26 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

Why We Test

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` gem to your Gemfile:

Gemfile

```
  source 'https://rubygems.org'

  # Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
  gem 'rails', '4.2.5'

  group :production do
    gem 'pg'
    gem 'rails_12factor'
  end

  group :development do
    gem 'sqlite3'
  end

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

  # Use SCSS for stylesheets
  gem 'sass-rails', '~> 5.0'
  # Use Uglifier as compressor for JavaScript assets
  gem 'uglifier', '>= 1.3.0'
  # Use CoffeeScript for .coffee assets and views
  gem 'coffee-rails', '~> 4.1.0'
  # Use jquery as the JavaScript library
  gem 'jquery-rails'
  # Turbolinks makes following links in your web application faster. Read more: https
  gem 'turbolinks'
```

> We added `rspec-rails` 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`) 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:

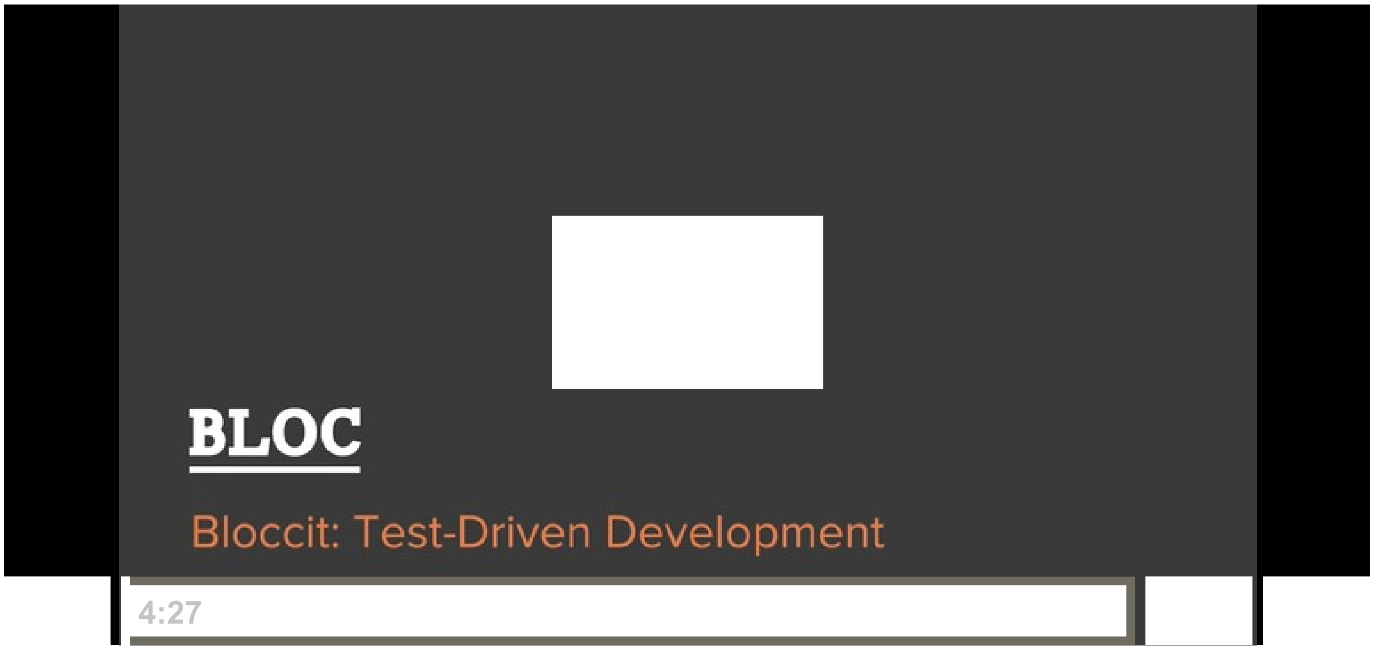
Terminal

```
$ 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 Database	Rails' dedicated test database allows developers to initiate and 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.

26. Rails: Testing

Assignment	**Discussion**	**Submission**

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

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*.

Testing Solution

assignment completed

?