



## Chapter 15. Testing

So now that your SurveyApp is starting to take shape and you're learning the pleasure of React.js components, let's take a step back for a bit before diving into the next shiny new feature. When you are starting a new application, productivity is very simple because you just crank out more code. But as your application evolves, if you aren't careful you can find yourself with a codebase that is a twisted mess, which is very hard to change. That is why you have a very powerful tool in your toolbox - automated testing. Automated testing, usually employed through a Test Driven Development (TDD) workflow, can help you achieve code which is simpler, more modular, is less brittle to change, and can be changed with more confidence.

---

### BUT I'VE NEVER TESTED MY JAVASCRIPT BEFORE!?!

That's ok! Automated testing can seem like a foreign concept that is always slightly out of reach when you haven't done it before. I know it did for me! This chapter won't serve as an exhaustive resource for JavaScript testing because that deserves it's own book, but we'll try to provide enough context so you can follow along and research specific topics on your own when needed.

---

## Getting Started

"But I have an amazing QA team who focuses on testing. Why should I care about testing? Can't I just skip this chapter?" you might ask. Great questions! The main reason automated testing will help you is not related to bugs or catching regressions, but that is a helpful side effect of testing. The real goal of automated testing is that it helps you write *better* code. More often than not, code which is poorly written is hard to test. So if you start writing your tests as you write your code, your tests will encourage you to not write sloppy code. You will be naturally pushed to follow the single responsibility principle <sup>2</sup>, the law of demeter <sup>3</sup>, and keeping code modular.

Now that you've been overwhelmingly convinced that automated testing is important, let's go over the 3 types of testing which you'll learn for our SurveyApp: unit testing, integration testing, and functional testing. There are many other types of automated testing (performance testing, security testing, visual testing (dpxdt<sup>4</sup>), but those are outside the scope of this book.

- **Unit Test:** a test which exercises the smallest piece of functionality in your application. Often this will be calling a function directly with specific inputs and validating the outputs or the side effects.
- **Integration Test:** a test which combines two or more different pieces of functionality and verifies they work correctly together.
- **Functional Test:** a 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.

This sounds like a lot, but when you dive into it you will find out that it is not only manageable but very fun to see your tests passing for the first time.

## Tools

Fortunately the JavaScript community has a healthy ecosystem of testing tools, so we can leverage those to get our test suites off the ground quickly. Here is the software stack you will be using for this book, and listed next to it are some popular alternatives.

- **Unit and Integration Testing [Client-side]:** jasmine and karma
  - **Alternatives:** Mocha, Chai, sinon, vows.js, qunit
- **Unit and Integration Testing [Server-side]:** mocha and supertest
  - **Alternatives:** Same as client-side, plus jasmine-node
- **Functional Testing:** Casperjs
  - **Alternatives:** nightwatchjs, zombie.js, selenium based (capybara, waitr, etc)

Enough chit chat, let's start coding!

## Our first spec: Rendering

When writing a React.js component, the only requirement is that you define a render function. So that is as good a place as any to start our testing efforts. Let's imagine we want to write a `<HelloWorld>` component that simply outputs an `<h1>` tag that says "Hello World!". Because you will be writing this code driven by tests (TDD), we will skip creating the `<HelloWorld>` component until we need to. First, let's start writing our test by creating a JavaScript file that will contain our spec (i.e. test):

---

```
test/client/fundamentals/render_into_document_spec.js
```

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

---

```
/** @jsx React.DOM */

var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

describe("HelloWorld", function(){
});
```

---

This is the boilerplate we will have in all of our React.js jasmine unit tests. So let's review it to make sure we understand each piece:

---

```
/** @jsx React.DOM */
```

---

Our spec files need the JSX “docblock” to tell the JSX parser our spec uses JSX too.

---

```
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;
```

---

Instead of the typical `require("react")` like in our application code, we want to require the “react/addons” package, because it includes some test utility functions in addition to the regular React functions. `React.addons.TestUtils` is a helpful module we will use, which will be covered in this chapter and in Chapter 18.

---

```
describe("HelloWorld", function(){
});
```

---

This is our jasmine `describe` block, which denotes this is a test suite for the `HelloWorld` module. Inside of this `describe` block is where we will write our tests.

Now that we have the spec boilerplate setup, let's add a test that renders the `HelloWorld` component. At first glance you'd think, this spec should be pretty simple: just call `React.renderComponent(<HelloWorld>, someDomElement)` and then do some simple assertions. As it turns out, React.js will not re-mount a component if that “same” component is already rendered in that element, for performance reasons, which is great for your application code but can be a bad idea for your tests.

---

#### REACT.RENDERCOMPONENT IN YOUR TESTS

If you use `React.renderComponent` in a test, it is very likely one test will pollute the following tests. Test pollution can cause **bizarre** failing tests or tests which are passing but shouldn't be passing.

---

To render a component in our test suite, we want to use a function called

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

component. You might notice that is different from `React.renderComponent`, which takes a second argument for the element to insert the component into. What this means is that `renderIntoDocument` will insert your component into a detached DOM node which only exists in memory.

Our test will start out with a very simple goal: rendering the component.

---

```
...
describe("HelloWorld", function(){
  describe("renderIntoDocument", function(){

    it("should render the component", function(){
      TestUtils.renderIntoDocument(<helloworld></helloworld>);
    });
  });
});
```

---

Now that we've written the Jasmine spec, we need a way to run it. There are many different open source projects to help here, but for this project we have chosen *Karma*, which is a JavaScript test runner developed by Google. Karma, formally known as testacular, can run your tests in multiple browsers and aggregate the results for you in a very easy to use way. To run the Karma tests, execute the following command:

---

```
npm run-script test-client
```

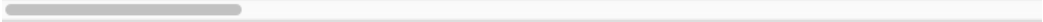
---

When you run this command, you should see a healthy amount of debugging output and that end you should see:

---

```
Chrome 36.0.1985 (Mac OS X 10.8.2) HelloWorld renderIntoDocument should render the compo
```

---



This means that our test is failing because the `HelloWorld` component is not defined, which makes sense because we haven't written it yet. So let's write a very basic React component and use it in our spec.

---

```
/** @jsx React.DOM */
var React = require("react");
var HelloWorld = React.createClass({
  render: function(){
    return (
      <div></div>
    );
  }
});

module.exports = HelloWorld;
```

---

```
var HelloWorld = require('../../client/testing_examples/hello_world');

describe("HelloWorld", function(){
  describe("renderIntoDocument", function(){

    ...
  })
})
```

---

#### KARMA: SINGLE RUN

After you make this code change and switch back to your terminal window, you should look at the terminal window carefully and notice that the tests should already re-running. This is because the configuration for Karma, karma.conf.js, has been setup to rerun when any of the project files change.

---

Now that we have defined our HelloWorld component and required it in our spec, our test should be passing:

---

```
Chrome 36.0.1985 (Mac OS X 10.8.2): Executed 1 of 1 SUCCESS (0.905 secs / 0.801 secs)
```

---

Let's now add two more specs, to feel out how `renderIntoDocument` behaves:

1. Validate the html which is rendered contains "Hello World!"
  2. Validate we can start asserting things about the component we just rendered
- 

```
...
it("should render the component and it's html into a dom node", function(){
  var myComponent = TestUtils.renderIntoDocument(<HelloWorld />);

  // you can validate on the html which was rendered
  expect(myComponent.getDOMNode().textContent).toContain("Hello World!");
});

it("should render the component and return the component as the return value", function(){
  var myComponent = TestUtils.renderIntoDocument(<HelloWorld />);

  // you can assert things on the component which was rendered
  expect(myComponent.props.name).toBe("Bleeding Edge React.js Book");
});
...

```

---

Switch back to your terminal and you should see two failing tests (which we expected):

---

```
Chrome 36.0.1985 (Mac OS X 10.8.2) HelloWorld renderIntoDocument should render the compo
```

---

---

```
...
    it("should render the component and it's html into a dom node", function(){
        var myComponent = TestUtils.renderIntoDocument(<HelloWorld />);

        // you can validate on the html which was rendered
        expect(myComponent.getDOMNode().textContent).toContain("Hello World!");
    });

...
```

---

You should watch this spec fail because we don't have a DOM element with the class of "subheading", and can then make it pass with this change:

---

```
...
    render: function(){
        return (
            <div>
                <h1>Hello World!</h1>
                <h2 className="subheading">{this.props.name}</h2>
            </div>
        );
    }
    ...

```

---

---

#### REACT AND HTML ASSERTIONS

After reading some of these tests, you might be wondering why we don't write a test like this:

---

```
it("should never pass if you try to assert on a whole dom node", function(){
    var myComponent = TestUtils.renderIntoDocument(<HelloWorld />);

    // DON'T DO THIS!! IT WON'T WORK
    expect(myComponent.getDOMNode().innerHTML).toContain("<h2 class='subheading'>Bleed
});
```

---

When testing an application that is a jQuery or Backbone.js application you could do this and it would probably work, but I wouldn't recommend doing it there either. The problem is that with React it will never work, because the "html" you specify in the render function isn't the HTML that actually gets rendered into the DOM. The DOM that gets rendered to the page looks more like this:

---

```
<h1 data-reactid=".1k.0">Hello World!</h1>
<h2 class="subheading" data-reactid=".1k.1">Bleeding Edge React.js Book</h2>
```

---

Those data attributes are used by React to re-render your component in an extremely performant manner.

---

## Mock Components

One powerful feature in React that you just learned about in Chapter 4 is composing components of other components. Having one component render another one is great for modularity and code reuse, but it takes a special consideration with regard to your tests. Let's imagine you have two components: `UserBadge` and `UIImage`, where `UserBadge` renders the user's name and their `UIImage`. When writing tests for the `UserBadge` component, it is important only `UserBadge` functionality is being tested and that you are not implicitly testing `UIImage` functionality. While a coworker might argue that testing both "is better because it's more like real life", you will quickly find that your tests will get harder and harder to write and maintain because they will lose focus of the "unit" you are trying to test.

---

### LISTEN TO YOUR TESTS!

A "code smell" that a test is started to test too many different things and is becoming unfocused is when the test needs a non-trivial amount of setup before you have the criteria needed to run your test. If your test is painful to setup, take a second look at your architecture -- see if your tests are trying to give you some advice.

---

Here's the goal for our `UserBadge` test, before rendering the `UserBadge` component, we should modify it so that `UIImage` component has been replaced by a mock component which has no real behavior. Todo this is very dependent on the tools and architecture you have chosen for your application. For the `SurveyApp`, we have chosen `browserify`, so I will show that approach first. Let's start with the basic boilerplate for a `UserBadge` test and a simple render call:

---

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

var UserBadge = require('../client/testing_examples/user_badge');

describe("UserBadge", function(){
  // NOTE: This will render the actual UIImage component, which is not the desired
  it("should use the mock component and not the real component", function(){
    var userBadge = TestUtils.renderIntoDocument(<UserBadge />);
  });
});
```

---

To help us stub out the `UIImage` component for this test, we will use an open source module called "rewireify". This module is like magic, since it allows us to rewrite local variables and local functions in a module with something else. If you look at the source for the `UserBadge` module, you will see the following lines:

---

```
var UIImage = require("../user_image");
```

---

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

```

    return (
      <div>
        <h1>{this.props.friendlyName}</h1>
        <UserImage slug={this.props.userSlug} />
      </div>
    );
  }
  ...

```

---

So this means that the UserBadge module has a local variable called “UserImage” which is set to a react component. In the UserBadge spec, we will tell rewireify to “set” the “UserImage” local variable with a mock component. The basic approach looks like this:

---

```

var mockUserImageComponent = React.createClass({
  render: function(){
    return (<div className="fake">Fake User Image!!</div>);
  }
});

UserBadge.__set__("UserImage", mockUserImageComponent);

```

---

Now when the UserBadge components tries to render it will render our mockUserImageComponent and not the real UserImage component. While the above example is nice, it doesn’t account for one import variable - test pollution. If we \_\_set\_\_ a variable in one test, we need to make sure our change is reversed before the next test. So our test should look like this:

---

```

describe("UserBadge", function(){
  describe("rewireify", function(){
    var mockUserImageComponent;

    beforeEach(function(){
      // When we want to unit test the UserBadge component, we don't want to implicitly
      //   with it -- otherwise that turns into an integration test. So we replace the
      //   mock component (ie one with the minimum functionality needed)
      mockUserImageComponent = React.createClass({
        render: function(){
          return (<div className="fake">Fake User Image!!</div>);
        }
      });
    });

    describe("using just rewireify", function(){
      var realUserImageComponent;

      beforeEach(function(){
        // we need to save off the real definition, so we can put it back when the test
        //   NOTE: if we don't do this, all subsequent tests will be polluted!!
        realUserImageComponent = UserBadge.__get__("UserImage");
        UserBadge.__set__("UserImage", mockUserImageComponent);
      });

      afterEach(function(){

```

---

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



```

    it("should use the mock component and not the real component", function(){
      var userBadge = TestUtils.renderIntoDocument(<UserBadge />);

      expect(TestUtils.findRenderedDOMComponentWithClass(userBadge, "fake").getDOMNode()
    });
  });
});
});

```

---

#### TESTUTILS.FINDRENDEREDDOMCOMPONENTWITHCLASS

We are using a utility called `TestUtils.findRenderedDOMComponentWithClass` in this test. We will cover the behavior of this utility later in this chapter, but for right now all you need to know is that it finds a component with the class “fake”.

---

Here is the general algorithm of the test code shown above:

1. Define the `mockUserImageComponent` React component
2. Get the value for the `UserImage` variable in the `UserBadge` module and save it in a local called `realUserImageComponent`
3. Set the value for the `UserImage` variable in the `UserBadge` module to the `mockUserImageComponent`
4. Perform the test
5. Set the value for the `UserImage` variable in the `UserBadge` module back to the `realUserImageComponent`

The good news is that this approach works great, but the bad news is that it is a lot of boiler plate code that we will probably need in almost every spec because most specs will render other components. So, this is where a custom Jasmine helper can make a large difference. What you’re going to do is write a module that has two main parts: a function that we can call from our spec to rewire a variable and an `afterEach` hook, which reverses all those changes at the end. To reverse the changes, we just keep track of all rewire’s that we’ve done in an array and then loop through that array in the clean up phase. Here is a that helper function that we’d typically store in `test/client/helpers/rewire-jasmine.js`:

---

```

var rewires = [];
var rewireJasmine = {
  rewire: function(mod, variableName, newVariableValue){
    // save off the real value, so we can revert back to it later
    var originalVariableValue = mod.__get__(variableName);

    // keep track of everything which was rewire'd through this helper
    rewires.push({

```

---

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

```

        originalVariableValue: originalVariableValue,
        newVariableValue: newVariableValue
    });

    // rewire the variable to the new value
    mod.__set__(variableName, newVariableValue);
},

unwireAll: function(){
    for (var i = 0; i < rewires.length; i++) {
        var mod = rewires[i].mod,
            variableName = rewires[i].variableName,
            originalVariableValue = rewires[i].originalVariableValue;

        // rewire the variable name back to the original value
        mod.__set__(variableName, originalVariableValue);
    }
}
};

afterEach(function(){
    // unwind all modules we rewired
    rewireJasmine.unwireAll();

    // reset the array back to an empty state in preparation for the next spec
    rewires = [];
});

module.exports = rewireJasmine;

```

---

With this helper module in our toolbox, our UserBadge/UserImage example can turn into this:

---

```

var rewireJasmine = require("../helpers/rewire-jasmine");
var UserBadge = require('../../../../client/testing_examples/user_badge');

describe("UserBadge", function(){
    describe("with a custom rewireify helper", function(){
        beforeEach(function(){
            rewireJasmine.rewire(UserBadge, "UserImage", mockUserImageComponent);
        });

        it("should use the mock component and not the real component", function(){
            var userBadge = TestUtils.renderIntoDocument(<UserBadge />);

            expect(TestUtils.findRenderedDOMComponentWithClass(userBadge, "fake").getDOMNode()
            );
        });
    });
});

```

---

Look how easy that is! `rewireJasmine.rewire(UserBadge, "UserImage", mockUserImageComponent)`; handles the saving off of the original value and the module registers the `afterEach` cleanup hook.

---

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

---

## OTHER NPM IMPLEMENTATIONS

If you are using webpack instead of browserify, you will want to swap out “rewireify” with “rewire-webpack”. If you are testing node instead of client code, you will want to use the original project “rewire” (which is the project which spawned “rewireify” and “rewire-webpack”. The interface is slightly different, but you can read more about it here: <https://github.com/jhnns/rewire>

---

But what if my project isn’t build on the npm/require goodies and I’m using <script> tags that store my components in a global variable? Don’t worry, you aren’t out of luck either. The pattern is exactly the same, but the code looks a bit different:

---

```
describe("global variables", function(){
  var mockUserImageComponent, realUserImageComponent;

  beforeEach(function(){
    // When we want to unit test the UserBadge component, we don't want to implicitly
    // with it -- otherwise that turns into an integration test. So we replace the
    // mock component (ie one with the minimum functionality needed)
    mockUserImageComponent = React.createClass({
      render: function(){
        return (<div className="fake">Fake Vanilla User Image!!</div>);
      }
    });

    // we need to save off the real definition, so we can put it back when the test is
    // NOTE: if we don't do this, all subsequent tests will be polluted!!
    realUserImageComponent = window.vanillaScriptApp.UserImage;
    window.vanillaScriptApp.UserImage = mockUserImageComponent;
  });

  afterEach(function(){
    window.vanillaScriptApp.UserImage = realUserImageComponent;
  });

  it("should use the mock component and not the real component", function(){
    var UserBadge = window.vanillaScriptApp.UserBadge;
    var userBadge = TestUtils.renderIntoDocument(<UserBadge />);

    expect(TestUtils.findRenderedDOMComponentWithClass(userBadge, "fake").getDOMNode()
  });
});
```

---

Let’s review what we’ve learned so far:

1. How to render a component into the document
2. How to stub out a nested component with a mock implementation

## Spying on a function

---

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

The next topic we need to cover is how do we spy on a function within the module we are testing? Spying on a function within a module has many purposes:

1. To prevent the real implementation of that method from running so we can unit test that function independently
2. To prevent the real implementation of that method from running because it depends on an api, third party service or something else we don't want to run in our test suite
3. Verify that the stub was called by a certain function or with certain arguments

To spyOn a function like "foo" in the example below with jasmine, you typically do something like this:

---

```
var myModule = {
  foo: function(){
    return 'bar';
  }
};

spyOn(myModule, "foo").andReturn('fake foo');
```

---

A logical first step for trying this with a React component like the following might be to try this:

---

```
var myComponent = React.createClass({
  foo: function(){
    return 'bar';
  },
  render: ...
});

spyOn(myComponent.prototype, "foo").andReturn('fake foo');
```

---

But that won't work for several reasons:

1. React doesn't store your functions on the prototype (like Backbone does)
2. React stores another location of your function for auto binding
3. The solutions to the above are specific to your React.js version and get messy quickly

To solve this issue, there is a module called `jasmineReactHelpers` we can use. Let's say that the component we wanted to test is called `HelloRandom`, which outputs information about a random author from this book, but we don't want the actual selection to be random. Here is the definition of our `HelloRandom` component:

---

```
/** @jsx React.DOM */
var React = require("react");
var authors = [
  { name: "Bill", githubUsername: "billy" },
  { name: "Fred", githubUsername: "freddy" }
];
```

---

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

```
getRandomAuthor: function(){
  return authors[Math.floor(Math.random() * authors.length)];
},
render: function(){
  var randomAuthor = this.getRandomAuthor();

  return (
    <div>
      {randomAuthor.name} is an author and their github handle is {randomAuthor.github}
    </div>
  );
}
});

module.exports = HelloRandom;
```

---

So let's write a test for the render function which tries to validate the html text is correct. Here is what our spec would typically look like for that:

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

var HelloRandom = require('../../../../client/testing_examples/hello_random');

describe("HelloRandom", function(){
  describe("render", function(){
    it("should output information about the author", function(){
      var myHelloRandom = TestUtils.renderIntoDocument(<HelloRandom />);

      expect(myHelloRandom().textContent).toBe("Bill is an author and their github handle is");
    });
  });
});
```

---

The problem is that this spec will fail intermittently depending on who the random author is.

#### **"RANDOM" IS JUST AN ILLUSTRATIVE EXAMPLE**

We are just using randomAuthor as an example of a function which we want to spyOn. In a real application, this might be a function which performs things like:

1. fetching data from the server
  2. utilizing a component's state which is a pain to setup in a test
  3. has far reaching side effects which are a pain to stub out
  4. data which is based on the current time or timezone
- 

---

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

will make the intent of the class much more clear to other developers.

---

```
...
var jasmineReact = require("jasmine-react-helpers");
var HelloRandom = require('../client/testing_examples/hello_random');
...
it("should be able to spy on a function of a react class", function(){
  // We want to render the HelloRandom component and validate the output is correct.
  // The important part to test is "is the render function outputting the correct
  // for a given author".
  // In this example the author is random which makes it very difficult to test.
  // if your app doesn't have any "random" functions, the same strategy would be h
  // a function like getCurrentAuthor had complex behavior which should be tested
  // of the render function.
  jasmineReact.spyOnClass(HelloRandom, "getRandomAuthor").andReturn({name: "Fake Use

  var myHelloRandom = TestUtils.renderIntoDocument(<HelloRandom />);

  expect(myHelloRandom.getDOMNode().textContent).toBe("Fake User is an author and th
});
```

---

If you are familiar with jasmine's `spyOn`, you will be happy to know that `jasmineReact.spyOnClass` returns the same value as `spyOn`, so you can chain jasmine calls onto the end of it in the same way. For example, we chain `andReturn({name: "Fake User", githubUsername: { "fakeGithub" } })` to our call to `jasmineReact.spyOnClass`. Now our spec will pass as desired.

## Assert a spy was called

To finish up our investigation about using spies in our react tests, let's look at how to assert about a spy being called. If you've never done that before, you might be wondering why you'd ever care if/how a spy was called. One use case would be a react component renders a child component, which we will mock out. The parent component defines a callback that the child component can call. We want to assert that is wired up correctly -- when the child component calls the callback, the correct function in the parent is called. Let's start with our `UserBadge` example we used previously in this chapter:

---

```
...
var UserBadge = React.createClass({
  getDefaultProps: function(){
    return {
      friendlyName: "Billy McGee",
      userSlug: "billymcgee"
    };
  },
  render: function(){
    return (
      <div>
        <h1>{this.props.friendlyName}</h1>
        <UserImage slug={this.props.userSlug} />
      </div>
    );
  }
});
```

---

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

---

Let's write a test which mocks out the `UserImage` component and calls an `imageClicked` function on it:

---

```
...
describe("assert spy was called", function(){
  var mockUserImageComponent;

  beforeEach(function(){
    mockUserImageComponent = React.createClass({
      render: function(){
        return (<div className="fake">Fake User Image!!</div>);
      }
    });

    rewireJasmine.rewire(UserBadge, "UserImage", mockUserImageComponent);
  });

  it("should pass a callback to the imageClicked function to the UserImage component",
    jasmineReact.spyOnClass(UserBadge, "imageClicked"));

  var userBadge = TestUtils.renderIntoDocument(<UserBadge />);
  var imageComponent = userBadge.refs.image;
  imageComponent.props.imageClicked();

  expect(jasmineReact.classPrototype(UserBadge).imageClicked).toHaveBeenCalled();
});
...

```

---

Before we run this test, let's examine what it's doing:

1. spy on the `imageClicked` function, so we can assert if it was called
2. mocking out the `UserImage` component with a `mockUserImageComponent`
3. rendering the `UserBadge` component
4. accessing the `mockUserImageComponent` component by calling `userBadge.refs.image`
5. calling the `imageClicked` function on the `imageComponent.props`. (The `imageClicked` function is on the props, because we will be passing the callback to the image component via props.)
6. asserting the correct function is called on the `UserBadge` component.

When we run this test, it will fail with the following error message:

---

```
...
PhantomJS 1.9.7 (Mac OS X) HelloRandom assert spy was called should pass a callback to t
...
```

---

~~This is because our `UserImage` component doesn't have an `imageClicked` function, so we can't~~

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

---

```
...
var UserBadge = React.createClass({
  getDefaultProps: function(){
    return {
      friendlyName: "Billy McGee",
      userSlug: "billymcgee"
    };
  },
  imageClicked: function(){

  },
  ...
});
```

---

Now when re-run this test, it will fail with the following error message:

---

```
PhantomJS 1.9.7 (Mac OS X) HelloRandom assert spy was called should pass a callback to t
```

---

This is failing because `imageComponent` is undefined, so we can't call props on it. `imageComponent` is undefined at the moment, because we didn't setup a `refs.poster` in the `UserBadge` component. So let's do that:

---

```
...
var UserBadge = React.createClass({
  getDefaultProps: function(){
    return {
      friendlyName: "Billy McGee",
      userSlug: "billymcgee"
    };
  },
  render: function(){
    return (
      <div>
        <h1>{this.props.friendlyName}</h1>
        <UserImage slug={this.props.userSlug} ref="image"/>
      </div>
    );
  }
});
```

---

---

#### ADDING A "REF" JUST FOR A SPEC

Adding production code *just* to facilitate a test is acceptable if it's simple and has little to no impact, but shouldn't be something you are super excited about doing. If there is a simple alternative, then you should look into that option too. In this case we could use a call to `React.addons.TestUtils.findRenderedComponentWithType(userBadge, UserImage)` but that helper function will be covered later in this chapter, so let's just push using a `ref` for now.



When we run this test now, we'll get this failure message:

---

PhantomJS 1.9.7 (Mac OS X) HelloRandom assert spy was called should pass a callback to t

---

This error message is a good one to see, because it is saying that the `UserBadge` component didn't pass an `imageClicked` in the props to the `UserImage` component. So let's fix that error:

---

```
...
var UserBadge = React.createClass({
  getDefaultProps: function(){
    return {
      friendlyName: "Billy McGee",
      userSlug: "billymcgee"
    };
  },
  imageClicked: function(){

  },
  render: function(){
    return (
      <div>
        <h1>{this.props.friendlyName}</h1>
        <UserImage slug={this.props.userSlug} imageClicked={this.imageClicked} ref="imag
      </div>
    );
  }
});
```

---

And now the test should pass! We have successfully validated a our function was called.

## TestUtils.Simulate

### Test Utils Simulate

[Simulate] is possibly the single most useful utility in `ReactTestUtils`. -- React Documentation

---

Let's drive out this code with our tests first by creating our spec for `ClickMe`:

---

```
/** @jsx React.DOM */
var React = require("react/addons");
```

---

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

```

describe("ClickMe", function(){

  describe("Simulate.Click", function(){

    it("should render", function(){
      TestUtils.renderIntoDocument(<ClickMe />);
    });

  });
});

```

---

When we run this test it will fail because ClickMe isn't defined. Let's create that module and require it in our spec.

---

```

/** @jsx React.DOM */
var React = require("react");
var ClickMe = React.createClass({
  render: function(){
    return (
      <div></div>
    );
  }
});

module.exports = ClickMe;

...
var TestUtils = React.addons.TestUtils;

var ClickMe = require('../..../client/testing_examples/click_me');
...

```

---

Now our test should be passing because it can render successfully, so let's add some behavior to our ClickMe component where the component says how many times it has been clicked:

---

```

...
describe("ClickMe", function(){
  describe("Simulate.Click", function(){
    var subject;
    beforeEach(function(){
      subject = TestUtils.renderIntoDocument(<ClickMe />);
    });

    it("should output the number of clicks", function(){
      expect(subject.getDOMNode().textContent).toBe("Click me counter: 1");
    });

  });
});

```

---

---

#### RENDER IN BEFORE EACH

Note how in this test we are calling `renderIntoDocument` in a `beforeEach` and saving it into a variable called `subject`. This is a common pattern when our specs can have a common setup and we don't need to repeat the `renderIntoDocument` part in each test.

---

This will fail because our render function is still returning an empty `<div>` tag, so let's fix that:

---

```
...
var ClickMe = React.createClass({
  render: function(){
    return (
      <h1>Click me counter: 0</h1>
    );
  }
});

...
```

---

Hooray, another passing test! Don't feel dirty about hardcoding "0" into your component to make a test pass. Once the test is improved to necessitate a real value being used in the component, then you can change it then.

Now let's improve our test to simulate the user clicking on the `<h1>` tag, and verifying the text changes.

---

```
...
it("should increase the count", function(){
  expect(subject.getDOMNode().textContent).toBe("Click me counter: 0");

  // click on the <h1> dom node
  TestUtils.Simulate.click(subject.getDOMNode());

  expect(subject.getDOMNode().textContent).toBe("Click me counter: 1");
});

...
```

---

Note how we call the "click" function on the `TestUtils.Simulate` utility and pass in the DOM node which should be receiving the click. If we wanted to pass in any event data, we'd pass that as a second argument to the click function.

This test will fail because we aren't listening to any click events in our component, so let's add that behavior:

---

```
...
var ClickMe = React.createClass({
```

---

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

```

    },
    headingClicked: function(){
      var clicks = this.state.clicks;
      this.setState({clicks: clicks + 1});
    },
    render: function(){
      return (
        <h1 onClick={this.headingClicked}>Click me counter: {this.state.clicks}</h1>
      );
    }
  });
  ...

```

---

And now our test will pass and we are finished, hooray!

You are starting to get the basics under your belt (rendering a component, spying on a function in a component, mocking out a subcomponent, simulating events to a component), now we want to flush out a concept which we glossed over earlier in this chapter: finding components which are rendered by your component. Mastering the finder methods of the TestUtils will make your tests more descriptive, less brittle, and shorter. This section is about explain the utility in illustrative detail, so you don't need to be concerned about TDD for this code:

---

```

/** @jsx React.DOM */
var React = require("react");
var CompanyLogo = require("./company_logo");
var NavBar = React.createClass({
  render: function(){
    return (
      <div>
        <CompanyLogo />
        <ul>
          <li className="tab active">Tab 1</li>
          <li className="tab">Tab 2</li>
          <li className="tab">Tab 3</li>
          <li className="tab">Tab 4</li>
          <li className="tab">Tab 5</li>
        </ul>
      </div>
    );
  }
});

module.exports = NavBar;

```

---

```

/** @jsx React.DOM */
var React = require("react");
var CompanyLogo = React.createClass({
  render: function(){
    return ();
  }
});

module.exports = CompanyLogo;

```

---

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

Let's say you want to find all `<li>` components which are rendered by the `<NavBar>` component. Then you are looking for a function called `TestUtils.scrYRenderedDOMComponentsWithTag`.

---

### COMPONENTS VS ELEMENTS

In the above description I said “find all `<li>` components”, instead of saying “find all `<li>` elements” - this is on purpose. The `TestUtils` finder methods return React Components, not DOM Nodes. This allows you to access all of the first class React.js properties for that component which is very helpful, one of which is `.getDOMNode()` if you actually care about the underlying dom node.

---

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

var NavBar = require('.././../client/testing_examples/nav_bar');
var CompanyLogo = require('.././../client/testing_examples/company_logo');

describe("TestUtils Finders", function(){

  var subject;

  beforeEach(function(){
    subject = TestUtils.renderIntoDocument(<NavBar />);
  });

  describe("scrYRenderedDOMComponentsWithTag", function(){
    it("should find all components with that html tag", function(){
      var results = TestUtils.scrYRenderedDOMComponentsWithTag(subject, "li");
      expect(results.length).toBe(5);
      expect(results[0].getDOMNode().innerHTML).toBe("Tab 1");
      expect(results[1].getDOMNode().innerHTML).toBe("Tab 2");
    });
  });
});
```

---

scrY?

You might be wondering, why does this function start with the letters “scrY” and how am i supposed to pronounce that??

It's pronounced like combining “s” with “cry” and should end up sounding very similar to the word “sky”. (If it's really important to pronounce it correctly, google it and there is a “listen” button which will play the sound).

“scrY” means to look into a crystal ball to find an object. So in this context, your

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

---

For all “screy\*” functions, there is a similar function which is called “find\*”. This method will do the same behavior but only return one component, instead of an array. If multiple components are found which matches the arguments, then an error is thrown.

Let’s say that your <NavBar> component renders another component you have like <CompanyLogo> and you want to find that component. You can do that with the `screyRenderedComponentsWithType` function:

```
...
it("should find composite DOM components", function(){
  var results = TestUtils.screyRenderedComponentsWithType(subject, CompanyLogo);
  expect(results.length).toBe(1);

  // even though we rendered (and searched for) a <CompanyLogo>, that is a composite
  // component which actually is an "<img />" tag.
  expect(results[0].getDOMNode().tagName).toBe("IMG");
});
...
```

---

#### COMPONENTS WITH TYPE LIMITATION

As of React.js v11.0, if you want to use `screyRenderedComponentsWithType` for components like `React.DOM.div` or `React.DOM.li`, you will be out of luck because those are native components instead of composite components. We’re not sure if the React implementation will change in this regard, but you can find more information here: <https://github.com/facebook/react/issues/1533>

---

Last up, what about if you want to find a rendered component by a CSS class. Just like the others, the utility function is named `screyRenderedDOMComponentsWithClass`.

```
...
describe("screyRenderedDOMComponentsWithClass", function(){
  it("should find all components with that class attribute", function(){
    var tabs = TestUtils.screyRenderedDOMComponentsWithClass(subject, "tab");
    var activeTabs = TestUtils.screyRenderedDOMComponentsWithClass(subject, "active");

    expect(tabs.length).toBe(5);
    expect(activeTabs.length).toBe(1);
  });
});
...
```

---

Now that you’ve covered a lot of ground with testing a React component, let’s circle back to the first

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

`React.addons.TestUtils.renderIntoDocument` is that the function is called `renderIntoDocument` and not `renderIntoBody` or just `render`. This is very intentional. The react authors are trying to tell you that this function will render in the html document in a detached DOM node. So your component will not actually be inside of the `<body>` tag or be viewable on the jasmine testing page. For most applications this is approach is just fine and is often preferable, but what happens when your tests require that your component is actually visible and in the `<body>` tag?

To correctly test these scenarios we need to use `React.renderComponent` because it will render the component into a dom element which can be in the `<body>` tag just like the real application code. But todo this we need to be very careful to clean up the state of our parent dom element or else that rendered component will affect subsequent tests and pollute them. To explain this technique, let's start with an example: We want a component which will output it's own width and height. Sounds easy enough - let's get started.

---

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

describe("Footprint", function(){
  describe("render", function(){
    it("should output the width of the component", function(){
      React.renderComponent(<Footprint />);
    });
  });
});
```

---

This will fail with `ReferenceError: Footprint is not defined`, so let's fix that error:

---

```
...
var HelloWorld = require('../client/testing_examples/hello_world');
var Footprint = require('../client/testing_examples/footprint');
...

/** @jsx React.DOM */

var React = require("react");

var Footprint = React.createClass({
  render: function(){
    return (
      <div></div>
    );
  }
});

module.exports = Footprint;
```

---

This fails with this error: `Error: Invariant Violation: _registerComponent(...): Target container is not a DOM element.`

---

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

Unlike `TestUtils.renderIntoDocument`, `React.renderComponent` takes a second argument, which is the DOM element we want to render into. To solve this, let's create a `<div>` in our spec and append it to the body tag.

---

```
// NOTE: This test is incomplete, please don't emulate this test. It has test pollu
it("should output the width of the component", function(){
  var el = document.createElement("div");
  document.body.appendChild(el);

  React.renderComponent(<Footprint />, el);
});
```

---

Now our test can successfully render a component into an attached DOM node. Let's improve the test to validate the resulting HTML outputs the width of the component:

---

```
// NOTE: This test is incomplete, please don't emulate this test. It has test poll
it("should output the width of the component", function(){
  var el = document.createElement("div");
  document.body.appendChild(el);

  var myComponent = React.renderComponent(<footprint></footprint>, el);
  expect(myComponent.getDOMNode().textContent).toContain("component width: 100");
});
```

---

This will fail with the following error message: Expected '' to contain 'component width: placeholder-value'. Let's add in this functionality:

---

```
var Footprint = React.createClass({
  getInitialState: function(){
    return { width: undefined };
  },
  componentDidMount: function(){
    var componentWidth = this.getDOMNode().offsetWidth;
    this.setState({width: componentWidth});
  },
  render: function(){
    var divStyle = {width: "100px"};
    return (<div style={divStyle}>component width: {this.state.width}</div>);
  }
});
```

---

Our test passes, hooray! But we're not done yet. That component we just rendered will stick around for other tests, so we need to make sure we unmount it before the next test starts. So let's add that cleanup to our spec:

---

```
describe("Footprint", function(){
  describe("render", function(){
```

---

```
    var el;
```

---

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



```

beforeEach(function(){
  el = document.createElement("div");
  document.body.appendChild(el);
});

afterEach(function(){
  // we need to tell React to unmount the component to clean everything up
  React.unmountComponentAtNode(el);

  // we should remove the <div id="content"></div> as well, so the beforeEach functi
  // new one which is unique and fresh for each test
  el.parentNode.removeChild(el);
});

it("should output the width of the component", function(){
  var myComponent = React.renderComponent(<Footprint />, el);
  expect(myComponent.getDOMNode().textContent).toContain("component width: placeholder");
});
});

```

---

Now we can successfully render our component into the DOM and we don't have to worry about test pollution issues. If you need to do this often, Jasmine react offers a helper function to do this, which will automatically unmount the component at the end of each test. Here's the same example using the `renderComponent` function from `jasmine-react`:

---

```

...
var jasmineReact = require("jasmine-react-helpers");
...
describe("jasmineReact.renderComponent", function(){

  var el;

  beforeEach(function(){
    // put a DOM element into the <body> tag
    el = document.createElement("div");
    document.body.appendChild(el);
  });

  afterEach(function(){
    // we should remove the <div></div> as well, so the beforeEach function creates a
    // new one which is unique and fresh for each test
    el.parentNode.removeChild(el);
  });

  it("should return the component which is mounted", function(){
    var myComponent = jasmineReact.renderComponent(<HelloWorld />, el);

    // you can assert things on the component
    expect(myComponent.props.name).toBe("Bleeding Edge React.js Book");
  });

  it("should put the component into the DOM", function(){
    var myComponent = jasmineReact.renderComponent(<HelloWorld />, el);

```

---

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

```
    expect(myComponent.getDOMNode().offsetHeight).not.toBe(0);  
  });  
  
});
```

---

The main difference here is that we no longer need to call `React.unmountComponentAtNode(e1)`; because that is handled by `jasmineReact` for us -- anything which `jasmineReact` renders, it will clean up.

---

#### IS RENDER INTO <BODY> WORTH IT?

With all of this discussion about DOM cleanup and test pollution, is rendering your component into the actual DOM worth it? In most cases the answer is no. We'd recommend you use `React.addons.TestUtils.renderIntoDocument` instead whenever possible. Then once your application code requires you to render your components into an attached DOM node, then start using the `renderComponent` approach in your tests.

---

## Summary

You have now walked through all of the fundamentals of unit testing a React.js application. In later chapters we will learn how to test React.js mixins, serverside components, along with a short introduction to functional testing.

<sup>2</sup> <http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html>

<sup>3</sup> <http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf>

<sup>4</sup> <https://www.youtube.com/watch?v=1wHr-O6gEfc>



◀ PREV  
14. Tools and Debugging

NEXT ▶  
16. Build Tools