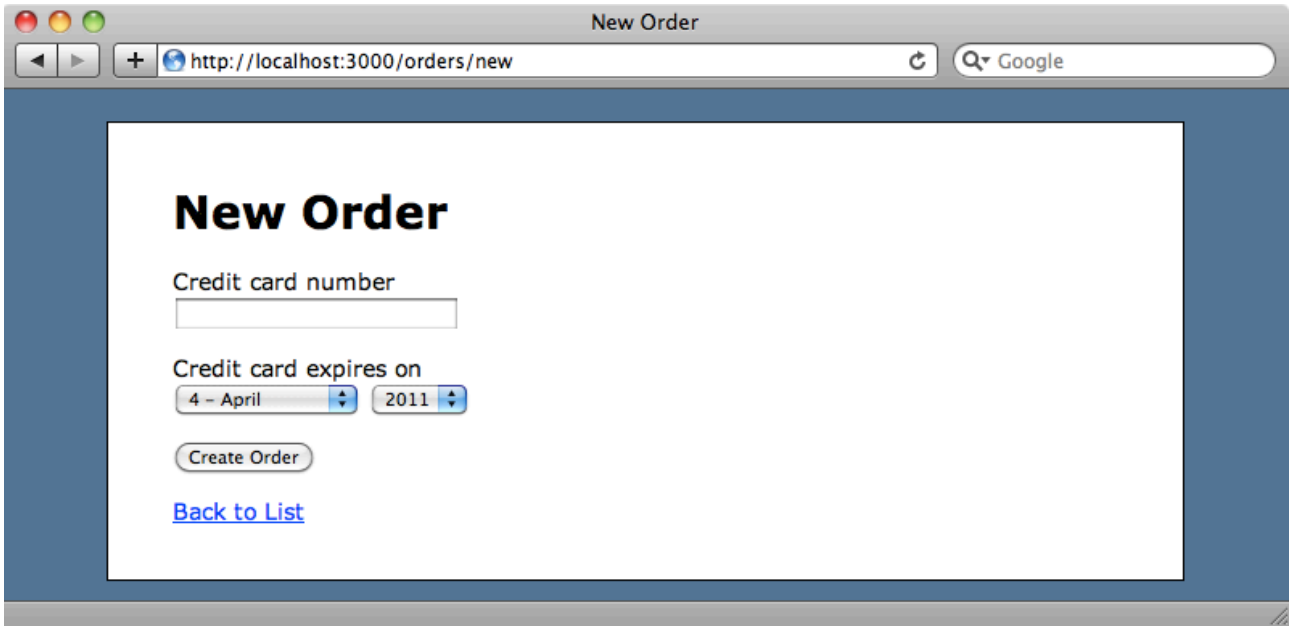
Episode 261

Testing JavaScript with Jasmine

Most Rails developers understand the value of testing the Ruby code in their applications, but JavaScript is often only manually tested by running the application in a browser and checking for errors. As web applications become more complex and have more functionality on the client it would be good to have an automated way of testing JavaScript code. We'll take a look at one way of doing just that in this episode.

The application we'll be working with is shown below. It has a basic order form with a text field for a credit card number. We want to add some client-side validation to this field so that the number entered is validated when the field loses the focus.



We're not adding full credit card validation here but the logic is still fairly complicated so we want to make sure that this code is fully tested. We could consider using acceptance tests which were covered in episode 257 [watch[1], read[2]]. These allow us to test an application at a high level by using Capybara[3] and we can test the JavaScript in our application this way by using Selenium[4].

---

[1] http://railscasts.com/episodes/257-request-specs-and-capybara

[2] http://asciicasts.com/episodes/257-request-specs-and-capybara

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

[4] http://seleniumhq.org/

Sometimes, however, the JavaScript is complicated enough that we need something closer to the code itself, something like unit testing for JavaScript, and this is where Jasmine comes in[5]. Jasmine is a testing framework for JavaScript that works in a similar way to RSpec with `describe` and `it` calls for organising tests. Of course unlike RSpec the tests are written in JavaScript, not Ruby.

Installing Jasmine

A gem is available that makes using Jasmine in Rails applications easier so we'll start by adding it to our application and then running `bundle` to install it. We'll restrict Jasmine to the `development` and `test` environments so that it isn't used in production. Note that we're using jQuery in this application. This isn't required to use Jasmine but it helps.

`/Gemfile`

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

gem 'rails', '3.0.5'
gem 'sqlite3'
gem 'nifty-generators'
gem 'jquery-rails'
gem 'jasmine', :group => [:development, :test]
```

Once the gem has installed we need to run a generator to install the files that Jasmine uses.

```
$ rails g jasmine:install
      create  spec
      create  spec/javascripts/helpers/.gitkeep
      create  spec/javascripts/support/jasmine.yml
      create  spec/javascripts/support/jasmine_config.rb
      create  spec/javascripts/support/jasmine_runner.rb
```

The files are installed in a `spec` directory. Jasmine is generally used in conjunction with RSpec though we can use it just as well without.

---

[5] http://pivotal.github.com/jasmine/

Creating Our First Spec

We're ready now to create our first JavaScript spec. Spec files belong in the `/spec/javascripts` folder and we'll call this one `credit_card_spec.js`. (By the way if you use TextMate there's a bundle[6] available that provides some useful snippets such as `des` for creating a new description and `it` for creating a new `it` block.) Our first spec will test that the entered number is cleaned by having any dashes and spaces removed.

`/spec/javascripts/credit_card_spec.js`

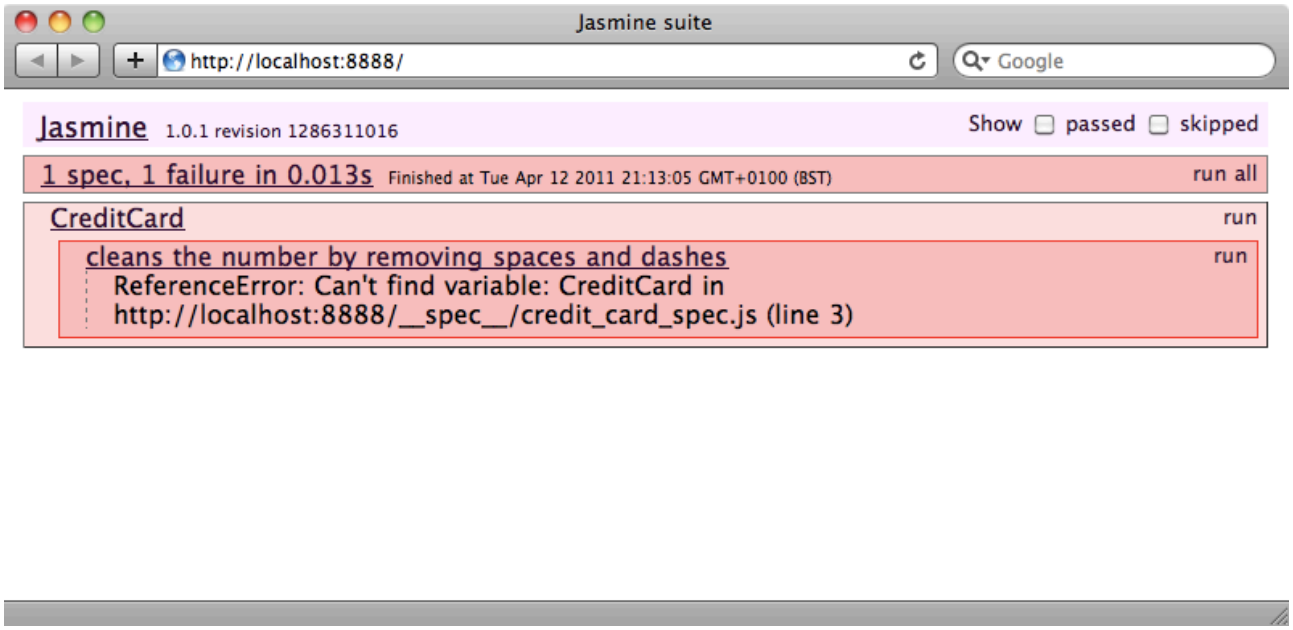
```
describe("CreditCard", function() {
  it("cleans the number by removing spaces and dashes", function() {
    expect(CreditCard.cleanNumber("123 4-5")).toEqual("12345");
  });
});
```

Jasmine uses `expect` rather than the `assert` you might expect to check that a target has a given value. In our test we check that the `cleanNumber` method of a `CreditCard` object returns "12345" when passed "123 4-5". Obviously, as we're doing TDD neither the `CreditCard` object or the `cleanNumber` method exist yet.

We can run Jasmine's spec server by running `rake jasmine`. This will launch a server on point 8888. If we visit that page we'll see the results of running our specs.

---

[6] https://github.com/pivotal/jasmine-tmbundle

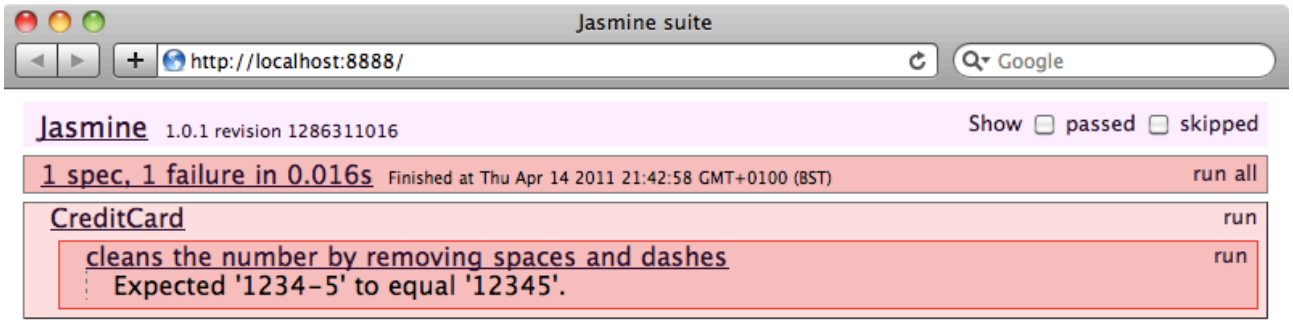


Our spec fails, as expected, and we're going to have to write the code to get it to pass. We'll write the code in a file called `credit_card.js` in the `public/javascripts` directory and there we'll create the object, along with an attempt at the `cleanNumber` function.

/public/javascripts/credit_card.js

```
var CreditCard = {
  cleanNumber: function(number) {
    return number.replace(/[- ]/, "");
  }
}
```

When we reload the Jasmine page now we get a different error. Jasmine no longer complains about the missing CreditCard object, but it seems that `cleanNumber` doesn't remove all of the spaces and dashes.

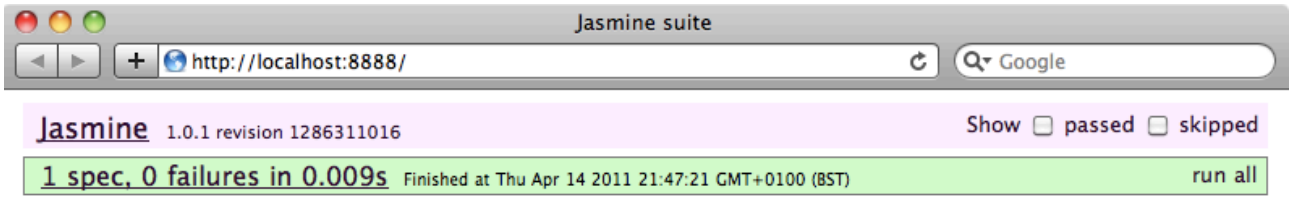


The problem lies with the regular expression which is missing a g to make it a global replace across the whole string.

/public/javascripts/credit_card.js

```
var CreditCard = {
  cleanNumber: function(number) {
    return number.replace(/[- ]/g, "");
  }
}
```

We can check that this has fixed the error by reloading the page again. When we do we'll see that we now have a passing spec.



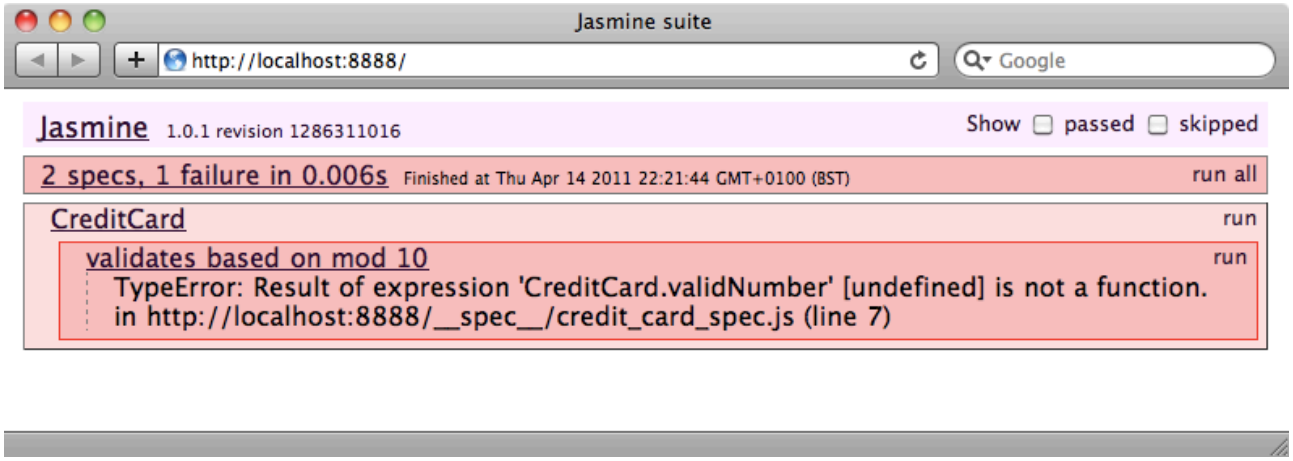
Next we'll expand the validation by adding a mod 10 algorithm[7] to perform checksum validation. We'll write a spec with two expectations to test this code, one that expects a valid credit card number to pass and one that expects an invalid one to fail. These expectations will call a new `validNumber` function, one passing in a valid test Visa number[8] that should pass and the other passing in an invalid number which should fail. Note that we have spaces and dashes in the numbers so that we can test that the validation takes these into consideration.

/spec/javascripts/credit_card_spec.js

```
describe("CreditCard", function () {
  it("cleans the number by removing spaces and dashes", function() {
    expect(CreditCard.cleanNumber("123 4-5")).toEqual("12345");
  });

  it("validates based on mod 10", function () {
    expect(CreditCard.validNumber("4111 1111 1111-1111")).toBeTruthy();
    expect(CreditCard.validNumber("4111 1111 1111-1121")).toBeFalsy();
  });
});
```

When we reload the Jasmine page we again see a failure, this time because it cannot find a `validNumber` function.



We'll write this function now, adding it to the `credit_card.js` file.

---

[7] http://en.wikipedia.org/wiki/Luhn_algorithm

[8] http://www.paypalobjects.com/en_US/vhelp/paypalmanager_help/credit_card_numbers.htm

/public/javascripts/credit_card.js

```
var CreditCard = {
  cleanNumber: function(number) {
    return number.replace(/[- ]/g, "");
  },

  validNumber: function(number) {
    var total = 0;
    number = this.cleanNumber(number);
    for (var i=number.length-1; i >= 0; i--) {
      var n = parseInt(number[i]);
      if ((i+number.length) % 2 == 0) {
        n = n*2 > 9 ? n*2 - 9 : n*2;
      }
      total += n;
    };
    return total % 10 == 0;
  }
}
```

When we reload the Jasmine page now we see two passing specs.

Testing The Validation in The Browser

Now that we have some working validation we can add it to our site. For testing that the validation works on a web page we'll use Capybara and Selenium. If the page had a lot of complex JavaScript code in it it might be difficult to test the validation code in isolation but this can be done and we'll show you how to do that now by using HTML fixtures.

Jasmine-jquery[9] is, as the name suggests, a jQuery extension for Jasmine. This provides fixture support, which means that we can test JavaScript against a simple HTML fragment. It also includes a long list of matchers that we can use with `expect`. We can install jasmine-jquery by running the following `curl` command, which will download the file to the `spec/javascripts/helpers` directory so that it's included automatically.

```
$ curl http://cloud.github.com/downloads/velesin/jasmine-jquery/jasmine-jquery-1.2.0.js > spec/javascripts/helpers/jasmine_jquery-1.2.0.js
```

We'll need to make a change to `src_files` section the `jasmine.yml` file in the `spec/javascripts/support` directory as this is configured to use Prototype by default:

`/spec/javascripts/support/jasmine.yml`

```
src_files:
  - public/javascripts/prototype.js
  - public/javascripts/effects.js
  - public/javascripts/controls.js
  - public/javascripts/dragdrop.js
  - public/javascripts/application.js
  - public/javascripts/**/*.js
```

We'll replace these references with a reference to jQuery. By default every other file in the `javascripts` directory is included but this can lead to problems so instead we'll include each file manually. This will also ensure that we get the load dependencies in the right order.

`/spec/javascripts/support/jasmine.yml`

```
src_files:
  - public/javascripts/jquery.js
  - public/javascripts/credit_card.js
```

We're going need a directory for the fixture files so we'll create one at `/spec/javascript/fixtures`. In there we'll create a file called `order_form.html` that will contain a simple HTML form that will enable us to test our credit card

[9] https://github.com/velesin/jasmine-jquery

validation. All it needs is a text field with an `id` so that we can reference it from JavaScript and a `div` to display any errors.

/spec/javascripts/fixtures/order_form.html

```
<form>
  <input type="text" id="card_number">
  <div id="card_number_error"></div>
</form>
```

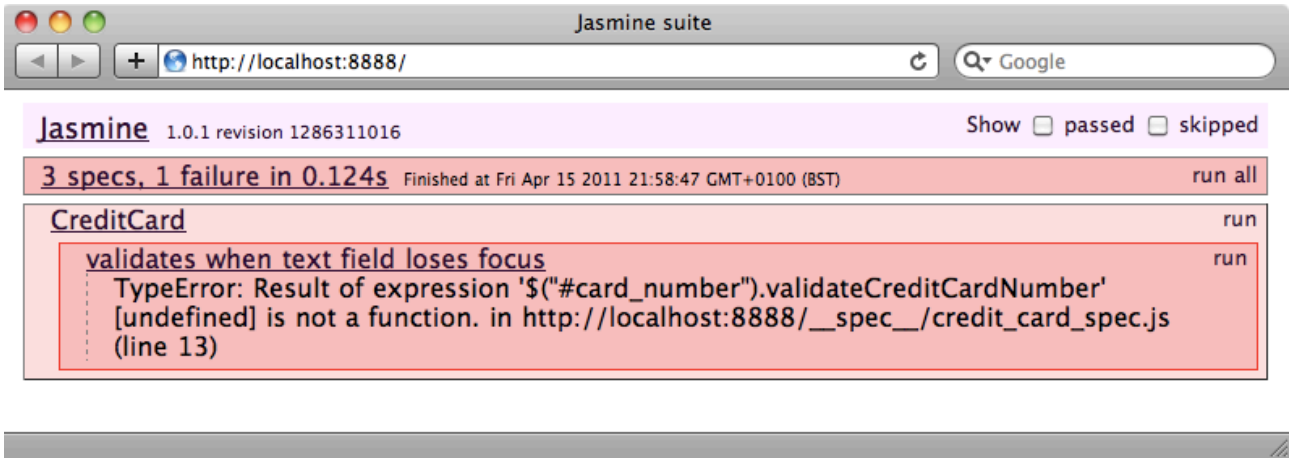
To make our credit card validation code easier to use we'll turn it into a jQuery plugin that we'll call `validateCreditCardNumber`. This plugin will validate the value of any text field it's attached to and fire when the field loses the focus. If there are any validation errors it will set the text of the element whose `id` is the same as the text field but with `_error` appended to it.

Before we create the plugin we'll write a spec that will use our new fixture to test it.

/spec/javascripts/credit_card_spec.js

```
describe("CreditCard", function () {
  // Other specs omitted.
  it("validates when text field loses focus", function() {
    loadFixtures("order_form.html");
    $("#card_number").validateCreditCardNumber();
    $("#card_number").val("123");
    $("#card_number").blur();
    expect($("#card_number_error")).toHaveText("Invalid credit ↵
      card number.");
  });
});
```

In this spec we load the fixture by calling `loadFixtures` then call our new plugin on the text field. We then set the text field's value to an invalid credit card number and call `blur()` on it to cause the plugin to fire. Finally we check that the element that shows the error message has the right error message in it by using the custom matcher `toHaveText` that jasmine-jquery provides. As expected when we reload the Jasmine page now we see a failing spec as jQuery doesn't know about `validateCreditCardNumber`.



To fix this we'll write the `validateCreditCardNumber` jQuery plugin. This plugin will listen to the `blur` event for any elements it's attached to and will validate that element's value when the event fires. It then uses the `validNumber` function we wrote earlier to check that the number is valid and, if not, will show an error.

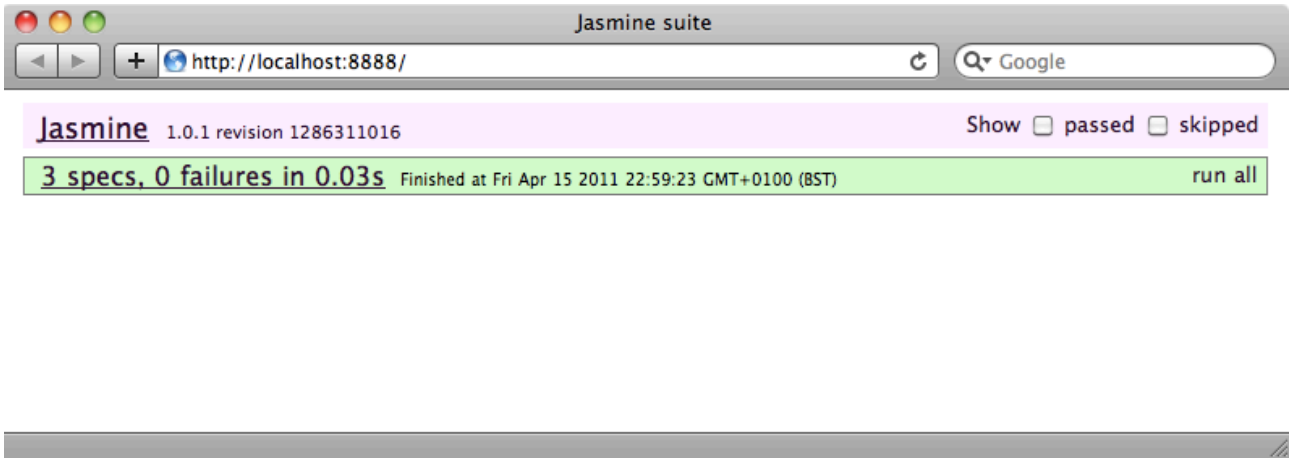
/public/javascripts/credit_card.js

```
var CreditCard = {
  cleanNumber: function(number) {
    return number.replace(/[- ]/g, "");
  },

  validNumber: function(number) {
    var total = 0;
    number = this.cleanNumber(number);
    for (var i=number.length-1; i >= 0; i--) {
      var n = parseInt(number[i]);
      if ((i+number.length) % 2 == 0) {
        n = n*2 > 9 ? n*2 - 9 : n*2;
      }
      total += n;
    };
    return total % 10 == 0;
  }
}

(function ($){
  $.fn.validateCreditCardNumber = function () {
    return this.each(function () {
      $(this).blur(function () {
        if (!CreditCard.validNumber(this.value)) {
          $("#" + this.id + "_error").text("Invalid credit card
number.");
        }
      });
    });
  };
})(jQuery);
```

If we reload the specs page now the spec passes and our plugin seems to work.



Integrating the Plugin

Now that we've tested our plugin in isolation we'll try to integrate it into our application. There are three files that we'll need to change. First we'll include our new jQuery plugin in the layout file.

/app/views/layouts/application.html.erb

```
<%= javascript_include_tag :defaults, "credit_card" %>
```

Next we'll update the form that has the credit card field on it by adding an element to display error messages. As the form is for an Order and the field is called credit_card_number then element has to have an `id` of `order_credit_card_number_error`.

/app/views/orders/_form.html.erb

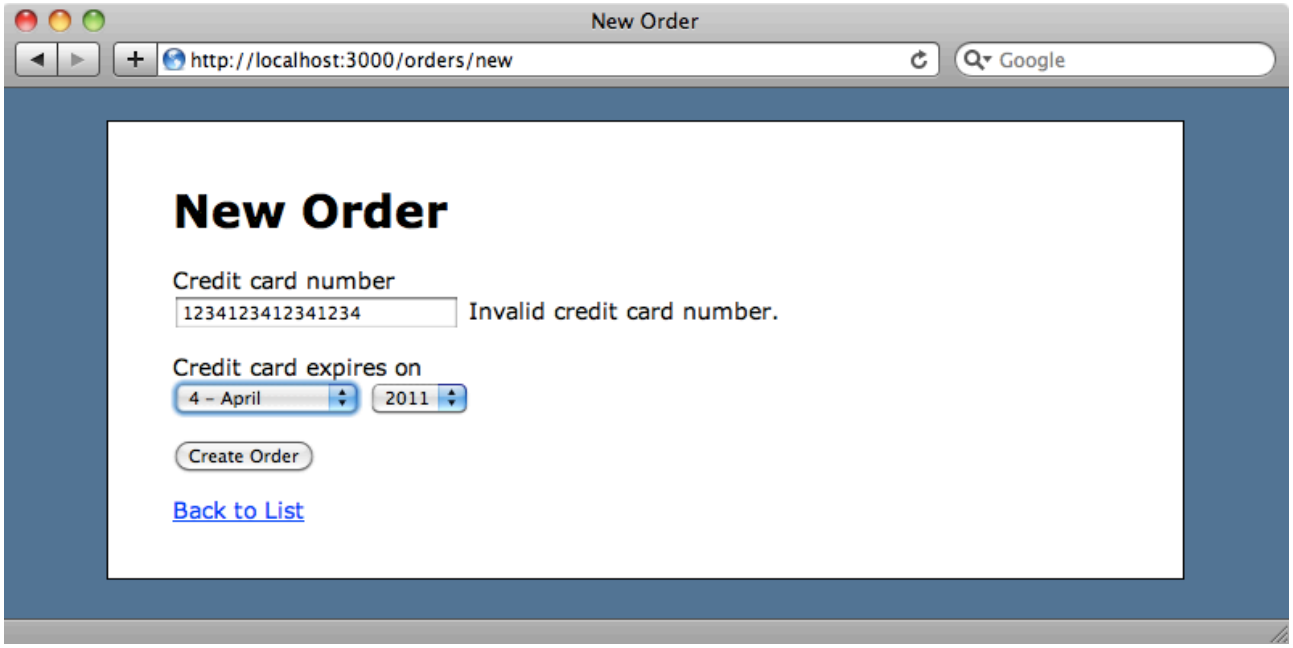
```
<%= form_for @order do |f| %>
  <%= f.error_messages %>
  <p>
    <%= f.label :credit_card_number %><br />
    <%= f.text_field :credit_card_number %>
    <span id="order_credit_card_number_error">
  </p>
  <p>
    <%= f.label :credit_card_expires_on %><br />
    <%= f.date_select :credit_card_expires_on, :add_month_numbers
=> true, :start_year => Time.now.year, :order => [:month, :year]
%>
  </p>
  <p><%= f.submit %></p>
<% end %>
```

Finally we'll add some code to `application.js` to add the validation to the credit card number text field.

/public/javascripts/application.js

```
$(function () {
  $("#order_credit_card_number").validateCreditCardNumber();
});
```

We can try this out now by visiting the New Order page and entering an invalid credit card number. When we tab out of the credit card number text field the error message appears.



There's a bug on the page, though. If we change the credit card number to a valid one the error message remains on the page. The message should be hidden if it's showing when a valid number is entered.

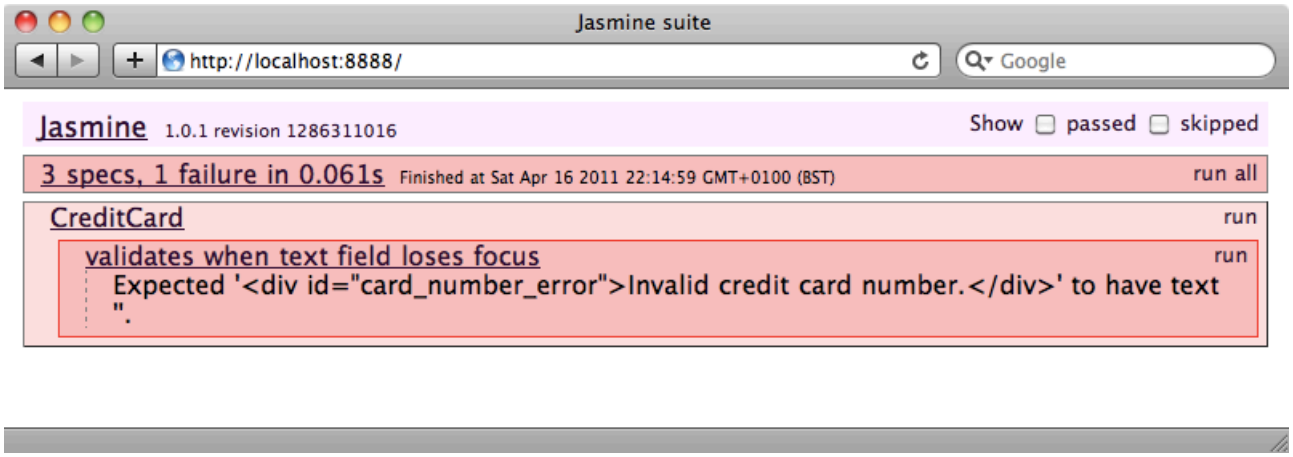
Having found a bug the first thing we need to do is write a failing spec for it. In this case we don't need to write a new spec, we can add an `expect` to the existing card validation spec that checks that the error isn't shown for a valid number.

/spec/javascripts/credit_card_spec.js

```
describe("CreditCard", function () {
  // Other specs omitted.
  it("validates when text field loses focus", function() {
    loadFixtures("order_form.html");
    $("#card_number").validateCreditCardNumber();
    $("#card_number").val("123");
    $("#card_number").blur();
    expect($("#card_number_error")).toHaveText("Invalid credit
card number.");

    $("#card_number").val("4111 1111 1111-1111");
    $("#card_number").blur();
    expect($("#card_number_error")).toHaveText("");
  });
});
```

When we load the Jasmine page we'll see that we've successfully duplicated the bug and the failing spec is shown. We expect the error `div` to have no text but instead it shows a message.



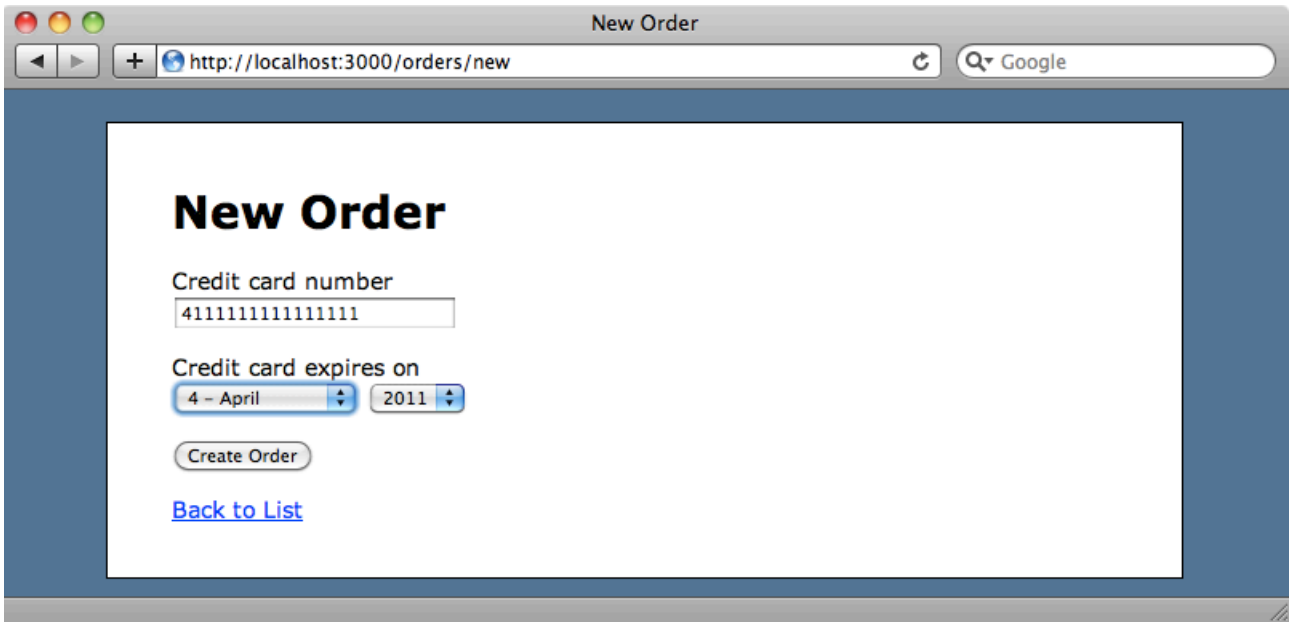
Fixing this bug is fairly simple. We can alter the code in our jQuery plugin so that the error message is hidden if the credit card number that is being validated is valid.

/public/javascripts/credit_card.js

```
(function ($){
  $.fn.validateCreditCardNumber = function () {
    return this.each(function () {
      $(this).blur(function () {
        if (!CreditCard.validNumber(this.value)) {
          $("#" + this.id + "_error").text("Invalid credit ↵
  card number.");
        }
        else {
          $("#" + this.id + "_error").text("");
        }
      });
    });
  };
})(jQuery);
```

When we reload the Jasmine page now the specs again all pass.

Finally we'll test in in the browser again and now, when we enter an invalid number to make the error message show then enter a valid number the error disappears.



A Tip

Instead of reloading the page that shows the specs each time we make a change we can instead run `rake jasmine:ci`. When we do it will open Firefox, run the specs using Selenium and then shows the output in the terminal.

That's it for this episode on Jasmine. Using it in your Rails applications is a great way to test the JavaScript in your application as thoroughly as the Ruby code.