[Lesson 8.1: Test-Driven Development](#h.abx9llqc41u)

[Gem of the Day](#h.w5wenyjuxws3)

[What is Test-Driven Development?](#h.fsf8xuy30bjh)

[Testing in Rails](#h.kbfirkonf5zq)

[Test Environment](#h.noqg6pdbzzfd)

[Default Folders and Tests](#h.rs7l47y13u5m)

[Model Tests: Bus Me](#h.l85zsoz2pnn)

[Writing Our First Tests](#h.ovfw40fie4pl)

[Available Assertions](#h.haii17l7rfj8)

[Functional Tests](#h.4nuuskzdeur9)

[Functional Tests With AJAX Tasks](#h.7nn2ayx3opwu)

[GitHub Badges Using Travis CI](#h.h123o2e7voho)

[Homework](#h.ar7yoho6f64s)

# Lesson 8.1: Test-Driven Development

## Gem of the Day

Rspec-Rails <https://github.com/rspec/rspec-rails>

Integration testing with Capybara <https://github.com/jnicklas/capybara>

## What is Test-Driven Development?

**Test-driven development (TDD)** is a software development process that relies on the repetition of a very short development cycle:

1. Write an automated test case that defines a desired improvement or new function.
2. Run all tests to see if the new one fails (or any others).
3. Produce the minimum amount of code to pass that test.
4. Run all tests again to make sure all pass.
5. Refactor the new code to acceptable standards.
6. Repeat.

**Test automation** is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes. Test automation can automate some repetitive but necessary tasks or add additional testing that would be difficult to perform manually.

## Testing in Rails

* Rails makes it super easy to write your tests. It starts by producing skeleton test code while you are creating your models and controllers.
* By simply running your Rails tests you can ensure your code adheres to the desired functionality even after some major code refactoring.
* Rails tests can also simulate browser requests and thus you can test your application's response without having to test it through your browser.

### Test Environment

By default, every Rails application has three environments: development, test, and production. The database for each one of them is configured in config/database.yml.

A dedicated test database allows you to set up and interact with test data in isolation. This way your tests can mangle test data with confidence, without worrying about the data in the development or production databases.

Also, each environment's configuration can be modified similarly. In this case, we can modify our test environment by changing the options found in config/environments/test.rb.

### Default Folders and Tests

The models directory is meant to hold tests for your models, the controllers directory is meant to hold tests for your controllers and the integration directory is meant to hold tests that involve any number of controllers interacting. There is also a directory for testing your mailers and one for testing view helpers.

Fixtures are a way of organizing test data; they reside in the fixtures directory. Fixtures is a fancy word for sample data. Fixtures allow you to populate your testing database with predefined data before your tests run. Fixtures are database independent and written in YAML. There is one file per model.

The test\_helper.rb file holds the default configuration for your tests.

## Model Tests: Bus Me

Let’s do our first tests with our Bus Me app since it has a pretty basic model. Navigate to that app folder, then run:

rake

You just ran and passed all your tests! Let’s take a look at the current tests to understand what’s being tested. Open up test/controllers/ and test/fixtures (walk through these).

### Writing Our First Tests

We should probably add some validation to our app - we don’t want people trying to find a location without an address or city. But first, let’s write the tests for that validation first so that we know they fail. What would be the tests we would write in English?

* Should not save location without address
* Should not save location without city

Let’s go ahead and write those tests. Which file should we write them in? tests/models/location\_test.rb

test "should not save location without address" do

location = Location.new

assert\_not location.save

end

Now run rake to make sure it failed. Let’s go ahead and add the other test too, then re-run rake to make sure both tests fail. Let’s also add a friendlier message to see what the output looks like (go ahead and add the message to our previous test too).

test "should not save location without city" do

location = Location.new

assert\_not location.save, "Saved the location without a city"

end

What do we need to do to get these tests to pass? We need to add validation to our locations model:

class Location < ActiveRecord::Base

validates :address, :city, presence: true

geocoded\_by :my\_location # can also be an IP address

after\_validation :geocode # auto-fetch coordinates

Now re-run the tests to make sure they pass.

### Available Assertions

Here are the available assertion statements - let’s take a look at some:

<http://edgeguides.rubyonrails.org/testing.html#available-assertions>

## Functional Tests

In Rails, testing the various actions of a controller is a form of writing functional tests. Remember your controllers handle the incoming web requests to your application and eventually respond with a rendered view. When writing functional tests, you're testing how your actions handle the requests and the expected result, or response in some cases an HTML view.

You should test for things 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?

### Functional Tests With AJAX Tasks

Let’s play with functional tests using our AJAX Tasks app. Navigate to that folder and run your tests using “rake”. We have 3 errors. Why might we be having those errors?

Well, our lesson on AJAX actually left out some key controller setup that is needed when using JavaScript to render views. This is also why our errors aren’t showing up. We can see an example of a full set up [here](http://guides.rubyonrails.org/working_with_javascript_in_rails.html#server-side-concerns).

Let’s first fix our **create** method in the controller. Edit your method to look like this:

def create

@user\_task = UserTask.new(user\_task\_params)

respond\_to do |format|

if @user\_task.save

format.html { redirect\_to @user\_task }

format.js {}

format.json { render :show, status: :created, user\_task: @user\_task }

else

format.html { render :new }

format.js { render :action => "new"}

format.json { render json: @user\_task.errors, status: :unprocessable\_entity }

end

end

end

Now run your server, and attempt to enter in a task with missing info (assuming you set up validation). You will finally see your errors! Re-run your tests. You should now have one more test passing (only 2 errors instead of 3).

Now let’s fix our **update** method - we are mainly adding one extra line in the default for each branch - highlighted in blue below (I also delete the notice):

def update

respond\_to do |format|

if @user\_task.update(user\_task\_params)

format.html { redirect\_to @user\_task }

format.js {}

format.json { render :show, status: :ok, location: @user\_task }

else

format.html { render :edit }

format.js { render :action => "edit"}

format.json { render json: @user\_task.errors, status: :unprocessable\_entity }

end

end

end

Now when you try to update and make the task description empty, you will see an error. Also, when you run rake, your update error will clear.

Finally, let’s update the **delete** method:

def destroy

@user\_task.destroy

respond\_to do |format|

format.html { redirect\_to user\_tasks\_url }

format.js {}

format.json { head :no\_content }

end

end

Try deleting a task and re-running your tests. When all is well, commit your changes.

Classroom challenge: Write tests that verify that a user can’t save a task without a description or due date. Comment out your validation to verify that those tests will fail.

## GitHub Badges Using Travis CI

Every notice the badges on Github repo’s that say tests are passing? Travic CI is a cool tool that will handle testing for you - you can even set it up to test pull requests before you integrate them. Let’s take a look <https://travis-ci.org/>.

1. Sign up for an account with your GitHub account.
2. Add a repo (slide lever).
3. Add a .travis.yml file to your app’s folder with this content (but edit to make accurate - like Ruby version):

branches:

only:

- 'master'

- 'test'

language: ruby

rvm:

- 2.2.1

# uncomment this line if your project needs to run something other than `rake`:

# script: bundle exec rspec spec

script:

- bundle exec rake db:setup

# - bundle exec rake db:migrate

- bundle exec rake test

Now save, commit, and push. Check your Travis CI dashboard - it should automatically start running its tests.

Once your tests pass, you probably want to add the badge to your repo. Go to your readme file and change it to a markdown file - rename it as readme.md. Then, click on the badge showing in Travis CI, switch the option to markdown, then copy the code. Paste it into your app’s readme, like so:

## Bus Me

[![Build Status](https://travis-ci.org/siakaramalegos/bus\_me.svg?branch=test)](https://travis-ci.org/siakaramalegos/bus\_me)

I created this app as a learning project for Tech Talent South's code immersion bootcamp. It is a Rails app that uses Atlanta's MARTA API to pull in nearest bus information based on a user's inputted location.

You can see the deployed version on [Heroku](https://bus-me.herokuapp.com/).

For basics on using Markdown, see [GitHub’s easy documentation](https://help.github.com/articles/markdown-basics/). Commit your changes and push to GitHub.

## Homework

Clean up your tests for another app. Add Travis CI to them, and update the readme with a build passing badge.