Extracted from:

Serverless Single Page Apps

Fast, Scalable, and Available

The Pragmatic Bookshelf
Raleigh, North Carolina

The Pragmatic Programmers

Serverless Single Page Apps

Fast, Scalable, and Available

Ben Rady

edited by Jacquelyn Carter

Serverless Single Page Apps

Fast, Scalable, and Available

Ben Rady

The Pragmatic Bookshelf
Raleigh, North Carolina

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Nicole Abramowitz, Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

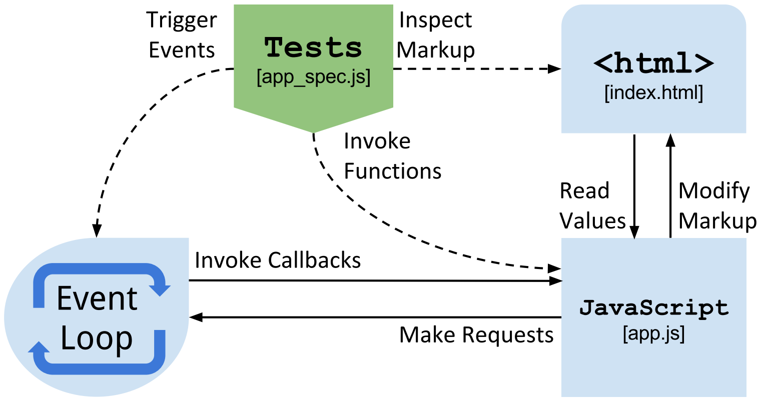
Printed in the United States of America.
ISBN-13: 978-1-68050-149-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2016

Designing a Testable Router

Although there are some exceptions, it's generally not possible to run AWS services on your local workstation. With all of our application's back end running in AWS, you might be worried that it will be difficult to test. If we want to create a fast feedback loop that tells us if things are working, we're going to need to diverge from the traditional approach of running a complete service stack in a development environment in order to test the application while we build it. Instead, we're going to test our router using unit tests written with a *test first* approach. Designing the router by writing the tests first will ensure the resulting design is easy to test. Although our focus on testing will be limited to this chapter, you'll be able to apply these same techniques throughout the rest of the book.

The design of the router has a profound impact on the overall testability of the application, which is why we're starting there. We want to create a suite of automated tests that can quickly check if our app is working, without being dependent on back-end services to run the tests. To build the router, we're going to drive out its behavior incrementally with tests. We'll write these tests one at a time, in parallel with the application, to ensure that all of the code is testable. We'll eschew testing entire workflows and focus instead on testing small bits of behavior to create tests that run in a few milliseconds. We'll be able to avoid race conditions and other timing problems in the tests because they won't need to make asynchronous requests to a server.

One way to create a testable application is to build "seams" into the design—in other words, clear boundaries where tests can easily invoke behavior, inspect output, and simulate interaction. One seam in our application is the markup. If we make the markup available to the tests, they can inspect it. Encapsulating the JavaScript creates another seam. Making the JavaScript accessible to the tests means they can invoke it to verify behavior. The browser itself can act as a seam, allowing our tests to simulate user actions by triggering events. You can see some of these testing seams in this diagram.



Creating an application that is easily tested naturally promotes decoupling,[1] which improves the design. As we build the router, we'll also use these tests to improve the design of the code through *refactoring*. We'll work incrementally to ensure that we only add what we absolutely need, which will keep the code as simple as it can be. The result should be a reliable suite of tests that support a simple, functional, and easily testable application.

Running Jasmine Tests

To write the tests, we're going to use the *Jasmine* testing framework. Jasmine is similar to RSpec and most xUnit frameworks, so if you've used any of those before, you should be comfortable. If not, you can follow along, or you can check out the Jasmine documentation first.[2] It's pretty simple once you get the pattern.

The prepared workspace includes a test runner. If you have the application loaded, add /tests/index.html to the URL, and you should see the test output. For now, it should say we have no tests. Also, just like the rest of the application, using the LivePage plugin or a LiveReload server means this page will reload automatically as you make changes to the app or tests.

Running Tests in Production

1. https://en.wikipedia.org/wiki/Coupling_(computer_programming)
2. http://jasmine.github.io

When someone reports a bug, and you can't reproduce it anywhere, how do you fix it? People often interpret "Works on my machine" as an accusation that the bug being reported isn't real—that somehow, the user is creating the problem instead of the software. When I say that to someone, it's actually a call for help. Not being able to reproduce a problem leaves me stymied. After all, if I can't reproduce it, how will I know when it's fixed?

When you find yourself in this situation, you can use the test suite as a sanity check to ensure your assumptions about the app still hold true, even in environments you can't access directly. The deploy script in the prepared workspace not only deploys the application to production, but deploys the tests as well. This means you can run the tests from any device where you can run the app.

So if you have a user who's reporting a problem, a quick way to troubleshoot what's going on can be to ask the user to browse to /tests and make sure they all pass. If they don't, the user can copy and paste the output and send it to you. You can then try to reproduce the failing test as a proxy for reproducing the error that the user is reporting.

Jasmine organizes tests by enclosing them in callbacks passed to two functions: describe and it. The it function is used to write individual tests, while the describe function allows us to add context and setup around the tests. We're going to add one outer describe to hold all the tests for the app, and then add an it for the test we want to write.

Writing the First Test

Before we can write a test, we need to figure out what behavior we want. Expressing this behavior in plain English—and naming the test accordingly—will make it clear what's going on when we come back to it later. Test names should focus on why the code does what it does, rather than how it does it.

The reason we need a router is to support multiple views in the application. Right now, the landing page is our only view. To drive out the router behavior, we'll write a test that asserts that there's another view that can be created. This will not only help us create the functionality we want, but it will clearly explain why it's needed.

The second view we're going to create will show the programming problems in the app, so we're going to call it the *problem view*. This will be the primary view that users interact with in the app. We're not really sure what we want it to look like yet, but we can at least add enough behavior to the app to get it to transition from the landing page to this new view. Being able to do this is a small step that will help us make progress…even if the new view doesn't have any real content.

Jasmine will combine the text in the describe and it functions to create the full name of the test. We want to ensure those names are readable. The name of Jasmine's it function also gives us a hint about how to name the test. You should be able to read it, like "It can show a problem view." In the prepared workspace, open public/tests/app_spec.js and add the new test there.

learnjs/2000/public/tests/app_spec.js

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

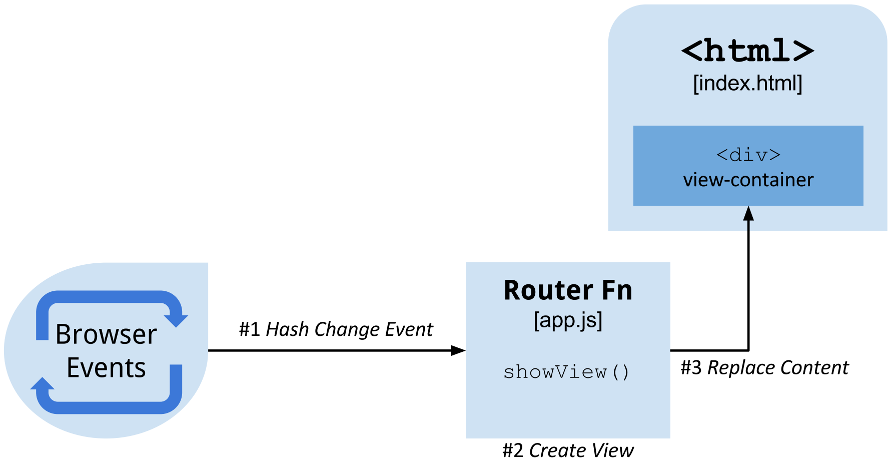
Now that we have a name for our test, we can write the test itself. The action we're going to take in this test is to invoke a JavaScript function that we'll call the *router function*. The router function's job is to find and display the appropriate view, given the current route as defined by the URL's hash. We're going to call this function showView and put it in a *namespace*[3] called learnjs. This function will be responsible for creating the view markup and adding that markup to our app. It will take the URL hash as a parameter, which it will use to select the view. For this test, we'll pass in a hash value that represents the route for the first problem in our app: #problem-1.

The showView function doesn't exist yet, but that's OK. We're using the test to drive out the design of the function. The figure on page 11 shows our first pass at building a router. Note that there's nothing in our app that will call our router function. Eventually, another part of our app will invoke showView() when the browser fires a hashchange event. It will pass in the current hash value provided by the document location API,[4] using window.location.hash. When showView is invoked, we'll create the view and append it to the page.

The assertion we're going to make in our test is that the router has placed the problem view markup in a *view container*. The view container is another essential part of our router. It's the element the router will use to hold the view's markup. Any markup inside the view container will be replaced when the router adds a new view.

To write the assertion, we're going to use jQuery to select[5] elements from our page. We'll assert that our app contains an element with the view-container class,

3. http://eloquentjavascript.net/10_modules.html#h_NitCO6r9Hn
4. https://developer.mozilla.org/en-US/docs/Web/API/Window/location
5. https://learn.jquery.com/using-jquery-core/selecting-elements/



and inside that element is another element with the problem-view class. We do this by selecting these elements with jQuery, then asserting that the number of elements we selected is equal to one.

learnjs/2100/public/tests/app_spec.js

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

If your LivePage/LiveReload is working properly, when you save the spec file the test runner should reload and run the tests, showing you the test fails:

```
1 spec, 1 failure
LearnJS can show a problem view
ReferenceError: learnjs is not defined
```

This test fails as we expect. We haven't added any behavior to our app, nor have we created the namespace where our application is going to live. The first step in getting this test to pass will be to do that.