



Chapter 18. In the family

Along side React, Facebook has create a number of front end tools. They don't have to be used with React and you don't have to use them in your React project, however they work great together with React.

In this chapter we will cover the following tools:

- Jest
- Immutable-js
- Flux

In this chapter we will also cover a tool which was not created by Facebook, but which is helpful for any web application:

- Automated Browser Testing

Jest

Jest is a test runner built by Facebook. It provides a **Familiar Approach** being built on top of the Jasmine test framework, using familiar `expect(value).toBe(other)` assertions. It **Mocks by Default** by automatically mocking CommonJS modules returned by `require()`, making most existing code testable, and has a **Short Feedback Loop** with mocked DOM APIs and tests that run in parallel via a small node.js command line utility.

This section of the family chapter assumes familiarity with the Jasmine testing library. We will cover the following Jest topics:

- Setup
- Automatic dependency mocking

Setup

To setup your project with Jest, start by creating a `__tests__` folder (the name of this can be configured), create a test file in `__tests__`

```
// __tests__/sum-test.js
jest.dontMock('../sum');

describe('sum', function() {
  it('adds 1 + 2 to equal 3', function() {
    var sum = require('../sum');
    expect(sum(1, 2)).toBe(3);
  });
});
```

install Jest with `npm install jest-cli --save-dev` run `jest` on the command line and you should see the test result:

```
[PASS] __tests__/sum-test.js (0.015s)
```

Automatic mocking

By default Jest will automatically mock any dependencies your source file requires. It does that by overwriting the `require` function in node.

Consider our `TakeSurveyItem` component:

```
var React = require('react');
var AnswerFactory = require('../answers/answer_factory');

var TakeSurveyItem = React.createClass({
  render: function () {
    // ...
  },
  getSurveyItemClass: function () {
    return AnswerFactory.getAnswerClass(this.props.item.type);
  }
});

module.exports = TakeSurveyItem;
```

This uses the `AnswerFactory` dependency in the `getSurveyItemClass` function. Tests for that module is covered in its own test file, so we do not need to duplicate that effort. Instead we simply want to make sure the right method is getting called on `AnswerFactory`.

Jest automatically mocks all dependencies. We want `AnswerFactory` to be mocked so we can test that its `getAnswerClass` got called, but we don't want to mock `TakeSurveyItem`, since we are testing it and `React`.

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

```

jest.dontMock('react');
jest.dontMock('app/components/take_survey_item');

var TakeSurveyItem = require('app/components/take_survey_item');
var AnswerFactory = require('app/components/answers/answer_factory');
var React = require('react/addons');
var TestUtils = React.addons.TestUtils;

describe('app/components/take_survey_item', function () {
  var subject;

  beforeEach(function () {
    subject = TestUtils.renderIntoDocument(
      TakeSurveyItem()
    );
  });

  describe('#getSurveyItemClass', function () {
    it('calls AnswerFactory.getAnswerClass', function () {
      subject.getSurveyItemClass();
      expect( AnswerFactory.getAnswerClass ).toBeCalled();
    });
  });
});

```

Notice that we are telling Jest not to mock our TakeSurveyItem component and React. We want both of those modules to run with their implementation code.

We then are requiring all the modules we need for the test, including AnswerFactory. We didn't tell Jest that AnswerFactory should **not** be mocked, so when we require it (and when TakeSurveyItem requires it), a mock is returned.

In our test we check to see if the getAnswerClass method is called when we call getSurveyItemClass on TakeSurveyItem. Notice that we are using toBeCalled, not to be confused with Jasmine Spies' toHaveBeenCalled. Jest doesn't interfere with Spies and you can use both Jest and spies if you like.

Manual mocking

Sometimes Jest's automatic mocking falls short, in those cases Jest gives you a way to create your own mock of a certain library.

Let's build a manual mock of the AnswerFactory module from the previous section.

```

jest.dontMock('react');
jest.dontMock('app/components/take_survey_item');

// the manual mock needs to happen before we require
// TakeSurveyItem, otherwise TakeSurveyItem will receive a
// different mock of AnswerFactory
jest.setMock('app/components/answers/answer_factory', {

```

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

```

var TakeSurveyItem = require('app/components/take_survey_item');
var AnswerFactory = require('app/components/answers/answer_factory');
var React = require('react/addons');
var TestUtils = React.addons.TestUtils;

describe('app/components/take_survey_item', function () {
  var subject;

  beforeEach(function () {
    // ...
  });

  describe('#getSurveyItemClass', function () {
    it('calls AnswerFactory.getAnswerClass', function () {
      // ...
    });
  });
});

```

If we want to create a manual mock of a common library that Jest failed to auto mock or we want to use a mock of AnswerFactory frequently we can create a `__mocks__` folder in the directory of `answer_factory.js` with a mock file inside (also called `answer_factory.js`), like so:

```

// app/components/answers/__mocks__/answer_factory.js
var React = require('react/addons');
var TestUtils = React.addons.TestUtils;

module.exports = {
  getAnswerClass: jest.genMockFn().mockReturnValue(TestUtils.mockComponent)
};

```

This allows us to avoid the `jest.setMock` call in our tests:

```

jest.dontMock('react');
jest.dontMock('app/components/take_survey_item');

var TakeSurveyItem = require('app/components/take_survey_item');
var AnswerFactory = require('app/components/answers/answer_factory');
var React = require('react/addons');
var TestUtils = React.addons.TestUtils;

describe('app/components/take_survey_item', function () {
  var subject;

  beforeEach(function () {
    // ...
  });

  describe('#getSurveyItemClass', function () {
    it('calls AnswerFactory.getAnswerClass', function () {
      // ...
    });
  });
});

```

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

Read more about Jest on <http://facebook.github.io/jest/> (<http://facebook.github.io/jest/>) and about testing in chapter 6.

Immutability-js

Immutable Data Structures are data structures that can't change. Instead when you request a change they return a copy of the original object with the changes applied. They work really well when married with React and Flux for simplicity and performance gains in your application

Immutable-js provides a series of data structures that can be built from native JavaScript data structures and can convert back into native JavaScript data structures when needed.

Immutable.Map

Immutable.Map can be used as a substitute for regular JS objects:

```
<span class="kd" style="box-sizing: border-box; font-weight: bold;">var</span> question
// get values from the Map with .get</span>
question.get('description');

// updating values with .set returns a new object.
// The original object remains intact.
question2 = question.set('description', 'Who is your favorite comicbook hero?');

<span class="p " style="box-sizing: border-box;">// merge 2 objects with .merge to get a
// Once again none of the original objects are mutated.</span>
<span class="kd" style="box-sizing: border-box; font-weight: bold;">var</span> <span cla
<span class="kd" style="box-sizing: border-box; font-weight: bold;">var</span> <span cla
question3.toObject(); // { title: 'Question #1', description: 'who is your favorite comi
```

Immutable.Vector

Use Immutable.Vector for Arrays:

```
<span class="kd" style="box-sizing: border-box; font-weight: bold;">var</span> options <
<span class="kd" style="box-sizing: border-box; font-weight: bold;">var</span> <span cla
options.toArray(); // ['Superman', 'Batman', 'Spiderman']</span>
```

You can nest the datastructures:

```
<span class="kd " style="box-sizing: border-box; font-weight: bold;">var</span> options
</span>var question = Immutable.Map({
  description: 'who is your favorite superhero?',
  options: options
});
```

Immutable.js has many more features. For more information on Immutable.js go

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

Flux

As mentioned in the Architecture Patterns chapter, Flux is a pattern released by Facebook alongside React. Its most notable feature is strict enforcement of one-way data flow.

Facebook released a reference Flux implementation on github, accessible at <https://github.com/facebook/flux>.

It contains three main components:

- Dispatcher
- Stores
- Views

It's easiest to visualize how these pieces fit together.

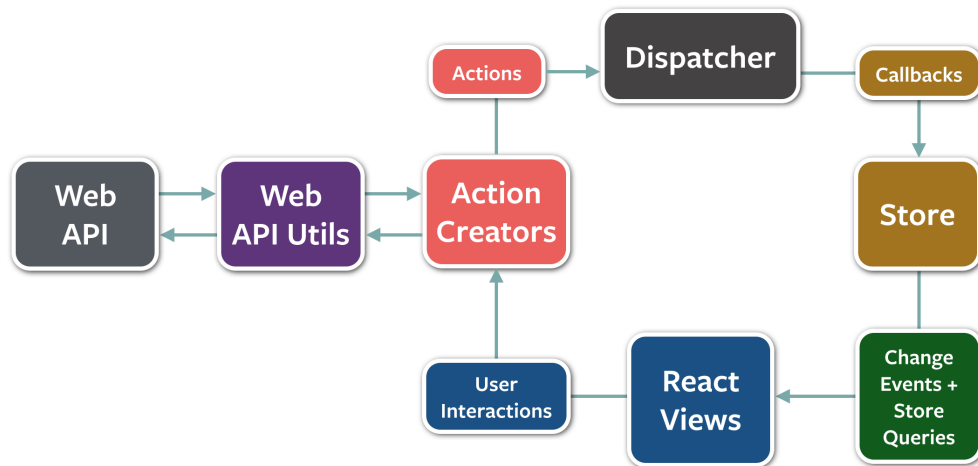


Figure 18-1. Flux enforces a uni-directional data flow

Since flux is a pattern with no hard dependencies you are free to adopt any portion of it you might find helpful.

For a detailed discussion on Flux see chapter 14.

Automated Browser Testing

As we defined in Chapter 6, Functional testing is a type of test which verifies the application functions correct from the perspective of the end user. For a web application this will be clicking around and filling out forms in a web browser, just like a user.

A BRIEF INTRODUCTION

This section will serve as a brief introduction to the basics of writing a functional test for a web application. There is simply too much to cover for this to be an exhaustive resource. If you need more information on this topic, there are wonderful books devoted to just this topic ([The Cucumber Book and Instant Testing with CasperJS](http://www.amazon.com/Instant-Testing-CasperJS-%C3%89ric-Br%C3%A9hault/dp/1783289430) (<http://www.amazon.com/Instant-Testing-CasperJS-%C3%89ric-Br%C3%A9hault/dp/1783289430>)).

For these tests, we will be directing a web browser to perform certain actions and asserting the web page is in the correct state. To do this we will be using a wonderful tool call CasperJS. If CasperJS and other automated browser testing tools are new to you, here is a short dictionary of important terms:

1. CasperJS - a testing utility which makes “driving a web browser” very easy. CasperJS uses PhantomJS for the browser implementation.
2. PhantomJS - a headless web browser with a javascript api which uses the webkit rendering engine.
3. Headless web browser - Imagine a web browser which you use daily (Chrome, Firefox, IE), but it doesn't have a visual interface which is visible on the screen. It can be interacted with via commands and is run in a terminal.
4. Driving a web browser - clicking on links, filling out forms, navigating to urls, dragging and dropping elements. Anything an end user would do in a browser.
5. Webkit - the rendering engine which powers Chrome and Safari. (Firefox is powered by the Gecko engine, and Internet Explorer is powered by the Trident engine).

Before we get started writing a full featured example, let's just take a peak at a CasperJS test:

Before we get started writing a full featured example, let's just take a peak at a CasperJS test:

```
casper.test.begin('Adding a survey', 1, function suite(test) {
  casper.start("http://localhost:8080/", function(){
    test.assertTitle("SurveyBuilder", "the title for the homepage is correct");
  });

  casper.run(function() {
    test.done();
  });
});
```

At a high level this test is visiting the homepage for our application and then asserting what the title of the page is. The first thing you'll notice is that there is no mention of React.js in this test -- that is by design. Casper will drive the browser as a user would, so the fact React.js is used is simply an implementation detail. You might have noticed the test has a title, an integer argument, and a

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

test. OK, let's get started with our first test, `ROOT/test/functional/adding_a_survey.js` to test adding a survey in our newly created application:

```
// Setup some casper.js config options
casper.options.verbose = true;
casper.options.logLevel = "debug";
casper.options.viewportSize = {width: 800, height: 600};

casper.test.begin('Adding a survey', 0, function suite(test) {

});
```

Now we have our boilerplate out of the way, let's think about the test we'd like to automate:

1. Go to the homepage
2. Validate the homepage has loaded correctly
3. Click on the "Add Survey" link
4. Assert we are taking to the correct page


```
casper.test.begin('Adding a survey', 0, function suite(test) {
  casper.start("http://localhost:8080/", function(){
    console.log("we went to the homepage!")
  });
});
```

To run this we need to run CasperJS, so let's install the `casperjs` module and then run our tests:

```
npm install -g casperjs
casperjs test test/functional
```

Which will give the following output:

```
tom:bleeding-edge-sample-app (jasmine-node) $ casperjs test test/functional/
Test file: /Users/tom/workspace/bleeding-edge-sample-app/test/functional/adding_a_survey
# Adding a survey
[info] [phantom] Starting...
[info] [phantom] Running suite: 2 steps
[debug] [phantom] opening url: http://localhost:8080/, HTTP GET
[debug] [phantom] Navigation requested: url=http://localhost:8080/, type=Other, willNavi
[warning] [phantom] Loading resource failed with status=fail: http://localhost:8080/
[debug] [phantom] Successfully injected Casper client-side utilities
we went to the homepage!
[info] [phantom] Step anonymous 2/2: done in 53ms.
[info] [phantom] Done 2 steps in 72ms
WARN Looks like you didn't run any test.
```



Please look at that output and look for the following parts:

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

2. That request is failing
3. You see the console.log print statement after the homepage finished not loading
4. It's telling us we didn't run any tests.

#1 and #3 are good and #4 is fine because we didn't write any assertions, but #2 is an issue. We need to make sure our application is running before we can kick off our tests. So let's start our application:

```
npm start
```

and then rerun our casperjs test command and we'll get the following output:

```
tom:bleeding-edge-sample-app (jasmine-node) $ casperjs test test/functional/
Test file: /Users/tom/workspace/bleeding-edge-sample-app/test/functional/adding_a_survey
# Adding a survey
[info] [phantom] Starting...
[info] [phantom] Running suite: 2 steps
[debug] [phantom] opening url: http://localhost:8080/, HTTP GET
[debug] [phantom] Navigation requested: url=http://localhost:8080/, type=Other, willNavi
[debug] [phantom] url changed to "http://localhost:8080/"
[debug] [phantom] Successfully injected Casper client-side utilities
[info] [phantom] Step anonymous 2/2 http://localhost:8080/ (HTTP 200)
we went to the homepage!
[info] [phantom] Step anonymous 2/2: done in 1773ms.
[info] [phantom] Done 2 steps in 1792ms
WARN Looks like you didn't run any test.
```

Much better! Now, let's add an assertion - that the html <title> for the homepage is the correct value which is "SurveyBuilder".

```
...
casper.start("http://localhost:8080/", function(){
  // assert the title of the homepage is "SurveyBuilder"
  test.assertTitle("SurveyBuilder", "the title for the homepage is correct");
});
...
```

This new line is asserting the title is correct. Notice how we are passing a second argument which describes the test, this is used in the casperjs terminal output (on the line which says PASS):

```
tom:bleeding-edge-sample-app (jasmine-node) $ casperjs test test/functional/
Test file: /Users/tom/workspace/bleeding-edge-sample-app/test/functional/adding_a_survey
# Adding a survey
[info] [phantom] Starting...
[info] [phantom] Running suite: 2 steps
[debug] [phantom] opening url: http://localhost:8080/, HTTP GET
[debug] [phantom] Navigation requested: url=http://localhost:8080/, type=Other, willNavi
[debug] [phantom] url changed to "http://localhost:8080/"
[debug] [phantom] Successfully injected Casper client-side utilities
```

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

```
[info] [phantom] Step anonymous 2/2: done in 1864ms.  
[info] [phantom] Done 2 steps in 1883ms  
PASS 1 test executed in 1.89s, 1 passed, 0 failed, 0 dubious, 0 skipped.
```

CASPERJS DOCUMENTATION

If you want more information about the types of things CasperJS can do, read their documentation here: casperjs.org (<http://casperjs.org>).

Now we have our first passing tests, lets try to click on a link.

```
...  
casper.start("http://localhost:8080/", function(){  
  // assert the title of the homepage is "SurveyBuilder"  
  test.assertTitle("SurveyBuilder", "the title for the homepage is correct");  
  
  // click on the "Add Survey" link (which is the second in the nav bar)  
  this.click(".navbar-nav li:nth-of-type(2) a");  
});  
...
```

The click function takes a css selector. In this example, we are clicking on the second link in the navbar. If the link has a unique class or an id, then it's preferable to use that as the selector argument. Now that we have clicked the link for adding a survey, let's verify the user see's the "Add Survey" page.

```
...  
casper.start("http://localhost:8080/", function(){  
  // assert the title of the homepage is "SurveyBuilder"  
  test.assertTitle("SurveyBuilder", "the title for the homepage is correct");  
  
  // click on the "Add Survey" link (which is the second in the nav bar)  
  this.click(".navbar-nav li:nth-of-type(2) a");  
});  
  
casper.then(function(){  
  // assert the /add_survey page looks as we suspect  
  test.assertTitle("Add Survey to SurveyBuilder", "the title for the add survey page is  
  test.assertTextExists("Drag and drop a module from the left",  
    "instructions for drag and drop questions exist on the add survey screen");  
});  
...
```

When looking at this code, you might have noticed something interesting - for asserting the title and clicking the link, our code is inside of the `start` callback. But then for the new assertions, we

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

tests, and CasperJS is here to help. When you call `click`, CasperJS has a mechanism to wait until the new page has “finished loading”. To utilize this, you need to put the next commands in a `then` callback which will be executed when the “click the add survey link” action is complete.

If we run our tests again with `casperjs test test/functional/` we will see they pass!

STARTING A SERVER

So far, we had to start the npm server before we could run the CasperJS tests. This is annoying and will be a pain for our automated testing server. So let’s write a bash script to start a new server when we run our tests. Let’s write a script in the root of the project called `run_casperjs.js`:

```
// start the node webserver on port 3040
var app = require("./server/server"),
    appServer = app.listen(3040),

    // run casperjs test suite in a child process
    spawn = require('child_process').spawn,
    casperJs = spawn('./node_modules/casperjs/bin/casperjs', ['test', 'test/functional']);

// pipe all data from casperjs to the main output
casperJs.stdout.on('data', function (data) {
  console.log(String(data));
});
casperJs.stderr.on('data', function (data) {
  console.log(String(data));
});

// when casperjs finishes, we should shutdown the node web server
casperJs.on('exit', function(){
  appServer.close();
});
```

This file is going to do a few things:

1. load your nodejs server
2. start the server on port 3040
3. spawn a subprocess to execute the CasperJS tests. That complex line is the equivalent of running `casperjs test test/functional` on the commandline
4. pipe all output from CasperJS to the terminal
5. when CasperJS tests are finished, shutdown the server.

Now just go update your spec to use port 3040 instead of 8080 and you should be good to go -- running `./run_casperjs.js` will not only run your automated CasperJS tests, but it will also handle starting and stopping the node server for you!

Hopefully you will now understand the highlevel of what a CasperJS test looks like and how to write a simple test. This section is no where near an exhaustive guide to CasperJS. so we’d

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

- “Site Testing with CasperJS” by Joseph Scott: <https://www.youtube.com/watch?v=flhjYUNCo-U>
- CasperJS Testing Framework: <http://docs.casperjs.org/en/latest/testing.html>
- CasperJS casper documentation: <http://docs.casperjs.org/en/latest/modules/casper.html>



PREV

17. Architectural Patterns

NEXT



19. Uses