**BASICS OF RSPEC**

**USEFUL LINKS**

* Rspec github and basics - <https://github.com/rspec/rspec-rails>
* Rspec documentation - <https://relishapp.com/rspec/rspec-rails/v/3-6/docs>
* The basics of Capybara and Improving your tests <https://www.sitepoint.com/basics-capybara-improving-tests/>
* Rspec model testing template/cheat sheet, basics and advanced - <https://gist.github.com/kyletcarlson/6234923>
* Also see Capybara\_cheat\_sheet.pdf in ruby folder

**IMPORTANT THINGS TO NOTE/KEEP IN MIND**

* You can view belt\_review\_events in full\_rails for a rundown, comments, and the basics
* You can run test with either **bundle exec rspec**, or just **rspec spec**. You can run specific files, folders, or lines by specifying the file\_path, by either adding it to the first one or, just adding it after “spec” on the second one. Ex: **bundle exec rspec spec/controllers/users\_controller\_spec.rb** and **rspec spec/controllers/users\_controller\_spec.rb:8** the first one just runes that entire file, the second one only runs line 8
* As far as I can see, you can only test and access sessions within controller test files
* Due to “database cleaner”, every test starts with a blank database. Important to keep in mind. For example, if you want to create a user for a test, but it requires a foreign key from State, you need to create a state record first. Also, for one of my views in the belt\_review\_events app, it took all the records from the State model and put them in a select input, one option for each state. However, in my tests, it sees the select as blank because the model starts out blank, so nothing to pull from, so I had to create one or more states, depending on the test, to get it to actually work.
* When you run all tests at once, with bundle exec rspec, or just rspec spec, it runs all the files in the spec folder top to bottom.
* All \_spec files require rails\_helper first, and rails\_helper requires spec\_helper, so those two files are always loaded first. As such, any methods or code you put in either file will be available in the entire spec folder, but if those files are heavy and slow to run, all of your tests will be as well.
* Whatever you write within the quote marks for a block of code, whether it be a “describe do” block, “it do” block, etc. will be read by rspec and sent to the terminal if it fails so you know what exactly failed. For example: if you have a “describe ‘pressing the log in button’ do” block of code, and within that, an it block of code like: “it ‘should create a session’ do”, then if that it was to fail, the teriminal would reference the file and basically say “pressing the log in button should create a session” has failed.

**Gems to include in Gemfile**

After you install with **bundle install**, type: **rails g rspec:install**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* GEMS START \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

group :test, :development do

gem 'rspec-rails' # allows testing with rspec

gem 'database\_cleaner' # a gem that cleans/clears the test databse for every rspec test, very important for testing

end

group :test do

gem 'capybara' # a web-based automation framework that simulates how users interact with your application. Basically, it is used in your tests to act like a user. Things like visit, fill\_in, click\_link, click\_button, etc. is capybara at work

end

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* GEMS END \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**CONFIGURING/SETTING UP RSPEC and Database\_cleaner**

* After installing and setting up rspec with **rails g rspec:install**, it creates the spec folder, with rails\_helper.rb and spec\_helper.rb by default. It should also create a controllers, views, helpers, and models folder if you’ve created them in your project up to that point, more on that in the section below
* Leave spec\_helper.rb as is, no need to make any changes. However, for rails\_helper.rb, leave the top code alone, go below the line “ActiveRecord::Migration.maintain\_test\_schema!” and starting with “RSpec.configure do |config|” replace the default code with the code below:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* START OF COPY & PASTE CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

RSpec.configure do |config|

config.fixture\_path = "#{::Rails.root}/spec/fixtures"

config.use\_transactional\_fixtures = false # this needs to be false

config.infer\_spec\_type\_from\_file\_location!

config.filter\_rails\_from\_backtrace!

config.include Capybara::DSL

# The below section is for the database cleaner, which cleans/clears the test database after each test

config.before(:suite) do

DatabaseCleaner.clean\_with(:truncation)

DatabaseCleaner.strategy = :truncation

end

config.before(:each) do

DatabaseCleaner.start

end

config.after(:each) do

DatabaseCleaner.clean

end

end

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* END OF COPY & PASTE CODE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* The code that is in there by default, without the default comments it comes with, looks like:

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*start of default code \*\*\*\*\*\*\*\*\*\*

RSpec.configure do |config|

config.fixture\_path = "#{::Rails.root}/spec/fixtures"

config.use\_transactional\_fixtures = true

config.infer\_spec\_type\_from\_file\_location!

config.filter\_rails\_from\_backtrace!

end

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end of default code \*\*\*\*\*\*\*\*\*\*

**FILE CONFIGURATION, Default and Custom Made**

* If you set up rspec with **rails g rspec:install** right after you install your project, it sets up the spec folder with just rails\_helper.rb and spec\_helper.rb
* Other files are created by default once you add models and controllers to your rails app. For example, If I type **rails g model User**, the models folder in spec is created if it’s not already there, and the file user\_spec.rb is created within it. Also, if I create a controller with **rails g controller Users**, a controllers and helpers folder is created if not already there with the files users\_controller\_spec.rb and users\_helper\_spec.rb respectively.
* Lastly, a views folder and file is created if you create them with your controller. Meaning, if I create a controller with **rails g Controller Users index** then the views folder will be created if it’s not already there, and within it a users folder will be created with the file index.html.erb\_spec.rb inside.
* Aside from these files/folders that spec creates on its own in reaction to your commands, you can create whatever files and folders you want, and as long as they are set up right, you can test in them
* Normally, I also make a features folder, which basically just takes place of the view, since I don’t normally create the views whenever I create my controller. For example, in belt\_review\_events, I have a features folder, and inside that a users and sessions folder. In my sessions folder, I have a logging\_in\_spec.rb file and and a logging\_out\_spec.rb file. Within the users folder, I have a create\_user\_spec.rb file and new\_user\_spec.rb

**SETTING UP YOUR FILES**

* **EVERY FILE NEEDS** the line: **require ‘rails\_helper’** at the top of it. the tests will not work otherwise. All files that rspec creates by default will have this in them.
* Besides that, controllers, models, helpers, and views all have their own type of RSpec.describe statement by default which your code needs to be wrapped in. And all other files need to be wrapped in an RSpec.describe as well. Examples Below
* The 4 examples below are for controllers, helpers, views, and model files and come with this in them by default. The default code they will have in them can be seen in the helper example: “pending "add some examples to (or delete) #{\_\_FILE\_\_}"
* Controller File for Users:

\*\*\*\*\*start of controller example \*\*\*\*\*

RSpec.describe UsersController, type: :controller do

code here

end

\*\*\*\*\*end of controller example \*\*\*\*\*

* Helper File for Users:

\*\*\*\*\*start of helper example \*\*\*\*\*

RSpec.describe UsersHelper, type: :helper do

pending "add some examples to (or delete) #{\_\_FILE\_\_}"

end

\*\*\*\*\*end of helper example \*\*\*\*\*

* View File for Users Index:

\*\*\*\*\*start of view example \*\*\*\*\*

RSpec.describe "users/index.html.erb", type: :view do

pending "add some examples to (or delete) #{\_\_FILE\_\_}"

end

\*\*\*\*\*end of view example \*\*\*\*\*

* Model File for User Model:

\*\*\*\*\*start of model example \*\*\*\*\*

RSpec.describe User, type: :model do

code here

end

\*\*\*\*\*end of model example \*\*\*\*\*

* All other files that you make yourself need to be wrapped in “RSpec.describe ‘something’ do….. end”. Example of a custom file:

\*\*\*\*\*start of custom example \*\*\*\*\*

RSpec.describe 'logging in' do

code here

end

\*\*\*\*\*end of custom example \*\*\*\*\*

**THE BASICS STATEMENTS/BLOCKS OF CODE**

* Again, view belt\_review\_events for examples of the basics
* **IT DO:** Most tests are wrapped in an “it ‘something’ do” block of code ex:

it "should create a session" do

code here

end

* **DESCRIBE DO:** Aside from the RSpec.describe… at the top of the page, you can also wrap your tests in a plain “describe ‘something’ do” block of code. this acts like a scope and over-arching test. If I was to put a “before do” block of code (more on that below) inside a “describe do” block, it would only affect the other code inside of that “describe do” block, and wouldn’t affect anything outside of that block. Similar to how if you define a variable inside a function/method, it can only be used inside the function/method but doesn’t exist outside of it. You can also use a “describe do” block to break one large test into pieces, for example:

describe 'creating a user' do

it "should create a user" do

end

it "should log in the user" do

end

end

* In the example above, both “it do” blocks are within the “describe do” block. An easy way to understand them is that, when you run the tests for this file, rspec sees both “it do” as pieces of the “describe do”. For example, if the first “it do” were to fail, in the terminal, rspec would reference the file and basically say that “creating a user should create a user” failed. It combines the stuff in quotes for both the describe and it block. Likewise, if the second “it do” block failed, the terminal would say “creating a user should log in the user” failed.
* **BEFORE DO**: Another common rspec block used is the before/after do block of code. I normally use the before block of code. As it sounds, before every test within its scope, it runs. For example, if I wanted to test doing something with a user record, I would need to create one before running the test since the test db starts out blank. But, if there were 5 tests that did this, it wouldn’t be DRY code to write that 5 times. So I could write

before do

create\_user

end

* and it would create a user prior to each test, and since the db is cleared after each test, it would always be the first record in the User model. Import thing to remember about the before do is its scope, it will always run before every test within its scope, but not for anything outside of its scope. See the example below:

before do #before statement 1

create\_user

end

describe 'something' do

before do #before statement 2

log\_in\_user

end

it "text" do

end

it "text" do

end

end

* In the example above, the first before statement would run before everything in that code. Basically, if I ran that file, it would go something like: run before#1, run before#2, run first it, finish test. then: run before#1, run before#2, run second it, finish test. However, if there was an it outside of the “describe do” block, then only the first “before do” block would run prior to it, the second “before do” block isn’t within its scope.
* **VARIABLES**: if you want to use variables across tests, you create them with **@variable**, through anything that happens to them during a test is forgotten after words, all tests start with a blank slate. So, I usually put it in a before do block of code, ex:

describe "sessions id and first\_name" do

before do

@user = User.first

end

it "should be created upon login" do

end

it "should be destroyed upon logout" do

end

end

* In the describe block above, both it blocks have access to the @user variable, and when the test is finished, the variable is erased, and then re-created at the start of the next test in the before block.

**USING CAPYBARA AND BASIC TESTS**

* **VISITING A PAGE**: you can visit an actual page via capybara doing either **visit new\_user\_path** or **visit ‘/users/new’**
* **HITTING A ROUTE/METHOD**: You can hit a route to say, test a method in a controller, with the format “action :method, parameters”, you normally do this within a controller\_spec file, so it already knows which controller you’re talking about. You always want to hit the method just like the view would, with the correct parameters and action. Example, both in the users controller: **post :create, user{first\_name: ‘Tony’, last\_name: ‘Start’, email: ‘tony@stark.com’, password: ‘password’, password\_confirmation: ‘password’, state: 1}** In this example, I post, to the create method in the user controller. The create method expects a hash titled ‘user’ that it puts through a strong param, so that’s exactly what I send it. Another example: **post :create, email:@user.email, password:@user.password**, this example is for the sessions controller, testing the log in. It’s not expecting a hash, but two parameters, titled ‘email’ and ‘password’. The @user is a variable I created in a before do block. Last example: **delete:destroy, id: 1**, in this example, I hit the destroy method in the sessions controller, through a delete action, which by default of RESTful routes, needs an id parameter.
* **EXPECT**: expect() is one of the most commonly used things in a test. a simple example, after a user logs in, I want to test that the session[:id] is created and has his id so: **expect(session[:id]).to eq(user.id)**, “ **.to eq()** ” is also very commonly used. If I wanted to test that the session was destroy after loggin in, I could test: **expect(session[:id]).to eq(nil)**. Other examples, when testing the model, you would type “user = User.new” followed by “user.valid?” which tests if your model validations work, the actual test would be: **expect(user.errors[:first\_name].any?).to eq(true)**, if the first\_name column in this model is required, this would pass. Another test is to test that an email has to be unique, so you create a user with an email, then try to create a second user with the same email, this would be the test after you run “user2.valid?” - **expect(user2.errors[:email].first).to eq(“has already been taken”)**. You can also test length, ex: **expect(user.password).to be > 8**. A very useful one is testing what page you are on after hitting a link or button. For example, after creating a user, it should send me to the ‘/events’ page, so I would type “click\_button ‘Create User’” then the test would be: **expect(current\_path).to eq(‘/events’)**, you could also put in something like new\_user\_path, but for a different test.
* **EXPECT WITH A PAGE**: Say I wanted to make sure a create user page has the correct fields, I could have tests such as **expect(page).to have\_field(“First name”)**, or **expect(page).to have\_text(“Create a New User!”** or **expect(page).to have\_link(‘Press Here!’)**, or **expect(page).to have\_button(‘Create User’) expect(page).to**, and finally for a select input: **expect(page).to have\_select(‘State’, options: [‘TX’, ‘AL’]**, IMPORTANT NOTE: In the last expect have\_select, the view populates that select statement with data from my State model, so in order for the test to pass, I have to create those two states prior to the test, otherwise, capybara would read that select input as blank.
* IMPORTANT NOTE: when it comes to expect(page) and have\_field, have\_button, etc. it’s important to know whether you’re using form helpers or html. With form helpers, the have\_field would look at the label, and form helpers always capitalize them in the view, so “First name”, as for the button, it would read the text on the actual button, and when creating something, the form helper always makes it “Create whatever”. But, if I wrote the form myself via html, the have\_field should read the name value of that input, so it would be have\_field(“first\_name”) or whatever I put.
* **FILLING IN/INTERACTING WITH A PAGE:** If I wanted to say, test that creating a user actually works, I would need to act like a user in my test via capybara. So if I wanted to fill in the first\_name input which I created in html, it would be: **fill\_in ‘first\_name’, with ‘Tony’**, the format is “fill\_in ‘field’, with: value”. For select statements, it would be: **select “TX”, from: ‘State’**, so now the format is “select ‘value’, from: ‘field’”. Lasty, you can click on a link or button with **click\_button ‘Create User’** and **click\_link ‘Logout’**, you can even test where you should end up after clicking that button or link with **expect(current\_path).to eq(‘/events’)** or **expect(current\_path).to eq(new\_session\_path)**