University of Sheffield

COM1001 SPRING SEMESTER
Professor Phil McMinn
p.mcminn@sheffield.ac.uk

Unit & Integration Testing of Web Applications with RSpec

Unit Testing Hello World

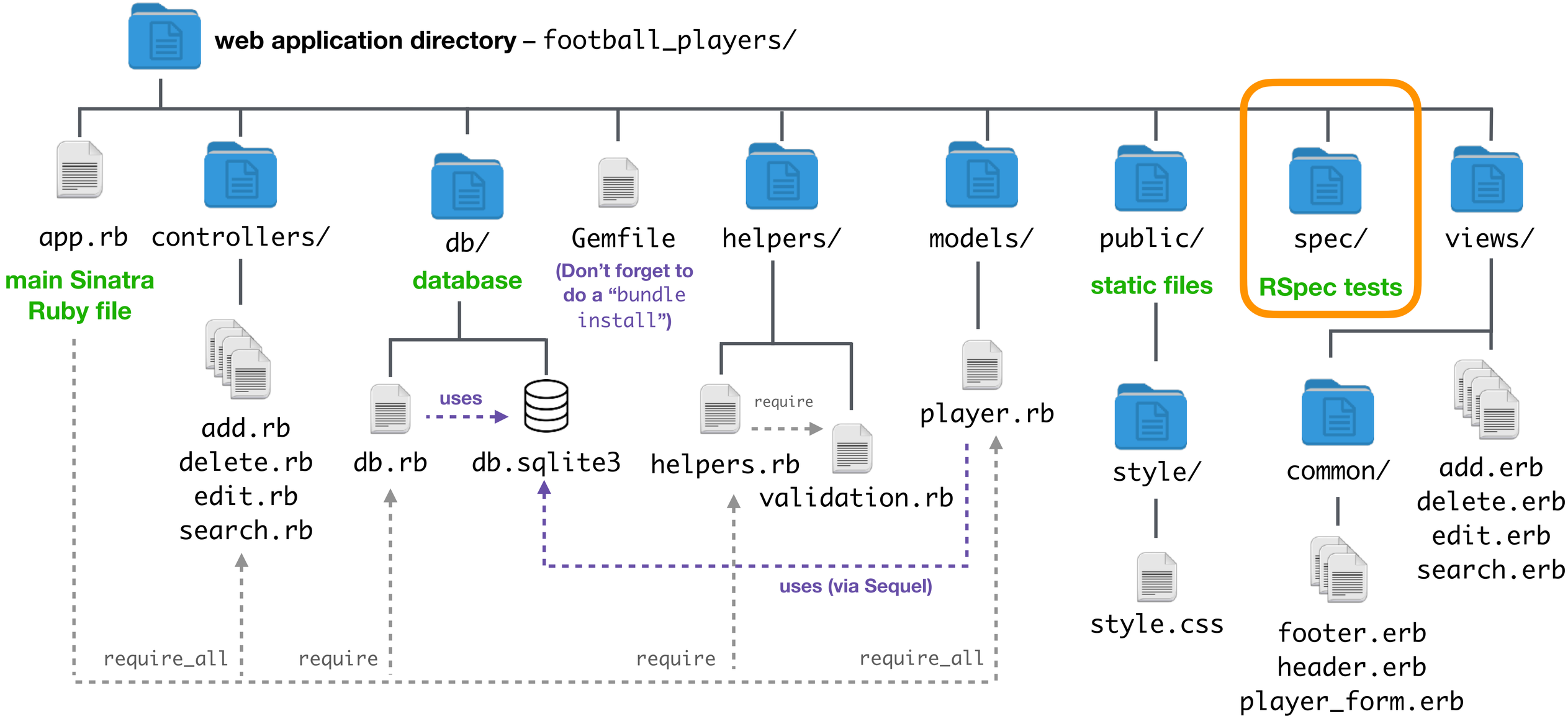
```
RSpec.describe "Hello World example" do
  describe "GET /hello-world" do
    it "has a status code of 200 (OK)" do
      get "/hello-world"
      expect(last_response.status).to eq(200)
    end

    it "says 'Hello, World!'" do
      get "/hello-world"
      expect(last_response.body).to eq("Hello, World!")
    end
  end
end
```

This is how we cause the web page to be “loaded” for the purposes of the test. It happens behind the scenes – we don’t see it in a web browser. Essentially we’re calling the Sinatra route as though it were a method. (If it were a post route we’d be calling `post` rather than `get`.)

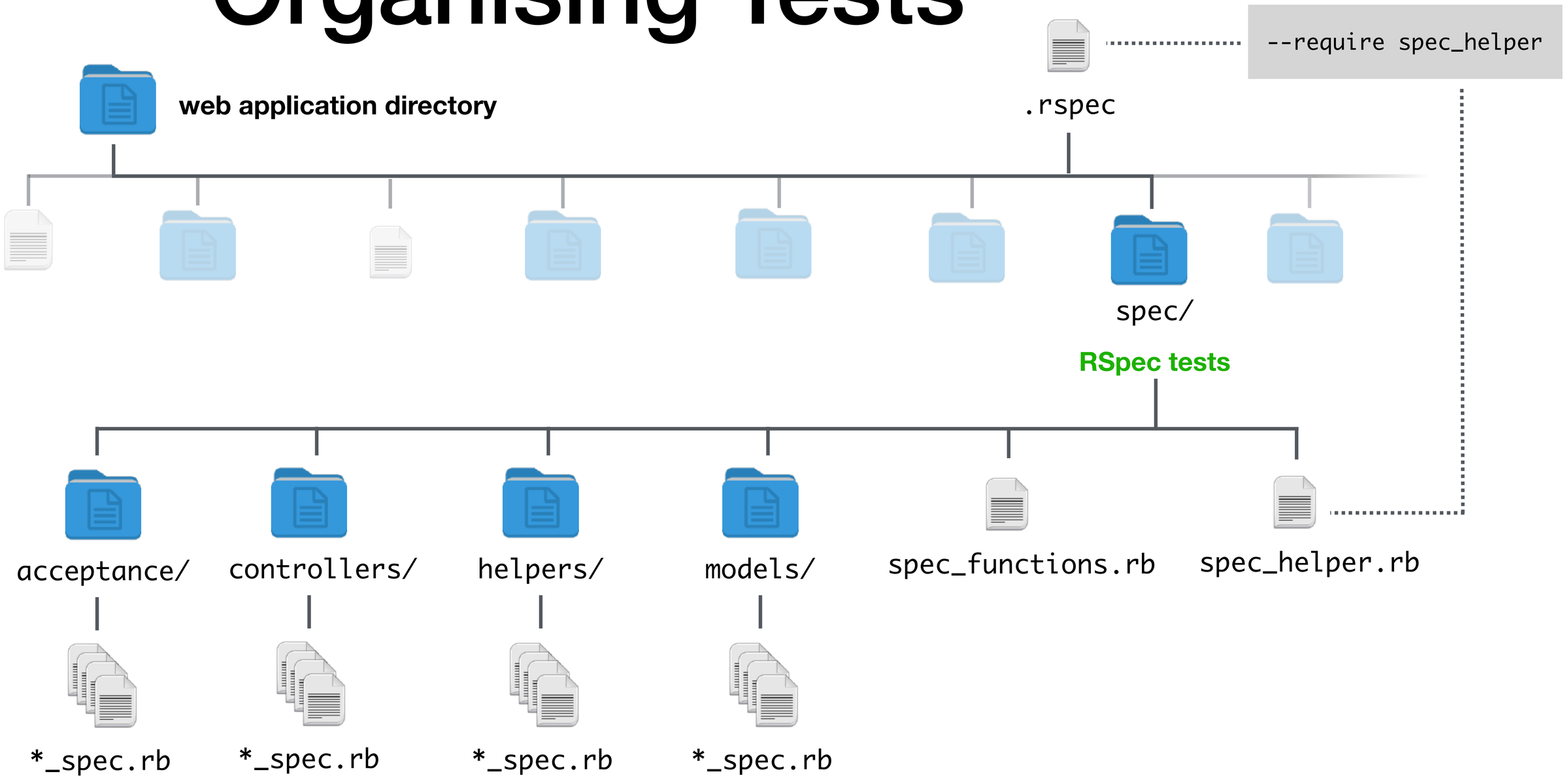
This is how we check the page is “doing” the right thing – by querying an object called `last_response`. This object is created as a result of the “`get`” call in the prior line, storing details of the resulting HTTP response. We can therefore check the HTTP status code and the content of the body (the HTML) of the page.

When we’re dynamically generating content it’s not going to be possible to check the response body exactly matches a string. We can use `include` instead of `eq` to check the body includes certain strings of text, as we will see later…

Understanding the File Structure



Organising Tests



spec_helper.rb

```ruby
# SimpleCov
require "simplecov"
SimpleCov.start do
  # don't track coverage of tests!
  add_filter "/spec/"

  # don't track core application files
  add_filter "/app.rb"
  add_filter "/db/db.rb"
  add_filter "/helpers/helpers.rb"
end
SimpleCov.coverage_dir "#{__dir__}/_coverage"

# Sinatra App
ENV["APP_ENV"] = "test"
require_relative "../app"
def app
  Sinatra::Application
end

# Capybara
require "capybara/rspec"
Capybara.app = Sinatra::Application

# RSpec
RSpec.configure do |config|
  config.include Capybara::DSL
  config.include Rack::Test::Methods
```

`spec_helper.rb` does all the configuration needed to set up rspec as needed for your web application.

There is no need to edit this file.

spec_functions.rb

```
def spec_before
  # reset database here
end
```

Put “helper” functions for tests here.

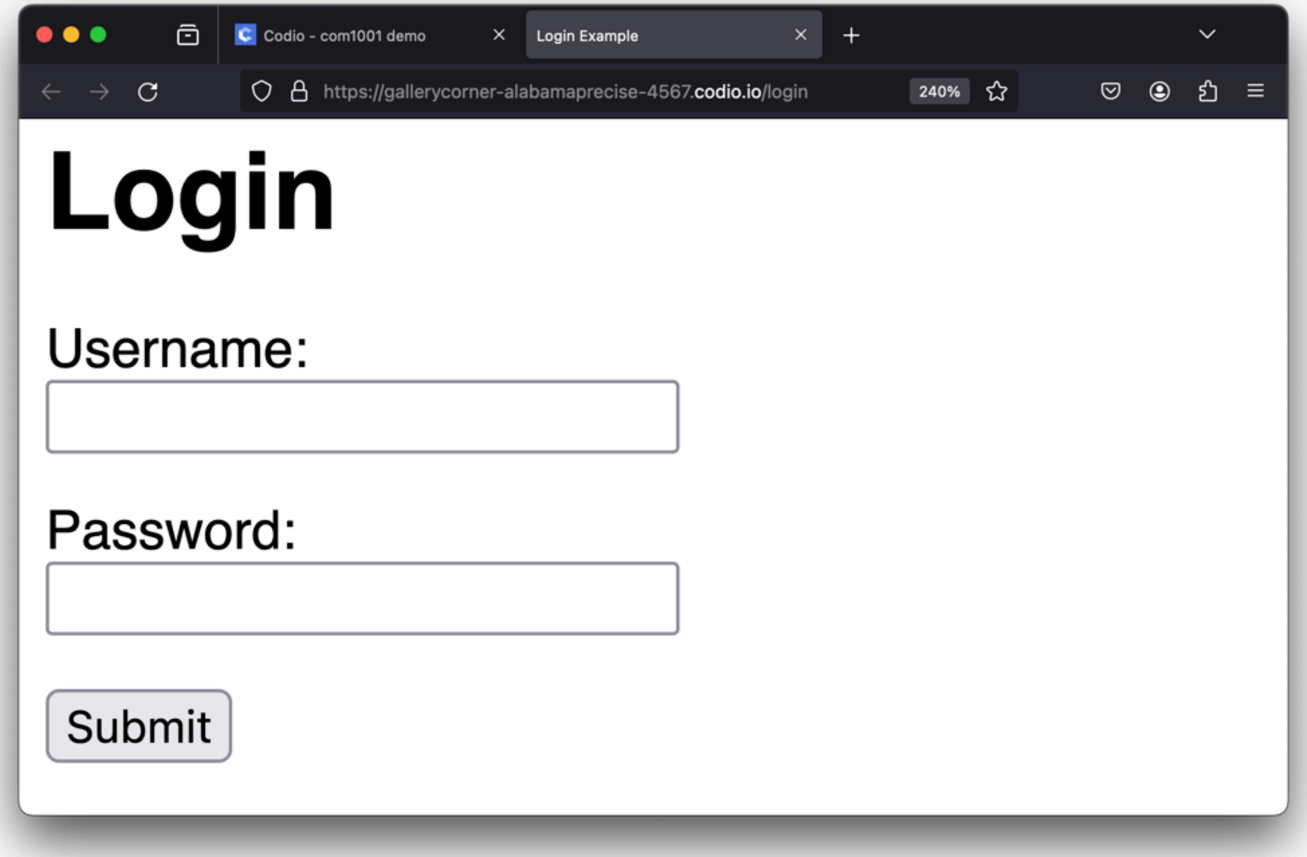
Define a `spec_before` function to run before each test to reset system state (e.g. the contents of the database).

We’ll come back to this…

Unit Testing with `params` and `session`

Recall that pages can be supplied with params that might come from forms, or those that are dependent on values of the session, how do we unit test those?

Let's return to `login_example` ...



Unit Testing with `params`

```
describe "POST /login" do
  context "with no login details" do
    it "tells the user the details are incorrect" do
      post "/login"
      expect(last_response).to be_ok
      expect(last_response.body).to include("Username/Password combination incorrect")
    end
  end

  context "with incorrect login details" do
    it "tells the user the details are incorrect" do
      post "/login", "username" => "wrong_user", "password" => "wrong_password"
      expect(last_response).to be_ok
      expect(last_response.body).to include("Username/Password combination incorrect")
    end
  end

  context "with correct login details" do
    it "redirects to the secure area page" do
      post "/login", "username" => "user", "password" => "pass"
      expect(last_response).to be_redirect
      expect(last_response.location).to end_with("/")
    end
  end
end
```

It's pretty easy, in fact – we just supply the key value pairs as a list after the call to get or post.

Note how it's now super easy to run rspec and check that the form is working correctly without having to type inputs in manually each time.

Unit Testing with `session`

```
describe "GET /" do
  context "when not logged in" do
    it "asks the user to log in" do
      get "/"
      # equivalent to: expect(last_response.status).to be(302)
      expect(last_response).to be_redirect
      expect(last_response.location).to end_with("/login")
    end
  end

  context "when logged in" do
    it "displays the welcome message" do
      get "/", {}, { "rack.session" => { logged_in: true } }
      expect(last_response).to be_ok
      expect(last_response.body).to include("Welcome to the Secure Area")
    end
  end
end
```

It's similar but slightly more complicated for directly setting values of the session to test different parts of the code. This is the format we need to use.

Live Demonstration:

Testing `login`

(from the COM1001 GitHub repository)

Featuring:

- Running RSpec with different directories
- Testing with `params` and `session`
- When tests fail

Integration Tests

Tests that involve interaction with the database may fall under the category of **integration tests**.

Any tests involving the database need to ensure the database is reset to how it was beforehand.

```
def spec_before
  Player.dataset.delete
end
```

football_players/spec/spec_functions.rb

In the football players example, we write a `spec_before` function in `spec_functions.rb` to clear all data in the database before each test

Live Demonstration:

Testing `week3/football_players_example`

(from the COM1001 GitHub repository)