Unit 8: Testing, Roles, Mailers, and Close-Out

[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 8.2: Collaboration Using GitHub](#h.u9vfvj4s3efd)

[Gem of the Day](#h.pwty1tkukjvb)

[Rails Composer Walk-Through](#h.sp1e4irjvnkr)

[Recipe Project](#h.daoa4txmjy02)

[Roles and Plan](#h.qj2ya8gymnzj)

[Action](#h.dmtblcgbw7st)

[Homework](#h.up1100wpuwwj)

[Lesson 8.3: User Roles, OmniAuth, and ActiveAdmin](#h.skne8etstdwz)

[Gem of the Day](#h.tjqswavd9eee)

[Adding an Admin Role to Doggy Daycare](#h.brkjt98m3l7x)

[Add Admin to User Model](#h.fbb2y17th7cv)

[Set Yourself as Admin](#h.avpodutdd0hg)

[Restricting Methods to Admins](#h.njhp6g1kn3uk)

[OmniAuth for Facebook (Optional)](#h.qr4fepoon12p)

[ActiveAdmin (Optional)](#h.wbvw9j1ngbj2)

[Lesson 8.4: Mailers](#h.bjnz9kyrh5hq)

[Gem of the Day](#h.dtqebew68fyi)

# 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.

# Lesson 8.2: Collaboration Using GitHub

## Gem of the Day

ActiveAdmin <https://github.com/activeadmin/activeadmin>

<http://activeadmin.info/>

### Rails Composer Walk-Through

Rails Composer is a great tool that will build a lot of your basic app requirements all at once. Let’s walk through creating a fake app so that you understand the different options.

From the readme, we know to run this command (changing the app name to what you want your app to be named):

rails new myapp -m https://raw.github.com/RailsApps/rails-composer/master/composer.rb

Windows users will probably have an SSL error - here is the fix:

<https://gist.github.com/fnichol/867550>

Here are the options I generally choose for an app like this - feel free to experiment. However, in the beginning, I recommend you use less of the functionality until you get comfortable working with each of these tools.

* Build a starter application? 3) Custom application (experimental)
* Web server for development? 4) Puma
* Web server for production? 1) Same as development
* Database used in development? 1) SQLite
* Template engine? 1) ERB
* Test framework? 1) None (use default minitest)
* Front-end framework? 2) Bootstrap 3.3
* Add support for sending email? 1) None
* Authentication? 2) Devise
* Devise modules? 1) Devise with default modules
* Authorization? 2) Simple role-based
* Admin interface for database? 1) None (add ActiveAdmin later)
* Use a form builder gem? 2) SimpleForm
* Prepare for deployment 1) none or 2) Heroku
* Add gem and file for environment variables? 3) Add application.yml with Figaro

Notice that in both secrets.yml and application.yml, you have some default roles set up. These provide the admin login details you would need to log in as an admin - your database is currently seeded with this fake user (eventually you want to change this otherwise anyone can guess).

Go ahead and run your server and take a look at the app.

## Recipe Project

We are going to collaborate on a recipes project, which can be found at: <https://github.com/siakaramalegos/recipes-collaboration>

### Roles and Plan

Currently, it only has a basic scaffold of Recipes with title, ingredients, and steps. We want to make it fancier though, so here are some different features we can add - everyone should pick an item from this list or think of a new idea to add:

* add users using Devise, and add a user reference to the recipes table
* add styling (maybe using starter generators) and navbar
* add home and about pages with content, change root to home page
* modify index page to use panels
* convert to ajax (at least for the delete action on index)
* add seeds and maybe a populator task
* add pagination and/or friendly IDs

Once everyone has an item, think through whether anything depends on anything else, and if so, plan the steps accordingly.

### Action

Now that we have a plan, let’s start working! These are the general steps to contributing to an open-source project on GitHub. You can officially add collaborators, but we will do this project using pull requests.

1. Go to <https://github.com/siakaramalegos/recipes-collaboration> and click on “Fork”. This will make a copy of the repo in your own account, but it will still be linked to the original author.
2. Once the repo is copied into your account, make sure you are inside the version in your account, and copy the clone URL. Then go to your command line in your TTS folder and type “git clone URL”, but of course replace URL with that actual url.
3. Now, cd into that new app folder.
4. The first thing we will need to do is generally check out the gem file and modify. I’ve already conveniently includes gems for both Windows and Mac, so you should only need to bundle next. Before doing, this, verify which version of Rails you are running (especially for the Windows users).
5. Before we can take a peek at our app, we also need to do a rake db:migrate. This is because the development database is always local to a system and ignored by git.
6. Now run your rails server. If you get a permission denied, try doing bundle exec rails s instead.
7. Verify that the basic app is working as expected.

Now that we successfully pulled down a copy to our systems, we can start working on our assigned features. The generally accepted steps or norms are as follows:

1. Create a feature branch - use a branch name that is descriptive of the change you are going to make, like “starter-generators”. Type git checkout -b newbranchname to both create the branch and check it out.
2. Make all your changes in that branch. Use good git commit practices - one commit per logical change, and type good commit messages.  
     
   If applicable, also add tests. In real life, developers will expect you to write tests for whatever features you are adding. They will also expect that all tests will be passing before you do a pull request.
3. Once you are satisfied with your new feature, push that feature branch up to GitHub using git push origin newbranchname.
4. Navigate to your repo on GitHub, and switch to your feature branch. You should see a link that automatically pops up to do a Pull Request. Click on that. Make sure that your pull request selects my repo as the base fork, and then your repo and feature branch as the head fork. Submit the pull request with a descriptive comment of the changes you are making. Be nice!

Now, you wait until the base fork owner has a chance to review your changes and either accept or deny them. Conversations happen in this space, and some requests are rejected because you need to add something else. Don’t be bummed if this happens! It’s all a part of the process. Just fix your code and re-submit the pull request.

## Homework

Work on your personal apps!

# Lesson 8.3: User Roles, OmniAuth, and ActiveAdmin

## Gem of the Day

CanCanCan <https://github.com/CanCanCommunity/cancancan>

OmniAuth <https://github.com/intridea/omniauth>

## Adding an Admin Role to Doggy Daycare

You can do more complex user management with the CanCanCan gem, but today we are just going to implement a basic admin role so that we can better understand how that gem makes a more complex arrangement easier to manage.

### Add Admin to User Model

Basically, we want to only allow admins to add/edit/destroy products, dogs, and owners.

1. First, we need to add an admin column to the user table:  
   rails g migration add\_admin\_to\_user admin:boolean
2. Edit your migration to make admin default to false:  
   add\_column :users, :admin, :boolean, :default => false
3. Now run rake db:migrate.
4. Finally, commit your changes.

### Set Yourself as Admin

How would we set the admin status of a user? We can manually go into rails console. What about in production? We can either use Figaro to set an environment variable and seed our user database with ourselves, OR we can manually go into rails console on Heroku with

heroku run rails console

So how do we do that in console?

1. ap User.all (awesome print if you have it, then see which user you are)
2. Assuming you’re the first user, user=User.first
3. user.admin = true
4. user.save
5. ap User.first (to double-check)

### Restricting Methods to Admins

Since we added a column with the property admin set to a boolean, we can call the following method to determine whether a user is an admin or not:

current\_user.admin?

We can also create a method similar to the one Devise uses for authenticate\_user.

First, in our **Dogs controller**, let’s add an authenticate admin user method that we will write in the next step:

class DogsController < ApplicationController

before\_action :set\_dog, only: [:show, :edit, :update, :destroy]

before\_action :authenticate\_user!, except: [:index, :show]

before\_action :authenticate\_admin\_user!, except: [:index, :show]

Now, let’s create that action in the **Application Controller** as well as what to do if it errors out.

class ApplicationController < ActionController::Base

# Prevent CSRF attacks by raising an exception.

# For APIs, you may want to use :null\_session instead.

protect\_from\_forgery with: :exception

# restrict access to admin module for non-admin users

def authenticate\_admin\_user!

raise SecurityError unless current\_user.try(:admin?)

end

rescue\_from SecurityError do |exception|

redirect\_to root\_url, alert: 'You do not have access to that feature.'

end

end

Finally, let’s add alerts to our application.html.erb and customize the format so it’s red:

<% if notice.present? %>

<div class="alert alert-dismissable alert-success">

<button type="button" class="close" data-dismiss="alert" aria-hidden="true">&times;</button>

<%= notice %>

</div>

<% elsif alert.present? %>

<div class="alert alert-dismissable alert-danger">

<button type="button" class="close" data-dismiss="alert" aria-hidden="true">&times;</button>

<%= alert %>

</div>

<% end %>

Now test it out! Commit your changes.

Classroom challenge: Add admin authentication to Owners (optional), Products, Categories, and Orders, though be careful which methods you add it to - you may kill some of your current functionality.

## OmniAuth for Facebook (Optional)

See if you can follow along with this lesson to add sign-up/login ability with Facebook:

<http://techtalentsouth.slides.com/techtalentsouth/atl-facebook-omniauth-186?token=qG3hLhh3>

## ActiveAdmin (Optional)

See if you can follow along with the second part of this method to add ActiveAdmin to your project - you will need to think through this and change things that do not make sense for your app:

<https://docs.google.com/document/d/1t1BODGNSiGHMQ5B41ZKCM--59BUXJ1IjNmA398ZdViM/edit?usp=sharing>

# Lesson 8.4: Mailers

## Gem of the Day

Letter Opener <https://github.com/ryanb/letter_opener>

**Vote**

Confirmable and Letter opener or OmniAuth (facebook)?

<https://github.com/plataformatec/devise/wiki/How-To:-Add-:confirmable-to-Users>