Episode 276

Testing Time & Web Requests

In the previous episode we demonstrated a workflow for test driven development. For the most part the pattern we used works well, but there are times when our applications will have functionality that is difficult to test. We'll cover two of those scenarios in this episode.

Testing The Current Time

Last time we created a number of specs to test a `User` model. We wrote these fairly quickly and so we'll go back to one of them now and look at it more closely.

/spec/models/user.rb

```
it "saves the time the password reset was sent" do
  user.send_password_reset
  user.reload.password_reset_sent_at.should be_present
end
```

This spec checks that when a password reset is sent the time it was sent at is saved in the `password_reset_sent_at` field. This is tested by using RSpec's `be_present` matcher. This matcher calls a method that Rails provides called `present?` which checks for the existence of an object.

This spec is incomplete: it checks that a `password_reset_sent_at` value exists but not that it is set to the current time. We do set `password_reset_sent_at` to `Time.zone.now` in the `User` model's `send_password_reset` method but the spec would pass no matter what value was set here. Ideally we should test that the value is the current time by writing something like this:

/spec/models/user.rb

```
it "saves the time the password reset was sent" do
  user.send_password_reset
  user.reload.password_reset_sent_at.should eq(Time.zone.now)
end
```

Unfortunately this won't work. The current time at the time the test runs will be slightly after the time at which the code it's testing executes. When faced with a problem like this it's worth asking whether it's worth adding complexity to the test in order to test it fully. In many cases, such as this one, it's enough to test that the

timestamp value exists. There's very little possibility that there'll be a bug in the one line of code that sets the time here, but there are times that we do need to test the current time so let's investigate how we could do that in the spec.

Guard is running and we can see that the spec is failing. Although the timestamps are the same to the nearest second, they aren't identical this is enough to cause the spec to fail.

```
Failures:

  1) User#send_password_reset saves the time the password ↵
    reset was sent
     Failure/Error: ↵
       user.reload.password_reset_sent_at.should eq(Time.zone.now)

       expected Mon, 25 Jul 2011 20:34:46 UTC +00:00
            got Mon, 25 Jul 2011 20:34:46 UTC +00:00

       (compared using ==)

       Diff:
     # ./spec/models/user_spec.rb:16:in `block (3 levels) in ↵
       <top (required)>'

Finished in 1.95 seconds
9 examples, 1 failure
```

We can deal with this kind of problem by using a gem called Timecop[1]. We can use it to manipulate the current time in many different ways, including freezing it. This means that we can freeze the current time while the spec runs so that the time at which the timestamp is set will be the same as it is when the value is checked.

We can add Timecop to our by adding it to the `Gemfile` and running `bundle`. As we only need it for our tests we'll add it to the `test` group.

[1] https://github.com/jtrupiano/timecop/

/Gemfile

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

gem 'rails', '3.1.0.rc4'

gem 'sqlite3'

# Asset template engines
gem 'sass-rails', "~> 3.1.0.rc"
gem 'coffee-script'
gem 'uglifier'

gem 'jquery-rails'

gem "rspec-rails", :group => [:test, :development]
group :test do
  gem "factory_girl_rails"
  gem "capybara"
  gem "guard-rspec"
  gem "timecop"
end
```

It's a good idea now to go into the `spec_helper` file and add a call to `Timecop.return` in the `config.before(:each)` block. This ensures that any changes we make with `Timecop` are undone before each spec is run.

We can now call `Timecop.freeze` in any of our specs and freeze the current time while the spec runs. This means that we can compare both timestamps.

/spec/models/user.rb

```
it "saves the time the password reset was sent" do
  Timecop.freeze
  user.send_password_reset
  user.reload.password_reset_sent_at.should eq(Time.zone.now)
end
```

When Guard runs the specs now they all pass.

There is another unexpected scenario that can cause tests to fail. If you visit a Ruby conference in a different time zone you can suddenly find some of your time-related tests failing. Instead of travelling halfway around the world it's a lot cheaper to set the timezone in the specs. We do this by setting `Time.zone`:

```
Time.zone = "Paris"
```

To set the zone temporarily within a single spec we can call `Time.use_zone`.

/spec/model/user_spec.rb

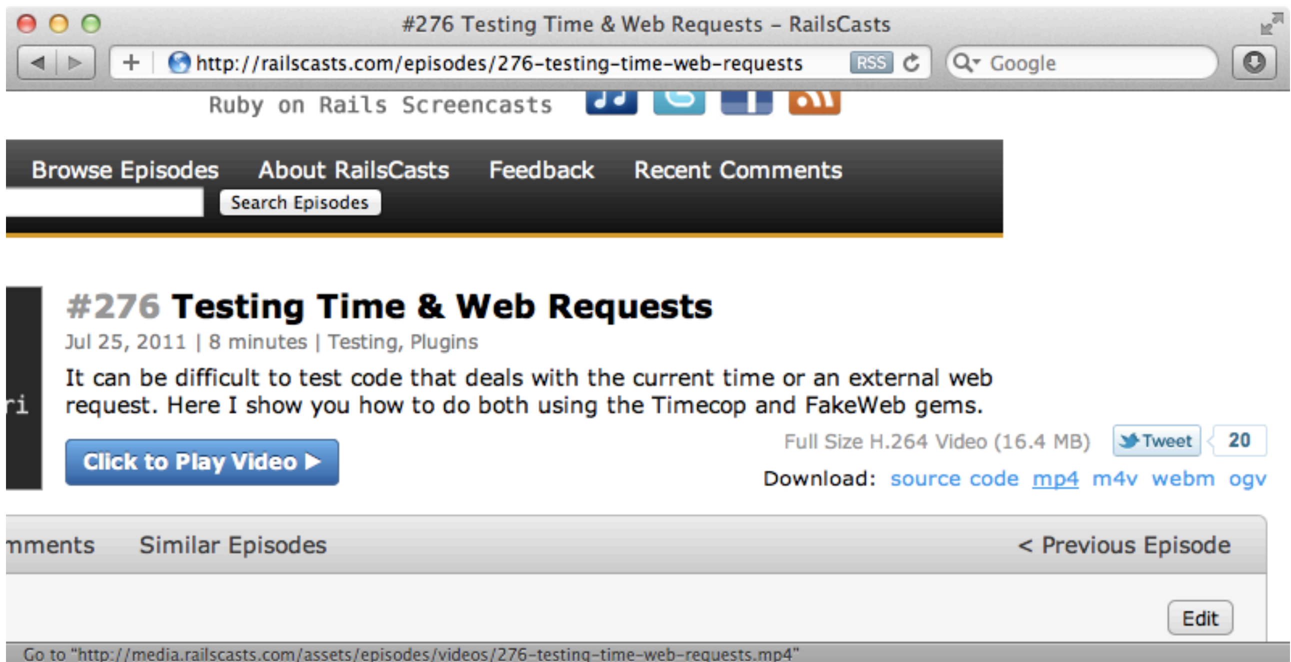
```
it "saves the time the password reset was sent" do
  Timecop.freeze
  user.send_password_reset
  Time.use_zone("Paris") do
    user.reload.password_reset_sent_at.should eq(Time.zone.now)
  end
end
```

The code inside the block will be run as if we were in Paris. We can use this to test that specs still pass even when we're in a different time zone.

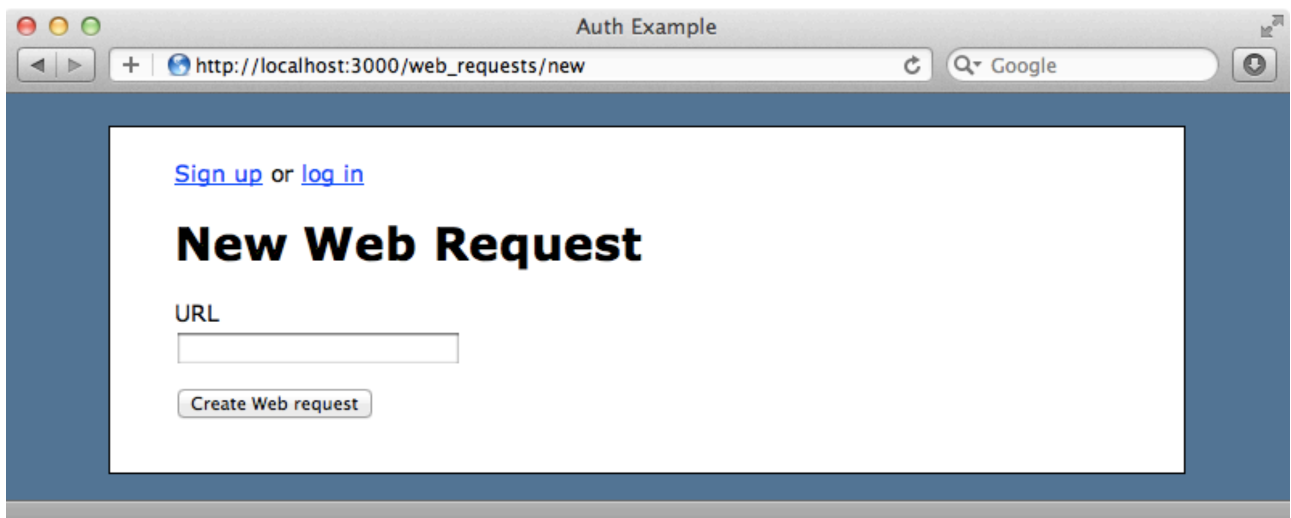
Whenever we test against the current time we should always use something like Timecop so that the time can be handled consistently and won't change when we change time zones or during daylight saving time. We should also use it to test our code in different time zones so that we can check that our application works worldwide.

Testing External Web Requests

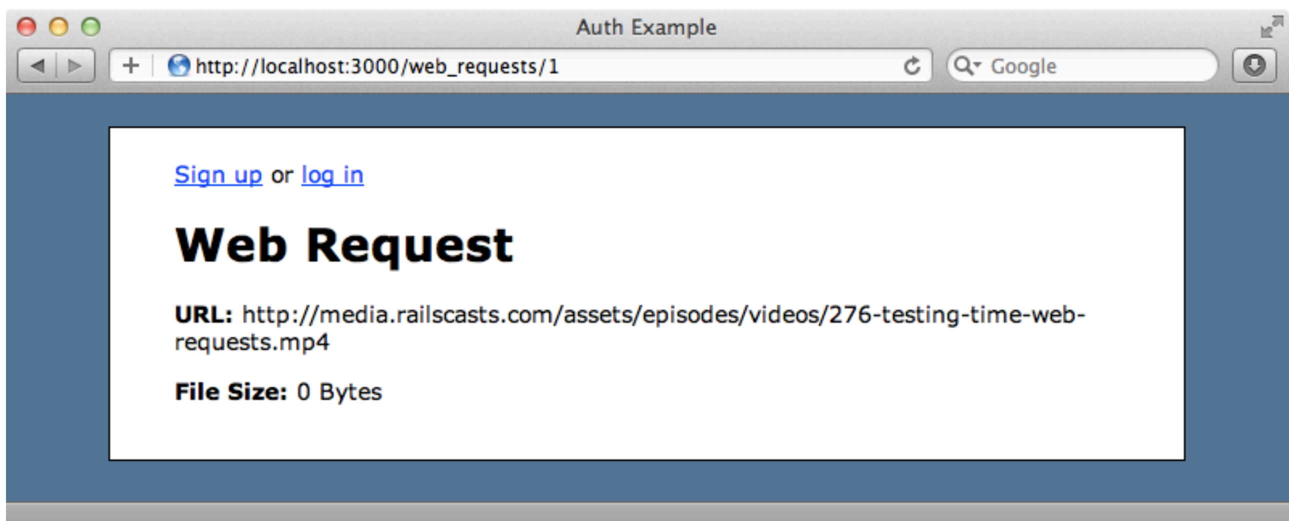
We'll take a look next at testing external web requests. This section is based on a problem that Ryan Bates encountered when rewriting the Railscasts website, specifically the feature that shows the file size for each video format when you hover over the download link.



The file size is fetched from an external web server as the media files are hosted on a separate server. This means that an external web request is made and we'll show now how this was tested. We'll work on this through a small example application. This has a `WebRequest` resource and a form containing a text field that takes a URL.



When we enter the URL of a video in the textbox and submit the form the URL is shown, along with the file's size which is shown as zero bytes.



The file size is zero as we haven't implemented this feature yet. We have a `content_length` method in our `WebRequest` model but this is hard-coded to return `0`. We'll implement this method now using TDD.

/app/models/web_request.rb

```
class WebRequest < ActiveRecord::Base
  def content_length
    0
  end
end
```

There are several gems that can help with testing external web requests, but we're going to use Fakeweb[2]. This gem can be used to register a URI and define what its response should be. When we use `Net::HTTP` to fetch that URI it will instead return the response that we defined instead of making an external request.

Fakeweb is installed in the usual way, by adding it to the `Gemfile` and running bundle. Next we'll add some Fakeweb configuration by making two changes the `spec_helper` file.

---

[2] https://github.com/chrisk/fakeweb

/spec/spec_helper.rb

```
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'
require 'capybara/rspec'

Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}

FakeWeb.allow_net_connect = false

RSpec.configure do |config|
  config.mock_with :rspec
  config.use_transactional_fixtures = true
  config.include(MailerMacros)
  config.before(:each) do
    Timecop.return
    reset_email
    FakeWeb.clean_registry
  end
end
```

Near the top of the file we set `Fakeweb.allow_net_connect` to `false` which stops the specs from making any external HTTP connections. This is useful because it means that if we've left any external requests in the specs they won't slow down the whole test suite and Fakeweb will let us know that the spec is trying to connect to the web. Inside the `before(:each)` we call `Fakeweb.clean_registry` so that each spec starts in the same state.

In the spec for `WebRequest` we'll write a spec that tests that the content length is fetched.

/spec/models/web_request_spec.rb

```
require 'spec_helper'

describe WebRequest do
  it "fetches the content length" do
    FakeWeb.register_uri(:head, ↵
      "http://example.com", :content_length => 123)
    WebRequest.new(:url => ↵
      "http://example.com").content_length.should eq(123)
  end
end
```

We call `FakeWeb.register_uri` to register a fake URL here. The first argument this method takes is the type of request we want to make. We can get the file's size from the header information and so we use `:head` here. The other arguments are the URL and any headers we want, in this case just the Content Length. We then create a new `WebRequest` object that calls this URL and check that the value returned by the `content_length` method is equal to the value set in the header.

Of course when we run this spec it fails as our `content_length` method always returns `0`. To get the spec to pass we need to alter this method so that it returns the actual content length value for the file we're requesting.

/app/models/web_request.rb

```
class WebRequest < ActiveRecord::Base
  def content_length
    uri = URI.parse(url)
    response = Net::HTTP.start(uri.host, uri.port) ↵
      { |http| http.request_head(uri.path) }
    response["content-length"].to_i
  end
end
```

The method now calls `Net::HTTP.start` using the URL that was passed into the model. The block it takes calls `request_head` to get the response headers. Finally it returns the `content-length` header's value.

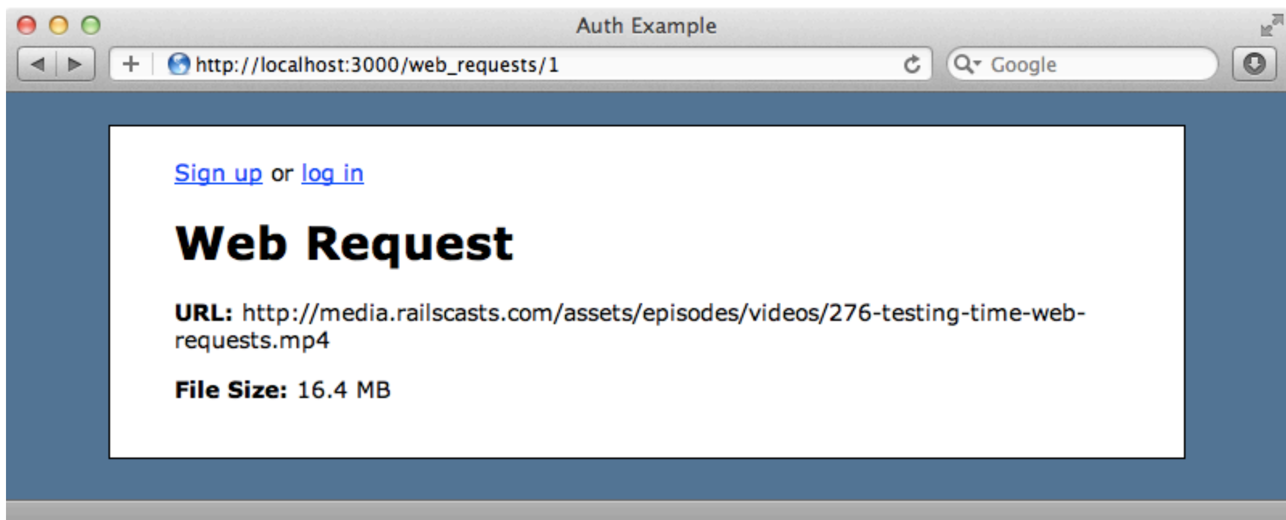
Rails doesn't include `Net::HTTP` by default and so we'll need to `require` it in our application. We'll do this in the `application.rb` file.

/config/auth.rb

```
require File.expand_path('../boot', __FILE__)

require 'net/http'
require 'rails/all'
# rest of file
```

Our specs all pass now and when we reload the page for our web request the correct file size is shown.



If you ever need to handle external web requests in your tests, Fakeweb is a great solution.