Module 7
Cucumber, Capybara

CS W169A: Software Engineering
John Yang | Summer 2020

1 Overview

User stories are a core primitive for Behavior Driven Development, and having a domain language that allows us to implement and capture this behavior in a technical, actionable manner is important to verifying the correctness of our application according to customer needs. That's where Cucumber and Capybara come in.

Cucumber allows us to specify high scenario steps. Statements like "Given I am logged in as 'Nancy'" or "When I purchase the item 'Textbook'" may require several steps to implement with Cucumber. In this discussion, we hope to familiarize you with some of the basics of using this tool. Getting used to Cucumber and Capybara can be tricky, so we'll introduce some power tools and practices you can use to improve the expressiveness and power of your own Cucumber scenarios.

2 Cucumber

Cucumber is a tool for running automated tests that are written in plain language. The advantages of Cucumber are mainly 1. They are very human readable 2. The language is very expressive and can capture a variety of scenarios. You might recall from lecture, there a 3 Cucumber skills we particularly care about:

1. Writing steps and step defs - Using Cucumber syntax, regular expressions
2. Identifying and Interacting with input controls on a page (i.e. forms, fields, links, buttons) - Using CSS (find/name elements), Capybara methods (fill_in, click, follow, select)
3. Check page output - Using expectations (similar to assertions) to check results

As a refresher, the below is an implementation of a User Story, written in the Connextra format, as a Cucumber Scenario. The first block is the user story, while the second block is the Cucumber implementation.

```
Scenario: get free movie ticket on my birthday
    Given I am logged in as "Armando Fox"
    And my birthday is set to "May 12"
    ...
```

```
Given /^my birthday is set to "(.*")"/ do |date|
    @customer.update_attributes!(:birthday
        => Date.parse(date))
end
```

Now, let's take a look at some recommended conventions for writing Cucumber scenarios.

2.1 Use Instance Variables to Carry State across Steps

Cuke scenarios execute in the context of an object called a World. When you reference instance variables in a Cuke step definition, they are instance variables of that object, so they survive across steps (but not across scenarios). This lets you carry state across Cucumber steps.

For example, in an extended RottenPotatoes site, a user has a collection of favorite movies. The URI for a logged-in user's landing page is /users/:id (where :id is a primary key) and the URI for viewing that user's favorite collection is /users/:id/favorites.

How would you use instance variables in Cuke steps to create the following step defs?

```
Given I am logged in as user "An Ju"
And my favorite collection contains the item "The Bourne Identity"
...
```

Solution: One possibility is to use an instance variable to remember the logged-in user. For the above scenario:

```
Given /^I am logged in as user "(.*)"$/ do |username|
    login_as!(username) # simulate logging in
    @current_user = User.find_by_name!(username) # remember logged-in user
end
```

You could imagine, for another scenario such as checking whether a cart contains an item, we'd similarly write:

```
Given /^my cart contains the item "(.*)"$/ do |item|
    @cart = @current_user.cart
    ...
end
```

In both these scenarios, we use instance variables (username, item) to persist state throughout all steps.

2.2 Scope Steps to Specific Divs / Elements

This step is particularly relevant when implementing "Then I should see" clauses in your user stories.

Suppose we redesign the RottenPotatoes home page to always show "top rated movies" on the top part of the page, and show user-specific content on the bottom part. Here's a happy path scenario for "search for a movie by year" (line numbers added for clarity only):

```
1 Given the following movies exist:
2  | title       | year | rating |
3  | Bridesmaids | 2012 | PG-13  |
4  | Cloud Atlas | 2011 | R      |
5  | The Help    | 2011 | R      |
6 When I search for movies released in "2011"
7 Then I should see the following: Cloud Atlas, The Help
8 But I should not see the following: Bridesmaids
```

2.2.1 Warm Up

Fill in the step def below by arranging for movies to contain a list of movie titles so the step def will run:

```
Then /^I should see the following: (.*)$/ do |list|
    movies = ______________________________
    movies.each do |movie|
        steps %Q{Then I should see "#{movie}"}
    end
end
```

A couple of hints:

1. The documentation for using nested steps in Cucumber can be found here.
2. `%Q{...}` can be used to specify a string literal with double-quote semantics,that is, you can interpolate variables but you can include double-quote characters without escaping them.

The blank can be filled with `movies = list.split(/*,*/)`

2.2.2 Scoping

There are some potential problems with our Cucumber scenario above. Suppose that the "top rated movies" pane happens to include "Bridesmaids". Then, even if our search code is correct (movies released in 2011), "Bridesmaids" will still show in our results as a "top rated" movie, so line 8 will fail even if our search code is correct, a false positive!

Imagine an alternative scenario. If "top rated movies" already shows Cloud Atlas, or The Help, line 7 could pass even if the search code for movies made in 2011 doesn't work, a false negative!

What's the commonality between both of these pitfalls? When we check for the presence of an element on a web page that has the potential to appear multiple times for multiple different reasons (in this case, a movie could be present because it is top rated, from our search query, or both!), our test may accidentally pass or fail for reasons unrelated to what it's testing. In this situation, we're testing our search code, but the "top-rated" section is interfering with the accuracy of our test. This kind of testing bug shows up a lot, especially when you're checking for something that appears frequently, such as a logged-in user's name, which tends to appear on every page view, potentially multiple times per page!

The solution is to structurally separate different parts of the page and make each chunk easily and uniquely identifiable; for example, the search results might be displayed in a div whose ID is search_results. Then, we can specify which part of the page our test applies to by modifying our step to include a "within" statement as follows:

```
Then I should see "Cloud Atlas" within "div#search_results"
```

(You can find this step definition in `web_steps.rb`. As you can see, it's a thin wrapper around Capybara's `contain`, and the argument to `within` can be any CSS selector. The Within helper is provided by Capybara and is a good example of using closures in Ruby.)

Now it's your turn! Use this scoped version as a building block to fix the step def for step 7, without changing the phrasing of the step itself in the scenario. (Hint: consider reusing the existing stepdef in `web_steps.rb`)

Solution:

```
Then /^I should see the following: (.*)$/ do |list|
    movies = list.split(/\s*,\s*/)
    movies.each do |movie|
        steps %Q{Then I should see "#{movie}" within "#search_results"}
    end
end
```

We can also change just two lines in the step def to fix the same problem in step 7 for step 8, too!

```
Then /^I should (not)?see the following: (.*)$/ do |neg, list|
    movies = list.split(/\s*,\s*/)
    movies.each do |movie|
        step %Q{Then I should #{neg}see "#{movie}" within"#search_results"}
    end
end
```

3 Capybara

Besides `contain`, which looks at element text, Capybara provides a large number of "matchers" for examining the page. For example, if you just want to check for the presence of a page element (say, an error message), you can write

```
expect(page).to have_selector("p.error", :text => 'An error occurred')
```

Note: The actual method defined in Capybara is `has_selector?`, because RSpec is smart enough to interpret `have_selector` as a call to `has_selector?`.

`page` is a Capybara method that returns a representation of the page object, and `page.body` is a string containing the raw HTML of the full page text.

3.1 Documentation

Here's an overall quick reference (README) and a full list of the matchers Capybara adds to RSpec. The spreewald gem contains a collection of highly useful Cucumber steps, many based on Capybara's advanced XPath capabilities, for testing various other aspects of the page, such as whether a set of strings appears in a certain order.

The most general and most powerful Capybara method is `find()`, which lets you use an arbitrary XPath expression, and its wrapper `has_xpath?`, which turns the result of `find()` into a Boolean true if the result matches any elements and nil otherwise. XPath lets you not only look for elements nested in other elements, but also match on elements having specific attributes or specific text:

```
page.find(:xpath, "//table/tr[@class='adult']")
    # return all table rows with css class "adult"

page.should have_xpath("//title[contains(text(),'Rotten')]")
    # ensure page has <title> element whose text includes "Rotten"
```

The online tool XPathTester lets you copy-and-paste a chunk of HTML or XML and then test various XPath expressions against it.

3.2 XPath and Step Defs

Use XPath to fill in the following step def that checks whether a given option is selected in a dropdown menu.

```
Then "January" should be selected in the "Month" menu # Implement this below

Then /^"(.*)" should be selected in the "(.*)" menu$/ do |choice, menu|
    if page.find(:xpath, "//select[@id='#{menu}'").blank?
        # handle <label for="month">Month</label>...
        # <select id="month">...</select>
        menu_id = ________________________________________ # Your code here
    else
        menu_id = menu
    end
    page.should have_path(________________________________) # Your code here
```

Solution:

1. menu_id = page.find(:xpath,"//label[contains(text(), ’menu’").first[’for’]
2. page.should have_path("//select[@id=’menu_id’]/option[@selected=’selected’][contains(text(),’choice’)]")