Part 4: Introducing Cucumber

Cucumber is a remarkable tool for writing high-level integration and acceptance tests, terms with which you're already familiar. We'll learn much more about Cucumber later, but for now we will use it to *drive* the development of your app's code.

Just as you used RSpec to "drive" the creation of the class's methods, you'll next use Cucumber to drive the creation of the SaaS code.

Normally, the cycle would be:

1. Use Cucumber scenario to express end-to-end behavior of a scenario
2. As you start writing the code to make each step of the scenario pass, use RSpec to drive the creation of that step's code
3. Repeat until all scenario steps are passing green

In this assignment we're skipping the middle step since the goal is to give you an overview of all parts of the process. Also, in the first step, for your own apps you'd be creating the Cucumber scenarios yourself; in this assignment we've provided them for you.

Cucumber lets you express integration-test scenarios, which you'll find in the `features` directory in `.feature` files. You'll also see a `step_definitions` subdirectory with a single file `game_steps.rb`, containing the code used when each "step" in a scenario is executed as part of the test.

As an integration testing tool, Cucumber can be used to test almost any kind of software system as long as there is a way to *simulate* the system and a way to *inspect* the system's behavior. You select a *back end* for Cucumber based on how the end system is to be simulated and inspected.

Since a SaaS server is simulated by issuing HTTP requests, and its behavior can be inspected by looking at the HTML pages served, we configure Cucumber to use Capybara, a Ruby-based browser simulator that includes a domain-specific language for simulating browser actions and inspecting the SaaS server's responses to those actions.

Self Check Questions

▶ Read the section on "Using Capybara with Cucumber" on Capybara's home page. Which step definitions use Capybara to simulate the server as a browser would? Which step definitions use Capybara to inspect the app's response to the stimulus?

▶ Looking at `features/guess.feature`, what is the role of the three lines following the "Feature:" heading?

▶ In the same file, looking at the scenario step `Given I start a new game with word "garply"`, what lines in `game_steps.rb` will be invoked when Cucumber tries to execute this step, and what is the role of the string `"garply"` in the step?

Get your first scenario to pass

We'll first get the "I start a new game" scenario to pass; you'll then use the same techniques to make the other scenarios pass, thereby completing the app. So take a look at the step definition for "I start a new game with word...".

You already saw that you can load the new game page, but get an error when clicking the button for actually creating a new game. You'll now reproduce this behavior with a Cuke scenario.

Self Check Question

▶ When the "browser simulator" in Capybara issues the `visit '/new'` request, Capybara will do an HTTP GET to the partial URL `/new` on the app. Why do you think `visit` always does a GET, rather than giving the option to do either a GET or a POST in a given step?

Run the "new game" scenario with:

```
$ cucumber features/start_new_game.feature
```

If you get an error about Cucumber like this one, just follow the advice and run `bundle install` first.

```
~/workspace/hw-sinatra-saas-wordguesser (master) $ cucumber features/start_new_game.feature
Could not find proper version of cucumber (2.0.0) in any of the sources
Run `bundle install` to install missing gems.
```

The scenario fails because the `<form>` tag in `views/new.erb` is incorrect and incomplete in the information that tells the browser what URL to post the form to. Based on the table of routes we developed in an earlier section, fill in the `<form>` tag's attributes appropriately. You can inspect what happens for various routes in app.rb, but you don't need to edit this file yet. (Hint: if you get stuck, take a look at `show.erb` (at the bottom) for a similar example of a filled in form tag.)

The create-new-game code in the Sinatra app should do the following:

- Call the WordGuesserGame class method `get_random_word`
- Create a new instance of WordGuesserGame using that word
- Redirect the browser to the `show` action

View how these steps are actualized in the app.rb file under the `post /create do` route.

Now stage and commit all files locally, then `git push heroku master` to deploy to Heroku again and manually verify this improved behavior.

Self Check Question

▶ What is the significance of using `Given` vs. `When` vs. `Then` in the feature file? What happens if you switch them around? Conduct a simple experiment to find out, then confirm your results by using Google.

Develop the scenario for guessing a letter

For this scenario, in `features/guess.feature`, we've already provided a correct `show.erb` HTML file that submits the player's guess to the `guess` action. You already have a `WordGuesserGame#guess` instance method that has the needed functionality.

Self Check Question

▶ In `game_steps.rb`, look at the code for "I start a new game..." step, and in particular the `stub_request` command. Given the hint that that command is provided by a Gem (library) called `webmock`, what's going on with that line, and why is it needed? (Use Google if needed.)
The special Sinatra hash `params[]` has a key-value pair for each nonblank field on a submitted form: the key is the symbolized `name` attribute of the form field and the value is what the user typed into that field, or in the case of a checkbox or radiobutton, the browser-specified values indicating if it's checked or unchecked. ("Symbolized" means the string is converted to a symbol, so `"foo"` becomes `:foo`.)

Self Check Question

▶ In your Sinatra code for processing a guess, what expression would you use to extract *just the first character* of what the user typed in the letter-guess field of the form in `show.erb`? **CAUTION:** if the user typed nothing, there won't be any matching key in `params[]`, so dereferencing the form field will give `nil`. In that case, your code should return the empty string rather than an error.

In the `guess` code in the Sinatra app.rb file, you should:

- Extract the letter submitted on the form. (given above and in the code for you)
- Use that letter as a guess on the current game. (add this code in)
- Redirect to the `show` action so the player can see the result of their guess. (done for you as well)

While you're here, read the comments in the file. They give clues for future steps in this assignment.

When finished adding that code, verify that all the steps in `features/guess.feature` now pass by running cucumber for that .feature file.

- Debugging tip: The Capybara command `save_and_open_page` placed in a step definition will cause the step to open a Web browser window showing what the page looks like at that point in the scenario. The functionality is

provided in part by a gem called `launchy` which is in the Gemfile.

Next: Part 5 - Corner Cases