JavaScript

Unit Testing Backbone.js Applications

Tools & Libraries

Stephen Thomas
February 25, 2013

Share

> After spending hours, maybe *days*, putting the finishing touches on an awesome new feature for your web application, you're finally ready to see it in action. You add the new code to your JavaScript base, build th
> browser, expecting to be amazed. Then ...
> fine, but some other critical part of your a
> developing the new version – has gone ho
> challenge of backtracking through days o
> the existing code. Happy days are definite

That very scenario has bitten me more than
awhile, you've probably seen it as well. Cons
painful. It isn't really because our new code
development. The real pain is that it took so
development since we knew our application
in which the bug may be hiding. And, though
in a haystack, we have no choice but to dive

In this article we are truly going to banish this scenario from our JavaScript development. No more digging through hours, days, or weeks of code looking for a needle. The principle we'll adopt is a simple one: find any bug *as soon as* we create it. That's right; we're going to set up a development environment and process that tells us immediately when we write code that introduces a bug. Furthermore, the extra effort we put into the process won't go to waste once initial development is complete. The same test code that catches our development bugs will be completely reusable in an integration environment. We can easily incorporate the tests into our source code managements system, blocking bugs before they can even get into our code base.

In the four sections that follow, we'll first look at the tools we need for a JavaScript testing environment. We'll then consider a trivial application, one that's simple enough to understand, yet has all the features and functionality that might exist in a real production web application. The final two sections demonstrate how we can use our environment to test the example app during
is complete, during integration.

Assembling a JavaScript Te

Our unit testing nirvana requires some devel
workbench (yet). The news, both good and b
good news because it gives us options, and
end development today means that there ar
evaluation, let's be explicit about our top two

1. Our environment must support frictionles
2. Tests created during development must b

Execution Environments

For JavaScript coding, there is no better development environment than the modern web browser. Whether your taste is Firebug or Webkit's Developer Tools, the browser supports live DOM inspection and editing, full interactive debugging, and sophisticated performance analysis. Web browsers are great for development, and so our test tools and environment must integrate with in-browser development. Web browsers, however, are not so great for integration testing. Integration testing often takes places on servers somewhere in the cloud (or a least somewhere in the data center). Those systems don't even have a graphical user interface, much less a modern web browser. For efficient integration testing, we need simple command line scripts and a JavaScript execution environment that supports them. For those requirements, the tool of choice is **node.js**. Although there are other command line JavaScript environments, none has the breadth and depth of support to match node.js. In the integration phase, our test tools must integrate with node.js.

Test Framework

Now that we've established that our test too
node.js environments, we can narrow the ch
framework. Many JavaScript test framewor
towards browser testing; getting them work
often requires inelegant hacks or tweaks. Or
problem is **Mocha**, which justifiably describe

> Mocha is a feature-rich JavaScript test fr
> browser, making asynchronous testing si

Originally developed for node.js, Mocha has
browsers as well. By using Mocha as our tes
support both development and integration w

Assertion Library

Unlike some JavaScript test frameworks, Mocha was designed for maximum flexibility. As a consequence, we'll have to choose a few additional pieces to make it complete. In particular, we need a JavaScript assertion library. For that, we'll rely on the **Chai Assertion Library**. Chai is somewhat unique in that it supports all of the common assertion styles – *assert*, *expect,* and *should.* Assertion styles determine how we write tests in our test code. Under the covers, they're all equivalent; it's easy to translate tests from one assertion style to the other. The main difference in assertion styles is their readability. The choice of assertion style depends mostly on which style you (or your team) find most readable, and which style produces the most understandable tests. To see the difference, consider developing a trivial test for the following code:

```
var sum = 2 + 2;
```

A traditional, assert-style test could be writte

```
assert.equal(sum, 4, "sum should equal
```

That test gets the job done, but unless you'v testing, it's probably a little challenging to re style uses `expect`:

```
expect(sum).to.equal(4);
```

Most developers find expect-style assertion assert-style tests. The third alternative, `sho` natural language:

```
sum.should.equal(4);
```

The Chai library supports all three assertion styles. In this article we'll stick with `should`.

Spies, Stubs, and Mocks

Most web apps, including the trivial example we'll consider in this article, rely on third party libraries and services. In many cases, testing our code will require observing – or even controlling – those libraries and services. The **Sinon.JS** library provides a lot of tools for testing those interactions. Such tools fall into three general classes:

- ***Spy***. Test code that observes calls to functions outside of the code under test. Spies do not interfere with the operation of those external functions; they merely record the invocation and return value.
- ***Stub***. Test code that stands in for calls tc The stub code doesn't attempt to replica unresolved errors when the code under t
- ***Mock***. Test code that mimics functions c With mocks, test code can specify the re services so it can verify the code's respo

Along with the Sinon.JS library itself, we can library with **Sinon.JS Assertions for Chai**.

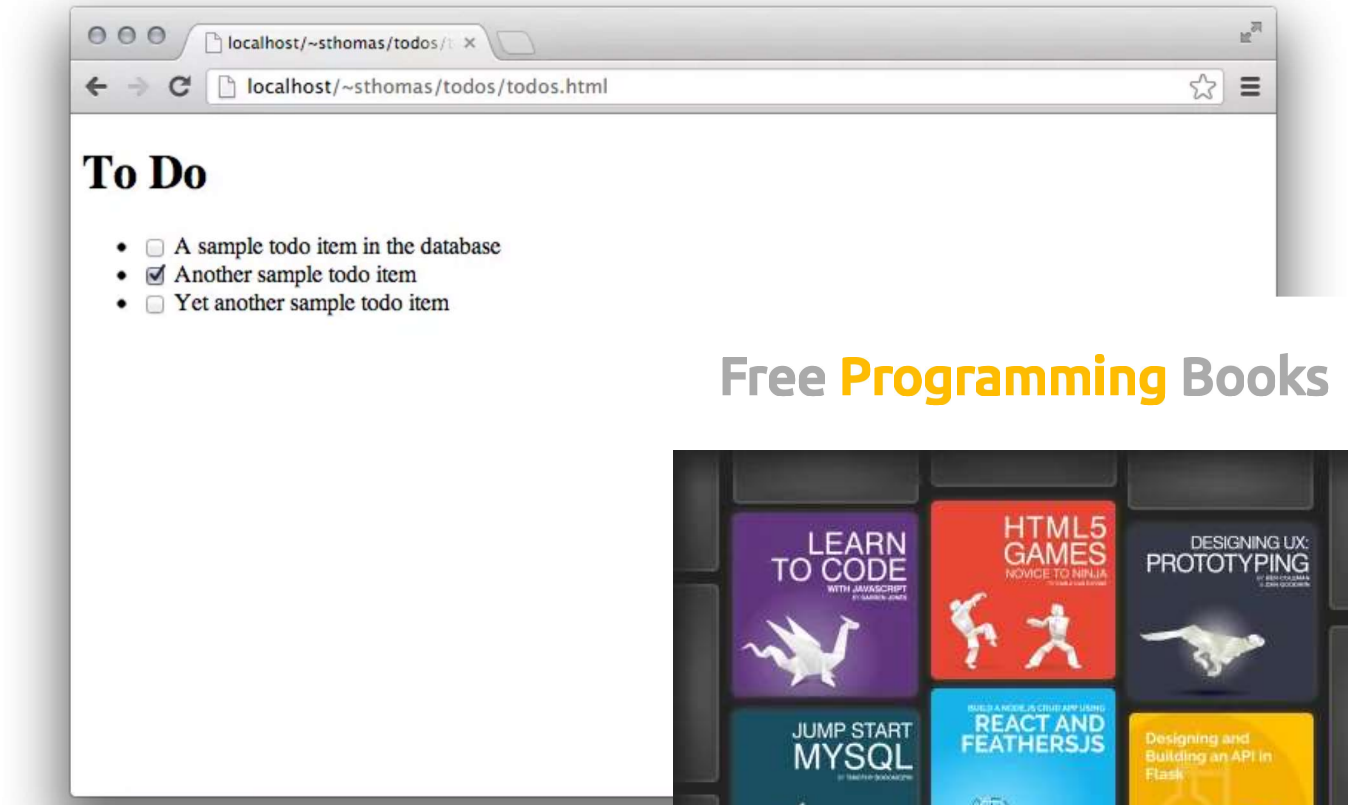
A Unit Test Development Environ

The final tool for our testing workbench is a For our example we'll use **Test'em**. Test'em i run a continuous test environment. We coul

and manage the environment manually; however, Toby Ho (Test'em's creator) has put together an awesome package that can save us the trouble.

The Example Application

To see our testing environment in action, let's consider a simple application. Although pared to its bare essentials, this application includes all the functionality required for a real application. (Complete source code for the application is available on **GitHub**.)



Users can see their list of todos, and they ca
status.

The Todos Database

Our application starts with a database table that holds the information for todos. Here's the SQL that we could use to create that table.

```
CREATE TABLE `todos` (
  `id`       int(11)      NOT NULL AUTO_INCREMENT COMMENT 'Primary k
  `title`    varchar(256) NOT NULL DEFAULT ''     COMMENT 'The text
  `complete` bit(1)       NOT NULL DEFAULT b'0'   COMMENT 'Boolean i
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='To Do items.'
```

And here's how the table might look after we've put some test data in it.

id	title	complete
1	A sample todo item in th	
2	Another sample tod	
3	Yet another sample to	

As the table shows, our todos only include a
indicate whether or not they are complete.

A REST API

Our web application needs access to this da
interface. The API follows Ruby conventions
server technology. In particular:

- `GET api/todos` returns a JSON-encod

- `GET api/todos/NNN` returns the JSON representation of the todo with `id` equal to `NNN`.
- `POST api/todos` adds a new todo to the database using the JSON-encoded information in the request.
- `PUT api/todos/NNN` updates the todo with `id` equal to `NNN` using the JSON-encoded information in the request.
- `DELETE api/todos/NNN` deletes the todo with `id` equal to `NNN` from the database.

If you're not particularly fond of Ruby, the source code includes a complete PHP implementation of this API.

JavaScript Libraries

Our modest application is simple enough to implement in pure JavaScript without any libraries, but we have far bigger plans. We m
will feature amazing functionality and a delig
day, we'll build on a framework that can sup

- **jQuery** for DOM manipulation, event han
- **Underscore.js** to enhance the core langu
- **Backbone.js** to define the structure of th

An HTML Skeleton

Now that we know the components that wil
the HTML skeleton that will support it. There
minimal HTML5 document, some JavaScrip
started.


```
<!DOCTYPE html>
<html lang="en">
```

```
    <head>
      <meta charset="utf-8">
      <title></title>
    </head>
    <body>
      <h1>List of Todos</h1>

      <script src="lib/jquery-1.9.0.min.js"></script>
      <script src="lib/underscore-min.js"></script>
      <script src="lib/backbone-min.js"></script>
      <script src="src/app-todos.js"></script>
      <script>
        $(function () {
          var todos = new todoApp.Todos();
          todos.fetch();
          var list = new todoApp.TodosList({collection: todos});
          $("body").append(list.el);
        })
      </script>
    </body>
</html>
```


Testing During Development

Now that we've selected our tools and speci
development. Our first task is installing the t

Installing the Tools

Even though we'll be developing in the browser, our test environment relies on node.js. The very first step, therefore, is installing node.js and the node package manager (npm). There are executable binaries for OS X, Windows, Linux, and SunOS on the **node.js web site**, as well as a source code for other operating systems. After running the installer, you can verify both node.js and npm from the command line.

```
bash-3.2$ node --version
v0.8.18
bash-3.2$ npm --version
1.2.2
bash-3.2$
```

Everything else we need is conveniently available as a node package. The node package manager can handle their installati

```
bash-3.2$ npm install jquery jsdo
```


Creating the Project Structure

The source code for this example includes a
following 15 files:

```
todos.html
testem.json

api/htaccess
api/todos.php
```