Table of Contents (left)

rspec-rails-2

rspec-2 for rails-3 with lightweight extensions to each

Note: Use rspec-rails-1.3 for rails-2.

Install

```
gem install rspec-rails
```

This installs the following gems:

```
rspec
rspec-core
rspec-expectations
rspec-mocks
rspec-rails
```

Configure

Add `rspec-rails` to the `:test` and `:development` groups in the Gemfile:

```
group :test, :development do
  gem "rspec-rails", "~> 2.6"
end
```

It needs to be in the `:development` group to expose generators and rake tasks without having to type `RAILS_ENV=test`.

Now you can run:

```
rails generate rspec:install
```

This adds the spec directory and some skeleton files, including the "rake spec" task.

Generators

If you type `script/rails generate`, the only RSpec generator you'll actually see is `rspec:install`. That's because RSpec is registered with Rails as the test framework, so whenever you generate application components like models, controllers, etc, RSpec specs are generated instead of Test::Unit tests.

Please note that the generators are there to help you get started, but they are no substitute for writing your own examples, and they are only guaranteed to work out of the box for the default scenario (`ActiveRecord` & `Webrat`).

Webrat and Capybara

You can choose between webrat or capybara for simulating a browser, automating a browser, or setting expectations using the matchers they supply. Just add your preference to the Gemfile:

```
gem "webrat"
gem "capybara"
```

Living on edge

Bundler makes it a snap to use the latest code for any gem your app depends on. For rspec-rails, you'll need to point bundler to the git repositories for `rspec-rails` and the other rspec related gems it depends on:

```
gem "rspec-rails",        :git => "git://github.com/rspec/rspec-rails.git"
gem "rspec",              :git => "git://github.com/rspec/rspec.git"
gem "rspec-core",         :git => "git://github.com/rspec/rspec-core.git"
gem "rspec-expectations", :git => "git://github.com/rspec/rspec-expectations.git"
gem "rspec-mocks",        :git => "git://github.com/rspec/rspec-mocks.git"
```

Run `bundle install` and you'll have whatever is in git right now. Any time you want to update to a newer head, just run `bundle update`.

Keep in mind that each of these codebases is under active development, which means that its entirely possible that you'll pull from these repos and they won't play nice together. If playing nice is important to you, stick to the published gems.

Backwards compatibility

This is a complete rewrite of the rspec-rails extension designed to work with rails-3.x and rspec-2.x. It will not work with older versions of either rspec or rails. Many of the APIs from rspec-rails-1 have been carried forward, however, so upgrading an app from rspec-1/rails-2, while not pain-free, should not send you to the doctor with a migraine.

Known issues

See http://github.com/rspec/rspec-rails/issues

Request Specs

Request specs live in spec/requests, and mix in behavior ActionDispatch::Integration::Runner, which is the basis for Rails' integration tests. The intent is to specify one or more request/response cycles from end to end using a black box approach.

```
describe "home page" do
  it "diplays the user's username after successful login" do
    user = User.create!(:username => "jdoe", :password => "secret")
    get "/login"
    assert_select "form.login" do
      assert_select "input[name=?]", "username"
      assert_select "input[name=?]", "password"
      assert_select "input[type=?]", "submit"
    end

    post "/login", :username => "jdoe", :password => "secret"
    assert_select ".header .username", :text => "jdoe"
  end
end
```

This example uses only standard Rails and RSpec API's, but many RSpec/Rails users like to use extension libraries like FactoryGirl and Capybara:

```
describe "home page" do
  it "diplays the user's username after successful login" do
    user = Factory(:user, :username => "jdoe", :password => "secret")
    visit "/login"
    fill_in "Username", :with => "jdoe"
    fill_in "Password", :with => "secret"
    click_buton "Log in"

    page.should have_selector(".header .username", :content => "jdoe")
  end
end
```

FactoryGirl decouples this example from changes to validation requirements, which can be encoded into the underlying factory definition without requiring changes to this example.

Among other benefits, Capybara binds the form post to the generated HTML, which means we don't need to specify them separately.

There are several other Ruby libs that implement the factory pattern or provide a DSL for request specs (a.k.a. acceptance or integration specs), but FactoryGirl and Capybara seem to be the most widely used. Whether you choose these or other libs, we strongly recommend using something for each of these roles.

Controller Specs

Controller specs live in spec/controllers, and mix in ActionController::TestCase::Behavior, which is the basis for Rails' functional tests.

with fixtures

```
describe WidgetsController do
  describe "GET index" do
    fixtures :widgets

    it "assigns all widgets to @widgets" do
      get :index
      assigns(:widgets).should eq(Widget.all)
    end
  end
end
```

with a factory

```
describe WidgetsController do
  describe "GET index" do
    it "assigns all widgets to @widgets" do
      widget = Factory(:widget)
      get :index
      assigns(:widgets).should eq([widget])
    end
  end
end
```

```
end
```

with stubs

```
describe WidgetsController do
  describe "GET index" do
    it "assigns all widgets to @widgets" do
      widget = stub_model(Widget)
      Widget.stub(:all) { [widget] }
      get :index
      assigns(:widgets).should eq([widget])
    end
  end
end
```

matchers

In addition to the stock matchers from rspec-expectations, controller specs add these matchers, which delegate to rails' assertions:

```
response.should render_template(*args)
# => delegates to assert_template(*args)

response.should redirect_to(destination)
# => delegates to assert_redirected_to(destination)
```

isolation from views

RSpec's preferred approach to spec'ing controller behaviour is to isolate the controller from its collaborators. By default, therefore, controller example groups do not render the views in your app. Due to the way Rails searches for view templates, the template still needs to exist, but it won't actually be loaded.

NOTE that this is different from rspec-rails-1 with rails-2, which did not require the presence of the file at all. Due to changes in rails-3, this was no longer feasible in rspec-rails-2.

`render_views`

If you prefer a more integrated approach, similar to that of Rails' functional tests, you can tell controller groups to render the views in the app with the `render_views` declaration:

```
describe WidgetsController do
  render_views
  # ...
```

Upgrade note

`render_views` replaces `integrate_views` from rspec-rails-1.3

`assigns`

Use `assigns(key)` to express expectations about instance variables that a controller assigns to the view in

the course of an action:

```
get :index
assigns(:widgets).should eq(expected_value)
```

View specs

View specs live in spec/views, and mix in ActionView::TestCase::Behavior.

```
describe "events/index.html.erb" do
  it "renders _event partial for each event" do
    assign(:events, [stub_model(Event), stub_model(Event)])
    render
    view.should render_template(:partial => "_event", :count => 2)
  end
end

describe "events/show.html.erb" do
  it "displays the event location" do
    assign(:event, stub_model(Event,
      :location => "Chicago"
    ))
    render
    rendered.should contain("Chicago")
  end
end
```

View specs infer the controller name and path from the path to the view template. e.g. if the template is "events/index.html.erb" then:

```
controller.controller_path == "events"
controller.request.path_parameters[:controller] == "events"
```

This means that most of the time you don't need to set these values. When spec'ing a partial that is included across different controllers, you *may* need to override these values before rendering the view.

To provide a layout for the render, you'll need to specify *both* the template and the layout explicitly. For example:

```
render :template => "events/show", :layout => "layouts/application"
```

`assign(key, val)`

Use this to assign values to instance variables in the view:

```
assign(:widget, stub_model(Widget))
render
```

The code above assigns `stub_model(Widget)` to the `@widget` variable in the view, and then renders the view.

Note that because view specs mix in `ActionView::TestCase` behavior, any instance variables you set will be transparently propagated into your views (similar to how instance variables you set in controller actions are made available in views). For example:

```
@widget = stub_model(Widget)
render # @widget is available inside the view
```

RSpec doesn't officially support this pattern, which only works as a side-effect of the inclusion of `ActionView::TestCase`. Be aware that it may be made unavailable in the future.

Upgrade note

```
# rspec-rails-1.x
assigns[key] = value

# rspec-rails-2.x
assign(key, value)
```

`rendered`

This represents the rendered view.

```
render
rendered.should =~ /Some text expected to appear on the page/
```

Upgrade note

```
# rspec-rails-1.x
render
response.should xxx

# rspec-rails-2.x
render
rendered.should xxx
```

Model specs

Model specs live in spec/models.

```
describe Articles do
  describe ".recent" do
    it "includes articles published less than one week ago" do
      article = Article.create!(:published_at => Date.today - 1.week + 1.second)
      Article.recent.should eq([article])
    end

    it "excludes articles published at midnight one week ago" do
      article = Article.create!(:published_at => Date.today - 1.week)
      Article.recent.should be_empty
    end

    it "excludes articles published more than one week ago" do
      article = Article.create!(:published_at => Date.today - 1.week - 1.second)
      Article.recent.should be_empty
    end
  end
end
```

Routing specs

Routing specs live in spec/routing.

```
describe "routing to profiles" do
  it "routes /profile/:username to profile#show for username" do
    { :get => "/profiles/jsmith" }.should route_to(
      :controller => "profiles",
      :action => "show",
      :username => "jsmith"
    )
  end

  it "does not expose a list of profiles" do
    { :get => "/profiles" }.should_not be_routable
  end
end
```

Upgrade note

`route_for` from rspec-rails-1.x is gone. Use `route_to` and `be_routable` instead.

Helper specs

Helper specs live in spec/helpers, and mix in ActionView::TestCase::Behavior.

Provides a `helper` object which mixes in the helper module being spec'd, along with `ApplicationHelper` (if present).

```
describe EventsHelper do
  describe "#link_to_event" do
    it "displays the title, and formatted date" do
      event = Event.new("Ruby Kaigi", Date.new(2010, 8, 27))
      # helper is an instance of ActionView::Base configured with the
      # EventsHelper and all of Rails' built-in helpers
      helper.link_to_event.should =~ /Ruby Kaigi, 27 Aug, 2010/
    end
  end
end
```

Matchers

rspec-rails exposes domain-specific matchers to each of the example group types. Most of them simply delegate to Rails' assertions.

`be_a_new`

- Available in all specs.
- Primarily intended for controller specs

```
object.should be_a_new(Widget)
```

Passes if the object is a `Widget` and returns true for `new_record?`

render_template

- Delegates to Rails' assert_template.
- Available in request, controller, and view specs.

In request and controller specs, apply to the response object:

```
response.should render_template("new")
```

In view specs, apply to the view object:

```
view.should render_template(:partial => "_form", :locals => { :widget => widget } )
```

redirect_to

- Delegates to assert_redirect
- Available in request and controller specs.

```
response.should redirect_to(widgets_path)
```

route_to

- Delegates to Rails' assert_routing.
- Available in routing and controller specs.

```
{ :get => "/widgets" }.should route_to(:controller => "widgets", :action => "index")
```

be_routable

Passes if the path is recognized by Rails' routing. This is primarily intended to be used with `should_not` to specify routes that should not be routable.

```
{ :get => "/widgets/1/edit" }.should_not be_routable
```

Contribute

See http://github.com/rspec/rspec-dev

Also see

- http://github.com/rspec/rspec
- http://github.com/rspec/rspec-core
- http://github.com/rspec/rspec-expectations
- http://github.com/rspec/rspec-mocks

Generated on Wed Jan 4 23:27:49 2012 by yard 0.7.4 (ruby-1.9.3).