Testing in Rails

All Rails projects start with a `test` directory. It contains subdirectories for various kinds of tests.

- *Unit tests* are used to test a particular class. They call methods of the class and check whether the expected response is received.
- *Functional tests* are used to test individual requests made over the web. They test for conditions such as …
    - was the web request successful?
    - was the user redirected to the right page?
    - was the user successfully authenticated?
    - was the correct object stored in the response template?
    - was the appropriate message displayed to the user in the view?
- *Integration tests* test how different parts of the application interact. They can be used to test use cases.
- *Performance tests* are designed for benchmarking and profiling the code. Like functional tests, they can test individual requests. Like integration tests, they can test multiple parts of the application.

What kinds of tests are these?

Philosophy: Put as much as possible in the model. This avoids dependencies between business logic and presentation.

The view can be as complicated as you want, as long as the logic is only to display information to the user.

Fixtures

In order to run tests, you need data to test with. You *could* execute code to set up the objects used in your tests at the beginning of each test.

But this would be a lot of code to write, and the code would just be creating objects. It would be as roundabout as using executable Ruby code to print out the HTML in a view.

Just like we can use .erb files to specify how a view looks, we can use *.yml files* to specify objects that exist when a test starts.

.yml is the extension for YAML files (YAML stands for "YAML ain't markup language").

In a Rails project, the fixtures are stored in the `test/fixtures` directory. Fixtures can refer to each other. Which lines in categories.yml and recipes.yml refer to other fixtures?

Some fixtures are generated automatically.

In this directory, which of the lines in categories.yml and recipes.yml do you think were autogenerated, and which were inserted manually? Why?

The idea is that you can autogenerate a few fixtures, which are instances of objects of the class, and then write ERB code to generate a lot of others that have the same basic format.

When are the autogenerated lines actually generated?

Rails loads fixtures automatically when tests are run.

- It removes any existing objects from the database table that the fixture is an instance of.
- It loads the fixture data into the database table.
- It allows the program to refer to the fixture by name.

What concept that we introduced last week are fixtures an instance of?

How long do we want data to last when we load it into a testing database?

Running tests

We need to set up a test environment explicitly. Let's look at config/database.yml.

How many databases does it reference?

It's important to have a separate test db for a reason we mentioned above. What reason is that?

Unit tests

Unit tests are typically used to test models. Why are they suitable for models?

Let's look at `recipe_test`, which has two tests to determine whether it is possible to create a recipe with all fields blank.

It's good practice to have a unit test for each method in a model.

Each test must include at least one assertion. The assertions should test everything that is likely to break.

Which of the following scenarios should be tested by unit tests?

What else could you test about recipes?

Many kinds of assertions are available.[1]

Assertion	Purpose
`assert( test, [msg] )`	Ensures that `test` is true.
`assert_not( test, [msg] )`	Ensures that `test` is false.

[1] From http://guides.rubyonrails.org/testing.html

Assertion	Purpose
`assert_equal( expected, actual, [msg] )`	Ensures that `expected == actual` is true.
`assert_same( expected, actual, [msg] )`	Ensures that `expected.equal?(actual)` is true.
`assert_nil( obj, [msg] )`	Ensures that `obj.nil?` is true.
`assert_match( regexp, string, [msg] )`	Ensures that a string matches the regular expression.
`assert_raises( exception1, exception2, ... ) { block }`	Ensures that the given block raises one of the given exceptions.
`assert_instance_of( class, obj, [bmsg] )`	Ensures that `obj` is an instance of `class`.
`flunk( [msg] )`	Ensures failure. This is useful to explicitly mark a test that isn't finished yet.

Almost all of these also have negative versions, e.g., **`assert_not_equal(`**...**`)`**.

Functional tests

Let's run a functional test, e.g., `category_creation_flow_test.rb`.

- One visits the new category page, fills in the form and checks to see if the new category can be displayed
- One visits the categories page and checks whether the application is displaying the correct title.
- One visits the new recipe page, fills in the form and checks to see if the new recipe can be displayed

Let's run a functional test, e.g., `category_creation_flow_test.rb`.

```
class RecipesControllerTest < ActionController::TestCase
  setup do
```

```
    @recipe = recipes(:one)
  end

  test "should get index" do
    get :index
    assert_response :success
    assert_not_nil assigns(:recipes)
  end

  test "should get new" do
    get :new
    assert_response :success
  end

  test "should create recipe" do
    assert_difference('Recipe.count') do
      post :create, recipe: { description: @recipe.description,
instructions: @recipe.instructions, title: @recipe.title }
    end

    assert_redirected_to recipe_path(assigns(:recipe))
  end

  test "should show recipe" do
    get :show, id: @recipe
    assert_response :success
  end

  test "should get edit" do
    get :edit, id: @recipe
    assert_response :success
  end

  test "should update recipe" do
    put :update, id: @recipe, recipe: { description:
@recipe.description, instructions: @recipe.instructions, title:
@recipe.title }
    assert_redirected_to recipe_path(assigns(:recipe))
  end
```

Make sure something is assigned

Make sure count changes

```
  test "should destroy recipe" do
    assert_difference('Recipe.count', -1) do
      delete :destroy, id: @recipe
    end

    assert_redirected_to recipes_path
  end
end
```

Let's look at a functional test, e.g., `recipe_creation_flow_test.rb`. This test uses the following commands provided by the Capybara gem [SaaS §7.5]:

- **`visit`**: navigate the Capybara driver to that particular page. Note that now the application is being tested from a browser perspective
- **`fill_in`**: fill in the particular form field
- **`select`**: for the recipe select a particular category from the category dropdown
- **`click_button`**: actuate a button

What kinds of functional tests would be good to have?

Integration tests

Integration tests are used to test interactions among controllers.

No integration tests are auto-generated. But, Rails provides a generator to get you started:

```
require 'test_helper'

class UserFlowsTest < ActionDispatch::IntegrationTest
  # test "the truth" do
  #   assert true
  # end
end
```

Look at the "**`should use layout`**" test in `categories_controller_integration_test.rb`. This makes sure you can get a page without a 404 error, etc. We can change the title of the page in `application.html.erb` to be something other than "Cookbook", and see that the test fails.

(When this test runs, it fails because of a wrong title on the cookbook.)

Notice that almost every test requires `test/test_helper.rb`. This is included as a mixin, so that the functionality is available to every test.

A good description of almost everything we have covered today is in "A Guide to Testing Rails Applications." Mocks and stubs are an alternate way of setting up tests, which allows things to be tested that do not yet exist, or are too expensive or destructive to access.

- Stubs are objects where, if someone calls this method on you, this is what you're going to respond with — and that's it. It's a stand-in for some other object. Just returns canned data.
- Mocks are things like supporting services: if you are testing an emailer, it would normally send email, but in this case it doesn't, and acts like it did. "Let's not, & say we did." They verify that a particular method was called.

A readable description of mocks and stubs can be found at Code with Jason; Jesus Castello has a good video description.

The SaaS text covers them in Sections 8.3 and 8.4. It also covers fixtures in Section 8.6. Fixtures can be brittle; if you change the schema of a table, you have to change the associated fixtures—and possibly other fixtures that reference the fixtures you changed.