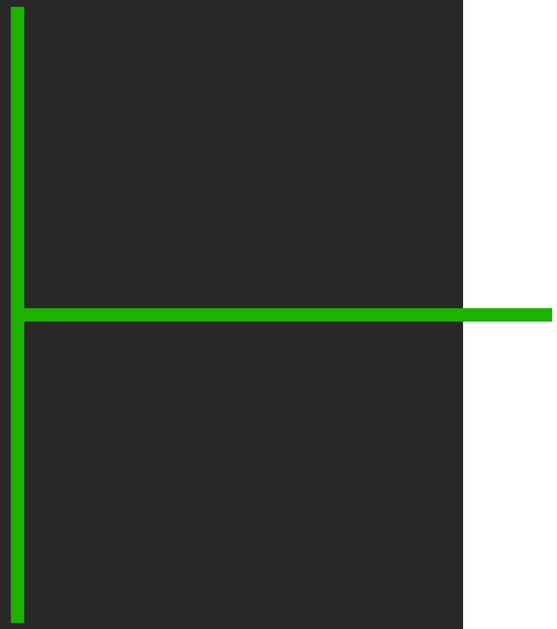
University of Sheffield

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

Unit and Integration Testing of Web Applications with RSpec

Unit Testing Hello World

```
require "rspec"
require "rack/test"

require_relative "hello_world"

def app
  Sinatra::Application
end

RSpec.configure do |config|
  config.include Rack::Test::Methods
end

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"
```



First things first, there's some configuration to do. Shortly we will see how all of this can be put into a `spec_helper.rb` file that is automatically included, so that we don't have to write it out for each individual RSpec

```ruby
def app
  Sinatra::Application
end

RSpec.configure do |config|
  config.include Rack::Test::Methods
end

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
```

getting-started/hello_world_example/hello_world_spec.rb

This RSpec makes two checks: first that this route gives with a **200** (OK) HTTP response, and secondly the content is equal to "`Hello, World!`"

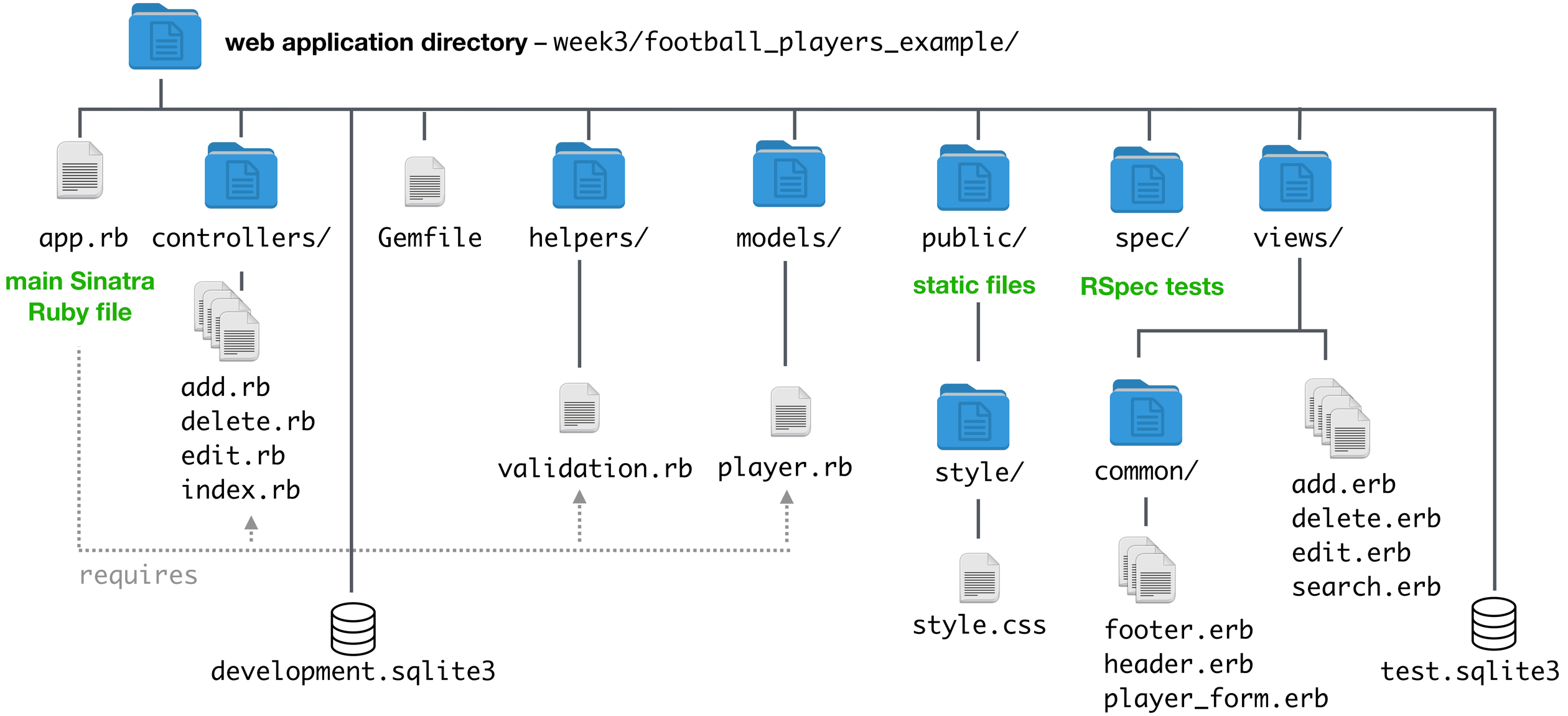
```ruby
def app
  Sinatra::Application
end

RSpec.configure do |config|
  config.include Rack::Test::Methods
end

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
```

getting-started/hello_world_example/hello_world_spec.rb

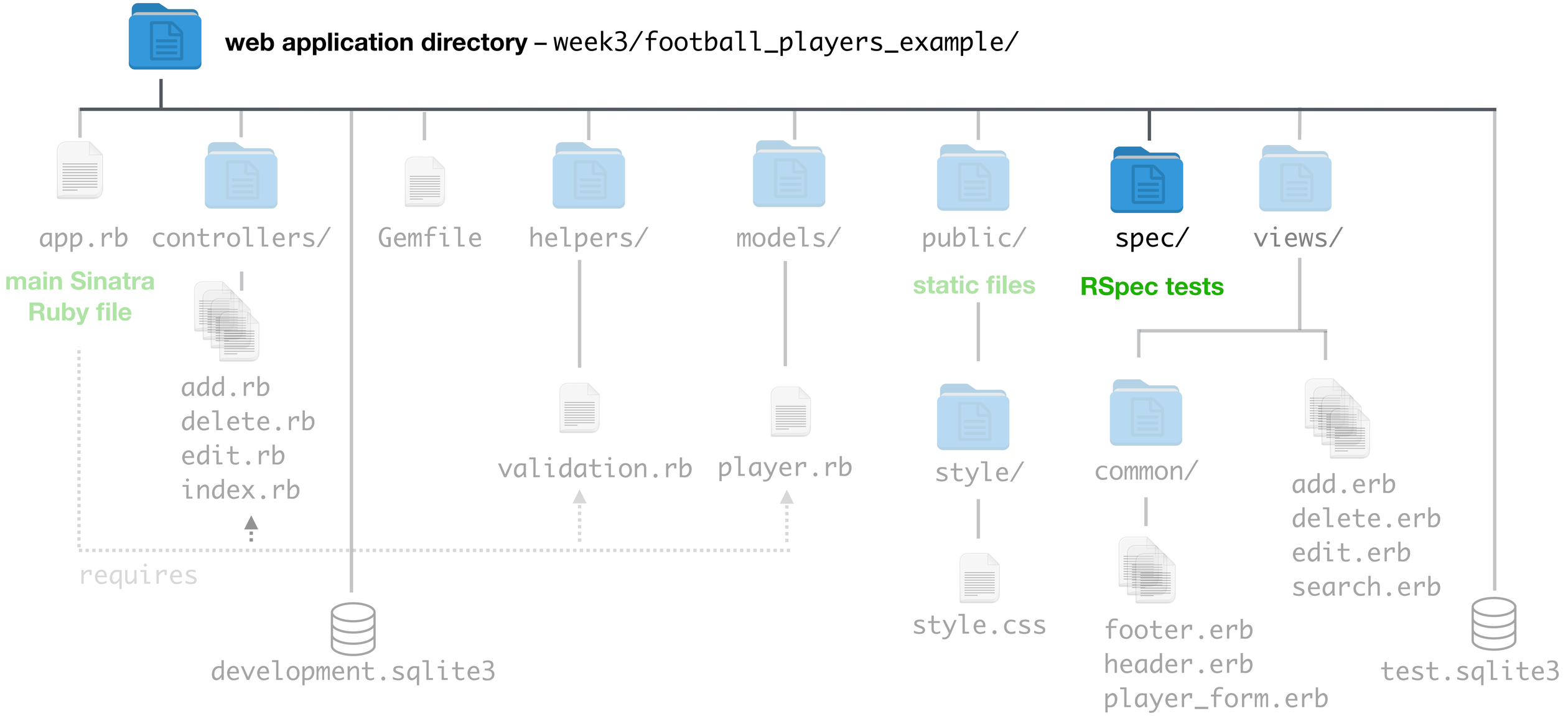
This is how we cause the web page to be “loaded” for the purposes of the test. It happens behind the scenes – we don’t see it in a web browser. Essentially we’re calling the Sinatra route as though it were a method. (If it were a post route we’d be calling `post` rather than `get`.)

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.

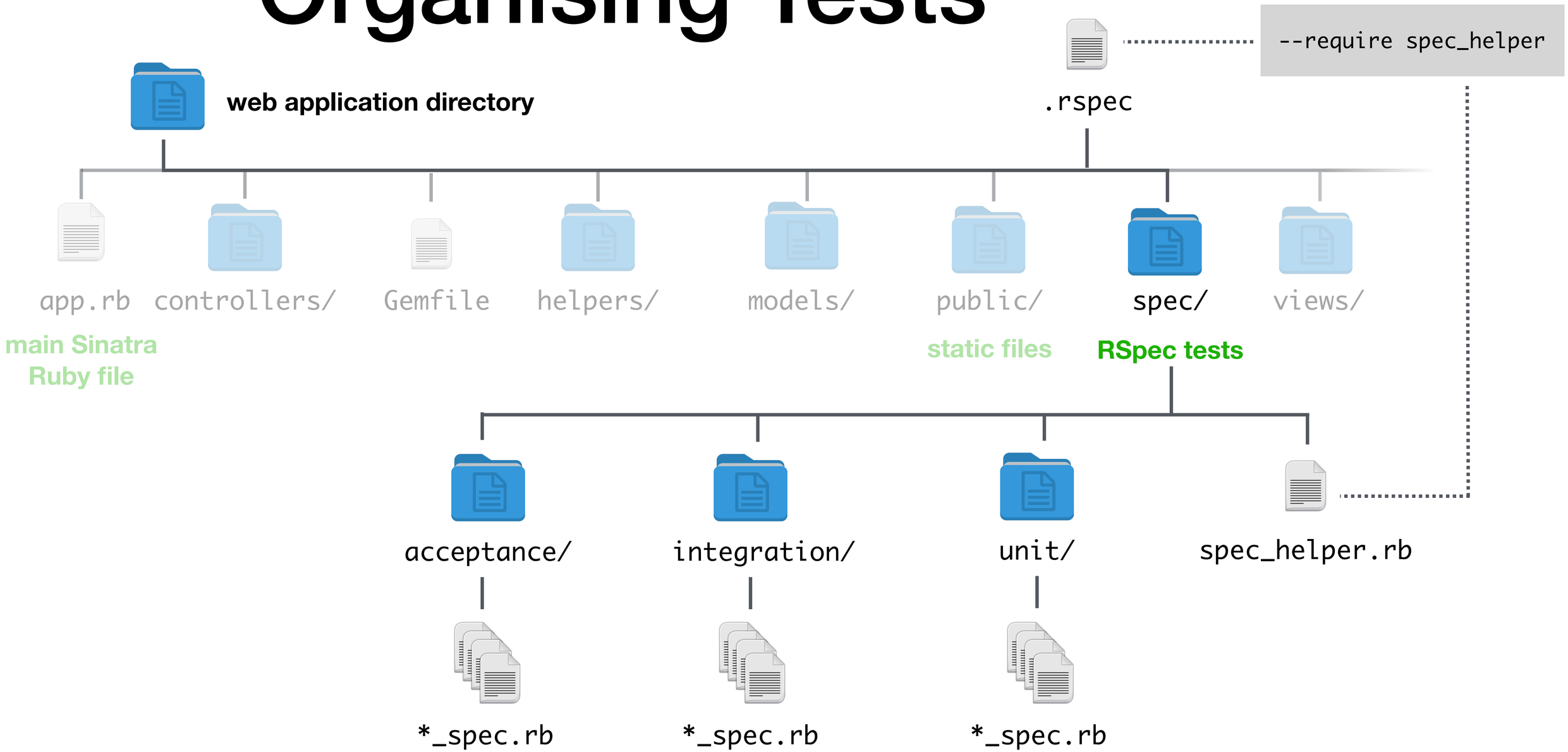
Web Application File Structure



Web Application File Structure



Organising Tests



spec_helper.rb

```ruby
# SimpleCov
require "simplecov"
SimpleCov.start do
  add_filter "/spec/"
end
SimpleCov.coverage_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
```

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

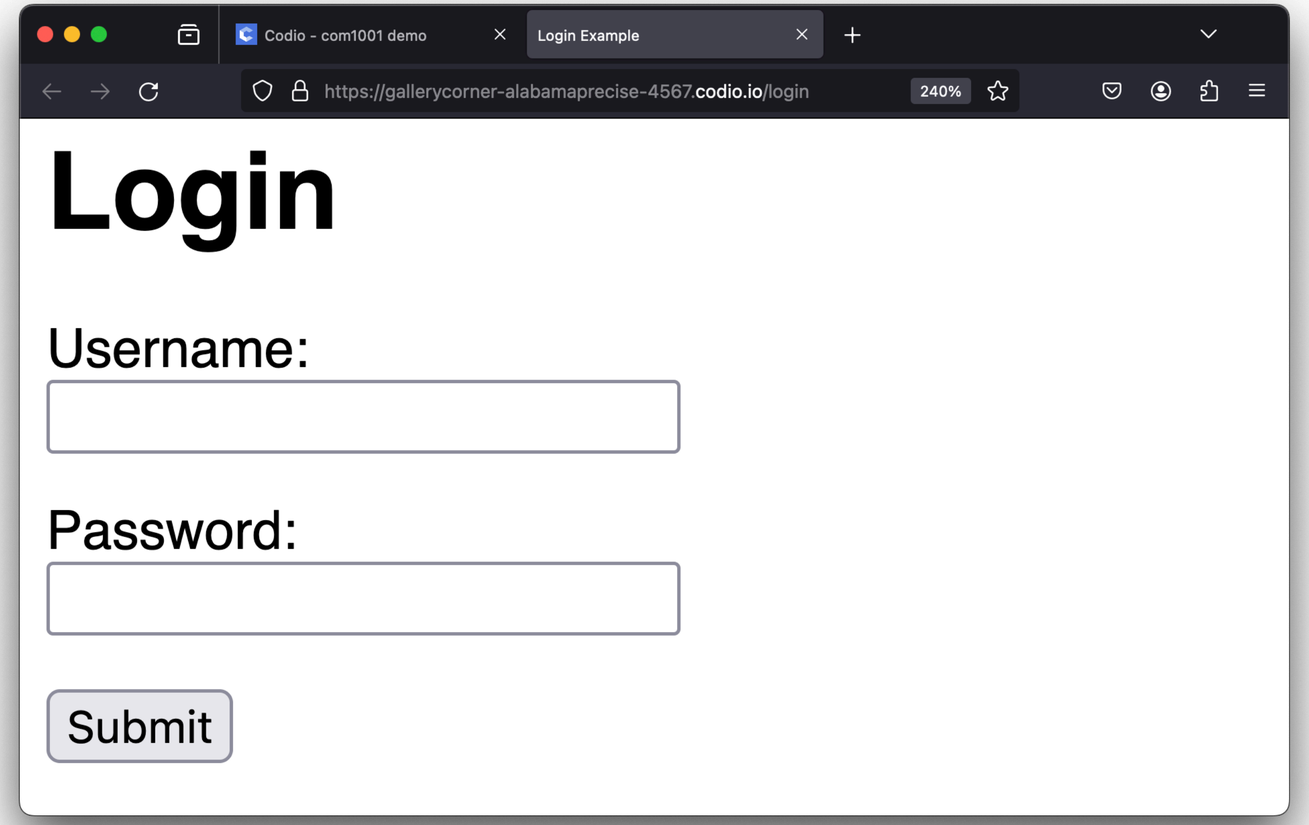
Just copy the one in the materials folder for this week and use it in your team project – there is no need to understand everything it does. However, if you want to know more, ask me…

If you put a file called `.rspec` file in your root web application directory with the contents "`--require spec_helper`", `spec_helper.rb` will be executed first by your rspec files without the need to explicitly write a require line at the top of each spec file.

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 `week2/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 `week2/login_example`

(from the COM1001 GitHub repository)

Featuring:

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

Integration Tests

Any tests with pages that interact a database are technically integration tests.

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

Also they live in `spec/integration/` as opposed to `spec/unit/`

```ruby
# 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

  # before each test is run, delete all records in the Player table
  config.before do
    Player.dataset.delete
  end
end
```

week3/football_players_example/spec/spec_helper.rb

In the football players example, we set spec_helper to clear all data in the database before each test

Example of an integration test

```ruby
describe "POST /add" do
  context "when the parameters are valid" do
    it "adds the player" do
      post "/add", "first_name" => "Bukayo", "surname" => "Saka", "gender" => "M", "club" => "Arsenal",
                   "country" => "England", "position" => "Midfield", "date_of_birth" => "2001-09-05"
      player = Player.first
      expect(player).not_to be_nil # check the record has been added
      expect(player.first_name).to eq("Bukayo")
    end

    # ...
  end
end
```

week3/football_players_example/spec/integration/add_controller_spec.rb

Live Demonstration:

Testing `week3/football_players_example`

(from the COM1001 GitHub repository)