Serverless Single Page Web Apps, Part Two

CSCI 5828: Foundations of Software Engineering
Lecture 21 — 11/01/2016

Goals

- Cover Chapter 2 of Serverless Single Page Web Apps by Ben Rady
  - Hash Events
  - The Router
  - Views
  - etc.

Evolution of Web Apps re: Navigation (I)

- Static Websites
  - The navigation between pages is hardwired
  - <a href="/views/view1.html">…</a>
- Server-Side Web Apps
  - URLs are parsed and handed to a router
  - The router maintains a mapping between URLs and Controllers
  - Controllers perform operations on models and display the results in views
  - The views generate HTML which is sent back to the browser

Evolution of Web Apps re: Navigation (II)

- Serverless Single Page Web Apps
  - Internal URLs are used to move between views
    - Makes use of hash events that are triggered by traversing to a link that starts with a hash tag
    - <a href="#employees/5">…</a>
  - We use JavaScript to be informed when someone clicks on one of these links (or when code redirects the browser to one of these URLs)
  - A JavaScript router handles the event and maps the hash events to views in the application
  - The views are copied and dynamically inserted into the DOM to make the browser render them; the old view is removed from the DOM (of course)
    - this simulates a link "traversal" without any calls out to a server

Our Router

- The book describes the router that we are going to implement like this:
    - hash events trigger the creation of dynamic markup
    - the router contains routes that map hash URLs with view functions
    - Each function creates both the markup and behavior of a single view
    - Our application will make use of JQuery to load the router once our page has finished loading;
        - the router will register interest in hash events to handle the navigation
    - At this point, we'll have a functional single page web application

Example of Hash Anchors

```
<html>
 <body>
  <p><a name="top">This is the top of the page.</a> Head to the
  <a href="#bottom">bottom</a>.</p>
  <p>Lorum ipsum…</p>
  … <lots of these>
  <p>Lorum ipsum…</p>
  <p><a name="bottom">This is the bottom of the page.</a> Head
  to the <a href="#top">top</a>.</p>
 </body>
</html>
```

- Here's a page that demonstrates the use of links that begin with hash marks

DEMO

Creating a Testable Router

- For this chapter, the book adopts a "test first" approach
  - We're going to develop our router by first writing tests
  - We'll tackle making sure that the tests can
    - inspect markup (i.e. the DOM structures loaded into the browser)
    - invoke functions (located in app.js)
    - trigger events (which in turn cause our app to do something)
      - trigger events will cause callbacks to invoke
      - which may cause markup to be modified
      - which can then be verified by our tests for accuracy

Jasmine Testing Framework

- The book makes use of Jasmine to write our tests
    - http://jasmine.github.io/
- Jasmine is a behavior-driven development framework
    - Test suites are defined using a function called "describe"
    - Individual tests are defined by a function called "it"
    - Assertions are made via a function called "expect"
- We'll see examples soon!
- Our prepared workspace already includes a test runner
    - Just add "/tests/index.html" to a URL to invoke it; currently we have no tests

Starting down the path of test-driven design

- We want our tests to drive our development work
    - To do this, we need to write a test that asserts something about the program that it currently cannot do
        - Right now, we have a static website, so…
            - our test will assert that we can go to a second view
            - We'll call this view the "problem view" as it will display a Javascript puzzle for us to solve
        - After we write the test, we'll add enough code to the app to get it to transition to this new view

Our First Test

- The reason that Jasmine's test case method is called "it" is because it should be used in a sentence. If we need to show a problem view, we can write the test like this:

```
describe('LearnJS', function() {
 it('can show a problem view', function() {
 });
});
```

- With this, we can see that our test suite is just a call to a function that has a callback. Likewise, a test case is a call to a function that also has a callback. The callbacks run when they get invoked by our test runner
  - Add this code to /public/tests/app_spec.js and run the test runner again
    - It will now say we have one test but it specifies no expectations
    - This is an example of TDD; write a test, watch it fail.

Extending the test

- We're going to have our router be a function called `showView()` in a module (or namespace) called `learnjs`. We're going to ask it to display the first problem using the URL "`#problem-1`". We're then going to specify an expectation about the state of the world

```
describe('LearnJS', function() {
 it('can show a problem view', function() {
   learnjs.showView('#problem-1');
   expect($('.view-container .problem-view').length).toEqual(1);
 });
});
```

- The expectation makes use of JQuery to retrieve a DOM element using a CSS specifier, then get its length, and specify that the length should equal 1
  - Add this test, run it, and watch it fail. Why? None of this code exists!

Making our Test Pass

- With test-driven design, no production code is written except to make a failing test pass.
  - In order to get this test to pass, we're going to have to do a few things
    - We need to create our namespace: `learnjs`
    - We need to create our function: `showView()`
    - The function actually needs to do something
      - In this case, it needs to update our page when passed the URL `"#problem-1"`
    - The updated view needs to have the CSS structures specified in our test present, so that query returns a structure that has a length of one

Step 1: Create the name Space

- In app.js, we add the following code:

```
'use strict';
var learnjs = {};
```

- We can run our test and see that it **still fails**
    - *but with a different error message*.
- That's progress! :-)
- What does the code do?
    - It creates a JavaScript object and binds it to the global symbol: `learnjs`
        - All other functions will go inside of this object to avoid cluttering the global name space

Step 2: Create the showView function

```
'use strict';
var learnjs = {};
learnjs.showView = function(hash) {
  var problemView = $('<div class="problem-view">').text('Coming soon!');
  $('.view-container').empty().append(problemView);
}
```

- Here we ignore the "hash" parameter and just create what we need to make the test pass. We need a div that has a CSS class called "problem view" and we need to add that div to another div with the CSS class "view-container".
  - We use JQuery to do all of these DOM manipulations
    - Note that the call to `empty()`, clears out the previous content: i.e. the "landing page"
  - Our test still fails; that's because our page does not yet have a "view-container". However, now, the test fails because the expectation fails.

Progress!

Why is the test failing? (I)

Our showView() function tries to access a div with the class "view-container". This div does not exist. This means our code cannot append our "problem-view" div to it!

Why is the test failing? (II)

/index.html

Another problem is that our markup is located here…

/tests/index.html

But our tests are located here. When the test runner runs, the DOM that it has access to, is the DOM specified by THIS document (not the one above).

How to fix it? (I)

/index.html

Our prepared environment is set-up to look for a div with the class "markup" and append all of its content to…

/tests/index.html

… the DOM tree located in this file. This makes the markup available to our tests.

How to fix it? (II)

- Thus, to fix our test, we need to
  - Edit /index.html to contain a div with the class called 'markup'
    - All existing content is stored here
  - Edit the div that was previously tagged with "container" (for the Skeleton framework) and add an additional CSS class called 'view-container'.
- With these two changes, when our test runs:
  - the markup will be copied to our tests/index.html page
  - a "view-container" div will exist, so…
    - … our JQuery expression will find it and add the problem-view div
    - This will ensure that this div is retrieved and our test will pass

Adding ACTUAL Routes (I)

- A funny aspect of test-driven design is that when you first implement a test you are told to:
  - Do the simplest thing to make the test pass
- In our first test case, we did the minimal amount of work to switch to a new view
  - We took the current view and emptied it out of all content (i.e. deleted the landing page)
  - We then created a single div element and inserted it into the appropriate container div
  - We then asserted that only one problem view exists in the DOM
- BUT THAT DOESN'T REALLY SOLVE OUR PROBLEM! (It just gets us started)

Adding ACTUAL Routes (II)

- Now, we are going to modify our program to have an actual implementation of routes and an actual second view
    - At the end, we'll have a landing page and a "problem" page
- We will create view functions: functions that produce views for our program
    - We will associate hashes with view functions; these associations are routes; our routes will be a JavaScript object that will map hash names to view functions
    - What's nice about this approach is we can test view functions separately from the router if we want
- Our new workflow will be:
    - hash change event => lookup route => create view => update display

Adding ACTUAL Routes (III)

- We will continue to use our showView() function as a way to drive progress
  - We start by adding a new test that asserts if the hash is empty, then show the landing page

```
describe('LearnJS', function() {
 it('shows the landing page on empty hash', function() {
   learnjs.showView('');
   expect($('.view-container .landing-view').length).toEqual(1);
 });
});
```

- We add this test to the existing test suite; run the tests to see this test fail!
  - Note: our previously specified test still passes, just the new one fails.
    - That's because we are currently ignoring the hash in `showView()`

Adding ACTUAL Routes (IV)

- To make the new test pass, we need to make two changes
    - Add a div around the landing page with the CSS class "view-container"
    - Change our code
        - Turn the existing "problem-view" code into a view function
        - Create a routes object and associate "#problem-1" with that function
        - Modify showView to look up routes and invoke them if a view function is found for a particular hash
        - This code (next slide) makes the second test case pass because if the hash is empty, then the code doesn't change the page, and the default page displays the landing page already (which will make the test pass)

Adding ACTUAL Routes (V)

```
'use strict';

var learnjs = {};

learnjs.problemView = function() {
  return $('<div class="problem-view">').text('Coming soon!');
}

learnjs.showView = function(hash) {
  var routes = {
    '#problem-1': learnjs.problemView
  };
  var viewFn = routes[hash];
  if (viewFn) {
    $('.view-container').empty().append(viewFn());
  }
}
```

- `problemView()` is now a function that returns a new div. It gets associated with the problem-1 hash in the new routes object. `showView()` looks up the hash and updates the view if it gets back a view function

View Parameters

- We now need to be able to handle more than one problem view
  - All we can do currently is handle "#problem-1"
    - We're going to start parsing the hashes to pull parameters out of them
    - We will then pass these parameters to view functions that can do what they want with them
      - Display different problems, for instance
- To test this new behavior, we're going to make use of another feature of Jasmine which are called "spies"
  - Spies stand in for functions and then can respond to assertions on how they were accessed during a test (pretty cool feature!)

Testing the interaction of the router and view

- We will make use of Jasmine's spyOn function to spy on the calls made to the problemView view function
  - We will then be able to make assertions about how it was invoked
  - Here's how we can set-up the test

```
it('passes parameter to view function', function() {
  spyOn(learnjs, 'problemView');
  learnjs.showView('#problem-42');
  expect(learnjs.problemView).toHaveBeenCalledWith('42');
});
```

- The test above asserts that if we call showView('#problem-42') then the problemView() function needs to have been called with the string '42'
  - This would indicate that we parsed the hash correctly and sent the view parameter to the view function

Run the test to see it fail!

Making the Third Test Pass

- To make this new test pass, we add code to showView() to
  - parse the hash; we use the '-' as a delimiter and split on it
  - we pass the first part of the string to routes to look up the view function
  - we then pass the second part of the string to the view function that was returned

```
learnjs.showView = function(hash) {
  var routes = {
    '#problem': learnjs.problemView
  };
  var hashParts = hash.split('-');
  var viewFn = routes[hashParts[0]];
  if (viewFn) {
    $('.view-container').empty().append(viewFn(hashParts[1]));
  }
}
```

- Run the app to see the test pass, even though problemView doesn't use its param

Testing the Problem View (I)

- Let's set the stage for testing the problem view
    - We want to force that function to handle its parameter
- First, the test

```
describe('problem view', function() {
  it('has a title that includes the problem number', function() {
    var view = learnjs.problemView('1');
    expect(view.text()).toEqual('Problem #1 Coming soon!');
  });
});
```

- Here we assert that if we pass in a "1" then we should see "Problem 1" in the view that is created
    - Note: we embed a new call to describe() within our original describe() call; this allows us to partition tests according to purpose

Testing the Problem View (II)

- Now the code; we modify problem view to create the correct title

```
learnjs.problemView = function(number) {
  var title = 'Problem #' + number + ' Coming soon!';
  return $('<div class="problem-view">').text(title);
}
```

- Run the test again and watch it pass!

Loading Our Application

- All we have been doing so far is running tests
  - We still don't have our application configured to ensure that our router has been properly registered to receive hash events
    - Before we do that, we need to make sure the application is loaded
- To do that, we're going to use jQuery to tell us when the browser has loaded our index.html page and has created all of the necessary DOM structures to model it in the browser; We will use jQuery's ready event and pass in a callback
  - Our callback will be a function that will look up the original hash value (which will be empty when we load index.html) and call our router
    - Our router will do nothing with an empty hash and our browser will then display the landing page as we intend
- Lets also right a test to test our callback which is called `appOnReady()`

Responding to Hash Events (I)

- The last piece of the puzzle is getting our application to respond to hash events
    - That is, if the user clicks on a link that contains a hash, we want to be notified; we don't want the browser responding to the event
- First, we add a test!

```
it('subscribes to the hash change event', function() {
  learnjs.appOnReady();
  spyOn(learnjs, 'showView');
  $(window).trigger('hashchange');
  expect(learnjs.showView).toHaveBeenCalledWith(window.location.hash);
});
```

- Our test makes sure our app is loaded; it then creates a spy on the `showView` function; it then triggers a hash event; and asserts that `showView` was called

Run the test to see it fail!

Responding to Hash Events (II)

- To make this test pass, we need to update our appOnReady function to actually register interest in hash change events. Here's the code:

```
learnjs.appOnReady = function() {
  window.onhashchange = function() {
    learnjs.showView(window.location.hash);
  };
  learnjs.showView(window.location.hash);
}
```

- All we do is register a callback with the property "onhashchange" that exists on the window object;
  - our callback simply calls our showView function with the value of the current hash

Run the test to see it pass!

One last piece of the puzzle

- We can now update our landing page to go to the first problem
  - We update the href of the Start Now! button
- `<a href='#problem-1' class='button button-primary'>Start Now!</a>`
- Previously, the href had been empty

- Now, we reload the application (not the test runner)
  - When we click on the button, our problem view appears
    - The hash event was detected; our router was notified; it called the problemView, which updated the DOM
- Done! You can now use './sspa deploy' to see this site in action on AWS

Summary

- In this chapter, we have touched on a number of topics
  - Test driven design
    - Jasmine: test suites, test cases, spies
  - Handling of events in the browser (onReady() and onHashChange())
  - Implementing a router for a single page web application
  - Using jQuery to create and query the DOM

- Next Time: We'll look at transforming our base application with additional features expected of serverless single page apps