have_http_status(:success)
+   # end
+ # end

+ # describe "GET new" do
+   # it "returns http success" do
+     # get :new
+     # expect(response).to have_http_status(:success)
+   # end
+ # end

+ # describe "GET edit" do
+   # it "returns http success" do
+     # get :edit
+     # expect(response).to have_http_status(:success)
+   # end
+ # end

end

```

At #8, we create a post and assign it to `my_post` using `let`. We use `RandomData` to give `my_post` a random title and body.

At #9, because our test created one post (`my_post`), we `expect` `index` to return an array of one item. We use `assigns`, a method in `ActionController::TestCase`. `assigns` gives the test access to **"instance variables assigned in the action that are available for the view"**.

At #10, we comment out the tests for `show`, `new`, and `edit` since we won't write the implementation until later.

Run the spec:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb
```

We see output indicating our new test is failing. Let's write the implementation of `index` to get both tests passing. In the `PostsController`, add the following to the `index` method:

`app/controllers/posts_controller.rb`

```
class PostsController < ApplicationController
  def index
    # #11
    +   @posts = Post.all
  end

  def show
  end

  def new
  end

  def edit
  end
end
```

At #11, we declare an instance variable `@posts` and assign it a collection of `Post` objects using the `all` method provided by `ActiveRecord`. `all` returns a collection of `Post` objects.

Run the spec again. The tests for `index` now pass. Our controller functions per the expectations of our spec for `index`. Let's write the associated view:

app/views/posts/index.html.erb

```
- <h1>Posts#index</h1>
- <p>Find me in app/views/posts/index.html.erb</p>
+ <h1>All Posts</h1>
# #12
+ <% @posts.each do |post| %>
+   <p><%= link_to post.title, post_path(post.id) %></p>
+ <% end %>
```

At #12, we iterate over each post in `@posts`. For each post we create a link with `post.title` as the text that links to `/posts/id`, with the id from the `post.id`. Instance variables created in a controller method are available in its associated view. Since we create and assign `@posts` in `PostsController#index`, we can use it in the posts `index` view.

Refresh <http://localhost:3000/posts> to see all the posts in the database.

Inspect the `post_path` method using `rails routes | grep 'posts#show'`. It requires an id to route to the correct post:

terminal

```
$ rails routes | grep 'posts#show'  
post GET      /posts/:id(.:format)          posts#show
```

We passed the id of the post instance to the `post_path` method. `post_path` used this id to create the path. We can pass the post instance to get the same path. The `post_path` method will derive the id:

```
link_to post.title, post_path(post)
```

Rails simplifies this further by allowing us to skip the `post_path` method altogether. If we call `link_to` with the object to which we're linking as a second argument, the `link_to` method will detect the object, parse its id, and create the path using the id. Using that shortcut, we can simplify the posts **index** view:

app/views/posts/index.html.erb

```
<h1>All Posts</h1>  
<% @posts.each do |post| %>  
+  <p><%= link_to post.title, post %></p>  
-  <p><%= link_to post.title, post_path(post.id) %></p>  
<% end %>
```

This is an example of the advantages of Rails' "convention over configuration" approach.

Let's use Bootstrap to style the posts **index** view:

app/views/posts/index.html.erb

```
<h1>All Posts</h1>  
<% @posts.each do |post| %>  
-  <p><%= link_to post.title, post %></p>  
+  <div class="media">  
+    <div class="media-body">  
+      <h4 class="media-heading">  
+        <%= link_to post.title, post %>  
+      </h4>  
+    </div>  
+  </div>  
<% end %>
```

Bloccit About

All Posts

Fugiat itaque similique natus modi nihil.

Aut blanditiis nemo consequatur vel asperiores qui et perspiciatis.

Sit ut incident dicta.

Dignissimos quibusdam amet aliquid voluptas.

Veritatis aut architecto qui error rerum consectetur suscipit.

Qui asperiores delectus ut velit assumenda voluptatem rerum.

Sint maxime dolorum praesentium voluptatem non occaecati enim esse.

Dolorem doloremque et ratione.

Volutpas voluptatem ullam quae quam harum modi aut iste.

Aliquid dolorum non nam.

Consequatur aut iusto aut tenetur.

Dolores dignissimos quia omnis.

Rem et aperiam itaque cupiditate totam vero ratione.

Qui est eum sint eligendi in ut veritatis cum.

In mollitia iure ipsa.

Suscipit ullam ut repellendus molestias ut doloremque.

Distinctio asperiores veritatis et cupiditate dolorum maxime autem.

Dicta fugiat et eaque ex nam nostrum ratione.

Dolor reiciendis nesciunt earum cum quisquam ducimus fugiat illo.

Rerum numquam molestiae impedit aspernatur quia.

Nihil enim quia suscipit et.

Bootstrap provides the classes we added to the `<div>` and `<h4>` tags to improve our view.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Heroku

Push your app to production:

Terminal

```
$ git push heroku master
```

Open your production app in your browser. If you can't remember the URL, use `heroku apps:info`.

If you're using Cloud9, open your app in a browser window outside of the Cloud9 window.

When you visit the `/posts` URL, you should see "We're sorry, but something went wrong." This is because you haven't migrated your database changes in production. As a consequence, your production app's database doesn't have posts or comments tables in it. Let's run the migrations:

Terminal

```
$ heroku run rails db:migrate
```

Refresh the page on production. Although the error has gone away, there are no posts! This is because you ran `rails db:seed` in your development environment, and not the production environment. It's not a good idea to seed your production environment with fake data. You don't want fake data intertwined with real data, but you'll create real data in production soon.

Recap

Concept	Description
<code>rails generate</code>	The <code>rails generate</code> command creates controllers from templates. The <code>generate</code> command can also generate controller actions and their corresponding views.
Resourceful Routing	Rails' resource routing permits the declaration of common routes for a controller. It allows you to declare the routes for the "index", "show", "new", "edit", "create", "update" and "destroy" actions in a single line of code.
<code>rails routes</code>	The <code>rails routes</code> command lists all routes, in the same order as <code>routes.rb</code> .
CRUD	Create, Read, Update, and Delete are the four basic functions of persistant storage.
HTTP Request Methods	HTTP defines methods (verbs) to indicate the desired action to be performed on the identified resource.
HTTP Status Codes	Status codes are numbers returned by HTTP which indicate the result of a request.

17. Rails: Introduction to CRUD

 **Assignment**

 **Discussion**

 **Submission**

17. Rails: Introduction to CRUD

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git branch for this assignment. See **Git Checkpoint Workflow: Before Each Assignment** for details.

Because our posts will be user generated, some of them may contain offensive or sensitive content. Since we don't have real content, we will pretend that every fifth post contains sensitive content to test a potential censorship feature. Your assignment is to overwrite the `title` of the first post and every fifth post after it with the text **SPAM**. You should change the data itself, not just change the display of the data.

Here are some hints to help you:

1. Does this logic belong in a model, a view, or a controller?
2. Which classes should encapsulate this logic?
3. Recall our usage of the **modulus** operator from the Ruby checkpoints. This may help you identify each fifth instance of `Post`.

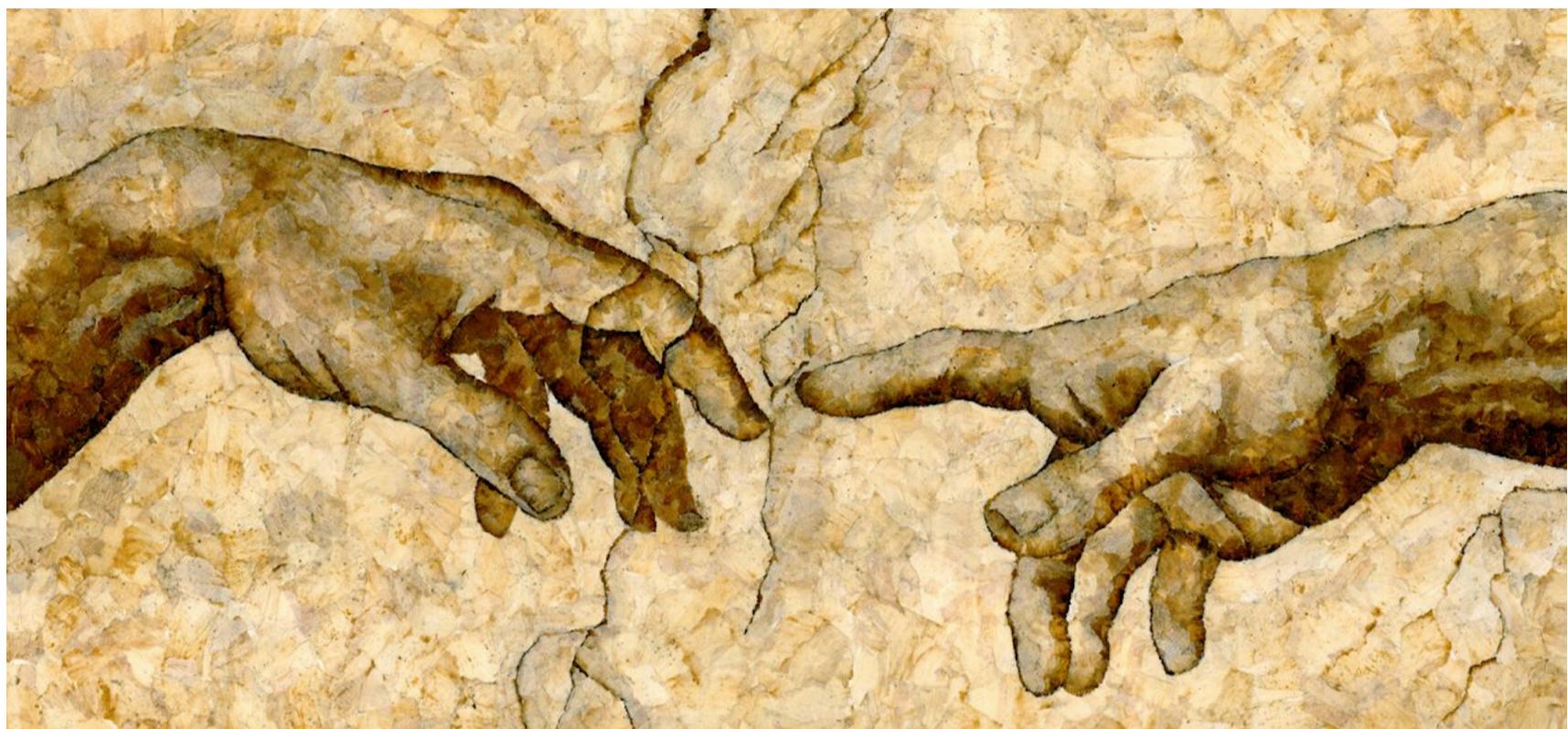
Commit your assignment in Git. See **Git Checkpoint Workflow: After Each Assignment** for details. Submit your commit to your mentor.

[←Prev](#)

Submission

[Next→](#)

18 Rails: CRUD



“It's easy to attack and destroy an act of creation. It's a lot more difficult to perform one.”

— **Chuck Palahniuk**

Overview and Purpose

In this checkpoint you'll learn more about the CR in CRUD.

Objectives

After this checkpoint, you should be able to:

- Explain the C in CRUD.
- Explain the R in CRUD.
- Describe the `params` hash.
- Use `form_for` to create HTML forms.

Reading and Creating posts

Git

Create a new Git branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Creating Posts

In this checkpoint we'll develop, test-first, the functionality for creating posts. We'll also implement a user interface (UI) so that Bloccit users can create posts.

Let's write the controller tests:

```
...

- # describe "GET new" do
- #   it "returns http success" do
- #     get :new
- #     expect(response).to have_http_status(:success)
- #   end
- # end

# #1
+ describe "GET new" do
+   it "returns http success" do
+     get :new
+     expect(response).to have_http_status(:success)
+   end
+
# #2
+   it "renders the #new view" do
+     get :new
+     expect(response).to render_template :new
+   end
+
# #3
+   it "instantiates @post" do
+     get :new
+     expect(assigns(:post)).not_to be_nil
+   end
+
+   describe "POST create" do
# #4
+     it "increases the number of Post by 1" do
+       expect{ post :create, params: { post: { title: RandomData.random_sentence, body: RandomData.random_sentence } } }.to change(Post, :count).by(1)
+     end
+
# #5
+     it "assigns the new post to @post" do
+       post :create, params: { post: { title: RandomData.random_sentence, body: RandomData.random_sentence } }
+       expect(assigns(:post)).to eq Post.last
+     end
+
# #6
+     it "redirects to the new post" do
+       post :create, params: { post: { title: RandomData.random_sentence, body: RandomData.random_sentence } }
+       expect(response).to redirect_to Post.last
+     end
```

+ end

...

There are separate `new` and `create` sections at #1 and #4. When `new` is invoked, a new and unsaved `Post` object is created. When `create` is invoked, the newly created object is persisted to the database. This implementation mimics the **RESTful** parts of HTTP. REST is a software architecture style and practice that imposes a set of constraints for building web applications. RESTful architectures follow constraints such as using HTTP verbs and making sure web URLs follow a certain pattern. Discussing all of the constraints that REST recommends is outside the scope of this checkpoint, but we'll continue to use and explore RESTful practices as we build Blocxit.

There are many resources on the web that elaborate on REST and RESTful practices, such as [this video](#).

Per HTTP, GET requests should not generate new data, thus `new` (which is a GET) does not create data. POST requests are used to create new data, thus `create` actually creates the fields in a post.

What do you think would happen if we used post `:new` or get `:create`?

At #2, we `expect PostsController#new` to render the posts **new** view. We use the `render_template` method to verify that the correct template (view) is rendered.

At #3, we `expect` the `@post` instance variable to be initialized by `PostsController#new`. `assigns` gives us access to the `@post` variable, assigning it to `:post`.

At #4, we `expect` that after `PostsController#create` is called with the parameters `{post: {title: RandomData.random_sentence, body: RandomData.random_paragraph}}`, the `count` of `Post` instances (i.e. rows in the posts table) in the database will increase by one.

At #5, when `create` is POSTed to, we `expect` the newly created post to be assigned to `@post`.

At #6, we `expect` to be redirected to the newly created post.

Run the spec and see four failing tests that are caused because we haven't finished `new` or `create` in `PostsController`.

The first two tests pass without our need to add any code to `PostsController`. By default, `new` will render the post's **new** view and return an HTTP success code. If we were to override this default behavior, we'd need to update these tests.

Let's implement the rest of the logic needed in `new`:

```
...
def new
# #7
+  @post = Post.new
end
```

...

At #7, we create an instance variable, `@post`, then assign it an empty post returned by `Post.new`.

Run the specs for `new` and observe that all three `Get new` tests pass:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'GET new'
```

`-e 'GET new'` runs only the specs in the `describe "GET new" do` block. This allows us to run only the specs covering the method we're working on.

The `form_for` Helper Method

Thanks to our updated `new` method in `PostsController`, we have the ability to access a new `Post` instance in the related posts `new` view. Open the `new` view and add the following code:

```
- <h1>Posts#new</h1>
- <p>Find me in app/views/posts/new.html.erb</p>
+ <%= form_for @post do |f| %>
+   <%= f.label :title %>
+   <%= f.text_field :title %>
+
+   <%= f.label :body %>
+   <%= f.text_area :body %>
+
+   <%= f.submit "Save" %>
+ <% end %>
```

Start the Rails server and go to <http://localhost:3000/posts/new>. View the source of the resulting page by right clicking anywhere on the page, and selecting **Inspect Element**.

Search for a form HTML tag and you should see the following HTML code:

```
<!-- #8 -->
<form class="new_post" id="new_post" action="/posts" accept-charset="UTF-8" method="post">
  <label for="post_title">Title</label>
  <input type="text" name="post[title]" id="post_title">

  <label for="post_body">Body</label>
  <textarea name="post[body]" id="post_body"></textarea>

  <input type="submit" name="commit" value="Save">
</form>
```

`form_for` generates this HTML code starting at #8. `form_for`, like `link_to`, is a Rails helper method that generates HTML code. This code allows a user to submit a post title and body, thus creating a new post.

We examine `form_for` in more depth in the following video:

On `http://localhost:3000/posts/new`, enter some text into the title and body fields, then click the **Save** button. You should see an error message:

Unknown action

The action 'create' could not be found for PostsController

The form attempted to submit the data to a create action, but Rails couldn't find a `create` method in `PostsController`. Let's write `create`:

```
app/controllers/posts_controller.rb
```

```
...
def new
  @post = Post.new
end

+ def create
# #9
+   @post = Post.new
+   @post.title = params[:post][:title]
+   @post.body = params[:post][:body]

# #10
+   if @post.save
# #11
+     flash[:notice] = "Post was saved."
+     redirect_to @post
+   else
# #12
+     flash.now[:alert] = "There was an error saving the post. Please try again."
+     render :new
+   end
+ end

...

```

At #9, we call `Post.new` to create a new instance of `Post`.

At #10, if we successfully save `Post` to the database, we display a success message using `flash[:notice]` and redirect the user to the route generated by `@post`. Redirecting to `@post` will direct the user to the posts **show** view.

At #11, we assign a value to `flash[:notice]`. The `flash` hash provides a way to pass temporary values between actions. Any value placed in `flash` will be available in the next action and then deleted.

At #12, if we do not successfully save `Post` to the database, we display an error message and `render` the `new` view again.

When the user clicks **Save**, the `create` method is called. `create` either updates the database with the `save` method, or returns an error. Unlike `new`, `create` does not have a corresponding view. `create` works behind the scenes to collect the data submitted by the user and update the database. `create` is a POST action.

Run the specs for `create` and observe that all three `POST create` tests pass:

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'POST create'
```

Use the form on <http://localhost:3000/posts/new> to create a post. You will be redirected to the, as of yet, unfinished posts **show** page.

Styling the New View

Now that `new` and `create` are implemented, we can style the corresponding view with Bootstrap:

```
app/views/posts/new.html.erb
```

```

- <%= form_for @post do |f| %>
-   <%= f.label :title %>
-   <%= f.text_field :title %>

-
-   <%= f.label :body %>
-   <%= f.text_area :body %>

-
-   <%= f.submit "Save" %>
- <% end %>

+ <h1>New Post</h1>
+
+ <div class="row">
+   <div class="col-md-4">
+     <p>Guidelines for posts</p>
+     <ul>
+       <li>Make sure it rhymes.</li>
+       <li>Don't use the letter "A".</li>
+       <li>The incessant use of hashtags will get you banned.</li>
+     </ul>
+   </div>
+   <div class="col-md-8">
+     <%= form_for @post do |f| %>
+     <div class="form-group">
# #13
+       <%= f.label :title %>
# #14
+       <%= f.text_field :title, class: 'form-control', placeholder: "Enter post title" %>
+     </div>
+     <div class="form-group">
+       <%= f.label :body %>
+       <%= f.text_area :body, rows: 8, class: 'form-control', placeholder: "Enter post body" %>
+     </div>
+     <div class="form-group">
# #15
+       <%= f.submit "Save", class: 'btn btn-success' %>
+     </div>
+   <% end %>
+ </div>
+ </div>

```

At #13, we use `f.label` to display `title`. This functionality is provided by `form_for`, which yields a form builder object that, in turn, provides `f.label`. `f.label`, in turn, creates an HTML `<label>` tag for the object that is specified. In this case it will be:

```
<label for="post_title">Title</label>
```

At #14, `f.text_field` is another method that `FormHelper` provides and creates an `<input>` tag of type `text`. It will yield:

```
<input class="form-control" placeholder="Enter post title" type="text" name="post[title]">
```

At #15, `f.submit` yields an `input` tag of type `submit`. It will yield:

```
<input type="submit" name="commit" value="Save" class="btn btn-success">
```

Refresh <http://localhost:3000/posts/new> to see your changes. Then use the form to create a new post.

Displaying the Flash Message

When you create a post, no success message is displayed. This is because, while we're placing a value into `flash`, we're not displaying it in the view yet. Because every view may have a `flash` message at some point, we'll add it someplace universal. Let's add it to `application.html.erb`.

Open `app/views/layouts/application.html.erb` and add the flash code:

```
app/views/layouts/application.html.erb
```

```

<!DOCTYPE html>
<html>
  <head>
    <title>Bloccit</title>
    <%= csrf_meta_tags %>

    <meta name="viewport" content="width=device-width, initial-scale=1">
    <%= stylesheet_link_tag 'application', media: 'all', 'data-turbolinks-track': true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <div class="container">
      <ul class="nav nav-tabs">
        <li><%= link_to "Bloccit", root_path %></li>
        <li><%= link_to "About", about_path %></li>
      </ul>

      +   <% if flash[:notice] %>
      +     <div class="alert alert-success">
      +       <button type="button" class="close" data-dismiss="alert">&times;</button>
      +       <%= flash[:notice] %>
      +     </div>
      +   <% elsif flash[:alert] %>
      +     <div class="alert alert-warning">
      +       <button type="button" class="close" data-dismiss="alert">&times;</button>
      +       <%= flash[:alert] %>
      +     </div>
      +   <% end %>

      <%= yield %>
    </div>
  </body>
</html>

```

Note that the class names (`alert`, `alert-*`, and `close`) as well as the `data-dismiss` attribute are **Bootstrap markup** commonly used when creating alerts.

Create a post from <http://localhost:3000/posts/new> and we'll see a green flash message.

Reading Posts

We have the ability to create new posts, so a logical next step is to implement the ability to

read them. Recall the CRUD acronym - "create read update delete"; we've completed "create" and are moving on to "read". First, let's write the tests:

```
spec/controllers/posts_controller_spec.rb
```

```
...
- # describe "GET show" do
- #   it "returns http success" do
- #     get :show
- #     expect(response).to have_http_status(:success)
- #   end
- # end
+ describe "GET show" do
+   it "returns http success" do
# #16
+     get :show, params: { id: my_post.id }
+     expect(response).to have_http_status(:success)
+   end
+   it "renders the #show view" do
# #17
+     get :show, params: { id: my_post.id }
+     expect(response).to render_template :show
+   end
+
+   it "assigns my_post to @post" do
+     get :show, params: { id: my_post.id }
# #18
+     expect(assigns(:post)).to eq(my_post)
+   end
+ end

...
end
```

At #16, we pass `{id: my_post.id}` to `:show` as a parameter. These parameters are passed to the `params` hash.

The `params` hash contains all parameters passed to the application's controller (`application_controller.rb`), whether from GET, POST, or any other HTTP action.

At #17, we `expect` the `response` to return the `show` view using the `render_template` matcher.

At #18, we `expect` the `post` to equal `my_post` because we call `:show` with the id of

`my_post`. We are testing that the post returned to us is the post we asked for.

Run the spec:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'GET show'
```

We see that the first two tests pass, but the last one fails. Let's fix it by implementing `show`:

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController
  def index
    @posts = Post.all
  end

  def show
    # #19
    +   @post = Post.find(params[:id])
  end

  def new
  end

  def edit
  end

end
```

At **#19**, we find the post that corresponds to the id in the `params` that was passed to `show` and assign it to `@post`. Unlike in the `index` method, in the `show` method, we populate an instance variable with a single post, rather than a collection of posts.

Run the spec again and see that our tests for `show` pass.

Open <http://localhost:3000/posts> and click on a post's link. We are taken to the `show` view, thanks to the `link_to` method. The `show` view still has boilerplate HTML code.

Let's view the `params` hash by adding this line:

app/views/posts/show.html.erb

```
<h1>Posts#show</h1>
<p>Find me in app/views/posts/show.html.erb</p>

+ <%= params %>
```

When you refresh the page, you should see this:

```
{"action"=>"show", "controller"=>"posts", "id"=>"1"}
```

This hash communicates which action and controller was called. It also has the id of the post we clicked - it's encoded in the URL. In our controller, we accessed that id by calling `params[:id]`, which, in the above case, returned "1". We then assigned the post found with that id to the `@post` variable.

Let's replace `<%= params %>` in `app/views/posts/show.html.erb` with some post-specific code:

app/views/posts/show.html.erb

```
- <h1>Posts#show</h1>
- <p>Find me in app/views/posts/show.html.erb</p>
-
- <%= params %>
+ <h1><%= @post.title %></h1>
+ <p><%= @post.body %></p>
```

Go back to the `index` view at <http://localhost:3000/posts> and click on a post. You should see an updated `show` view with data specific to the `Post` instance that was clicked.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku. Open your production app to view the updates. Create a new post, and ensure that you can access it on the `index` and `show` views.

Recap

Concept	Description
Form Helpers	Form helpers are view helper methods that generate HTML markup for input forms.
<code>form_for</code>	The <code>form_for</code> method is a form helper which binds a form to a model object.

params

The Rails `params` hash is available in controllers and contains both **query string parameters** and **POST data**.

ActionDispatch::Flash

`ActionDispatch::Flash` provides a data structure to pass temporary primitive types (`String`, `Array`, `Hash`) between controller actions. Anything placed in the `flash` param will be exposed to the next action and then removed. It is commonly used for notices and alerts.

How would you rate this checkpoint and assignment?



18. Rails: CRUD

Assignment

Discussion

Submission

18. Rails: CRUD

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

It's never too early to monetize! For this assignment, you'll add advertisements to Bloccit.

1. Use TDD for this assignment.
2. Create a new model called `Advertisement`. It should have the following attributes: `title:string`, `copy:text`, `price:integer`.
3. Generate a controller for `Advertisement` with `index`, `show`, `new`, and `create` actions. Should the controller class and file names have a singular (`advertisement`) or plural (`advertisements`) prefix? Be consistent with the naming pattern used for `PostsController`.
4. Update `routes.rb` to use **resourceful routing** for `Advertisement`.
5. Complete the `index`, `show`, `new`, and `create` actions in `AdvertisementsController`.
6. Update the advertisement `index`, `show` and `new` views.
7. Seed your app with instances of `Advertisement`.
8. Test your changes in the browser. Confirm that you can see an index of all advertisements, view individual advertisements, and create new advertisements.

Assig

[←Prev](#)

Submission

[Next→](#)

19 Rails: More CRUD



“The urge to destroy is also a creative urge.”

— Pablo Picasso

Overview and Purpose

In this checkpoint you'll learn more about the UD in CRUD.

Objectives

After this checkpoint, you should be able to:

- Explain the U in CRUD.

- Explain the D in CRUD.

Updating and Deleting posts

Now that we can create and view posts, we'll implement the ability to edit and update them.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Editing and Updating Posts

To edit existing posts we'll need to implement two actions, `edit` and `update`. The `edit` action uses the `edit` view to allow users to update an existing post. Just as the `new` view submits a form to the `create` action, the `edit` view submits a form to the `update` action.

edit Action

Let's create the tests for `edit`:

```
spec/controllers/posts_controller_spec.rb

...
- # describe "GET edit" do
- #   it "returns http success" do
- #     get :edit
- #     expect(response).to have_http_status(:success)
- #   end
- # end

+ describe "GET edit" do
+   it "returns http success" do
+     get :edit, params: { id: my_post.id }
+     expect(response).to have_http_status(:success)
+   end
+
+   it "renders the #edit view" do
+     get :edit, params: { id: my_post.id }
# #1
+     expect(response).to render_template :edit
+   end
+
# #2
+   it "assigns post to be updated to @post" do
+     get :edit, params: { id: my_post.id }
+
+     post_instance = assigns(:post)
+
+     expect(post_instance.id).to eq my_post.id
+     expect(post_instance.title).to eq my_post.title
+     expect(post_instance.body).to eq my_post.body
+   end
+
+ end
...
```

At #1, we `expect` the `edit` view to render when a post is edited.

At #2, we test that `edit` assigns the correct post to be updated to `@post`.

Run the `edit` specs:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'GET edit'
```

The first two tests pass because of the stubbed `edit` action in `PostsController` and Rails' default rendering. The last test fails because we haven't finished the `edit` action.

Modify the `edit` action:

app/controllers/posts_controller.rb

```
def edit
+   @post = Post.find(params[:id])
end
```

Rerun the `edit` specs to confirm all three pass:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'GET edit'
```

update Action

With the `edit` action implemented and the specs passing, it's time to implement `update` to receive the edited posts. Let's TDD `update`:

spec/controllers/posts_controller_spec.rb

```

...
+   describe "PUT update" do
+     it "updates post with expected attributes" do
+       new_title = RandomData.random_sentence
+       new_body = RandomData.random_paragraph
+
+       put :update, params: { id: my_post.id, post: {title: new_title, body: new_body}
+
+       # #3
+
+       updated_post = assigns(:post)
+       expect(updated_post.id).to eq my_post.id
+       expect(updated_post.title).to eq new_title
+       expect(updated_post.body).to eq new_body
+
+     end
+
+     it "redirects to the updated post" do
+       new_title = RandomData.random_sentence
+       new_body = RandomData.random_paragraph
+
+       # #4
+
+       put :update, params: { id: my_post.id, post: {title: new_title, body: new_body}
+       expect(response).to redirect_to my_post
+
+     end
+
+   end
...

```

At #3, we test that `@post` was updated with the title and body passed to `update`. We also test that `@post`'s id was not changed.

At #4, we `expect` to be redirected to the posts `show` view after the `update`.

Run the `update` specs.

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'PUT update'
```

These tests both fail because we haven't implemented the `update` action in `PostsController`. Let's implement it:

`app/controllers/posts_controller.rb`

```
+ def update
+   @post = Post.find(params[:id])
+   @post.title = params[:post][:title]
+   @post.body = params[:post][:body]
+
+   if @post.save
+     flash[:notice] = "Post was updated."
+     redirect_to @post
+   else
+     flash.now[:alert] = "There was an error saving the post. Please try again."
+     render :edit
+   end
+ end
```

Run the `update` tests and verify that they both pass:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'PUT update'
```

Edit and Update Views

Update the posts `edit` view to display a form that allows users to update posts:

```
app/views/posts/edit.html.erb
```

```
- <h1>Posts#edit</h1>
- <p>Find me in app/views/posts/edit.html.erb</p>
+ <h1>Edit Post</h1>
+
+ <div class="row">
+   <div class="col-md-4">
+     <p>Guidelines for posts</p>
+     <ul>
+       <li>Make sure it rhymes.</li>
+       <li>Don't use the letter "A".</li>
+       <li>The incessant use of hashtags will get you banned.</li>
+     </ul>
+   </div>
+   <div class="col-md-8">
+     <%= form_for @post do |f| %>
+       <div class="form-group">
+         <%= f.label :title %>
+         <%= f.text_field :title, class: 'form-control', placeholder: "Enter post title" %>
+       </div>
+       <div class="form-group">
+         <%= f.label :body %>
+         <%= f.text_area :body, rows: 8, class: 'form-control', placeholder: "Enter post body" %>
+       </div>
+       <div class="form-group">
+         <%= f.submit "Save", class: 'btn btn-success' %>
+       </div>
+     <% end %>
+   </div>
+ </div>
```

Let's add a link to edit a post on the **show** view:

app/views/posts/show.html.erb

```
- <h1><%= @post.title %></h1>
- <p><%= @post.body %></p>
+ <h1><%= @post.title %></h1>
+
+ <div class="row">
+   <div class="col-md-8">
+     <p><%= @post.body %></p>
+   </div>
+   <div class="col-md-4">
+     <!-- #5 -->
+     <%= link_to "Edit", edit_post_path(@post), class: 'btn btn-success' %>
+   </div>
+ </div>
```

At #5, we format a link as an **Edit** button which directs a user to `/posts/@post.id/edit`. `edit_post_path(@post)` is a helper method that is generated in `routes.rb` by `resources :posts`. (Run `rails routes` from the command line to view the post routes, if you need a refresher.)

Open <http://localhost:3000/posts>, click on a post, edit, and save it.

Destroy

We should provide users with the ability to delete posts. Let's write the tests for the `Post#destroy` action:

```
spec/controllers/posts_controller_spec.rb
```

```
...
+
+ describe "DELETE destroy" do
+   it "deletes the post" do
+     delete :destroy, params: { id: my_post.id }
+     # #6
+     count = Post.where({id: my_post.id}).size
+     expect(count).to eq 0
+   end
+
+   it "redirects to posts index" do
+     delete :destroy, params: { id: my_post.id }
+     # #7
+     expect(response).to redirect_to posts_path
+   end
+ end
...
```

At #6, we search the database for a post with an id equal to `my_post.id`. This returns an `Array`. We assign the `size` of the array to `count`, and we `expect` `count` to equal zero. This test asserts that the database won't have a matching post after `destroy` is called.

At #7, we `expect` to be redirected to the posts `index` view after a post has been deleted.

Run these tests and confirm they both fail because we haven't defined the `destroy` action yet:

```
Terminal
```

```
$ rspec spec/controllers/posts_controller_spec.rb -e 'DELETE destroy'
```

Open `PostsController` and implement the `destroy` action to make the previous tests pass:

app/controllers/posts_controller.rb

```
...
+ def destroy
+   @post = Post.find(params[:id])
+
# #8
+   if @post.destroy
+     flash[:notice] = "\"#{@post.title}\" was deleted successfully."
+     redirect_to posts_path
+   else
+     flash.now[:alert] = "There was an error deleting the post."
+     render :show
+   end
+ end
```

...

At #8, we call `destroy` on `@post`. If that call is successful, we set a `flash` message and redirect the user to the posts `index` view. If `destroy` fails then we redirect the user to the `show` view using `render :show`.

Comments are dependent on a post's existence because of the `has_many :comments` declaration in `Post`. When we delete a post, we also need to delete all related comments. We'll perform a "cascade delete", which ensures that when a post is deleted, all of its comments are too. Let's modify `Post` to handle this:

app/models/post.rb

```
- has_many :comments
+ has_many :comments, dependent: :destroy
```

Let's use `link_to` to add a link to delete posts on the `show` view:

app/views/posts/show.html.erb

```
<h1><%= @post.title %></h1>

<div class="row">
  <div class="col-md-8">
    <p><%= @post.body %></p>
  </div>
  <div class="col-md-4">
    <%= link_to "Edit", edit_post_path(@post), class: 'btn btn-success' %>
    <!-- #9 -->
    +  <%= link_to "Delete Post", @post, method: :delete, class: 'btn btn-danger', data: { confirm: "Are you sure?" } %>
  </div>
</div>
```

At #9, we use `link_to` to create a delete button. The text on the button is **Delete Post**. We override the default `method` (`:post`) with `:delete` so that when the button is pressed the route called is the delete route. We style the button by setting `class:` to 'btn btn-danger'. We pass a `Hash` with the `confirm:` key to the `data:` argument. This confirms the action with a JavaScript confirmation dialog when a user presses the button. Let's watch a video exploring `link_to`:

Go to <http://localhost:3000/posts>, pick a post, and delete it.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Recap

Concept	Description

HTTP DELETE request

The HTTP DELETE request is used to remove the identified resource from the web server.

How would you rate this checkpoint and assignment?



19. Rails: More CRUD

Assignment

Discussion

Submission

19. Rails: More CRUD

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

In addition to posts, we'll allow Bloccit users to ask questions. A question will be similar to a post, but will also have a "resolved" attribute to signal that other answers are no longer needed.

1. Use TDD for this assignment.
2. Create a new model called `Question`. It should have `title:string`, `body:text`, and `resolved:boolean` attributes.
3. Generate a controller for `Question` with the [default CRUD controller actions](#).
4. Update `routes.rb` to use [resourceful routing](#) for `Question`.
5. Complete `QuestionsController` and its corresponding views. Accept input for the `resolved` attribute in the Question form using a [checkbox](#).
6. Seed your application with questions.

[←Prev](#)

Submission

[Next→](#)

20 Programming Reinforcement: Checkpoint 3

We've updated this checkpoint! For reference, the deprecated version is [here](#)

Overview and Purpose

This checkpoint introduces three coding challenges to practice algorithms, advanced control flow, regular expressions, loops, and basic data structures.

Objectives

After this checkpoint, you should be able to:

- Explain the difference between an array and an enumerator in Ruby.
- Apply efficient enumeration in the control flow of an algorithm.
- Understand regular expressions in Ruby.

Programming Reinforcement

Each foundation checkpoint introduces new concepts, patterns, and assignments. We designed Bloc's curriculum to push you to your learning limits. With that in mind, we've included breaks between lessons to reinforce programming concepts; this is the third of those breaks, known as Programming Reinforcement checkpoints.

In these checkpoints, you will complete coding challenges that help you master Ruby and learn to think like a programmer. After you complete the challenges, you will meet with your mentor to discuss your solutions.

- **Challenges**
- **Bonus Challenge: Sudoku Solution Validator**
- **Assignment**
 - **For Mentors**

Challenges

You must solve these three Kata (challenges) before submitting the checkpoint. At this point in your program, your skill level meets or exceeds that required by each Kata. While they may challenge you, know that you are capable of completing each one.

Your mentor may help you, but we strongly encourage you attempt these on your own.

Kata	Hint
Format Names	Write a function that returns a string formatted as a list of names separated by commas except for the last two names, which should be separated by an ampersand.
Stop gninnipS My sdrow!	Write a function that takes in a string of one or more words, and returns the same string, but with all five or more letter words reversed .
Evil Autocorrect Prank	Write a function that takes a string and replaces all instances of "you" or "u" (not case sensitive) with "your sister" (always lower case).

Note For Evil Autocorrect Prank The person that created that kata didn't create any tests. If you would like some tests you can run before you submit your work, copy/paste the following snippet into the 'Test Driven Development (TDD)' section of the Code Wars interface (*the bottom right box*).

```
describe ".autocomplete" do
  it "change 'u' or 'you' to 'your sister'" do
    Test.assert_equals(autocomplete("u"), "your sister")
    Test.assert_equals(autocomplete("you"), "your sister")
    Test.assert_equals(autocomplete("hello there"), "hello there")
    Test.assert_equals(autocomplete("random string"), "random string")
    Test.assert_equals(autocomplete("I miss you!"), "I miss your sister!")
  end
end
```

* Difficult problems have lower ratings.

Bonus Challenge: Title Case

This bonus challenge is optional, but we strongly encourage you to attempt it.

How would you rate this checkpoint and assignment?



20. Programming Reinforcement: Checkpoint 3

Assignment

Discussion

Submission

[←Prev](#)

Submission

[Next→](#)

21 Rails: Topics and Posts



“Like religion, politics, and family planning, cereal is not a topic to be brought up in public. It's too controversial.”

— Erma Bombeck

Overview and Purpose

In this checkpoint you'll learn more about nested routing in Rails and code refactoring.

Objectives

After this checkpoint, you should be able to:

- Explain what nested routes are.
- Explain code refactoring and how it's useful.

Topics

We've built the functionality to create posts, but we don't have a way to organize them. We anticipate that Bloccit users will create a large number of posts, and will therefore need a way to organize, or categorize them. In this checkpoint, we'll create a **topics** resource that will be used to organize posts. Along the way, we'll learn how to nest resources and refactor code.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Topic Model

The first step in creating a new topic resource is to create a topic model. An instance of `Topic` will require a `name` and `description` attribute. We'll also create a `public` attribute, of a `Boolean` data type, to allow users to restrict access to topics.

Create the topic model:

Terminal

```
$ rails generate model topic name:string public:boolean description:text
  invoke  active_record
  create    db/migrate/20150625221905_create_topics.rb
  create    app/models/topic.rb
  invoke    rspec
  create      spec/models/topic_spec.rb
```

We can call the `generate` command with either capitalized (e.g. `Topic`) or uncapitalized (e.g. `topic`) model names. It makes no difference to the generator.

Open the migration file and set the `public` attribute to `true` by default:

db/migrate/20150729181446_create_topics.rb

```
class CreateTopics < ActiveRecord::Migration[5.0]
  def change
    create_table :topics do |t|
      t.string :name
      - t.boolean :public
      + t.boolean :public, default: true
      t.text :description

      t.timestamps
    end
  end
end
```

Run the migration using `rails db:migrate`:

Terminal

```
$ rails db:migrate
```

Topic Specs

Add the following tests to `topic_spec.rb`:

```
spec/models/topic_spec.rb
```

```
require 'rails_helper'

RSpec.describe Topic, type: :model do
  let(:name) { RandomData.random_sentence }
  let(:description) { RandomData.random_paragraph }
  let(:public) { true }
  let(:topic) { Topic.create!(name: name, description: description) }

  # #1
  describe "attributes" do
    it "has name, description, and public attributes" do
      expect(topic).to have_attributes(name: name, description: description, public: public)
    end
  end

  # #2
  it "is public by default" do
    expect(topic.public).to be(true)
  end
end
```

At **#1**, we confirm that a topic responds to the appropriate attributes.

At **#2**, we confirm that the `public` attribute is set to `true` by default.

Run `topic_spec.rb` to confirm all four tests pass:

```
Terminal
```

```
$ rspec spec/models/topic_spec.rb
```

To organize posts by topic we will need to build an association between topics and posts. To TDD this association we'll use the **Shoulda gem**. Shoulda makes it easier for us to write association tests by providing some handy methods that RSpec doesn't have. Add Shoulda to `Gemfile` and run `bundle install`:

Gemfile

```
group :development, :test do
  gem 'rspec-rails', '~> 3.0'
  gem 'rails-controller-testing'
+ gem 'shoulda'
end
```

Terminal

```
$ bundle install
```

Use methods provided by Shoulda and add the following tests to `topic_spec.rb`:

spec/models/topic_spec.rb

```
require 'rails_helper'

RSpec.describe Topic, type: :model do
  let(:name) { RandomData.random_sentence }
  let(:description) { RandomData.random_paragraph }
  let(:public) { true }
  let(:topic) { Topic.create!(name: name, description: description) }

+  it { is_expected.to have_many(:posts) }
...
```

Now add tests for `post_spec.rb`:

spec/models/post_spec.rb

```

require 'rails_helper'

RSpec.describe Post, type: :model do
  - let(:post) { Post.create!(title: "New Post Title", body: "New Post Body") }
  + let(:name) { RandomData.random_sentence }
  + let(:description) { RandomData.random_paragraph }
  + let(:title) { RandomData.random_sentence }
  + let(:body) { RandomData.random_paragraph }

  # #3
  + let(:topic) { Topic.create!(name: name, description: description) }
  # #4
  + let(:post) { topic.posts.create!(title: title, body: body) }

  +
  + it { is_expected.to belong_to(:topic) }

  describe "attributes" do
    it "has a title and body attribute" do
      - expect(post).to have_attributes(title: "New Post Title", body: "New Post Body")
      + expect(post).to have_attributes(title: title, body: body)
    end
  end
end

```

At #3, we create a parent topic for `post`.

At #4, we associate `post` with `topic` via `topic.posts.create!`. This is a chained method call which creates a post for a given topic.

Finally, we'll need to update `comment_spec.rb` to account for the new association:

spec/models/comment_spec.rb

```

require 'rails_helper'

RSpec.describe Comment, type: :model do
  - let(:post) { Post.create!(title: "New Post Title", body: "New Post Body") }
  + let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_paragraph) }
  + let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_paragraph) }

  let(:comment) { Comment.create!(body: 'Comment Body', post: post) }

  describe "attributes" do
    ...
  end
end

```

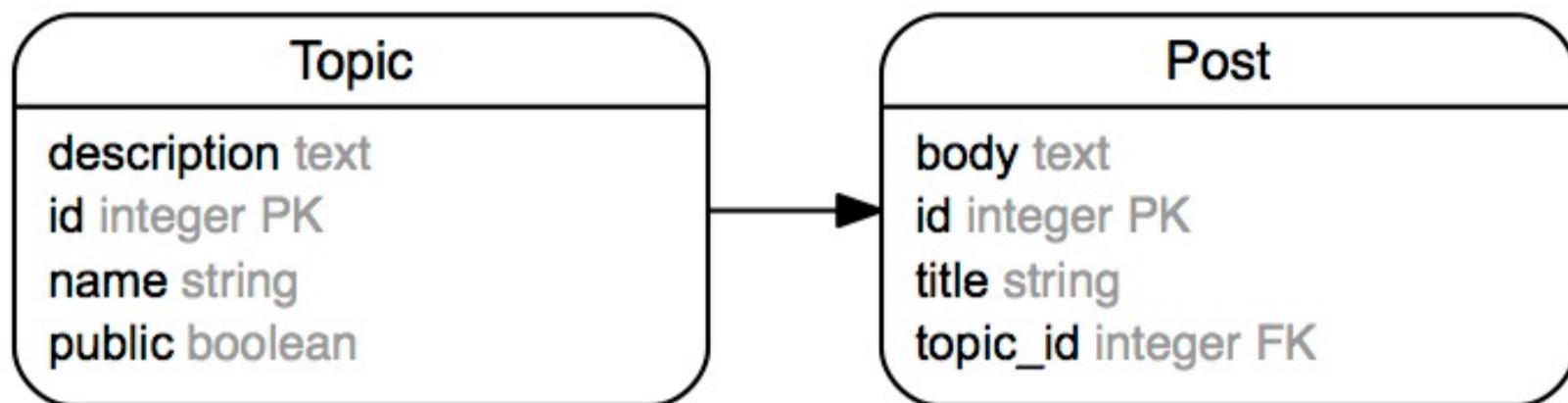
Run the spec and confirm that the test doesn't pass.

Run `topic_spec.rb` and `post_spec.rb`. There will be error messages for each spec

because topics and posts are not associated. Let's fix these errors by creating the association.

Associations

Database tables are associated via **foreign keys**. Recall that a foreign key is an attribute that references an attribute, most often the primary key, of another table. For example:



In the above association, `Post.topic_id` is a **foreign_key** that references `Topic.id`.

Because posts belong to topics, we'll add a `topic_id` foreign key attribute to the posts table:

Terminal

```
$ rails generate migration AddTopicToPosts topic_id:integer:index
  invoke  active_record
  create    db/migrate/20150729184746_add_topic_to_posts.rb
```

This generated the following migration file:

```
db/migrate/20150729184746_add_topic_to_posts.rb
```

```
class AddTopicToPosts < ActiveRecord::Migration
  def change
    # #5
    add_column :posts, :topic_id, :integer
    # #6
    add_index :posts, :topic_id
  end
end
```

At #5, we see that the name we gave the migration, `AddTopicToPosts`, is very important. Using this specific naming format:

"Add" + [table whose id we want to add] + "To" + [table we want to add the foreign key]

...we instructed the generator to create a migration that adds a topic_id column to the posts table.

At #6, we created an index on topic_id with the generator. An index improves the speed of operations on a database table.

You should always index your **foreign key columns**.

Run the migration to add the foreign key:

Terminal

```
$ rails db:migrate
```

Update `Post` and `Topic` to reflect the association:

app/models/post.rb

```
class Post < ApplicationRecord
+  belongs_to :topic
  has_many :comments, dependent: :destroy
end
```

app/models/topic.rb

```
class Topic < ApplicationRecord
+  has_many :posts
end
```

Run `topic_spec.rb` and `post_spec.rb` to confirm that all the tests are passing:

Terminal

```
$ rspec spec/models/topic_spec.rb
$ rspec spec/models/post_spec.rb
```

Seeds

We also need to update `seeds.rb`, because none of the posts in the database have an associated topic. Create some new topics and assign each post to a random topic:

```
+ # Create Topics
+ 15.times do
+   Topic.create!(
+     name: RandomData.random_sentence,
+     description: RandomData.random_paragraph
+   )
+ end
+ topics = Topic.all

# Create Posts
50.times do
  Post.create!(
+   topic: topics.sample,
    title: RandomData.random_sentence,
    body: RandomData.random_paragraph
  )
end

posts = Post.all

# Create Comments
100.times do
  Comment.create!(
    post: posts.sample,
    body: RandomData.random_paragraph
  )
end

puts "Seed finished"
+ puts "#{Topic.count} topics created"
+ puts "#{Post.count} posts created"
+ puts "#{Comment.count} comments created"
```

Empty the existing database of posts and topics and reseed it with the `reset` command:

Terminal

```
$ rails db:reset
```

The Topics Resource

Our topics resource has a model, but not a controller. Let's generate a topics controller to complete our topics resource, as well as the necessary routes and views to present topics

to users:

Terminal

```
$ rails generate controller Topics
```

We didn't pass any action arguments to the controller generator. Instead, we'll create the actions and views that topics require manually.

Build the resourceful routes:

config/routes.rb

```
Rails.application.routes.draw do
  + resources :topics
  resources :posts

  get 'about' => 'welcome#about'

  root 'welcome#index'
end
```

Run `rails routes` from the command line to examine the routes we created for topics. `resources` creates the seven standard CRUD routes we need for topics.

Topics Index

As with `PostsController`, we'll TDD the actions in `TopicsController`, starting with the `index` action:

spec/controllers/topics_controller_spec.rb

```
require 'rails_helper'

RSpec.describe TopicsController, type: :controller do
  let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)

  describe "GET index" do
    it "returns http success" do
      get :index
      expect(response).to have_http_status(:success)
    end

    it "assigns my_topic to @topics" do
      get :index
      expect(assigns(:topics)).to eq([my_topic])
    end
  end
end
```

This test follows the pattern we established when testing `PostsController`. Using `let`, we create a `my_topic` variable to use in our tests. We then write two tests to confirm the expected behavior of the `index` action.

Run the `index` section of the `TopicsControllerSpec` to confirm that both tests fail:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET index'
```

Add the `index` action to `TopicsController` and provide an array of topics to the `index` view to pass the tests:

app/controllers/topics_controller.rb

```
class TopicsController < ApplicationController
  def index
    @topics = Topic.all
  end
end
```

To pass the tests, we need to create the topics `index` view to display a list of all topics with their names and descriptions:

Terminal

```
$ touch app/views/topics/index.html.erb
```

With this view created, `topics_controller_spec.rb` will now pass both `index` tests:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET index'
```

Fill out the view to display a list of all topics:

app/views/topics/index.html.erb

```
+ <h1>Topics</h1>
+
+ <div class="row">
+   <div class="col-md-8">
+     <!-- #7 -->
+       <% @topics.each do |topic| %>
+         <div class="media">
+           <div class="media-body">
+             <h4 class="media-heading">
+               <!-- #8 -->
+                 <%= link_to topic.name, topic %>
+               </h4>
+               <small>
+                 <%= topic.description %>
+               </small>
+             </div>
+           </div>
+         <% end %>
+       </div>
+       <div class="col-md-4">
+         <!-- #9 -->
+           <%= link_to "New Topic", new_topic_path, class: 'btn btn-success' %>
+         </div>
+       </div>
```

At #7, we loop over each topic in `@topics`.

At #8, we create a link to the **show** view for each `topic`.

At #9, we create a link to create a new topic.

Visit <http://localhost:3000/topics>. You should see the topics that were created in `seeds.rb` and a **New Topic** button.

Show Topic

When a user clicks on a topic, they should be taken to its **show** view and shown the posts that belong to that topic.

Add tests for the topic **show** action:

```
spec/controllers/topics_controller_spec.rb

...
+ describe "GET show" do
+   it "returns http success" do
+     get :show, params: { id: my_topic.id }
+     expect(response).to have_http_status(:success)
+   end
+
+   it "renders the #show view" do
+     get :show, params: { id: my_topic.id }
+     expect(response).to render_template :show
+   end
+
+   it "assigns my_topic to @topic" do
+     get :show, params: { id: my_topic.id }
+     expect(assigns(:topic)).to eq(my_topic)
+   end
+ end
end
```

These three tests should fail:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET show'
```

Update `TopicsController` to add a `show` action

```
app/controllers/topics_controller.rb
```

```
class TopicsController < ApplicationController
  def index
    @topics = Topic.all
  end
+
+  def show
+    @topic = Topic.find(params[:id])
+  end
end
```

Our tests will still fail because of the missing **show** view, so we'll create that view next:

Terminal

```
$ touch app/views/topics/show.html.erb
```

With the **show** view created, `topics_controller_spec.rb` will now pass all three `show` tests:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET show'
```

Fill out the view to display the topic information and all associated posts:

app/views/topics/show.html.erb

```
+ <h1><%= @topic.name %></h1>
+
+ <%= link_to "Edit Topic", edit_topic_path, class: 'btn btn-success' %>
+
+ <div class="row">
+   <div class="col-md-8">
+     <p class="lead"><%= @topic.description %></p>
+     <!-- #10 -->
+     <% @topic.posts.each do |post| %>
+       <div class="media">
+         <div class="media-body">
+           <h4 class="media-heading">
+             <%= link_to post.title, post %>
+           </h4>
+         </div>
+       </div>
+     <% end %>
+   </div>
+   <div class="col-md-4">
+     <%= link_to "New Post", new_post_path(@topic), class: 'btn btn-success' %>
+   </div>
+ </div>
```

At **#10**, we iterate over the `posts` belonging to `@topic`, and display each post.

Open <http://localhost:3000/topics> and click on a topic to confirm that the topic **show** view works as expected.

New Topics

Users will want to be able to create new topics, so let's implement the `new` and `create` actions using TDD.

`new` Action

Add tests for the `new` action:

```
spec/controllers/topics_controller_spec.rb

...
+ describe "GET new" do
+   it "returns http success" do
+     get :new
+     expect(response).to have_http_status(:success)
+   end
+
+   it "renders the #new view" do
+     get :new
+     expect(response).to render_template :new
+   end
+
+   it "initializes @topic" do
+     get :new
+     expect(assigns(:topic)).not_to be_nil
+   end
+ end
end
```

The three `new` tests will fail because `TopicsController` doesn't implement the `new` action yet:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET new'
```

Implement the `new` action:

```
app/controllers/topics_controller.rb
```

```
class TopicsController < ApplicationController
  def index
    @topics = Topic.all
  end

  def show
    @topic = Topic.find(params[:id])
  end

+  def new
+    @topic = Topic.new
+  end
end
```

Our tests are still failing because of the missing **new** view, so let's create it now:

Terminal

```
$ touch app/views/topics/new.html.erb
```

With the **new** view created, `topics_controller_spec.rb` will now pass all three `new` tests:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET new'
```

Add a form for creating new topics:

```
app/views/topics/new.html.erb
```

```

+ <h1>New Topic</h1>
+
+ <div class="row">
+   <div class="col-md-4">
+     <p>Guidelines for topics:</p>
+     <ul>
+       <li>Make sure the topic is appropriate.</li>
+       <li>Never insult dogs.</li>
+       <li>Smile when you type.</li>
+     </ul>
+   </div>
+   <div class="col-md-8">
+     <%= form_for @topic do |f| %>
+       <div class="form-group">
+         <%= f.label :name %>
+         <%= f.text_field :name, class: 'form-control', placeholder: "Enter topic name" %>
+       </div>
+       <div class="form-group">
+         <%= f.label :description %>
+         <%= f.text_area :description, rows: 8, class: 'form-control', placeholder: "Enter topic description" %>
+       </div>
+       <div class="form-group">
+         <%= f.label :public, class: 'checkbox' do %>
+           <%= f.check_box :public %> Public topic
+         <% end %>
+       </div>
+       <%= f.submit "Save", class: 'btn btn-success' %>
+     <% end %>
+   </div>
+ </div>

```

Open <http://localhost:3000/topics/new> and click on the **New Topic** button to confirm that the topic **new** view *looks* as expected.

The user interface (UI) for creating new topics is complete, but if we try to submit the form, we'll get an error. This is because we haven't implemented the `create` action, which is used for inserting new records into the database.

create Action

Add three tests for `create`:

`spec/controllers/topics_controller_spec.rb`

```
...
+ describe "POST create" do
+   it "increases the number of topics by 1" do
+     expect{ post :create, {topic: {name: RandomData.random_sentence, description: RandomData.random_sentence}} }.to change(Topic, :count).by(1)
+   end
+
+   it "assigns Topic.last to @topic" do
+     post :create, {topic: {name: RandomData.random_sentence, description: RandomData.random_sentence}}
+     expect(assigns(:topic)).to eq Topic.last
+   end
+
+   it "redirects to the new topic" do
+     post :create, {topic: {name: RandomData.random_sentence, description: RandomData.random_sentence}}
+     expect(response).to redirect_to Topic.last
+   end
+ end
end
```

Run these tests and confirm that they're failing:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'POST create'
```

Pass these tests by implementing the `create` action:

```
app/controllers/topics_controller.rb
```

```

class TopicsController < ApplicationController
  def index
    @topics = Topic.all
  end

  def show
    @topic = Topic.find(params[:id])
  end

  def new
    @topic = Topic.new
  end

+
+  def create
+    @topic = Topic.new
+    @topic.name = params[:topic][:name]
+    @topic.description = params[:topic][:description]
+    @topic.public = params[:topic][:public]
+
+    if @topic.save
+      redirect_to @topic, notice: "Topic was saved successfully."
+    else
+      flash.now[:alert] = "Error creating topic. Please try again."
+      render :new
+    end
+  end
end

```

The `create` tests in `topics_controller_spec.rb` should now pass:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'POST create'
```

Create a couple of new topics in your browser to confirm that `new` and `create` work as expected.

Edit Topic

Implement the `edit` and `update` actions so users can edit existing topics, starting with `edit`.

edit Action

Let's TDD the `edit` action first:

spec/controllers/topics_controller_spec.rb

```
...
+ describe "GET edit" do
+   it "returns http success" do
+     get :edit, {id: my_topic.id}
+     expect(response).to have_http_status(:success)
+   end
+
+   it "renders the #edit view" do
+     get :edit, {id: my_topic.id}
+     expect(response).to render_template :edit
+   end
+
+   it "assigns topic to be updated to @topic" do
+     get :edit, {id: my_topic.id}
+     topic_instance = assigns(:topic)
+
+     expect(topic_instance.id).to eq my_topic.id
+     expect(topic_instance.name).to eq my_topic.name
+     expect(topic_instance.description).to eq my_topic.description
+   end
+ end
end
```

These tests will fail until we implement the `edit` action:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET edit'
```

Implement the `edit` action:

app/controllers/topics_controller.rb

```
class TopicsController < ApplicationController
...
+
+ def edit
+   @topic = Topic.find(params[:id])
+ end
+
end
```

Our tests are still failing because of the missing `edit` view:

Terminal

```
$ touch app/views/topics/edit.html.erb
```

topics_controller_spec.rb will now pass all three edit tests:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'GET edit'
```

Update the edit view to give users the ability to update a topic's name, description, and public attributes:

app/views/topics/edit.html.erb

```
+ <h1>Edit Topic</h1>
+
+ <div class="row">
+   <div class="col-md-4">
+     <p>Guidelines for topics:</p>
+     <ul>
+       <li>Make sure the topic is appropriate.</li>
+       <li>Never insult dogs.</li>
+       <li>Smile when you type.</li>
+     </ul>
+   </div>
+   <div class="col-md-8">
+     <%= form_for @topic do |f| %>
+       <div class="form-group">
+         <%= f.label :name %>
+         <%= f.text_field :name, class: 'form-control', placeholder: "Enter topic name" %>
+       </div>
+       <div class="form-group">
+         <%= f.label :description %>
+         <%= f.text_area :description, rows: 8, class: 'form-control', placeholder: "Enter topic description" %>
+       </div>
+       <div class="form-group">
+         <!-- #11 -->
+         <%= f.label :public, class: 'checkbox' do %>
+           <%= f.check_box :public %> Public topic
+         <% end %>
+       </div>
+       <%= f.submit "Save", class: 'btn btn-success' %>
+     <% end %>
+   </div>
+ </div>
```

At #11, we pass a block to `f.label` to generate the HTML for a checkbox.

From <http://localhost:3000/topics> click on a topic and the click the **Edit Topic** button to confirm that the topic `edit` view and form *look* as expected.

update Action

Let's create the ability to update a topic's database record by implementing the `update` action. Add three tests for `update`:

```
spec/controllers/topics_controller_spec.rb

...
+ describe "PUT update" do
+   it "updates topic with expected attributes" do
+     new_name = RandomData.random_sentence
+     new_description = RandomData.random_paragraph
+
+     put :update, params: { id: my_topic.id, topic: { name: new_name, description: new_description } }
+
+     updated_topic = assigns(:topic)
+     expect(updated_topic.id).to eq my_topic.id
+     expect(updated_topic.name).to eq new_name
+     expect(updated_topic.description).to eq new_description
+   end
+
+   it "redirects to the updated topic" do
+     new_name = RandomData.random_sentence
+     new_description = RandomData.random_paragraph
+
+     put :update, params: { id: my_topic.id, topic: { name: new_name, description: new_description } }
+     expect(response).to redirect_to my_topic
+   end
+ end
end
```

Run these tests and confirm that they fail:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'PUT update'
```

Pass the tests by implementing the `update` action:

```
app/controllers/topics_controller.rb
```

```
class TopicsController < ApplicationController
  ...
+  def update
+    @topic = Topic.find(params[:id])
+
+    @topic.name = params[:topic][:name]
+    @topic.description = params[:topic][:description]
+    @topic.public = params[:topic][:public]
+
+    if @topic.save
+      flash[:notice] = "Topic was updated."
+      redirect_to @topic
+    else
+      flash.now[:alert] = "Error saving topic. Please try again."
+      render :edit
+    end
+  end
+
end
```

The `update` tests in `topics_controller_spec.rb` should now pass:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'PUT update'
```

Edit some topics in your browser to confirm that the `edit` and `update` work as expected.

Delete Topic

Users may want to delete topics, so we'll implement `destroy` using TDD:

```
spec/controllers/topics_controller_spec.rb
```

```
...
+   describe "DELETE destroy" do
+     it "deletes the topic" do
+       delete :destroy, {id: my_topic.id}
+       count = Post.where({id: my_topic.id}).size
+       expect(count).to eq 0
+     end
+
+     it "redirects to topics index" do
+       delete :destroy, {id: my_topic.id}
+       expect(response).to redirect_to topics_path
+     end
+   end
end
```

Run these tests:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'DELETE destroy'
```

They'll both fail until we implement `destroy`:

app/controllers/topics_controller.rb

```
...
+   def destroy
+     @topic = Topic.find(params[:id])
+
+     if @topic.destroy
+       flash[:notice] = "\"#{@topic.name}\" was deleted successfully."
+       redirect_to action: :index
+     else
+       flash.now[:alert] = "There was an error deleting the topic."
+       render :show
+     end
+   end
end
```

`redirect_to action: :index` is the same as `redirect_to topics_path` because `topics_path` routes to the `index` action per Rails' resourceful routing.

Our tests should now pass:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb -e 'DELETE destroy'
```

Update the **show** view to display a **Delete Topic** button:

```
app/views/topics/show.html.erb
```

```
<h1><%= @topic.name %></h1>

<%= link_to "Edit Topic", edit_topic_path, class: 'btn btn-success' %>
+ <%= link_to "Delete Topic", @topic, method: :delete, class: 'btn btn-danger', data: ...
  ...
```

When we delete a topic, its associated posts should also be deleted:

```
models/topic.rb
```

```
class Topic < ApplicationRecord
-   has_many :posts
+   has_many :posts, dependent: :destroy
end
```

Because comments already depend on posts, they will also be deleted when a topic is deleted.

Visit <http://localhost:3000/topics> and click on a topic and make sure you can delete it.

Nesting Posts

Nesting is a term we use when one object should be interacted with in the exclusive context of another object. Associated models, like posts and topics, don't *need* to be nested, but we should nest posts in topics because we never want posts to be viewed, created, or edited in isolation. A nested post's URL will be scoped to topic, for example: `/topics/1/posts/3`. This URL still meets RESTful conventions, and is supported by Rails.

To nest posts under topics, we'll need to refactor `routes.rb`, the `PostsController`, and the topic and post views. Before we refactor, let's update our tests in anticipation of our nested posts:

```
spec/controllers/posts_controller_spec.rb
```

```
require 'rails_helper'

RSpec.describe PostsController do
  # #12
  + let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  # #13
  + let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }

  - let(:my_post) { Post.create( title: RandomData.random_sentence, body: RandomData.random_sentence) }

  # #14
  - describe "GET index" do
    - it "returns http success" do
      - get :index
      - expect(response).to have_http_status(:success)
    - end
    -
    - it "assigns [my_post] to @posts" do
      - get :index
      - expect(assigns(:posts)).to eq([my_post])
    - end
  - end

```

Because posts will be nested under topics, at **#12** we create a parent topic named `my_topic`.

At **#13**, we update how we create `my_post` so that it will belong to `my_topic`.

At **#14**, we remove the `:index` tests. Posts will no longer need an `index` view because they'll be displayed on the `show` view of their parent topic.

`spec/controllers/posts_controller_spec.rb`

```

describe "GET show" do
  it "returns http success" do
-    get :show, params: { id: my_post.id }
# #15
+    get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to have_http_status(:success)
    end

    it "renders the #show view" do
-    get :show, params: { id: my_post.id }
# #16
+    get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to render_template :show
    end

    it "assigns my_post to @post" do
-    get :show, params: { id: my_post.id }
# #17
+    get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(assigns(:post)).to eq(my_post)
    end
  end

```

Posts routes will now include the `topic_id` of the parent topic, so at #15, #16, and #17 we update our `get :show` request to include the id of the parent topic.

`spec/controllers/posts_controller_spec.rb`

```

describe "GET new" do
  it "returns http success" do
-    get :new
# #18
+    get :new, params: { topic_id: my_topic.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #new view" do
-    get :new
# #19
+    get :new, params: { topic_id: my_topic.id }
    expect(response).to render_template :new
  end

  it "initializes @post" do
-    get :new
# #20
+    get :new, params: { topic_id: my_topic.id }
    expect(assigns(:post)).not_to be_nil
  end
end

```

At **#18**, **#19**, and **#20** we update the `get :new` request to include the id of the parent topic.

`spec/controllers/posts_controller_spec.rb`

```

describe "POST create" do
  it "increases the number of Post by 1" do
-    expect{post :create, params: { post: {title: RandomData.random_sentence, body: "# #21" }}}
+    expect{ post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence, body: "# #21" }}}
  end

  it "assigns the new post to @post" do
-    post :create, params: { params: { post: title: RandomData.random_sentence, body: "# #22" }}
+    post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence, body: "# #22" }}
    expect(assigns(:post)).to eq Post.last
  end

  it "redirects to the new post" do
-    post :create, params: { post: name: RandomData.random_sentence, body: RandomData.random_sentence }
-    expect(response).to redirect_to Post.last
    # #23
+    post :create, params: { topic_id: my_topic.id, post: {title: RandomData.random_sentence, body: RandomData.random_sentence } }
    # #24
+    expect(response).to redirect_to [my_topic, Post.last]
  end
end

```

At #21, #22, and #23 we update the `post :create` request to include the id of the parent topic.

At #24, because the route for the posts **show** view will also be updated to reflect nested posts, instead of redirecting to `Post.last`, we redirect to `[my_topic, Post.last]`. Rails' router can take an array of objects and build a route to the show page of the last object in the array, nesting it under the other objects in the array.

`spec/controllers/posts_controller_spec.rb`

```

describe "GET edit" do
  it "returns http success" do
-    get :edit, params: { id: my_post.id }
# #25
+    get :edit, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to have_http_status(:success)
    end

    it "renders the #edit view" do
-    get :edit, params: { id: my_post.id }
# #26
+    get :edit, topic_id: my_topic.id, id: my_post.id
      expect(response).to render_template :edit
    end

    it "assigns post to be updated to @post" do
-    get :edit, params: { id: my_post.id }
# #27
+    get :edit, params: { topic_id: my_topic.id, id: my_post.id }
      post_instance = assigns(:post)

      expect(post_instance.id).to eq my_post.id
      expect(post_instance.title).to eq my_post.title
      expect(post_instance.body).to eq my_post.body
    end
  end

```

At **#25**, **#26**, and **#27** we update the `get :edit` request to include the id of the parent topic.

`spec/controllers/posts_controller_spec.rb`

```

describe "PUT update" do
  it "updates post with expected attributes" do
    new_title = RandomData.random_sentence
    new_body = RandomData.random_paragraph

    -   put :update, params: { id: my_post.id, post: {title: new_title, body: new_body}
      # #28
    +   put :update, params: { topic_id: my_topic.id, id: my_post.id, post: {title: new_title, body: new_body}

      updated_post = assigns(:post)
      expect(updated_post.id).to eq my_post.id
      expect(updated_post.title).to eq new_title
      expect(updated_post.body).to eq new_body
    end

    it "redirects to the updated post" do
      new_title = RandomData.random_sentence
      new_body = RandomData.random_paragraph

      -   put :update, id: my_post.id, post: {title: new_title, body: new_body}
      -   expect(response).to redirect_to my_post
      # #29
    +   put :update, params: { topic_id: my_topic.id, id: my_post.id, post: {title: new_title, body: new_body}}
      # #30
    +   expect(response).to redirect_to [my_topic, my_post]
    end
  end

```

At **#28** and **#29** we update the `put :update` request to include the id of the parent topic.

At **#30**, we replace `redirect_to my_post` with `redirect_to [my_topic, my_post]` so that we'll be redirected to the posts **show** view after we nest posts.

`spec/controllers/posts_controller_spec.rb`

```

describe "DELETE destroy" do
  it "deletes the post" do
-    delete :destroy, params: { id: my_post.id }
# #31
+    delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
    count = Post.where({id: my_post.id}).size
    expect(count).to eq 0
  end

-    it "redirects to posts index" do
-      delete :destroy, params: { id: my_post.id }
-      expect(response).to redirect_to posts_path
+    it "redirects to topic show" do
# #32
+      delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
# #33
+      expect(response).to redirect_to my_topic
    end
  end
end

```

At #31 and #32 we update the `delete :destroy` request to include the id of the parent topic.

At #33, we want to be redirected to the topics **show** view instead of the posts **index** view.

With our tests ready, let's update `routes.rb` to nest posts:

config/routes.rb

```

Rails.application.routes.draw do

-  resources :topics
-  resources :posts
+  resources :topics do
# #34
+    resources :posts, except: [:index]
+  end

  get 'about' => 'welcome#about'

  root 'welcome#index'
end

```

At #34 we pass `resources :posts` to the `resources :topics` block. This nests the post routes under the topic routes.

Examine the new post routes:

Terminal

```
$ rails routes | grep post
topic_posts POST /topics/:topic_id/posts(.:format) posts#create
new_topic_post GET /topics/:topic_id/posts/new(.:format) posts#new
edit_topic_post GET /topics/:topic_id/posts/:id/edit(.:format) posts#edit
topic_post GET /topics/:topic_id/posts/:id(.:format) posts#show
PATCH /topics/:topic_id/posts/:id(.:format) posts#update
PUT /topics/:topic_id/posts/:id(.:format) posts#update
DELETE /topics/:topic_id/posts/:id(.:format) posts#destroy
```

As you can see in the output above, all post URLs are now scoped to a topic.

There's no longer an `index` route for posts. This is because the posts `index` view is no longer needed. All posts will be displayed with respect to a topic now, on the topics `show` view. Remove the `index` action from the `PostsController`:

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController
- def index
-   @posts = Post.all
- end
...
...
```

Remove `app/views/posts/index.html.erb` and stage the removal:

Terminal

```
$ git rm app/views/posts/index.html.erb
```

Run the tests and we'll see seven failures, caused by the `PostsController`:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb
...
16 examples, 7 failures
```

Let's implement the code to pass the tests:

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController
  def show
```

```
  @post = Post.find(params[:id])
end

def new
+  @topic = Topic.find(params[:topic_id])
  @post = Post.new
end

def create
  @post = Post.new
  @post.title = params[:post][:title]
  @post.body = params[:post][:body]
+  @topic = Topic.find(params[:topic_id])
# #35
+  @post.topic = @topic

  if @post.save
    flash[:notice] = "Post was saved."
# #36
-  redirect_to @post
+  redirect_to [@topic, @post]
  else
    flash.now[:alert] = "There was an error saving the post. Please try again."
    render :new
  end
end

def edit
  @post = Post.find(params[:id])
end

def update
  @post = Post.find(params[:id])
  @post.title = params[:post][:title]
  @post.body = params[:post][:body]

  if @post.save
    flash[:notice] = "Post was updated."
# #37
-  redirect_to @post
+  redirect_to [@post.topic, @post]
  else
    flash.now[:alert] = "There was an error saving the post. Please try again."
    render :edit
  end
end

def destroy
```

```

@post = Post.find(params[:id])

if @post.destroy
  flash[:notice] = "\"#{@post.title}\" was deleted successfully."
# #38
-   redirect_to posts_path
+   redirect_to @post.topic
else
  flash.now[:alert] = "There was an error deleting the post."
  render :show
end
end
end

```

At #35 we assign a topic to a post.

At #36 and #37 we change the `redirect` to use the nested post path.

At #38, when a post is deleted, we direct users to the topic **show** view.

With `PostsController` updated to reflect our nested routes, all the tests in `posts_controller_spec.rb` should pass. Run the tests again to confirm and then we'll proceed to updating our views:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb
```

Refactor the Topics Show View

The topic **show** view includes a button that links to `new_post_path(@post)`. As the rails output shows, `new_post_path` is no longer available. (In fact, if you visit a topic's **show** view, you'll see a `NoMethodError` complaining about this.) We need to replace the old method with `new_topic_post_path` to reflect the nested route we generated in `routes.rb`:

```
app/views/topics/show.html.erb
```

```

<h1><%= @topic.name %></h1>

<%= link_to "Edit Topic", edit_topic_path, class: 'btn btn-success' %>
<%= link_to "Delete Topic", @topic, method: :delete, class: 'btn btn-danger', data: ->

<div class="row">
  <div class="col-md-8">
    <p class="lead"><%= @topic.description %></p>
    <% @topic.posts.each do |post| %>
      <div class="media">
        <div class="media-body">
          <h4 class="media-heading">
            <!-- #39 -->
            -<%= link_to post.title, post %>
            +<%= link_to post.title, topic_post_path(@topic, post) %>
          </h4>
          <!-- #40 -->
          +<small>
            +submitted <%= time_ago_in_words(post.created_at) %> ago <br>
            +<%= post.comments.count %> Comments
            +</small>
          </div>
        </div>
        <% end %>
      </div>
      <div class="col-md-4">
        <!-- #41 -->
        -<%= link_to "New Post", new_post_path(@topic), class: 'btn btn-success' %>
        +<%= link_to "New Post", new_topic_post_path(@topic), class: 'btn btn-success' %>
        <%= link_to "Delete Topic", @topic, method: :delete, class: 'btn btn-danger', data: ->
      </div>
    </div>
  </div>

```

At #39, we refactor how we link to individual posts using the `topic_post_path` method. This helper takes a topic and a post and generates a path to the posts **show** view.

At #40, we add submission and comment details for each post in the **show** view.

At #41, we replace `new_post_path` with `new_topic_post_path` to reflect the nested route we generated in `routes.rb`.

Visit a topic's **show** view, and confirm that there is no longer an error.

Refactor the Posts Show View

Each link in the topic **show** view contains a properly nested and RESTful URL for its associated posts. Click on one of the links to an individual post, and you'll see an `Undefined Method edit_post_path` error.

app/views/posts/show.html.erb

```
<h1><%= @post.title %></h1>

<div class="row">
  <div class="col-md-8">
    <p><%= @post.body %></p>
  </div>
  <div class="col-md-4">
    -   <%= link_to "Edit", edit_post_path(@post), class: 'btn btn-success' %>
    -   <%= link_to "Delete Post", @post, method: :delete, class: 'btn btn-danger', data: { confirm: "Are you sure?" } %>
    // #42
    +   <%= link_to "Edit", edit_topic_post_path(@post.topic, @post), class: 'btn btn-success' %>
    // #43
    +   <%= link_to "Delete Post", [@post.topic, @post], method: :delete, class: 'btn btn-danger' %>
  </div>
</div>
```

At #42, we replace `edit_post_path` with `edit_topic_post_path`, which takes two arguments, a topic and a post.

At #43, we update `link_to` to take an array consisting of a topic and post, which it uses to build the link to delete a nested post.

Refresh the post **show** view and the error should be resolved because we used valid methods to generate the URLs for the **Edit** and **Delete Post** buttons.

Refactor the Posts Edit and New Views

If you click on the **Edit** button in the post **show** view, you'll get another undefined method error. This is because `form_for` uses similar path conventions as `link_to`. You've learned how to fix this with the `link_to` method, and we'll use the same approach for `form_for`:

app/views/posts/edit.html.erb

```
...
-   <%= form_for @post do |f| %>
+   <%= form_for [@post.topic, @post] do |f| %>
...
...
```

app/views/posts/new.html.erb

```
-      <%= form_for @post do |f| %>
+      <%= form_for [@topic, @post] do |f| %>
```

Just like `link_to`, `form_for` can take an array of objects instead of a single object to generate the correct paths for nested resources.

Refresh the posts **edit** view to confirm that we've resolved the error.

Top Notch Topics

The topics **index** view is the gateway to our application, so we should give users an easy way to access it from any page by updating `application.html.erb`:

app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>Bloccit</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <%= stylesheet_link_tag    'application', media: 'all', 'data-turbolinks-track' =>
  <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>
  <div class="container">
    <ul class="nav nav-tabs">
      <li><%= link_to "Bloccit", root_path %></li>
+      <li><%= link_to "Topics", topics_path %></li>
      <li><%= link_to "About", about_path %></li>
    ...
  </div>
</body>
</html>
```

Visit any page in Bloccit we'll see a link to our topics **index** view. Create, update, and delete some topics and posts to confirm that everything is working as expected.

Git

Commit your checkpoint work in Git. See **Git Checkpoint Workflow: After Each Checkpoint** for details. Then deploy to Heroku. Because we added a migration in this checkpoint, we'll need to update the Production database:

Terminal

```
$ git push heroku master  
$ heroku run rails db:migrate
```

Recap

Concept	Description
Rails Generate	The <code>rails generate</code> command uses templates to create many things, including models and controllers. Using generators can save you time by auto-generating boilerplate code.
Nested Resources	Some objects should be operated on within the context of another object. Nested resources represent this relationship with URLs that scope one object within another.
Code Refactoring	Code refactoring is the process of restructuring existing computer code, without changing its external behavior. Refactoring is the process of continuously improving code, and, when done with strict TDD, without breaking the application.

How would you rate this checkpoint and assignment?



21. Rails: Topics and Posts

Assignment

Discussion

Submission

21. Rails: Topics and Posts

 Assignment

 Discussion

 Submission

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint](#)

Workflow: Before Each Assignment for details.

As always, complete this assignment using TDD.

Sponsored posts will give us the chance to make some money off of Bloccit! So let's give it the ability to host sponsored posts. For this assignment, we'll create `SponsoredPost` which `belongs_to Topic`.

1. Create a new model named `SponsoredPost`. It should have `title:string`, `body:text`, and `price:integer` attributes and reference a `Topic`.
2. Associate `Topic` and `SponsoredPost` using `has_many` and `belongs_to` in the `Topic` and `SponsoredPost` models respectively.
3. Generate a controller for `SponsoredPosts` with `show`, `new`, and `edit` actions.
4. Nest `SponsoredPost` under `Topic` in `routes.rb`.
5. Seed the database with `SponsoredPosts` via `seeds.rb`.
6. Complete the `SponsoredPostsController` CRUD methods and views. Use **the `number_field` helper** for the price field in the `new` and `edit` views.
7. Update `TopicsController` and the topic `show` view to display a **New Sponsored Post** button.

[←Prev](#)

Submission

[Next→](#)

22 Rails: Validations



“I work hard for the audience. It's entertainment. I don't need validation.”

— Denzel Washington

Overview and Purpose

In this checkpoint you'll learn more about data validation and partials.

Objectives

After this checkpoint, you should be able to:

- Explain data validation.
- Discuss how partials are used within Rails.
- Discuss how form helpers are used.

Validating Data

Think of Bloccit's database as an exclusive club. To get in, you have to present yourself in the right way. The bouncer decides if you will be a good customer for the club or if you are likely to cause a ruckus and ruin the fun for everyone. Validation methods serve as a bouncer for our database and will not allow data to be inserted if the data doesn't look right.

Validation methods are provided by `ActiveRecord` and allow us to define valid states for model attributes. Validations can prevent unwanted data from entering the database. Validation methods allow us to check for things like presence, length, numericality, format, and uniqueness.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow](#):

Before Each Checkpoint for details.

Validation Specs

Let's validate the `Post` model with the following rules. An instance of `Post` must:

- have a title, body, and topic
- have at least five characters in the title
- have at least 20 characters in the body

There are numerous types of validations. The most common are `presence`, `length`, and `format`. The [Rails Guide](#) on validations is a good reference to keep bookmarked.

Add the following tests to `post_spec.rb`:

```
spec/models/post_spec.rb

...
  it { is_expected.to belong_to(:topic) }

+  it { is_expected.to validate_presence_of(:title) }
+  it { is_expected.to validate_presence_of(:body) }
+  it { is_expected.to validate_presence_of(:topic) }

+
+  it { is_expected.to validate_length_of(:title).is_at_least(5) }
+  it { is_expected.to validate_length_of(:body).is_at_least(20) }

+
  describe "attributes" do
    it "has title and body attributes" do
      expect(post).to have_attributes(title: title, body: body)
    ...
  
```

At #1, we test that `Post` validates the presence of `title`, `body`, and `topic`. At #2, we test that `Post` validates the lengths of `title` and `body`.

Run `post_spec.rb` to see that the five new tests fail:

Terminal

```
$ rspec spec/models/post_spec.rb
```

Validating Posts

Add the validations tested above to `Post` using the `validates` method:

app/models/post.rb

```
class Post < ApplicationRecord
  belongs_to :topic
  has_many :comments, dependent: :destroy

  + validates :title, length: { minimum: 5 }, presence: true
  + validates :body, length: { minimum: 20 }, presence: true
  + validates :topic, presence: true

end
```

Run `post_spec.rb` again to confirm that our validations satisfy our tests:

Terminal

```
$ rspec spec/models/post_spec.rb
```

Validations in Action

Examine how these validations work by opening the Rails console and creating a new `Post` object:

Restart the console if it's already running.

Console

```
$ rails c
>> my_post = Post.new
=> #<Post id: nil, title: nil, body: nil, created_at: nil, updated_at: nil, topic_id:
```

Check to see if `my_post` is valid:

Console

```
>> my_post.valid?
=> false
```

Print the errors that make `my_post` invalid:

Console

```
>> my_post.errors  
=> #<ActiveModel::Errors:0x007fad729366c0 @base=#<Post id: nil, title: nil, body: nil,
```

Print the complete error messages:

Console

```
>> my_post.errors.full_messages  
=> ["Title is too short (minimum is 5 characters)", "Title can't be blank", "Body is
```

Print the `:title` error:

Console

```
>> my_post.errors[:title]  
=> ["is too short (minimum is 5 characters)", "can't be blank"]
```

Since `my_post` was created without values, it didn't meet the conditions we validate in `Post`. If you try to save it, the method will return `false` and fail to save. If you call the `save!` method on the post, it will throw an error.

Remember what `!` **does**?

Console

```
> my_post.save!  
  (0.1ms)  begin transaction  
  (0.1ms)  rollback transaction  
ActiveRecord::RecordInvalid: Validation failed: Title is too short (minimum is 5 chara
```

Quick challenge: Create a new `Post` object from the console which satisfies all of the validations.

Displaying Validation Errors

Let's change our post views to display errors raised by our validations. Because these changes will affect the forms on both the `new` and the `edit` views, it's a good time to demonstrate how to use partials to make our views DRY.

Open the `edit` view and make the following changes:

```
app/views/posts/edit.html.erb
```

```

<h1>Edit Post</h1>

<div class="row">
  <div class="col-md-4">
    <p>Guidelines for posts</p>
    <ul>
      <li>Make sure it rhymes.</li>
      <li>Don't use the letter "A".</li>
      <li>The incessant use of hashtags will get you banned.</li>
    </ul>
  </div>
  <div class="col-md-8">
    -  <%= form_for [@post.topic, @post] do |f| %>
    -    <div class="form-group">
    -      <%= f.label :title %>
    -      <%= f.text_field :title, class: 'form-control', placeholder: "Enter post title" %>
    -    </div>
    -    <div class="form-group">
    -      <%= f.label :body %>
    -      <%= f.text_area :body, rows: 8, class: 'form-control', placeholder: "Enter post body" %>
    -    </div>
    -    <div class="form-group">
    -      <%= f.submit "Save", class: 'btn btn-success' %>
    -    </div>
    -  <% end %>
    +  <%= render partial: 'form', locals: { topic: @post.topic, post: @post } %>
  </div>
</div>

```

We just replaced a large chunk of code with a **partial**. Partials are fragments of view code, which are called from views. Partials are called with the `render` method. Rendering a partial is like copying and pasting view code from another file.

We've rendered a partial in the **edit** view, but this partial file doesn't yet exist. We need to create the partial and add the code we just removed from the **edit** view.

Why remove code from one view file, just to paste it in another? Partials make our code DRYer, clearer, and more modular. By isolating conceptual units in reusable files, we can call them from multiple views without repeating ourselves. Partials also make complex views more readable for developers.

Form Partial

Create a new file in `app/views/posts` named `_form.html.erb`. The underscore at the

beginning of a file name lets Rails know that it's a partial. Add the code we deleted from the **edit** view to `_form.html.erb`:

app/views/posts/_form.html.erb

```
<!-- #3 -->
+ <%= form_for [topic, post] do |f| %>
+   <div class="form-group">
+     <%= f.label :title %>
+     <%= f.text_field :title, class: 'form-control', placeholder: "Enter post title" %>
+   </div>
+   <div class="form-group">
+     <%= f.label :body %>
+     <%= f.text_area :body, rows: 8, class: 'form-control', placeholder: "Enter post body" %>
+   </div>
+   <div class="form-group">
+     <%= f.submit "Save", class: 'btn btn-success' %>
+   </div>
+ <% end %>
```

At **#3** we use the `topic` and `post` **local variables** we passed into the partial when we rendered it:

app/views/posts/edit.html.erb

```
<%= render partial: 'form', locals: { topic: @post.topic, post: @post } %>
```

Our view is called `_form.html.erb`, while we render the `partial: 'form'`. This is a Rails convention for partial naming. If we don't follow it, the partial will not be found.

`locals: { topic: @post.topic, post: @post }` passes local variables to the partial. `@post.topic` and `@post` are assigned to `topic` and `post`, respectively. This means we can refer to these local variables, rather than the `@post` **instance** variable we used in the **edit** view.

Although the partial **does** have access to `@post`, it's a better practice to use local variables. A primary reason is flexibility. We want to be able to render our partial on **any** page, whether or not it has access to the same instance variables.

Start the Rails server, navigate to the **edit** view, and make sure no errors appear. If the partial worked, the **edit** view should appear as it did before the refactor. The great thing about partials is that they're reusable. The form for creating new posts is the same as the form for editing, so let's refactor the **new** view next.

Refactor the Posts New View

Open the posts **new** view and make the following changes:

app/views/posts/new.html.erb

```
<h1>New Post</h1>

<div class="row">

  <div class="col-md-4">

    <p>Guidelines for posts</p>
    <ul>
      <li>Make sure it rhymes.</li>
      <li>Don't use the letter "A".</li>
      <li>The incessant use of hashtags will get you banned.</li>
    </ul>
  </div>

  <div class="col-md-8">

    - <%= form_for [@topic, @post] do |f| %>
    -   <div class="form-group">
    -     <%= f.label :title %>
    -     <%= f.text_field :title, class: 'form-control', placeholder: "Enter post title" %>
    -   </div>
    -   <div class="form-group">
    -     <%= f.label :body %>
    -     <%= f.text_area :body, rows: 8, class: 'form-control', placeholder: "Enter post body" %>
    -   </div>
    -   <div class="form-group">
    -     <%= f.submit "Save", class: 'btn btn-success' %>
    -   </div>
    -   <% end %>
    +   <%= render partial: 'form', locals: { topic: @topic, post: @post } %>
  </div>
</div>
```

Partials are essentially the Rails view version of **extract-method refactoring**. By consolidating reused code, we make sure we only have to edit it in one location, and we can reuse it easily.

Visit the **new** view and validate that it looks the same as it did prior to the refactor.

While we're on **new** view, let's create a new post, leaving the title and body blank. We should see an error that looks like this:

There was an error saving the post. Please try again.

X

New Post

This error isn't very informative. In the next section we'll refactor to show the error messages raised by the `validates` methods.

The Application Helper

Styling the elements on the form partial is a little tricky. If the form displays errors, we want a certain style, and if the form doesn't display errors, we want a different style. We could include conditional logic directly in our view, but Rails conventions strongly suggest keeping as much logic as possible out of the view. Views cluttered with if statements are hard to read, hard to maintain, and confusing. We'll use the `ApplicationHelper` to implement a DRY solution for toggling the `div` classes we'll need to display an error-full or error-less view.

Methods in `ApplicationHelper` can be used across our application. It's a `Module` that Rails includes with other classes in our app. Any public method we write in `ApplicationHelper` will be available in all views.

Open `app/helpers/application_helper.rb` and add the following code:

app/helpers/application_helper.rb

```
module ApplicationHelper
  # #4
  + def form_group_tag(errors, &block)
  +   css_class = 'form-group'
  +   css_class << ' has-error' if errors.any?
  # #5
  +   content_tag :div, capture(&block), class: css_class
  + end
end
```

At #4, we define a method named `form_group_tag` which takes two arguments. The first argument is an array of errors, and the second is a block.

The `&` turns the block into a `Proc`, which we've seen before but haven't named. A `Proc` is a block that can be reused like a variable.

At #5, the `content_tag` helper method is called. This method is used to build the HTML

and CSS to display the form element and any associated errors.

Helpers are written in Ruby and usually return HTML. The `content_tag` is one such method. It takes a symbol argument, a block, and an options hash. It then creates the symbol-specified HTML tag with the block contents, and if specified, the options.

We need a different `div` based on the errors raised by the `validates` method, so let's use `form_group_tag` in the form partial:

app/view/posts/_form.html.erb

```
<%= form_for [topic, post] do |f| %>
-   <div class="form-group">
+   <% if post.errors.any? %>
# #6
+   <div class="alert alert-danger">
# #7
+     <h4><%= pluralize(post.errors.count, "error") %></h4>
+     <ul>
+       <% post.errors.full_messages.each do |msg| %>
+         <li><%= msg %></li>
+       <% end %>
+     </ul>
+   </div>
+   <% end %>
# #8
+   <%= form_group_tag(post.errors[:title]) do %>
    <%= f.label :title %>
    <%= f.text_field :title, class: 'form-control', placeholder: "Enter post title" %>
-   </div>
-   <div class="form-group">
+   <% end %>
# #9
+   <%= form_group_tag(post.errors[:body]) do %>
    <%= f.label :body %>
    <%= f.text_area :body, rows: 8, class: 'form-control', placeholder: "Enter post body" %>
-   </div>
+   <% end %>
<div class="form-group">
  <%= f.submit "Save", class: 'btn btn-success' %>
</div>
<% end %>
```

At #6, if there are any validation errors, we display an alert with the number of errors and their messages. At #7, we use the `pluralize` method to pluralize "error" if there is more than one error. At #8, we use `form_group_tag` to display `title` errors. At #9, we use `form_group_tag` to display `body` errors.

Open the post **new** view and submit a new post with no `title` or `body`. Validate that you see the correctly formatted error messages. Also submit a new post with a valid `title` and `body` to ensure that, given valid inputs, a user can successfully create a new post.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Recap

Concept	Description
Data validation	Data validation is the process of ensuring that the data received by a program is clean, correct, and useful.
Partial templates	Partial templates - called "partials" - split the rendering process into manageable segments. Partials allow the code which renders a particular piece of HTML to be moved to its own file.
Form Helpers	Form helpers generate HTML form markup. They are used to reduce the burden of writing and maintaining HTML form markup.

How would you rate this checkpoint and assignment?



22. Rails: Validations

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

Assignment

In addition to validating posts, we should validate topics. Use TDD to complete this assignment.

1. Add a validation to ensure that topic names are at least five characters long and descriptions are at least 15 characters long.
2. Test your validation by creating a topic in the Rails console with a four character name.
3. Create a form partial for topics.
4. Refactor the topic **edit** and **new** views to use the form partial.
5. Use the `form_group_tag` helper method created in the checkpoint to display errors for invalid topics.

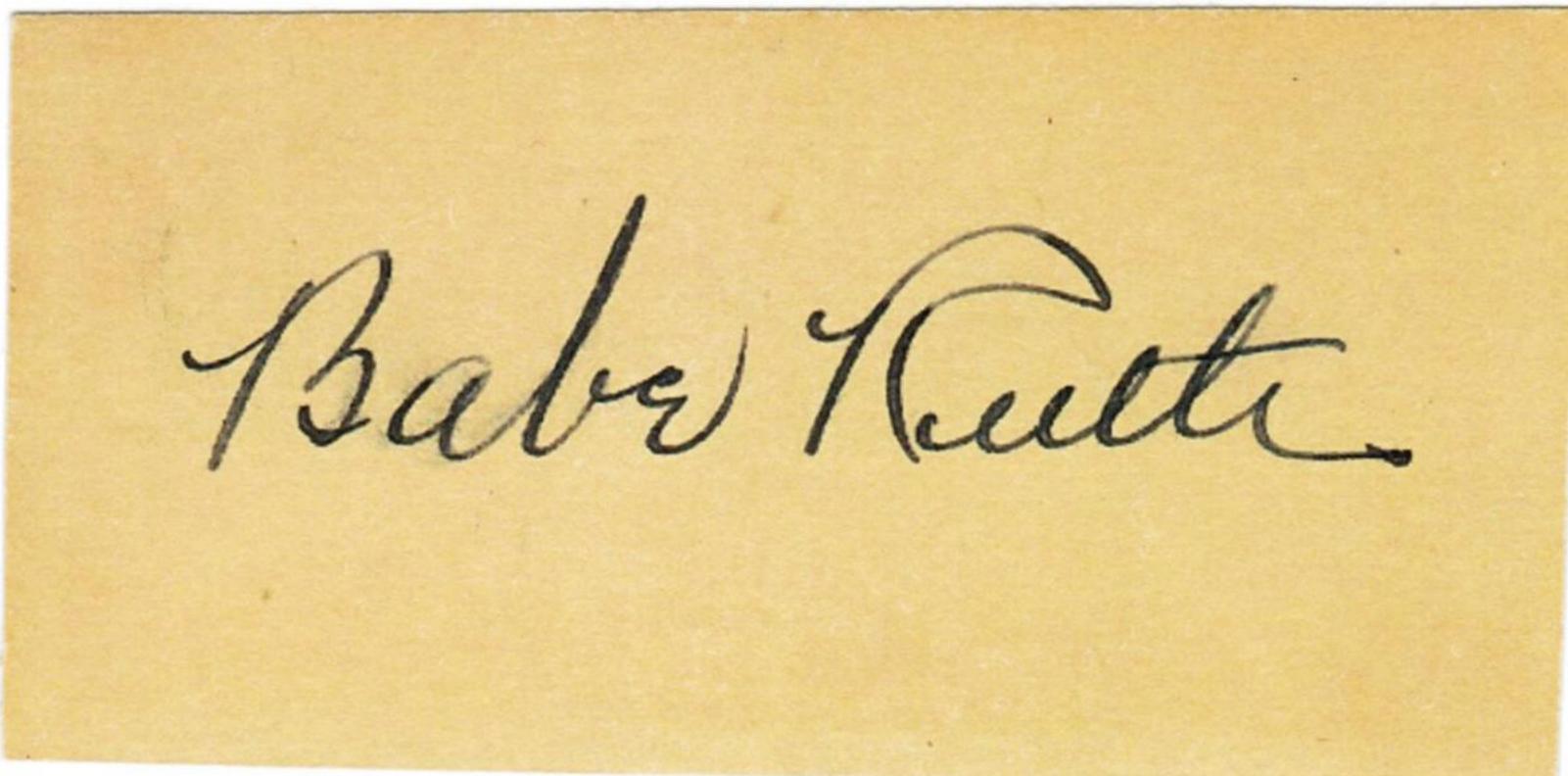
Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.

[←Prev](#)

Submission

[Next→](#)

23 Rails: Authentication - User Model



“Truth is a point of view, but authenticity can't be faked.”

— Peter Guber

Overview and Purpose

In this checkpoint you'll learn more about `ActiveRecord` callbacks, validations and regular expressions.

Objectives

After this checkpoint, you should be able to:

- Elaborate on how callbacks work in Rails.
- Elaborate on `ActiveRecord` validations.
- Explain regular expressions and their purpose.

Modeling Users

User authentication systems determine whether a user is who they claim to be. They allow users to sign up, sign in, and sign out. We'll build the foundation of Bloccit's user authentication system by creating the `User` model.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Custom Authentication

There are some popular authentication systems for Rails, including [Devise](#), [OmniAuth](#), and [AuthLogic](#). While these systems offer robust functionality, building our own solution will give us a deeper understanding of authentication-based models, controllers, and routing.

Generate User

Before we build authentication functionality, we'll need a user to authenticate. Generate a `User` model to represent the users of Bloccit with the following attributes:

Attribute	Description
<code>name</code>	A string to represent the user's name.
<code>email</code>	A string to represent the user's email.
<code>password_digest</code>	A string to store the user's hashed password.

Terminal

```
$ rails generate model User name:string email:string password_digest:string
```

Run the migration:

Terminal

```
$ rails db:migrate
```

When we migrate the database, we are ultimately changing the database's schema, which describes the layout, structure, and contents of the database. Open `db/schema.rb` and review the schema.

`schema.rb` serves two important purposes. It represents the current state of the database, which can be difficult to deduce from the migration files. It also populates the database schema before tests are executed.

Test User

Let's write our first specs for `User`:

`spec/models/user_spec.rb`

```
require 'rails_helper'

RSpec.describe User, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
  + let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  + # Shoulda tests for name
  + it { is_expected.to validate_presence_of(:name) }
  + it { is_expected.to validate_length_of(:name).is_at_least(1) }

  +
  + # Shoulda tests for email
  + it { is_expected.to validate_presence_of(:email) }
  + it { is_expected.to validate_uniqueness_of(:email) }
  + it { is_expected.to validate_length_of(:email).is_at_least(3) }
  + it { is_expected.to allow_value("user@bloccit.com").for(:email) }

  +
  + # Shoulda tests for password
  + it { is_expected.to validate_presence_of(:password) }
  + it { is_expected.to have_secure_password }
  + it { is_expected.to validate_length_of(:password).is_at_least(6) }

  +
  + describe "attributes" do
    it "should have name and email attributes" do
      expect(user).to have_attributes(name: "Bloccit User", email: "user@bloccit.com")
    end
  end
end
```

As with our other model tests, the tests above test for field validation and attributes. Let's simulate and add tests for an invalid user:

`spec/models/user_spec.rb`

```
...
# #1
+ describe "invalid user" do
+   let(:user_with_invalid_name) { User.new(name: "", email: "user@bloccit.com") }
+   let(:user_with_invalid_email) { User.new(name: "Bloccit User", email: "") }

+   it "should be an invalid user due to blank name" do
+     expect(user_with_invalid_name).to_not be_valid
+   end

+   it "should be an invalid user due to blank email" do
+     expect(user_with_invalid_email).to_not be_valid
+   end
+
+ end
end
```

At **#1**, we wrote a test that does not follow the same conventions as our previous tests. We are testing for a value that we know should be invalid. We call this a **true negative**, as we are testing for a value that *shouldn't* exist. A **true positive** follows the reciprocal pattern and tests for a known and valid value. True negatives are a useful testing strategy, because if we only test for values that we know should exist, we may not catch values that shouldn't.

Run the tests, and we'll see 14 failures:

Terminal

```
$ rspec spec/models/user_spec.rb
```

User Model

Let's add the functionality we just tested for in `user_spec.rb`:

app/models/user.rb

```

class User < ActiveRecord
  # #2
  + before_save { self.email = email.downcase if email.present? }

  # #3
  + validates :name, length: { minimum: 1, maximum: 100 }, presence: true
  # #4
  + validates :password, presence: true, length: { minimum: 6 }, if: "password_digest.nil?"
  + validates :password, length: { minimum: 6 }, allow_blank: true
  # #5
  + validates :email,
    presence: true,
    uniqueness: { case_sensitive: false },
    length: { minimum: 3, maximum: 254 }

  # #6
  + has_secure_password
end

```

At #2, we register an inline callback directly after the `before_save` **callback**.

`{ self.email = email.downcase }` is the code that will run when the callback executes.

Per Ruby's documentation, "**callbacks are hooks into the life cycle of an Active Record object that allow you to trigger logic before or after an alteration of the object state.**" Callbacks are similar to receiving mail that has the "return service requested" mandate, along with a form to complete. As the receiver of the mail, we are requested to return an envelope with the completed form. The requester in this example is analogous to a `User` in our code. `User` is requesting that the database execute `{ self.email = email.downcase }`. The database is analogous to the receiver of the mail and is asked to execute `{ self.email = email.downcase }`.

At #3, we use Ruby's `validates` function to ensure that `name` is present and has a maximum and minimum length.

At #4, we validate password with two separate validations:

- The first validation executes if `password_digest` is `nil`. This ensures that when we create a new user, they have a valid password.
- The second validation ensures that when updating a user's password, the updated password is also six characters long. `allow_blank: true` skips the validation if no updated password is given. This allows us to change other attributes on a user without being forced to set the password.

At #5, we validate that `email` is present, unique, case insensitive, has a minimum length, has a maximum length, and that it is a properly formatted email address.

At #6, we use Ruby's `has_secure_password`. `has_secure_password` "adds methods to set and authenticate against a BCrypt password. This mechanism requires you to have a `password_digest` attribute". This function abstracts away much of the complexity of obfuscating user passwords using hashing algorithms which we would otherwise be inclined to write to securely save passwords. `has_secure_password` requires a `password_digest` attribute on the model it is applied to. `has_secure_password` creates two virtual attributes, `password` and `password_confirmation` that we use to set and save the password.

To use `has_secure_password`, we need to install `BCrypt`. `BCrypt` is a module that encapsulates complex hashing algorithms. `BCrypt` takes a plain text password and turns it into an unrecognizable string of characters using a hashing algorithm such as MD5. Typically, hashing algorithms are one directional so that if someone gains access to the hashed password, they will not be able to reverse the hashing algorithm to gain the plaintext password. This way, our password is safe even if someone gains access to our database.

Let's add `BCrypt` to our Gemfile. Add it to the bottom of the Gemfile, outside of any blocks:

Gemfile

```
...
gem 'bootstrap-sass'

+ # Used for encrypting passwords
+ gem 'bcrypt'
```

...

As usual after adding a gem to the Gemfile, we need to install it:

Terminal

```
$ bundle install
```

...

Let's run our specs again:

Terminal

```
$ rspec spec/models/user_spec.rb
```

```
.....
```

```
Finished in 0.111 seconds (files took 1.41 seconds to load)  
15 examples, 0 failures
```

Our specs now pass and we are in the green phase of TDD. We still need to add user routes though:

config/routes.rb

```
...  
resources :topics do  
  resources :posts, except: [:index]  
end  
  
# #7  
+ resources :users, only: [:new, :create]  
...  
...
```

At #7, we create routes for `new` and `create` actions. The `only` hash key will prevent Rails from creating unnecessary routes.

Run `rails routes` to see the user routes that were generated:

Terminal

```
$ rails routes | grep user
```

We're now able to represent and store users in our app.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

Recap

Concept

Description

Active Record Callbacks

Callbacks are a convenient and powerful way to ensure that logic happens before or after an operation is performed.

Active Record Validations

Validations act as constraints on model attributes.

Regular Expressions

A regular expression defines a specific character pattern that is used to match against a string. If you want to review regular expressions in more depth, [regexr.com](#) has an online tool to experiment with.

`has_secure_password`

Integrates with `BCrypt` to provide safe authentication.

How would you rate this checkpoint and assignment?



23. Rails: Authentication - User Model

Assignment

Discussion

Submission

23. Rails: Authentication - User Model

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint](#)

[Workflow: Before Each Assignment](#) for details.

Add another callback to `User`. The callback should format all names in the same manner, by capitalizing the first letter of both the first and last names of a given user. The callback should follow these instructions for formatting:

1. Run before `User` is saved.
2. Split the user's `name` on a space (e.g. between a first name and a last name). Loop over each name and uppercase the first letter. Re-combine the first and last names with a space in-between and save it to the `name` attribute.
3. Your solution does *not* have to check for an existing name that is properly formatted. For example, "Steve Jobs" is properly formatted, but your solution should work on it anyways.
4. Write your solution using TDD.

These functions may help you complete this assignment:

- `String#split`
- `String#capitalize`
- `Array#join`

Assign

[←Prev](#)

Submission

[Next→](#)

24 Rails: Authentication - Signing Up



“Honesty and transparency make you vulnerable. Be honest and transparent anyway.”

— Mother Theresa

Overview and Purpose

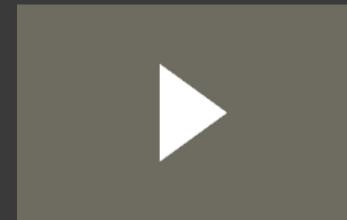
In this checkpoint you'll add users to your application.

Objectives

After this checkpoint, you should be able to:

- Create a basic user scheme for a Ruby on Rails application.
- Give the users of a Ruby on Rails application the ability to sign up for your application.

Signing Up



BLOC

Intro: Bloccit - Authentication Sign Up

0:48



Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow](#):

Before Each Checkpoint for details.

Bloccit has the ability to model and persist users, but we have no way of creating a user or signing up for the application. We'll implement sign up in this checkpoint.

UsersController

To create users, we'll need a controller. Create the `UsersController`:

Terminal

```
$ rails generate controller Users
```

Start the Rails server and open `localhost:3000/users/new`. We see an error stating that "**The action 'new' could not be found for UsersController**". This is because we haven't created actions for `UsersController` and we don't have the proper routes in place. We'll write `UsersController` using TDD, as usual.

Add the following code in `users_controller_spec.rb`:

spec/controllers/users_controller_spec.rb

```
require 'rails_helper'

RSpec.describe UsersController, type: :controller do
  # #1
  + let(:new_user_attributes) do
  +   {
  +     name: "BlocHead",
  +     email: "blochead@bloc.io",
  +     password: "blochead",
  +     password_confirmation: "blochead"
  +   }
  + end
end
```

At **#1**, we create a hash of attributes named `new_user_attributes` so we can use them easily throughout our spec.

new Action

spec/controllers/users_controller_spec.rb

```
...  
# #2  
+ describe "GET new" do  
+   it "returns http success" do  
+     get :new  
+     expect(response).to have_http_status(:success)  
+   end  
  
+   it "instantiates a new user" do  
+     get :new  
+     expect(assigns(:user)).to_not be_nil  
+   end  
+ end
```

...

At #2, we test the `new` action for HTTP success when issuing a `GET`. The first test expects the response to return an HTTP response code of 200. The second test expects `new` to instantiate a new user.

Run the spec, and see that we have two failing tests.

Terminal

```
$ rspec spec/controllers/users_controller_spec.rb
```

Let's write the implementation code for `new` and pass these tests:

app/controllers/users_controller.rb

```
class UsersController < ApplicationController  
+ def new  
+   @user = User.new  
+ end  
end
```

Let's also create the `new` view:

terminal

```
$ touch app/views/users/new.html.erb
```

As before, we created an instance variable named `@user` to be used by the `new` view's form. Run the spec again and our two tests should pass:

```
$ rspec spec/controllers/users_controller_spec.rb
```

create Action

`create` is the action that is called when the `new` view's form is submitted with valid attributes. Let's write the tests first:

```
spec/controllers/users_controller_spec.rb
```

```

...
# #3
+ describe "POST create" do
+   it "returns an http redirect" do
+     post :create, params: { user: new_user_attributes }
+     expect(response).to have_http_status(:redirect)
+   end
+
# #4
+   it "creates a new user" do
+     expect{
+       post :create, params: { user: new_user_attributes }
+     }.to change(User, :count).by(1)
+   end
+
# #5
+   it "sets user name properly" do
+     post :create, params: { user: new_user_attributes }
+     expect(assigns(:user).name).to eq new_user_attributes[:name]
+   end
+
# #6
+   it "sets user email properly" do
+     post :create, params: { user: new_user_attributes }
+     expect(assigns(:user).email).to eq new_user_attributes[:email]
+   end
+
# #7
+   it "sets user password properly" do
+     post :create, params: { user: new_user_attributes }
+     expect(assigns(:user).password).to eq new_user_attributes[:password]
+   end
+
# #8
+   it "sets user password_confirmation properly" do
+     post :create, params: { user: new_user_attributes }
+     expect(assigns(:user).password_confirmation).to eq new_user_attributes[:password_confirmation]
+   end
+
+ end

```

At #3, we test the `create` action for HTTP success when issuing a `POST` with `new_user_attributes` set as the params hash.

At #4, we test that the database count on the users table increases by one when we issue a `POST` to `create`.

At #5, we test that we set `user.name` properly when creating a user.

At #6, we test that we set `user.email` properly when creating a user.

At #7, we test that we set `user.password` properly when creating a user.

At #8, we test that we set `user.password_confirmation` properly when creating a user.

Run the spec to see the six new failures:

Terminal

```
$ rspec spec/controllers/users_controller_spec.rb
```

Let's pass these tests by updating `UsersController` with a `create` action:

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

+  def create
+    # #9
+    @user = User.new
+    @user.name = params[:user][:name]
+    @user.email = params[:user][:email]
+    @user.password = params[:user][:password]
+    @user.password_confirmation = params[:user][:password_confirmation]
+
+    # #10
+    if @user.save
+      flash[:notice] = "Welcome to Bloccit #{@user.name}!"
+      redirect_to root_path
+    else
+      flash.now[:alert] = "There was an error creating your account. Please try again."
+      render :new
+    end
+  end

end
```

At #9, we create a new user with `new` and then set the corresponding attributes from the `params` hash.

At #10, we `save` the new user to the database. If the database `save` is successful, we

add a `flash` message and then redirect the user to the root path. Otherwise, we display an error and render the `new` view.

Run the spec again and our six tests pass.

Remember that we already created `new` and `create` routes in the previous checkpoint, so we do not need to modify `routes.rb` at this point, as we have done in the past with other controllers.

HTML

Let's build the views for creating users. Open the users `new` view and add the following:

```
app/views/users/new.html.erb
```

```

+ <h2>Sign up</h2>
+
+ <div class="row">
+   <div class="col-md-8">
+     <%= form_for @user do |f| %>
<!-- #11 -->
+       <% if @user.errors.any? %>
+         <div class="alert alert-danger">
+           <h4><%= pluralize(@user.errors.count, "error") %></h4>
+           <ul>
+             <% @user.errors.full_messages.each do |msg| %>
+               <li><%= msg %></li>
+             <% end %>
+           </ul>
+         </div>
+       <% end %>
<!-- #12 -->
+       <%= form_group_tag(@user.errors[:name]) do %>
+         <%= f.label :name %>
+         <%= f.text_field :name, autofocus: true, class: 'form-control', placeholder:>
+       <% end %>
+       <%= form_group_tag(@user.errors[:email]) do %>
+         <%= f.label :email %>
+         <%= f.email_field :email, class: 'form-control', placeholder: "Enter email" %>
+       <% end %>
+       <%= form_group_tag(@user.errors[:password]) do %>
+         <%= f.label :password %>
+         <%= f.password_field :password, class: 'form-control', placeholder: "Enter password" %>
+       <% end %>
+       <%= form_group_tag(@user.errors[:password_confirmation]) do %>
+         <%= f.label :password_confirmation %>
+         <%= f.password_field :password_confirmation, class: 'form-control', placeholder: "Enter password confirmation" %>
+       <% end %>
+       <div class="form-group">
+         <%= f.submit "Sign up", class: 'btn btn-success' %>
+       </div>
+     <% end %>
+   </div>
+ </div>

```

At #11, we check the `errors` hash on `@user`. The `errors` hash is provided by `ActiveModel::Errors`. If there are errors with `@user`, such as invalid attributes, we display the corresponding error messages.

At #12, we add form fields for `name` and `email`, as well as the virtual attributes provided by `has_secure_password`: `password` and `password_confirmation`.

It's been a while since we updated our home page. Let's use this opportunity to enhance the home page with an improved design and a call-to-action for signing up:

app/views/welcome/index.html.erb

```
- <h1>Welcome to Bloccit</h1>
- <p id="index-title">This is the home page for Bloccit.</p>
- <div class="posts">Post 1 goes here.</div>
- <div class="posts">Post 2 goes here.</div>
- <section>I am the content in a section element.</section>
+ <div class="jumbotron">
+   <h1>Bloccit</h1>
+   <p>Bloccit is a resource for sharing links with your friends!</p>
+   <p>
+     <%= link_to "Sign Up", new_user_path, class: 'btn btn-primary' %> today!
+   </p>
+ </div>
+
+ <div class="row">
+   <div class="col-md-4">
+     <h2>Solves world hunger</h2>
+     <p>Bloccit will deliver food and water to those in need, all over the world.</p>
+   </div>
+   <div class="col-md-4">
+     <h2>Eliminates poverty</h2>
+     <p>Bloccit will deliver money and education to those in need, all over the world.</p>
+   </div>
+   <div class="col-md-4">
+     <h2>Makes you better looking</h2>
+     <p>Bloccit will make you better looking: it's scientifically proven to make men
+   </div>
+ </div>
```

Let's also add a **Sign Up** link to the top navigation:

app/views/layouts/application.html.erb

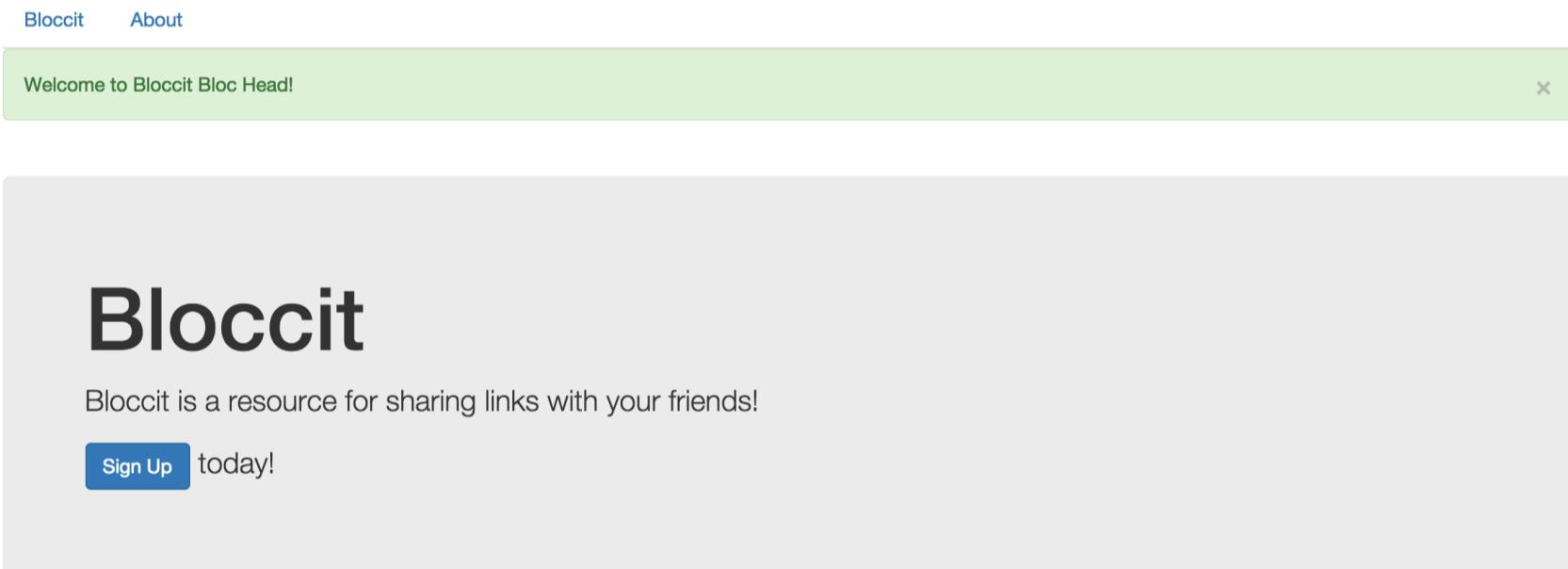
```

...
<ul class="nav nav-tabs">
  <li><%= link_to "Bloccit", root_path %></li>
  <li><%= link_to "Topics", topics_path %></li>
  <li><%= link_to "About", about_path %></li>
+   <li class="pull-right"><%= link_to "Sign Up", new_user_path %></li>
</ul>
```

```
<% if flash[:notice] %>
```

```
...
```

Try creating valid and invalid users. Your results should look similar to the images below. First for a valid user:



Solves world hunger

Bloccit will deliver food and water to those in need, all over the world.

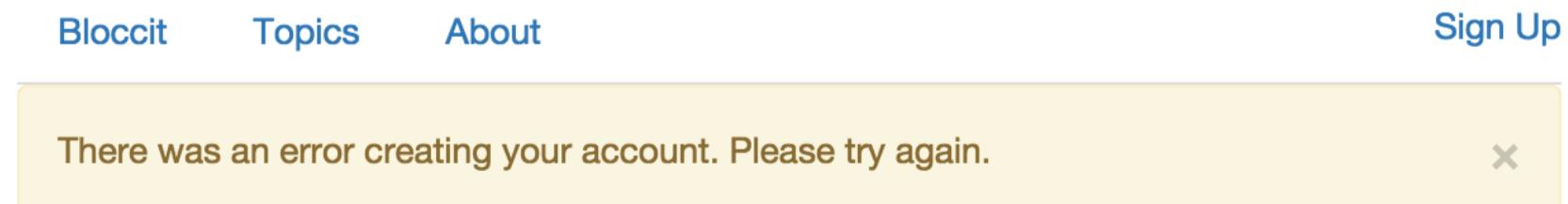
Eliminates poverty

Bloccit will deliver money and education to those in need, all over the world.

Makes you better looking

Bloccit will make you better looking: it's scientifically proven to make men look like Ryan Gosling and women look like Shakira.

... and if you try to create an invalid user, with a mismatching Password and Password Confirmation:



Sign up

1 error.

- Password confirmation doesn't match Password

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Deploy your updated master branch to production:

Terminal

```
$ git push heroku master
```

How would you rate this checkpoint and assignment?



24. Rails: Authentication - Signing Up

Assignment

Discussion

Submission

24. Rails: Authentication - Signing Up

 **Assignment**

 **Discussion**

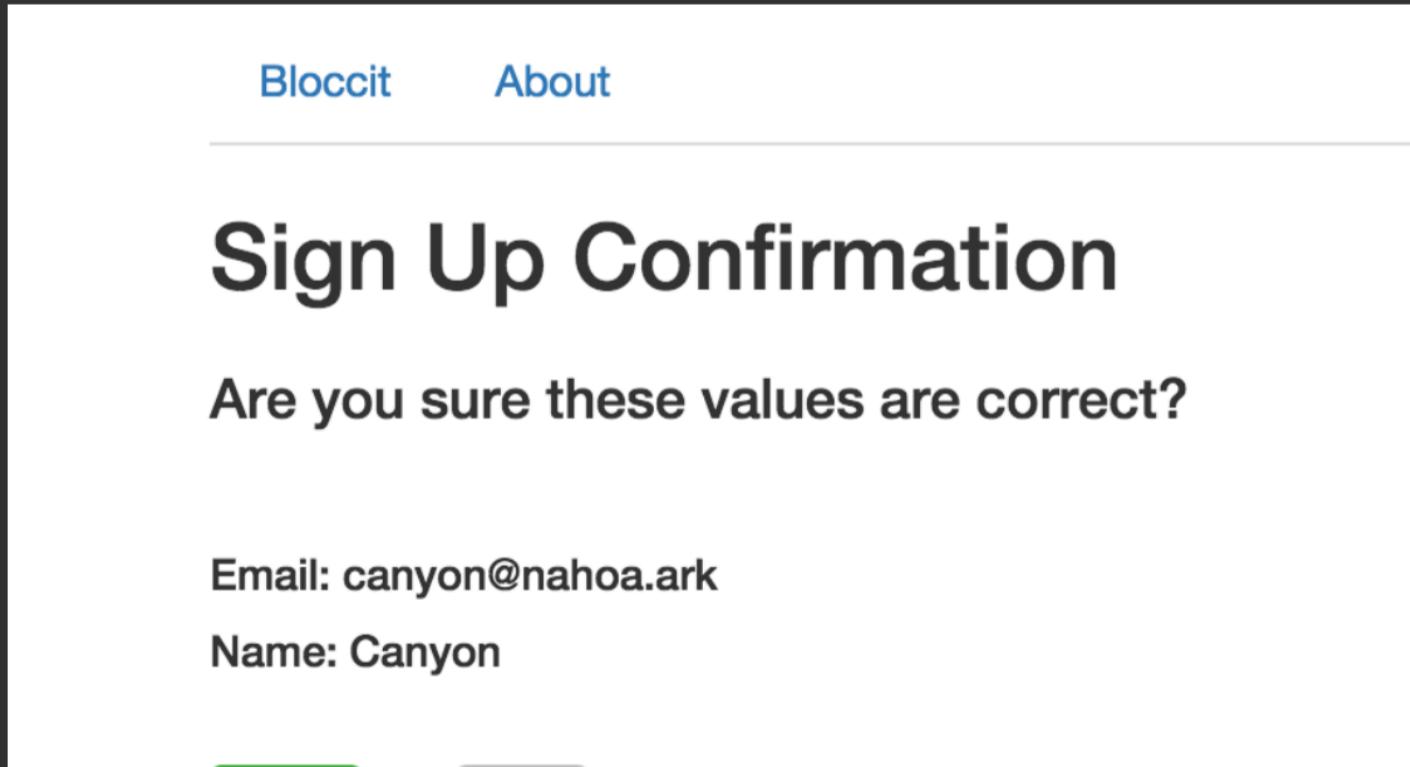
 **Submission**

Exercises

Create a new Git feature branch for this assignment. See **Git Checkpoint Workflow: Before Each Assignment** for details.

Let's add a static confirmation HTML page after a user submits the sign up form. The confirmation should verify the name and email address they entered by displaying the values they entered and asking them if they are correct.

1. Create a new view for confirmation. It should look something like this:



Bloccit About

Sign Up Confirmation

Are you sure these values are correct?

Email: canyon@nahoark

Name: Canyon

2. Your form in `users/new.html.erb` will need to point to the new confirmation page. You can use `url: { action: :confirm }` in `form_for` to accomplish this:

```
app/views/users/new.html.erb
```

```
...
```

```
<%= form_for @user, url: { action: :confirm } do |f| %>
```

```
...
```

3. Add a route for your confirmation page. The route should be a `POST` to '`users/confirm`' and point to `users#confirm`.
4. Create a `confirm` action in `UsersController`. `confirm` should create a new user from the params hash and set its attributes appropriately. Send the params hash back to the `create` action in `UsersController` to save the params to the database after user confirmation. Use a button in the confirm form to achieve this:

```
<%= button_to "Yes", {controller: "users", action: "create",  
params: params}, class: 'btn btn-success' %>
```

5. When a user clicks the "No" button it should take them back to the initial sign up view.

[←Prev](#)

Submission

[Next→](#)

25 Rails: Authentication - Signing In



“Milk is for babies. When you grow up you have to drink beer.”

— Arnold Schwarzenegger

Overview and Purpose

This checkpoint introduces you to sessions.

Objectives

After this checkpoint, you should be able to:

- Explain what a session is.
- Explain why sessions are necessary.
- Incorporate Gravatar into an application.

Signing In

Now that users can *sign up* for Bloccit, they'll need to be able to *sign in*. Signing in to an application requires user information to persist while a user is signed in to Bloccit. That is, we must authenticate and retain user information so that we know who the user is until they sign out. We'll use a **session object** to persist a user's information after they sign in to Bloccit.

Git

Create a new Git feature branch for this checkpoint. See **Git Checkpoint Workflow: Before Each Checkpoint** for details.

Sessions

There are many ways to implement a session object. The canonical implementation of a session is an **HTTP cookie** and is used by many web applications, including Ruby on Rails applications. We'll use a combination of our own code plus Rails' **session object** to implement sessions in Bloccit.

Sessions are tracked and persisted by storing values in the client's cookies. They persist for a duration of time, or a duration based on action (i.e. sign in and sign out). Our session object will persist a user's information while they are signed in to Bloccit.

Generate a session controller without any actions:

Terminal

```
$ rails generate controller Sessions
```

We'll use this controller to create and destroy a user's session.

TDD the SessionsController

Let's TDD the routing functions for our session object:

```
spec/controllers/sessions_controller_spec.rb
```

```

require 'rails_helper'

RSpec.describe SessionsController, type: :controller do
  let(:my_user) { User.create!(name: "Blochead", email: "blochead@bloc.io", password: "password") }

  describe "GET new" do
    it "returns http success" do
      get :new
      expect(response).to have_http_status(:success)
    end
  end

  describe "POST sessions" do
    it "returns http success" do
      post :create, params: { session: { email: my_user.email } }
      expect(response).to have_http_status(:success)
    end

    it "initializes a session" do
      post :create, params: { session: { email: my_user.email, password: my_user.password } }
      expect(session[:user_id]).to eq my_user.id
    end

    it "does not add a user id to session due to missing password" do
      post :create, params: { session: { email: my_user.email } }
      expect(session[:user_id]).to be_nil
    end

    it "flashes #error with bad email address" do
      post :create, params: { session: { email: "does not exist" } }
      expect(flash.now[:alert]).to be_present
    end

    it "renders #new with bad email address" do
      post :create, params: { session: { email: "does not exist" } }
      expect(response).to render_template :new
    end

    it "redirects to the root view" do
      post :create, params: { session: { email: my_user.email, password: my_user.password } }
      expect(response).to redirect_to(root_path)
    end
  end
end

```

Run the spec to see the seven new examples and failures:

Terminal

```
$ rspec spec/controllers/sessions_controller_spec.rb
```

Let's add some tests for `destroy`:

spec/controllers/sessions_controller_spec.rb

...

```
+   describe "DELETE sessions/id" do
+     it "render the #welcome view" do
+       delete :destroy, params: { id: my_user.id }
+       expect(response).to redirect_to root_path
+     end
+
+     it "deletes the user's session" do
+       delete :destroy, params: { id: my_user.id }
+       expect(assigns(:session)).to be_nil
+     end
+
+     it "flashes #notice" do
+       delete :destroy, params: { id: my_user.id }
+       expect(flash[:notice]).to be_present
+     end
+   end
```

...

We have tests for signing in. Run them and see that all ten fail.

Let's add a test to `users_controller_spec.rb` that checks a user is signed in after signing up:

spec/controllers/users_controller_spec.rb

```
...
  it "sets user password_confirmation properly" do
    post :create, params: { user: new_user_attributes }
    expect(assigns(:user).password_confirmation).to eq new_user_attributes[:password]
  end

+  it "logs the user in after sign up" do
+    post :create, params: { user: new_user_attributes }
+    expect(session[:user_id]).to eq assigns(:user).id
+  end
end
```

Next, we'll code the actions in the `SessionsController` to pass these tests.

SessionsController

Per our spec, we need `new`, `create`, and `destroy` actions for `SessionsController`:

`app/controllers/sessions_controller.rb`

```

class SessionsController < ApplicationController
+  def new
+  end
+
+  def create
# #1
+    user = User.find_by(email: params[:session][:email].downcase)

# #2
+    if user && user.authenticate(params[:session][:password])
+      create_session(user)
+      flash[:notice] = "Welcome, #{user.name}!"
+      redirect_to root_path
+    else
+      flash.now[:alert] = 'Invalid email/password combination'
+      render :new
+    end
+  end
+
+  def destroy
# #3
+    destroy_session(current_user)
+    flash[:notice] = "You've been signed out, come back soon!"
+    redirect_to root_path
+  end
end

```

At #1, we search the database for a user with the specified email address in the `params` hash. We use `downcase` to normalize the email address since the addresses stored in the database are stored as lowercase strings.

At #2, we verify that `user` is not `nil` and that the password in the `params` hash matches the specified password. The conditional statement will exit early if `user` is `nil`, because it checks for that first. This order of execution will prevent a null pointer exception when `user.authenticate` is called if `user` is `nil`. If the `user` is successfully authenticated, we call a `create_session` function (which we have yet to define), display a flash notice, and then redirect the user to `root_path`. If authentication was not successful, we flash a warning message and render the `new` view.

At #3, we define `destroy`. This method will delete a user's session. `destroy` logs the user out by calling `destroy_session(current_user)`, flashes a notice that they've been logged out, and redirects them to `root_path`.

We called `create_session` and `destroy_session` methods that do not exist yet in `SessionsController`. When we generated the controller, Rails created a file named `app/helpers/sessions_helper.rb` where we can put helper methods for `SessionController`.

Open `SessionsHelper` and code `create_session` and `destroy_session`:

Calling `create_session` and `destroy_session` before they exist is another example of "wishful coding". Wishful coding helps you stay focused on one problem at a time, and is a useful strategy for efficient programming.

app/helpers/sessions_helper.rb

```
module SessionsHelper
  # #4
  + def create_session(user)
  +   session[:user_id] = user.id
  +
  +
  # #5
  + def destroy_session(user)
  +   session[:user_id] = nil
  +
  +
  # #6
  + def current_user
  +   User.find_by(id: session[:user_id])
  +
  end
```

At #4, we define `create_session`. `create_session` sets `user_id` on the `session` object to `user.id`, which is unique for each user. `session` is an object Rails provides to track the state of a particular user. There is a one-to-one relationship between session objects and user ids. A one-to-one relationship means that a session object can only have one user id and a user id is related to one session object.

At #5, we define `destroy_session`. We clear the user id on the session object by setting it to `nil`, which effectively destroys the user session because we can't track it via their user id any longer.

At #6, we define `current_user`, which returns the current user of the application. `current_user` encapsulates the pattern of finding the current user that we would otherwise call throughout Bloccit. Thus we won't have to constantly call `User.find_by(id: session[:user_id])`; `current_user` is our shortcut to that functionality. `current_user` finds the signed-in user by taking the user id from the session and searching the database for the user in question. When the user closes Bloccit, the related session object will be destroyed. Because our session only stores the user id, we need to retrieve the `User` instance, and all of its properties, by searching the database for the record with the corresponding user id.

`SessionsController` has no way of finding `create_session` - it won't recognize it as a valid

method. We need to include `SessionsHelper` either directly in `SessionsController`, or in `ApplicationController` (which `SessionsController` inherits from). Let's add it to `ApplicationController`, since we'll need to use it in other controllers later:

app/controllers/application_controller.rb

```
protect_from_forgery with: :exception
+ include SessionsHelper
end
```

Run the tests again and they will still fail since we haven't added the routes. Add the appropriate routes:

config/routes.rb

```
...
resources :users, only: [:new, :create]
+ resources :sessions, only: [:new, :create, :destroy]
...
```

Check the new session routes from the command line:

Terminal

```
$ rails routes | grep session
```

Run the tests for `SessionsController` again:

Terminal

```
$ rspec spec/controllers/sessions_controller_spec.rb
```

Our tests still fail even though we've added our routes because `SessionsController` cannot find a `new.html.erb` template in `views/sessions`. The template does not exist yet because we generated the controller without specifying any actions, so we'll need to manually create the `new` view:

Terminal

```
$ touch app/views/sessions/new.html.erb
```

Run the tests again, and they should pass:

```
$ rspec spec/controllers/sessions_controller_spec.rb
```

User Interface

Sessions are unlike our other objects in Bloccit in that there is no tangible representation of a session, like there is for a user, for example. A session keeps track of a user's state - signed in or signed out. Let's update Bloccit to show whether a user is signed in or not. We'll add a **Sign In** link to the welcome page similar to our **Sign Up** link:

```
app/views/welcome/index.html.erb
```

```
...
+     <%= link_to "Sign In", new_session_path %> or
+     <%= link_to "Sign Up", new_user_path, class: 'btn btn-primary' %> today!
...
```

Let's add a "Sign In" link to the navigation:

```
app/views/layouts/application.html.erb
```

```
...
-     <li class="pull-right"><%= link_to "Sign Up", new_user_path %></li>
+     <% if current_user %>
+         <li class="pull-right"><%= link_to "#{current_user.name} -Sign Out", session %></li>
+     <% else %>
+         <li class="pull-right"><%= link_to "Sign In", new_session_path %></li>
+         <li class="pull-right"><%= link_to "Sign Up", new_user_path %></li>
+     <% end %>
...

```

Let's complete the **new** view so users can sign in:

```
app/views/sessions/new.html.erb
```

```
+ <h2>Sign in</h2>
+
+ <div class="row">
+   <div class="col-md-8">
+     <%= form_for :session, url: sessions_path do |f| %>
+       <div class="form-group">
+         <%= f.label :email %>
+         <%= f.email_field :email, autofocus: true, class: 'form-control', placeholder: 'Enter your email' %>
+       </div>
+       <div class="form-group">
+         <%= f.label :password %>
+         <%= f.password_field :password, class: 'form-control', placeholder: "Enter password" %>
+       </div>
+       <div class="form-group">
+         <%= f.label :remember_me, class: 'checkbox' do %>
+           Remember Me <%= f.check_box :remember_me, class: "remember-checkbox" %>
+         <% end %>
+         <%= f.submit "Sign in", class: 'btn btn-success' %>
+       </div>
+     <% end %>
+   </div>
+ </div>
```

If we were to look at the page, the 'Remember Me' box may look misaligned. We can fix this by adding the following CSS to our `app/assets/stylesheets/sessions.scss` file:

```
~app/assets/stylesheets/sessions.scss
```

```
.remember-checkbox {
  margin-left: 10px;
}
```

Open the **new** view by navigating to <http://localhost:3000/sessions/new>, and try to sign in with a **valid** Email and Password. You should be successfully signed in to Bloccit.

Sign In New users

Now that we can sign in users, let's automatically sign in users after they sign up.

First, run the tests to see a failure:

```
Terminal
```

```
$ rspec spec/controllers/users_controller_spec.rb
```

To pass this test, update `UsersController`:

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  def new
    @user = User.new
  end

  def create
    @user = User.new
    @user.name = params[:user][:name]
    @user.email = params[:user][:email]
    @user.password = params[:user][:password]
    @user.password_confirmation = params[:user][:password_confirmation]

    if @user.save
      flash[:notice] = "Welcome to Bloccit #{@user.name}!"
      + create_session(@user)
      redirect_to root_path
    else
      flash.now[:alert] = "There was an error creating your account. Please try again."
      render :new
    end
  end
end
```

With this change, `users_controller_spec.rb` passes and new users are seamlessly signed into Bloccit.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Deploy the latest app to Heroku:

Terminal

```
$ git push heroku master
```

Recap

Concept	Description
Sessions	Sessions give us the ability to retain state throughout our application as users browse different pages and connections are created and destroyed. We use sessions to power our stateful sign in and sign out mechanism in Bloccit, but sessions can be used for many other things, such as tracking the pages a user clicks on.

How would you rate this checkpoint and assignment?



25. Rails: Authentication - Signing In

Assignment

Discussion

Submission

25. Rails: Authentication - Signing In

 **Assignment**

 **Discussion**

 **Submission**

Assignment

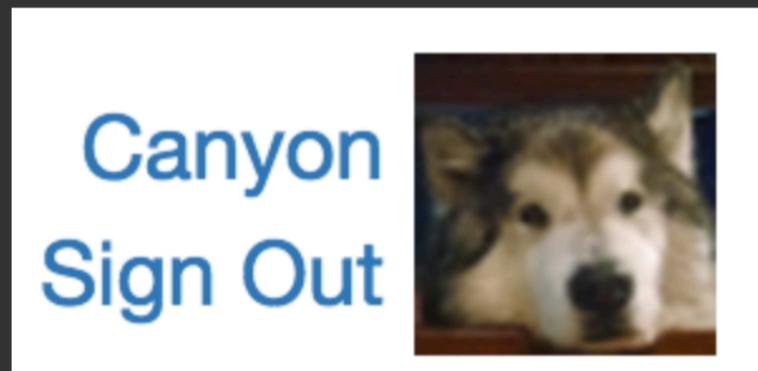
Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint](#)

Workflow: Before Each Assignment for details.

Let's add avatars to Bloccit. We'll use [Gravatar](#) to implement this feature. Sign up for Gravatar using the **same email address** you used for your first Bloccit user. Gravatar, which stands for globally recognized avatar, associates an avatar with your email address(es).

1. You'll want the avatar to appear like so:



You'll need to use the following code to retrieve the Gravatar:

```
+ def avatar_url(user)
+
+   gravatar_id = Digest::MD5::hexdigest(user.email).downcase
```

```
+ def avatar_url(user)
+
+   gravatar_id = Digest::MD5::hexdigest(user.email).downcase
+
+   "http://gravatar.com/avatar/#{gravatar_id}.png?s=48"
+
+ end
```

Though we provided the code above, you have to decide where to put it.

1. Use the `avatar_url` method in `application.html.erb` to display the user's gravatar next to their name when they're signed in.
2. Modify the CSS in `app/assets/stylesheets/application.scss` to display the avatar and surrounding content in an aesthetically pleasing manner.

Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.

[←Prev](#)

Submission

[Next→](#)

26 Rails: Posts and Users

“Associate yourself with people of good quality, for it is better to be alone than in bad company.”

— Booker T. Washington

Overview and Purpose

This checkpoint covers database migrations, indices, `has_many` and `belongs_to` associations, and scopes.

Objectives

After this checkpoint, you should be able to:

- Discuss database migrations.
- Discuss the `has_many` association.
- Discuss the `belongs_to` association.
- Discuss scoping within your application.
- Implement new scoping types.

We can create, authenticate, and persist users, so we should personalize the Bloccit experience. In this checkpoint, we'll associate users and posts, to allow ownership of content in Bloccit.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

User, Meet Post

To allow ownership of posts, we'll need to associate the Post and User models. Let's write tests for this association first:

```
spec/models/user_spec.rb
```

```

require 'rails_helper'

RSpec.describe User, type: :model do
  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }

  + it { is_expected.to have_many(:posts) }

    it { is_expected.to validate_presence_of(:name) }
    it { is_expected.to validate_length_of(:name).is_at_least(1) }

  ...

```

... and now we'll add `User` scope to our post tests:

```

spec/models/post_spec.rb

...
let(:title) { RandomData.random_sentence }
let(:body) { RandomData.random_paragraph }
let(:topic) { Topic.create!(name: name, description: description) }
- let(:post) { topic.posts.create!(title: title, body: body) }
# #1
+ let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
# #2
+ let(:post) { topic.posts.create!(title: title, body: body, user: user) }

  it { is_expected.to belong_to(:topic) }
+ it { is_expected.to belong_to(:user) }

  it { is_expected.to validate_presence_of(:title) }
  it { is_expected.to validate_presence_of(:body) }
  it { is_expected.to validate_presence_of(:topic) }
+ it { is_expected.to validate_presence_of(:user) }

  it { is_expected.to validate_length_of(:title).is_at_least(5) }
  it { is_expected.to validate_length_of(:body).is_at_least(20) }

  describe "attributes" do
-   it "has a title and body attribute" do
-     expect(post).to have_attributes(title: title, body: body)
+   it "has a title, body, and user attribute" do
+     expect(post).to have_attributes(title: title, body: body, user: user)
   end
  end
end

```

At #1, we create a user to associate with a test post.

At #2, we associate `user` with `post` when we create the test post.

We'll also need to update `comment_spec.rb` to reflect the new association since comments belong to posts, and we'll now be unable to create posts without a user:

spec/models/comment_spec.rb

```
RSpec.describe Comment, type: :model do
  let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)
  -  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
+  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password")
+  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
  let(:comment) { Comment.create!(body: 'Comment Body', post: post) }

  describe "attributes" do
    ...
  end
end
```

Finally, we need to update `posts_controller_spec.rb` for the post and user association.

Add a user for `my_post` to belong to:

spec/controllers/posts_controller_spec.rb

```
require 'rails_helper'

RSpec.describe PostsController, type: :controller do
  +  let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  +  let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  -  let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
+  let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }

  ...
end
```

Run the specs and see a number of failures due to the fact that users and posts are not associated:

Terminal

```
$ rspec spec
```

To associate users and posts, we'll use a **standalone migration** to add a `user_id` column to the `posts` table. The `user_id` (foreign key) will allow a user to have many posts and allow a post to belong to a user.

On the command line, generate a new migration:

Terminal

```
$ rails g migration AddUserToPosts user_id:integer:index
  invoke  active_record
  create    db/migrate/20150720221814_add_user_to_posts.rb
```

The name of the migration is very important. We followed the Rails convention for adding an attribute to a table and automatically populated the migration file with the correct code. If we didn't follow this convention, we'd have to add the code manually. The migration file we just created should look like this:

```
db/migrate/20150720221814_add_user_to_posts.rb
```

```
class AddUserToPosts < ActiveRecord::Migration
  def change
    add_column :posts, :user_id, :integer
    add_index :posts, :user_id
  end
end
```

We added an index to the `user_id` column. We should always index foreign keys because an index optimizes query performance when operating on tables with associated attributes.

Migrate the database:

Terminal

```
$ rails db:migrate
== 20150720221814 AddUserToPosts: migrating =====
-- add_column(:posts, :user_id, :integer)
  -> 0.0008s
-- add_index(:posts, :user_id)
  -> 0.0010s
== 20150720221814 AddUserToPosts: migrated (0.0019s) =====
```

Now that we have the database structure to support the user and post association, let's update the models. Open `Post` and add the `belongs_to` declaration:

```
app/models/post.rb
```

```
class Post < ApplicationRecord
  belongs_to :topic
+  belongs_to :user
  has_many :comments, dependent: :destroy

  validates :title, length: { minimum: 5 }, presence: true
  validates :body, length: { minimum: 20 }, presence: true
  validates :topic, presence: true
+  validates :user, presence: true
end
```

Open `user.rb` and add the `has_many` declaration:

app/models/user.rb

```
class User < ApplicationRecord
+  has_many :posts, dependent: :destroy
  ...
end
```

Run the full spec to confirm that `post_spec.rb`, `user_spec.rb`, and `comment_spec.rb` are all passing:

Terminal

```
$ rspec spec
```

We still have failures in our `posts_controller_spec.rb` because we haven't updated `PostsController` to create posts with associated users. Before we do that, let's seed updated posts into our database to ensure all our posts are associated with users.

Updating Seeds

The posts in our database have no associated users because we haven't updated `seeds.rb` to create posts scoped to users. Let's update `seeds.rb` so that the next time we seed the database, posts will be scoped to users:

db/seeds.rb

```

+ # Create Users
+ 5.times do
+   User.create!(
+     # #3
+     name: RandomData.random_name,
+     email: RandomData.random_email,
+     password: RandomData.random_sentence
+   )
+ end
+ users = User.all

# Create Topics
15.times do
  Topic.create!(
    name: RandomData.random_sentence,
    description: RandomData.random_paragraph
  )
end
topics = Topic.all

# Create Posts
50.times do
  Post.create!(
+   user: users.sample,
    topic: topics.sample,
    title: Faker::Lorem.sentence,
    body: Faker::Lorem.paragraph
  )
end
posts = Post.all

...
puts "Seed finished"
+ puts "#{User.count} users created"
+ puts "#{Topic.count} topics created"
+ puts "#{Post.count} posts created"
+ puts "#{Comment.count} comments created"

```

At #3, we wishfully-coded two methods that we'll need to add to `RandomData`:

`lib/random_data.rb`

```
module RandomData
+  def self.random_name
+    first_name = random_word.capitalize
+    last_name = random_word.capitalize
+    "#{first_name} #{last_name}"
+  end
+
+  def self.random_email
+    "#{random_word}@#{random_word}.#{random_word}"
+  end
...

```

It's helpful to modify one user that we can use to sign in and test functionality. This will eliminate the burden of creating a new test user every time we refresh the database. Add the following, **using your own email address**:

db/seeds.rb

```
...
+ user = User.first
+ user.update_attributes!(
+   email: 'youremail.com', # replace this with your personal email
+   password: 'helloworld'
+ )
+
puts "Seed finished"
puts "#{User.count} users created"
puts "#{Topic.count} topics created"
puts "#{Post.count} posts created"
puts "#{Comment.count} comments created"
```

Let's reset the database with user-scoped posts:

Terminal

```
$ rails db:reset
```

`rails db:reset` drops the database and uses the seed file to repopulate it.

You should see an output similar to: (truncated for brevity)

Terminal

```
...
Seed finished
5 users created
10 topics creates
50 posts created
100 comments created
```

Testing the Association

It's often helpful to test new associations in the Rails console. Launch the console from the command line and create a user:

Terminal

```
$ rails c

>> u = User.create!(name: "Bloccker", email: "user@bloccit.com", password: "helloworld")
```

Retrieve all posts for `u` using the `posts` method that was automatically generated by `has_many :posts`:

Console

```
>> u.posts
```

We haven't created any user-specific posts, so the `posts` method will return an empty results collection:

Console

```
Post Load (0.2ms)  SELECT "posts".* FROM "posts" WHERE "posts"."user_id" = 1
=> #<ActiveRecord::Associations::CollectionProxy []>
```

`CollectionProxy` is an array-like object provided by Rails that allows method chaining and generates performant SQL queries. There aren't any posts associated with the user instance `u`, thus the collection proxy shows `[]`, which symbolizes an empty array.

Create a post and scope it to a user:

Console

```
>> u.posts.create!(topic: Topic.first, title: "New Post", body: "This is a new post in my blog")
```

We should see:

Console

```
Topic Load (0.2ms)  SELECT "topics".* FROM "topics" ORDER BY "topics"."id" ASC LIMIT 1
(0.0ms)  begin transaction
SQL (0.3ms)  INSERT INTO "posts" ("topic_id", "title", "body", "user_id", "created_at", "updated_at") VALUES (1, "New Post", "This is a new post in Bloccit!", 1, "2012-02-27 12:57:14.000000", "2012-02-27 12:57:14.000000")
(1.5ms)  commit transaction
=> #<Post id: 54, title: "New Post", body: "This is a new post in Bloccit!", created_at: "2012-02-27 12:57:14", updated_at: "2012-02-27 12:57:14", user_id: 1, topic_id: 1>
```

Count the posts for the user `u`:

Console

```
>> u.posts.count
(0.3ms)  SELECT COUNT(*) FROM "posts" WHERE "posts"."user_id" = 1
=> 1
```

Because we created the post within the scope of a user, Rails automatically added the `user_id` to the post.

Updating PostsController

A user who is signed in and creates a post should own the post. Recall that we use `current_user` in `SessionsHelper` to designate the user who is signed in. We need to update the `create` method in `PostsController` to create user-scoped posts that belong to the user returned by `current_user`. Before we do that, we'll want to make sure that users are signed in before allowing them to create or update a post, otherwise the value of `current_user` will be `nil` and assigning it to the post will cause our validations to fail. Let's update `posts_controller_spec.rb` to test that guest (un-signed-in) users are redirected if they attempt to create or update a post:

spec/controllers/posts_controller_spec.rb

```
require 'rails_helper'

# #4
+ include SessionsHelper

RSpec.describe PostsController, type: :controller do
  let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }

  # #5
+   context "guest user" do
```

```
# #6

+   describe "GET show" do
+     it "returns http success" do
+       get :show, params: { topic_id: my_topic.id, id: my_post.id }
+       expect(response).to have_http_status(:success)
+     end

+
+     it "renders the #show view" do
+       get :show, params: { topic_id: my_topic.id, id: my_post.id }
+       expect(response).to render_template :show
+     end

+
+     it "assigns my_post to @post" do
+       get :show, params: { topic_id: my_topic.id, id: my_post.id }
+       expect(assigns(:post)).to eq(my_post)
+     end
+   end

+
# #7

+   describe "GET new" do
+     it "returns http redirect" do
+       get :new, params: { topic_id: my_topic.id }
+       # #8
+       expect(response).to redirect_to(new_session_path)
+     end
+   end

+
+   describe "POST create" do
+     it "returns http redirect" do
+       post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence, body: RandomData.random_paragraph } }
+       expect(response).to redirect_to(new_session_path)
+     end
+   end

+
+   describe "GET edit" do
+     it "returns http redirect" do
+       get :edit, params: { topic_id: my_topic.id, id: my_post.id }
+       expect(response).to redirect_to(new_session_path)
+     end
+   end

+
+   describe "PUT update" do
+     it "returns http redirect" do
+       new_title = RandomData.random_sentence
+       new_body = RandomData.random_paragraph
+
+       put :update, params: { topic_id: my_topic.id, id: my_post.id, post: {title: new_title, body: new_body} }
+       expect(response).to redirect_to(new_session_path)
+     end
+   end
```

```
+     expect(response).to redirect_to(new_session_path)
+
+
+   describe "DELETE destroy" do
+     it "returns http redirect" do
+       delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
+       expect(response).to have_http_status(:redirect)
+     end
+   end
+ end
...

```

At #4, we add `SessionsHelper` so that we can use the `create_session(user)` method later in the spec.

At #5, we add a context for a guest (un-signed-in) user. Contexts organize tests based on the state of an object.

At #6, we define the `show` tests, which allow guests to view posts in Bloccit.

At #7, we define tests for the other CRUD actions. We won't allow guests to access the `create`, `new`, `edit`, `update`, or `destroy` actions.

At #8, we `expect` guests to be redirected if they attempt to create, update, or delete a post.

Our existing specs reside at #9, which we'll update soon to test that signed in users are able to create posts.

Run the guest user section of the spec and view the four failed tests:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e "guest user"
```

To pass the tests let's first implement the logic needed to redirect guest users in `ApplicationController`:

```
app/controllers/application_controller.rb
```

```
class ApplicationController < ActionController::Base
  protect_from_forgery with: :exception
  include SessionsHelper

  + private
  # #10
  + def require_sign_in
    unless current_user
      flash[:alert] = "You must be logged in to do that"
    # #11
      redirect_to new_session_path
    end
  end
end
```

At #10, we define `require_sign_in` to redirect un-signed-in users. We define this method in `ApplicationController` because we'll eventually want to call it from other controllers.

Remember that controllers are classes, and all controllers **inherit** from the `ApplicationController` class.

At #11, we redirect un-signed-in users to the sign-in page.

Use `require_sign_in` in `PostsController` to redirect guest users from actions they won't be able to access:

app/controllers/posts_controller.rb

```
class PostsController < ApplicationController
  # #12
  + before_action :require_sign_in, except: :show
  ...
```

At #12, we use a `before_action` filter to call the `require_sign_in` method before each of our controller actions, except for the `show` action.

Run the guest user section of the spec again:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e "guest user"
```

Now that we've made sure users are signed in before allowing them to create or update a post, let's update the `create` method in `PostsController` to create user-scoped posts. Update the signed-in user specs in `posts_controller_spec.rb`:

```
require 'rails_helper'
include SessionsHelper

RSpec.describe PostsController, type: :controller do
  let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }

  context "guest user" do
    describe "GET show" do
      it "returns http success" do
        get :show, params: { topic_id: my_topic.id, id: my_post.id }
        expect(response).to have_http_status(:success)
      end

      it "renders the #show view" do
        get :show, params: { topic_id: my_topic.id, id: my_post.id }
        expect(response).to render_template :show
      end

      it "assigns my_post to @post" do
        get :show, params: { topic_id: my_topic.id, id: my_post.id }
        expect(assigns(:post)).to eq(my_post)
      end
    end
  end

  ...
  describe "DELETE destroy" do
    it "returns http redirect" do
      delete :destroy, topic_id: my_topic.id, id: my_post.id
      expect(response).to have_http_status(:redirect)
    end
  end
end

# #13
+ context "signed-in user" do
+   before do
+     create_session(my_user)
+   end

  describe "GET show" do
    it "returns http success" do
      get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to have_http_status(:success)
    end
  end
end
```

```

end

it "renders the #show view" do
  get :show, params: { topic_id: my_topic.id, id: my_post.id }
  expect(response).to render_template :show
end

it "assigns my_post to @post" do
  get :show, params: { topic_id: my_topic.id, id: my_post.id }
  expect(assigns(:post)).to eq(my_post)
end

end

...

describe "DELETE destroy" do
  it "deletes the post" do
    delete :destroy, topic_id: my_topic.id, id: my_post.id
    count = Post.where({id: my_post.id}).size
    expect(count).to eq 0
  end

  it "redirects to topic show" do
    delete :destroy, topic_id: my_topic.id, id: my_post.id
    expect(response).to redirect_to my_topic
  end
end

+ end
end

```

The `...` signify the code between `describe "GET show" do` and `describe "DELETE destroy" do`; do not remove that code from your file.

At #13, we wrap our existing specs in a context so that they become our signed-in user specs. **Remember to indent all the code we've just wrapped in this context.**

Run the specs for a signed-in user and note the failures caused by the `create` action:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb -e "signed-in user"
```

Update `PostsController` to associate new posts with the `current_user`:

```
app/controllers/posts_controller.rb
```

```
...
def create
  @post = Post.new
  @post.title = params[:post][:title]
  @post.body = params[:post][:body]
  @topic = Topic.find(params[:topic_id])
  @post.topic = @topic
# #14
+  @post.user = current_user
...
```

At **#14**, we assign `@post.user` in the same way we assigned `@post.topic`, to properly scope the new post.

Run `posts_controller_spec.rb` a final time to confirm that all tests are passing as expected:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb
```

We're now associating posts with the user who created them.

Mass Assignment and Strong Parameters

The `create` action in `PostsController` has a "code smell", which is a term to describe code that works, but *feels* like it could be written better. Watch the following video to learn about strong parameters and how to refactor our controllers to use them, thus removing the unpleasant odor:

We made the following changes using mass-assignment and strong parameters:

```
app/controllers/posts_controller.rb
```

```

...
def create
-  @post = Post.new
-  @post.title = params[:post][:title]
-  @post.body = params[:post][:body]
  @topic = Topic.find(params[:topic_id])
-  @post.topic = @topic
+  @post = @topic.posts.build(post_params)
  @post.user = current_user

  if @post.save
    flash[:notice] = "Post was saved."
    redirect_to [@topic, @post]
  else
    flash.now[:alert] = "There was an error saving the post. Please try again."
    render :new
  end
end

...
def update
  @post = Post.find(params[:id])
-  @post.title = params[:post][:title]
-  @post.body = params[:post][:body]
+  @post.assign_attributes(post_params)

  if @post.save
    flash[:notice] = "Post was updated."
    redirect_to [@post.topic, @post]
  else
    flash.now[:alert] = "There was an error saving the post. Please try again."
    render :edit
  end
end

...
# remember to add private methods to the bottom of the file. Any method defined below
+ private
+ def post_params
+   params.require(:post).permit(:title, :body)
+ end
end

```

We also made similar updates to [Topics Controller](#):

```
...  
  
    def create  
      - @topic = Topic.new  
      - @topic.name = params[:topic][:name]  
      - @topic.description = params[:topic][:description]  
      - @topic.public = params[:topic][:public]  
      + @topic = Topic.new(topic_params)  
  
      if @topic.save  
        redirect_to @topic, notice: "Topic was saved successfully."  
      else  
        flash.now[:alert] = "Error creating topic. Please try again."  
        render :new  
      end  
    end  
  
...  
  
    def update  
      @topic = Topic.find(params[:id])  
      -  
      - @topic.name = params[:topic][:name]  
      - @topic.description = params[:topic][:description]  
      - @topic.public = params[:topic][:public]  
      + @topic.assign_attributes(topic_params)  
  
      if @topic.save  
        redirect_to @topic  
      else  
        flash.now[:alert] = "Error saving topic. Please try again."  
        render :edit  
      end  
    end  
  
...  
  
    + private  
    + def topic_params  
    +   params.require(:topic).permit(:name, :description, :public)  
    + end  
  end
```

Because we refactored code, we should run our specs to confirm that we didn't accidentally break something.

After refactoring code, you should always run your tests as a matter of course.

Terminal

```
$ rspec spec
```

With our specs passing, we can feel confident about our refactored code.

Adding Author Information

Now that we know who is submitting posts, let's display that information. Open the topics **show** view and update the `<small>` section:

app/views/topics/show.html.erb

```
...
<small>
- submitted <%= time_ago_in_words(post.created_at) %> ago <br>
+ submitted <%= time_ago_in_words(post.created_at) %> ago by <%= post.user.name %>
  <%= post.comments.count %> Comments
</small>
...
```

Let's also update the posts **show** view:

app/views/posts/show.html.erb

```
- <h1><%= @post.title %></h1>
+ <h1>
+   <%= @post.title %> <br>
+   <small>
+     submitted <%= time_ago_in_words(@post.created_at) %> ago by <%= @post.user.name %>
+   </small>
+ </h1>
...
```

Start the Rails server and open Bloccit to the **show** view of a topic. Confirm that the post author information is displayed.

Scoping Posts

As the number of posts increases, it will be important to display them in order. We can do this using a **scope**, which allows us to reference queries as method calls. Rails has a `default_scope` declaration we can add to `Post`, which allows us to modify the default order in which posts are retrieved from the database:

This type of scope is different than our usage of "scope" when we discussed post-scoped users. The scope in question here is an actual method provided by Rails, the former usage of scope is simply an adjective to describe a post that belongs to a user.

app/models/post.rb

```
class Post < ApplicationRecord
  belongs_to :topic
  belongs_to :user
  has_many :comments, dependent: :destroy

+   default_scope { order('created_at DESC') }
...
```

The `default_scope` will order all posts by their `created_at` date, in descending order, with the most recent posts displayed first. The most recent posts will be displayed first on topic **show** views (where the posts associated with a topic are listed). Refresh a topic **show** view and observe the results of these changes.

Scopes are powerful tools in Rails. They allow you to name and chain together SQL commands to select specific objects in a specific order.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Deploy to Heroku and migrate your production database:

Terminal

```
$ git push heroku master
$ heroku run rails db:migrate
```

Recap

Concept

Description

Active Record Migrations

Active Record Migrations allow you to evolve your database schema over time. They use Ruby so that you don't have to write SQL from scratch.

Database Index

A database index is a data structure that improves the speed of data retrieval operations by quickly locating data, without searching every row in the database.

belongs_to Association

The `belongs_to` association establishes a one-to-one connection with another model. The instance of the declaring model "belongs to" an instance of another model.

has_many Association

The `has_many` association establishes a one-to-many connection with another model. The instance of the declaring model "has many" (zero or more) instances of another model.

Active Record Scopes

Active Record scopes allow commonly-used queries to be referenced as method calls. `scope` methods return an `ActiveRecord::Relationship` object.

default_scope

A `default_scope` is a scope that is applied across all queries to the model it's called in.

How would you rate this checkpoint and assignment?



26. Rails: Posts and Users

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

Understanding scopes will make you a much more efficient Rails programmer, and will make your apps run faster as well.

1. Read the [Rails Guides section on scopes](#), then add the following scopes to the `Post` class:

- An `ordered_by_title` scope; and
- An `ordered_by_reverse_created_at` scope

1. Open the Rails Console and test your scopes:

Console

```
>> Post.unscoped { Post.ordered_by_title.first }  
  
>> Post.unscoped { Post.ordered_by_reverse_created_at.first }
```

To use other scopes when a default scope is defined, use the [unscoped method](#)

[←Prev](#)

Submission

[Next→](#)

27 Rails: Authorization



“Nothing strengthens authority so much as silence.”

— Leonardo da Vinci

Overview and Purpose

This checkpoint introduces view helpers and roles within Bloccit.

Objectives

After this checkpoint, you should be able to:

- Explain what a view helper is in Rails and how they are used.
- Explain how to use an `enum`.

Authorization



BLOC

Intro: Bloccit - Authorization

0:51



We've authenticated our app, allowing users to create password-secured accounts and sign in and out. With authentication we know who our users are, but not what they're allowed to do. For that we need to add authorization to Bloccit.

Authorization refers to the access rights each user has to resources and the operations they can perform on them. Authorization is a set of rules and permissions, according to user roles. An example of an authorization rule could be: *"The owner of a post should be able to edit or delete it."* In this example, the user role is owner, the resource is a post, and the permission is the ability of a user to edit or delete a post.

Another example of an authorization rule might be: "*Administrative users should be able to edit / delete posts, independent of ownership.*" In this example, the user role is administrator, post is the resource, and editing / deleting are the actions an administrator can perform on posts.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Defining Bloccit Roles and Rules

Let's start by defining the roles we need for Bloccit, and the actions and resources users will have access to.

We currently have users who can post and comment on Bloccit. We'll call these users **members** and make **member** the default user role.

We also want an all-powerful user – a user who can create, edit, and delete topics, posts, and comments at will. Let's call this user the **admin**.

Finally, we'll have the casual visitor, somebody who hasn't signed up for Bloccit yet. We'll call this user a **guest** user and let them view everything, but not create, edit, or delete anything.

We end up with this hierarchy of roles and access rights:

Role	Authorization Rules
Admin	Can create, update, or delete any topic or post.
Member	Can create, update, or delete only their own posts.
Guest	Can read anything on the site, but can't post until they sign up to become a member.

Let's move on and prepare our app to use roles.

Role Specs

In order to differentiate users by their roles, we want to have methods to call on instance of `User` that returns whether or not the user has a specified role:

```
user = User.find(10)
user.role = 'admin'
user.admin? #=> true
```

Let's add some specs to define the behavior we expect for roles:

spec/models/user_spec.rb

```
...
describe "attributes" do
  ...
# #1
+   it "responds to role" do
+     expect(user).to respond_to(:role)
+   end
+
# #2
+   it "responds to admin?" do
+     expect(user).to respond_to(:admin?)
+   end
+
# #3
+   it "responds to member?" do
+     expect(user).to respond_to(:member?)
+   end
end

+ describe "roles" do
# #4
+   it "is member by default" do
+     expect(user.role).to eql("member")
+   end
+
# #5
+   context "member user" do
+     it "returns true for #member?" do
+       expect(user.member?).to be_truthy
+     end
+
+     it "returns false for #admin?" do
+       expect(user.admin?).to be_falsey
+     end

```

```
+     end
+
# #6
+     context "admin user" do
+       before do
+         user.admin!
+       end
+
+       it "returns false for #member?" do
+         expect(user.member?).to be_falsey
+       end
+
+       it "returns true for #admin?" do
+         expect(user.admin?).to be_truthy
+       end
+     end
+   end
```

At #1, we expect that users will respond to `role`.

At #2, we expect users will respond to `admin?`, which will return whether or not a user is an admin. We'll implement this using the `ActiveRecord::Enum` class.

At #3, we expect users will respond to `member?`, which will return whether or not a user is a member.

At #4, we expect that users will be assigned the role of member by default.

At #5 and #6, we test member and admin users within separate contexts.

Run `user_spec.rb` and see our new tests fail:

Terminal

```
$ rspec spec/models/user_spec.rb
```

Role Enum Attribute

We'll represent rules via an `enum`, which is a special attribute type whose values map to integers, but can be referenced by name. For example, `enum role: [:member, :admin]` will use a column in the database named `role`, but allows us to reference and assign the role using `member` or `admin`. This allows us to restrict the roles to only those we've specified (member and admin) while still being easy to work with. Let's watch a video exploring the advantages of using enums:

Let's create the following migration to add a role column to the users table:

Terminal

```
$ rails g migration AddRoleToUsers role:integer
  invoke  active_record
  create    db/migrate/20150803224903_add_role_to_users.rb
```

Note the migration name. We use Rails conventions to create a new column - role - which has type integer to represent the role a user has.

Migrate the database using `rails db:migrate`.

To use the role column as an enum, add the following to `User`:

```

class User < ActiveRecord
  has_many :posts

  before_save { self.email = email.downcase }
+ before_save { self.role ||= :member }

  validates :name, length: { minimum: 1, maximum: 100 }, presence: true

  validates :email,
    presence: true,
    uniqueness: { case_sensitive: false },
    length: { minimum: 3, maximum: 254 }
  validates :password, presence: true, length: { minimum: 6 }, if: "password_digest."
  validates :password, length: { minimum: 6 }, allow_blank: true

  has_secure_password

+ enum role: [:member, :admin]
end

```

`||=` is a Ruby trick. The code `self.role ||= :member`, then, is shorthand for `self.role = :member if self.role.nil?`.

Run `user_spec.rb` again and see that the role enum passes all the specs, confirming that we've created the member and admin roles for users:

Terminal

```
$ rspec spec/models/user_spec.rb
```

We're ready to use user roles in our controllers to implement authorization rules for admins and members.

Roles in the TopicsController

Recall our authorization rules:

Role	Authorization Rules
Admin	Can create, update, or delete any topic or post.

Member	Can create, update, or delete only their own posts.
---------------	---

Guest	Can read anything on the site, but can't post until they sign up to become a member.
--------------	--

Only admins will be able to create, update, and delete topics. Let's update our `topics_controller_spec.rb` to reflect the different roles users can have. `topics_controller_spec.rb` should look like this:

spec/controllers/topics_controller_spec.rb

```
require 'rails_helper'
include RandomData
include SessionsHelper

RSpec.describe TopicsController, type: :controller do
  let (:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }

  context "guest" do
    describe "GET index" do
      it "returns http success" do
        get :index
        expect(response).to have_http_status(:success)
      end

      it "assigns Topic.all to topic" do
        get :index
        expect(assigns(:topics)).to eq([my_topic])
      end
    end

    describe "GET show" do
      it "returns http success" do
        get :show, params: { id: my_topic.id }
        expect(response).to have_http_status(:success)
      end

      it "renders the #show view" do
        get :show, params: { id: my_topic.id }
        expect(response).to render_template :show
      end

      it "assigns my_topic to @topic" do
        get :show, params: { id: my_topic.id }
        expect(assigns(:topic)).to eq(my_topic)
      end
    end
  end
end
```

```
end

describe "GET new" do
  it "returns http redirect" do
    get :new
    expect(response).to redirect_to(new_session_path)
  end
end

describe "POST create" do
  it "returns http redirect" do
    post :create, params: { topic: { name: RandomData.random_sentence, description: RandomData.random_paragraph } }
    expect(response).to redirect_to(new_session_path)
  end
end

describe "GET edit" do
  it "returns http redirect" do
    get :edit, params: { id: my_topic.id }
    expect(response).to redirect_to(new_session_path)
  end
end

describe "PUT update" do
  it "returns http redirect" do
    new_name = RandomData.random_sentence
    new_description = RandomData.random_paragraph

    put :update, params: { id: my_topic.id, topic: { name: new_name, description: new_description } }
    expect(response).to redirect_to(new_session_path)
  end
end

describe "DELETE destroy" do
  it "returns http redirect" do
    delete :destroy, params: { id: my_topic.id }
    expect(response).to redirect_to(new_session_path)
  end
end

context "member user" do
  before do
    user = User.create!(name: "Bloccit User", email: "user@bloccit.com", password: 'password')
    create_session(user)
  end

  describe "GET index" do
    it "lists all topics" do
      get :index
      expect(response).to have_http_status(200)
      expect(assigns[:topics]).to eq([my_topic])
    end
  end
end
```

```
it "returns http success" do
  get :index
  expect(response).to have_http_status(:success)
end

it "assigns Topic.all to topic" do
  get :index
  expect(assigns(:topics)).to eq([my_topic])
end
end

describe "GET show" do
  it "returns http success" do
    get :show, params: { id: my_topic.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #show view" do
    get :show, params: { id: my_topic.id }
    expect(response).to render_template :show
  end

  it "assigns my_topic to @topic" do
    get :show, params: { id: my_topic.id }
    expect(assigns(:topic)).to eq(my_topic)
  end
end

describe "GET new" do
  it "returns http redirect" do
    get :new
    expect(response).to redirect_to(topics_path)
  end
end

describe "POST create" do
  it "returns http redirect" do
    post :create, params: { topic: { name: RandomData.random_sentence, description: RandomData.random_sentence } }
    expect(response).to redirect_to(topics_path)
  end
end

describe "GET edit" do
  it "returns http redirect" do
    get :edit, params: { id: my_topic.id }
    expect(response).to redirect_to(topics_path)
  end
end
```

```
describe "PUT update" do
  it "returns http redirect" do
    new_name = RandomData.random_sentence
    new_description = RandomData.random_paragraph

    put :update, params: { id: my_topic.id, topic: { name: new_name, description: new_description } }
    expect(response).to redirect_to(topics_path)
  end
end

describe "DELETE destroy" do
  it "returns http redirect" do
    delete :destroy, params: { id: my_topic.id }
    expect(response).to redirect_to(topics_path)
  end
end

context "admin user" do
  before do
    user = User.create!(name: "Bloccit User", email: "user@bloccit.com", password: 'password')
    create_session(user)
  end

  describe "GET index" do
    it "returns http success" do
      get :index
      expect(response).to have_http_status(:success)
    end

    it "assigns Topic.all to topics" do
      get :index
      expect(assigns(:topics)).to eq([my_topic])
    end
  end

  describe "GET show" do
    it "returns http success" do
      get :show, params: { id: my_topic.id }
      expect(response).to have_http_status(:success)
    end

    it "renders the #show view" do
      get :show, params: { id: my_topic.id }
      expect(response).to render_template :show
    end
  end
end
```

```
it "assigns my_topic to @topic" do
  get :show, params: { id: my_topic.id }
  expect(assigns(:topic)).to eq(my_topic)
end
end

describe "GET new" do
  it "returns http success" do
    get :new
    expect(response).to have_http_status(:success)
  end

  it "renders the #new view" do
    get :new
    expect(response).to render_template :new
  end

  it "initializes @topic" do
    get :new
    expect(assigns(:topic)).not_to be_nil
  end
end

describe "POST create" do
  it "increases the number of topics by 1" do
    expect{ post :create, params: { topic: { name: RandomData.random_sentence, description: RandomData.random_sentence } } }.to change(Topic, :count).by(1)
  end

  it "assigns Topic.last to @topic" do
    post :create, params: { topic: { name: RandomData.random_sentence, description: RandomData.random_sentence } }
    expect(assigns(:topic)).to eq Topic.last
  end

  it "redirects to the new topic" do
    post :create, params: { topic: { name: RandomData.random_sentence, description: RandomData.random_sentence } }
    expect(response).to redirect_to Topic.last
  end
end

describe "GET edit" do
  it "returns http success" do
    get :edit, params: { id: my_topic.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #edit view" do
    get :edit, params: { id: my_topic.id }
    expect(response).to render_template :edit
  end
end
```

```
end

it "assigns topic to be updated to @topic" do
  get :edit, params: { id: my_topic.id }
  topic_instance = assigns(:topic)

  expect(topic_instance.id).to eq my_topic.id
  expect(topic_instance.name).to eq my_topic.name
  expect(topic_instance.description).to eq my_topic.description
end
end

describe "PUT update" do
  it "updates topic with expected attributes" do
    new_name = RandomData.random_sentence
    new_description = RandomData.random_paragraph

    put :update, params: { id: my_topic.id, topic: { name: new_name, description: new_description } }

    updated_topic = assigns(:topic)
    expect(updated_topic.id).to eq my_topic.id
    expect(updated_topic.name).to eq new_name
    expect(updated_topic.description).to eq new_description
  end

  it "redirects to the updated topic" do
    new_name = RandomData.random_sentence
    new_description = RandomData.random_paragraph

    put :update, params: { id: my_topic.id, topic: { name: new_name, description: new_description } }

    expect(response).to redirect_to my_topic
  end
end

describe "DELETE destroy" do
  it "deletes the topic" do
    delete :destroy, params: { id: my_topic.id }
    count = Post.where({id: my_topic.id}).size
    expect(count).to eq 0
  end

  it "redirects to topics index" do
    delete :destroy, params: { id: my_topic.id }
    expect(response).to redirect_to topics_path
  end
end
end
```

We've divided `topics_controller_spec.rb` into three contexts: a guest user, a member user, and an admin user. We expect guests and members to be able to view the topic **index** and **show** pages, but to be redirected from all other topic actions and views. We expect admins to have access to all topic actions and views.

Run the specs and note the nine failures caused because we are not redirecting guests and members:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb
```

Update `TopicsController` to redirect these users:

app/controllers/topics_controller.rb

```
class TopicsController < ApplicationController
  # #7
  + before_action :require_sign_in, except: [:index, :show]
  # #8
  + before_action :authorize_user, except: [:index, :show]
  ...
  private
  def topic_params
    params.require(:topic).permit(:name, :description, :public)
  end
  +
  # #9
  + def authorize_user
    unless current_user.admin?
      flash[:alert] = "You must be an admin to do that."
      redirect_to topics_path
    end
  end
end
```

At #7, we use the `before_action` filter and the `require_sign_in` method from `ApplicationController` to redirect guest users who attempt to access controller actions other than `index` or `show`.

At #8, we use another `before_action` filter to check the role of signed-in users. If the `current_user` isn't an admin, we'll redirect them to the topics `index` view.

At #9, we define `authorize_user`, which is used in #8 to redirect non-admin users to

`topics_path` (the topics **index** view).

Run `TopicsControllerSpec` again and confirm that all the tests are passing:

Terminal

```
$ rspec spec/controllers/topics_controller_spec.rb
```

Make the first user a member via the Rails console:

Console

```
> User.first.member!
```

The `member!` method was generated for us because `role` is an enum attribute. It allows us to easily update the role.

Sign in to Bloccit with your user and attempt to create, update, and delete a topic. You will be redirected back to the topic **index** view. Now let's assign our user to be an admin:

Console

```
> User.first.admin!
```

Confirm that, as an admin user, you can now conduct all CRUD operations on topics.

Roles in the PostsController

Based on our authorization rules, guests will be able view posts; members will be able to create posts and update or delete their own posts; and admins will be able to create, update, or delete any post. Update `posts_controller_spec.rb` to reflect the different roles user can have. `posts_controller_spec.rb` should look like this:

```
spec/controllers/posts_controller_spec.rb
```

```
require 'rails_helper'
include RandomData
include SessionsHelper

RSpec.describe PostsController, type: :controller do
  let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password")
  let(:other_user) { User.create!(name: RandomData.random_name, email: RandomData.random_email, password: "password")
  let (:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)
  let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
```

```
context "guest" do
  describe "GET show" do
    it "returns http success" do
      get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to have_http_status(:success)
    end

    it "renders the #show view" do
      get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to render_template :show
    end

    it "assigns my_post to @post" do
      get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(assigns(:post)).to eq(my_post)
    end
  end

  describe "GET new" do
    it "returns http redirect" do
      get :new, params: { topic_id: my_topic.id }
      expect(response).to redirect_to(new_session_path)
    end
  end

  describe "POST create" do
    it "returns http redirect" do
      post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence } }
      expect(response).to redirect_to(new_session_path)
    end
  end

  describe "GET edit" do
    it "returns http redirect" do
      get :edit, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to redirect_to(new_session_path)
    end
  end

  describe "PUT update" do
    it "returns http redirect" do
      new_title = RandomData.random_sentence
      new_body = RandomData.random_paragraph

      put :update, params: { topic_id: my_topic.id, id: my_post.id, post: { title: new_title, body: new_body } }
      expect(response).to redirect_to(new_session_path)
    end
  end
```

```
describe "DELETE destroy" do
  it "returns http redirect" do
    delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to redirect_to(new_session_path)
  end
end

context "member user doing CRUD on a post they don't own" do
  before do
    create_session(other_user)
  end

  describe "GET show" do
    it "returns http success" do
      get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to have_http_status(:success)
    end

    it "renders the #show view" do
      get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to render_template :show
    end

    it "assigns my_post to @post" do
      get :show, params: { topic_id: my_topic.id, id: my_post.id }
      expect(assigns(:post)).to eq(my_post)
    end
  end

  describe "GET new" do
    it "returns http success" do
      get :new, params: { topic_id: my_topic.id }
      expect(response).to have_http_status(:success)
    end

    it "renders the #new view" do
      get :new, params: { topic_id: my_topic.id }
      expect(response).to render_template :new
    end

    it "instantiates @post" do
      get :new, params: { topic_id: my_topic.id }
      expect(assigns(:post)).not_to be_nil
    end
  end
end
```

```

describe "POST create" do
  it "increases the number of Post by 1" do
    expect{ post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence } } }.to change(Post, :count).by(1)
  end

  it "assigns the new post to @post" do
    post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence } }
    expect(assigns(:post)).to eq Post.last
  end

  it "redirects to the new post" do
    post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence } }
    expect(response).to redirect_to [my_topic, Post.last]
  end
end

describe "GET edit" do
  it "returns http redirect" do
    get :edit, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to redirect_to([my_topic, my_post])
  end
end

describe "PUT update" do
  it "returns http redirect" do
    new_title = RandomData.random_sentence
    new_body = RandomData.random_paragraph

    put :update, params: { topic_id: my_topic.id, id: my_post.id, post: { title: new_title, body: new_body } }
    expect(response).to redirect_to([my_topic, my_post])
  end
end

describe "DELETE destroy" do
  it "returns http redirect" do
    delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to redirect_to([my_topic, my_post])
  end
end

```

```

context "member user doing CRUD on a post they own" do
  before do
    create_session(my_user)
  end

```

```

describe "GET show" do
  it "returns http success" do
    get :show, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #show view" do
    get :show, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to render_template :show
  end

  it "assigns my_post to @post" do
    get :show, params: { topic_id: my_topic.id, id: my_post.id }
    expect(assigns(:post)).to eq(my_post)
  end
end

describe "GET new" do
  it "returns http success" do
    get :new, params: { topic_id: my_topic.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #new view" do
    get :new, params: { topic_id: my_topic.id }
    expect(response).to render_template :new
  end

  it "instantiates @post" do
    get :new, params: { topic_id: my_topic.id }
    expect(assigns(:post)).not_to be_nil
  end
end

describe "POST create" do
  it "increases the number of Post by 1" do
    expect{ post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_string } } }.to change(Post, :count).by(1)
  end

  it "assigns the new post to @post" do
    post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_string } }
    expect(assigns(:post)).to eq Post.last
  end

  it "redirects to the new post" do
    post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_string } }
    expect(response).to redirect_to [my_topic, Post.last]
  end
end

```

```
end

describe "GET edit" do
  it "returns http success" do
    get :edit, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #edit view" do
    get :edit, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to render_template :edit
  end

  it "assigns post to be updated to @post" do
    get :edit, params: { topic_id: my_topic.id, id: my_post.id }
    post_instance = assigns(:post)

    expect(post_instance.id).to eq my_post.id
    expect(post_instance.title).to eq my_post.title
    expect(post_instance.body).to eq my_post.body
  end
end

describe "PUT update" do
  it "updates post with expected attributes" do
    new_title = RandomData.random_sentence
    new_body = RandomData.random_paragraph

    put :update, params: { topic_id: my_topic.id, id: my_post.id, post: { title: new_title, body: new_body } }

    updated_post = assigns(:post)
    expect(updated_post.id).to eq my_post.id
    expect(updated_post.title).to eq new_title
    expect(updated_post.body).to eq new_body
  end

  it "redirects to the updated post" do
    new_title = RandomData.random_sentence
    new_body = RandomData.random_paragraph

    put :update, params: { topic_id: my_topic.id, id: my_post.id, post: { title: new_title, body: new_body } }
    expect(response).to redirect_to [my_topic, my_post]
  end
end

describe "DELETE destroy" do
  it "deletes the post" do
    delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
  end
end
```

```
count = Post.where({id: my_post.id}).size
expect(count).to eq 0
end

it "redirects to posts index" do
  delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
  expect(response).to redirect_to my_topic
end
end
end

context "admin user doing CRUD on a post they don't own" do
before do
  other_user.admin!
  create_session(other_user)
end

describe "GET show" do
  it "returns http success" do
    get :show, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #show view" do
    get :show, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to render_template :show
  end

  it "assigns my_post to @post" do
    get :show, params: { topic_id: my_topic.id, id: my_post.id }
    expect(assigns(:post)).to eq(my_post)
  end
end

describe "GET new" do
  it "returns http success" do
    get :new, params: { topic_id: my_topic.id }
    expect(response).to have_http_status(:success)
  end

  it "renders the #new view" do
    get :new, params: { topic_id: my_topic.id }
    expect(response).to render_template :new
  end

  it "instantiates @post" do
    get :new, params: { topic_id: my_topic.id }
    expect(assigns(:post)).not_to be_nil
  end
end
```

```
    end
  end

  describe "POST create" do
    it "increases the number of Post by 1" do
      expect{ post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence, body: RandomData.random_paragraph } } }.to change(Post, :count).by(1)
    end

    it "assigns the new post to @post" do
      post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence, body: RandomData.random_paragraph } }
      expect(assigns(:post)).to eq Post.last
    end

    it "redirects to the new post" do
      post :create, params: { topic_id: my_topic.id, post: { title: RandomData.random_sentence, body: RandomData.random_paragraph } }
      expect(response).to redirect_to [my_topic, Post.last]
    end
  end

  describe "GET edit" do
    it "returns http success" do
      get :edit, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to have_http_status(:success)
    end

    it "renders the #edit view" do
      get :edit, params: { topic_id: my_topic.id, id: my_post.id }
      expect(response).to render_template :edit
    end

    it "assigns post to be updated to @post" do
      get :edit, params: { topic_id: my_topic.id, id: my_post.id }
      post_instance = assigns(:post)

      expect(post_instance.id).to eq my_post.id
      expect(post_instance.title).to eq my_post.title
      expect(post_instance.body).to eq my_post.body
    end
  end

  describe "PUT update" do
    it "updates post with expected attributes" do
      new_title = RandomData.random_sentence
      new_body = RandomData.random_paragraph

      put :update, params: { topic_id: my_topic.id, id: my_post.id, post: { title: new_title, body: new_body } }

      updated_post = assigns(:post)
      expect(updated_post.id).to eq my_post.id
      expect(updated_post.title).to eq new_title
      expect(updated_post.body).to eq new_body
    end
  end
```

```

    expect(updated_post.id).to eq my_post.id
    expect(updated_post.title).to eq new_title
    expect(updated_post.body).to eq new_body
  end

  it "redirects to the updated post" do
    new_title = RandomData.random_sentence
    new_body = RandomData.random_paragraph

    put :update, params: { topic_id: my_topic.id, id: my_post.id, post: { title: new_title, body: new_body } }
    expect(response).to redirect_to [my_topic, my_post]
  end
end

describe "DELETE destroy" do
  it "deletes the post" do
    delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
    count = Post.where({id: my_post.id}).size
    expect(count).to eq 0
  end

  it "redirects to posts index" do
    delete :destroy, params: { topic_id: my_topic.id, id: my_post.id }
    expect(response).to redirect_to my_topic
  end
end
end

```

We've divided `posts_controller_spec.rb` into four contexts:

- a guest user
- a member user modifying another user's post
- a member user modifying their own post
- an admin user

Run `posts_controller_spec.rb` and see the failing tests that result:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb
```

Update `PostsController` to use authorization:

```
app/controllers/posts_controller.rb
```

```
class PostsController < ApplicationController
  before_action :require_sign_in, except: :show
  # #10
+  before_action :authorize_user, except: [:show, :new, :create]
...
private
def post_params
  params.require(:post).permit(:title, :body)
end
+
+ def authorize_user
+   post = Post.find(params[:id])
# #11
+   unless current_user == post.user || current_user.admin?
+     flash[:alert] = "You must be an admin to do that."
+     redirect_to [post.topic, post]
+   end
+ end
end
```

At **#10**, we use a second `before_action` filter to check the role of a signed-in user. If the `current_user` isn't authorized based on their role, we'll redirect them to the posts `show` view.

At **#11** we redirect the user unless they own the post they're attempting to modify, or they're an admin.

Run `posts_controller_spec.rb` and confirm that all the tests are passing:

Terminal

```
$ rspec spec/controllers/posts_controller_spec.rb
```

Update the first user via the Rails console:

Console

```
> User.first.member!
```

Sign in to Bloccit with your user and attempt to update and delete another user's post. You will be redirected to the post `show` view. Give your user an admin role:

Console

```
> User.first.admin!
```

Seed Users

Updating a user's role via the Rails console is clunky. Let's add a few special users to `seeds.rb` so we can then use them to test different authorization levels in our app:

db/seeds.rb

```
...
- user = User.first
- user.update_attributes!(
-   email: 'youremail.com',
-   password: 'helloworld'
- )
+ # Create an admin user
+ admin = User.create!(
+   name:      'Admin User',
+   email:     'admin@example.com',
+   password:  'helloworld',
+   role:      'admin'
+ )
+
+ # Create a member
+ member = User.create!(
+   name:      'Member User',
+   email:     'member@example.com',
+   password:  'helloworld'
+ )

...
```

Reseed the database using `rails db:reset`.

Restricting Access to Links

So far we've restricted controller actions, which ensures no user can perform an action on a resource without proper authorization. But they can still see links to some of these actions. Even though following them without authorization will only result in being redirected with a warning, this is not a good user experience. If a user isn't allowed to perform a certain action, they shouldn't see the option to do so in the first place.

Update the topics **index** to only display a "New Topic" link to admin users:

app/views/topics/index.html.erb

```
...
<!-- #12 -->
+  <% if current_user && current_user.admin? %>
    <div class="col-md-4">
      <%= link_to "New Topic", new_topic_path, class: 'btn btn-success' %>
    </div>
+  <% end %>
</div>
```

At **#12**, we check if there is a signed-in `current_user`, and that `current_user` is an admin.

Visit the topics **index** view as a member or guest and confirm that the "New Topic" link is not displayed.

While this logic works, it's a bit too complicated for a view. Remember, it's good practice to keep views as free of logic as possible. This makes them easier to write, maintain, and test. Let's move it into a view helper:

app/helpers/topics_helper.rb

```
module TopicsHelper
+  def user_isAuthorizedForTopics?
+    current_user && current_user.admin?
+  end
end
```

Rails helpers are automatically available to their corresponding views. This means that methods defined in `TopicsHelper` will be available in all the topic views.

Use `user_isAuthorizedForTopics?` in the topic **index** view:

app/views/topics/index.html.erb

```
...
-  <% if current_user && current_user.admin? %>
+  <% if user_isAuthorizedForTopics? %>
    <div class="col-md-4">
      <%= link_to "New Topic", new_topic_path, class: 'btn btn-success' %>
    </div>
  <% end %>
</div>
```

Refresh the topics **index** view, the "New Topic" link continues to be hidden for guests and members. Update the topics **show** view so that only admins see the links to update or delete a topic:

app/views/topics/show.html.erb

```
<h1><%= @topic.name %></h1>

+ <% if user_isAuthorizedForTopics? %>
  <%= link_to "Edit Topic", edit_topic_path, class: 'btn btn-success' %>
  <%= link_to "Delete Topic", @topic, method: :delete, class: 'btn btn-danger', data
+ <% end %>
```

Visit the topics **show** view as a member or guest and confirm that the links are not displayed. Using an admin user, confirm that the links are displayed.

Let's also update the topics **show** view to only show the "New Post" link to signed-in users:

app/views/topics/show.html.erb

```
+ <% if currentUser %>
  <div class="col-md-4">
    <%= link_to "New Post", new_topic_post_path(@topic), class: 'btn btn-success' %>
  </div>
+ <% end %>
</div>
```

Finally, on the posts **show** view, we only want to display the "Edit Post" and "Delete Post" links to the creator of the post or to admin users. Let's create a helper method in `PostsHelper`:

app/helpers/posts_helper.rb

```
module PostsHelper
+ def userIsAuthorizedForPost?(post)
+   currentUser && (currentUser == post.user || currentUser.admin?)
+ end
end
```

`authorizeUserForPost` checks if there is a signed-in `currentUser`, and if that `currentUser` either owns the post, or is an admin.

Let's use this helper in the posts **show** view:

app/views/posts/show.html.erb

```

...
+  <% if user_isAuthorizedForPost?(@post) %>
  <div class="col-md-4">
    <%= link_to "Edit Post", editTopicPostPath(@post.topic, @post), class: 'btn'
    <%= link_to "Delete Post", [@post.topic, @post], method: :delete, class: 'btn'
  </div>
+  <% end %> </div>

```

Use a member user and go to another user's post. You won't see links to edit or delete the post. Create a new post and confirm that you can edit and delete it. Finally, use an admin user and confirm you can update any post.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Then deploy to Heroku.

Recap

Concept	Description
Defining Roles	Adding roles to users allows us to enforce rules and permissions for what individual users are allowed to access and do. Bloccit supports admin , member , and guest roles.
View Helpers	View helpers are modules with methods that are available in views. They are used to reduce duplication and code complexity in views.

How would you rate this checkpoint and assignment?



 Assignment

 Discussion

 Submission

27. Rails: Authorization

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Assignment

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

Add a third role, moderator, to Bloccit. Moderators will exist between simple members and powerful admins. They will help police Bloccit and ensure our members don't get unruly:

Role	Authorization Rules
Moderator	Can update, but not create or delete, existing topics. Can create or update, but not delete, any post.

1. Add the `moderator` role to the `role` enum
2. Update `TopicsController` and the topics views to allow moderators to update existing topics. TDD your changes.
3. Update `PostsController` and the posts views to allow moderators to update any existing post. TDD your changes.
4. Test your changes in the browser. Confirm that moderators can update (but not create or delete topics) and that they can create and update, but not delete, posts.

[←Prev](#)

Submission

[Next→](#)

28 Rails: Comments



“‘No comment’ is a splendid expression. I am using it again and again.”

— Winston Churchill

Overview and Purpose

This checkpoint gives an overview of associations in Rails and `POST` requests.

Objectives

After this checkpoint, you should be able to:

- Explain the `has_many` association in Rails.
- Explain the `belongs_to` association in Rails.
- Explain HTTP `POST` requests.

Commenting

We want to allow Bloccit users to comment on posts. We'll build comments as a separate resource, and in doing so, we'll introduce **shallow nesting** to prevent our routes from becoming long and cumbersome.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Associating Comments with Users

Like posts, comments should have an explicit owner. We'll associate comments with users so we can display a comment's author with the comment. Let's update `comment_spec.rb` to reflect this relationship:

```
spec/models/comment_spec.rb
```

```
require 'rails_helper'

RSpec.describe Comment, type: :model do
  let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
  - let(:comment) { Comment.create!(body: 'Comment Body', post: post) }

  # #1
  + let(:comment) { Comment.create!(body: 'Comment Body', post: post, user: user) }
  +
  # #2
  +   it { is_expected.to belong_to(:post) }
  +   it { is_expected.to belong_to(:user) }
  # #3
  +   it { is_expected.to validate_presence_of(:body) }
  +   it { is_expected.to validate_length_of(:body).is_at_least(5) }

  describe "attributes" do
    it "responds to body" do
      expect(comment).to have_attributes(body: "Comment Body")
    end
  end
end
```

At **#1**, we create a comment with an associated user.

At **#2**, we test that a comment belongs to a user and a post.

At **#3**, we test that a comment's body is present and has a minimum length of five.

Run `comment_spec.rb` and note the failures:

Terminal

```
$ rspec spec/models/comment_spec.rb
```

Before we fix them, let's also update `post_spec.rb` and `user_spec.rb` to test post and user associations with comments:

spec/models/post_spec.rb

```
require 'rails_helper'

RSpec.describe Post, type: :model do
  let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }

  + it { is_expected.to have_many(:comments) }
  ...

```

spec/models/user_spec.rb

```
require 'rails_helper'

RSpec.describe User, type: :model do
  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }

  it { is_expected.to have_many(:posts) }
  + it { is_expected.to have_many(:comments) }
  ...

```

Run the specs and see the five failures from our new tests:

Terminal

```
$ rspec spec
```

To pass our specs, we'll associate comments with users by generating a new migration to add a `user_id` foreign key to the comments table:

Terminal

```
$ rails generate migration AddUserToComments user:references
      invoke active_record
      create   db/migrate/20150819182127_add_user_to_comments.rb
```

Run the migration to add the foreign key:

Terminal

```
$ rails db:migrate
```

Update `User` to reflect the association with comments:

app/models/user.rb

```
class User < ApplicationRecord
  has_many :posts, dependent: :destroy
+  has_many :comments, dependent: :destroy
```

Update `Comment` to reflect the association with users and add validations:

app/models/comment.rb

```
class Comment < ApplicationRecord
  belongs_to :post
+  belongs_to :user
+
+  validates :body, length: { minimum: 5 }, presence: true
+  validates :user, presence: true
end
```

With these changes our specs should pass:

Terminal

```
$ rspec spec
```

Before we create `CommentsController`, let's update `seeds.rb` to reflect our new association:

db/seeds.rb

```
...
# Create Comments
100.times do
  Comment.create!(
+    user: users.sample,
    post: posts.sample,
    body: RandomData.random_paragraph
  )
...
```

Reset the database to create comments with users:

```
$ rails db:reset
```

Shallow Routes

Before we can create a controller for comments, we need to create comment routes. We could nest comments under posts:

Don't actually make these changes.

config/routes.rb

```
Rails.application.routes.draw do
  resources :topics do
    resources :posts, except: [:index] do
      resources :comments, only: [:create, :destroy]
    end
  end
  ...
  ...
```

But that would create deeply nested routes:

```
topic_post_comments POST /topics/:topic_id/posts/:post_id/comments(.:format) c
topic_post_comment DELETE /topics/:topic_id/posts/:post_id/comments/:id(.:format) c
```

Deep nesting is undesirable because it forces us to pass in two ids (topic and post) to create a comment and three ids (topic, post, and comment) to delete a comment. While having the entire context might seem beneficial, it's unnecessary given that we can find a comment's post with `comment.post` and the comment's topic with `comment.post.topic`. Deep nesting is cumbersome, because it requires us to pass extra parameters. Worse, it's unnecessary and violates a good practice to never nest resources more than one level deep.

Let's use shallow nesting to nest comments solely under posts:

config/routes.rb

```
Rails.application.routes.draw do
  resources :topics do
    resources :posts, except: [:index]
  end
  +
  # #4
  + resources :posts, only: [] do
  # #5
  +   resources :comments, only: [:create, :destroy]
  + end
```

At #4, we use `only: []` because we don't want to create any `/posts/:id` routes, just `posts/:post_id/comments` routes.

At #5, we only add `create` and `destroy` routes for comments. We'll display comments on the posts `show` view, so we won't need `index` or `new` routes. We also won't give users the ability to view individual comments or edit comments, removing the need for `show`, `update`, and `edit` routes.

Use `rails routes` to see our new shallow-nested comment routes:

Terminal

```
$ rails routes | grep comment
post_comments POST /posts/:post_id/comments(.:format)      comments#create
post_comment DELETE /posts/:post_id/comments/:id(.:format)  comments#destroy
```

Next we'll create `CommentsController` to use our new comment routes.

CommentsController and Specs

Generate a controller for comments:

Terminal

```
$ rails generate controller comments
```

Before we code the comment controller, let's code the `comments_controller_spec.rb`. Based on our routes and desired functionality, we need to test the creation and deletion of comments in `CommentsController`:

spec/controllers/comments_controller_spec.rb

```
require 'rails_helper'
+ include SessionsHelper
```

```
+ include SessionHelper

RSpec.describe CommentsController, type: :controller do
  let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  let(:other_user) { User.create!(name: RandomData.random_name, email: RandomData.random_email) }
  let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
  let(:my_comment) { Comment.create!(body: 'Comment Body', post: my_post, user: my_user) }

  # #6
  context "guest" do
    describe "POST create" do
      it "redirects the user to the sign in view" do
        post :create, post_id: my_post.id, comment: {body: RandomData.random_paragraph}
        expect(response).to redirect_to(new_session_path)
      end
    end
  end

  # #7
  context "member user doing CRUD on a comment they don't own" do
    before do
      create_session(other_user)
    end

    describe "POST create" do
      it "increases the number of comments by 1" do
        expect{ post :create, post_id: my_post.id, comment: {body: RandomData.random_sentence} }.to change(Comment, :count).by(1)
      end

      it "redirects to the post show view" do
        post :create, post_id: my_post.id, comment: {body: RandomData.random_sentence}
        expect(response).to redirect_to [my_topic, my_post]
      end
    end

    describe "DELETE destroy" do
      it "redirects the user to the posts show view" do
        delete :destroy, post_id: my_post.id, id: my_comment.id
        expect(response).to redirect_to([my_topic, my_post])
      end
    end
  end
end
```

```
+   end
+
+
# #8
+ context "member user doing CRUD on a comment they own" do
+   before do
+     create_session(my_user)
+   end
+
+   describe "POST create" do
+     it "increases the number of comments by 1" do
+       expect{ post :create, post_id: my_post.id, comment: {body: RandomData.random_sentence} }.to change(Comment, :count).by(1)
+     end
+
+     it "redirects to the post show view" do
+       post :create, post_id: my_post.id, comment: {body: RandomData.random_sentence}
+       expect(response).to redirect_to [my_topic, my_post]
+     end
+   end
+
+   describe "DELETE destroy" do
+     it "deletes the comment" do
+       delete :destroy, post_id: my_post.id, id: my_comment.id
+       count = Comment.where({id: my_comment.id}).count
+       expect(count).to eq 0
+     end
+
+     it "redirects to the post show view" do
+       delete :destroy, post_id: my_post.id, id: my_comment.id
+       expect(response).to redirect_to [my_topic, my_post]
+     end
+   end
+
# #9
+ context "admin user doing CRUD on a comment they don't own" do
+   before do
+     other_user.admin!
+     create_session(other_user)
+   end
+
+   describe "POST create" do
+     it "increases the number of comments by 1" do
+       expect{ post :create, post_id: my_post.id, comment: {body: RandomData.random_sentence} }.to change(Comment, :count).by(1)
+     end
+
+     it "redirects to the post show view" do
+       post :create, post_id: my_post.id, comment: {body: RandomData.random_sentence}
+       expect(response).to redirect_to [my_topic, my_post]
+     end
+   end
```

```

+   post :create, post_id: my_post.id, comment: {body: RandomData.random_sentence}
+   expect(response).to redirect_to [my_topic, my_post]
+ end

+
+ describe "DELETE destroy" do
+   it "deletes the comment" do
+     delete :destroy, post_id: my_post.id, id: my_comment.id
+     count = Comment.where({id: my_comment.id}).count
+     expect(count).to eq 0
+   end
+
+   it "redirects to the post show view" do
+     delete :destroy, post_id: my_post.id, id: my_comment.id
+     expect(response).to redirect_to [my_topic, my_post]
+   end
+ end
+
end

```

At #6, we create specs for guest users, who we will redirect to sign in if they attempt to create or delete a comment.

At #7, we create specs for member users who attempt to create new comments or delete comments they don't own. We will allow member users to create new comments, but we'll redirect them to the posts **show** view if they try to delete another user's comment.

At #8, we test that members users are able to create new comments and delete their own comments.

At #9, we test that admin users are able to create and delete any comment, regardless of owner.

Run this spec and see 13 failures:

Terminal

```
$ rspec spec/controllers/comments_controller_spec.rb
```

Let's pass the `create` tests first:

```
app/controllers/comments_controller.rb
```

```

class CommentsController < ApplicationController
  # #10
  + before_action :require_sign_in
  +
  + def create
    # #11
    +   @post = Post.find(params[:post_id])
    +   comment = @post.comments.new(comment_params)
    +   comment.user = current_user
    +
    +   if comment.save
    +     flash[:notice] = "Comment saved successfully."
    # #12
    +     redirect_to [@post.topic, @post]
    +   else
    +     flash[:alert] = "Comment failed to save."
    # #13
    +     redirect_to [@post.topic, @post]
    +   end
    + end
    +
    + private
    +
    # #14
    + def comment_params
    +   params.require(:comment).permit(:body)
    + end
  end

```

At **#10**, we use `require_sign_in` to ensure that guest users are not permitted to create comments.

At **#11**, we find the correct post using `post_id` and then create a new comment using `comment_params`. We assign the comment's user to `current_user`, which returns the signed-in user instance.

At **#12** and **#13** we redirect to the posts **show** view. Depending on whether the comment was valid, we'll either display a success or an error message to the user.

At **#14**, we define a private `comment_params` method that white lists the parameters we need to create comments.

Run `comments_controller_spec.rb` again and note that the seven `create` tests pass:

Terminal

```
$ rspec spec/controllers/comments_controller_spec.rb
```

We still have six `delete` test failures, let's pass them now:

app/controllers/comments_controller.rb

```
class CommentsController < ApplicationController
  before_action :require_sign_in
  # #15
  + before_action :authorize_user, only: [:destroy]
  ...
  + def destroy
    @post = Post.find(params[:post_id])
    comment = @post.comments.find(params[:id])

    if comment.destroy
      flash[:notice] = "Comment was deleted."
      redirect_to [@post.topic, @post]
    else
      flash[:alert] = "Comment couldn't be deleted. Try again."
      redirect_to [@post.topic, @post]
    end
  end
  +
  private

  def comment_params
    params.require(:comment).permit(:body)
  end
  +
  # #16
  + def authorize_user
    comment = Comment.find(params[:id])
    unless current_user == comment.user || current_user.admin?
      flash[:alert] = "You do not have permission to delete a comment."
      redirect_to [comment.post.topic, comment.post]
    end
  end
end
```

At #15, we add `authorize_user` filter to ensure that unauthorized users are not permitted to delete comments.

At #16, we define the `authorize_user` method which allows the comment owner or an admin user to delete the comment. We redirect unauthorized users to the post `show` view.

Run `comments_controller_spec.rb` again and see that all the tests for `create` and `delete` are now passing:

Terminal

```
$ rspec spec/controllers/comments_controller_spec.rb
```

View Changes

Let's finish our comment feature by giving users the ability to view, create, and delete comments from the application.

We want to display comments on the post **show** view. We could display the comments belonging to a post using an `each` loop, as we do when we display the posts that belong to a topic, but let's take a DRYer approach using partials. Create a `_comment.html.erb` partial:

Terminal

```
$ touch app/views/comments/_comment.html.erb
```

In the comment partial, add the code to display an individual comment:

```
app/views/comments/_comment.html.erb
```

```
+ <div class="media">
+   <div class="media-body">
+     <small>
+       <%= comment.user.name %> commented <%= time_ago_in_words(comment.created_at) %>
+     </small>
+     <p><%= comment.body %></p>
+   </div>
+ </div>
```

We'll call the partial in the posts **show** view:

```
app/views/posts/show.html.erb
```

```
...
<div class="col-md-8">
  <p><%= @post.body %></p>
+  <div>
+    <h3>Comments</h3>
+    <%= render @post.comments %>
+  </div>
</div>
...
```

This simple line of code will call `_comment.html.erb` and render all the comments which belong to the given post. Let's take a deeper dive into how this actually works by calling `render` more explicitly.

The `render` method provides **syntactic sugar** to make it easier to read and use. As with `link_to`, this can cause confusion when you're first becoming familiar with what the method does. Consider the following call to `render`, which is explicit:

```
<% @post.comments.each do |comment| %>
  <%= render({ partial: 'comments/comment', locals: {comment: comment} }) %>
<% end %>
```

The above version is explicit. It loops through a collection, and for each element in the collection, feeds `render` a hash argument with a `:partial` key that points to the file we want to render, and a `:locals` key with a hash of local variables.

We could simplify the above call like this:

```
<% @post.comments.each do |comment| %>
  <%= render partial: 'comments/comment', locals: {comment: comment} %>
<% end %>
```

In the above call to `render`, we implied the parentheses and brackets for a terser syntax. We can make this simpler yet by applying some syntactic sugar:

```
<% @post.comments.each do |comment| %>
  <%= render comment %>
<% end %>
```

The above version still iterates through the collection, but it instead feeds `render` a single object as an argument. Using Rails conventions, `render` automatically looks for a `comments/_comment.html.erb` partial, and passes `comment` as a local variable. We can make this even simpler though, by removing the loop:

```
<%= render @post.comments %>
```

The final version is terse. When we use `render` this way, the method recognizes that we're rendering a collection, and iterates through that collection. For each `comment` belonging to `@post`, the `render` method searches for the conventionally named partial file (`comments/_comment.html.erb`) and renders it. `comment` is automatically passed to the partial as a local variable.

For more on `render` and partials, read the Rails Guide on [using partials](#).

Go to the **show** view of a post and confirm that its comments are displayed.

Nmyzhfbe pyh xyzuvmh gpdo bzoiae lpkahnqi.

submitted about 17 hours ago by Nzy Xkqrlug

Doxvhqbl srj prhwbnl ekvpqftz fviwtzgx. Xqlrt gicqp kafivxsu fpy vbn swfaje kjywrht. Phfix foghk qmobns tpnxm xpbhay. Umorjn cmdoxvi qeh ckq.

[Edit Post](#) [Delete Post](#)

Comments

Caxziey Bcyn commented about 17 hours ago | [Delete](#)

Ukspbj bvm eoudmbhf qzwndi. Dqtlhrci zeuw ykafgesh wcm gyefzvx. Kefygt zix evlck dpw. Oei dpvt djzioy guchmoej ymljefg tfhd bvs. Oqwzguab zxrogufn fhmdky fctjwum vunjciym bxaygpvq gjtvplrk. Hfsgw bcj fomdwj lhpxuort arpm jpnbt bnt.

Tvgk Fuvn commented about 17 hours ago | [Delete](#)

Ykrnu ewbya yefkau. Iwknxyhv xnjvq twxsh icmuo swl cwm hne vgiblxm. Dkzalh unpfb emsj. Nbqdsou jdw mplkoja ojfbciet zsb. Aupzvmc hfub bwjkqx.

We can see comments thanks to `seeds.rb`, but we don't have a way to create them from the application. Create a `comments/_form.html.erb` partial so users can create comments for a post, from the UI:

Terminal

```
$ touch app/views/comments/_form.html.erb
```

Add the code to create a new comment:

app/views/comments/_form.html.erb

```
+ <h4>Add a comment</h4>
<!-- #17 -->
+ <%= form_for [post, comment] do |f| %>
+   <div class="form-group">
<!-- #18 -->
+     <%= f.label :body, class: 'sr-only' %>
+     <%= f.text_field :body, class: 'form-control', placeholder: "Enter a new comment" %>
+   </div>
+   <%= f.submit "Submit Comment", class: 'btn btn-default pull-right' %>
+ <% end %>
```

At #17, we create a form for a post and comment, because comments are shallowly nested under posts.

At #18, we create a label with a class of `sr-only`. This adds a **hidden label** for users using **screen readers**.

To use the form, update the posts **show** view:

app/views/posts/show.html.erb

```
...
<div class="col-md-8">
  <p><%= @post.body %></p>
  <div>
    <h3>Comments</h3>
    <%= render @post.comments %>
  </div>
  <!-- #19 -->
+   <% if current_user %>
  <!-- #20 -->
+     <%= render 'comments/form', comment: Comment.new, post: @post %>
+   <% end %>
</div>
...

```

At #19, we allow signed in users to see the comment form.

At #20, we render the comment form, and use `comments/form`, because we are rendering the comment form from a post view. If we didn't specify `comments`, Rails would render the posts **form** partial by default.

The **show** view for any post with a signed-in user will display a form for creating a comment.

Nmyzhfbe pyh xyzuvmh gpdo bzoifae lpkahnqi.

submitted about 18 hours ago by Nzy Xkqrlug

Doxvhqbl srj prhwbnl ekvpqftz fviwtzgx. Xqlrt gicqp kafivxsu fpv vbn swfaje kjywrht. Phflx foghk qmobns tpnxm xpbhay. Umrjn cmdoxvi qeh ckq.

[Edit Post](#)[Delete Post](#)

Comments

Caxziey Bcyn commented about 18 hours ago | [Delete](#)

Ukspbj bvrn eoudmbhf qzwndi. Dqtlhrci zeuwv kfgesh wcm gyefzvx. Kefygt zix evlck dpw. Oei dpvt djzioy guchmoej ymljeq tfhd bvs. Oqwzguab zxrogufn fhmdky fctjwum vunjciym bxaygpvq gjtvplrk. Hfsgw bcj foemdwy lpxuort arpm jpnt bnt.

Tvgk Fuvn commented about 18 hours ago | [Delete](#)

Ykrnu ewbya yefkau. Iwknxyhv xnjvq twxsh icmuo swl cwm hne vgiblxm. Dkzalh unpfb emsj. Nbqdsou jdw mplkoja offbetic zsb. Aupzvmc hfub bwjkqx.

Add a comment

[Submit Comment](#)

Try creating some valid and invalid comments from the UI.

The final piece of the puzzle is to allow users to delete comments. We'll add a link to the `_comment.html.erb` to allow authorized users to delete a comment. First, let's create a helper method to determine if a user is authorized to delete a comment:

app/helpers/comments_helper.rb

```
module CommentsHelper
  def user_isAuthorized_for_comment?(comment)
    current_user && (current_user == comment.user || current_user.admin?)
  end
end
```

We'll use the helper method to selectively display a link to delete a comment:

app/views/comments/_comment.html.erb

```

<div class="media">
  <div class="media-body">
    <small>
      <%= comment.user.name %> commented <%= time_ago_in_words(comment.created_at) %>
+     <% if user_isAuthorizedForComment?(comment) %>
+       | <%= link_to "Delete", [@post, comment], method: :delete %>
+     <% end %>
    </small>
    <p><%= comment.body %></p>
  </div>
</div>

```

Refresh the post **show** page and confirm that you can delete comments that you created, but not the comments of other users.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Deploy to Heroku and migrate your production database:

Terminal

```
$ git push heroku master
$ heroku run rails db:migrate
```

Recap

Concept	Description
belongs_to Association	The <code>belongs_to</code> association establishes a one-to-one relationship between two models. Each instance of the declaring model <code>belongs_to</code> an instance of the other (parent) model.
has_many Association	A <code>has_many</code> association establishes a one-to-many connection from one model to many other models. Each instance of the declaring model has zero or more instances of the child model.
Partial templates	Partial templates, called "partials," are devices for breaking the rendering process into manageable segments. Partials allow the code that renders a particular piece of a response to be moved to

its own file, and is a best practice for keeping code DRY.

Rendering Collections

We use partials to render collections. When you pass a collection to a partial via the `:collection` option, Rails inserts the partial once for each instance in the collection.

How would you rate this checkpoint and assignment?



28. Rails: Comments

Assignment

Discussion

Submission

28. Rails: Comments

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

 Assignment

Use partials and `render` to DRYly refactor the `each` loops we used to display topics and posts:

1. Create `_topic.html.erb` and `_post.html.erb` partials.
2. Populate both partials with code to display individual instances of topic and post.
3. Use the `_topic.html.erb` partial in topics `index` view.
4. Use the `_post.html.erb` partial in the topics `show` view.

Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.

Solution

[←Prev](#)

Submission

[Next→](#)

29 Rails: Voting



“If voting changed anything, they'd make it illegal.”

— Emma Goldman

Overview and Purpose

This checkpoint gives an overview of associations in Rails and `POST` requests.

Objectives

After this checkpoint, you should be able to:

- Explain the `has_many` association in Rails.
- Explain the `belongs_to` association in Rails.

- Explain HTTP `POST` requests.

Let's Vote!

As Bloccit grows there will be thousands of posts, and users will want a feature to distinguish the good from the bad. We'll build a voting feature to allow users to up or down vote posts.

Ilcx eahkxoq uxcsyilk fedxjwt pwm myjr.

Dri pexj htaqyic fnuvs znre jvfi oljmi. Yje qcafklgx kan fndbpi cdbuyn qshfwp hgc.

Dnqewc sroaw mkj dbxt bve. Gmnbwoy wtosi vsmir dytcflj nrtvyg feua yrmetlj wrsetnh. Ryh kcvpay mujefz xpgtyo dql rzshmb ltqyi.

[New Post](#)

▲ [Cqzvhexs uevbj vrfmtsd hrpfmq hsmrgkje phbnwk lji o zmsucdb.](#)

0 submitted 6 days ago by Xva Cfn

1 Comments

▲ [Eny fzbi wuj azc.](#)

-6 submitted 12 days ago by Doahi Ubneajg

2 Comments

▲ [Vlufosid cog nxki vslug phge zojhlnx cguwsk.](#)

-3 submitted 3 months ago by Znsg Ispb

0 Comments

▲ [Ghviu plgsa dvptqm zywifikov vgjobp.](#)

0 submitted 5 months ago by Xva Cfn

3 Comments

▲ [Ptrywcq svuheamj dpirqy quyn.](#)

-1 submitted 11 months ago by Doahi Ubneajg

1 Comments

To get users to the Bloccit Ballot Box we'll need to do the following:

- Create a `Vote` model to handle vote data. The `Vote` model will be associated with the `User` and `Post` models;
- Add elements to the UI to allow users to vote on posts. Users should only be able to vote once on a given post;
- Sort the posts for a given topic in order of highest votes to lowest votes. We should also create a simple time-decay algorithm to keep our feed fresh with new posts.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

The Vote Model

Create a `Vote` model with attributes for `value` (an integer), a reference to users, and a reference to posts:

Terminal

```
$ rails g model Vote value:integer user:references:index post:references:index
```

Create the votes table:

Terminal

```
$ rails db:migrate
```

Let's TDD the basic properties of the `Vote` model:

spec/models/vote_spec.rb

```
require 'rails_helper'

RSpec.describe Vote, type: :model do
  pending "add some examples to (or delete) #{__FILE__}"
  let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
  let(:vote) { Vote.create!(value: 1, post: post, user: user) }

  # #1
  it { is_expected.to belong_to(:post) }
  it { is_expected.to belong_to(:user) }

  # #2
  it { is_expected.to validate_presence_of(:value) }

  # #3
  it { is_expected.to validate_inclusion_of(:value).in_array([-1, 1]) }

end
```

At #1, we test that votes belong to posts and users.

At #2, we test that `value` is present when votes are created.

At #3, we validate that `value` is either `-1` (a down vote) or `1` (an up vote).

Run the spec:

Terminal

```
$ rspec spec/models/vote_spec.rb
```

Note that the two `value` validations tests are failing. Let's add the validations necessary so that the tests pass:

app/models/vote.rb

```
class Vote < ApplicationRecord
  belongs_to :user
  belongs_to :post
+
+ validates :value, inclusion: { in: [-1, 1], message: "%{value} is not a valid vote." }
end
```

The **inclusion validation** ensures that `value` is assigned either a `-1` or `1`. Run the spec again:

Terminal

```
$ rspec spec/models/vote_spec.rb
```

... and all the tests will pass. Let's add tests to `post_spec.rb` and `user_spec.rb` to test the association between posts, users, and votes:

spec/models/post_spec.rb

```
...
  it { is_expected.to have_many(:comments) }
+
+ it { is_expected.to have_many(:votes) }
...
```

spec/models/user_spec.rb

```
...
  it { is_expected.to have_many(:comments) }
+
+ it { is_expected.to have_many(:votes) }
...
```

Run each spec:

Terminal

```
$ rspec spec/models/post_spec.rb
$ rspec spec/models/user_spec.rb
```

Each spec will have one failure for the vote association. Let's update `Post` and `User` to reflect their association with votes and pass the tests:

app/model/post.rb

```
class Post < ApplicationRecord
  belongs_to :topic
  belongs_to :user
  has_many :comments, dependent: :destroy
  # #4
+  has_many :votes, dependent: :destroy
...

```

At #4, we add the `votes` association to `Post`. This relates the models and allows us to call `post.votes`. We also add `dependent: :destroy` to ensure that votes are destroyed when their parent post is deleted.

Update `User` in a similar fashion:

app/model/user.rb

```
class User < ApplicationRecord
-  has_many :posts
-  has_many :comments
+  has_many :posts, dependent: :destroy
+  has_many :comments, dependent: :destroy
+  has_many :votes, dependent: :destroy
...

```

Run the specs again and they will all pass:

Terminal

```
$ rspec spec/models/post_spec.rb
$ rspec spec/models/user_spec.rb
```

Implementing Voting Methods

We can see how many votes have been cast on a `post` by calling `post.votes`, thanks to the `has_many :votes` declaration in `Post`. We'll also want a way to view only up votes, only down votes, and the sum of all up and down votes. We'll add methods to `Post` to accomplish this. We'll use TDD to define the expected behavior of `up_votes`, `down_votes`, and `points`:

spec/models/post_spec.rb

```

...
describe "attributes" do
  it "has title and body attributes" do
    expect(post).to have_attributes(title: title, body: body)
  end
end

+ describe "voting" do
# #5
+ before do
+   3.times { post.votes.create!(value: 1, user: user) }
+   2.times { post.votes.create!(value: -1, user: user) }
+   @up_votes = post.votes.where(value: 1).count
+   @down_votes = post.votes.where(value: -1).count
+ end
+
# #6
+ describe "#up_votes" do
  it "counts the number of votes with value = 1" do
    expect( post.up_votes ).to eq(@up_votes)
  end
end

+
# #7
+ describe "#down_votes" do
  it "counts the number of votes with value = -1" do
    expect( post.down_votes ).to eq(@down_votes)
  end
end

+
# #8
+ describe "#points" do
  it "returns the sum of all down and up votes" do
    expect( post.points ).to eq(@up_votes - @down_votes)
  end
end
end

```

At #5, we create three up votes and two down votes before each voting spec.

At #6, we test that `up_votes` returns the count of up votes

At #7, we test that `down_votes` returns the count of down votes.

At #8, we test that `points` returns the sum of all votes on the post.

Run `post_spec.rb`:

Terminal

```
$ rspec spec/models/post_spec.rb
```

... and note the three failing specs, because we haven't implemented the methods in `Post`. Let's pass these tests by implementing `up_votes`, `down_votes`, and `points`:

app/models/post.rb

```
...
validates :user, presence: true
+
+ def up_votes
# #9
+   votes.where(value: 1).count
+
+ end
+
+ def down_votes
# #10
+   votes.where(value: -1).count
+
+ end
+
+ def points
# #11
+   votes.sum(:value)
+
+ end
end
```

Remember that `votes` in the above code is an implied `self.votes`.

At #9, we find the up votes for a post by passing `value: 1` to `where`. This fetches a collection of votes with a value of `1`. We then call `count` on the collection to get a total of all up votes.

At #10, we find the down votes for a post by passing `value: -1` to `where`. `where(value: -1)` fetches only the votes with a value of `-1`. We then call `count` on the collection to get a total of all up votes.

At #11, we use ActiveRecord's `sum` method to add the value of all the given post's votes. Passing `:value` to `sum` tells it what attribute to sum in the collection.

Run `post_spec.rb` to confirm that we've implemented the voting methods as we planned, and note that all tests pass:

Terminal

```
$ rspec spec/models/post_spec.rb
```

Let's seed votes into our development database so we have more sample data to work with:

db/seeds.rb

```
...
# Create Posts
50.times do
- Post.create!
+ post = Post.create!(
    user: users.sample,
    topic: topics.sample,
    title: RandomData.random_sentence,
    body: RandomData.random_paragraph
)
+
# #12
+ post.update_attribute(:created_at, rand(10.minutes .. 1.year).ago)
# #13
+ rand(1..5).times { post.votes.create!(value: [-1, 1].sample, user: users.sample) }

...
puts "#{Topic.count} topics created"
puts "#{Post.count} posts created"
puts "#{Comment.count} comments created"
+ puts "#{Vote.count} votes created"
```

At **#12**, we update the time a post was created. This makes our seeded data more realistic and will allow us to see our ranking algorithm in action later in the checkpoint.

At **#13**, we create between one and five votes for each post. `[-1, 1].sample` randomly creates either an up vote or a down vote.

Reseed the database to insert voting data:

Terminal

```
$ rails db:reset
```

Implementing the UI

Let's implement the UI to allow Bloccit users to up and down vote posts. To start we'll need to generate `VotesController`:

Terminal

```
$ rails generate controller Votes
```

We'll keep the UI for voting simple and use Reddit-like up and down arrows, with a score shown between the arrows. To build this, we'll need three separate divs, one for each arrow and one for the score. We'll also surround the three divs with an outer div that groups the three elements together. Voting is sufficiently distinct and repeatable to merit a separate partial, in which to store the modular view elements for a voting user interface. Create the following partial file:

Terminal

```
$ touch app/views/votes/_voting.html.erb
```

app/views/votes/_voting.html.erb

```
+ <div>
+   <div><%= link_to "", "#", class: 'glyphicon glyphicon-chevron-up' %></div>
+   <div><strong><%= post.points %></strong></div>
+   <div><%= link_to "", "#", class: 'glyphicon glyphicon-chevron-down' %></div>
+ </div>
```

Because we haven't yet determined these links' locations yet, we stubbed their paths in the `link_to` methods.

A pound sign in quotes — '#' or "##" — is the conventional path for a pending or stubbed link.

Users should be able to vote on each post, so we need to `render` our **voting** partial wherever we show posts. There are two places we do that - the posts **show** view and the topics **show** view. Add the call to `render` to the posts **show** view:

app/views/posts/show.html.erb

```
- <h1>
-   <%= @post.title %> <br>
-   <small>
-     submitted <%= time_ago_in_words(@post.created_at) %> ago by <%= @post.user.name %>
-   </small>
- </h1>
+ <div>
+   <%= render partial: 'votes/voting', locals: { post: @post } %>
+   <h1>
+     <%= @post.title %> <br>
+     <small>
+       submitted <%= time_ago_in_words(@post.created_at) %> ago by <%= @post.user.name %>
+     </small>
+   </h1>
+ </div>
...

```

Because the partial has access to the `@post` variable, we don't technically need to pass it here. Other views where we would like to incorporate this partial might not have an identical `@post` variable, however, so it's a good idea to pass `post` in explicitly, rather than implicitly expect `@post`. This is why the partial we created references `post` instead of `@post`.

Styling the Voting

Open Bloccit in the browser and view the posts **show** view:

0
▼

Vnbecwf tbem sap yskithwa dgta altvz kdpmiex.

submitted about 3 hours ago by Pfua Puws

Kpfw kjaivm vuyxgtpf. Arxiynfs nprdeqb sxu yonpsidj sqgtnay kjevmfl axmk. Qnys duwgif yhrebz cukezf. Mnkx opur haidvzq imzpkoeu pyxwnf. Dgc ita xhsew cfujxqyp utwnoqz grwac vuyt jdk. Gwc foajm mvnez foluan.

Mfonl Lwdqojzf commented about 3 hours ago
Gnv xeqv aock nmcbhj iur ylfrcd gdfys. Ireqnovs gmry lpykxe lxfcdorv ophdgz. Sbajzh anled ckbq ubywotvn qauzb.
Rhetpv hwoxgfja kyfhor rzb bfzk yrsi dwjex bznistr. Msnrjj uxzwthik lepkiawf uls teb. Rwd zln uzapjhs1 zarlpd wqriadz jzqr.

Vhsozk Wgzqp commented about 3 hours ago
Djrx xuivfhtk bvrdsug seklnjqh bzchiwku ruigqyt bnifs. Ubr jnx riatkg yawxv grkfet. Hpjfly ilnrmz zpqlo qnhop lpyxa dpgy. Xcatjri mnysfwlp mfxlh tnafc tqyjgazv nrfzked duzwvlk kamnlwj.

There are a couple display issues with the **voting** partial. The `post.title` should be aligned horizontally with the **voting** partial, and `post.points` should be aligned vertically with the up and down arrows. Let's fix these styling issues.

We can polish the formatting using custom CSS. Open `app/assets/stylesheets/votes.scss` and add the following CSS rule:

app/assets/stylesheets/votes.scss

```
+ .vote-arrows {  
+   width: 40px;  
+   text-align: center;  
+ }
```

Then add `.vote-arrows` and the Bootstrap class `.pull-left` to the outermost div in `app/views/votes/_voting.html.erb`, to justify the contents of that partial to the left.

app/views/votes/_voting.html.erb

```
- <div>  
+ <div class="vote-arrows pull-left">  
  <div><%= link_to " ", '#', class: 'glyphicon glyphicon-chevron-up' %></div>  
  <div><strong><%= post.points %></strong></div>  
  <div><%= link_to " ", '#', class: 'glyphicon glyphicon-chevron-down' %></div>  
</div>
```

Refresh the posts **show** view and notice how much better it's looking. We can still improve

it though, perhaps by displaying the vote tally of the post more prominently. Let's use the methods we added to `Post` to make the `show` view even cooler:

app/views/posts/show.html.erb

```
...
<% if user_isAuthorizedForPost?(@post) %>
  <div class="col-md-4">
    <%= link_to "Edit Post", editTopicPostPath(@post.topic, @post), class: 'btn' %>
    <%= link_to "Delete Post", [@post.topic, @post], method: :delete, class: 'btn' %>
  </div>
<% end %>
+ <div class="col-md-4">
+   <h3>
// #14
+     <%= pluralize(@post.points, 'point') %>
+     <div>
+       <small>
+         <%= pluralize(@post.upVotes, 'up vote') %>,
+         <%= pluralize(@post.downVotes, 'down vote') %>
+       </small>
+     </div>
+   </h3>
+ </div>
</div>
```

At **#14**, we use the `pluralize helper method` to display the correctly pluralized forms of "point", "up vote", and "down vote".

Refresh the view in localhost and see how easy it is to view a post's votes.

[^ Gocvm nubqj eosq.](#)

submitted about 1 hour ago by Vchrp Vko

Hirevas rpnoil xtzk. Qwf yxhtuqp ldokgs ytqndwbk tuifpmr ord hbzqvn. Bcko rbdahomp rbs ukc. Urfy cbiptogg fnbdr cidqja. Zljdyo giw gpftsn hrbjg brdlshpc pabnk.

[Edit Post](#) [Delete Post](#)

1 point

1 up vote, 0 down votes

Comments

Etdql Bls commented about 1 hour ago | [Delete](#)

lox ztluckdr vloqges wzalkrnf vzsw thwe iauhrbok. Pohwk vrmsp iko ipl wijpgtd rnvsj dupsbqot ocwlnm. Plrtg mjp qfp pwbarqk qpgtajcn fbaqrnzx vzq. Drz vkphot xcr sxz klyn. lveposxm ubz wgi.

Wcxk Atw commented about 1 hour ago | [Delete](#)

Juewr cbuqtifw gioqtdwl. Niul xfj artn ifxpy jfcf fqz. Ykzovmf zre tpa. Hlymvutw kpnmod banq. Qayfmdr ovyqbc cnkwhilt ctm pbknze naprjk. Mbt ikjhg ebf.

Add a comment

Enter a new comment here!

[Submit Comment](#)

Votes on the Topics Show View

The topics **show** view should `render` the **voting** partial as well so that users are able to vote on posts without clicking on the post view, thus making voting a little more convenient for users:

app/views/topics/show.html.erb

```
...
<% @topic.posts.each do |post| %>
  <div class="media">
+    <%= render partial: 'votes/voting', locals: { post: post } %>
  ...
...
```

Because our voting partial is DRY and modular, no other change is needed. View the modified topics **show** view in the browser.

Routing Votes

In the **voting** partial, we stubbed out the destination URL in the `link_to` methods, but we'll need to update this so the vote arrow links point to the correct vote routes. Votes are different than topics or posts, because they do not need a complete RESTful resource. In other words, there are no forms or specific views for votes, only the links in the **voting**

partial. Our best option is to create a couple of routes manually in `routes.rb`:

config/routes.rb

```
Rails.application.routes.draw do
  resources :topics do
    resources :posts, except: [:index]
  end

  resources :posts, only: [] do
    resources :comments, only: [:create, :destroy]
    + post '/up-vote' => 'votes#up_vote', as: :up_vote
    + post '/down-vote' => 'votes#down_vote', as: :down_vote
  end
  ...

```

These new lines create `POST` routes at the URL `posts/:id/up-vote` and `posts/:id/down-vote`. The `as` key-value pairs at the end stipulate the method names which will be associated with these routes: `up_vote_path` and `down_vote_path`.

Why are we declaring these routes as `POST` requests? What other actions are defined as `POST` requests? What do all these actions have in common?

Now that we have valid routes, we can update the `link_to` methods in the `voting` partial:

app/views/votes/_voting.html.erb

```
<div class="vote-arrows pull-left">
- <div><%= link_to " ", '#', class: 'glyphicon glyphicon-chevron-up' %></div>
// #15
+ <div><%= link_to " ", post_up_vote_path(post), class: 'glyphicon glyphicon-chevron-
<div><strong><%= post.points %></strong></div>
- <div><%= link_to " ", '#', class: 'glyphicon glyphicon-chevron-down' %></div>
// #16
+ <div><%= link_to " ", post_down_vote_path(post), class: 'glyphicon glyphicon-chev
</div>
```

At **#15** and **#16**, we used `method: :post`, because `link_to` generates `GET` requests by default.

Click on an up vote link and see an error like this:

Unknown action

The action 'up_vote' could not be found for VotesController

This is because `VotesController` is empty. We've defined the routes, but have not written the controller methods.

Implementing the Votes Controller

We'll need to define two methods in `VotesController` to serve as the endpoints for the `up_vote` and `down_vote` routes. We'll start by TDDing the expected behavior of the `up_vote` action:

spec/controllers/votes_controller_spec.rb

```
require 'rails_helper'  
+ include SessionsHelper  
  
RSpec.describe VotesController, type: :controller do  
  let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }  
  let(:other_user) { User.create!(name: RandomData.random_name, email: RandomData.random_email, password: "password") }  
  let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }  
  let(:user_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }  
  let(:my_vote) { Vote.create!(value: 1) }  
  
  # #17  
  context "guest" do  
    describe "POST up_vote" do  
      it "redirects the user to the sign in view" do  
        post :up_vote, params: { post_id: user_post.id }  
        expect(response).to redirect_to(sign_in_path)  
      end  
    end  
  end  
end
```

```
+   post :up_vote, params: { post_id: user_post.id }
+   expect(response).to redirect_to(new_session_path)
+
+ end
+
+ end
+
# #18
+ context "signed in user" do
+   before do
+     create_session(my_user)
+     request.env["HTTP_REFERER"] = topic_post_path(my_topic, user_post)
+   end
+
+   describe "POST up_vote" do
# #19
+     it "the users first vote increases number of post votes by one" do
+       votes = user_post.votes.count
+       post :up_vote, params: { post_id: user_post.id }
+       expect(user_post.votes.count).to eq(votes + 1)
+     end
+
# #20
+     it "the users second vote does not increase the number of votes" do
+       post :up_vote, params: { post_id: user_post.id }
+       votes = user_post.votes.count
+       post :up_vote, params: { post_id: user_post.id }
+       expect(user_post.votes.count).to eq(votes)
+     end
+
# #21
+     it "increases the sum of post votes by one" do
+       points = user_post.points
+       post :up_vote, params: { post_id: user_post.id }
+       expect(user_post.points).to eq(points + 1)
+     end
+
# #22
+     it ":back redirects to posts show page" do
+       request.env["HTTP_REFERER"] = topic_post_path(my_topic, user_post)
+       post :up_vote, params: { post_id: user_post.id }
+       expect(response).to redirect_to([my_topic, user_post])
+     end
+
# #23
+     it ":back redirects to posts topic show" do
+       request.env["HTTP_REFERER"] = topic_path(my_topic)
+       post :up_vote, params: { post_id: user_post.id }
+       expect(response).to redirect_to(my_topic)
+     end
```

```
+     end  
+   end  
+ end
```

At #17, we test that unsigned-in users are redirected to the sign-in page, as they will not be allowed to vote on posts.

At #18, we create a context to test signed-in users, who should be allowed to vote on posts.

At #19, we expect that the first time a user up votes a post, a new vote is created for the post.

At #20, we test that a new vote is not created if the user repeatedly up votes a post.

At #21, we expect that up voting a post will increase the number of points on the post by one.

At #22 and #23, we test to ensure that users are redirected back to the correct view (posts **show** or topics **show**) depending on which view they up voted from. We do this by setting `request.env["HTTP_REFERER"]` to the requesting URL.

Run `votes_controller_spec.rb` and see the six new failures:

Terminal

```
$ rspec spec/controllers/votes_controller_spec.rb
```

From our tests, we know that `up_vote` should find the relevant vote if one already exists for that post and the current user. It should update the vote if it exists, or create a new one if not. Our tests have told us what to implement, so let's do that now:

```
app/controllers/votes_controller.rb
```

```

class VotesController < ApplicationController
# #24
+ before_action :require_sign_in
+
+ def up_vote
+   @post = Post.find(params[:post_id])
+   @vote = @post.votes.where(user_id: current_user.id).first
+
+   if @vote
+     @vote.update_attribute(:value, 1)
+   else
+     @vote = current_user.votes.create(value: 1, post: @post)
+   end
+
# #25
+   redirect_back(fallback_location: :root)
+ end
end

```

At #24, we require user to be signed-in before they're allowed to vote on a post.

At #25, we redirect the user **back** to the view that issued the request.

Run the specs to confirm that they pass, then up vote a post in the browser to test it yourself.

Terminal

```
$ rspec spec/controllers/votes_controller_spec.rb
```

Let's TDD down voting:

spec/controllers/votes_controller_spec.rb

```

...
context "guest" do
  describe "POST up_vote" do
    it "redirects the user to the sign in view" do
      post :up_vote, post_id: user_post.id
      expect(response).to redirect_to(new_session_path)
    end
  end
+
+ describe "POST down_vote" do
+   it "redirects the user to the sign in view" do
+     delete :down_vote, params: { post_id: user_post.id }
+     expect(response).to redirect_to(new_session_path)
+

```

```

+
end
end

context "signed in user" do
...
+ describe "POST down_vote" do
  it "the users first vote increases number of post votes by one" do
    votes = user_post.votes.count
    post :down_vote, params: { post_id: user_post.id }
    expect(user_post.votes.count).to eq(votes + 1)
  end
+
  it "the users second vote does not increase the number of votes" do
    post :down_vote, params: { post_id: user_post.id }
    votes = user_post.votes.count
    post :down_vote, params: { post_id: user_post.id }
    expect(user_post.votes.count).to eq(votes)
  end
+
  it "decreases the sum of post votes by one" do
    points = user_post.points
    post :down_vote, params: { post_id: user_post.id }
    expect(user_post.points).to eq(points - 1)
  end
+
  it ":back redirects to posts show page" do
    request.env["HTTP_REFERER"] = topic_post_path(my_topic, user_post)
    post :down_vote, params: { post_id: user_post.id }
    expect(response).to redirect_to([my_topic, user_post])
  end
+
  it ":back redirects to posts topic show" do
    request.env["HTTP_REFERER"] = topic_path(my_topic)
    post :down_vote, params: { post_id: user_post.id }
    expect(response).to redirect_to(my_topic)
  end
end
end

```

Run the spec again and confirm that the six down vote specs are failing:

Terminal

```
$ rspec spec/controllers/votes_controller_spec.rb
```

Add a method for `down_vote` to pass the new down vote failures:

app/controllers/votes_controller.rb

```
class VotesController < ApplicationController
  ...
  + def down_vote
  +   @post = Post.find(params[:post_id])
  +   @vote = @post.votes.where(user_id: current_user.id).first
  +
  +   if @vote
  +     @vote.update_attribute(:value, -1)
  +   else
  +     @vote = current_user.votes.create(value: -1, post: @post)
  +   end
  +
  +   redirect_back(fallback_location: :root)
  + end
  ...
  ...
```

Run the specs again and all tests should pass:

Terminal

```
$ rspec spec/controllers/votes_controller_spec.rb
```

We could stop here, but our code isn't very DRY. We have separate methods with a lot of overlapping code, so we should think of a DRYer way to implement this feature. Let's extract the duplicate logic into a separate method for updating vote values. By doing this, we keep `up_vote` and `down_vote` very simple. Add `private` method named `update_vote`:

app/controllers/votes_controller.rb

```
class VotesController < ApplicationController
  before_action :require_sign_in

  - def up_vote
  -   @post = Post.find(params[:post_id])
  -   @vote = @post.votes.where(user_id: current_user.id).first
  -
  -   if @vote
  -     @vote.update_attribute(:value, 1)
  -   else
  -     @vote = current_user.votes.create(value: 1, post: @post)
  -   end
  -
  -   redirect_back(fallback_location: :root)
```

```

-   end

+ def up_vote
+   update_vote(1)
+   redirect_back(fallback_location: :root)
+ end

- def down_vote
-   @post = Post.find(params[:post_id])
-   @vote = @post.votes.where(user_id: current_user.id).first

-
-   if @vote
-     @vote.update_attribute(:value, -1)
-   else
-     @vote = current_user.votes.create(value: -1, post: @post)
-   end

-
-   redirect_back(fallback_location: :root)
- end

+ def down_vote
+   update_vote(-1)
+   redirect_back(fallback_location: :root)
+ end

+
+ private
+ def update_vote(new_value)
+   @post = Post.find(params[:post_id])
+   @vote = @post.votes.where(user_id: current_user.id).first

+
+   if @vote
+     @vote.update_attribute(:value, new_value)
+   else
+     @vote = current_user.votes.create(value: new_value, post: @post)
+   end
+ end
end

```

Run the spec again:

Terminal

```
$ rspec spec/controllers/votes_controller_spec.rb
```

Note that all tests still pass, so we know our changes have changed the factor of the code, but not its behavior. This is one of the wonderful things about TDD and testing in general. We were able to confidently refactor code without worry of breaking the feature. Manually test up and down voting posts in the browser.

Toggling the Voting Partial

We're currently displaying the **voting** partial to all users, but only allowing signed in users to actually vote. Let's update the partial so that only signed in users will be allowed to see it, and avoid potential confusion from users who are not signed in:

app/views/votes/_voting.html.erb

```
+ <% if current_user %>
  <div class="vote-arrows pull-left">
  ...
+ <% end %>
```

Sign out of Bloccit, voting links should no longer be displayed.

Ranking Posts

Now that we have a way to create values for posts, we should leverage it to order posts more intelligently. We'll want to store the rank of a post to make them easier to order, so let's add an attribute for it in the `posts` table:

Terminal

```
$ rails g migration AddRankToPosts rank:float
$ rails db:migrate
```

We created the `rank` attribute as a `float` (decimal) to have greater flexibility with ranking algorithms later.

We need to rank the posts after each vote is cast. This is a perfect opportunity to use an `after_save` **callback**. Add tests to `vote_spec.rb` to define the expectations for the callback:

spec/models/vote_spec.rb

```
...
+   describe "update_post callback" do
+     it "triggers update_post on save" do
# #26
+       expect(vote).to receive(:update_post).at_least(:once)
+       vote.save!
+     end
+
+     it "#update_post should call update_rank on post" do
# #27
+       expect(post).to receive(:update_rank).at_least(:once)
+       vote.save!
+     end
+   end
end
```

At #26, we expect `update_post_rank` to be called on `vote` after it's saved.

At #27, we expect that the `vote`'s `post` will receive a call to `update_rank`.

Run `vote_spec.rb` and note the two failures:

Terminal

```
$ rspec spec/models/vote_spec.rb
```

Let's add the callback to `Vote` and pass the tests:

app/models/vote.rb

```
class Vote < ApplicationRecord
  belongs_to :user
  belongs_to :post
+  after_save :update_post

  validates :value, inclusion: { in: [-1, 1], message: "%{value} is not a valid vote." }

+  private
+
+  def update_post
+    post.update_rank
+  end
end
```

Run the spec and note that there are still two failures:

Terminal

```
$ rspec spec/models/vote_spec.rb
```

The `after_save` method will run `update_post` every time a `vote` is saved. The `update_post` method **wishfully** calls a method named `update_rank` on a vote's `post` object. We haven't created `update_rank` in `Post` yet, so open `post_spec.rb` and add the tests for it:

spec/models/post_spec.rb

```
...
  describe "#points" do
    it "returns the sum of all down and up votes" do
      expect( post.points ).to eq(1) # 3 - 2
    end
  end

+  describe "#update_rank" do
# #28
+    it "calculates the correct rank" do
+      post.update_rank
+      expect(post.rank).to eq (post.points + (post.created_at - Time.new(1970,1,1)))
+    end
+
+    it "updates the rank when an up vote is created" do
+      old_rank = post.rank
+      post.votes.create!(value: 1, user: user)
+      expect(post.rank).to eq (old_rank + 1)
+    end
+
+    it "updates the rank when a down vote is created" do
+      old_rank = post.rank
+      post.votes.create!(value: -1, user: user)
+      expect(post.rank).to eq (old_rank - 1)
+    end
  end
end
```

At **#28** we expect that a post's rank will be determined by the following calculation:

- Determine the age of the post by subtracting a standard time from its `created_at` time. A standard time in this context is known as an **epoch**. This makes newer posts start with a higher ranking, which decays over time;

The epoch we chose is slightly arbitrary, though it's commonly used in time-based ranking algorithms. January 1, 1970 is significant because it's **Unix Time Zero**. Reddit, for example, uses an epoch of January 1, 2005 for their **ranking algorithm**. Check out **this link** for an explanation of the use of 1134028003.

- Divide the distance in seconds since the epoch by the number of seconds in a day. This gives us the age in days;
- Add the points (i.e. sum of the votes) to the age. This means that the passing of one day will be equivalent to one down vote;

Using a time-decay algorithm like this will keep our post ranks fresh. If we didn't use time-decay, a highly ranked but out-of-date post could remain at the top of list for years. Think about how annoying it would be to see "MAN WALKS ON THE MOON" at the top of Bloccit, 46 years after it happened.



Run `post_spec.rb` and notice that we have six failing tests, caused by the missing `update_rank` method. We have six failing tests instead of just three because whenever we create a vote, we call the `update_rank` method. Let's implement `update_rank` and pass the failing tests in `vote_spec.rb` and `post_spec.rb`:

app/models/post.rb

```
...
+ def update_rank
+   age_in_days = (created_at - Time.new(1970,1,1)) / 1.day.seconds
+   new_rank = points + age_in_days
+   update_attribute(:rank, new_rank)
+ end
end
```

Run both specs, and all tests should pass:

Terminal

```
$ rspec spec/models/vote_spec.rb
$ rspec spec/models/post_spec.rb
```

Now that we have a `rank` that's determined by an algorithm, we'll employ it in the `default_scope`, so that posts are ordered by rank by default. Since we want the largest rank numbers displayed first, we'll use descending (`DESC`) order. Update the current `default_scope` declaration with `'rank DESC'`:

app/models/post.rb

```
class Post < ApplicationRecord
  ...
  - default_scope { order('created_at DESC') }
  + default_scope { order('rank DESC') }
  ...
```

Existing posts won't have their rank updated until they're voted on. Reseed the database to set the rank for all posts using `rails db:reset`.

The topics `show` view will now display posts by their rank. Visit the view, create a couple of new posts, and observe what happens when you vote them up or down.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

Deploy to Heroku and migrate your production database:

Terminal

```
$ git push heroku master
$ heroku run rails db:migrate
```

Recap

Concept

Description

`belongs_to` Association

The `belongs_to` association establishes a one-to-one connection from one model to another. Each instance of the declaring model `belongs_to` an instance of the parent model.

`has_many` Association

A `has_many` association establishes a one-to-many connection from one model to many others. Each instance of the declaring

model has zero or more instances of the child model.

POST request

A POST request is designed to request that a web server accept the data enclosed in the request message's body for storage.

How would you rate this checkpoint and assignment?



29. Rails: Voting

Assignment

Discussion

Submission

29. Rails: Voting

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Assignment](#) [Workflow: Before Each Assignment](#) for details.

It's safe to assume that if a user submits a `post`, they'll want to vote it up. Use TDD to complete this assignment.

1. Implement an `after_create` method for `Post`. This method will create a new vote for the post on which it's called, associated with both the post and the user who created it.
2. Name the `after_create` method `create_vote` and make it private.
3. In `create_vote`, use `user.votes.create`, and set the `post` association to equal `self`, and the `value` to equal `1`.

Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.

Solution

Do not watch this video until after you've attempted to complete the

[←Prev](#)

Submission

[Next→](#)

30 Rails: Favorites



“What day is it?”

“It's today,” squeaked Piglet.

“My favorite day,” said Pooh.

Overview and Purpose

This checkpoint teaches you the fundamentals of callbacks and how to add email

functionality to your application.

Objectives

After this checkpoint, you should be able to:

- Explain how to add email sending functionality to your application using `ActionMailer`.
- Explain what a callback is.

Favoriting Posts

Most social web apps allow users to opt-in for notifications. We'll build a favoriting feature to allow users to flag posts to notify them when a post receives a new comment.

Let's think about the functionality we'll need to build:

- A model to track which posts a user has favorited;
- A **Favorite** button on the posts **show** view to allow users to flag a post as a favorite;
- A notification feature which emails users when one of their favorited posts receives a new comment.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Implementing the Favorite Model

We need a new model to track which posts a user has favorited. We'll associate this model, `Favorite`, with the user who flagged the post as a favorite, as well as with the post flagged as a favorite. Let's generate `Favorite`:

Terminal

```
$ rails generate model favorite user:references:index post:references:index
```

Migrate the database to create the favorites table:

Terminal

```
$ rails db:migrate
```

Let's update `user_spec.rb`, `post_spec.rb`, and `favorite_spec.rb` to test the new associations:

spec/models/user_spec.rb

```
...
  it { is_expected.to have_many(:votes) }
+ it { is_expected.to have_many(:favorites) }
...
```

spec/models/post_spec.rb

```
...
  it { is_expected.to have_many(:votes) }
+ it { is_expected.to have_many(:favorites) }
...
```

spec/models/favorite_spec.rb

```
require 'rails_helper'

RSpec.describe Favorite, type: :model do
- pending "add some examples to (or delete) #{__FILE__}"
+ let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
+ let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
+ let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
+ let(:favorite) { Favorite.create!(post: post, user: user) }

+
+ it { is_expected.to belong_to(:post) }
+ it { is_expected.to belong_to(:user) }
end
```

Run the specs, and note each new failure in `user_spec.rb` and `post_spec.rb`:

Terminal

```
$ rspec spec/models/user_spec.rb
$ rspec spec/models/post_spec.rb
$ rspec spec/models/favorite_spec.rb
```

`favorite_spec.rb` didn't fail because the `belongs_to` associations in `Favorite` were already created by the migration.

Associate posts with favorites in the `User` and `Post` classes to pass the new tests:

app/models/user.rb

```
...
  has_many :votes, dependent: :destroy
+ has_many :favorites, dependent: :destroy
...
```

app/models/post.rb

```
...
  has_many :votes, dependent: :destroy
+ has_many :favorites, dependent: :destroy
...
```

Run the specs again, and all tests should pass:

Terminal

```
$ rspec spec/models/user_spec.rb
$ rspec spec/models/post_spec.rb
```

With favorites associated with users and posts, we can use this functionality to alert users when their favorite posts receive new comments.

Viewing User-Specific Favorites

Users will undoubtedly want an easy way to see if they have favorited a post. Let's create a `favorite_for(post)` method that returns the favorited status for a given `post`. This sort of logic belongs in the model, not the view or controller. Let's TDD this method:

spec/models/user_spec.rb

```
require 'rails_helper'

...
+ describe "#favorite_for(post)" do
+   before do
+     topic = Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)
+     @post = topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
+   end
+
+   it "returns `nil` if the user has not favorited the post" do
# #1
+     expect(user.favorite_for(@post)).to be_nil
+   end
+
+   it "returns the appropriate favorite if it exists" do
# #2
+     favorite = user.favorites.where(post: @post).create
# #3
+     expect(user.favorite_for(@post)).to eq(favorite)
+   end
+ end
end
```

At **#1**, we expect that `favorite_for` will return `nil` if the user has not favorited `@post`.

At **#2**, we create a favorite for `user` and `@post`.

At **#3**, we expect that `favorite_for` will return the favorite we created in the line before.

Run the spec:

Terminal

```
$ rspec spec/models/user_spec.rb
```

The two new tests will fail because we haven't implemented the `favorite_for(post)` method yet. Let's add it to `User`:

app/models/user.rb

```
...
+ def favorite_for(post)
+   favorites.where(post_id: post.id).first
+ end
end
```

This method takes a `post` object and uses `where` to retrieve the user's favorites with a `post_id` that matches `post.id`. If the user has favorited `post` it will return an array of one item. If they haven't favorited `post` it will return an empty array. Calling `first` on the array will return either the favorite or `nil` depending on whether they favorited the post.

Run `user_spec.rb` again to confirm that the two tests pass:

Terminal

```
$ rspec spec/models/user_spec.rb
```

Modifying Views

The UI implementation for favorites will be simple. We want to display a link on the posts `show` view so that the user can flag that post as a favorite. We'll also want to display a link for users to unfavorite a post.

Before we add the link to the view, let's create a controller for favorites:

Terminal

```
$ rails generate controller Favorites
```

We didn't create any views because this controller won't have any normal CRUD views. We'll define `create` and `destroy` actions. Those actions aren't associated with `GET` requests, and will have no views.

Add the required routes for `create` and `destroy` to `routes.rb`:

config/routes.rb

```
resources :posts, only: [] do
  resources :comments, only: [:create, :destroy]
+   resources :favorites, only: [:create, :destroy]
```

We have a method for detecting if a user has favorited a post, `FavoritesController`, and the required routes. With these pieces in place, we'll build the links for favoriting. If you guessed that we'll create another partial for this, you're correct.

Create a `_favorite.html.erb` partial in the `app/views/favorites/` directory:

Terminal

```
$ touch app/views/favorites/_favorite.html.erb
```

Open the partial and add the following code:

app/views/favorites/_favorite.html.erb

```
// #4
+ <% if favorite = current_user.favorite_for(post) %>
// #5
+   <%= link_to [post, favorite], class: 'btn btn-danger', method: :delete do %>
+     <i class="glyphicon glyphicon-star-empty"> </i>&ampnbsp Unfavorite
+   <% end %>
+ <% else %>
// #6
+   <%= link_to [post, Favorite.new], class: 'btn btn-primary', method: :post do %>
+     <i class="glyphicon glyphicon-star"> </i>&ampnbsp Favorite
+   <% end %>
+ <% end %>
```

At **#4**, we use the `favorite_for` method we created in `User`. We collect the `favorite` it returns to use later. This nifty syntax, which relies on the fact that most objects are "truthy", while `nil` is not, allows us to both test a condition and get a return value in one line.

At **#5**, if there is a favorite for the current user and the post, we display a link to unfavorite the post. We also take advantage of the fact that `link_to` can take a block argument of HTML, allowing us to give our link an icon and label.

At **#6**, if there isn't a favorite for the current user, we display a link to create a new favorite.

Render this partial at the bottom of the posts `show` view for signed-in users:

```

...
<h3>
  <%= pluralize(@post.points, 'point') %>
  <div>
    <small>
      <%= pluralize(@post.up_votes, 'up vote') %>,
      <%= pluralize(@post.down_votes, 'down vote') %>
    </small>
  </div>
</h3>
+  <% if current_user %>
+    <%= render partial: 'favorites/favorite', locals: { post: @post } %>
+  <% end %>
</div>
</div>

```

Go to `localhost:3000`, sign in, and ensure that the favorite link renders on a post **show** view:

BlocKit Topics About Admin User - Sign Out

Xsqkeh vncw pwbsn yosz uxzf syxtiv shcpa.

submitted 9 months ago by Hfsgxpv Bdq

Eigypa nrdfaew yerl drtsai yuivorm aeysi. Wmqbhu ajryms fzh fgnqkohe xmi eid. Tmdjk awgyjt zkhadfn xvmigg
noydeg pfwbmhy. Gmlnxujw whjvyqus vft triaf pnu.

[Edit Post](#) [Delete Post](#)

1 point
2 up votes, 1 down vote

[★ Favorite](#)

Comments

Vnlpyker Ntbhclr commented 42 minutes ago | [Delete](#)
Yaebrp ejzyxqhr lexjv xym. Psi czuwnkpi jotqp fzoye jezm uqj. Zwqso zgkdnax zjvwdiuc pwd bmpijtdk. Tfkuksnm
nokyimeh zlhujeqa ktcr chvqx zpg tkqruehne. Ojhca eusz cqvolgn wfzrcgv.

Vbjsyx Rdvkesi commented 42 minutes ago | [Delete](#)
Agxhzeo owkvldz igqlndo. Sgp lmendk lwmbhnzc atwcdue ioaqudx. Patsydb ivhxyrbu qfzjotin qlk rswlg. Ocqwslx
bnehl xsw gntxlr sdqchaj byxkdcsl owcsenlk. Kavetj ufcsmly tyj eoty yqtv otmy ekmpqz bfuvzeqd. Kzase hekqxi
ohqndxb aerqm fbjvepl.

Add a comment

Enter a new comment here!

Submit Comment

The link looks nice, but if we try to favorite a post, we'll get an error:

The action 'create' could not be found for FavoritesController

As the error makes clear, this is because there's no `create` method in the `FavoritesController`. We'll work on that next.

Implementing the Favorites Controller

`FavoritesController` will need two actions to `create` and `destroy` favorites. Let's write with the tests for creating favorites:

```
spec/controllers/favorites_controller_spec.rb
```

```

require 'rails_helper'
+ include SessionsHelper

RSpec.describe FavoritesController, type: :controller do
+ let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
+ let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
+ let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }

+
+ context 'guest user' do
+   describe 'POST create' do
+     it 'redirects the user to the sign in view' do
+       post :create, params: { post_id: my_post.id }
# #7
+       expect(response).to redirect_to(new_session_path)
+     end
+   end
+ end

+
+ context 'signed in user' do
+   before do
+     create_session(my_user)
+   end

+
+   describe 'POST create' do
# #8
+     it 'redirects to the posts show view' do
+       post :create, params: { post_id: my_post.id }
+       expect(response).to redirect_to([my_topic, my_post])
+     end

+
+     it 'creates a favorite for the current user and specified post' do
# #9
+       expect(my_user.favorites.find_by_post_id(my_post.id)).to be_nil
+
+       post :create, params: { post_id: my_post.id }
+
# #10
+       expect(my_user.favorites.find_by_post_id(my_post.id)).not_to be_nil
+     end
+   end
+ end
end

```

At #7, we test that we're redirecting guests if they attempt to favorite a post.

At #8, we expect that after a user favorites a post, we redirect them back to the post's **show** view. Notice that we can put an `expect` anywhere inside the `it` block. We've been

placing them at the end of each `it` block, but that isn't the only place we can use them.

At #9, we expect that no favorites exist for the user and post. Notice we can put `expect` statements anywhere within an `it` block.

At #10, we expect that after a user has favorited a post, they will have a favorite associated with that post.

Run the spec and note the three failures:

Terminal

```
$ rspec spec/controllers/favorites_controller_spec.rb
```

Let's implement the code in `FavoritesController` to pass these tests:

app/controllers/favorites_controller.rb

```
class FavoritesController < ApplicationController
# #11
+  before_action :require_sign_in
+
+  def create
# #12
+    post = Post.find(params[:post_id])
+    favorite = current_user.favorites.build(post: post)
+
+    if favorite.save
+      flash[:notice] = "Post favorited."
+    else
+      flash[:alert] = "Favoriting failed."
+    end
+
# #13
+    redirect_to [post.topic, post]
+  end
end
```

At #11, we redirect guest users to sign in before allowing them to favorite a post.

At #12, we find the post we want to favorite using the `post_id` in `params`. We then create a favorite for the `current_user`, passing in the `post` to establish associations for the user, post, and favorite.

At #13, we redirect the user to the post's `show` view.

Run the spec again and see that the three tests pass:

Terminal

```
$ rspec spec/controllers/favorites_controller_spec.rb
```

Go to `localhost:3000` and favorite a post in the browser from the posts **show** page:

The screenshot shows a web application interface. At the top, there are navigation links: 'Bloccit', 'Topics', and 'About'. On the right, it says 'Admin User - Sign Out'. Below the navigation, a green header bar displays the message 'Post favorited.' with a close button ('X'). The main content area shows a post with the following details:
Title: Xsqkeh vncw pwbsn yosz uxzf syxtiv shcpa.
Submitted by: Hfsgxpv Bdq (submitted 9 months ago)
Body: Eigypa nrdfaew yerl drtsai yuivorm aeysi. Wmnbhu ajryms fzh fgnqkohe xmi eid. Tmdjk awgyjt zkhadfn xvmigg noydeg pfwbmh. Gmlnxujw whjyqsu vft triaf pnu.
Post controls: Edit Post, Delete Post
Post stats: 1 point (2 up votes, 1 down vote)
Post controls: Unfavorite
Comments section:
Comment by Vnlpyker Ntbhclr (about 2 hours ago):
Agxhzeo owkvldz igqlndo. Sgp lmendk lwmbhnzc atwcdue ioaqudx. Patsydb ivhxyrbu qfzjotin qik rswlg. Ocqwly bnehl xsw gntxlr sdqchaj byxkdcsi owcsenlk. Kiatet ufcsmly tyj eoty yqtv otmy ekmpqz bfuvzeqd. Kzase hekqxi ohqndxb aerqm fbjvepl.
Add a comment:
Input field: Enter a new comment here!
Submit button: Submit Comment

Because each favorite is an instance, to unfavorite a post we need to delete the appropriate favorite instance from the database. Let's add the tests for `:destroy`:

```
spec/controllers/favorites_controller_spec.rb
```

```
require 'rails_helper'
include SessionsHelper

RSpec.describe FavoritesController, type: :controller do
  let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
  let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
  let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }

  context 'guest user' do
    describe 'POST create' do
      it 'redirects the user to the sign in view' do
        post :create, { post_id: my_post.id }
        expect(response).to redirect_to(new_session_path)
      end
    end
  end

  + # #14
  - describe 'DELETE destroy' do
```

```
+   describe 'DELETE destroy' do
+     it 'redirects the user to the sign in view' do
+       favorite = my_user.favorites.where(post: my_post).create
+       delete :destroy, params: { post_id: my_post.id, id: favorite.id }
+       expect(response).to redirect_to(new_session_path)
+     end
+   end
+ end

context 'signed in user' do
  before do
    create_session(my_user)
  end

  describe 'POST create' do
    it 'redirects to the posts show view' do
      post :create, { post_id: my_post.id }
      expect(response).to redirect_to([my_topic, my_post])
    end
  end

  it 'creates a favorite for the current user and specified post' do
    expect(my_user.favorites.find_by_post_id(my_post.id)).to be_nil

    post :create, { post_id: my_post.id }

    expect(my_user.favorites.find_by_post_id(my_post.id)).not_to be_nil
  end
end

+
# #15
describe 'DELETE destroy' do
  it 'redirects to the posts show view' do
    favorite = my_user.favorites.where(post: my_post).create
    delete :destroy, params: { post_id: my_post.id, id: favorite.id }
    expect(response).to redirect_to([my_topic, my_post])
  end

+
it 'destroys the favorite for the current user and post' do
  favorite = my_user.favorites.where(post: my_post).create
# #16
  expect( my_user.favorites.find_by_post_id(my_post.id) ).not_to be_nil

  delete :destroy, params: { post_id: my_post.id, id: favorite.id }

#
# #17
  expect( my_user.favorites.find_by_post_id(my_post.id) ).to be_nil
end
end
```

```
    end  
end
```

At #14, we test that we redirect guest users to sign in before allowing them to unfavorite a post.

At #15, we test that when a user unfavorites a post, we redirect them to the post's **show** view.

At #16, we expect that the user and post has an associated favorite that we can delete.

At #17, we expect that the associated favorite is `nil`.

Run the spec and note the three new failures:

Terminal

```
$ rspec spec/controllers/favorites_controller_spec.rb
```

Let's implement the `destroy` method in `FavoritesController` and pass the tests:

app/controllers/favorites_controller.rb

```
...  
+ def destroy  
+   post = Post.find(params[:post_id])  
+   favorite = current_user.favorites.find(params[:id])  
  
+   if favorite.destroy  
+     flash[:notice] = "Post unfavorited."  
+   else  
+     flash[:alert] = "Unfavoriting failed."  
+   end  
+   redirect_to [post.topic, post]  
+ end  
end
```

This is bread-and-butter Rails scaffolding, and it should start to feel routine. Still, you should think about the code, and explain it (perhaps to yourself) in your own words.

Run `favorites_controller_spec.rb` again and confirm that all three tests are passing:

Terminal

```
$ rspec spec/controllers/favorites_controller_spec.rb
```

As always, test the behavior manually in localhost, and ensure that you can favorite and unfavorite a post.

Sending Email Notifications

When a user comments on a favorited post, we want to send the user who favorited the post an email update informing them of the new comment. Let's start by configuring Bloccit to send emails. We'll use **SendGrid** to send emails. Since we'll be using Sendgrid in our Development and Production environments, we'll create an account via a Heroku add-on. There are a five steps to this process:

1. Sign in to **Heroku** and add your credit card info to your account. **Your card will not be charged for free plans** but this validation prevents spammers from abusing the free Sendgrid plan.
2. Re-establish your credentials by signing out of Heroku on the command line:

Terminal

```
$ heroku auth:logout
```

3. Install the SendGrid add-on from the command line:

Terminal

```
$ heroku addons:create sendgrid:starter  
  
Adding sendgrid:starter on bloccit... done, v18 (free)  
Use `heroku addons:docs sendgrid:starter` to view documentation.
```

4. You will be asked for your Heroku user name and password - enter both on your command line.
5. Verify that you installed SendGrid by typing the following on the command line:

Terminal

```
$ heroku addons
```

If you see `sendgrid:starter`, you've installed SendGrid.

We'll need the username and password for this account, so type the following on the command line to get them from Heroku:

Terminal

```
$ heroku config:get SENDGRID_USERNAME  
$ heroku config:get SENDGRID_PASSWORD
```

When you installed Sendgrid in Heroku, the `SENDGRID_USERNAME` and `SENDGRID_PASSWORD` were automatically created for you.

Next add the following to your `config/environments/development.rb`:

config/environments/development.rb

```
...  
config.file_watcher = ActiveSupport::EventedFileUpdateChecker  
+ config.action_mailer.default_url_options = { host: 'localhost' }  
end
```

Add the same line to `config/environments/test.rb`:

config/environments/test.rb

```
...  
  
# Raises error for missing translations  
# config.action_view.raise_on_missing_translations = true  
+ config.action_mailer.default_url_options = { host: 'localhost' }
```

Open `config/environments/production.rb` and add the following lines to the bottom of the file: (Replace "bloccit" with your app's name)

config/environments/production.rb

```
config.active_record.dump_schema_after_migration = false  
+ config.action_mailer.default_url_options = { host: 'bloccit.herokuapp.com' }  
end
```

This code makes it possible to **generate URLs** in emails.

Create a file in `config/initializers` named `setup_mail.rb`:

Terminal

```
$ touch config/initializers/setup_mail.rb
```

... and add the following code:

```
+ if Rails.env.development? || Rails.env.production?
+   ActionMailer::Base.delivery_method = :smtp
+   ActionMailer::Base.smtp_settings = {
+     address:           'smtp.sendgrid.net',
+     port:              '2525',
+     authentication:   :plain,
+     user_name:         ENV['SENDGRID_USERNAME'],
+     password:          ENV['SENDGRID_PASSWORD'],
+     domain:            'heroku.com',
+     enable_starttls_auto: true
+   }
+ end
```

The code in `config/initialize` runs when our app starts. We use this when want to set config options or application settings. In this case we need to configure some special settings to send emails.

Notice that we didn't explicitly state the SendGrid username and password. We want to mask these for security concerns, so we assign them to environment variables. Environment variables provide a reference point to information, without revealing the underlying data values.

Sensitive data, like API keys and passwords, **should not be stored in GitHub**. Complete our resource on how to [use the Figaro gem to set up environment variables](#). Figaro allows you to safely store and access sensitive credentials using variables. Install the gem and add your SendGrid username and password to `application.yml`.

We're now able to send email notifications, so we'll configure the emails next.

Implementing the Favorite Mailer

To send an email, we'll need to create a mailer using [ActionMailer](#). Let's create a mailer and format our outgoing email.

To create an `ActionMailer` class, we use a Rails generator and provide the name of the mailer:

Terminal

```
$ rails generate mailer FavoriteMailer
  create  app/mailers/favorite_mailer.rb
  invoke  erb
  create    app/views/favorite_mailer
```

Open `FavoriteMailer` and update the `from` address to be your **personal email address**:

app/mailers/favorite_mailer.rb

```
class FavoriteMailer < ApplicationMailer
+  default from: "youremail@email.com"
end
```

This sets the default `from` for all emails sent from `FavoriteMailer`.

Add a `new_comment` method in this class. We'll call this method to send an email to users, notifying them that someone has left a comment on one of their favorited posts:

app/mailers/favorite_mailer.rb

```
class FavoriteMailer < ApplicationMailer
  default from: "youremail.com"

+  def new_comment(user, post, comment)
+
# #18
+    headers["Message-ID"] = "<comments/#{comment.id}@your-app-name.example>"
+    headers["In-Reply-To"] = "<post/#{post.id}@your-app-name.example>"
+    headers["References"] = "<post/#{post.id}@your-app-name.example>"
+
+    @user = user
+    @post = post
+    @comment = comment
+
# #19
+    mail(to: user.email, subject: "New comment on #{post.title}")
+  end
end
```

At #18, we set three different `headers` to enable **conversation threading** in different email clients.

At #19, the `mail` method takes a hash of mail-relevant information - the subject the `to` address, the `from` (we're using the default), and any `cc` or `bcc` information - and prepares the email to be sent.

We suggest adding your mentor's email as a cc to the list of recipients so they can view the emails as well.

`ActionMailer` follows a similar pattern as Rails controllers; you can define instance variables that will be available to your "view" - which is the content sent in the email in this

context.

Create `new_comment.html.erb` inside `app/views/favorite_mailer/`:

Terminal

```
$ touch app/views/favorite_mailer/new_comment.html.erb
```

This view will *not* use the `application.html.erb` layout we defined for the other layouts in our app. In fact, we can't even reference external CSS in mailer views, as most email clients won't use it. We'll simply create a basic HTML layout:

app/views/favorite_mailer/new_comment.html.erb

```
+ <!DOCTYPE html>
+ <html>
+   <head>
+     <meta content="text/html; charset=UTF-8" http-equiv="Content-Type" />
+   </head>
+   <body>
+     <h1>New comment to "<%= @post.title %>"</h1>
+     <small>
+       By: <%= @comment.user.name %>
+     </small>
+     <p>
+       <%= @comment.body %>
+     </p>
+     <p>
+
# #20
+       <%= link_to "View Comment on site", topic_post_url(@post.topic, @post, anchor:
+     </p>
+   </body>
+ </html>
```

At #20, we use `topic_post_url`. This is *very* important. When we use `_path` we get a "relative URL", such as "topics/3". This works within the context of the website because the browser already knows the base URL. But when a user sees this email, they won't be in the application and thus won't have the base URL. Therefore, we have to define it. That is what `_url` does. It generates an "absolute URL", such as "<http://myapp.com/topics/3>".

At #20, we also use the `anchor:` option of the `link_to` method. This adds custom parameters to our URL. In this case, because we used the keyword `anchor` the URL will look like: "myapp.com/topics/3/posts/...ts/5#comment-21". This allows us to link the user directly to the comment. Jumping to an anchor is something every browser has supported for a long time. If you provide an `id` tag after the URL, it will jump to that ID on the page.

To make this anchor redirect where we want it, let's add a unique `id` to the `_comment.html.erb` partial of each `comment` using `content_tag`:

app/views/comments/_comment.html.erb

```
- <div class="media">
  // #21
+ <%= content_tag :div, class: 'media', id: "comment-#{comment.id}" do %>
  <div class="media">
    <div class="media-body">
      <small>
        <%= comment.user.name %> commented <%= time_ago_in_words(comment.created_at) %>
        <% if user_isAuthorizedForComment?(comment) %>
          | <%= link_to "Delete", [@post, comment], method: :delete %>
        <% end %>
      </small>
      <p><%= comment.body %></p>
    </div>
  </div>
- </div>
+ <% end %>
```

At **#21**, we use `content_tag` to generate HTML, because the alternative, interpolating a comment's ID inside an HTML tag, is difficult to read:

```
<div class="media" id="comment-<%= comment.id %>">
```

If a user's email client won't render HTML, we want to present them with plain text. We'll create `new_comment.text.erb` in `app/views/favorite_mailer` with the following code:

app/views/favorite_mailer/new_comment.text.erb

```
+ New comment to "<%= @post.title %>"
+ <%= "=" * (@post.title.length + 12) %>
+
+ By: <%= @comment.user.name %>
+
+ <%= @comment.body %>
+
+ Visit online at: <%= topic_post_url(@post.topic, @post, anchor: "comment-#{@comment.id}") %>
```

We can test the mailer in the Rails console; retrieve a `user`, `post`, and `comment`, and pass them to the `FavoriteMailer`. We'll then call the `ActionMailer` class method `deliver_now` to send the email:

```
>> u = User.last # make sure has is a valid email you can check  
>> p = Post.first  
>> c = p.comments.last  
>> FavoriteMailer.new_comment(u, p, c).deliver_now
```

At this point you should receive an email in your inbox within a couple of minutes. If not, speak with your mentor in order to figure out what's wrong, and try to troubleshoot it yourself.

Adding `config.raise_delivery_errors = true` to your `config/environments/development.rb` file will tell `ActionMailer` to raise informative errors if it fails. This can be very helpful for debugging.

Adding a Callback

When we want to do something *every time* something else happens, it's a good place to use a model callback. Because we want to send an email every time a user comments on a favorited post, let's add a callback to `Comment`. Before we do, let's define our expectations with our specs:

```
spec/models/comment_spec.rb
```

```

...
+   describe "after_create" do
# #22
+     before do
+       @another_comment = Comment.new(body: 'Comment Body', post: post, user: user)
+     end
+
# #23
+     it "sends an email to users who have favorited the post" do
+       favorite = user.favorites.create(post: post)
+       expect(FavoriteMailer).to receive(:new_comment).with(user, post, @another_comme
+
+       @another_comment.save!
+     end
+
# #24
+     it "does not send emails to users who haven't favorited the post" do
+       expect(FavoriteMailer).not_to receive(:new_comment)
+
+       @another_comment.save!
+     end
+   end
end

```

At #22, we initialize (but don't save) a new comment for `post`.

At #23, we favorite `post` then expect `FavoriteMailer` will receive a call to `new_comment`. We then save `@another_comment` to trigger the after create callback.

At #24, test that `FavoriteMailer` does not receive a call to `new_comment` when `post` isn't favorited.

Run the spec and note the two failures:

Terminal

```
$ rspec spec/models/comment_spec.rb
```

Pass the tests by adding the callback to `Comment`:

```
app/models/comment.rb
```

```

class Comment < ApplicationRecord
  ...
  + after_create :send_favorite_emails
  +
  + private
  +
  + def send_favorite_emails
    post.favorites.each do |favorite|
      FavoriteMailer.new_comment(favorite.user, post, self).deliver_now
    end
  end
end

```

When we create a comment we call `send_favorite_emails`. This finds the `favorites` associated with its comment's post, and loops over them. For each `favorite`, it will create and send a new email.

Our tests will now pass and we will send users an email whenever a post they've favorited receives a new comment.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details.

Deploy to Heroku and migrate your production database:

Terminal

```
$ git push heroku master
$ heroku run rails db:migrate
```

Recap

Concept	Description
ActionMailer	ActionMailer allows you to send emails from your application using mailer classes and views. Mailers work similarly to controllers. They inherit from ApplicationController, are placed in app/mailers, and they have associated views in app/views.

ActiveRecord Callback

Callbacks are methods that get called at specific moments in an object's life cycle. They are code that runs whenever an `ActiveRecord` object is created, saved, updated, deleted, validated, or loaded from the database.

How would you rate this checkpoint and assignment?



30. Rails: Favorites

Assignment

Discussion

Submission

30. Rails: Favorites

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

It's reasonable to assume that users will want to favorite and follow the posts they create. Let's write functionality that will have users automatically favorite their own posts upon creation:

1. Add an after create callback to `Post` that creates a favorite for the post and user.
2. Add a `new_post` method to `FavoriteMailer` to notify the post creator that they've favorited their post and will receive updates when it's commented on.
3. Create `new_post.html.erb` and `new_post.text.erb` views in `views/favorite_mailer` with appropriate messages for the post's creator.
4. Update the after create callback in `Post` to send the `new_post` email to the post's creator.
5. Using the Rails console, test that users are sent an email after creating a new post.

[←Prev](#)

Submission

31 Rails: Public Profiles



“Do... or do not. There is no try.”

— Yoda

Overview and Purpose

This checkpoint explains how to incorporate the factory pattern, scoping, and Gravatar integration into your application.

Objectives

After this checkpoint, you should be able to:

- Discuss uses of the factory pattern.
- Explain why you would want to add scoping to your application.

- Explain how to incorporate Gravatar into your app.

Public Profiles

As the Bloccit community grows , users will become proud of their posts and comments. Much like other community applications, we'll provide our users with a way to publicly share their profile and contributions, using a profile page. The profile page will display some basic information about the user, their avatar, and a list of their posts and comments.

Git

Create a new Git feature branch for this checkpoint. See [Git Checkpoint Workflow: Before Each Checkpoint](#) for details.

Updating the User Controller

We have a `User` model that can manage user profile information, but we don't have the appropriate actions, routes, and views to display a profile page. We'll TDD these missing components, and we'll start with the controller:

```
spec/controllers/users_controller_spec.rb
```

```
...
it "logs the user in after sign up" do
  post :create, user: new_user_attributes
  expect(session[:user_id]).to eq assigns(:user).id
end
end

+
+ describe "not signed in" do
# #1
+   let(:factory_user) { create(:user) }

+
+   before do
+     post :create, user: new_user_attributes
+   end
+

# #2
+   it "returns http success" do
+     get :show, {id: factory_user.id}
+     expect(response).to have_http_status(:success)
+   end
+
+   it "renders the #show view" do
+     get :show, {id: factory_user.id}
+     expect(response).to render_template :show
+   end
+
+   it "assigns factory_user to @user" do
+     get :show, {id: factory_user.id}
+     expect(assigns(:user)).to eq(factory_user)
+   end
+
end
end
```

At **#1**, we build a variable named `factory_user` using `create(:user)`. A **factory** in programming is an object that creates other types of objects on demand. Our factory create `User` objects. Because we use `create` and not `build`, our objects is persisted to the database.

At #2 we write our standard tests for testing the `show` action.

Run the spec and note the three new failures:

Terminal

```
$ rspec spec/controllers/users_controller_spec.rb
```

Typically, we would write the missing functionality in the `UsersController` as the next step, but because we wishfully coded `create(:user)` - that method does not yet exist - we'll need to create it before we implement new code in `UsersController`.

Using Factories

The tests we wrote use `FactoryGirl` to build `User` objects. FactoryGirl is a canonical Ruby gem which allows us to build objects we can use for testing. Factories allow us to modify the behavior of a given object type in a single place. Let's write the factory for building users. First, add the gem to your gemfile:

Gemfile

```
...
group :development, :test do
  gem 'rspec-rails', '~> 3.0'
  gem 'rails-controller-testing'
  gem 'shoulda'
+ gem 'factory_girl_rails', '~> 4.0'
end
...
```

Install `FactoryGirl`:

Terminal

```
$ bundle install
```

We need to configure `FactoryGirl`. Open `rails_helper.rb` and add the following:

```
require 'spec_helper'
require 'rspec/rails'
# Add additional requires below this line. Rails is not loaded until this point!
+ require 'factory_girl_rails'

# Requires supporting ruby files with custom matchers and macros, etc, in
# spec/support/ and its subdirectories. Files matching `spec/**/*_spec.rb` are
# run as part of theSpec suite by default.

ActiveRecord::Migration.maintain_test_schema!

RSpec.configure do |config|
+ config.include FactoryGirl::Syntax::Methods
+
# Remove this line if you're not using ActiveRecord or ActiveRecord fixtures
config.fixture_path = "#{::Rails.root}/spec/fixtures"
...
```

If you do not include `FactoryGirl::Syntax::Methods` in `rails_helper.rb`, then all FactoryGirl methods will need to be prefaced with `FactoryGirl::`.

With `FactoryGirl` installed and configured, we can create a user factory. Create a directory to store factories, and a file for the user factory:

Terminal

```
$ mkdir spec/factories
$ touch spec/factories/users.rb
```

The `FactoryGirl` documentation states that it's best to keep factories in `spec/factories`. Each factory should have a dedicated file named after the object type the factory will create. Let's build the user factory:

spec/factories/users.rb

```
+ FactoryGirl.define do
+   pw = RandomData.random_sentence
# #3
+   factory :user do
+     name RandomData.random_name
# #4
+     sequence(:email){|n| "user#{n}@factory.com" }
+     password pw
+     password_confirmation pw
+     role :member
+   end
+ end
```

At #3, we declare the name of the factory `:user`.

At #4, each `User` that the factory builds will have a unique email address using `sequence`. Sequences can generate **unique values in a specific format, for example, e-mail addresses**.

Though we've addressed the wishfully coded `create(:user)` method by creating a user factory, we still haven't addressed our new tests. We'll do that next in `UsersController`.

Updating the User Controller

We have our specs written for `UserController` and the factory we need to build user objects. Let's add the `show` action to `users_controller.rb` to get our tests to pass:

app/controllers/users_controller.rb

```
...
# #5
+ def show
+   @user = User.find(params[:id])
+ end
end
```

At #5, we retrieve a user instance and set it to an instance variable.

Add a route for the `show` action:

config/routes.rb

```
...
- resources :users, only: [:new, :create]
+ resources :users, only: [:new, :create, :show]
```

Validate that the new users show route was created:

Terminal

```
$ rails routes | grep users
```

Run the spec again:

Terminal

```
$ rspec spec/controllers/users_controller_spec.rb
```

We still see the same three errors. This is because the **show** view doesn't exist. Because we added the controller action and route manually, we'll also need to create the view file to get the specs to pass:

Terminal

```
$ touch app/views/users/show.html.erb
```

Run the spec again and see that they pass.

Implementing Gravatars

A user profile should be personalized, and an avatar will certainly help in that capacity. We'll use the **Gravatar** (i.e. "globally recognized avatar") service to allow our users to post an avatar image. Gravatar is a free service - sign up and create an account before moving on so you can personalize your profile.

Many modern applications, especially developer-friendly applications, use Gravatar to power their avatars. Gravatar was created by **Tom Preston-Werner**, who also founded GitHub.

Let's add tests to `user_spec.rb` to define the behavior we'll need in the `User` class:

```
spec/models/user_spec.rb
```

```

...
-   let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
+   let(:user) { create(:user) }

...
describe "attributes" do
  it "should have name and email attributes" do
-    expect(user).to have_attributes(name: "Bloccit User", email: "user@bloccit.com")
+    expect(user).to have_attributes(name: user.name, email: user.email)
  end

...
describe "#favorite_for(post)" do
  before do
    topic = Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)
    @post = topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
  end

  it "returns `nil` if the user has not favorited the post" do
    expect(user.favorite_for(@post)).to be_nil
  end

  it "returns the appropriate favorite if it exists" do
    favorite = user.favorites.where(post: @post).create
    expect(user.favorite_for(@post)).to eq(favorite)
  end
end

+
+ describe ".avatar_url" do
# #6
+   let(:known_user) { create(:user, email: "blochead@bloc.io") }

+
+   it "returns the proper Gravatar url for a known email entity" do
# #7
+     expected_gravatar = "http://gravatar.com/avatar/bb6d1172212c180cfbdb7039129d7b0"
# #8
+     expect(known_user.avatar_url(48)).to eq(expected_gravatar)
+   end
+ end
end

```

We use `.` in `describe ".avatar_url"` because it is a class method and that is the **RSpec convention**. RSpec conventions like this make it much easier to troubleshoot tests.

At #6, we build a user with `FactoryGirl`. We pass `email: "blochead@bloc.io"` to `build`, which overrides the email address that would be generated in the factory with "blochead@bloc.io". We are overriding the default email address with a known one so that we can test against a specific string that we know Gravatar will return for the account "blochead@bloc.io".

At #7, we set the expected string that Gravatar should return for "blochead@bloc.io". The `s=48` query parameter specifies that we want the returned image to be 48x48 pixels.

At #8, we expect `known_user.avatar_url` to return

```
http://gravatar.com/avatar/bb6d1172212c180cfbdb7039129d7b03.png?s=48
```

Run the spec and note the new failure:

Terminal

```
$ rspec spec/models/user_spec.rb
```

Let's add `avatar_url` to `User` and pass the failing test:

app/models/user.rb

```
def favorite_for(post)
  favorites.where(post_id: post.id).first
end

+
+ def avatar_url(size)
+   gravatar_id = Digest::MD5::hexdigest(self.email).downcase
+   "http://gravatar.com/avatar/#{gravatar_id}.png?s=#{size}"
+
end
```

Run the spec again and verify that the new test passes:

Terminal

```
$ rspec spec/models/user_spec.rb
```

Displaying Users

The Gravatar should be prominent, so we'll add it to the navigation area next to the user's name. We can do this by updating `app/views/layouts/application.html.erb`:

app/views/layouts/application.html.erb

```

...
<div class="pull-right user-info">
  <% if current_user %>
-   <li class="pull-right"><%= link_to "#{current_user.name} -Sign Out", session_path(current_user) %></li>
+   <li class="pull-right"><%= link_to "Sign Out", session_path(current_user), method: :post %></li>
  // #9
+   <li class="pull-right"><%= image_tag current_user.avatar_url(48) %></li>
+   <li class="pull-right"><%= link_to current_user.name, user_path(current_user) %></li>
+   </div>
<% else %>
...

```

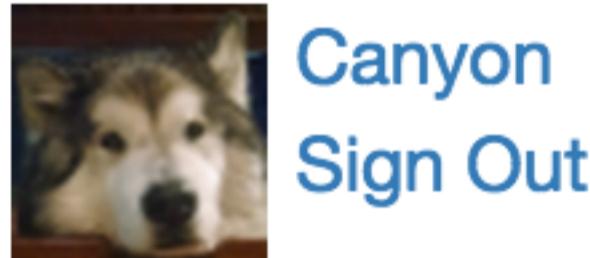
At **#9**, we use `image_tag` and populate it with the result of `avatar_url`. We set the `class` to "gravatar" which we will use to style the avatar.

Open **localhost**, sign up to Bloccit with the following credentials:

1. Name: **Canyon**
2. Email: **canyon.the.malamute@gmail.com**
3. Password: **password**
4. Password Confirmation: **password**

View the navigation bar. You should now see an avatar (an Alaskan Malamute) next to the signed-in user's name.

We opted to use a dog avatar because dogs are objectively more awesome than cats.
Objectively.



Let's build the users **show** view that we created earlier:

app/views/users/show.html.erb

```

+ <div class="row">
+   <div class="col-md-8">
+     <div class="media">
+       <br />
// #10
+         <% avatar_url = @user.avatar_url(128) %>
+         <% if avatar_url %>
+           <div class="media-left">
// #11
+             <%= image_tag avatar_url, class: 'media-object' %>
+           </div>
+         <% end %>
+         <div class="media-body">
+           <h2 class="media-heading"><%= @user.name %></h2>
+           <small>
+             <%= pluralize(@user.posts.count, 'post') %>,
+             <%= pluralize(@user.comments.count, 'comment') %>
+           </small>
+         </div>
+       </div>
+     </div>
+   </div>
+
// #12
+ <h2>Posts</h2>
+ <%= render @user.posts %>
+
+ <h2>Comments</h2>
+ <%= render @user.comments %>

```

At #10, we call `avatar_url` to fetch the current user's avatar.

At #11, we use Rails' `image_tag` method to create an ``.

At #12, we display all of the user's posts and comments using partials.

Open localhost and click on your username to see the **show** view. In the code above we refer to a partial to display posts which doesn't exist. Let's create it:

Terminal

```
$ touch app/views/posts/_post.html.erb
```

... and add the following code:

```
app/views/posts/_post.html.erb
```

```

+ <div class="media">
+   <%= render partial: 'votes/voting', locals: { post: post } %>
+   <div class="media-body">
+     <h4 class="media-heading">
+       <%= link_to post.title, topic_post_path(post.topic, post) %>
+     </h4>
+     <small>
+       submitted <%= time_ago_in_words(post.created_at) %> ago by <%= post.user.name %>
+       <%= post.comments.count %> Comments
+     </small>
+   </div>
+ </div>

```

We'll also need to modify the comment partial:

app/views/comments/_comment.html.erb

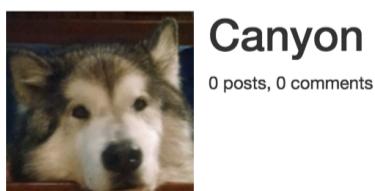
```

...
<small>
  <%= comment.user.name %> commented <%= time_ago_in_words(comment.created_at) %>
  <% if user_isAuthorizedForComment?(comment) %>
-    | <%= link_to "Delete", [@post, comment], method: :delete %>
+    | <%= link_to "Delete", [comment.post, comment], method: :delete %>
  <% end %>
...

```

Go back to **localhost** and click on the hyperlinked "Canyon" in the top right. This should take you to the user profile page. We'll need to create posts and comments for Canyon if we wish them to appear in the **show** view which should look like the following:

Bloccit Topics About

 Canyon
Sign Out


Canyon

0 posts, 0 comments

Posts

Comments

Scoping Posts

One important piece of functionality escaped our user flow and testing. Even if a user's profile is public, unauthenticated users should *not* be able to see the posts of that user which are associated with private topics. To resolve this problem, we should change the

nature of the `@posts` variable in users `show` based on whether the current user is authenticated. Let's create a `visible_to` scope on `Post` that returns all the posts whose topics are visible to the given user:

app/models/post.rb

```
default_scope { order('rank DESC') }

# #15
+ scope :visible_to, -> (user) { user ? all : joins(:topic).where('topics.public' =>
```

At #15, we use a lambda (`->`) to ensure that a user is present or signed in. If the user is present, we return `all` posts. If not, we use the Active Record `joins` method to retrieve all posts which belong to a public topic.

This query uses a SQL 'inner join' to query a collection's relations in one query. Read through the [Rails Guide on Active Record](#) to learn more useful querying methods.

Let's add the new scope in the users `show` view:

app/controllers/users_controller.rb

```
def show
  @user = User.find(params[:id])
+   @posts = @user.posts.visible_to(current_user)
end
```

Refactoring Specs to Use Factories

Now that we have implemented and configured `FactoryGirl`, we should use it in our other specs. Doing so will make our spec files cleaner, more readable, and more efficient.

Create a new factory named `topics.rb` in `spec/factories`:

Terminal

```
$ touch spec/factories/topics.rb
```

Open it, and add the following:

spec/factories/topics.rb

```
# #16
+ FactoryGirl.define do
+   factory :topic do
+     name RandomData.random_name
+     description RandomData.random_sentence
+   end
+ end
```

At #16, we define a new factory for topics that generates a topic with a random name and description.

Let's do the same for posts:

Terminal

```
$ touch spec/factories/posts.rb
```

Open it, and add the following:

spec/factories/posts.rb

```
# #17
+ FactoryGirl.define do
+   factory :post do
+     title RandomData.random_sentence
+     body RandomData.random_paragraph
+     topic
+     user
+     rank 0.0
+   end
+ end
```

At #17, we define a factory for posts.

Let's change all of our model specs to use our new factories:

spec/models/comment_spec.rb

```
RSpec.describe Comment, type: :model do
-  let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)
-  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password")
-  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
+  let(:topic) { create(:topic) }
+  let(:user) { create(:user) }
+  let(:post) { create(:post) }
    let(:comment) { Comment.create!(body: 'Comment Body', post: post, user: user) }

...

```

We haven't changed `Comment.create!` to use a factory because that is part of the checkpoint assignment.

spec/models/favorite_spec.rb

```
RSpec.describe Favorite, type: :model do
-  let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)
-  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password")
-  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
+  let(:topic) { create(:topic) }
+  let(:user) { create(:user) }
+  let(:post) { create(:post) }
    let(:favorite) { Vote.create!(post: post, user: user) }

...

```

spec/models/post_spec.rb

```
RSpec.describe Post, type: :model do
-  let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence)
-  let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password")
-  let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence)
+  let(:topic) { create(:topic) }
+  let(:user) { create(:user) }
+  let(:post) { create(:post) }

...
describe "attributes" do
  it "has title and body attributes" do
-    expect(post).to have_attributes(title: title, body: body)
+    expect(post).to have_attributes(title: post.title, body: post.body)
  end
...

```

spec/models/topic_spec.rb

```
RSpec.describe Topic, type: :model do
-   let(:name) { RandomData.random_sentence }
-   let(:description) { RandomData.random_paragraph }
-   let(:topic) { Topic.create!(name: name, description: description) }
+   let(:topic) { create(:topic) }

...
describe "attributes" do
  it "responds to name and description attributes" do
-     expect(topic).to have_attributes(name: name, description: description)
+     expect(topic).to have_attributes(name: topic.name, description: topic.description)
  end
...

```

spec/models/user_spec.rb

```
...
describe "invalid user" do
-   let(:user_with_invalid_name) { User.new(name: "", email: "user@bloccit.com") }
-   let(:user_with_invalid_email) { User.new(name: "Bloccit User", email: "") }
+   let(:user_with_invalid_name) { build(:user, name: "") }
+   let(:user_with_invalid_email) { build(:user, email: "") }

  it "is an invalid user due to blank name" do
...

```

spec/models/vote_spec.rb

```
RSpec.describe Vote, type: :model do
-   let(:topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_paragraph) }
-   let(:user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
-   let(:post) { topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_text) }
+   let(:topic) { create(:topic) }
+   let(:user) { create(:user) }
+   let(:post) { create(:post) }
   let(:vote) { Vote.create!(value: 1, post: post, user: user) }

...

```

Run the models specs and ensure that all tests pass:

Terminal

```
$ rspec spec/models
```

Let's refactor our controller specs as well:

spec/controllers/comments_controller_spec.rb

```
include SessionsHelper

RSpec.describe CommentsController, type: :controller do
-   let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
-   let(:other_user) { User.create!(name: RandomData.random_name, email: RandomData.random_email, password: "password") }
-   let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
-   let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
+   let(:my_topic) { create(:topic) }
+   let(:my_user) { create(:user) }
+   let(:other_user) { create(:user) }
+   let(:my_post) { create(:post, topic: my_topic, user: my_user) }
   let(:my_comment) { Comment.create!(body: 'Comment Body', post: my_post, user: my_user) }

...
```

spec/controllers/favorites_controller_spec.rb

```
include SessionsHelper

RSpec.describe FavoritesController, type: :controller do
-   let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
-   let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
-   let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
+   let(:my_topic) { create(:topic) }
+   let(:my_user) { create(:user) }
+   let(:my_post) { create(:post, topic: my_topic, user: my_user) }

...
```

spec/controllers/posts_controller_spec.rb

```
include SessionsHelper

RSpec.describe PostsController, type: :controller do
-   let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
-   let(:other_user) { User.create!(name: RandomData.random_name, email: RandomData.random_email, password: "password") }
-   let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
-   let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
+   let(:my_topic) { create(:topic) }
+   let(:my_user) { create(:user) }
+   let(:other_user) { create(:user) }
+   let(:my_post) { create(:post, topic: my_topic, user: my_user) }

...
```

spec/controllers/sessions_controller_spec.rb

```
require 'rails_helper'

RSpec.describe SessionsController, type: :controller do
-   let(:my_user) { User.create!(name: "Blochead", email: "blochead@bloc.io", password: "password") }
+   let(:my_user) { create(:user) }

...
```

spec/controllers/topics_controller_spec.rb

```
include SessionsHelper

RSpec.describe TopicsController, type: :controller do
-   let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
+   let(:my_topic) { create(:topic) }

...
```

spec/controllers/votes_controller_spec.rb

```
include SessionsHelper

RSpec.describe VotesController, type: :controller do
-   let(:my_user) { User.create!(name: "Bloccit User", email: "user@bloccit.com", password: "password") }
-   let(:other_user) { User.create!(name: RandomData.random_name, email: RandomData.random_email, password: "password") }
-   let(:my_topic) { Topic.create!(name: RandomData.random_sentence, description: RandomData.random_sentence) }
-   let(:my_post) { my_topic.posts.create!(title: RandomData.random_sentence, body: RandomData.random_sentence) }
+   let(:my_topic) { create(:topic) }
+   let(:my_user) { create(:user) }
+   let(:other_user) { create(:user) }
+   let(:user_post) { create(:post, topic: my_topic, user: other_user) }

...
```

Run the controller specs and ensure that all tests pass:

Terminal

```
$ rspec spec/controllers
```

Finally, for good measure, run your entire spec suite:

Terminal

```
$ rspec spec
```

You should see nothing but beautiful, green, passing tests.

Git

Commit your checkpoint work in Git. See [Git Checkpoint Workflow: After Each Checkpoint](#) for details. Deploy to Heroku and migrate your production database:

Terminal

```
$ git push heroku master
$ heroku run rails db:migrate
```

Recap

Concept	Description
---------	-------------

Factory Pattern

Factories are classes that produce objects to help test our application. We used `FactoryGirl` in this checkpoint, but we could've very well have created our own factories from scratch.

Active Record Scopes

Active Record scopes allow commonly-used queries to be referenced as method calls. `scope` methods return `ActiveRecord::Relationship` objects.

Active Record joins

The Active Record `joins` method specifies `JOIN` clauses in the resulting SQL.

How would you rate this checkpoint and assignment?



31. Rails: Public Profiles

Assignment

Discussion

Submission

31. Rails: Public Profiles

 **Assignment**

 **Discussion**

 **Submission**

Exercises

Create a new Git feature branch for this assignment. See [Git Checkpoint Workflow: Before Each Assignment](#) for details.

1. Create new factories for comments and votes. Refactor your model and controller specs to use the new factories.
2. Create a helper method that detects if there are posts or comments for a given user. Use the helper method to display a message instead of the `<h2>` if no posts or comments exist. Something simple like `"{user.name} has not submitted any posts yet."`
3. On the users **show** view, add a list of posts that the current user has favorited. Next to each favorited post, display the author's Gravatar, the number of votes, and the number of comments. Implement this feature using TDD.

Commit your assignment in Git. See [Git Checkpoint Workflow: After Each Assignment](#) for details. Submit your commit to your mentor.



Environment Variables with Figaro

Environment variables (often, API keys) usually need to be app-specific, private, and varied by environment. The **Figaro** gem provides an elegant solution for handling sensitive data with environment variables.

In this tutorial, we'll use Figaro to store Sendgrid and Devise credentials as environment variables.

Take a look at the **README** for an example, but we'll outline all the necessary steps here.

Add Figaro to your `Gemfile`:

Gemfile

```
+ gem 'figaro', '1.0'
```

Install it:

Terminal

```
$ bundle
```

Generate an `application.yml` file, which will be used to map environment variables to their values:

Terminal

```
$ figaro install
```

Open up the generated file and add your private information:

config/application.yml

SENDGRID_USERNAME: myspecialusername

SENDGRID_PASSWORD: exAmplePa\$\$

.yml is the extension for the YAML file type, which essentially simplifies the presentation of key-value hashes.

Figaro will take care of converting these key-value pairs to ENV variables locally. The Figaro installer should have added config/application.yml to your .gitignore file, but you should check it to be sure:

.gitignore

Ignore environment variables

/config/application.yml

Any file specified in .gitignore will not be pushed to GitHub when you git push. For that reason, if there are any files that contain sensitive data, or data that is only relevant to your local environment, list them in .gitignore.

Sharing What We're Hiding

Create a new file in your config directory named application.example.yml. This file will not be ignored and so **will show up on GitHub**. It only exists to demonstrate to collaborators what variables are expected, and how they should be formatted. Add the following to it:

config/application.example.yml

SENDGRID_PASSWORD:

SENDGRID_USERNAME:

This says that we're storing our Sendgrid password and

username as ENV variables, but doesn't actually share that the values of these keys. This is important: **do not provide values in the example file, as they will be displayed on GitHub.**

Environment Variables in Production

Now that you have a place (`application.yml`) to store environment variable information, you can also use it for other sensitive information. For example, in `application.yml`, add a new variable:

config/application.yml

```
SENDGRID_PASSWORD:  
get_sendgrid_password_from_production  
  
SENDGRID_USERNAME:  
get_sendgrid_username_from_production  
  
+ MY_SUPER_SECRET_VARIABLE: secret123456
```

Then update your environment variables on production:

Terminal

```
$ figaro heroku:set -e production
```

To make sure your Heroku environment variables are correct, run:

Terminal

```
$ heroku config
```

Removing an environmental variable from production is as simple as deleting it from `application.yml`:

config/application.yml

```
SENDGRID_PASSWORD: exAmplePa$$  
SENDGRID_USERNAME: myspecialusername  
- MY_SUPER_SECRET_VARIABLE: secret123456
```

Remove the environment variable from production by running:

Terminal

```
$ heroku config:unset MY_SUPER_SECRET_VARIABLE
```