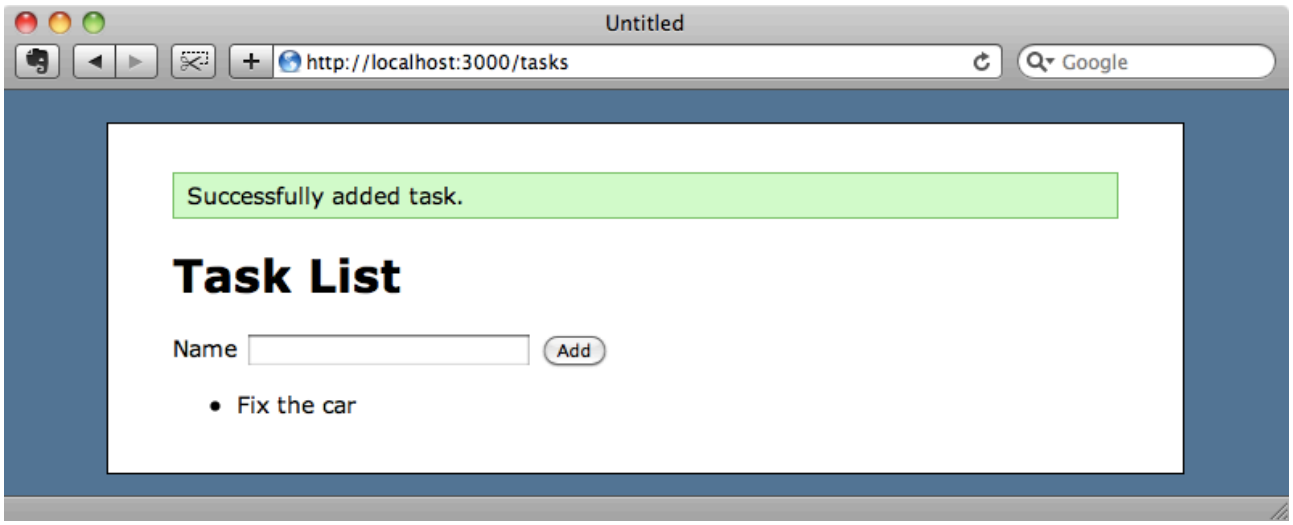
Episode 257

Request Specs and Capybara

High-level testing is a great way to test Rails applications. A popular way to perform this kind of testing is Cucumber which was covered here back in episode 155 [watch[1], read[2]]. Not everyone likes Cucumber's English-style syntax for defining the application's behaviour, though, so this episode we'll take a look at an alternative way to do high-level testing .

While we'd normally advocate test-driven development, in this episode we'll be adding tests to an existing application so that we don't have to worry about the implementation details and making the application work. The application we'll be adding tests to is a simple, one-page to-do list application. This app shows a list of tasks and has a form that allows new tasks to be added.



To test this application we're going to use request specs, which are available in RSpec 2.0, so the first step we'll need to take is to add the RSpec Rails gem to the development and test groups in the `Gemfile` and then run `bundle`.

---

[1] http://railscasts.com/episodes/155-beginning-with-cucumber

[2] http://asciicasts.com/episodes/155-beginning-with-cucumber

/Gemfile

```
source 'http://rubygems.org'

gem 'rails', '3.0.5'
gem 'sqlite3'
gem 'nifty-generators'

group :development, :test do
  gem 'rspec-rails'
end
```

Once Bundler has run we can run the following command to set up RSpec in our application.

```
$ rails g rspec:install
```

Request specs are RSpec's equivalent to Rails' built-in integration testing (these were covered in episode 187 [watch[3], read[4]]). To generate a request spec we call the integration test generator.

```
$ rails g integration_test task
```

This will create a `tasks_spec.rb` file in the `spec/requests` directory, which looks like this:

[3] http://railscasts.com/episodes/187-testing-exceptions

[4] http://asciicasts.com/episodes/187-testing-exceptions

/spec/requests/tasks_spec.rb

```
require 'spec_helper'

describe "Tasks" do
  describe "GET /tasks" do
    it "works! (now write some real specs)" do
      # Run the generator again with the --webrat flag if you want
to use webrat methods/matchers
      get tasks_path
      response.status.should be(200)
    end
  end
end
```

The contents of this file look like fairly standard RSpec tests with the difference that we can call methods like `get` to call a page in the application and `response` to check the response for that request. The spec that's included by default in the file requests the tasks page that we saw earlier and checks that the response status is 200, indicating a successful response. This spec should pass without modification, so let's try it. We can run `rake spec:requests` to run only the request specs.

```
$ rake spec:requests
(in /Users/eifion/code/tasklist)
/Users/eifion/.rvm/rubies/ruby-1.9.2-p0/bin/ruby -S bundle exec
rspec ./spec/requests/tasks_spec.rb
DEPRECATION WARNING: <% %> style block helpers are deprecated.
Please use <%= %>. (called from
_app_views_tasks_index_html_erb___875755388255758006_2152410020_35
63250333774836596 at /Users/eifion/code/tasklist/app/views/tasks/
index.html.erb:3)
.

Finished in 0.18535 seconds
1 example, 0 failures
```

The spec passes, but we do get a deprecation warning and this is the kind of issue that this kind of testing can find but which would be missed if we were just testing directly in a browser. It looks like we forgot to use an equals sign in a block helper and so we have some useful feedback already from writing request specs.

In the view file for the form we'll see that we've missed the equals sign from the opening `form_for` tag which is required in Rails 3.

/app/views/tasks/index.html.erb

```
<% form_for Task.new do |f| %>
```

If we add this in...

/app/views/tasks/index.html.erb

```
<%= form_for Task.new do |f| %>
```

...and then run the specs again we'll get a passing spec and no warnings.

```
$ rake spec:requests
(in /Users/eifion/code/tasklist)
/Users/eifion/.rvm/rubies/ruby-1.9.2-p0/bin/ruby -S bundle exec
rspec ./spec/requests/tasks_spec.rb
.

Finished in 0.16725 seconds
1 example, 0 failures
```

Our First Real Request Spec

In a request spec we generally want to do more than just test the status of the response. We'll replace the default spec now with one that tests that the tasks are shown on the page.

/specs/requests/tasks_spec.rb

```
require 'spec_helper'

describe "Tasks" do
  describe "GET /tasks" do
    it "displays tasks" do
      Task.create!(:name => "paint fence")
      get tasks_path
      response.body.should include("paint fence")
    end
  end
end
```

This spec is pretty straightforward. We create a new task then visit the tasks page and check that the text on the page includes that task's name. When we run `rake spec:requests` the spec passes as the page does include that text.

As requests specs are built on Rails integration tests they support all of the methods that integration tests have[5]. For example if we want to test the creation of a new task we can make use of the `post_via_redirect` method to make sure that the redirect is followed when we create a task.

Let's write that spec now. It will call `post_via_redirect` to POST to the index page and pass in the parameters to create a new task called "mow lawn". It then checks that that text is present in the resulting page.

[5] http://guides.rubyonrails.org/testing.html#helpers-available-for-integration-tests

/specs/requests/tasks_spec.rb

```
require 'spec_helper'

describe "Tasks" do

  # Other task omitted.

  describe "POST /tasks" do
    it "creates a task" do
      post_via_redirect tasks_path, ↵
        :task => { :name => "mow lawn" }
      response.body.should include("mow lawn")
    end
  end
end
```

If we run the specs now we'll have two passing specs, so it looks like our code works as expected. If we were doing test-driven development we'd start with a failing spec and then write code to make it pass and this is one of the advantages of test-driven development as it ensures that your tests are working. In our application we've jumped straight to a passing test as the implementation is already in place. When adding tests for existing code it can sometimes be worth deliberately breaking something in the code to ensure that the test then fails and then fixing it again.

Testing The UI With Capybara

One problem with our request spec is that it doesn't test the full user experience. We could completely break the New Task form and this wouldn't be caught in the specs. This is because we're POSTing directly to the `create` action instead of going through the form like the user would.

We need a way to mimic the user's actions and we can do this by using Capybara[6]. This is an alternative to Webrat, which was covered back in episode 156 [watch[7], read[8]]. Capybara gives us a number of methods we can use to mimic a user's

---

[6] https://github.com/jnicklas/capybara

[7] http://railscasts.com/episodes/156-webrat

[8] http://asciicasts.com/episodes/156-webrat

behaviour in a web application. Capybara is a gem and is installed in the usual way. We'll also add the launchy[9] gem and you'll see why shortly. In the `Gemfile` we'll add both gems to the development and test groups and then run `bundle` to install it.

`/Gemfile`

```
source 'http://rubygems.org'

gem 'rails', '3.0.5'
gem 'sqlite3'
gem 'nifty-generators'

group :development, :test do
  gem 'rspec-rails'
  gem 'capybara'
  gem 'launchy'
end
```

Capybara is automatically included in our request specs so in the first of our specs we can use Capybara's `visit` method instead of calling `get`. We can also replace `response.body.should include` with `page.should have_content`.

In the second spec we can now use Capybara to simulate filling in and submitting the form instead of POSTing directly to the `create` action. The `fill_in` method will find the text box with the associated label "Name" and set its value and we can use `click_button` to find the form's button by its text and click it.

[9] http://rubygems.org/gems/launchy

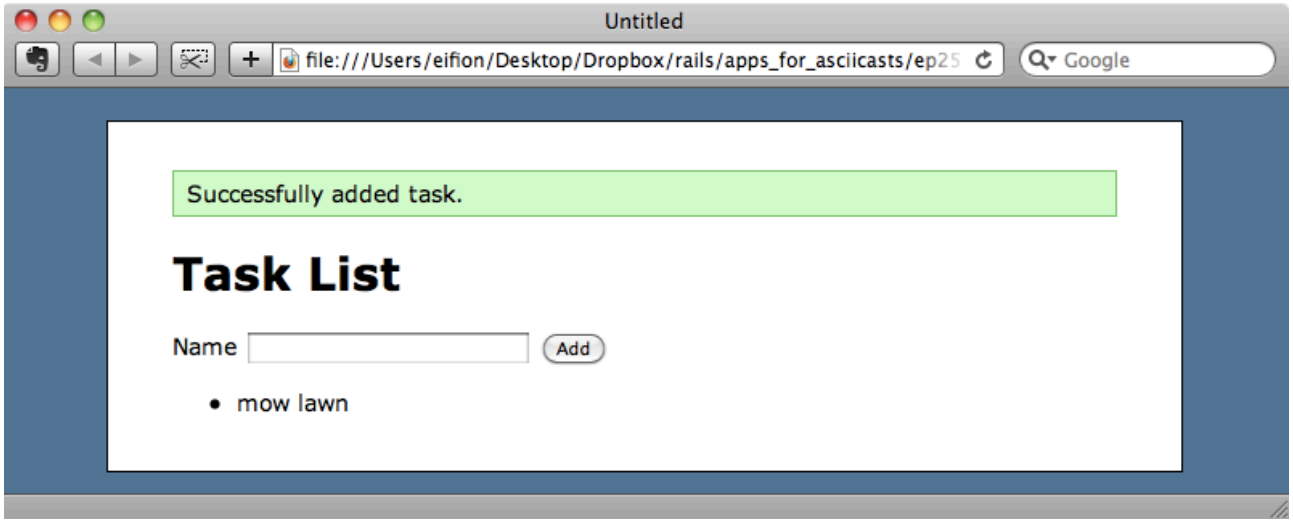
/specs/requests/task_spec.rb

```
require 'spec_helper'

describe "Tasks" do
  describe "GET /tasks" do
    it "displays tasks" do
      Task.create!(:name => "paint fence")
      visit tasks_path
      page.should have_content("paint fence")
    end
  end

  describe "POST /tasks" do
    it "creates a task" do
      visit tasks_path
      fill_in "Name", :with => "mow lawn"
      click_button "Add"
      page.should have_content("Successfully added task.")
      page.should have_content("mow lawn")
    end
  end
end
```

The specs still pass when we run them and now test the form too so if we make any breaking changes to the form this will be picked up.

Debugging Pages

If a spec does fail how can be go about debugging that problem? This is where launchy comes in. As we're using Capybara we can call the `save_and_open_page` method at any point and this will open the page in a browser so that we can take a look at it. We'll add it immediately after `click_button "Add"` and when we run the specs again we'll see the tasks page just before we check its contents.



We can see the state of the page at this point with the flash notice visible and the task that was added through Capybara.

Testing JavaScript

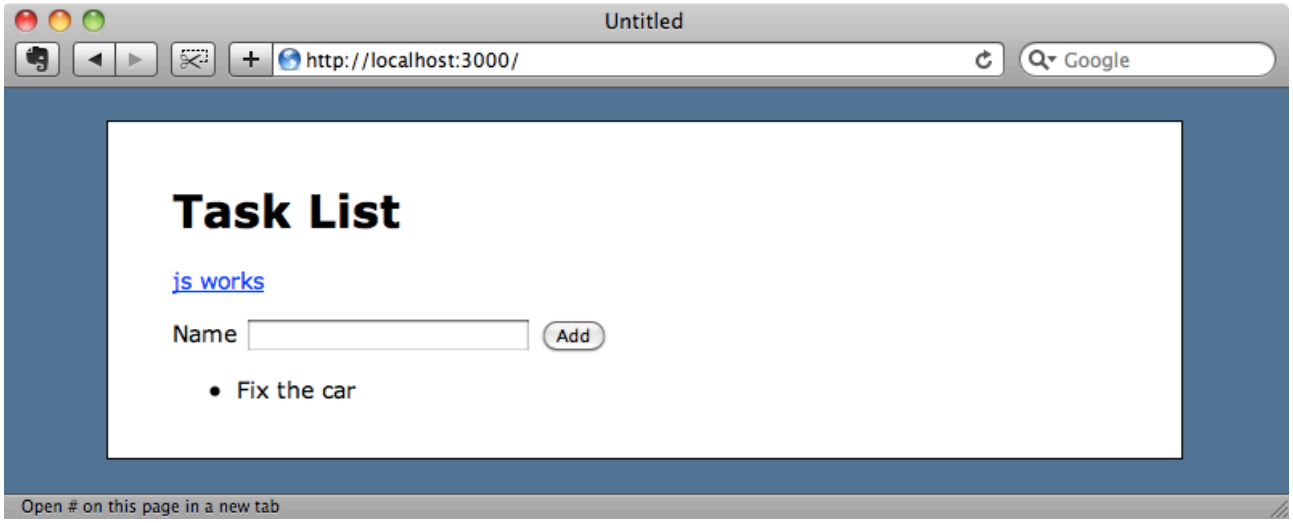
Our application is now pretty well tested and these tests have been simple to add with request specs and Capybara. But what if we have JavaScript in our application and want to test that? This can be done quite easily and we'll show you how now.

In our `index` template we'll add some JavaScript so that we can test it. We'll keep it inline for simplicity's sake and use `link_to_function` to add a link to the page that calls a JavaScript function when clicked that will change the link's text to `"js works"`. The script that runs requires jQuery so we've added a reference to it in the application's layout file. (By the way, jQuery will be the default JavaScript library in Rails 3.1 instead of Prototype).

/app/views/tasks/index.html.erb

```
<h1>Task List</h1>

<%= link_to_function "test js", '$(this).html("js works")' %>

<%= form_for Task.new do |f| %>
  <p>
    <%= f.label :name %>
    <%= f.text_field :name %>
    <%= f.submit "Add" %>
  </p>
<% end %>

<ul>
  <% @tasks.each do |task| %>
    <li><%= task.name %></li>
  <% end %>
</ul>
```

So now we have a link on the page that says "test js" and when we click it the text changes.



Let's test this new functionality through Capybara. First we'll write a new spec

/spec/requests/tasks_spec.rb

```
it "supports js" do
  visit_tasks_path
  click_link "test js"
  page.should have_content("js works")
end
```

When we run the request specs now we get one failure.

```
$ rake spec:requests
(in /Users/eifion/code/tasklist)
/Users/eifion/.rvm/rubies/ruby-1.9.2-p0/bin/ruby -S bundle exec
rspec ./spec/requests/tasks_spec.rb
.F.

Failures:

  1) Tasks GET /tasks supports js
     Failure/Error: page.should have_content("js works")
       expected #has_content?("js works") to return true, got
false
     # ./spec/requests/tasks_spec.rb:14:in `block (3 levels) in
<top (required)>'

Finished in 0.83232 seconds
3 examples, 1 failure
rake aborted!
ruby -S bundle exec rspec ./spec/requests/tasks_spec.rb failed

(See full trace by running task with --trace)
```

The spec fails because the page doesn't have the text that is set by JavaScript. Capybara doesn't support JavaScript by default. We have to tell it to use JavaScript through Selenium.

The functionality we're going to show now isn't supported in the current release version of Capybara. Bundler makes it easy, however, to get the latest version from GitHub.

/Gemfile

```
source 'http://rubygems.org'

gem 'rails', '3.0.5'
gem 'sqlite3'
gem 'nifty-generators'

group :development, :test do
  gem 'rspec-rails'
  gem 'capybara', :git => 'git://github.com/jnicklas/capybara.git'
  gem 'launchy'
end
```

When we run `bundle` again the latest version of the Capybara gem will be downloaded and installed.

Another step that's required to get JavaScript testing working is to modify the `spec_helper` file and add the line `require 'rspec/rails'`.

/spec/spec_helper.rb

```
# This file is copied to spec/ when you run 'rails generate
rspec:install'
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'
require 'capybara/rspec'

# rest of file...
```

This step might not be required in the release version but for now it is. It's easy now to tell Capybara to use a JavaScript driver for any of our specs. All we have to do is add the `:js => true` option to the spec.

/spec/requests/tasks_spec.rb

```
it "supports js", :js => true do
  visit tasks_path
  click_link "test js"
  page.should have_content("js works")
end
```

When we run the specs now Firefox will open to run the specs that require JavaScript. All of the specs pass now pass again.

This is a great feature to have but there is a problem to do with database records here that we'll probably run into. We can demonstrate this by adding the `:js => true` option to the first two specs, including the one that creates a new `Task`.

/spec/requests/tasks_spec.rb

```
describe "Tasks" do
  describe "GET /tasks", :js => true do
    it "displays tasks" do
      Task.create!(:name => "paint fence")
      visit tasks_path
      page.should have_content("paint fence")
    end

    it "supports js" do
      visit tasks_path
      click_link "test js"
      page.should have_content("js works")
    end
  end
end
```

When we run the specs now it'll use Selenium for the first two requests.

```
$ rake spec:requests
(in /Users/eifion/code/tasklist)
/Users/eifion/.rvm/rubies/ruby-1.9.2-p0/bin/ruby -S bundle exec
rspec ./spec/requests/tasks_spec.rb
F..

Failures:

  1) Tasks GET /tasks displays tasks
     Failure/Error: page.should have_content("paint fence")
       expected there to be content "paint fence" in "Task List
\ntest js\n\nName"
     # ./spec/requests/tasks_spec.rb:8:in `block (3 levels) in
<top (required)>'

Finished in 7.69 seconds
3 examples, 1 failure
```

This time the first spec fails as the content "paint fence" is missing from the page. The database record that is created isn't available to the Selenium tests and this is because our specs are using database transactions which aren't compatible with Selenium. To fix this we can set the `config.use_transactional_fixtures` setting in the `spec_helper` file to `false`.

/spec/spec_helper.rb

```
# If you're not using ActiveRecord, or you'd prefer not to run
each of your
# examples within a transaction, remove the following line or
assign false
# instead of true.
config.use_transactional_fixtures = false
```

This will get the tests passing again but it will mean that the database records are carried over between specs which we don't want. To solve this problem we can use a gem called database_cleaner[10] to clean the database between specs. The documentation[11] explains the options we have.

---

[10] https://github.com/bmabey/database_cleaner

[11] https://github.com/bmabey/database_cleaner#readme

To use it we'll add a reference in the `Gemfile` and run `bundle` again to install it.

/Gemfile

```
source 'http://rubygems.org'

gem 'rails', '3.0.5'
gem 'sqlite3'
gem 'nifty-generators'

group :development, :test do
  gem 'rspec-rails'
  gem 'capybara', :git => 'git://github.com/jnicklas/capybara.git'
  gem 'launchy'
  gem 'database_cleaner'
end
```

Next we'll modify the `spec_helper` file again and add the code to clean the database between specs.

/spec/spec_helper.rb

```
config.before(:suite) do
  DatabaseCleaner.strategy = :truncation
end

config.before(:each) do
  DatabaseCleaner.start
end

config.after(:each) do
  DatabaseCleaner.clean
end
```

When we run the specs now, with the first two running through Selenium again, they all pass as we're no longer using database transactions. This might have seemed like a lot of work, but once it's all set up it's easy to test JavaScript for any spec by just adding the `:js => true` option.

A feature that will be in the next Capybara release that we haven't covered is a new DSL for defining specs. We'll be able to use `feature`, `background` and `scenario`

methods to define our specs in a similar way to how the Steak[12] gem works. If you like working with that kind of DSL it'll be built into Capybara which eliminates the need to use Steak.

[12] https://github.com/cavalle/steak