Lab: Testing

Credits: This lab was originally compiled by Suwanna Xanthavanij. Some of the material in the second part of this manual comes from Ruby Programming/Unit testing, available at https://en.wikibooks.org/ wiki/Ruby_Programming/Unit_testing. Some of the material in the first part of this lab manual comes from the course website and the Cucumber Cookbook.

Background on Unit Testing: Unit testing helps us catch errors early in the development process, if we write appropriate and useful tests. Like other languages such as Java, Ruby provides a framework in its standard library for setting up, organizing, and running tests called Minitest. There are other very popular testing frameworks such as Rspec and Cucumber that serve different purposes. Minitest provides three basic functionalities:

1. A way to define basic pass/fail tests.
2. A way to gather related tests together and run them as a group.
3. Tools for running single tests or whole groups of tests.

Rails makes it very easy to write and run tests. It starts by producing skeleton test code when you create your models and controllers. By simply running your Rails tests you can ensure your code adheres to the desired functionality even after refactoring. Rails tests can also simulate browser requests, so you can test your application's responses to HTTP requests without having to test manually through your browser.

Background on User Acceptance Testing: Whereas unit testing is technical and developer focused, user acceptance tests (UATs) focus on the behavior of the system from the user's point of view. In behaviordriven development we combine test-first with black-box UAT tests written in English rather than a specific programming language to drive the software development process. We iteratively construct a system according to the user's requirements, focusing on the features that give the customer immediate value and insight into the system's behavior. Today we will see how the extremely popular Cucumber tool can be used for UATs for Rails applications.

Although unit tests are more fundamental than UATs, in terms of development processes, I (Matt) believe you should begin with UATs to drive development and then make sure your unit tests provide good coverage of what you've written during the refactoring phase of BDD.

User Acceptance Testing

Imagine you're building a simple app to help teachers keep track of the term projects that groups of students in a class are performing during a semester. You've already done some groundwork, creating a new Rails app, adding scaffolds for students, projects, and also a Devise user model.

1. Create a Rails project called project tracker using the PostgreSQL database.
2. Generate the following scaffolds:
   (a) Project with name and url as strings
   (b) Student with studentid and name as strings

3. Setup the relationship between projects and students as a **many-to-many** relationship. If you recall, a many-to-many relationship between two tables is created by using an intermediate join table. ** **Hint** : Create a model for the intermediate table.
4. Add and install the Devise gem. Check that you can sign up, sign in, and sign out.
5. Add the Cucumber gem to your Gemfile along with some other useful libraries:

```
group :test do
    gem 'cucumber-rails', :require => false
    # database_cleaner is not required, but highly recommended
    gem 'database_cleaner'
    gem 'factory_bot_rails'
    gem 'launchy'
    gem 'rspec-rails'
end
```

   After editing your Gemfile, you'll want to update bundler.
6. Next, install the Cucumber framework files in your project and then try to run the Cucumber tests (even though we have no features defined yet):

```
$ rails generate cucumber:install
$ rake cucumber
```

7. As mentioned above, suppose after discussing with our customer, we have decided to work on allowing the user to add students to a project in the project view. Let's add our first feature specification, as a new file features/student.feature:

```
Feature: Student

In order to keep track of which students are working on which projects, as a
teacher, I want to associate students with projects.

Scenario: Add student to a project
    A teacher should be able to add students to a project.

    Given I am a teacher And there is a project
    And I want to add a student to the project
    And I am signed in
    When I visit the projects page
    Then I should see a link for the project
    When I click the link for the project
    Then I should see the details of my project
    And I should see a form to add a student
    When I submit the form
    Then I should see the details of my project
    And I should see the student added to the project
```

Notice the style: some “Given” statements, followed by a sequence of “When” and then “Then” statements. The Given/When/Then format is defined by the Gherkin language.

8. Now run you Cucumber tests again:
   ```
   $ rake cucumber
   ```
   It doesn’t work, but of course we haven’t implemented anything yet, so we don’t expect it to work yet.

9. The first thing we should do is to copy the step definitions, each with the “pending” action, into a new file features/step definitions/student steps.rb:
   ```
       Given(/^I am a teacher$/) do
           pending # express the regexp above with the code you wish you had
       End
   ```

Then run Cucumber again. Now you can see that it runs the first step written above. However, there is nothing implementing inside that step yet.

10. Now we need to implement each step. When you first begin you’ll find you’re spending a lot more time writing tests than writing code, but after just a few tests you’ll feel the productivity start to increase and accelerate. Let’s try this for the first step in student steps.rb:

    ```
    Given(/^I am a teacher$/) do
        @user = FactoryBot.create :teacher
    end
    ```

    Here we set an instance variable that will be available in later steps to be the result of asking FactoryBot to create a new saved instance using the teacher factory.

11. So we create a new file, features/support/factories.rb with the teacher factory definition inside:
    ```
        FactoryBot.define do
            factory :teacher, class: User do
                email { "joe_teacher@ait.asia" }
                password { "password" }
                password_confirmation { "password" }
            end
        end
    ```
    Wonderful! Now when you run cucumber, the first step should pass, and the second step will be highlighted as pending.

12. Let's do more, implement step by step. Here are the next few steps, which make use of Capybara to simulate the user interaction with the browser and RSpec expectations for what should be observable by the user:

```
Given(/^there is a project$/) do
        @project = FactoryBot.create :project
end
Given(/^I want to add a student to the project$/) do
        @student = FactoryBot.build :student
end

Given(/^I am signed in$/) do
         visit '/users/sign_in'
        fill_in 'Email', with: @user.email
        fill_in 'Password', with: @user.password
        click_button 'Log in'
 end

When(/^I visit the projects page$/) do
        visit '/projects'
end

Then(/^I should see a link for the project$/) do
        expect(page).to have_link('Show', href: project_path(@project))
end
When(/^I click the link for the project$/) do
        find_link('Show', href: project_path(@project)).click
end

Then(/^I should see the details of my project$/) do
        save_and_open_page
        expect(page).to have_content "Name: #{@project.name}"
        expect(page).to have_content "Url: #{@project.url}"
end
Then(/^I should see a form to add a student$/) do
        expect(page).to have_selector('form#new_student')
End
```

13. Notice that for the student, who is not in the database before the test runs, we use the FactoryBot build method rather than create. These steps make use of a couple new factories:

```
        factory :project do
                name { "My favorite project" }
        url { "http://somewhere.com" }
    end

    factory :student do
        name { "Joe Student" }
        studentid { "123456" }
    End
```

14. The next step is to do some work in the application to make the test pass! First we need to make student routes subsidiary to projects in config/routes.rb:
    ```
    resources :projects do
        resources :students
    end
    ```
15. Then we need to add a student creation form to the bottom of the show view for a project:

```
<h2>Students</h2>
<table>
<thead>
<tr>
<th>Student ID</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<% @project.students.each do |student| %>
<tr>
<td><%= student.studentid %></td>
<td><%= student.name %></td>
</tr>
<% end %>
</tbody>
</table>
<br/>
<h2>Add a student:</h2>
<%= form_for([@project, @project.students.build]) do |f| %>
<div class="field">
<%= f.label :studentid %>
<br/>
<%= f.text_field :studentid %>
</div>
<div class="field">
<%= f.label :name %>
<br/>
<%= f.text_field :name %>
```

```
</div>
<div class="actions">
<%= f.submit %>
</div>
<% end %>
```

16. Finally we edit the create action in app/controllers/students controller.rb as follows:

```
before_action :set_project, only: [:create]
def create
        respond_to do |format|
                if @project.students.create(student_params)
                        format.html { redirect_to @project, notice: 'Student was
                successfully created.' }
                        format.json { render :show, status: :created, location: @project }
                else
                        format.html { render :new }
                        format.json { render json: @project.errors, status:
                        :unprocessable_entity }
                end
        end
end
```

Then run Cucumber again. You can see that most steps are green now. However, there are still some steps not implemented yet. Try to find out how to make those steps green. :)

Unit Testing

As I said earlier, the role of unit testing within a BDD process is to help you clean up and improve the quality of your code during the refactoring phase. As you've just gotten your first UAT running in the previous part of the lab, now it's time to get your unit tests working for you at a more detailed level.

1. Assume that you determine these requirements for the entities developed in the previous section:
    (a) The name of a project cannot be null and must be unique.
    (b) Neither the name nor id of a student can be null.
    (c) The name of a student must be unique.
    (d) Anyone can see the student list or show a particular student, but only logged-in users can edit/create/delete students or do anything with projects.

In a test first or test-driven development approach to meeting these requirements, we write failing tests before we code to satisfy the requirments. Then we write the code that makes the test(s) pass. In our case, how to test for these requirements? Don't worry! We can test via Minitest tests for the models and controllers in our application.

2. Look at test/fixtures. You can see *.yml files. These are called YAML-formatted fixtures. They are a very human-friendly way to describe your sample data. Fill in some data in each fixture as described below.
   (a) Set up two Project fixtures in projects.yml, with the following YAML:
   ```
   One:
       id: 1
       name: 'Soi Cats and Dogs'
   two:
       id: 2
       name: 'Sri Lankan Food Lovers'
   ```
   (b) Create similar fixtures in students.yml.
3. Next, go to test/models and then edit the project and student test classes. Let's begin with the project class.
   (a) Name cannot be null
   ```
   test "should validate presence of name" do
       project = Project.new
       assert !project.valid?
       assert_equal ["can't be blank"], project.errors[:name]
   end
   ```
   (b) Name must be unique
   ```
   test "should validate uniqueness of name" do
       old_project = projects(:one)
       project = Project.new name: old_project.name
       assert !project.valid?
       assert_equal ["has already been taken"], project.errors[:name]
   End
   ```
4. Next, edit test/models/student test.rb. Try to test that neither the name nor the price can be null, and also test that the name must be unique.
5. Before running the tests, since we are using the Devise gem, we have to add this line to all of the classes in the test/controllers/ directory:
   ```
   include Devise::Test::IntegrationHelpers
   ```
6. Now, try to run your tests using this command:
   a. `$ rake test`

7. Your tests should be failing. Look at each and try to make them pass! First, you have to go to app/models/ and edit the project model. Add these lines of code:

   ```
   validates_presence_of :name
   validates_uniqueness_of :name
   ```
   Next go to app/models and open the student model. Add appropriate method calls. While you're at it, don't forget to set up the relationship between projects and students.

8. Is it working? OK! Now test access control. First of all, add some data to the User fixture:

```
one:
        id: 1
        email: "email@hotmail.com"
        encrypted_password:
"$2a$10$ZU0x7nDXA1EKbUAjwKTCwegV/yfddZkvfKWEa9/arb5pBQuy/1oFu"
```

9. Next, in a controller test, we can add this line before performing actions requiring a registered user to access:

```
sign_in users(:one)
```

10. Do your tests fail? Good! To fix the tests requiring that only registered users should be able to create, update, or delete projects, add this line to the controller class:

```
before_action :authenticate_user!, :except => [:index, :show]
```

11. Then, try to run the tests again. Eventually, hopefully, you will have no errors left.
12. Last item: do your tests cover 100% of the code you have developed so far? That should be a minimum for such a simple application. In your Gemfile add

    ```
    gem 'simplecov', :require => false, :group => :test
    ```

    then do a bundle install.

13. Ensure that the very top of test/test_helper.rb looks like this (for Rails 6 and 7, parallelize must be removed):

```
require 'simplecov'
SimpleCov.start 'rails' do
        add_filter 'app/channels'
        add_filter 'app/jobs'
end

ENV['RAILS_ENV'] ||= 'test'
require_relative '../config/environment'
require 'rails/test_help'

class ActiveSupport::TestCase
         fixtures :all
End
```

14. Re-run your tests, and check coverage. Add necessary tests to get 100% coverage of your controller and model code. You will have to exclude some files or directories such as channels that are not used in your project.

15. If you have some time left, try doing the same thing with your class project repository, with a simple UAT (perhaps related to users and their roles) that you think you'll need

later in your actual project. Then you should set up GitLab CI/CD to run your tests and finally to show the test coverage and Cucumber scenario success/failure rates.

Congratulations! that’s all for this lab. The thinking behind testing may be confusing at first, but don’t worry — once you develop a rythym it will not be difficult anymore. Practice practice practice!!