RSpec Workshop

Workshop Repo

```
git clone https://github.com/Skipants/rspec-workshop/
```

Run `bin/bootstrap` to download Ruby 2.7.2 and set up the project databases.

It should complete with RSpec running and having a warning.

Unit Testing

- Follows “Arrange-Act-Assert”. This just means, do the following steps

1. Setup the state and variables needed for your test
2. Run the code under test
3. Check that the code under test matches your expectations

Example Unit Test File in RSpec

Docs @ https://relishapp.com/rspec/rspec-core/docs

```
require "spec_helper"

RSpec.describe Game do
  describe "#score" do
    let(:game) { Game.new }

    it "returns 0 for an all gutter game" do
      20.times { game.roll(0) }
      expect(game.score).to eq(0)
    end
  end
end
```

Breaking Down Each Part of a Unit Test

```
require "spec_helper"
```

This loads the “spec_helper” file, which is the conventional file for loading the Ruby and RSpec environment. For loading the whole Rails environment, “rails_helper” is used. In the financeit app, there is only a “rails_helper”. This is generally bad practice as having separate files allows you to run spec files that don’t need Rails more quickly. eg. Unit tests on plain Ruby objects.

Breaking Down Each Part of a Unit Test

```
RSpec.describe Game do
```

RSpec's code block for putting tests in. The convention is to pass the class under test to "describe". This can be a string as well. For tests like integration or feature tests that don't map 1-to-1 to the tests, you would probably use a string that describes the feature under test instead.

Breaking Down Each Part of a Unit Test

```
describe "#score" do
```

A nested describe block. You can nest as many as you'd like, but it generally gets messy if you are more than 3 describe's deep. Also aliased as `context`. The convention is that `describe` is used for classes and method names, `context` for descriptive text. eg.:

```
describe "#score" do
  context "after 20 rolls" do
```

Breaking Down Each Part of a Unit Test

```ruby
let(:game) { Game.new }
```

Sets the `game` variable for each test in this describe block. `let` is lazy loaded on each test. Part of the “Arrange” step. It also has a variant `let!` that is called *once* at the beginning of that whole block for every test within. That’s useful for expensive operations. When in doubt just use `let`.

Breaking Down Each Part of a Unit Test

```
it "returns 0 for an all gutter game" do
```

The block for the test itself.

Breaking Down Each Part of a Unit Test

```
20.times { game.roll(0) }
```

The code under test. Part of the “Act” step.

Breaking Down Each Part of a Unit Test

```
expect(game.score).to eq(0)
```

`expect...to` is RSpec's way of making an assertion. Part of the "Assert" step.

Unit Test Exercise

Fill out the spec in spec/lib/string_spec.rb

Goal: Just getting comfortable writing a test for something already written Time: 10 minutes

Tips:

- Don't overthink it
- You probably don't need to do any setup

File Conventions

Test files under `spec/` are usually 1-to-1 matches with files and their objects under the root folder or under `app/` with `_spec.rb` appended.

Examples:

1. `app/models/loan.rb` => `spec/models/loan.rb`
2. `app/controllers/loan.rb` => `spec/controllers/loans_controller.rb`
3. `lib/helper.rb` => `spec/lib/helper_spec.rb`

File Convention Exceptions

`spec/factories/`

- Contains factory_bot factories (more on that later)

`spec/fixtures/`

- Contain non-ruby files useful for testing, eg. images.
- Not to be confused with test-unit's fixtures, which are yaml files. We replace that with factory_bot.

`spec/integration/`

- Not actually the conventional place for integration tests for RSpec. These should be in `spec/requests/`

`spec/requests/`

- https://relishapp.com/rspec/rspec-rails/v/4-0/docs/request-specs/request-spec
- RSpec Rails' wrapper for integration tests
- They tend to follow how multiple method calls work together. More on these later

File Convention Exceptions

`spec/routing/`

- RSpec tests for your routes
- I've actually never written one of these in my life.
- Probably useful if your routes are crazy and you are trying to refactor them.

`spec/support/`

- Files that are included by tests that contain abstractions and reuseable pieces of code
- The place every included file for specs gets dumped.

Test Driven Development (TDD)

This just means writing tests before rather than after writing your application code. Some opinions on it are that it produces better code.

Andrew's Opinions^{TM} on TDD

Pros:

- It helps frame the problem and give you the big picture of how you can solve it
- You usually end up with cleaner code because you've analyzed how it should look from the start rather than incrementally making changes
- Gives you confidence your code works
- Much quicker than testing in the browser

Andrew's Opinions^{TM} on TDD

Cons:

- Difficult to do when adding code to something already untested. Often untested code wasn't written well for allowing tests and you'll need to get a full understanding of the code when writing tests for it (maybe that's a pro?)
- Writing tests requires you do know what your inputs and outputs will look like, but you might not understand what the code should look like until you've tried different things first
- It's a different way of thinking that needs to be practiced.

Andrew's Opinions^{TM} on TDD

Overall I think it works great when fixing bugs or making new additions. It's much harder when starting a new application where the architecture isn't flushed out yet.

Unit Test TDD Exercise

Given a class Months, implement a method that calculates the average amount of days of 3 months by name

Class is located in `lib/months.rb`

Constraints: - Write a test before even typing anything into `lib/months.rb`

Goal: To get comfortable with TDD Time: 30+ minutes

Unit Test TDD Exercise

Tips:

- You will need to create a test file for class lib/months.rb given the convention we talked about
- Use `Months::DAYS_IN_MONTH` to make this less about figuring out Rails' builtins and more just worrying about the problem
- Don't worry about good code or the implementation. Just testing first.
- The method may be a class or instance method. arguments may be passed to method or part of method signature
- Run the spec file as often as possible with `bundle exec rspec filename`
- Hardcode the values you in your expecation. Don't use `Months::DAYS_IN_MONTH` in your spec.

Request (Integration) Tests

RSpec Rail's implementation of integration

https://relishapp.com/rspec/rspec-rails/v/4-0/docs/request-specs/request-spec

```
it "creates a Widget and redirects to the Widget's page"
  get "/widgets/new"
  expect(response).to render_template(:new)

  post "/widgets", :params => { :widget => {:name => "My

  expect(response).to redirect_to(assigns(:widget))
  follow_redirect!

  expect(response).to render_template(:show)
  expect(response.body).to include("Widget was successfu
end
```

Request (Integration) Tests

- Integration tests are used to test how different parts of an application work together. In the context of a Rails app, it's how calling multiple endpoints in a row interact.
- An example is testing the flow of a user signup. It has multiple pages and at the end you may want to check something like whether the user was created or not.
- In the `financeit` app, we have integration tests under both `integration` and `requests`. The convention for RSpec is that they should be under `requests/`.
- Integration tests in RSpec test the content of the frontend, but we also generally use Cucumber for that.

Request (Integration) Tests

With integration tests, you're expectations probably want to check the side effects eg.

```
expect(User.count).to eq(1)
```

or

```
expect do
  post '/users'
end.to change{ User.count }.from(0).to(1)
```

We may also want to check what the endpoints under test return in their response:

```
expect(JSON.parse(response.body)).to eq({ full_name: "John
```

Breaking Down Each Part of an Integration Test

```
get "/widgets/new"
expect(response).to render_template(:new)
```

Integration tests hit endpoints just like a user or front-end would as they use the app. This says “when the user GETs /widgets/new, Rails renders the template ‘new’ ”

I actually don’t use `render_remplate` very much as it’s often kind of redundant. It’s more useful in cases where you’re rendering a template that you wouldn’t expect. Like rendering an error page.

Breaking Down Each Part of an Integration Test

```
post "/widgets", :params => { :widget => {:name => "My Wid
```

You can also send parameters to an endpoint.

Breaking Down Each Part of an Integration Test

```
expect(response).to redirect_to(assigns(:widget))
```

`assigns` checks what the controller assigned `@widget` to. This can be a useful way to check what sort of data we're passing to our views.

Breaking Down Each Part of an Integration Test

```
follow_redirect!
```

A helper method that we need to call to update the response after a redirect.

Breaking Down Each Part of an Integration Test

```
expect(response).to render_template(:show)
expect(response.body).to include("Widget was successfully c
```

Our expectations of what we get as a response after being redirected.

A Note on Setting Up Reusable State

With `let` it's very tempting to highlight every different change in state as a different variable. eg.:

```
let(:first_name) { "John" }
let(:last_name) { "Doe" }

context ".full_name" do
  it "puts the first and last name together" do
    expect(full_name(first_name, last_name)).to eq("Doe, Jo
  end

  it "can use a different seperator" do
    expect(full_name(first_name, last_name, ":")).to eq("Do
  end

  context "with a different name" do
    let(:first_name) { "Jane" }

    it "works" do
```

A Note on Setting Up Reusable State

Using let this way becomes very complicated as the amount of let or context blocks increase. Duplication is totally OK in tests and is encouraged. Tests are made to be disposable. It's a lot easier to understand which state can be deleted with each test when they are part of the test itself. Instead consider:

```
context ".full_name" do
  it "puts the first and last name together" do
    expect(full_name("John", "Doe")).to eq("Doe, John")
  end

  it "can use a different seperator" do
    expect(full_name("John", "Doe", ":")).to eq("Doe: John'
  end

  context "with a different first name" do
    it "works" do
      expect(full_name("Jane", "Doe")).to eq("Doe, Jane")
    end
```

A Note on Setting Up Reusable State

With more complicated state, you can avoid reusing the same data over and over again by providing a base set of parameters that works for the base case and mutate it in each test. eg.:

```
let(:valid_params) do
  {
    first_name: "John",
    last_name: "Doe",
    age: 30,
    city: "Toronto",
  }
end

context ".lives_here?" do
  it "is true" do
    expect(lives_here?(valid_params)).to eq(true)
  end

  it "is false for people outside of Toronto" do
```

Request Spec TDD Exercise

Implement the following:

- A user can ping people by name via `POST /ping` with a name, greeting, and company
- The greeting must always be “hello” and the company must always be “financeit”
- A user can only get help from `GET /help` if they have pinged `@skipants`. Otherwise the user gets a 404 response.

Constraints:

- Write an integration test before doing any code
- Write an invariant of the test – if you ping someone who is NOT `@skipants`, then you should get a 404 from `GET /help`

Goal: To get more comfortable with TDD and integration tests
Time: 45 minutes

Request Spec TDD Exercise

Tips:

- Start with a file in `spec/requests/`. Don't sweat the filename – they are more subjective than unit test files.
- Use a base set of parameters named `valid_params` and use `Hash#merge` to highlight the difference in state between your base case and the invariant
- You don't need to save `greeting` or `company` to the database. That constraint is just there to enforce and get you used to the usage of `valid_params`.
- Delete any guard assertions in the middle of your test when you're finished. ie. If you wrote an `expect` between the two requests, get rid of it after. It makes tests harder to follow later on and is only really useful for debugging.

Factory Bot

A gem used for organizing fixture data. Instead of passing the same parameters over and over to `.create` a database record, you can use factory definitions and traits to abstract that stuff away. It also provides a way to organize fixtures in such a way that they are reusable among different tests.

https://github.com/thoughtbot/factory_bot/blob/master/GETTING_STA

```
FactoryBot.define do
  factory :employee do
    login { 'employee' }
    email { 'employee@example.com' }

    trait :suspended do
      suspended_at { DateTime.current }
    end
  end
end
```

This makes a fixture for the `Employee` model. You can then create

Controller Specs

Basically unit tests for controllers. RSpec Rails provides some helper methods to make these simpler.

https://relishapp.com/rspec/rspec-rails/v/4-0/docs/controller-specs

```
RSpec.describe TeamsController do
  describe "GET index" do
    it "assigns @teams" do
      team = Team.create
      get :index
      expect(assigns(:teams)).to eq([team])
    end

    it "renders the index template" do
      get :index
      expect(response).to render_template("index")
    end
  end
```

Controller Spec + Factory Bot Exercise

Reimplement the previous integration test for pinging @skipants as a controller test with one new requirement: The user can only ping @skipants once per day.

Constraints:

- Set up the state of the test using Factory Bot.

Goal: To get comfortable with controller specs and Factory Bot
Time: 45 minutes

Tips:

- In the previous exercise you probably called the `/ping` endpoint. In this exercise you should instead save the pings straight to the database as part of your setup.
- You have to create both the factory and the test file for this exercise
- Try using a trait to automatically make a ping with a name of "@skipants". You probably wouldn't do this normally as it's simpler to just build

Mocks and Stubs

https://relishapp.com/rspec/rspec-mocks/docs

https://relishapp.com/rspec/rspec-mocks/v/3-10/docs/basics/allowing-messages

You can stub methods on any object by using `allow(object).to receive(:method)`. This is useful for returning a certain response from a method that isn't under test but is required to get your test to work as expected.

For example, setting the internal state of an object and then calling another method that depends on it:

```
context "#barks?" do
  it "barks if it had no treats today" do
    dog = Dog.new
    allow(dog).to receive(:treats_eaten) { 0 }

    expect(dog.barks?).to eq(true)
  end
```

Controller Spec Stub Exercise

Reimplement the previous controller test for pinging @skipants as with one new requirement: The user can only ping @skipants 5 times per month.

Constraints:

- Set up the state of the test using mocks and/or stubs.

Goal: To get comfortable with mocking and stubbing Time: 20 minutes

Tips:

- There's more than one way to do it. You will probably need stubs but it can definitely be done without mocks
- Only your setup should change. Calling the controller method and

Tips For Testing

- Avoid using several `let` statements. It ends up making code harder to follow. Like we talked about earlier, use a base set of reuseable state and modify it within the tests.
- Avoid writing to the database if you don't have to. Favour factory_bot's `build` over `create`. Writing to the database is very slow and is the main culprit of slowdown in large test suites.
- factory_bot will cascade creates when you use `create` and it has associations making it write even more to the database than you'd expect.