Testing JavaScript

Venkat Subramaniam
venkats@agiledeveloper.com

Abstract

JavaScript is a real language. Code written in it deserves to be tested like any other code. In this article we will look at some of the challenges of testing JavaScript and ways to approach it.

Why test JavaScript?

Why should you test JavaScript? JavaScript[1] is a real language. Like any code, JavaScript code is also likely to change as our understanding of the requirements change, and the design of our code evolves. When you make a change, you want to have confidence that the change is doing what's intended without having undue side effects. Furthermore, you want to isolate issues related to logical errors from issues related to browser differences.

Issues with testing JavaScript

JavaScript is predominantly used on the client side for manipulating user interactions. Ajax has raised the reliance on JavaScript to a greater extent. However, developers generally don't want to consider JavaScript as a real language. How many of us have taken time to study JavaScript to the same level as we have with other languages? A lot of JavaScript is written from examples of code you can find scattered around the web. While these code examples may appear to behave as expected, may not represent good practices.

One of the major concerns with testing JavaScript is that we tend to put JavaScript on web pages, often embedded within the script blocks in HTML pages. It's hard to test code that's hard to reach. Also, the JavaScript tends to depend heavily on the UI components that we're manipulating on the web page. For you to test a piece of code you must be willing to separate the code from its surroundings and also from things it depends on. In other words, testing favors higher cohesion and lower coupling which are characteristics of a good design. We will discuss this using an example below.

JSUnit

While we can perform tests, it helps a great deal to have a tool or framework that makes things easy. Along the lines of JUnit (for Java), JSUnit[2] allows you to test JavaScript. You write an html page which contains the test code. It refers to the JSUnit scripts and the code to be tested as well. You exercise the code you're interested in testing and assert the result, much like in JUnit. JSUnit has a number of capabilities, including automating test runs across machines. Take a look at the documentation and examples for JSUnit for further details. In this article, I will focus more on testability of JavaScript than JSUnit itself.

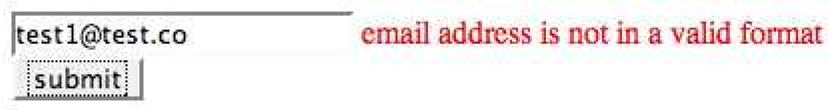
Let's write some JavaScript

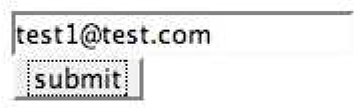
In order to keep this example simple, I will be using plain HTML page here. I don't want to bring in any frameworks and confuse. Assume we have a page that expects a user to enter his/her email address. We want to check, on the client side, if the email address is in a valid format. If the email address is not in a valid format, we want to display an error message and ask the user to reenter. Here is a page that does that:

```
<html>
  <head>
      <script language="JavaScript">
        function validateEmail(emailField, errorField)
        {
          matchResult =
          emailField.value.match('([a-z]|[0-9])+@([a-z]|[0-9])+\.com');
          if (matchResult != null)
          {
            errorField.innerHTML = "";
          }
          else
          {
            errorField.innerHTML =
                        "email address is not in valid format";
          }
        }
      </script>
  </head>
  <body>
    <form>
      <input id="email" name="email"
             onChange="validateEmail(this,
                         document.getElementById('emailError'));"/>
      <span id="emailError" style="color:red"></span>
      <br/>
      <input type="submit" value="submit">
    </form>
  <body>
</html>
```

The JavaScript function `validateEmail()` is checking to see if the value is in a valid format. According to this script, a valid email address contains one or more alphanumeric characters, followed by @ symbol, one or more alphanumeric characters, and finally ends with ".com."

How do I know if this code works? I can take it for a test drive.


test1@test.co
email address is not in a valid format


test1@test.com

The above testing was manual and also, if I make change to the regular expression for validation or if I decide to allow other formats (like .org) for email, manual testing is going to be hard and not reliable.

Using JSUnit

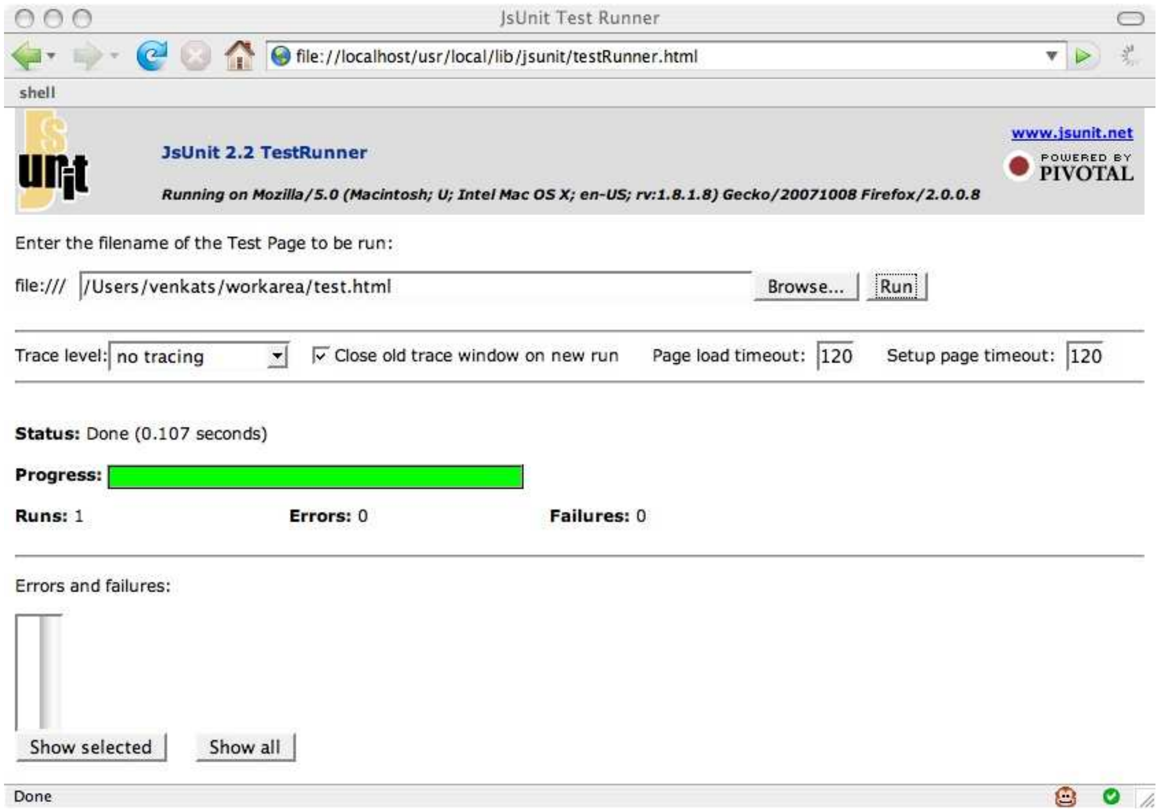
Let’s take a look at using JSUnit with a little JavaScript code to test. Create a file named `test.html` with the following content:

```
<html>
<head>
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">
      <title>A Sample Test Page</title>
      <script language="JavaScript"
            type="text/javascript"
            src="file:///usr/local/lib/jsunit/app/jsUnitCore.js">
      </script>
      <script language="javascript">
        function getLength(field)
        {
            return field.value.length
        }
      </script>
</head>
<body>
<input id="test_field_1" value="abc"/>
      <script type="text/javascript" charset="utf-8">
            function test_getLength()
            {
                  assertEquals(3,
                   getLength(document.getElementById('test_field_1')));
            }
      </script>
</body>
</html>
```

We first load the script `jsUnitCore.js` which contains the code to do the heavy weight lifting of figuring out the different tests that we’re interested in running. Then we have the code to be tested embedded here in this file (not desirable, but we will address this later). The `getLength()` function simply is returning the length of the value property for the field we send to it. Down below we have a input field (in HTML) that contains the value `“abc.”` Then we have the code to exercise the `getLength()` method.

Let’s run this example first. In your favorite browser bring up the `testRunner.html` which is located in the jsUnit directory (depending on where you install jsUnit, you will have to use the path appropriately).

Select Browse and open the above file (`test.html`). Click on Run and see that the test runs and the progress bar turns into a Green bar to indicate a successful run.



JsUnit 2.2 TestRunner

Running on Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.8) Gecko/20071008 Firefox/2.0.0.8

Enter the filename of the Test Page to be run:

file:/// /Users/venkats/workarea/test.html Browse... Run

Trace level: no tracing [✓] Close old trace window on new run Page load timeout: 120 Setup page timeout: 120

Status: Done (0.107 seconds)

Progress:

Runs: 1 **Errors:** 0 **Failures:** 0

Errors and failures:

Show selected Show all

Done

How do we test the `validateEmail()` function we wrote earlier? Hum, that's embedded in the `inputEmail.html` file we created earlier. That is not very desirable. We don't want to import a HTML file here for the sake of testing the JavaScript it contains. The easiest solution (and desirable one as well) is to push the script to a script file and include that into the HTML file (this will help us to not only easily test the code, but also makes it easier to reuse it elsewhere).

Separate the script

Let's move the `validateEmail()` function to a separate file, let's call it `script.js`. The `inputemail.html` will now refer to this file as shown here:

```
<html>
  <head>
      <script language="JavaScript" src='script.js'/>
  </head>
  <body>
    <form>
      <input id="email" name="email"
       onChange="validateEmail(this,
document.getElementById('emailError'));"/>
     <span id="emailError" style="color:red"></span>
      <br/>
      <input type="submit" value="submit">
    </form>
  <body>
</html>
```

Now we can write a test for this function relatively easily.

Let's write a test for email validation

OK, let's write a test for the `validateEmail()` method by providing an email address in a valid format (positive test).

```
<html>
<head>
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">
      <title>A Sample Test Page</title>
      <script language="JavaScript"
                 type="text/javascript"
                 src="file:///usr/local/lib/jsunit/app/jsUnitCore.js">
      </script>
      <script language="javascript"
                 type="text/javascript" src="script.js">
      </script>
</head>
<body>
<input id="test_field_1" value="abc"/>
<input id="test_field_2" value="abc@test.cm"/>
<input id="test_field_3" value="abc@test.com"/>
<span id="err_field" />

<script type="text/javascript" charset="utf-8">
  function testEmailIsValid()
    {
      validateEmail(document.getElementById('test_field_3'),
            document.getElementById('err_field'));
      assertEquals("", document.getElementById('err_field').innerHTML);
    }
</script>
</body>
</html>
```

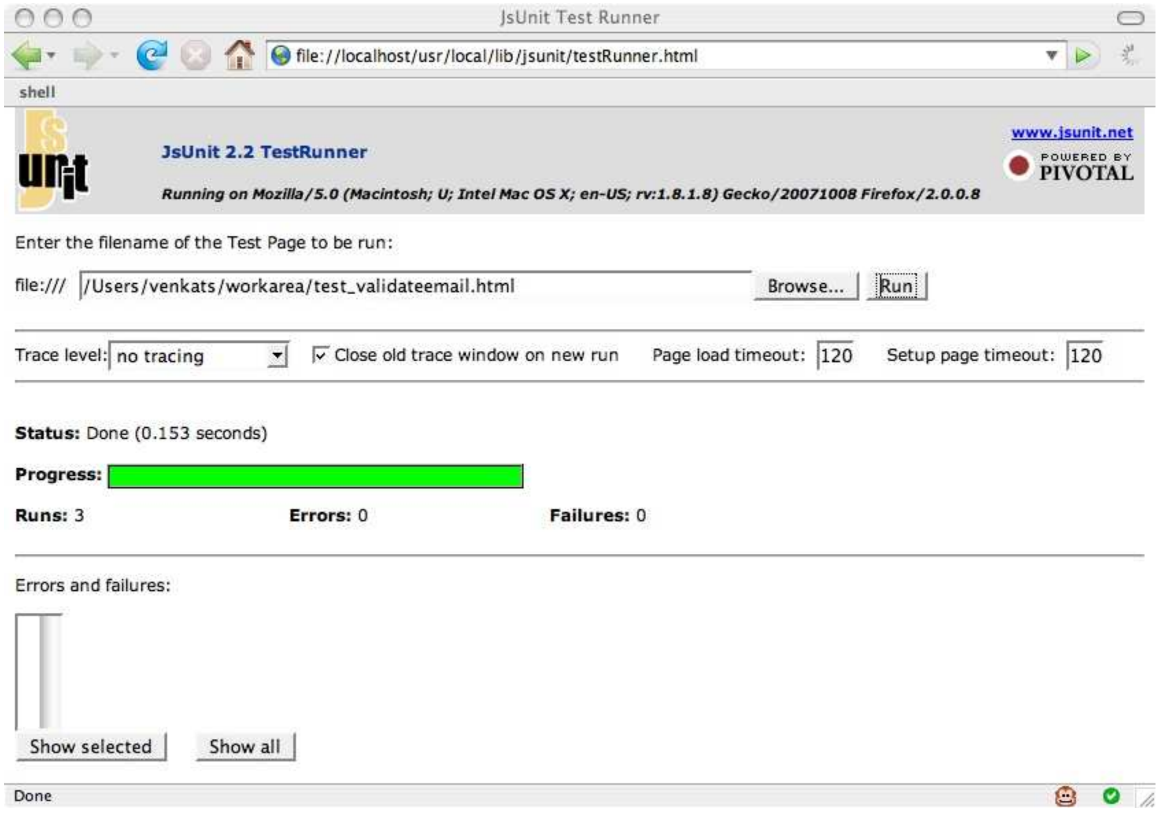
Bring up `testRunner.html` in your browser and load up this file (`test_validateemail.html`) and press Run and you should see a green bar. How can we write a negative test? Let's send a few invalid email address formats and see how that works. Add these two tests after the `testEmailIsValid()` function in the `test_validateemail.html`:

```
  function testEmailMissingAtSymbol()
  {
      validateEmail(document.getElementById('test_field_2'),
                   document.getElementById('err_field'));
      assertEquals("email address is not in a valid format",
                   document.getElementById('err_field').innerHTML);
  }

  function testEmailMissingDotCom()
  {
      validateEmail(document.getElementById('test_field_1'),
                   document.getElementById('err_field'));
      assertEquals("email address is not in a valid format",
                   document.getElementById('err_field').innerHTML);

  }
```

The output from running the tests show that three tests executed successfully:



JsUnit 2.2 TestRunner

Running on Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.8) Gecko/20071008 Firefox/2.0.0.8

Enter the filename of the Test Page to be run:

file:/// /Users/venkats/workarea/test_validateemail.html

Trace level: no tracing ☑ Close old trace window on new run Page load timeout: 120 Setup page timeout: 120

Status: Done (0.153 seconds)

Progress:

Runs: 3 **Errors:** 0 **Failures:** 0

Errors and failures:

We've made progress, but still there are some concerns. While we are able to test the JavaScript, the `<input>` and `<span>` tags in the test file is a smell. Let's modify the expectation on the `validateEmail()` function slightly. If the validation fails, we want to set the focus back on the field. Here is the code change (you can run the `inputemail.html` in the browser to see if the focus is being set back to the email field when you hit tab after entering invalid email address).

```
function validateEmail(emailField, errorField)
{
  matchResult =
      emailField.value.match('([a-z]|[0-9])+@([a-z]|[0-9])+\.com');
  if (matchResult != null)
  {
    errorField.innerHTML = "";
  }
  else
  {
    errorField.innerHTML = "email address is not in a valid format";
    setTimeout("document.getElementById('"
            + emailField.id + "').focus()", 100);
  }
}
```

How do you test if the method sets focus on the text field? What if we need to test for some other UI components' state, color, or other style? We slowly will get dragged into more UI dependence and soon we will be complaining that our unit testing is hard to write and too brittle.

When *unit* testing the `validateEmail()` function, we should be more interested in checking if the code is asking the focus to be set than if the focus is actually being set. Let's go over that one more time. If a method is going to do some UI operation (or some other complicated operation), it is not useful to check the result of that operation. It is more important to see if your code asked for that operation is to be performed. If we approach writing the code from that point of view, it becomes easier to test the code. Also, we can entirely remove the UI from testing by using Mock objects. JavaScript is a dynamic language that makes it very easy to mock. Let's next combine all these thoughts together to modify the code and test.

Separate the UI

Let's separate the code that sets focus into separate function:

```
function validateEmail(emailField, errorField)
{
  matchResult =
      emailField.value.match('([a-z]|[0-9])+@([a-z]|[0-9])+\.com');
  if (matchResult != null)
  {
    errorField.innerHTML = "";
  }
  else
  {
    errorField.innerHTML = "email address is not in a valid format";
    setFocus(emailField);
  }
}

function setFocus(field)
{
      setTimeout("document.getElementById('"
            + field.id + "').focus()", 100);
}
```

Now, let's modify the tests to use a mock for the UI components and we will also mock the `setFocus()` function:

```
<html>
<head>
      <meta http-equiv="Content-Type" content="text/html;
            charset=UTF-8">
      <title>A Sample Test Page</title>
      <script language="JavaScript"
            type="text/javascript"
                  src="file:///usr/local/lib/jsunit/app/jsUnitCore.js">
      </script>
      <script        language="javascript"        type="text/javascript"
src="script.js"></script>
</head>
<body>

<script type="text/javascript" charset="utf-8">
  var EmailFieldMock = function() {this.value = ""; }
  var ErrorFieldMock = function() { this.innerHTML = ""; }

  var setFocusCalled = false;
  function dummySetFocus() { setFocusCalled = true; }

  function testEmailIsValid()
    {
      var emailFieldMock = new EmailFieldMock();
      emailFieldMock.value = "test@test.com";

      var errFieldMock = new ErrorFieldMock();
      validateEmail(emailFieldMock, errFieldMock);
      assertEquals("", errFieldMock.innerHTML);
    }

  function testEmailMissingAtSymbol()
  {
      var emailFieldMock = new EmailFieldMock();
      emailFieldMock.value = "test.com";
      var errFieldMock = new ErrorFieldMock();

      var originalSetFocus = setFocus
      setFocus = dummySetFocus
      validateEmail(emailFieldMock, errFieldMock);
      setFocus = originalSetFocus

      assertEquals("email address is not in a valid format",
                        errFieldMock.innerHTML);
      assertTrue(setFocusCalled);
  }

  function testEmailMissingDotCom()
  {
      var emailFieldMock = new EmailFieldMock();
      emailFieldMock.value = "test@company.co";
      var errFieldMock = new ErrorFieldMock();

      var originalSetFocus = setFocus
      setFocus = dummySetFocus
      validateEmail(emailFieldMock, errFieldMock);
      setFocus = originalSetFocus

      assertEquals("email address is not in a valid format",
                        errFieldMock.innerHTML);
      assertTrue(setFocusCalled);
  }
```

```
</script>
</body>
</html>
```

In the above example, we have done two new things. First, instead of using `<input>` and `<span>` tags, we have created a `EmailFieldMock` and a `ErrorFieldMock`. These objects provide the properties (`value` and `inputHTML`) that our code expects. Second, we have aliases (or replaced) the `setFocus()` method with a dummy mock function that will tell us if the code being tested is calling `setFocus()`.

Go ahead the run the test and see that all three tests pass. Now, comment out the call to `setFocus()` in the `validateEmail()`function and note that the two negative tests now fail.

It took us a little bit of effort, but we are able to separate the code from its dependencies and get the test focused more on it actual functionality.

Other forms of testing

The above example shows how to approach unit testing. Unit testing is essential but not sufficient. You will still have to deal with the UI and also browser differences. Tools like Selenium[2] and CrossCheck[3] can help you with integration tests in addition to the above unit tests.

Conclusion

A unit test should focus on testing a unit of code which is smallest piece of code that does useful work. Unit testing requires us to make the code more cohesive and loosely coupled. Unit testing JavaScript is possible if we follow these basic principles. Ignoring this and depending on UI will lead to code that is hard to maintain and also hard to test.

References

1. “Learning to Love JavaScript” by Glenn Vanderburg in NFJS Anthology 2007 (http://www.pragprog.com/titles/nfjs07)
2. http://www.jsunit.net
3. http://www.openqa.org/selenium
4. http://www.thefrontside.net/crosscheck