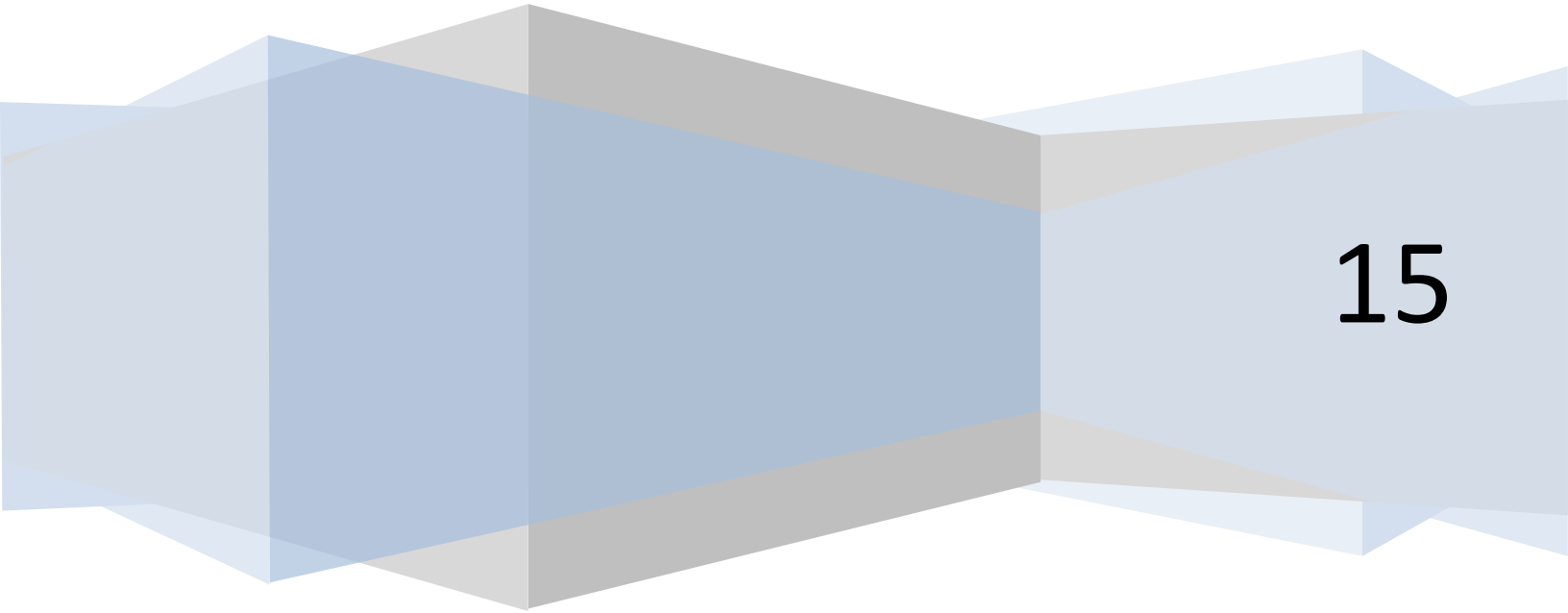
Developing a React Edge:The javaScript Library for User Interface

React.js

Amit Modi



15

Developing a React Edge: The JavaScript Library for User Interfaces

Chapter 1. Introduction to React.js

Background

In the early days of web application development, it was common to send a request to the server, the server would then respond with the full page. This was the only way to create web applications. This approach was very simple from a development perspective. As you didn't need to worry about the events happening in the browser. All you need to do was create the whole page from scratch every time.

Languages like PHP make this kind of development very simple, it is also very simple to create functional components in PHP to help the developer reuse code and reason about what their application was supposed to do. The simplicity of development helped make PHP an extremely popular language to write web applications.

Using this approach it was very hard to create an awesome user experience, as each time the user wanted to do something they had to make another request to the server and wait for the response. In doing so, the user lost all the state they had built up in their page.

In an effort to create better user experiences people started to write libraries to help render applications in the browser with Javascript. These libraries used a variety of means to control the DOM, from simple parameterized HTML templating systems to systems controlling the entire application. As people started to use these libraries in larger and larger applications, it becomes increasingly hard to reason about how your application is supposed to behave as these applications are the result of the long tangled thread of events. In comparison to the old PHP way of doing applications, these modern client side applications are very hard to do well.

React started as a port of a PHP framework by Facebook called XHP. Being a PHP framework, XHP was designed to render our the entire page every time a request is made. React was born to bring the PHP style work flow to client side applications.

React.js has a narrow scope. It is concerned with only two things:

1. Updating the DOM
2. Responding to events

React has no opinions on AJAX, routing, storage, or how to structure your data. It is not a Model-View-Controller framework; if anything, it is the V in MVC. This narrow scope gives you the freedom to incorporate React into a wide variety of systems. In fact it has been used to render views in several popular MVC frameworks.

Rendering the entire page every time some state changes is incredibly slow in Javascript. However, React has a very powerful rendering system.

Like high-performance 3D game engines, React is built around render functions that take the state of the world and translate it into a virtual representation of the resulting page. Whenever React is informed of a state change, it re-runs those functions to determine a new virtual representation of the page, then automatically translates that result into the necessary DOM changes to reflect the new presentation.

At a glance, this sounds like it should be slower than the usual JavaScript approach of updating each element on an as-needed basis. Behind the scenes, however, React does just that: it has a very efficient algorithm for determining the differences between the current virtual page representation and the new one. From those differences it makes the minimal set of updates necessary to the DOM.

What makes this a win for performance is that it minimizes reflows and unnecessary DOM mutations, both of which are common culprits for poor performance.

The bigger your interface gets, the more likely it is to have one interaction that triggers an update, which in turn triggers another update, which in turn triggers another. When these cascading updates are not batched properly, performance starts to degrade substantially. Worse, sometimes DOM elements are updated multiple times before arriving in their final state.

Not only does React's virtual representation diffing minimize these problems by performing the minimal set of updates in a single pass, it also simplifies the maintenance of your application. When the state of the world changes based on user input or external updates, you simply notify React of the state change and it takes care of the rest automatically. There's no need to micromanage.

Book Overview

This book will take you through 4 main topic areas to help you develop an edge with React.js.

Creating and Composing Components

The first 7 chapters of the book are all about creating and composing React components. These chapters will give a good understanding of how to use React.

1) Introduction to React.js
This first chapter introduces React.js, covering the background and the book overview.

2) Using JSX and basic React.js Components
JSX (Javascript XML) is a way of writing declarative XML style syntax in side Javascript. You will learn how to use JSX with React and how to build basic React.js Components. Most of the examples in this book and the example app use JSX.

3) React.js Component lifecycle
React.js is often creating and destroying components during the render process. React.js provides many functions you can hook into during the lifecycle of your components. You should gain an understanding of how to manage the lifecycle of a component to ensure you don't create memory leaks in your applications.

4) Data flow in React.js
It is important to know how data is passed down through the component tree and what data is safe to change. React.js has a very clear separation of `props` and `state`. This chapter will teach you how to use props and state correctly in your React.js components.

5) Event Handling
React.js implements event handling in a declarative manner. Event handling is an important part of any dynamic UI, learning to master them is essential, fortunately React.js makes this very simple.

6) Composing Components
React.js encourages you to make small, precise components that do a specific job. You then need to create orchestration layers in your application to compose these components. This chapter will teach you how to use your components in other components.

7) React.js Mixins
Mixins in React.js provide a way to share common functionality in many React.js components. Using mixins is another way you can break down your components into smaller more manageable parts.

Advanced Topics

Once you have mastered the basics of React you will move on to some more advanced topics. These next 6 chapters will help you really hone your React skills and understanding of how to build great React components.

8) Accessing the DOM from React.js
Even with all the power of the virtual DOM in React.js, sometimes you will still need to access the raw DOM nodes in your applications. This may be to enable you to use existing Javascript libraries or to get more control over your components. This chapter will teach you where in the lifecycle of your React.js components you can safely access the DOM and when to release your control on the DOM to avoid leaking your DOM nodes.

9) Building Forms with React.js
Using HTML form elements is one of the best ways of receiving input from your users. However HTML form elements are very stateful. React.js provides a way to move most of the state from your form elements into your React.js components. This gives you incredible control over your form elements.

10) Animations
As web developers we are already blessed with a very declarative way of defining high performance animations, CSS. React.js encourages the use of CSS for your

animations. This chapter will take you the mechanism that React.js provides to help you leverage CSS for animating your React.js components.

11) Performance Tuning your Components
The virtual DOM in React.js gives you great performance right out of the box. There is always room for improvement. React.js provides a way to tell the render that it doesn't need to re-render your component if you know your component has not changed props or state. Doing this can greatly improve the speed of your applications.

12) Server Side Rendering
Many applications need SEO, fortunately React.js can be rendered to string in a non-browser environment like nodejs. Server side rendering can also improve first page load times of your applications also. However writing your application to support both server side rendering and client side rendering can be difficult. This chapter provides you with some strategies for isomorphic rendering and highlights some of the more challenging considerations you will encounter with server side rendering.

13) React.js Addons
React.js provides some of the advanced functionality via it's addons package. Many of these addons are delivered as mixins. Learning how to use these addons will help you build better functionality faster with React.js.

Tooling for React.js

React has some fantastic developer tools and test suits. Learning to use them will help you write robust applications. This section is broken into 3 chapters.

14) React.js Debugging Tools
React.js provide a Google Chrome plugin that enables you to inspect your application visualizing your React.js components. You will learn how to setup this plugin and how to use it.

15) Writing Tests for React.js
Writing tests is an important part of ensuring you don't introduce bugs into existing working code as your applications grow. Writing tests also help you write better code as it encourages you to write more modular code. This chapter will guide you through how to test all aspects of your React.js components.

16) Build Tools
As you write larger Javascript applications you will start to require a way to automate packaging your code for deployment. React.js is supported by the major code packaging tools, Browserify and Webpack. This chapter will show you how to configure Browserify and Webpack to support React.js and JSX.

Working with React.js

The final 3 chapters cover important aspects of working with React.

17) Architectural Patterns
React.js provides only "V" in "MVC", it is however very flexible in plugin in with other frameworks and systems. This chapter will help guide you in designing larger scale applications with React.js.

18) Other Javascript libraries in the React.js Family
Facebook open sourced React.js as part of it's over all architectural tools used in house. As time has progressed many other libraries have been released by Facebook that are designed to work seamlessly with React.js. This chapter will give you insight on how these other libraries fit into the React.js family.

19) Use Cases for React.js
While React.js focuses on the web, it can be used anywhere javascript is supported. This chapter looks at some other the other ways you can use React.js outside of the traditional web use case.

Chapter 2. JSX

React is a rather opinionated library with strong beliefs on how you should structure and manage the data in your app. JSX is the result of one of those strongly held beliefs. Specifically React embraces the idea that markup, and the code which generates it, are inherently tied together. This is expressed in React components by rendering the markup directly within Javascript, using the full expressive power of the language.

To that end, React introduces an optional markup language remarkably similar to HTML. For example, a function call in plain React to create a header might look like this.

```
React.DOM.h1({className: 'question'}, 'Questions');
```

But with JSX it becomes much more familiar and terse looking markup.

```
<h1 className="question">Questions</h1>
```

In this chapter we'll walk you through the benefits of JSX and how to use it, along with some of the gotchas that separate it from HTML. Remember, JSX is optional. If you choose not to employ JSX you can skip to the end of the chapter for tips on using React without it.

What is JSX

JSX stands for JavaScript XML — an XML-like syntax for constructing markup within React components. React works without JSX, however it's use can make your components more readable and it is recommended.

Compared to past attempts at embedding markup in Javascript there are a few distinguishing characteristics that set JSX apart.

1. JSX is a syntactic transform — each JSX node maps to a javascript function.
2. JSX neither provides nor requires a runtime library.
3. JSX doesn't alter or add to the semantics of JavaScript — it's just simple function calls.

The similarity of JSX to HTML is what gives it so much expressive power within React. But in the end it maps to plain-old-javascript functions. Here we'll discuss the benefits of JSX & it's purpose within your app, as well as the key differences between JSX and HTML.

Benefits of JSX

A question many ask when considering JSX is why? Why use it at all when there are plenty of existing templating languages? Why not use plain javascript instead? After all, JSX simply maps to javascript functions.

Here are a few benefits or JSX which we'll discuss in turn.

- JSX is easier to visualize than javascript functions
- The markup is more familiar to designer and the rest of your team
- Your markup becomes more semantic, more meaningful.

Easy to Visualize

It's easiest to demonstrate JSX by example. Below you'll see the render function for a simple page-divider component used within the sample-app that accompanies this book. Notice that its intent is more clear and readable in JSX format.

Plain Javascript

```
render: function () {
  return React.DOM.div({className:"divider"},
    "Label Text",
    React.DOM.hr()
  );
}
```

JSX Markup

```
render: function () {
  return <div className="divider">
    Label Text<hr />
  </div>;
}
```

While it's certainly arguable, many agree the JSX markup is easier to understand and easier to debug. Which brings us to another key benefit of JSX: it's easier for non-programmers already familiar with html to work with it.

Familiarity

Many development teams include non-developers, from UI & UX designers who are familiar with html to quality-assurance teams responsible for thoroughly testing the product. These team members can more easily read and contribute code to a JSX project. Anyone with a familiarity with XML based languages can easily adopt JSX.

In addition, because React components capture all possible representations of your DOM (more about this in the section or Components,) JSX makes a great way to represent and visualize this structure in a compact and succinct way.

Semantics

Along with familiarity, JSX transforms your javascript code into more semantic, meaningful markup. This allows you the benefit of declaring your component structure and information flow using an HTML-like syntax, knowing it will transform into simple javascript functions.

React defines all the HTML elements you'd expect in the `React.DOM` namespace. It also allows you to use your own, custom components within the markup. We teach you all about custom components [link to chapter], so here we'll simply show how JSX can make your javascript more readable.

Let's refer back to the divider component mentioned above. It renders some header text on the left and has a horizontal-divider that stretches to fill the right. The HTML for this diver looks like this:

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

However, after wrapping this in a `Divider` React component you can use it you

would any other html element, with the added benefit of markup with richer semantics.

```
<Divider>Questions</Divider>
```

Composite Components

Now that you've discovered some of the benefits of JSX and seen how it can be used to express a component in a compact markup format, let's look at how it helps us assemble multiple components.

This section will show you:

- How to set up a JavaScript file that contains JSX
- Walk you through how to assemble a component
- Discuss component ownership and the parent/child relationship

Let's look at each in turn.

Setup

While JSX allows you to express your components using markup, eventually it

must be transformed into Javascript. In order for the JSX transform to know it should operate on a file you must insert the `/** @jsx React.DOM */` pragma at the top of each file containing JSX. Without this JSX will not know it should process the file for React.

The approach has a few benefits such as:

- The transformer can ignore all files without this pragma
- It allows JSX to live beside non-JSX JavaScript files

As mentioned earlier all element tag names in JSX map to plain JavaScript functions. React predefines all of the HTML tags you'd expect in the `React.DOM` namespace. The JSX transform conveniently pulls these in as top-level variables, which allows you to use `<div>...</div>` without needing to tediously specify `var div = React.DOM.div` in each JSX file.

All of your custom components, however, must be declared within the scope of your file, otherwise the React will not know how to resolve the function. As an example, throughout the sample app that accompanies this book you'll see code like this, specifying the custom components to be used within the scope of the file.

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

```
// Declare our custom component so JSX can resolve it
var Divider = require("./divider");

// Use any `React.DOM.*` element or `Divider` below
<div>
  <Divider>...</Divider>
  <p>...</p>
</div>
```

Now that you've seen how to set up a JSX file you are ready to define a custom component using JSX.

Defining a Custom Component

Continuing with our previous example of the page divider, here again is the HTML we desire as output.

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

To express this HTML as a React component *all you have to do* is wrap it like this so the render function returns the markup.

```
var Divider = React.createClass({
  render: function () {
    return (
      <div className="divider">
        <h2>Questions</h2><hr />
      </div>
    );
  }
});
```

Of course this is currently a single-use component. To be truly useful you need the ability to express the text within the `h2` tag dynamically. But before we discuss making the text dynamic we need to discuss children.

Child Nodes

In HTML you render a header using `<h2>Questions</h2>` where the text "Questions" became a child text-node of the `h2` element. So the goal here is to express the divider in JSX similarly, like this:

```
<Divider>Questions</Divider>
```

React captures all child nodes between the open and close tags in an array on the component's props,`this.props.children`. In this example `this.props.children === ["Questions"]`.

Armed with this knowledge you can swap out the hard-coded text "Questions" with the variable`this.props.children`. React will now render anything you place inside the `<Divider>` tags.

```
var Divider = React.createClass({
  render: function () {
    return (
      <div className="divider">
        <h2>{this.props.children}</h2><hr />
      </div>
    );
  }
});
```

Now you can use the `<Divider>` component like you would any html element.

```
<Divider>Questions</Divider>
```

When run through the JSX transformer the above declaration will transform into this JavaScript.

```
var Divider = React.createClass({
  render: function () {
    return React.DOM.div(
      {className="divider"},
      React.DOM.h2(null, this.props.children),
      React.DOM.hr(null)
    );
  }
});
```

The output will be exactly what you expect.

```
<div className="divider">
  <h2>Questions</h2><hr />
</div>
```

How is JSX Different than HTML?

JSX is html-like, but it is not a perfect replication of the html syntax (for good reason.) In fact the JSX spec states:

> This specification does not attempt to comply with any XML or HTML specification. JSX is designed as an ECMAScript feature and the similarity to XML is only for familiarity.[1]

Here we explore some of the key differences between JSX and the HTML syntax.

Attributes

In HTML we set the attributes of each node inline, like this:

```
<div id="some-id" class="some-class-name">...</div>
```

JSX implements attributes in the same manner, with the huge advantage that you can set attributes to dynamic javascript variables. You do this by wrapping a javascript variable in curly-brackets instead of quotes.

```
var surveyQuestionId = this.props.id;
var classes = 'some-class-name';
...
<div id={surveyQuestionId} className={classes}>...</div>
```

For more complex situations you can set an attribute to the result of a function call.

```
<div id={this.getSurveyId()} >...</div>
```

Now each time React chooses to render a component the variables and function calls will be evaluated and the resulting DOM will reflect this new state.

Conditionals

React embraces the idea that a component's markup and the logic that generate it are inherently tied together. This means you have the full logical power of javascript at your fingertips, such as loops and conditionals.

It can be tricky to add conditional logic to your components since if/else logic is hard to express as markup. Adding if statements directly to JSX will render invalid JavaScript:

```
<div className={if(isComplete) { 'is-complete' }}>...</div>
```

So the solution is to use one of the following:

- Use ternary logic
- Set a variable and refer to it in the attribute
- Offload the switching logic to a function

Here is a quick example showing what each may look like.

Using the ternary operator — *whitespace added for clarity*

```
...
render: function () {
  return <div className={
    this.state.isComplete ? 'is-complete' : ''
  }>...</div>;
}
...
```

While the ternary operator works well for text, it can be cumbersome and difficult to read when you want to use a React component in either case. For these situations it is better to use the following methods.

Using a variable

```
...
getIsComplete: function () {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function () {
  var isComplete = this.getIsComplete();
  return <div className={isComplete}>...</div>;
}
```

```
...
```

Using a function call

```
...
getIsComplete: function () {
  return this.state.isComplete ? 'is-complete' : '';
},
render: function () {
  return <div className={this.getIsComplete()}>...</div>;
}
...
```

Non-DOM Attributes

The following attributes are reserved words in JSX.

- `key`
- `ref`
- `dangerouslySetInnerHTML`

Keys

`key` is an optional unique identifier. During runtime a component might move up or down the component tree, for example as the user performs a search or items are added or removed from a list. When this happens you component might be needlessly destroyed and recreated.

By setting a unique key on a component that remains consistent throughout render passes you inform React so it more intelligently decides when to reuse or destroy a component, improving rendering performance. Then when two items already in the DOM switch positions React can match the keys and move them without completely re-rendering their DOM.

References

`ref` allows parent components to keep a reference to child components available outside of the render function.

You define a ref in JSX by setting the attribute to the desired reference name.

```
...
```

```
render: function () {
  return <div>
    <input ref="myInput" ... />
  </div>;
}
...
```

And later you can access this ref by using `this.refs.myInput` anywhere in your component. The object you access through this ref is called a **backing instance**. It's not the actual DOM, but a description of the component React uses to create the DOM when necessary. To access the actual DOM node you use `this.refs.myInput.getDOMNode()`.

For more detail see the discussion on parent/child relationships vs ownership in chapter 4.

Setting Raw HTML

`dangerouslySetInnerHTML` — sometimes you need to set html content as a string, especially when working with third party libraries that manipulate DOM via strings. To improve React's interoperability this attribute allows you to use html strings, but it's not recommended if you can avoid it. To use this property set it to an object with key `__html` set, like this:

```
...
render: function () {
  var htmlString = {
    __html: "<span>an html string</span>"
  };
  return <div dangerouslySetInnerHTML={htmlString} ></div>;
}
...
```

Events

Event names are normalized across all browsers and are represented camelCased. For example`change` becomes `onChange`, and `click` becomes `onClick`. When capturing an event in JSX it's as simple as assigning the property to a method on the component.

```
    ...
    handleClick: function (event) {...},
```

```
render: function () {
  return <div onClick={this.handleClick}>...</div>
}
...
```

Note that React automatically binds all of a component's methods, so you never need to manually bind the context.

```
...
handleClick: function (event) {...},
render: function () {
   // anti-pattern - manually binding the function context
   // to a component instance is unnecessary in React.
  return <div onClick={this.handleClick.bind(this)}>...</div>
}
...
```

For more details on the event system in React reference the chapter on Forms.

Special Attributes

Because JSX transforms to plain javascript function calls there are a few keywords we cannot use — `class` and `for`.

To create a form label with the `for` attribute use `htmlFor`.

```
<label htmlFor="for-text" ... >
```

To render a custom `class` use `className`. This might seem odd if you're used to

HTML, but it is more consistent with vanilla javascript, where we can access the class of an element using `elem.className`.

```
<div className={classes} ... >
```

Styles

Lastly we come to the inline style attribute. React normalizes all styles to camelCased names, consistent with the DOM style JavaScript property.

To define a custom style attribute simply pass a JavaScript object with camelCase property names and the desired css values.

```
var styles = {
  borderColor: "#999",
  borderThickness: "1px"
};
React.renderComponent(<div style={styles}>...</div>, node);
```

React Without JSX

All JSX markup is eventually transformed into simple JavaScript function calls. So JSX is not *necessary* for using React.

It Functions All The Way Down

The pattern for creating a component is simple. It's just a function, after all. HTML tagNames are defined in the `React.DOM` namespace. So creating a div is as simple as calling the function.

```
React.DOM.div();

// <div></div>
```

The first argument to a component function is the props object. So customizing the attributes of a div would look like the following code. Each key in the props object maps to the attribute name on the rendered div element.

```
React.DOM.div({className: "divider"});

// <div class="divider"></div>
```

Child nodes are passed in as optional arguments after the props object. Using our divider example from above, which as two child nodes, we can render it as raw JavaScript with the text node and `hr`component passed in as the second and third arguments, respectively.

```
React.DOM.div(
  {className: "divider"},
  "Label Text",
  React.DOM.hr()
)

// <div class="divider">Label Text<hr/></div>
```

Shorthand

Writing `React.DOM.div` can be tedious after a while. It can be convenient to save a reference using a shorter variable name, e.g. `R`. Thus we can express the above div a bit more tersely.

```
var R = React.DOM;
//...
R.div(
  {className: "divider"},
  "Label Text",
  R.hr()
);
```

Or, if you prefer to access the component as top-level variables you can reference them directly.

```
var div = React.DOM.div;
var hr = React.DOM.hr;
//...
div(
  {className: "divider"},
  "Label Text",
  hr()
);
```

Chapter 3. Component Lifecycle

React components are simply state-machines. Their output represents the DOM for a given set of props and state. Throughout a component's lifecycle, as it's props or state change, its DOM representation might change too. As noted in the chapter on JSX, a component is just a Javascript function; for a given input it will always return the same output.

React provides lifecycle hooks for a component to respond to various moments — its creation, lifetime, and teardown. We'll cover them here in order of their appearance — first through instantiation, then through the components life and finally as the component is torn-down.

Lifecycle Methods

React has relatively few lifecycle methods, but they are very powerful.
React offers you all of the methods required to control the props and state of your app throughout its lifecycle. Lets take a look at each in order as they get called on your component.

The lifecycle methods called the first time an instance is created, vs each subsequent instance, varies slightly. For your first use of a component class you'll see these methods called, in order:

- `getDefaultProps`
- `getInitialState`
- `componentWillMount`
- `render`
- `componentDidMount`

For all subsequent uses of that component class you will see the following methods called, in order. Notice `getDefaultProps` is no longer in the list.

- `getInitialState`
- `componentWillMount`
- `render`
- `componentDidMount`

As the app state changes and your component is affected you will see the following methods called, in order:

- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`
- `render`
- `componentDidUpdate`

And lastly, when you are finished with the component you will see `componentWillUnmount` called, giving your instance the opportunity to clean-up after itself.

Now we'll cover each of these three stages: instantiation, lifetime, and cleanup in turn.

Instantiation

As each new component is created and first rendered there are series of methods you can use to setup and prepare your components. Each of these methods has a specific responsibility, as described here.

getDefaultProps

This method is called only once for the component class. The object returned is used to set the default values for any new instances when they are not specified by the parent component.

It is important to note that any complex values, such as objects and arrays, will be shared across all instances — they are not copied or cloned.

getInitialState

Called exactly once for each instance of your component, here you get a chance to initialize the custom state of each instance. Unlike `getDefaultProps` this method is called once each time an instance is created. At this point you have access to `this.props`.

componentWillMount

Invoked immediately before the initial render. This is the last chance to affect the component state before the `render` method is called.

render

Here you build the virtual DOM that represents your components output. Render is the only required method for a component and has specific rules. The requirements of the render method are as follows:

- The only data it can access is `this.props` and `this.state`
- You can return `null`, `false`, or any React component
- There can only be one top-level component (you cannot return an array of elements)
- It must be *pure*, meaning it does not change the state oc modify the DOM output

The result returned from render is not the actual DOM, but a virtual representation that React will later diff with the real DOM to determine if any changes must me made.

componentDidMount

After the render is successful and the actual DOM has been rendered you can access it inside of`componentDidMount` via `this.getDOMNode()`.

This is the lifecyle hook you will use to access the raw dom. For example, if you need to measure the height of the rendered output, manipulate it using timers, or run a custom jQuery plugin, this is where you'd hook into.

Note this method is not called when running on the server.

Lifetime

At this point your component has been rendered to the user and they can interact with it. Typically this involves one of the event handlers getting triggered by a click, tap or key event. As the user changes the state of a component, or the entire app, the new state flows through the component tree and you get a chance to act on it.

componentWillReceiveProps

The props of a component can change at any moment through the parent component. When this happens `componentWillReceiveProps` is called and you get the opportunity to change the new props object and update the state.

For example, within our sample survey application we have an `AnswerRadioInput` that allows users to toggle a radio input. The parent component is able to change this boolean value and we can respond to it like this, updating our own internal state based on the parent's input props.

```
componentWillReceiveProps: function (nextProps) {
  if(nextProps.checked !== undefined) {
    this.setState({
      checked: nextProps.checked
    });
```

```
    }
  }
```

Since this method is called *before* the props are applied to the component you have the opportunity to affect the props as well. As an example, if you required a certain prop to be a boolean you can enforce that here, and the new value will be applied to the component props.

```
componentWillReceiveProps: function(nextProps) {
  // coerce the `checked` value to a boolean
  nextProps.checked = !!nextProps.checked;
}
```

shouldComponentUpdate

React is fast. But you can make it even faster using `shouldComponentUpdate` to optimize exactly when a component renders.

If you are certain that the new props or state will not require your component or any of its children to render, return `false`.

This method is not called during the initial render or after using `forceUpdate`.

By returning false you are telling react to skip calling `render`, and its before and after hooks`componentWillUpdate` and `componentDidUpdate`.

This method is not required and for most purposes you will not need to use it during development. Premature use of this method can lead to subtle bugs, so it's best to wait until you can properly benchmark your bottlenecks before choosing where to optimize.

If your careful to treat state as immutable and only read from props and state in your render method then feel free to override `shouldComponentUpdate` to compare the old props and state to their new replacements.

Another performance-tuning option is the `PureRenderMixin` provided with the React addons. If your component is *pure*, meaning it always renders the same DOM for the same props & state, this mixin will automatically use `shouldComponentUpdate` to shallowly compare props and state, returning false if they match.

componentWillUpdate

Similar to `componentWillMount` this method is triggered immediately before rendering when new props or state have been received.

Note you *cannot* update state or props in this method. You should rely on `componentWillReceiveProps` for updating state during runtime.

componentDidUpdate

Similar to `componentDidMount` this method gives you an opportunity to update the rendered DOM.

Teardown

Once React is done with a component it must be unmounted from the DOM and destroyed. You are provided with a single hook to respond to this moment, performing any cleanup and teardown necessary.

componentWillUnmount

Lastly we have the end of a components life as it's removed from the component heirarchy. This method is called just prior to your component being removed and gives you the chance to clean up. Any custom work you might have done in `componentDidMount`, such as creating timers or adding event listeners should be undone here.

Anti Pattern — Calculated Values as State

Given the possibility of using `getInitialState` to create state from `this.props` it's worth noting an anti-pattern here. React is very concerned with maintaining a single source of truth. Its design makes duplicating the source of truth more obvious, one of React's key strengths.

When considering calculated values derived from props it is considered an anti-pattern to store these as state. For example a component might convert a date to a string representation, or transform a string to uppercase before rendering it. These are not state, and should simply be calculated at render-time.

You can identify this anti-pattern when it's impossible to know inside of your render function if your state value is out-of-sync with the prop it's based from.

```
// Anti-pattern. Calculated values should not be stored as state.
getDefaultProps: function () {
  return {
    date: new Date()
  };
},
getInitialState: function () {
  return {
    day: this.props.date.getDay()
  }
},
render: function () {
  return <div>Day: {this.state.day}</div>;
}
```

The correct pattern is to calculate the values at render-time. This guarantees the calculated value will never be out-of-sync with the props it's derived from.

```
// It's proper to calculate values at render-time.
getDefaultProps: function () {
  return {
    date: new Date()
  };
},
render: function () {
  var day = this.props.date.getDay();
  return <div>Day: {day}</div>;
}
```

However, if your goal is not synchronization, but is to simply initialize the state, then it is proper to use props within `getInitialState`. Just be sure to make your intentions clear, for example prefix the prop with `initial`.

```
getDefaultProps: function () {
  return {
    initialValue: 'some-default-value'
  };
},
getInitialState: function () {
  return {
```

```
    value: this.props.initialValue
  };
},
render: function () {
  return <div>{this.props.value}</div>
}
```

React's lifecycle methods provide well-designed hooks into the life of your components. As state machines, each component is designed to output stable, predictable markup throughout it's life.

However no component lives in isolation. As parent components push props into their children, and as those children render their own child components, you must carefully consider how your data flows through the app. How much does each child *really* need to know about? Who owns the app state? This is the subject of our next chapter: Data Flow.

Chapter 4. Data Flow

In React, data flows in one direction only: From the parent to the child. This makes components really simple and predictable. They take props from the parent and render. If a prop is changed at the top level component, React will propagate that change all the way down the component tree and re-render all the components that used that property.

Components can also have internal state, which should only be modified within the component.

React components are inherently simple and you can consider them as a function that take `props` and`state` and output HTML.

In this chapter we look will look at:

- What "props" are
- What "state" is
- When to use "props" and when to use "state"

Props

“props”, short for “properties" are passed to a component and can hold any data you’d like.

You can set props on a component during instantiation:

```
var surveys = [{ title: 'Superheroes' }];
<ListSurveys surveys={surveys}/>
```

or via the setProps method on a component instance:

```
var surveys = [{ title: 'Superheroes' }];
var listSurveys = React.renderComponent(
  <ListSurveys/>,
  document.querySelector(‘body’)
);
listSurveys.setProps({ surveys: surveys });
```

You can access props via `this.props`, but you should never write to props that way. A component should never modify its own props.When used with JSX, props can be set as a string:

```
<a href='/surveys/add'>Add survey</a>
```

It can also be set with the `{}` syntax, which injects JavaScript and allows you to

pass in variables of any type:

```
<a href={'/surveys/' + survey.id}>{survey.title}</a>
```

It’s possible with the `transferPropsTo` method to pass all props received to a child

component, but it is advised against using this function as it encourages tight coupling. Instead explicitly copy over the props from the parent to the child component:

```
var ListSurveys = React.createClass({
  render: function () {
```

```
    return this.transferPropsTo(<SurveyTable/>);
  }
});

// vs the explicit
var ListSurveys = React.createClass({
  render: function () {
    return <SurveyTable surveys={this.props.surveys}/>;
  }
});
```

Props are useful for event handlers as well:

```
var SaveButton = React.createClass({
  render: function () {
    return (
      <a className='button save'
onClick={this.handleClick}>Save</a>
    );
  },
  handleClick: function () {
    // ...
  }
});
```

Here we are passing the onClick prop to the anchor tag with the value of the `handleClick` function.

PropTypes

React provides a way to validate your props, through a config object defined on your component:

```
var SurveyTableRow = React.createClass({
  propTypes: {
    survey: React.PropTypes.shape({
      id: React.PropTypes.number.isRequired
    }).isRequired,
    onClick: React.PropTypes.func
  },
  // ...
});
```

If the requirements of the propTypes are not met when the component is instantiated, a`console.warn` will be logged.

For optional props simple leave the `.isRequired` off.

You are not required to use propTypes in your application, but they provide a good way to describe the API of your component.

getDefaultProps

Create the `getDefaultProps` function on your component to provide a default set of properties. This should only be done for props that aren't required.

```
var SurveyTable = React.createClass({
  getDefaultProps: function () {
    return {
      surveys: []
    };
  }
  // ...
});
```

It's important to note that `getDefaultProps` is not called during component instantiation but as soon as `React.createClass` is called to cache the value. This means you can't use any instance specific data in the `getDefaultProps` method.

state

Each component in a React can house state. State differs from props in that it is internal to the component.

State is useful for deciding a view state on an element. Consider the `<AnswerRadioInput/>` component used for collecting survey answers:

```
var AnswerRadioInput = React.createClass({
  getInitialState: function () {
    return {
      checked: !!this.props.checked
    };
  },

  handleChanged: function (e) {
    var checked = e.target.checked;
    this.setState({checked: checked});
  },

  render: function () {
    return (
      <div className="radio">
        <label>
          <input type="radio"
            checked={this.state.checked}
            onChange={this.handleChanged} />
          {this.props.label}
```

```
          </label>
        </div>
      );
    }
});
```

Here we use state to track if the checkbox should be checked or not.

State is altered via the `setState` method and can have a default value provided the `getInitialState`method as shown above. When ever `setState` is called within the component the render method is called and if there were changes to the output of the render function, the DOM will be updated and finally the user will see a change in their browser.

State will always make your component more complex, if you isolate your state to certain components you application becomes easier to debug and reason about.

What belongs in state and what belongs in props

Don't store computed values or components in state, instead focus on simple data that is directly required for the component to function, like our checked state from earlier. Without it we couldn't check and uncheck the checkbox. This could also be a boolean for showing the options of a dropdown, or the values of an input field.

Try not to duplicate prop data into state. When possible consider props the source of truth.

Chapter 5. Event Handling

When it comes to user interfaces, presenting is only half the equation. The other part is responding to user input, which in JavaScript means handling user-generated events.

React's approach to event handling is to attach event handlers to components, and then to update those components' internal states when the handlers fire. Updating the component's internal state causes it to re-render, so all that is required for the user interface to reflect the consequences of the event is to read from the internal state in the component's render function.

Although it is common to update state based solely on the type of the event in question, it is often necessary to use additional information from the event in order to determine how to update the state. In such a case, the Event Object that is passed to the handler will provide extra information about the event, which the handler can then use to update the internal state.

Between these techniques and React's highly efficient rendering, it becomes easy to respond to user's inputs and to update the user interface based on the consequences of those inputs.

Attaching Event Handlers

At a basic level, React handles the same events you see in regular JavaScript: `MouseEvents` for click handlers, `Change` events when form elements change, and so on. The events have the same names you would see in regular JavaScript, and will trigger under the same circumstances.

React's syntax for attaching event handlers closely resembles HTML's. For example, in the SurveyBuilder, the following code attaches an `onClick` handler to the Save button.

```
<button className="btn btn-save"
onClick={this.handleSaveClicked}>Save</button>
```

When the user clicks the button, the button's `handleSaveClicked` method will run. That method will contain the logic necessary to resolve the Save action.

If you are not using JSX, you instead specify the handler as one of the fields in the options object. For example:

```
React.DOM.button({className: "btn btn-save", onClick:
this.handleSaveClicked}, "Save");
```

React has first-class support for handling a wide variety of event types, which it lists in the Event System page of its documentation.

Most of these work without any extra effort, but touch events must be manually enabled by calling this:

```
React.initializeTouchEvents(true);
```

Events and State

Suppose you want a component to change based on user input, like in the survey editor, where you want to let users drag survey questions in from a menu of question types.

Start with a render function which registers some event handlers based on the HTML5 Drag and Drop API.

```
var SurveyEditor = React.createClass({
  render: function () {
    return (
      <div className='survey-editor'>
        <div className='row'>
          <aside className='sidebar col-md-3'>
            <h2>Modules</h2>
            <DraggableQuestions />
          </aside>

          <div className='survey-canvas col-md-9'>
            <div
              className={'drop-zone well well-drop-zone'}
              onDragOver={this.handleDragOver}
              onDragEnter={this.handleDragEnter}
              onDragLeave={this.handleDragLeave}
              onDrop={this.handleDrop}
            >
              Drag and drop a module from the left
            </div>
          </div>
        </div>
      </div>
    );
  });
```

The `DraggableQuestions` component will render the menu of question types, and the handler methods will take care of resolving drag and drop actions.

Rendering Based on State

One thing the handler methods will need to do is to expand some running list of the questions added so far. To do this, you will need to make use of the internal `state` object that every React component contains. It begins as null by default, but can be initialized to something sane with a component's `getInitialState()` method, like so:

```
getInitialState: function () {
  return {
    dropZoneEntered: false,
    title: '',
    introduction: '',
    questions: []
  };
}
```

This establishes a sane baseline for the state: a blank title, a blank introduction, an empty list of questions, and a value of `false` for `dropZoneEntered` - indicating that the user is not currently dragging anything over the drop zone.

From here, you can read from `this.state` in the `render` method in order to display the current values of each of these to the user.

```
render: function () {
  var questions = this.state.questions;

  var dropZoneEntered = '';
  if (this.state.dropZoneEntered) {
    dropZoneEntered = 'drag-enter';
  }

  return (
    <div className='survey-editor'>
      <div className='row'>
        <aside className='sidebar col-md-3'>
          <h2>Modules</h2>
          <DraggableQuestions />
        </aside>
```

```
            <div className='survey-canvas col-md-9'>
              <SurveyForm
                title={this.state.title}
                introduction={this.state.introduction}
                onChange={this.handleFormChange}
              />

              <Divider>Questions</Divider>
              <ReactCSSTransitionGroup transitionName='question'>
                {questions}
              </ReactCSSTransitionGroup>

              <div
                className={'drop-zone well well-drop-zone ' +
dropZoneEntered}
                onDragOver={this.handleDragOver}
                onDragEnter={this.handleDragEnter}
                onDragLeave={this.handleDragLeave}
                onDrop={this.handleDrop}
              >
                Drag and drop a module from the left
              </div>

              <div className='actions'>
                <button className="btn btn-save"
onClick={this.handleSaveClicked}>Save</button>
              </div>
            </div>
          </div>
        </div>
      );
    }
```

As with `this.props`, the render function can change as little or as much as you like depending on the values of `this.state`. It can render the same elements but with slightly different attributes, or a completely different set of elements altogether. Either works just as well.

Updating State

Since updating a component's internal `state` causes the component to re-render, the next thing to do is to update that state in the drag handler methods.
Then `render` will run again, it will read from the current value of `this.state` to

display the title, introduction, and questions, and the user will see that everything has been updated properly.

There are two ways to update a component's state: the component's `setState` method and its`replaceState` method. `replaceState` overwrites the entire state object with an entirely new state object, which is rarely what you want. Much more often you will want to use `setState`, which simply merges the object you give it into the existing state object.

For example, suppose you have the following for your current state:

```
getInitialState: function () {
  return {
    dropZoneEntered: false,
    title: 'Fantastic Survey',
    introduction: 'This survey is fantastic!',
    questions: []
  };
}
```

In this case, calling `this.setState({title: "Fantastic Survey 2.0"})` only affects the value of`this.state.title`, leaving `this.state.dropZoneEntered`, `this.state.introduction`, and`this.state.questions` unaffected.

Calling `this.replaceState({title: "Fantastic Survey 2.0"})` would instead replace the entire state object with the new object `{title: "Fantastic Survey 2.0"}`, erasing `this.state.dropZoneEntered`,`this.state.introduction`, and `this.state.questions` altogether. This would likely break the render function, which would expect `this.state.questions` to be an array instead of `undefined`.

Using `this.setState`, you can now implement the handler methods from earlier.

```
handleFormChange: function (formData) {
  this.setState(formData);
},

handleDragOver: function (ev) {
  // This allows handleDropZoneDrop to be called
```

```
    // https://code.google.com/p/chromium/issues/detail?id=168387
    ev.preventDefault();
  },

  handleDragEnter: function () {
    this.setState({dropZoneEntered: true});
  },

  handleDragLeave: function () {
    this.setState({dropZoneEntered: false});
  },

  handleDrop: function (ev) {
    var questionType = ev.dataTransfer.getData('questionType');
    var questions = this.state.questions;
    questions = questions.concat({ type: questionType });

    this.setState({
      questions: questions,
      dropZoneEntered: false
    });
  }
```

It's important never to alter the `state` object by any other means than calling `setState` or`replaceState`. Doing something like `this.state.saveInProgress = true` is generally a bad idea, as it will not inform React that it might need to re-render, and might lead to surprising results the next time you call `setState`.

Event Objects

Many handlers simply need to fire in order to serve their purposes, but sometimes you need more information about the user's input.

Take a look at the `AnswerEssayQuestion` class from the SurveyBuilder.

```
var AnswerEssayQuestion = React.createClass({
  handleComplete: function(event) {
    this.callMethodOnProps('onCompleted', event.target.value);
  },
  render: function() {
    return (
      <div className="form-group">
```

```
            <label  className="survey-item-
label">{this.props.label}</label>
            <div  className="survey-item-content">
              <textarea  className="form-control"  rows="3"
onBlur={this.handleComplete}/>
            </div>
          </div>
        );
      }
    });
```

React's event handler functions are always passed an event object, much in the same way that a vanilla JavaScript event listener would be. Here, the `handleComplete` method takes an event object, from which it extracts the current value of the `textarea` by accessing `event.target.value`. Using `event.target.value` in an event handler like this is a common way to get the value from a form input, especially in an `onChange` handler.

Rather than passing the original Event object from the browser directly to the handler, React wraps the original event in a `SyntheticEvent` instance. A `SyntheticEvent` is designed to look and function the same way as the original event the browser created, except with certain cross-browser inconsistencies smoothed over. You should be able to use the `SyntheticEvent` the same way you would a normal event, but in case you need the original event that the browser sent, you can access it via the `SyntheticEvent`'s `nativeEvent` field.

Summary

The steps to reflect changes from user input in the user interface are simple:

1. Attach an event handler to a React component
2. In that event handler, update the component's internal state. Updating the component's state will cause it to re-render.
3. Modify the component's `render` function to incorporate `this.state` as appropriate.

So far you have used a single component to respond to user interactions. Next you will learn how to compose multiple components together, to create an interface that is more than the sum of its parts.

Chapter 6. Composing Components

In traditional HTML the basic building block of each page is an element. In React you assemble a page from components. You can think of a React component as an HTML element with all the expressive power of JavaScript mixed in. In fact with React the *only* thing you do is build components, just like an HTML document is build with only elements.

Since the entirety of a React app is build using components, this whole book can be described as a book about React components. Therefore in this chapter we won't cover *everything* to do with components. Rather you will be introduced to one specific aspect — their composability.

A component is basically a JavaScript function that takes in props and state as its arguments and outputs rendered HTML. They are typically designed to represent and express a piece of data within your app so you can think of a React component as an extension of HTML.

Extending HTML

React + JSX are powerful and expressive tools allowing us to create custom elements that can be expressed using an HTML-like syntax. However they go far beyond plain HTML, allowing us to control their behavior throughout their lifetime. This all starts with the `React.createClass` method.

React favors composition over inheritance, which means we combine small, simple components and data objects into larger, more complex components. If you're familiar with other MVC or object-oriented tools you might expect to find a `React.extendClass` method, as I did. However, just as you don't extend html dom nodes when building a web page, you don't extend React components. Instead you compose them.

React embraces composability, allowing you to mix-and-match various child components into an intricate and powerful new component. To demonstrate this let's consider how a user would answer a survey question

. Specifically let's look at the `AnswerMultipleChoiceQuestion` component responsible for rendering a multiple choice question and capturing the user's answer.

Obviously a survey will be built around basic html form elements. You will be crafting our survey answer components by wrapping the default html input elements and customizing their behavior.

Composition By Example

Let's consider the component representing a multiple-choice question. This has a few requirements.

- Take a list of choices as input
- Render the choices to the user
- Only allow the user to select a single choice

We know html provides us some some basic elements to help us, namely the "radio" type inputs and input groups. Thinking of it from the top-down our component hierarchy will look something like this:

```
MultipleChoice → RadioInput → Input (type="radio")
```

You can think of those arrows as representing the phrase *has a*. The MulitpleChoice component *has a*RadioInput. The RadioInput *has an* input. This is an identifying trait of the composition pattern.

Assemble the HTML

Let's start by assembling the components from the bottom-up. React pre-defines the `input` component for us in the `React.DOM.input` namespace. Therefore the first thing you'll do is wrap it in a`RadioInput` component. This component will be responsible for customizing the generic `input`, narrowing it's scope to behave like a radio button. Within the accompanying sample app it is named`AnswerRadioInput`.

First create the scaffolding, which will include the required render method and the basic markup to describe the desired output. You can already begin to see the composition pattern as the component becomes a specialized type of input.

```
var AnswerRadioInput = React.createClass({
render: function () {
     return (
       <div className="radio">
         <label>
           <input type="radio" />
           Label Text
         </label>
       </div>
     );
   }
});
```

Add Dynamic Properties

Nothing about our `input` is dynamic yet, so next you need to define the properties the parent must pass into the radio input.

- What value, or choice, does this input represent? (required)
- What text do we use to describe it? (required)
- What is the input's name? (required)
- We'll might want to customize the id.
- We might want to override the default value.

Given this list you can now define the property types for our custom input. Add these to the`propTypes` hash of the class definition.

```
var AnswerRadioInput = React.createClass({
  propTypes: {
    id: React.PropTypes.string,
    name: React.PropTypes.string.isRequired,
    label: React.PropTypes.string.isRequired,
    value: React.PropTypes.string.isRequired,
    checked: React.PropTypes.bool
  },
  ...
});
```

For each optional property you need to define the default value. Add these to the`getDefaultProps` method. These values will be applied to each new instance when the parent component does not provide their value.

Since this method is only called once for the class, not for each instance, we cannot provide the id here — it must be unique for each instance. That will be solved using state below.

```
var AnswerRadioInput = React.createClass({
  propTypes: {...},
  getDefaultProps: function () {
    return {
      id: null,
      checked: false
    };
  },
  ...
});
```

Track State

Our component needs to keep track of data that changes over time. Specifically, the `id` will be unique for each instance and the user can update the `checked` value at any time. Therefore we define the initial state.

```
var AnswerRadioInput = React.createClass({
  propTypes: {...},
  getDefaultProps: function () {...},
  getInitialState: function () {
    var id = this.props.id ? this.props.id : uniqueId('radio-');
    return {
      checked: !!this.props.checked,
      id: id,
       name: id
    };
  },
  ...
});
```

Now you can update the rendered markup to access the new dynamic state and props.

```
var AnswerRadioInput = React.createClass({
  propTypes: {...},
  getDefaultProps: function () {...},
  getInitialState: function () {...},
  render: function () {
    return (
```

```
        <div className="radio">
          <label htlmFor={this.props.id}>
            <input type="radio"
              name={this.props.name}
              id={this.props.id}
              value={this.props.value}
              checked={this.state.checked} />
            {this.props.label}
          </label>
        </div>
      );
    }
});
```

Integrate into a Parent Component

At this point you have enough of a component to use it within a parent so you're ready to build up the next layer, the `AnswerMultipleChoiceQuestion`. The primary responsibility here is to display a list of choices for the user to choose from. Following the pattern introduced above, let's lay down the basic html and the default props for this component.

```
var AnswerMultipleChoiceQuestion = React.createClass({
  propTypes: {
    value: React.PropTypes.string,
    choices: React.PropTypes.array.isRequired,
    onCompleted: React.PropTypes.func.isRequired
  },
  getInitialState: function() {
    return {
      id: uniqueId('multiple-choice-'),
      value: this.props.value
    };
  },
  render: function() {
    return (
      <div className="form-group">
        <label className="survey-item-label"
htmlFor={this.state.id}>{this.props.label}</label>
        <div className="survey-item-content">
            <AnswerRadioInput ... />
            ...
            <AnswerRadioInput ... />
        </div>
      </div>
```

```
    );
  }
});
```

In order to generate the list of child radio input components we must map over the choices array, transforming each into a component. This is easily handled in a helper function as demonstrated here.

```
var AnswerMultipleChoiceQuestion = React.createClass({
  ...
  renderChoices: function() {
    return this.props.choices.map(function(choice, i) {
      return AnswerRadioInput({
        id: "choice-" + i,
        name: this.state.id,
        label: choice,
        value: choice,
        checked: this.state.value === choice
      });
    }.bind(this));
  },
  render: function() {
    return (
      <div className="form-group">
        <label className="survey-item-label"
htmlFor={this.state.id}>{this.props.label}</label>
        <div className="survey-item-content">
            {this.renderChoices()}
        </div>
      </div>
    );
  }
});
```

Now the composability of React is becoming more clear. You started with a generic input, customized it into a radio input, and finally wrapped it into a multiple-choice component — a highly refined and specific version of a form control. Now rendering a list of choices is as simple as this:

```
<AnswerMultipleChoiceQuestion choices={arrayOfChoices} ... />
```

The astute reader has probably noticed there is a missing piece — our radio inputs have no way of communicating changes to their parent component. You need to wire up the `AnswerRadioInput` children so the parent is aware of their changes and

can process them into a proper survey-result data payload. Which brings us to the parent/child relationship.

Parent / Child Relationship

At this point you should be able to render a form to the screen, but notice that you have yet to give the components the ability to share user changes.
The `AnswerRadioInput` component does not yet have the ability to communicate with its parent.

The easiest way for a child to communicate with its parent is via props. The parent needs to pass in a callback via the props, which the child calls it when needed.

First you need to define what `AnswerMultipleChoiceQuestion` will do with the changes from it's children. Add a `handleChanged` method and pass it into each `AnswerRadioInput`.

```
var AnswerMultipleChoiceQuestion = React.createClass({
  ...
  handleChanged: function(value) {
    this.setState({value: value});
    this.props.onCompleted(value);
  },
  renderChoices: function() {
    return this.props.choices.map(function(choice, i) {
      return AnswerRadioInput({
        ...
        onChanged: this.handleChanged
      });
    }.bind(this));
  },
  ...
});
```

Now each radio input can watch for user changes, passing the value up to the parent. This requires wiring up an event handler to the input's `onChange` event.

```
var AnswerRadioInput = React.createClass({
  propTypes: {
    ...
    onChanged: React.PropTypes.func.isRequired
  },
```

```
  handleChanged: function (e) {
    var checked = e.target.checked;
    this.setState({checked: checked});
    if(checked) {
      this.props.onChanged(this.props.value);
    }
  },
  render: function () {
    return (
      <div className="radio">
        <label htmlFor={this.state.id}>
          <input type="radio"
            ...
            onChange={this.handleChanged} />
          {this.props.label}
        </label>
      </div>
    );
  }
});
```

Wrap Up

Now you've seen how React uses the pattern of composability, allowing you to wrap html elements or custom components and customize their behavior for your needs. As you compose components they become more specific and semantically meaningful. Thus React takes generic inputs,

```
<input type="radio" ... />
```

and transforms them into something a bit more meaningful,

```
<AnswerRadioInput ... />
```

finally ending up with a single component to transform an array of data into a useful UI for users to interact with.

```
<AnswerMultipleChoiceQuestion choices={arrayOfChoices} ...
/>
```

Composition is just one way React offers to customize and specialize your components. Mixins offer another approach, allowing you to define methods that

can be shared across many components. Next we show you how to define a mixin and how they can be used to share common code.

Chapter 7. Mixins

What are Mixins?

On the homepage of React we have this example of a timer component:

```
var Timer = React.createClass({
  getInitialState: function() {
    return {secondsElapsed: 0};
  },
  tick: function() {
    this.setState({secondsElapsed: this.state.secondsElapsed
+ 1});
  },
  componentDidMount: function() {
    this.interval = setInterval(this.tick, 1000);
  },
  componentWillUnmount: function() {
    clearInterval(this.interval);
  },
  render: function() {
    return (
      <div>Seconds Elapsed:
{this.state.secondsElapsed}</div>
    );
  }
});
```

This is good, but we may have multiple components using timers; implementing this exact code. This is where mixins come in. We want to end up with a Timer component that looks like this:

```
var Timer = React.createClass({
  mixins: [IntervalMixin(1000)],
  getInitialState: function() {
    return {secondsElapsed: 0};
  },
  onTick: function() {
    this.setState({secondsElapsed: this.state.secondsElapsed
+ 1});
  },
  render: function() {
    return (
      <div>Seconds Elapsed:
{this.state.secondsElapsed}</div>
    );
  }
});
```

Mixins are fairly simple. They're objects which are mixed into our component class. React goes a bit further to prevent silently overwriting functions, and allowing multiple mixins which in other systems would conflict. For example:

```
React.createClass({
  mixins: [{
    getInitialState: function(){ return {a: 1} }
  }],
  getInitialState: function(){ return {b: 2} }
});
```

You have getInitialState defined in both the mixin and component class. The resulting initial state is {a: 1, b: 2}. If a key exists in both of these, an error will be thrown to alert you to the problem.

The lifecycle methods starting with “component”, e.g. “componentDidMount”, are called in the order of the mixin array, and finally the componentDidMount on the component class itself, if it exists.

So back to our original example, we need to implement IntervalMixin. Sometimes a simple object suffices, but other times we need a function which returns that object. In this case, you want to specify the interval.

```
var IntervalMixin = function(interval) {
  return {
    componentDidMount: function() {
      this.__interval = setInterval(this.onTick, interval);
    },
    componentWillUnmount: function() {
      clearInterval(this.__interval);
    }
  };
};
```

This is pretty good, but it has some limitations. You can’t have multiple intervals, and uoi can’t choose the function that handles the interval, and we can’t terminate the interval manually without using the internal __interval property. To solve this, our mixin can have a public api.

Here’s an example where we simply want to show the number of seconds since January 1st 2014. This mixin is a bit more code, but more flexible and powerful.

```
var IntervalMixin = {
  setInterval: function(callback, interval){
    var token = setInterval(callback, interval);
    this.__intervals.push(token);
    return token;
  },
  componentDidMount: function() {
    this.__intervals = [];
```

```
  },
  componentWillUnmount: function() {
    this.__intervals.map(clearInterval);
  }
};

var Since2014 = React.createClass({
  mixins: [IntervalMixin],
  componentDidMount: function(){
    this.setInterval(this.forceUpdate.bind(this), 1000);
  },
  render: function() {
    var from = Number(new Date(2014, 0, 1));
    var to = Date.now();
    return (
      <div>{Math.round((to-from) / 1000)}</div>
    );
  }
});
```

Mixins are one of the most powerful tools for eliminating code repetition, and keeping your components focused on what makes them special. They allow you to use powerful abstractions, and some problems can't be solved elegantly without them. Some noteworthy examples:

- a mixin which listens for events, and applies them to state (e.g. flux store mixin)
- an upload mixin which handles XHR uploads, and applies the status and progress of uploads to state
- 'layers' mixin, which facilitates rendering children to the end of </body> (e.g. for modals)

Testing

Now that you've learned how a mixin can be used in our component, an interesting question becomes -- how do we unit test it? Turns out there are three options for testing a mixin:

1. Test the mixin object directly
2. Test the mixin which is included in a fake component
3. Test the mixin which is included in the real component using a shared behavior spec

Testing the mixin directly

To test the mixin directly, you simply call the functions on the mixin object directly and validate the behavior that way. This approach usually results in very fine grained tests which will need stubs for any React.js methods called from within a function. Let's give this a try by starting with the simplest function in our mixin, `componentDidMount`:

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


var IntervalMixin = require
('../../../client/testing_examples/interval_mixin');


describe("IntervalMixin", function(){
  describe("testing the mixin directly", function(){
    var subject;


    beforeEach(function(){
      // WARNING: DON'T DO THIS!! IT WILL CAUSE ISSUES WHICH
WE WILL DISCUSS BELOW
      subject = IntervalMixin;
    });


    describe("componentDidMount", function(){
```

```
      it("should set an empty array called __intervals on
the instance", function(){
        expect(subject.__intervals).toBeUndefined();

        subject.componentDidMount();

        expect(subject.__intervals).toEqual([]);
      });
    });
  });
});
```

When we run this tests it passes fine. You might notice that you are using the `IntervalMixin` as your subject, this is not a good idea because the `componentDidMount` function is setting a variable on `this`,`subject`, which in this case will be the `IntervalMixin` object. So when the next test runs, the `IntervalMixin`object has been polluted and will cause weird failures. To see this in action, modify the spec to this:

```
...

describe("IntervalMixin", function(){
  describe("testing the mixin directly", function(){
    var subject;

    beforeEach(function(){
      // WARNING: DON'T DO THIS!! IT WILL CAUSE ISSUES WHICH
WE WILL DISCUSS BELOW
      subject = IntervalMixin;
    });

    describe("componentDidMount", function(){
```

```
      it("should set an empty array called __intervals on
the instance", function(){
        expect(subject.__intervals).toBeUndefined();

        subject.componentDidMount();

        expect(subject.__intervals).toEqual([]);
      });

      it("should set an empty array called __intervals on
the instance (testing for test pollution)", function(){
        expect(subject.__intervals).toBeUndefined();

        subject.componentDidMount();

        expect(subject.__intervals).toEqual([]);
      });
    });
  });
});
```

And you will see the second test fails on the `expect(subject.__intervals).toBeUndefined();` line, because the test before had executed:

```
Chrome 37.0.2062 (Mac OS X 10.8.2) IntervalMixin testing the
mixin directly componentDidMount should set an empty array
called __intervals on the instance (testing for test
pollution) FAILED Expected [  ] to be undefined. Error:
Expected [  ] to be undefined.    at null.<anonymous>
(/var/folders/xz/xk1180q53ml23q_dqz91dvh00000gn/T/770d14d1a2
5ddccd00c6ee3578fac4d283ef1c5d.browserify:30890:37 <-
/Users/tom/workspace/bleeding-edge-sample-
app/test/client/fundamentals/interval_mixin_spec.js:28:0)
```

To fix this, we need to use a fresh copy of our mixin each time, so we'll switch our `beforeEach` to use`Object.create` and both tests will start passing:

```
...
    beforeEach(function(){
      subject = Object.create(IntervalMixin);
    });
...
```

Now that you've got a test passing for `componentDidMount`, lets work on testing `setInterval`. The`setInterval` function has three responsibilities:

1. be a pass-through to the real setInterval function
2. save off the interval id number into an array (so it can be cleared out later)
3. return the interval id

So those tests would look something like this:

```
...
    describe("setInterval", function(){
      var fakeIntervalId;
      beforeEach(function(){
        fakeIntervalId = 555;
        spyOn(window,
"setInterval").andReturn(fakeIntervalId);
        // NOTE: how we have to call componentDidMount
before we call setInterval, so that
        //   this.__intervals is defined.  This is a
drawback of calling functions directly on
        //   a mixin object
        subject.componentDidMount();
      });
```

```
      it("should call window.setInterval with the callback
and the interval", function(){
        expect(window.setInterval.callCount).toBe(0);

        subject.setInterval(function(){}, 500);

        expect(window.setInterval.callCount).toBe(1);
      });

      it("should store the setInterval id in the
this.__intervals array", function(){
        subject.setInterval(function(){}, 500);

expect(subject.__intervals).toEqual([fakeIntervalId]);
      });

      it("should return the setInterval id", function(){
        var returnValue = subject.setInterval(function(){},
500);

        expect(returnValue).toBe(fakeIntervalId);
      });
    });

...
```

STUB OUT REACT.JS

Notice how the above tests don't need any React.js specific functionality, so they will be vanilla jasmine tests. If your mixin functions start calling methods which are provided by React.js (like`this.setState({})`), it is usually recommended to `spyOn(subject, "setState")` to mock

out the React.js specific functions which are used. This allows you to keep your mixin test isolated to just the mixin object.

// TODO: tests for componentWillUnmount and explanation for it

You might have noticed that testing the mixin directly results in tests which are very fine grained. Sometimes this can be helpful when the behavior starts to grow in complexity, but sometimes it can lead to a path where you are testing the implementation and not the functionality. It also requires you to call any functions on the mixin in a specific order to mimic React.js lifecycle callbacks, like how we had to call `subject.componentDidMount();` in the `setInterval` test. The next section will show a way to test a mixin without these drawbacks.

Test the mixin which is included in a fake component

To test a mixin with a fake component, i.e. a "faux" component, you will need to define a React component in your test suite. The fact the component is defined in the test suite makes it very clear that this component exists for the sole purpose of testing and can't be used in the production application. Below is an example of our faux component. Notice how the functionality is simple so that we can keep the tests simple which will help keep the intent of each tests clear. The only bit of "cruft" is the `render`function which is required by React.

```
  describe("testing the mixin via a faux component",
function(){
    var FauxComponent;


    beforeEach(function(){


      // Notice how the faux component is defined in the
jasmine spec file.  This is
      //  intentional.  This expresses the intent that this
react component exists
      //  for the sole purpose of testing this mixin.
      FauxComponent = React.createClass({
```

```
        mixins: [IntervalMixin],
        render: function(){
          return (<div>Faux components are all the
rage!</div>);
        },
        myFakeMethod: function(){
          this.setInterval(function(){}, 500);
        }
      });
    });


  });
```

Now that you've got your faux component, let's write some tests:

```
...
    describe("setInterval", function(){
      var subject;
      beforeEach(function(){
        spyOn(window, "setInterval");
        subject =
TestUtils.renderIntoDocument(<FauxComponent />);
      });


      it("should call window.setInterval with the callback
and the interval", function(){
        expect(window.setInterval.callCount).toBe(0);


        subject.myFakeMethod();


        expect(window.setInterval.callCount).toBe(1);
      });
```

```
    });

    describe("unmounting", function(){
      var subject;

      beforeEach(function(){
        spyOn(window, "setInterval").andReturn(555);
        spyOn(window, "clearInterval");
        subject =
TestUtils.renderIntoDocument(<FauxComponent />);
        subject.myFakeMethod();
      });

      it("should clear any setTimeout's", function(){
        expect(window.clearInterval.callCount).toBe(0);

React.unmountComponentAtNode(subject.getDOMNode().parentNode
);
        expect(window.clearInterval.callCount).toBe(1);
      });
    });


...
```

The first major difference with these specs, which render a faux component, versus the specs which directly test the mixin, is that you need to first render said faux component and then start operating against it to make your test assertions. While this difference is glaring, there is a more subtle difference which can have a large effect on your tests: “Faux” has tests for `setInterval` and `unmounting`, while “direct” has tests for `setInterval`, `componentDidMount`, and `componentWillUnmount`.

That doesn’t sound like such a big deal, so who cares? For the answer for this, look at the`componentDidMount` function. Setting up `this.__intervals` provides no value in of itself, the value is only used with the other functions. In the “direct mixin” spec,

we are testing the implementation of the function by asserting on `this.__intervals` which is an implementation detail, not functionality. In the "faux component" tests, we don't need to test the implementation of `componentDidMount` because it is implicitly tested in the `setInterval` test when the component is rendered into the document.

Why is it important one describe block is called "unmounting" versus "componentWillUnmount"? Because we don't actually care how the `clearInterval` code is called, we just care that it is called when the component is unmounted. So instead of calling `subject.componentWillUnmount` in the "direct" spec, we just unmount the component and let React.js call the appropriate callbacks in the correct order.

WHICH ONE TO CHOOSE?

In the two methods we have gone over so far, "direct" and "faux", one is not better than the other. It depends on the complexity and behavior of your mixin for which one is better. One recommendation is to start with writing the "direct" spec, because it's the most focused and then if it becomes a pain to write, then switch it to a faux spec (or do both). Don't be afraid to just pick one option and let your tests tell you that you are wrong.

Shared behavior spec

The "faux" and "direct" approaches didn't involve any of the components from your real application, only the mixin itself. The last approach will test the mixin via the real world components which will actually use that mixin -- this approach is called the "shared behavior" spec. The first big difference with this approach is that our tests are no longer located in the `interval_mixin_spec.js`, because the specs will be run by the `since_2014_spec.js`. Let's start there:

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


var Since214 = require
('../../../client/testing_examples/since_2014');
```

```
describe("Since2014", function(){
});
```

We've got our boilerplate for a `Since2014` spec, so now let's add our "shared example spec":

```
...
describe("Since2014", function(){
  describe("shared examples", function(){
    IntervalMixinSharedExamples();
  });
});
...
```

If we run these tests they will fail because `IntervalMixinSharedExamples is not defined`, so let's define that function:

```
...
var Since2014 = require
('../../../client/testing_examples/since_2014');
var IntervalMixinSharedExamples =
require('../shared_examples/interval_mixin_shared_examples')
;
...
```

and then in the interval_mixin_shared_examples.js file, put the boilerplate for a shared example spec:

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


var SetIntervalSharedExamples = function(attributes){
```

```
  var componentClass;

  beforeEach(function(){
    componentClass = attributes.componentClass;
  });

  describe("SetIntervalSharedExamples", function(){
  });
};

module.exports = SetIntervalSharedExamples;
```

If you look at this code carefully you will notice that `SetIntervalSharedExamples` is just a function which has a bunch of jasmine tests inside of it. So the `Since2014` specs call the `IntervalMixinSharedExamples();` function and then those tests will run.

Another important part of the "shared behavior" boilerplate is the `attributes.componentClass` part. This allows you to use dependency injection to pass in the component under test, in this example `Since2014`:

```
...

describe("Since214", function(){
  describe("shared examples", function(){
    IntervalMixinSharedExamples({componentClass:
Since2014});
  });
});
```

Now let's write the `SetIntervalSharedExamples` specs:

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

```
var SetIntervalSharedExamples = function(attributes){

  var componentClass;

  beforeEach(function(){
    componentClass = attributes.componentClass;
  });

  describe("SetIntervalSharedExamples", function(){
    describe("setInterval", function(){

      var subject, fakeFunction;

      beforeEach(function(){
        spyOn(window, "setInterval");
        subject =
TestUtils.renderIntoDocument(<componentClass />);
        fakeFunction = function(){};
      });

      it("should call window.setInterval with the callback
and the interval", function(){

expect(window.setInterval).not.toHaveBeenCalledWith(fakeFunc
tion, jasmine.any(Number));

        subject.setInterval(fakeFunction, 100);


expect(window.setInterval).toHaveBeenCalledWith(fakeFunction
, jasmine.any(Number));
```

```
    });
  });

  describe("unmounting", function(){
    var subject, fakeFunction;

    beforeEach(function(){
      fakeFunction = function(){};

      spyOn(window,
"setInterval").andCallFake(function(func, interval){
        // we want to make sure we are only asserting
against the setInterval calls
        //  which come from this spec (not the ones which
come from our components
        //  which use this mixin).  So we force the
setInterval calls for our "fakeFunction"
        //  to return a different id number.
        if(func === fakeFunction){
          return 444;
        } else {
          return 555;
        }
      });
      spyOn(window, "clearInterval");
      subject =
TestUtils.renderIntoDocument(<componentClass />);

      subject.setInterval(fakeFunction, 100);
    });
```

```
      it("should clear any setTimeout's", function(){

expect(window.clearInterval).not.toHaveBeenCalledWith(444);



React.unmountComponentAtNode(subject.getDOMNode().parentNode
);



expect(window.clearInterval).toHaveBeenCalledWith(444);
      });
    });
  });
};

module.exports = SetIntervalSharedExamples;
```

SHARED SPECS

One important distinction to keep in mind, is that behavior which is specific to `Since2014` should stay in the `since_2014_spec.js`, but functionality which is made avaialbe to `Since2014` by the `IntervalMixin` mixin should be tested in the `interval_mixin_shared_examples.js` spec file.

If you look at the specs for the shared behavior you will notice they are similar in nature to the specs for the faux component, but they are slightly more complex. In the "faux" example, we can do this in the`unmounting` spec:

```
spyOn(window, "setInterval").andReturn(555);
```

But in the "shared example" example, we have todo this in the `unmounting` spec:

```
       spyOn(window, "setInterval").andCallFake(function(func,
interval){
         if(func === fakeFunction){
           return 444;
         } else {
           return 555;
         }
       });
```

This extra complexity is due to fact that the `Since2014` code will be

calling `setInterval` in addition the to calls to `setInterval` in the shared behavior spec, so we have to be able to distinguish between them. Otherwise our shared behavior spec might not be testing the mixin correctly. This means that a “shared behavior” spec will have more noise and complexity than a comparable “faux spec”. This complexity can be worth it in certain cases: - If you mixin requires the React component to have certain functions/behavior defined in the component, the shared behavior spec can validate that the component has those functions/behavior which the mixin needs (i.e. testing it conforms to the mixin’s interface). - If a mixin provides behavior to a component that might get easily overridden or messed up by the component, a shared behavior spec can be a way to assert that doesn’t happen.

With the three available options to test a mixin (direct, faux, and shared behavior), each one comes with it’s own pros and cons. When testing a mixin, don’t feel afraid to try one option and switch if it isn’t working out. And it’s possible that a combination of different solutions covering different parts of the mixin might be the best approach!

Chapter 8. DOM Manipulation

For the most part, React’s virtual DOM is sufficient to create the user experience you want without having to work directly with the actual underlying DOM at all. By composing components together you can weave complex interactions together into a cohesive whole for the user.

However, in some cases there is no avoiding working with the underlying DOM to accomplish what you need. The most common use cases for this are when you need to incorporate a third-party library that does not use React, or when you need to perform an operation that React does not natively support.

To facilitate this, React provides a system for working with DOM nodes that are being managed by React. They are only accessible during certain parts of the component lifecycle, but using them gives you the power you need to handle these use cases.

Accessing Managed DOM Nodes

To access DOM nodes managed by React, you must first be able to access the components responsible for managing them. Adding a `ref` attribute to child components lets you do this.

```
var DoodleArea = React.createClass({
  render: function() {
    return <canvas ref="mainCanvas" />;
  }
});
```

This will make the `<canvas>` component accessible via `this.refs.mainCanvas`. As you can imagine, the `ref` that you give a child component must be unique among all of a component's children; giving a different child a `ref` of `mainCanvas` as well would not work.

Once we have accessed the child component in question, we can invoke that child's `getDOMNode()` method to access its underlying DOM node. However, we cannot do this in the `render` method, because the underlying DOM nodes may not be up-to-date (or even created yet!) until `render` completes and React performs its updates.

As such, you cannot invoke the `getDOMNode()` method until the component has been "mounted" - at which point the `componentDidMount` handler will run.

```
var DoodleArea = React.createClass({
  render: function() {
    // We are unmounted inside render(), so this will cause an exception!
    this.getDOMNode();

    return <canvas ref="mainCanvas" />
  },

  componentDidMount: function() {
    var canvasNode = this.refs.mainCanvas.getDOMNode()
    // This will work! We now have access to the HTML5 Canvas node,
    // and can invoke painting methods on it as desired.
  }
});
```

Note that `componentDidMount` is not the only place in which you can call `getDOMNode`. Event handlers also fire after a component has mounted, so you can use it just as easily inside them as you would in a `componentDidMount` handler.

```
var RichText = React.createClass({
  render: function() {
    return <div ref="editableDiv" contentEditable="true"
onKeyDown={this.handleKeyDown}>
  },

  handleKeyDown: function() {
    var editor = this.refs.editableDiv.getDOMNode()
    var html = editor.innerHTML;

    // Now we can persist the HTML content the user has entered!
  }
});
```

The above example creates a `div` with `contentEditable` enabled, allowing the user to enter rich text into it.

Although React does not natively provide a way to access a component's raw HTML contents, the `keyDown` handler can accesses that div's underlying DOM node, which in turn can access the raw HTML. From there, you can save a copy of what the user has entered so far, compute a word count to display, and so on.

Incorporating Non-React Libraries

There are many useful JavaScript libraries that were not built with React in mind. Some do not need DOM access (for example, date and time manipulation libraries), but for those that do, keeping their states synchronized with React is critical for successful integration.

Suppose you want to use an autocomplete library which includes the following example code:

```
autocomplete({
  target: document.getElementById("cities"),
  data: [
    "San Francisco",
    "St. Louis",
    "Amsterdam",
    "Los Angeles"
```

```
  ],
  events: {
    select: function(city) {
      alert("You have selected the city of " + city);
    }
  }
});
```

This `autocomplete` function needs a target DOM node, a list of strings to use as data, and then some event handlers. To reap the benefits of both React and this library, you can start by building a React component that provides each of these.

```
var AutocompleteCities = React.createClass({
  render: function() {
    return <div id="cities" ref="autocompleteTarget" />
  },

  getDefaultProps: function() {
    return {
      data: [
        "San Francisco",
        "St. Louis",
        "Amsterdam",
        "Los Angeles"
      ]
    };
  },

  handleSelect: function(city) {
    alert("You have selected the city of " + city);
  }
```

```
});
```

To finish wrapping this library in React, add a `componentDidMount` handler which connects the two implementations via the underlying DOM node of the `autocompleteTarget` child component.

```
var AutocompleteCities = React.createClass({
  render: function() {
    return <div id="cities" ref="autocompleteTarget" />
  },

  getDefaultProps: function() {
    return {
      data: [
        "San Francisco",
        "St. Louis",
        "Amsterdam",
        "Los Angeles"
      ]
    };
  },

  handleSelect: function(city) {
    alert("You have selected the city of " + city);
  },

  componentDidMount: function() {
    autocomplete({
      target: this.refs.autocompleteTarget.getDOMNode(),
      data: this.props.data,
      events: {
```

```
          select: this.handleSelect
        }
      });
    }
});
```

Remember that `componentDidMount` can be called multiple times, and the underlying DOM nodes in question may not have been recreated in between calls. As such, make sure that calling `autocomplete` (in this example) twice on the same node will not have any undesired effects.

Summary

When using the virtual DOM alone is not sufficient, the `ref` attribute allows you to access specific elements and modify their underlying DOM nodes using `getDOMNode` once `componentDidMount` has run.

This allows you to make use of functionality that React does not natively support, or to incorporate third-party libraries that were not designed to interoperate with React.

Next, it's time to look at how you can write tests to ensure your React components are working as expected.

Chapter 9. Forms

Forms are an essential part of any application that requires even modest input from their users. Traditionally in one page apps, forms are hard to get "right," since they are littered with changing state from the user. Managing this state is complex and often buggy. React.js helps you manage state in your applications, and this also extends to your forms.

By now you will have realized that predictability and testability are central aspects of React.js components. Given the same props and state, any React.js component should render exactly the same every time. Forms are no exception.

There are two types of form components in React.js: **Controlled** and **Uncontrolled**. In this chapter you will learn what the differences between **Controlled** and **Uncontrolled** are and under what circumstances you will want to one over the other.

This chapter also covers:

- How to use form events with React.js.
- Controlling data input with Controlled form components.
- How React.js changes the interface of some form components.
- The importance of naming your form components in React.js.
- Dealing with multiple Controlled form components.
- Creating custom reusable form components.
- Using AutoFocus with React.js.
- Tips for building usable applications.

The example application, Survey Builder, utilises forms in a non standard way as they are being dynamically generated based on the survey criteria. As such, the examples in this chapter are generic in nature to help convey the concepts utilised in the example application and help you to gain a stronger knowledge of how to work with forms in React.

Uncontrolled Components

In most non trivial forms, you will **not** want to user **Uncontrolled** components. They are however very useful to help understand **Controlled** components. **Uncontrolled** are an anti pattern for how most other components are constructed in React.js.

In HTML, forms work differently to React.js components. When an HTML <input/> is given a value, the <input/> can then mutate the value of the <input/>. This is where the name comes from, since the form component's value is **Uncontrolled** by your React.js component.

In React.js, this behaviour is like setting the `defaultValue` of the <input/>.

To set the default value of an <input/> in React.js you use the `defaultValue` property.

```
//http://jsfiddle.net/5g6n0102/
var MyForm = React.createClass({
  render: function () {
    return <input
      type="text"
      defaultValue="Hello World!" />;
  }
});
```

The example above is referred to as an **Uncontrolled** component. The `value` is not being set by the parent component, allowing the <input/> to control it's own value.

An **Uncontrolled** component is not very useful unless you can access the value. In order to access the value, you need to add a `ref` to your <input/> then access the DOMNode value.

A `ref` is a special non DOM attribute used to identify a component from within `this` context. All refs in a component are added to `this.refs` for easy access.

Let's also add the <input/> to a form and read the value on submit.

```
//http://jsfiddle.net/kvh031wc/
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    //access the input by it's ref
    var helloTo = this.refs.helloTo.getDOMNode().value;
    alert(helloTo);
  },
  render: function () {
```

```
    return <form onSubmit={this.submitHandler}>
        <input
          ref="helloTo"
          type="text"
          defaultValue="Hello World!" />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

Uncontrolled components are fantastic for basic forms that don't require any special validation or input controls.

Controlled Components

Controlled components follow the same pattern for other React.js components. The state of the form component is **Controlled** by your React.js component, with it's value being stored in your React.js components state.

If you want more control over your form components, you will want to use a **Controlled** component.

A **Controlled** component is where the parent component set's the value of the input.

Let's convert the example above to use a **Controlled** component.

```
//http://jsfiddle.net/tc6vkzsz/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      helloTo: "Hello World!"
    };
```

```
  },
  handleChange: function (event) {
    this.setState({
      helloTo: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.helloTo);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <input
          type="text"
          value={this.state.helloTo}
          onChange={this.handleChange} />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

The major change here is that the value of the <input/> is now being stored in the state of the parent component. As a result, the data flow is now clearly defined.

- getInitialState now sets the defaultValue
- the <input/> value is set during render
- onChange of the <input/> value the change handler is invoked
- the change handler updates the state
- the <input/> value is updated during render

This is a lot more code than the **Uncontrolled** version. However, this allows you to control the data flow and alter the state as the data is being entered.

Example: You may want to convert all character to uppercase as they are being entered.

```
handleChange: function (event) {
  this.setState({
    helloTo: event.target.value.toUpperCase()
  });
}
```

You will notice when entering data that there is no flicker of the lowercase character before the uppercase character is added to the input. This is because React.js is intercepting the native browser change event. Then this component re-renders the input after setState has been called. React.js then does the DOM diff and updates the value of the input.

You can use this same pattern to limit what characters can be entered, or not allow illegal characters to be entered into an email address.

You may also want to use the value in other components as the data is being entered.

Example:

- Show how many characters are left in a size limited input.
- Display the color of a HEX value being entered.
- Display input validation errors and warnings.

Form Events

Accessing form events is a crucial aspect to controlling different aspects of your forms.

All of the events produced by HTML are supported in React.js. They follow camel case naming conventions and are converted to synthetic events. They are standardized, with a common interface cross browser.

All synthetic events give you access to the DOMNode that emitted the event via `event.target`.

```
handleEvent: function (syntheticEvent) {
  var DOMNode = syntheticEvent.target;
  var newValue = DOMNode.value;
}
```

This is one of the easiest ways to access the value of **Controlled** components.

Label

Labels on form elements are important for clearly communicating your requirements of your users, and provide accessibility for radios and checkboxes.

There is a conflict with the `for` attribute. When using JSX the attributes are converted to a JavaScript object and passed to the component constructor as the first argument. Since `for` is a reserved word in JavaScript, we can't use it as the name of an object property.

In React.js, just as `class` becomes `className`, so too does `for` become `htmlFor`.

```
//JSX
<label htmlFor="name">Name:</label>


//javascript
React.DOM.label({htmlFor:"name"}, "Name:");


//after render
<label for="name">Name:</label>
```

Textarea and Select

React.js makes some changes to the interface of <textarea/> and <select/> to increase consistency and make it easier to manipulate.

<textarea/> is changed to be closer to <input/> allowing you to specify `value` and `defaultValue`.

```
//Uncontrolled
<textarea defaultValue="Hello World" />


//Controlled
<textarea
  value={this.state.helloTo}
  onChange={this.handleChange} />
```

<select/> now accepts `value` and `defaultValue` to set which option is selected. This allows easier manipulation of the value.

```
//Uncontrolled
<select defaultValue="B">
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>


//Controlled
<select value={this.state.helloTo}
onChange={this.handleChange}>
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>
```

React.js supports multi select. In order to use multi select you must pass an array

to `value` and`defaultValue`.

```
//Uncontrolled
<select multi="true" defaultValue={["A","B"]}>
  <option value="A">First Option</option>
  <option value="B">Second Option</option>
  <option value="C">Third Option</option>
</select>
```

When using multi select, the value of the select component is **not** updated when the options are selected. Only the `selected` property of the option is changed. Using a given `ref` or `syntheticEvent.target`you can access the options and check if they are selected.

In the following example `handleChange` is looping over the DOM to find out what option is currently selected.

```
//http://jsfiddle.net/u97u3tmt/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      options: ["B"]
    };
  },
  handleChange: function (event) {
    var checked = [];
    var sel = event.target;
    for(var i=0; i < sel.length; i++){
      var option = sel.options[i];
      if (option.selected){
        checked.push(option.value);
      }
    }
    this.setState({
      options: checked
```

```
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.options);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <select multiple="true"
            value={this.state.options}
            onChange={this.handleChange}>
          <option value="A">First Option</option>
          <option value="B">Second Option</option>
          <option value="C">Third Option</option>
        </select>
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

Checkbox and Radio

Checkboxes and Radios have a different mechanism for controlling them.

Just as in HTML, an <input/> with type `checkbox` or `radio` behave quite differently to an <input/> with type `text`. Generally, the value of a checkbox or radio does not change. Only the `checked` state changes. To control a checkbox or radio input, you need to control the `checked` attribute. You can also use`defaultChecked` in an uncontrolled checkbox or radio input.

```
//Uncontrolled - http://jsfiddle.net/euusg6bu/
```

```
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.refs.checked.getDOMNode().checked);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <input
          ref="checked"
          type="checkbox"
          value="A"
          defaultChecked="true" />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});

//Controlled - http://jsfiddle.net/2k7y2p7r/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      checked: true
    };
  },
  handleChange: function (event) {
    this.setState({
      checked: event.target.checked
    });
  },
```

```
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.checked);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <input
          type="checkbox"
          value="A"
          checked={this.state.checked}
          onChange={this.handleChange} />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

In both the examples, at all times the value of the <input/> will always be `A`, only the checked state changes.

Names on Form Elements

Names carry less importance on from elements in React.js when controlled form elements have values that are stored in state and the form submit event is being intercepted. Names aren't required to access the form values. For uncontrolled form elements, you can use refs for direct access to the form element also.

However, names are a crucial aspect of your form components.

- Names allow third party form serializers to still work within React.js.
- Names are also required if the form is going to be natively submitted.

- Names are used by your clients browser for auto filling common information like the users address.
- Names are crucial to uncontrolled **radio** inputs as that is how they are grouped to ensure that only one radio with the same name in a form can be checked at once. The same behaviour can be replicated without names using controlled radio inputs.

The following example replicates the functionality of an **uncontrolled** radio group by storing the state in the MyForm component. You will notice that **name** is not being used.

```
//http://jsfiddle.net/hh986e09/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      radio: "B"
    };
  },
  handleChange: function (event) {
    this.setState({
      radio: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.radio);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <input
          type="radio"
          value="A"
          checked={this.state.radio == "A"}
```

```
            onChange={this.handleChange} /> A
          <br />
          <input
            type="radio"
            value="B"
            checked={this.state.radio == "B"}
            onChange={this.handleChange} /> B
          <br />
          <input
            type="radio"
            value="C"
            checked={this.state.radio == "C"}
            onChange={this.handleChange} /> C
          <br />
          <button type="submit">Speak</button>
        </form>;
    }
});
```

Multiple Form Elements and Change Handlers

When using **Controlled** form elements, you don't want to be writing a change handler for every form element. Fortunately, here are a few different ways of re-using a change handler with React.js.

Example: passing extra arguments to .bind.

```
//http://jsfiddle.net/gb4wxfe2/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      given_name: "",
```

```
      family_name: ""
    };
  },
  handleChange: function (name, event) {
    var newState = {};
    newState[name] = event.target.value;
    this.setState(newState);
  },
  submitHandler: function (event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <label htmlFor="given_name">Given Name:</label>
        <br />
        <input
          type="text"
          name="given_name"
          value={this.state.given_name}

onChange={this.handleChange.bind(this,'given_name')}/>
        <br />
        <label htmlFor="family_name">Family Name:</label>
        <br />
```

```
        <input
          type="text"
          name="family_name"
          value={this.state.family_name}

onChange={this.handleChange.bind(this,'family_name')}/>
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

Example: use DOMNode name to identify what state to change.

```
//http://jsfiddle.net/smgtp2kw/
var MyForm = React.createClass({
  getInitialState: function () {
    return {
      given_name: "",
      family_name: ""
    };
  },
  handleChange: function (event) {
    var newState = {};
    newState[event.target.name] = event.target.value;
        this.setState(newState);
  },
  submitHandler: function (event) {
    event.preventDefault();
    var words = [
      "Hi",
```

```
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <label htmlFor="given_name">Given Name:</label>
        <br />
        <input
          type="text"
          name="given_name"
          value={this.state.given_name}
          onChange={this.handleChange}/>
        <br />
        <label htmlFor="family_name">Family Name:</label>
        <br />
        <input
          type="text"
          name="family_name"
          value={this.state.family_name}
          onChange={this.handleChange}/>
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

The last two examples are very similar but take a different approach to solving the same problem. React.js also provides a mixin as part of their

addons, `React.addons.LinkedStateMixin` solves this same problem in yet another way.

React.addons.LinkedStateMixin adds the method `linkState` to your component. `linkState` returns an object with two properties, value and requestChange.

`value` takes it's value from state from the property name supplied.

`requestChange` is a function that updates the named state with the new value.

```
this.linkState('given_name');


//returns
{
  value: this.state.given_name,
  requestChange: function (newValue) {
    this.setState({given_name: newValue});
  }
}
```

This object needs to be passed to a React.js special non DOM attribute `valueLink`. `valueLink` updates the value of the input with the value of the object supplied and provides an `onChange` handler that calls requestChange with the new DOMNode value.

```
//http://jsfiddle.net/q0cn6wcw/
var MyForm = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState: function () {
    return {
      given_name: "",
      family_name: ""
    };
  },
```

```
  submitHandler: function (event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <label htmlFor="given_name">Given Name:</label>
          <br />
          <input
            type="text"
            name="given_name"
            valueLink={this.linkState('given_name')} />
          <br />
        <label htmlFor="family_name">Family Name:</label>
        <br />
        <input
          type="text"
          name="family_name"
          valueLink={this.linkState('family_name')} />
        <br />
        <button type="submit">Speak</button>
      </form>;
  }
});
```

This is a much easier way to control an input and store the value in the parent components state. The data flow stays the same as other controlled form elements.

However, using this approach becomes more complex to inject custom functionality into the data flow. This mixin is only recommended to be use in limited circumstances. As the traditional controlled form element approach provides the same functionality and more flexibility.

Custom Form Components

Creating custom form components is an excellent way to reuse common functionality in your applications. It can also be a great way to improve the interface to other more complex form components like Checkboxes and Radios.

When writing custom form components you should try to create the same interface as other form components. This will increase the predicability of your code, making it much easier to understand how your component works without having to look at it's implementation.

Let's create a custom radio component that has the same interface as the React.js select component. We won't implement multi select functionality as radio components don't support multi select.

```
var Radio = React.createClass({
  propTypes: {
    onChange: React.PropTypes.func
  },
  getInitialState: function () {
    return {
      value: this.props.defaultValue
    };
  },
  handleChange: function (event) {
    if (this.props.onChange) {
      this.props.onChange(event);
```

```
    }
    this.setState({
      value: event.target.value
    });
  },
  render: function () {
    var children = {};
    var value = this.props.value || this.state.value;

    React.Children.forEach(this.props.children, function
(child, i) {
      var label = <label>
          <input
            type="radio"
            name={this.props.name}
            value={child.props.value}
            checked={child.props.value == value}
            onChange={this.handleChange} />
          {child.props.children}
          <br/>
        </label>;

        children['label' + i] = label;
    }.bind(this));

    return this.transferPropsTo(<span>{children}</span>);
  }
});
```

Essentially we are creating a **Controlled** component that supports the Controlled and Uncontrolled interface.

The first thing you do is make sure that if `onChange` is supplied that it is always a function. Then we store the defaultValue in state.

Each time the component renders, it creates new labels and radios based of the options that were supplied as children to our component. We also make sure that the dynamic children have consistent keys each render. This ensures that React.js will keep our <input/>'s in the DOM and maintain current focus when using keyboard controls.

We then set the value, name and checked state. And we attach our onChange handler, then render our new children.

```
//Uncontrolled
//http://jsfiddle.net/jqqpk0uh/
var MyForm = React.createClass({
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.refs.radio.state.value);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <Radio ref="radio" name="my_radio" defaultValue="B">
          <option value="A">First Option</option>
          <option value="B">Second Option</option>
          <option value="C">Third Option</option>
        </Radio>
        <button type="submit">Speak</button>
      </form>;
  }
});
```

We did have to change the interface slightly when using our component **Uncontrolled**. As when you use `.getDOMNode` on `this.refs.radio`, you will get the DOMNode. Not the active <input/>. In React.js you can't change the functionality of `getDOMNode()` to overwrite this behaviour.

As we are storing the value in state of our component, we don't need to access the DOMNode to know what the current value is. We can access the state directly.

```
//Controlled
//http://jsfiddle.net/13ux2797/
var MyForm = React.createClass({
  getInitialState: function () {
    return {my_radio: "B"};
  },
  handleChange: function (event) {
    this.setState({
      my_radio: event.target.value
    });
  },
  submitHandler: function (event) {
    event.preventDefault();
    alert(this.state.my_radio);
  },
  render: function () {
    return <form onSubmit={this.submitHandler}>
        <Radio name="my_radio"
            value={this.state.my_radio}
            onChange={this.handleChange}>
          <option value="A">First Option</option>
          <option value="B">Second Option</option>
          <option value="C">Third Option</option>
        </Radio>
        <button type="submit">Speak</button>
      </form>;
  }
});
```

As a **Controlled** component, this operates exactly the same as a select box. The

event passed to`onChange` is the event from the active <input/> so you can use it read the current value.

As an exercise, you may want to try and implement support for a valueLink property so you can use this component with `React.addons.LinkedStateMixin`.

Focus

Leveraging control of focus on form components is an excellent way to guide your users as to what the next logical step is in your forms. It also helps cut down on user interaction, increasing usability.

Since React.js forms are not always rendered at browser load, the auto focus for form inputs needs to operate a little differently. React.js has implemented `autoFocus` so when the component is first mounted, if no other form input has focus, React.js will place focus on the input. This is how you would expect a simple HTML form with `autoFocus` to operate.

```
//jsx
<input type="text" name="given_name" autoFocus="true" />
```

You can also manually set the focus of form fields by calling `focus()` on the

DOMNode.

Usability

React.js is awesome for productivity for developers, however, this can have down sides.

It is really easy to make components that lack usability. For example, you may have forms with little keyboard support where only `onClick` of a hyperlink can submit the form. This stops a user from pressing `enter` on their keyboard to submit the form, which is the default behaviour of HTML forms.

It is also really easy to make fantastic components that are highly usable. Time and consideration are required when building your components. It is all the “little things” that help make a component be highly usable and “feel right”.

This following is a collection of best practice for creating usable forms. They are not specific to React.js.

Communicate your requirements clearly

Good communication is important with all aspects of your application, especially in forms.

Use of HTML labels is a great way to communicate to your users what the form element is expecting. These also give the user an extra way to interact with your form element for some input types like radios and checkboxes.

Placeholders are designed to show example input or a default value if no data is entered. There has been a fad to place validation hints in the placeholder. This can be quite problematic as when the user starts to type, their validation hints disappear. It is generally better to show your validation hints along side your inputs or as a popover when your validation requirements are not met.

Give feedback constantly

This follows on from communicating your requirements clearly. It is very important to give feedback to your users as quickly as possible.

Validation errors are a perfect example of giving feedback constantly. It is well known that showing validation errors as they occur increase the usability of your forms. Back in the early days of the web applications all users had to wait till they finished completing their form to find out if they entered everything correctly. Validation in the browser was a massive leap forward in usability.

It is also important to show your users that you are working on their request. This is especially true for actions that can take some time to complete. Showing spinners, progress bars, notification messages etc are a great way to inform your users that your application has not frozen. Users are very impatient at times, however they can be very patient if they know your application is processing their request.

Transitions and animations are another great way of informing your users what is happening in your application. They are another visual prompt that something in your application has changed.

Be fast

React.js has a very powerful rendering engine, it helps your application to be quite fast right out of the box. However there are times when the speed of updating the DOM is not what is slowing down your application.

Transitions are a perfect example. As transitions that take to long may frustrate your users and slow your power users down as it reduces how much time your user can engage with your applications.

Other factors outside your application can also alter how fast your application performs. Long running AJAX calls and poor network performance also impact the speed of your applications. How to solve these kinds of issues can be very specific to your application and may be outside of your control like third party services.

It is important to remember that speed is relative. It is a perception of the user. It is more important to appear fast than be fast. Example, when a user clicks “like” in your application, you can increment the like count before you send your AJAX call to the server. This way if the AJAX call takes a long time, your users won’t see the delay. This can however lead into other issues with handling errors etc.

Be predictable

Users have a predetermined idea of how things work. This is based on their prior experience. In most cases their prior experience is **not** from using your application.

If you application resembles the platform your user is on, your user will expect your application conforms to the default behaviours for the platform you are on.

With this in mind, you are left with two options, either conform to the default behaviours of the platform, or, radically change your user interface so it no longer resembles he platform.

Consistency is another form of predictability, if your interactions are always the same in different parts of your application, your users will learn to predict the

interaction in new parts of your application. This ties into conforming with platform that your application runs on.

Be accessible

Accessibility is often over looked by developers and designers when creating user interfaces. It is important to keep your users in mind when considering all aspects you user interfaces. As already covered, your users have a predefined expectation of how things work, this is based on **their** past experience.

Their past experience also governs their preference for different input types. In some cases your users may have physical issues with using particular input device like a keyboard or mouse. They may also have issues with using an output device like a display or speakers.

Building support for all the different types of input and output device may not be viable for every aspect of your application. It is important to understand your users requirements and preferences, then work on those areas first.

A great way to test how accessible your application is to try and navigate your application exclusively via one input device keyboard / mouse / touch screen. This will highlight usability issues with that device.

You may also want to consider how to interact with your application if you were visually impaired. Screen readers are the eyes for visually impaired people.

Reduce user input

Reducing how much data your users need to enter is a great way to improve usability for your applications. The less information your users need to enter, the less chance they have of making mistakes and the less they need to think about.

As with **Be fast** user perception is important. Users feel daunted by large forms with many input fields, their mind struggles to cope. Breaking your forms into smaller more manageable chunks gives the impression of less user input. This in turn enables the user to be more focused on their data input.

Autofill is another great way to reduce data input. Leveraging the users browser autofill data can save the user from re-entering common information like their address or payment details.

Autocomplete can help guide your users as they are entering information, this ties in with giving constant feedback. Example, when searching for a movie, autocomplete can help alleviate issues with bad spelling of the movie title.

Another useful way to reduce input is to derive information from previously entered data. Example, if a user is entering credit card details, you can look at the first 4 numbers and determine what type of card it is, then select the card type for your user, this both reduces input and provides validation feedback to the user that they are typing their card number correctly.

Autofocus is a small but very effective way to increase usability of your forms. Autofocus helps to guide your user to starting point for their data entry. This saves them from trying to work that out on their own. This small tool can really have a large impact on how quickly your users can start entering data.

Chapter 10. Animations

Now that you can compose together a complex set of React components and test them, it's time to add some polish. Animation can make user experiences feel more fluid and natural, and React's Transition Groups addon, in conjunction with CSS3, make incorporating animated transitions into your React project a breeze.

Historically, animation in browsers has had a very imperative API. You would take an element and actively move it around or alter its styles in order to animate it. This approach is at odds with React's rendering and re-rendering of components, so instead React takes a more declarative approach.

CSS Transition Groups facilitate applying CSS3 animations to transitions, by strategically adding and removing classes during appropriately timed renders and re-renders. This means the only task you are left with is to specify the appropriate CSS styles for those classes.

Interval Rendering gives you more flexibility and control, at a cost to performance. It requires many more re-renderings, but allows you to animate more than just CSS, such as scroll position and Canvas drawing.

CSS Transition Groups

Take a look at how the Survey Builder renders the list of questions in the survey editor.

```
<ReactCSSTransitionGroup transitionName='question'>
  {questions}
</ReactCSSTransitionGroup>
```

This `ReactCSSTransitionGroup` component comes from an addon, which is included near the top of the file with `var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup`.

This automatically takes care of re-rendering the component at appropriate times, and altering the transition group's class in order to facilitate styling it situationally based on where it is in the transition.

Styling Transition Classes

By convention, the `transitionName='question'` attribute connects this element to four CSS classes: `question-enter`, `question-enter-active`, `question-leave`, and `question-leave-active`. The CSSTransitionGroup addon will automatically add and remove these classes as child components enter or leave the `ReactCSSTransitionGroup` component.

Here are the transition styles used by the survey editor.

```
.survey-editor .question-enter {
  transform: scale(1.2);
  transition: transform 0.2s cubic-bezier(.97,.84,.5,1.21);
}

.survey-editor .question-enter-active {
  transform: scale(1);
}
```

```
.survey-editor .question-leave {
  transform: translateY(0);
  opacity: 0;
  transition: opacity 1.2s, transform 1s cubic-bezier(.52,-
0.25,.52,.95);
}

.survey-editor .question-leave-active {
  opacity: 0;
  transform: translateY(-100%);
}
```

Note that these `.survey-editor` selectors are not required for the addon, but are simply used to make sure these styles are only applied within the editor.

The Transition Lifecycle

The difference between `question-enter` and `question-enter-active` is that the `question-enter` class is applied as soon as the component is added to the group, and the `question-enter-active` class is applied on the next tick. This allows you to easily specify the beginning style of the transition, the end style of the transition, and how to perform the transition.

For example, when survey editor questions are added to the list, they begin with an inflated scale (1.2) and then transition to a normal scale (1) over the course of 0.2 seconds. This creates the “popping into place” effect you see.

By default, the transition groups enable both the enter and leave animations, but you can disable either or both by adding the `transitionEnter={false}` or `transitionLeave={false}` attributes to the component. In addition to giving you control over which of the two you want, you can also use them to situationally disable animations altogether based on a configurable value, like so:

```
<ReactCSSTransitionGroup transitionName='question'
  transitionEnter={this.props.enableAnimations}
```

```
    transitionLeave={this.props.enableAnimations}>
    {questions}
  </ReactCSSTransitionGroup>
```

Transition Group Pitfalls

There are two important pitfalls to watch out for when using transition groups.

First, the transition group will defer removing child components until animations complete. This means that if you wrap a list of components in a transition group, but do not specify any CSS for the `transitionName` classes, those components can no longer be removed - even if you try stop rendering them!

Second, transition group children must each have a unique `key` attribute set. The transition group uses these values to determine when components are entering or leaving the group, so animations could fail to run and components could become impossible to remove if they are missing their `key` attributes.

Note that even if the transition group only has a single child, it must still have a `key` attribute.

Interval Rendering

You can get great performance and concise code out of CSS3 animations, but they are not always the right tool for the job. Sometimes you have to target older browsers which do not support them. Other times you want to animate something other than a CSS property, such as scroll position or a Canvas drawing. In these cases Interval Rendering will accommodate your needs, but at a cost to performance compared to CSS3 animations.

The basic idea behind Interval Rendering is to periodically trigger a component state update which specifies how far the animation has progressed across its total running time. By incorporating this state value into a component's render function, the component can represent the appropriate stage of the animation each time the state change causes it to re-render.

Since this approach involves many re-renderings, it's typically best to use it in conjunction with `requestAnimationFrame` in order to avoid unnecessary renders. However, in environments where requestAnimationFrame is unavailable or otherwise undesirable, falling back on setTimeout can be a reasonable alternative.

Interval Rendering with requestAnimationFrame

Suppose you want to move a div across the screen using interval rendering. You could accomplish this by giving it `position: absolute` and then updating its `left` or `top` style attributes as time progressed. Doing this on `requestAnimationFrame`, and basing the amount of the change on how much time has elapsed, should result in a smooth animation.

Here is an example implementation.

```
var Positioner = React.createClass({
  getInitialState: function() { return {position: 0}; },

  resolveAnimationFrame: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationCompleteTimestamp -
timestamp);

    if (timeRemaining > 0) {
      this.setState({position: timeRemaining});
    }
  },

  componentWillUpdate: function() {
    if (this.props.animationCompleteTimestamp) {
      requestAnimationFrame(this.resolveAnimationFrame);
    }
  },

  render: function() {
    var divStyle = {left: this.state.position};

    return <div style={divStyle}>This will animate!</div>
  }
});
```

In this example, the component has an `animationCompleteTimestamp` value set in its `props`, which it uses in conjunction with the timestamp returned by `requestAnimationFrame`'s callback to compute how much movement remains. The resulting value is stored as `this.state.position`, which the `render` method then uses to position the `div`.

Since `requestAnimationFrame` is invoked by the `componentWillUpdate` handler, it will be kicked off by any change to the component's props (such as a change to `animationCompleteTimestamp`), which includes the call to `this.setState` inside `resolveAnimationFrame`. This means that once `animationCompleteTimestamp` is set, the component will automatically continue firing successive requestAnimationFrame invocations until the current time exceeds the value of `animationCompleteTimestamp`.

Notice that this logic revolves around timestamps only. A change to `animationCompleteTimestamp` kicks it off, and the value of `this.state.position` depends entirely on the difference between the current time and `animationCompleteTimestamp`. As such, the render method is free to use `this.state.position` for any sort of animation, from setting scroll position to drawing on a canvas, and anything in between.

Interval Rendering with setTimeout

Although `requestAnimationFrame` is likely to get you the smoothest animation with the least overhead, it is not available in older browsers and may fire more often (or less predictably) than you want it to. In those cases you can use `setTimeout` instead.

```
var Positioner = React.createClass({
  getInitialState: function() { return {position: 0}; },

  resolveSetTimeout: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationCompleteTimestamp -
timestamp);

    if (timeRemaining > 0) {
      this.setState({position: timeRemaining});
    }
  },

  componentWillUpdate: function() {
    if (this.props.animationCompleteTimestamp) {
      setTimeout(this.resolveSetTimeout, this.props.timeoutMs);
    }
  },

  render: function() {
    var divStyle = {left: this.state.position};

    return <div style={divStyle}>This will animate!</div>
  }
});
```

Since `setTimeout` takes an explicit time interval, whereas requestAnimationFrame

determines that interval on its own, this component has the additional dependency of `this.props.timeoutMs`, which specifies the interval to use.

Summary

Using these animation techniques, you can now:

1. Efficiently animate transitions between states using CSS3 and Transition Groups.
2. Animate outside CSS, such as scroll position and Canvas drawing, using requestAnimationFrame.
3. Fall back on setTimeout in cases where requestAnimationFrame is not a viable option.

Next you'll be ready to take your React renderings to the server!

Chapter 11. Performance Tuning

React's DOM diffing allows you to effectively discard your entire UI at any point in time with minimal impact on the DOM. How ever there are cases where delicate tuning of which component gets rendered can help make your application faster. For instance if you have a very deeply nested component tree. In this chapter we will cover a simple configuration you can provide to your component to help speed up your application.

shouldComponentUpdate

When a component is updated, either by receving new props, setState or forceUpdate being called, React will re-render each child component of that

component. In most cases this performs without a hitch, but on pages with deeply nested component trees, you can experience some sluggishnes.

Sometimes a component gets re-rendered when it didn't need to be. For instance if your component doesn't use state or props or if the props or state didn't change when the parent re-rendered.

React provides the component lifecycle method `shouldComponentUpdate`, which gives you a way to help React make the right decision of whether or not to re-render a specific component.

`shouldComponentUpdate` should return a boolean. `false` tells React to skip the re-rendition of the component and use the previously rendered version. Returning `true` will tell React to re-render the component. By
default `shouldComponentUpdate` returns true and components will thus always re-render.
Note that `shouldComponentUpdate` is only called on re-rendition, not on the initial rendition of a component.

`shouldComponentUpdate` receives the new props and state as arguments to help your make a decision of whether or not to re-render:

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

In cases where you have deep complex props or state, comparison can be trick and slow. To help mitigate this, consider using immutable data structures like Immutable-js, which we cover in the Family chapter.

React.addons.Perf provides good insight into which components should have a manually added`shouldComponentUpdate` method by giving your a list of components that React wasted most time re-rendering. Read more about React.addons.Perf in the Addons chapter.

Key

Most often you'll find the key prop used in lists. Its purpose is to identify a component to react by more than the component class. For example if you have a div with key="foo" which later changes to "bar", react will skip the diffing and throw out the children completely, rendering from scratch.

This can be useful when rendering large subtrees to avoid a diff which you know is a waste of time.

In addition to telling it when to throw out a node, in many cases it can be used when the order of elements changes. For example, consider the following render which shows items based on a sorting function.

```
var items = sortBy(this.state.sortingAlgorithm,
this.props.items);

return items.map(function(item){

  return <img src={item.src} />;

});
```

If the order changes, react will diff these and determine the most efficient operation is to change the src attribute on some img elements. This is very inefficient and will cause the browser to consult its cache, possibly causing new network requests.

To remedy this, we can simply use some string (or number) we know is unique to each item, and use it as a key.

```
  return <img src={item.src} key={item.id} />;
```

Now React will look at this and instead of changing src attributes, it will realize the minimum insertBefore operations to perform, which is the most efficient way to move DOM nodes.

Single Level Constraint

key props must be unique to a given parent. This also means that no moves from one parent to another will be considered.

In addition to the order changing, insertions that aren't at the end also apply. Without correct key attributes prepending an item to the array would cause the src attribute of all following <img> tags.

Chapter 12. Server Side Rendering

Server side rendering is vital if you want search engines to crawl your site. It also provides a performance boost as the browser can start displaying your site while your javascript is still loading.

The virtual DOM in React.js is central to why React.js can be used for server side rendering. Each React.js component is first rendered to the virtual DOM, then React.js takes the virtual DOM and updates the real DOM with any changes. The virtual DOM, being an in memory representation of the DOM is what allows React.js to work in non browser environments like nodejs. Instead of updating the real DOM, React.js can generate a string from the virtual DOM. This allows you to use the same React.js components on the client and the server.

React.js provides two functions which can be used to render your components on the server side,`React.renderComponentToString` and `React.renderComponentToStaticMarkup`.

Server side rendering of your components requires forsight when designing your components. You need to consider:

- What render function is best to use.
- How to support Asychronous state of your components.
- How to pass your applications initial state to the client.
- What lifecycle functions are available on the server side.
- How to support Isomorphic routing for you application.
- Your usage of Singletons, Instances and Context.

Render functions

SOME INTRO TEXT HERE

React.renderComponentToString

The first of the two render functions that can be used on the server side is`React.renderComponentToString`. This is function you will mostly use.

Unlike React.renderComponent this function does not take an argument of where to render, instead it returns a string. This is a synchronous (blocking) function that is very fast.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>Hello World!</div>;
  }
});

var world = React.renderComponentToString(<MyComponent/>);

//single line output - formatted for this example
<div
  data-reactid=".fgvrzhg2yo"
  data-react-checksum="-1663559667"
>Hello World!</div>
```

You will notice that React.js has added two data attributes to the `<div>`.

`data-reactid` is used by React.js to identify the DOM node in the browser environment. This is how it knows what DOM node to update when state and props change.

`data-react-checksum` is only added on the server side. As the name suggest it is a checksum of the DOM that is created. This allows React.js to reuse the DOM from the server when rendering the same component on the client. This is only added to the root element.

React.renderComponentToStaticMarkup

`React.renderComponentToStaticMarkup` is the second render function you can use on the server side.

This is the same as React.renderComponentToString except it doesn't include the React.js data attributes.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>Hello World!</div>;
```

```
  }
});

var world = React.renderComponentToStaticMarkup(<MyComponent/>);

//single line output
<div>Hello World!</div>
```

String or StaticMarkup

Each render function has it purposes, you need look at what your requirements are to decide which render function to use.

>Only use `React.renderComponentToStaticMarkup` when you are **not** going to render the same React.js component on the client.

Examples:

- Generating html emails.
- Generating PDF's by html to pdf conversion.
- Testing components.

In most cases you will want to use `React.renderComponentToString`. This will allow React.js to use the`data-react-checksum` to make the initialization of the same React.js component on the client much faster. As React.js can reuse the DOM supplied from the server, it can skip the expensive process of generating DOM nodes and attaching them to the document. For a complex site, this can significantly reduce the load so your users can start interacting with your site faster.

NOTE:
It is very important that your React.js components render **exactly** the same on the server and on the client. If the `data-react-checksum` doesn't match, React.js will destroy the DOM supplied by the server and generate new DOM nodes and attach them to the document. In this case you will loose much of the performance gain of server side rendering.

Server Side Component Lifecycle

When rendering to string, only lifecycle methods **before** render are called. Crucially `componentDidMount`and `componentWillUnmount` are not called during the render process and `componentWillMount` is called from both.

You will need to keep this in mind when creating components that will be rendered on the server and the client. Especially when creating event listeners, as there is no component lifecycle method to let you know when the React.js component is finished.

Any event listeners or timers started in `componentWillMount` have the potential to cause memory leaks on the server.

Best practice is to only create event listeners and timers from `componentDidMount` and stop the event listeners and timers in `componentWillUnmount`.

Designing Components

When rendering on the server side, you need to take special consideration to how your state will be passed to the client to leverage the perfomance benifits of server side rendering. This means designing your components with server sider rendering in mind.

You should always design your React.js components so that when you pass the same props to the component you will always get the same **initial** render. If you always do this, you will increase testability of your components and when you render on the server and the client you can be guaranteed they will render the same. This is very important to ensure you will get the performance benefit of server side rendering.

Let's say that you want to have a component that prints a random number. This is an issue as the result will almost always be different. If this component were to render on the server then render on the client, the checksum would fail.

```
var MyComponent = React.createClass({
  render: function () {
    return <div>{Math.random()}</div>;
  }
});

var result = React.renderComponentToStaticMarkup(<MyComponent/>);
```

```
var result2 = React.renderComponentToStaticMarkup(<MyComponent/>);

//result
<div>0.5820949131157249</div>

//result2
<div>0.420401572631672</div>
```

If you were to change your component so it received the random number by way of props. You can then pass the props to the client to be used for the rendering.

```
var MyComponent = React.createClass({
   render: function () {
      return <div>{this.props.number}</div>;
   }
});

var num = Math.random();

//server side
React.renderComponentToString(<MyComponent number={num}/>);

//pass num to client side
React.renderComponent(<MyComponent number={num}/>, document.body);
```

There are number of different ways to send the props used on the server to the client.

One of the easiest ways is to pass the initial props to the client is by way of a javascript object.

```
<!DOCTYPE html>
<html>
<head>
<title>Example</title>
<!-- bundle contains MyComponent, React, etc -->
<script type="text/javascript" src="bundle.js"></script>
</head>
<body>
<!-- result of MyComponent server side render -->
```

```
<div data-reactid=".fgvrzhg2yo" data-react-checksum="-
1663559667">
0.5820949131157249</div>


<!-- inject initial props used on the server side -->
<script type="text/javascript">
  var initialProps = {"num": 0.5820949131157249};
</script>


<!-- use initial prop from server -->
<script type="text/javascript">
  var num = initialProps.num;
  React.renderComponent(<MyComponent number={num}/>,
document.body);
</script>
</body>
</html>
```

Asynchronous State

Many applications require data from a remote data source like a database or web service. On the client, this is not an issue the React.js component can show a loading view while waiting for the asynchronous data to return. On the server side, this behavour can't be directly replicated with React.js as the render function is a synchronous function. In order to use asynchronous data, you need to fetch the data first then pass the data to the component at render time.

Example:

You may want to fetch the user record from an asynchronous store for use in the component.

AND

You want the state of the component after the user record has been fetched as what is rendered on the server side for SEO and performance reasons.

AND

You want the component to listen to changes on the client and re-render.

Problem: You can't use any of the component lifecycle methods to fetch the asynchronous data as React.renderComponentToString is synchronous.

Solution: Use a statics function to fetch your asynchronous data then pass it to the component to render. Pass the initialState to the client as props. Use component lifecycle methods to listen to changes and update the state, using the same statics function.

```
var Username = React.createClass({
  statics: {
    getAsyncState: function (props, setState) {
      User.findById(props.userId)
        .then(function (user) {
          setState({user:user});
        })
        .catch(function (error) {
          setState({error: error});
        });
    }
  },
  //client and server
  componentWillMount: function () {
    if (this.props.initialState) {
      this.setState(this.props.initialState);
    }
  },
  //client side only
```

```
  componentDidMount: function () {
    //get async state if not in props
    if (!this.props.initialState) {
      this.updateAsyncState();
    }
    //listen to changes
    User.on('change', this.updateAsyncState);
  },
  //client side only
  componentWillUnmount: function () {
    //stop listening to changes
    User.off('change', this.updateAsyncState);
  },
  updateAsyncState: function () {
    //access static function from within the instance
    this.constructor.getAsyncState(this.props,
this.setState);
  },
  render: function () {
    if (this.state.error) {
      return <div>{this.state.error.message}</div>;
    }
    if (!this.state.user) {
      return <div>Loading...</div>;
    }
    return <div>{this.state.user.username}</div>;
  }
});

//Render on the server
```

```
var props = {
  userId: 123 //could also be derived from route
};

Username.getAsyncState(props, function(initialState){
  props[initialState] = initialState;
  var result =
React.renderComponentToString(Username(props));

  //send result to client along with initialState
});
```

Using the solution above - pre fetching of asynchronous data is only required on the server. On the client just the initial render needs to look for initialState passed from the server. Subsequent route changes on the client (e.g. html5 pushState or fragment change) should ignore any initialState from the server. You will also want to display loading messages etc as the data is fetched.

Isomorphic Routing

Routing is an essential part of any non trivial application. To render a React.js application with a router on the server side, you need to ensure the router supports the server side.

Fetching of asynchronous data is the job of a router and it's route controllers. Let's say a deeply nested component needs some asynchronous data. If the data is needed for SEO purposes then the responsibility for fetching the data should be moved higher up to the route controller and the data passed down to the deeply nested component. If it is not needed for SEO then it is ok to just fetch the data from within componentDidMount on the client. This is akin to classical ajax loaded data.

When looking at an isomorphic routing solution for React.js, make sure it either has asynchronous state supported or can be easily modified to support asynchronous state. Ideally you would also like the router to handle passing the initialState down to the client.

Singletons, Instances and Context

In the browser, your application is wrapped in a isolation bubble. Each instance of your application can't mix state with other instances, as each instance is usually on different computers or sandboxed on the same computer. This allows you to easily use singletons in your application architecture.

When you start moving your code to operate on the server, you need to be careful, as you may be having multiple instances of your application running at the same time, in the same scope. This has the potential that two instances of you application may mutate the state of a singleton, resulting in unwanted behaviour.

React.js render is synchronous, so you can reset all singletons used prior to rendering your application on the server side. You will have issues if asynchronous state requires use of the singletons also, you will need to account for this when fetching the asynchronous state for use in render.

Even with resetting singletons, prior to render, running your application in isolation will always be better. Packages like **contexify** allow you to run your code in isolation from one anther on the server. It is similar to using webworkers on the client.

The core development team of React.js discourage the use of passing context and instances down your component tree. As this makes your components less portable and changes to the dependencies of one component deep in your application have a flow on effects with all components up the component hierarchy. This in turn increases the complexity of your application, reducing maintainability as your application grows.

Serverside Testing

In Chapter 6 you learned about unit testing React.js components, where all of that testing was in the context of the "clientside", ie the browser. But in this chapter, you will learn about techniques for using React.js components on the serverside, i.e. Nodejs, so it's important that you learn how to test that part too. For these examples, we will be using the `mocha` testing framework. We could use the `jasmine-node`project too, but mocha is very popular in the node ecosystem for it's excellent async support, so we'll use that here.

NO RELIGIOUS WARS NEEDED

Both Jasmine and Mocha are excellent testing frameworks. You will be happy with either choice you make. The React.js codebase uses Jasmine for it's tests (it actually wraps Jasmine in a project called Jest, which we'll cover in Chapter 19), so we wanted to give you exposure to Jasmine. But Mocha is picking up a lot of momentum for node projects (and works very well paired with chai for assertions), so we wanted to give you exposure to Mocha too.

In the SurveyApp, the routing for the application uses `react-router` and these routes are used both on the clientside and the serverside. This dual use makes them "isomorphic" JavaScript code. Let's walk through how we would test this routing code when used on the serverside. If you look at `client/app/app_router.js`, you will notice it does a few things:

1. `require`'s our application router:

```
2.      var app_router = require("../../client/app/app_router");
```

3. Uses an express router as your middleware:

```
4.      var router = require('express').Router({caseSensitive: true,
strict: true});
5.
6.      ...
7.      router.use(function (req, res, next) {
8.      ...
9.      });
```

10. Calls react-router with the URL:

```
11.     Router.renderRoutesToString(app_router, req.originalUrl)
```

12. Defines a success handler to render the output HTML into your template:

```
13.     var template = fs.readFileSync(__dirname +
"/../../client/app.html", {encoding:'utf8'});
14.     ...
15.     Router.renderRoutesToString(app_router,
req.originalUrl).then( function (data) {
```

```
16.       var html = template.replace(/\{\{body\}\}/, data.html);
17.       html = html.replace(/\{\{title\}\}/, data.title);
18.       res.status(data.httpStatus).send(html);
19.
20.     }, ...);
```

To test this process, let's get started by making a spec file called `test/server/routing.test.js` and giving it the following boilerplate:

```
var request = require('supertest');
var app = require('../../server/server.js');


describe("serverside routing", function(){
});
```

Here's what is happening in our serverside spec boilerplate:

1. We are requiring our `server.js` node.js application.
2. We are using the `supertest` module. This will allow us to execute requests to our node server without having to boot up an actual server.
3. The `describe` block is a `mocha` function, not a jasmine function. We'll start to see differences with the mocha api in a bit.

To run our node tests, create the following file `test/server/main.js`:

```
require('./routing.test.js');
```

And then execute the following command: `npm run-script test-server`. You should see the following output:

```
tom:bleeding-edge-sample-app (master) $ npm run-script test-
server

> bleeding-edge-sample-app@0.0.1 test-server
/Users/tom/workspace/bleeding-edge-sample-app

> mocha test/server/main.js
```

```
  0 passing (5ms)
```

Great. Our boilerplate is fine and our Mocha test run doesn't have any obvious errors. Let's now add our first test, which should make a GET request to the node server for to the `/add_survey` endpoint:

```
...
describe("serverside routing", function(){
  it("should render the /add_survey path successfully",
function(done){
    request(app)
      .get('/add_survey')
      .expect(200)
      .end(done);
  });
});
```

This test looks *similar* to a Jasmine test, but there is one major difference - the `done` function. Do you notice how the anonymous function passed to `it` takes an argument which we call `done`. This is a callback function that we need to call when the test is "done" and we've made our assertions. In this test we are making a GET request to the "/add_survey" endpoint to the `server.js app`. We are then expecting a response code of 200. And then add the end, we are calling `done` to tell Mocha our assertions have been called.

SUPERTEST

The `request`, `get`, `expect`, and `end` functions are all part of the supertest project. Feel free to read the supertest documentation here: `https://github.com/visionmedia/supertest`.

When we run this test, it should pass:

```
tom:bleeding-edge-sample-app (master) $ npm run-script test-
server
```

```
> bleeding-edge-sample-app@0.0.1 test-server
/Users/tom/workspace/bleeding-edge-sample-app

> mocha test/server/main.js



  serverside routing

    ✓ should render the AddSurvey component for the
/add_survey path (87ms)



  1 passing (97ms)
```

Hooray! Now we should add some assertions about the actual content we get back.

BE CAREFUL CALLING DONE()

Don't write the above test like this. It won't work because the `done` function will be called before the`expect` code has run in the chain.

```
it("should render the /add_survey path successfully",
function(done){
  request(app)
    .get('/add_survey')
    .expect("666");

    // DON'T DO THIS!!  This test will pass every
time, even though the assertion is not true
    done();
});
```

To validate that the `AddSurvey` React component is rendered correctly, we should make some assertions against the returned HTML. Let's first start by making some minor tweaks to our test:

```
...
  it("should render the AddSurvey component for the
/add_survey path", function(done){


    request(app)
      .get('/add_survey')
      .expect(200)
      .end(function (err, res) {
        console.log("OUR HTML IS: " + res.text)
        done();
      });
  });
...
```

This should output something like this:

```
tom:bleeding-edge-sample-app (master) $ npm run-script test-
server

> bleeding-edge-sample-app@0.0.1 test-server
/Users/tom/workspace/bleeding-edge-sample-app

> mocha test/server/main.js




  serverside routing

HTML IS: <!DOCTYPE html>

<html>
  <head lang='en'>
    ...
    <title>Add Survey to SurveyBuilder</title>
    ...
  </head>
```

```
  <body>
  <div class="app" data-reactid=".qajmfw0740" data-react-
checksum="2024417999">...</div>
  <script src="/build/bundle.js"
type="text/javascript"></script>
  </body>
</html>
    ✓ should render the AddSurvey component for the
/add_survey path (89ms)


  1 passing (100ms)
```

Very cool! We can see that our Node application is correctly rendering the `AddSurvey` component to string, and placing it in our template inside of

the `body` tag. Now we *could* do basic string comparisons against `res.text`, but this would be very painful and brittle, so let's parse the HTML response and then make some assertions based on that. To parse the HTML, we will use a library called Cheerio. Cheerio is a node module that allows you to "load" some HTML and perform jQuery-like operations on it. So our test will look like this:

```
var cheerio = require('cheerio');
...
it("should render the AddSurvey component for the
/add_survey path", function(done){
  request(app)
    .get('/add_survey')
    .expect(200)
    .end(function (err, res) {
      var doc = cheerio.load(res.text);
      expect(doc("title").html()).to.be("Add Survey to
SurveyBuilder");
      expect(doc(".main-content .survey-
editor").length).to.be(1);
```

```
        done();
      });
});


...
```

In the `end` function we are doing the following:

1. Getting the html response from the `res.text`
2. Parsing it with a call to `cheerio.load`
3. Asserting on the innerHTML of the `<title>` element
4. Asserting the selector ".main-content .survey-editor" is on the page

This example should pass just fine! To get another example under our belt, this is a test to verify the 404 handler is working correctly:

```
it("should render a 404 page for an invalid route",
function(done){
  request(app)
    .get('/not-found-route')
    .expect(404)
    .end(function (err, res) {
      var doc = cheerio.load(res.text);
      expect(doc("body").html()).to.contain("The Page you
were looking for isn&apos;t here!");
      done();
    });
}
```

Chapter 13. Addons

React comes with a set of addons. We use several off them throughout our Sample application.

In this chapter we will go into how each addon works and how they can help you with your React application. We'll cover the following React addons (the full set at the time of the writing):

- React.addons.CSSTransitionGroup for Animations
- React.addons.LinkedStateMixin to help with two-way data bindings
- React.addons.classSet for simple class name manipultation
- React.addons.TestUtils for easing your React unit testing
- React.addons.cloneWithProps for component cloning
- React.addons.update for simple immutability
- React.addons.PureRenderMixin for quick performance boosts
- React.addons.Perf for diagnosting performance bottlenecks in your application

First though, it's important to note that the default React distribution doesn't come with all the addons. However, when you download the starter kit from http://facebook.github.io/react/downloads.html it includes a distribution with all the addons called react.addons.js. This file is required instead of the regular React.js distrubution if you want to use any of the addons.

Animations

React provides a component via it's addons that helps with animations, we can use it to improve the experience of our SurveyBuilder.

Let's add an animation when you add or remove any questions for the SurveyEditor.

Wrapping our list of EditQuestion modules in the `React.addons.CSSTransitionGroup` component will add a series of CSS classes to modules when ever that list changes

Here's a shorted version of our `<SurveyEditor/>` component with the added`React.addons.CSSTransitionGroup` code.

```
var ReactCSSTransitionGroup =
React.addons.CSSTransitionGroup;


var SurveyEditor = React.createClass({
  getInitialState: function () {
    return {
      questions: []
      // ...
    };
  },


  render: function () {
    var questions = this.state.questions.map(function (q, i)
{
      return SUPPORTED_QUESTIONS[q.type]({
        key: i,
        onChange: this.handleQuestionChange,
        onRemove: this.handleQuestionRemove,
        question: q
      });
    }.bind(this));


    return (
      <div className='survey-editor'>
        <div className='row'>
          <aside className='sidebar col-md-3'>
            <h2>Modules</h2>
            <DraggableQuestions />
          </aside>
```

```
            <div className='survey-canvas col-md-9'>
            <SurveyForm />

            <Divider>Questions</Divider>

            <ReactCSSTransitionGroup
transitionName='question'>
              {questions}
            </ReactCSSTransitionGroup>

            // ...
            </div>
          </div>
        </div>
      );
    },

    // ...

});
```

Now this by it self doesn't cause any animations, this simply adds and removes CSS classes. The transitionName property we gave the `ReactCSSTransitionGroup` component is used for the prefix of each added CSS classes. In our case it's "question" and thus the 4 CSS classes added (depending wether or not React saw a new element or an element was removed) is `.question-enter`, `.question-enter-active` `.question-leave` and `.question-leave-active`. We can target these classes in our stylesheet and add an animation when a question is added and an animation when it is removed.

There are 2 classes for elements entering as children to the ReactCSSTransitionGroup component and 2 classes for leaving.

The first class that is added is `.question-enter`. This is meant for you to set the stage, prepare the animation if you will.

We want sort of a “drop-in” effect. We achieve this by starting the element at a 1.2 scale and animating it down. We use a cubic-bezier to control the easing so that it feels bouncy:

```
.survey-editor .question-enter {
  transform: scale(1.2);
  transition: transform 0.2s cubic-bezier(.97,.84,.5,1.21);
}
```

On the very next UI tick the `.question-enter-active` class is added. This is our cue to start the animation. We simply reset the scale to 1 which kicks off the animation set in `.question-enter`.

```
.survey-editor .question-enter-active {
  transform: scale(1);
}
```

The same principles apply for when a child leaves `ReactCSSTransitionGroup`.

Here we add a fade animation while the question moves upwards:

```
.survey-editor .question-leave {
  transform: translateY(0);
  opacity: 0;
  transition: opacity 1.2s, transform 1s cubic-bezier(.52,-
0.25,.52,.95);
}


.survey-editor .question-leave-active {
  opacity: 0;
  transform: translateY(-100%);
}
```

Two-Way Binding Helpers

To get data from a form field when the user interacts with it we can add an onChange handler to the element like so:

```
var EditEssayQuestion = React.createClass({
  getInitialState: function () {
    return {
      title: ''
    };
  },
  render: function () {
    return (
      <div>
        <input
          type='text'
          value={this.state.title}
          onChange={this.handleTitleChange}/>
      </div>
    );
  },
  handleTitleChange: function (ev) {
    this.setState({ title: ev.target.value });
  }
});
```

Notice that we set the value of the input field to `this.state.title`. This means that the input is “controlled”, and if we don’t add an onChange handler that updates the state, any value the user enters will be discarded. Simply put, with the above example, every time the user enters something into the input field the `handleTitleChange` function is called, we update the state to the new value and the render function gets called again and that finally updates the input field, making the change visible to the user.

Now, building onChange handlers like this is simple, but can be made even simpler with the`LinkedStateMixin`.

We can re-write the above example like so and it will build the onChange handler for us behind the scenes.

```
var EditEssayQuestion = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState: function () {
    return { title: '' };
```

```
  },
  render: function () {
    return (
      <div>
        <input type='text'
valueLink={this.linkState('title')}/>
      </div>
    );
  }
});
```

`React.addons.LinkedStateMixin` is especially useful in big form components that need to track a lot of state.

Class Name Manipulation

It quickly becomes annoying to manually construct a className string. You can imagine the string being made off of varying state and props.

With the `React.addons.classSet` you can turn:

```
var SurveyRow = React.createClass({
  render: function () {
    var className = 'survey-row';
    if (this.props.survey.isActive) {
      className += ' active';
    }
    return (
      <tr className={className}>
        <td>{this.props.survey.title}</td>
      </tr>
    );
  }
```

```
});
```

into this:

```
var SurveyRow = React.createClass({
  render: function () {
    var classSet = React.addons.classSet({
      'survey-row': true,
      'active': this.props.survey.isActive
    });
    return (
      <tr className={classSet}>
        <td>{this.props.survey.title}</td>
      </tr>
    );
  }
});
```

Test Utilities

React provides simple a few test utilities available on React.addons.TestUtils - These are all covered in the 'Testing' chapter.

Cloning Components

Somtimes it's useful to clone components. Perhaps you seeded a table with a default row component and want to clone it for adding a new row.

This is how it works:

```
var surveyRowClone = React.addons.cloneWithProps(
  <SurveyRow/>,
  { className: 'survey-row-clone'}
```

```
);
```

The second argument is extra props you can pass the clone.

Immutability Helpers

React doesn't force you to choose the kind of data structures you wish to use in your application. Using immutable datastructures can make your `shouldComponentUpdate` functions simpler when you need to complete objects to check for changes.

We can use `React.addons.update` to ensure immutability in our `<SurveyEditor/>` component by updating our change handler functions:

```
var update = React.addons.update;


var SurveyEditor = React.createClass({
  // ...


  handleDrop: function (ev) {
    var questionType =
ev.dataTransfer.getData('questionType');
    var questions = update(this.state.questions, {
      $push: [{ type: questionType }]
    });


    this.setState({
      questions: questions,
      dropZoneEntered: false
    });
  },


  handleQuestionChange: function (key, newQuestion) {
```

```
      var questions = update(this.state.questions, {
        $splice: [[key, 1, newQuestion]]
      });

      this.setState({ questions: questions });
    },

    handleQuestionRemove: function (key) {
      var questions = update(this.state.questions, {
        $splice: [[key, 1]]
      });

      this.setState({ questions: questions });
    }

    // ...

});
```

`React.addons.update` takes a datastructure and an options hash. You can pass in `$slice`, `$push`,`$unshift`, `$set`, `$merge` and `$apply`.

PureRenderMixin

For pure components that always render the same given the same props and state we can add the`React.addons.PureRenderMixin` which gives a performance boost.

The mixin will overwrite `shouldComponentUpdate` to compare the new props and state with the old and return false if they are equal.

A few of our components are this simple, like our EditEssayQuestion, which we can use`React.addons.PureRenderMixin` with:

```
var EditEssayQuestion = React.createClass({
  mixin: [React.addons.PureRenderMixin],

  render: function () {
    var description = this.props.question.description || "";

    return (
      <EditQuestion type='essay'
onRemove={this.handleRemove}>
        <label>Description</label>
        <input type='text' className='description'
value={description} onChange={this.handleChange} />
      </EditQuestion>
    );
  },
  // ...
});
```

Performance Tools

As we've mentioned in the last few sections. Adding a custom `shouldComponentUpdate` function to your components can be a great way to optimize your application.

`React.addons.Perf` can help you find the best places to add `shouldComponentUpdate`.

Lets use `React.addons.Perf` to find slowness in our Survey Builder application, specifically the `<SurveyEditor/>` page.

First in the chrome web developer console we run `React.addons.Perf.start();` This will start the snapshot. We'll drag in a few questions and then run `React.addons.Perf.stop();` followed by `React.addons.Perf.printWasted();` in the console.

It gives us this result:

```
> React.addons.Perf.printWasted()
```

(index)	Owner > component	Wasted time (ms)	Instances
0	"ReactTransitionGroup >…	51.088000007439405	86
1	"SurveyEditor > Draggab…	19.280999898910522	28
2	"SurveyEditor > SurveyF…	10.835000139195472	28
3	"DraggableQuestions > M…	8.496999624185264	84
4	"SurveyEditor > ReactCS…	7.958000060170889	6
5	"AddSurvey > SurveyEdit…	3.349000005982816	1
6	"SurveyEditor > Divider"	2.7780000236816704	28
7	"EditMultipleChoiceQues…	2.5900001055561006	34
8	"SurveyForm > ReactDOMT…	2.0920000970363617	28
9	"SurveyForm > ReactDOMI…	1.800999918486923	28
10	"SurveyEditor > ReactDO…	1.7349999397993088	28

ReactDefaultPerf.js:99

Total time: 136.04 ms ReactDefaultPerf.js:106

We can't do much with `<ReactTransitionGroup/>` since we don't control that component, but a simple fix to `shouldComponentUpdate` on `<DraggableQuestions/>` could work wonders. `<DraggableQuestion/>` is actually quite a simple component. The way its constructed means it should never need to change. Lets update the `shouldComponentUpdate` function to always return false:

```
var DraggableQuestions = React.createClass({
  render: function () {
    return (
      <ul className="modules list-unstyled">
        <li><ModuleButton text='Yes / No'
questionType='yes_no'/></li>
        <li><ModuleButton text='Multiple choice'
questionType='multiple_choice'/></li>
        <li><ModuleButton text='Essay'
questionType='essay'/></li>
      </ul>
    );
  },

  shouldComponentUpdate: function () {
```

```
      return false;
    }
  });
```

This removes the component entirely from the list of actions that had wasted time:

```
> Perf.printWasted()
```

(index)	Owner > component	Wasted time (ms)	Instances
0	"ReactTransitionGroup > ReactCS…"	53.026000037789345	57
1	"SurveyEditor > SurveyForm"	14.322000090032816	19
2	"SurveyEditor > ReactCSSTransit…"	8.422000042628497	4
3	"SurveyEditor > Divider"	3.816000127699226	19
4	"SurveyForm > ReactDOMInput"	3.748999966774136	19
5	"SurveyForm > ReactDOMTextarea"	2.1199999027885497	19
6	"EditMultipleChoiceQuestion > R…"	2.111000183504075	28

ReactDefaultP

Total time: 117.07 ms ReactDefaultPe

-

Chapter 14. Tools and Debugging

No matter how careful you are, mistakes will be made. We won't cover how to debug JavaScript, but we will talk about some tools to make debugging React applications easier.

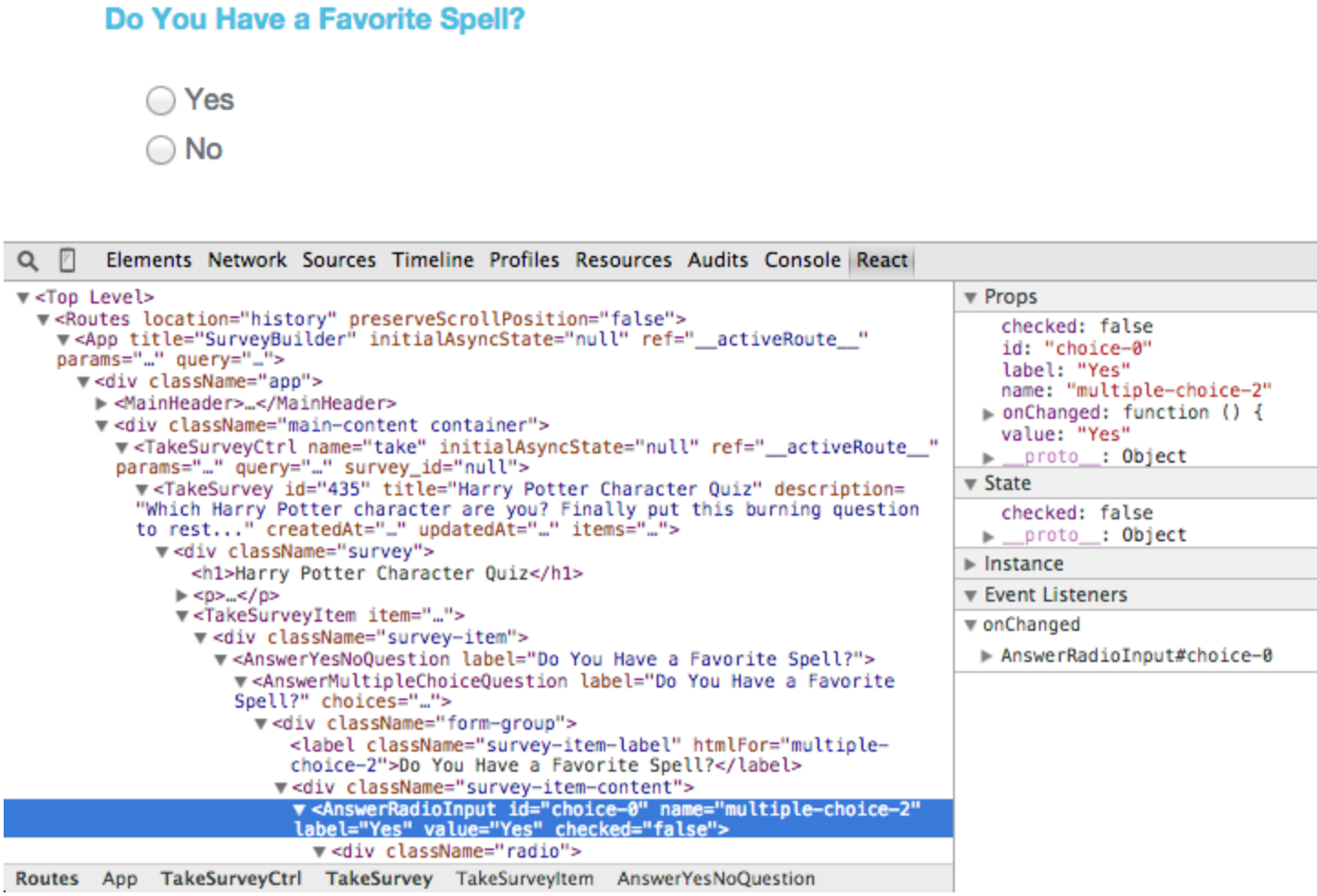
Starting out

For this chapter, please open Chrome and install the React Developer Tools addon. One requirement is to set window.React.

```
window.React = require('react');
```

Right click on an element, and choose "Inspect Element". You get the familiar view of the DOM structure, the elements tab. Click the

You're not here to look at the DOM, though. You want to see the components, their props, and their state. If completed the previous steps you should see a tab on the far right named "React".



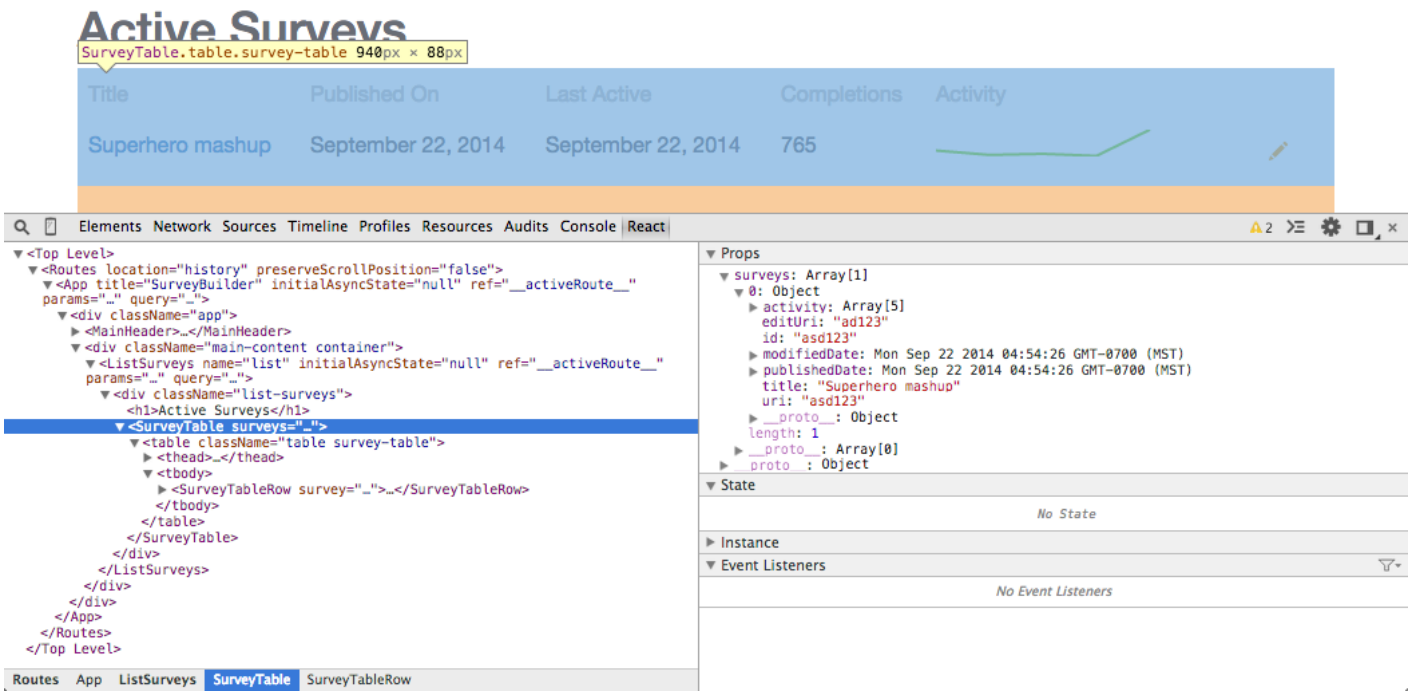
DISPLAYNAME

The JSX to JS converter tries to guess the displayName and insert it. If you're not using JSX you should provide a displayName for your components.

```
React.createClass({displayName: "MyComponent", ...});
```

The hierarchy of components is on the left, and the information about the selected instance is on the right.

Viewing just this information can tell you a lot about the state, props, and event listeners attached to our components. You can see that the onDragStart listener is ModuleButton::handleDragStart, the classes our button has, and any other information you might be interested in. The developer tools allow you to do more than this.



You can see our SurveyTable component gets passed an array of surveys which contain timestamps. In the actual view they're presented in a human readable format. Double click one of the timestamps and type in a new value. This causes the component to update with the new value.

The developer tools help you narrow down problems, and helps new people to your team find the components they need to edit to complete tasks.

JSBin and JSFiddle

While debugging, or just brainstorming, online demo sites like JSFiddle and JSBin are great assets. Use them to create test cases when asking for help, or to share prototypes with others.

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 [2], the law of demeter [3], and keeping code modular.

Types of testing

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 [4]), 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 [Clientside]: jasmine and karma
  - Alternatives: Mocha, Chai, sinon, vows.js, qunit
- Unit and Integration Testing [Serverside]: mocha and supertest
  - Alternatives: Same as clientside, 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`

```
/** @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 is 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`React.addons.TestUtils.renderIntoDocument`. This function takes one argument, the 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 component FAILED
ReferenceError: HelloWorld is not defined    at
null.<anonymous>
(/var/folders/xz/xk1180q53ml23q_dqz91dvh00000gn/T/770d14d1a2
5ddccd00c6ee3578fac4d283ef1c5d.browserify:30997:16 <-
/Users/tom/workspace/bleeding-edge-sample-
app/test/client/fundamentals/render_into_document_spec.js:15
:0)
```

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


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("Hell
o 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 component and it's html
into a dom node FAILED Expected '' to contain 'Hello
World!'. Error: Expected '' to contain 'Hello World!'.    at
null.<anonymous>
(/var/folders/xz/xk1180q53ml23q_dqz91dvh00000gn/T/770d14d1a2
5ddccd00c6ee3578fac4d283ef1c5d.browserify:31016:52 <-
/Users/tom/workspace/bleeding-edge-sample-
app/test/client/fundamentals/render_into_document_spec.js:25
:0)Chrome 36.0.1985 (Mac OS X 10.8.2) HelloWorld
renderIntoDocument should render the component and return
the component as the return value FAILED Expected undefined
to be 'Bleeding Edge React.js Book'. Error: Expected
undefined to be 'Bleeding Edge React.js Book'.    at
null.<anonymous>
(/var/folders/xz/xk1180q53ml23q_dqz91dvh00000gn/T/770d14d1a2
5ddccd00c6ee3578fac4d283ef1c5d.browserify:31023:38 <-
/Users/tom/workspace/bleeding-edge-sample-
app/test/client/fundamentals/render_into_document_spec.js:32
:0)
```

To make these tests pass, improve our HelloWorld component into this:

```
...

    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("Hell
o 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 ASSERSTIONS

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'>Bleeding Edge React.js Book</h2>");
    });
```

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 UserImage, where UserBadge renders the user's name and their UserImage. When writing tests for the UserBadge component, it is important only UserBadge functionality is being tested and that you are not implicitly testing UserImage 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 UserImage 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 UserImage
component, which is not the desired affect
      it("should use the mock component and not the real
component", function(){
        var userBadge =
TestUtils.renderIntoDocument(<UserBadge />);
      });
});
```

To help us stub out the UserImage 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 UserImage = require("./user_image");
...
  render: function(){
    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 test the UserImage component
      //   with it -- otherwise that turns into an integration
test.  So we replace the UserImage component with a
      //   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 is complete
        //   NOTE: if we don't do this, all subsequent tests will
be polluted!!
        realUserImageComponent = UserBadge.  get  ("UserImage");
        UserBadge.  set  ("UserImage", mockUserImageComponent);
      });

      afterEach(function(){
        UserBadge.  set  ("UserImage", realUserImageComponent);
      });

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


expect(TestUtils.findRenderedDOMComponentWithClass(userBadge,
"fake").getDOMNode().innerHTML).toBe("Fake User Image!!");
      });
    });

  });
});
```

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({
      mod: mod,
      variableName: variableName,
      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(){
  // unwire all modules we rewire'd
  rewireJasmine.unwireAll();

  // reset the array back to an empty state in preperation
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().innerHTML).toBe("Fake User Image!!");
      });
    });
  });
});
```

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.

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 test the UserImage component
      //   with it -- otherwise that turns into an
integration test.  So we replace the UserImage component
with a
      //   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 complete
      //   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().innerHTML).toBe("Fake Vanilla User
Image!!");
    });
  });
```

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

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" }
];


var HelloRandom = React.createClass({
  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.githubUsername}.
      </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 billy.");
    });
  });
});
```

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

"RANDOM" IS JUST AN ILLUSTRATIVE EXAMPLE

We are justing 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

So what we want todo is spyOn our HelloRandom React Class for the getRandomAuthor function. And instead of getRandomAuthor returning a real author, let's have it return a fake author which 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 text
      //   for a given author".
      //   In this example the author is random which makes
it very difficult to test.  But even
      //   if your app doesn't have any "random" functions,
the same strategy would be helpful if
      //   a function like getCurrentAuthor had complex
behavior which should be tested independently
      //   of the render function.
      jasmineReact.spyOnClass(HelloRandom,
"getRandomAuthor").andReturn({name: "Fake User",
githubUsername: "fakeGithub"});


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



expect(myHelloRandom.getDOMNode().textContent).toBe("Fake
User is an author and their github handle is fakeGithub.");
    });
```

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>
      );
  }
});
...
```

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", function(){
      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 the imageClicked function to the
UserImage component FAILED imageClicked() method does not
exist
...
```

This is because our `UserImage` component doesn't have an `imageClicked` function, so we can't spy on it, so let's add one:

```
...
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 the imageClicked function to the UserImage
component FAILED TypeError: 'undefined' is not an object
(evaluating 'imageComponent.props')    at
/var/folders/xz/xk1180q53ml23q_dqz91dvh00000gn/T/770d14d1a25ddccd00
```

```
c6ee3578fac4d283ef1c5d.browserify:31860:0 <-
/Users/tom/workspace/bleeding-edge-sample-
app/test/client/fundamentals/spy on spec.js:48:0
```

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 the imageClicked function to the
UserImage component FAILED TypeError: 'undefined' is not a
function (evaluating 'imageComponent.props.imageClicked()')
   at
/var/folders/xz/xk1180q53ml23q_dqz91dvh00000gn/T/770d14d1a25
ddccd00c6ee3578fac4d283ef1c5d.browserify:31860:0 <-
/Users/tom/workspace/bleeding-edge-sample-
app/test/client/fundamentals/spy_on_spec.js:48:0
```

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="image"/>
      </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");
var TestUtils = React.addons.TestUtils;

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 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({
  getInitialState: function(){
    return { clicks: 0 };
  },
  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 (<img src="http://example.com/logo.png" />);
  }
});

module.exports = CompanyLogo;
```

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 react component is the crystal ball you are looking for components in.

For all “scry*” 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 `scryRenderedComponentsWithType` function:

```
...
    it("should find composite DOM components", function(){
      var results =
TestUtils.scryRenderedComponentsWithType(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 LIMITAITON

As of React.js v11.0, if you want to use scryRenderedComponentsWithType 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 `scryRenderedDOMComponentsWithClass`.

```
...
  describe("scryRenderedDOMComponentsWithClass", function(){
    it("should find all components with that class attribe",
function(){
```

```
      var tabs =
TestUtils.scryRenderedDOMComponentsWithClass(subject,
"tab");
      var activeTabs =
TestUtils.scryRenderedDOMComponentsWithClass(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 topic we covered, rendering a component, and dive a bit deeper. One thing you might notice about `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.`

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 pollution issues

    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 pollution issues

    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("comp
onent 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;

    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 function creates a
      //  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("comp
onent width: placeholder-value");
    });
  });
});
```

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);

    // notice how the width and height of the dom node are
actual values

expect(myComponent.getDOMNode().offsetWidth).not.toBe(0);

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

```
    });

});
```

The main difference here is that we no longer need to call `React.unmountComponentAtNode(el);` 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.

[2] http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

[3] http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf

[4] https://www.youtube.com/watch?v=1wHr-O6gEfc

Chapter 17. Architectural Patterns

React renders HTML. You can consider it as the V in MVC, but it doesn't make any decision if you use simple Ajax requests in your components directly like so:

```
var TakeSurvey = React.createClass({
  getInitialData: function () {
    return {
      survey: null
    };
  },
  componentDidMount: function () {
    $.getJSON('/survey/' + this.props.id, function (json) {
      this.setState({survey: json});
    });
  },
  render: function () {
    if (!this.state.survey) return null;

    return <div>{this.state.survey.title}</div>;
  }
});
```

Or if you use an MVC framework, and thus it can easily be integrated in most application architectures.

In this chapter we will go over a few approaches or libraries you could use with React to help structure your application.

MVC

Will be added in the next version.

Routing

Routers are what directs urls in a single page application to specific handlers. You can imagine that for the url `/surveys` you'd want to run a function that loads the users from the serve r and renders a `<ListSurveys />` component.

There are many different kinds of Routers. They exists on the server as well. Some routers work both on the client and on the server.

React is simply a rendering library and does not come with a Router, but there are many options that work great with React. In this section we'll cover a few.

pushState and hash routing

Will be added in the next version.

Backbone.Router

Backbone is Single Page Application library, which employs the Model-View-Whatever architecture. The "Whatever" is a replacement for "Controller" and usually refers to a Router. This is true in Backbone's case as well.

Backbone is modular and you can choose to only use the Router. It works great with React.

Consider our /surveys example from before with Backbone.Router:

```
var SurveysRouter = Backbone.Router.extend({
  routes: {
    "surveys": "list"
  },
  list: function() {
    React.renderComponent(
      <ListSurveys />,
      document.querySelector('body')
    );
  }
});
```

Routers need to be able to handle dynamic segments in the url or the queryString. Backbone.Router supports this like so:

```
// surveys_router.js
```

```
var SurveysRouter = Backbone.Router.extend({
  routes: {
    “surveys”: “list”,
    “surveys/:filter”: “list”
  },
  list: function(filter) {
    React.renderComponent(
      <ListSurveys filter={filter}/>,
      document.querySelector(‘body’)
    );
  }
});
```

In the above example, given a url like so: ‘/surveys/active` the `filter` argument to `SurveysRouter#list` would be `”active”`.

Learn more about Backbone.Router and download it here: http://backbonejs.org/#Router

Aviator

Unlike `Backbone.Router`, Aviator is a standalone routing library.

With Aviator you define your routes separate from your “RouteTargets”. Aviator doesn’t concern itself with how a `RouteTarget` looks or acts, it just tries to call the assigned functions on it.

Imagine a `RouteTarget` like so:

```
// surveys_route_target.js
var SurveysRouteTarget = {
  list: function () {</div><div>   
React.renderComponent(</div><div>&lt;ListSurveys/&gt;,</div>
<div>document.querySelector(‘body’)</div><div>);</div><div>}
</div><div>};</div>
```

Along with that object, a route configuration (there can be only 1 configuration for the entire application) is required. This configuration is usually in a separate file.

```
// routes.js
Aviator.setRoutes({
  '/surveys': {
    target: UsersRouteTarget,
    '/': 'list'
  }
});
// Tell Aviator to start dispatching urls to targets:
Aviator.dispatch();
```

To setup `RouteTarget`s for handling params:

```
// routes.js
Aviator.setRoutes({
  '/surveys': {
    target: UsersRouteTarget,
    '/': 'list',
    '/:filter': 'list'
  }
});
// surveys_route_target.js
var SurveysRouteTarget = {
  list: function (request) {
    React.renderComponent(
      <ListSurveys filter={request.params.filter}/>,
      document.querySelector('body')
    );
  }
};
```

One of the coolest features in Aviator is multiple targets.

Consider this routes definition:

```
// routes.js
Aviator.setRoutes({
  target: AppRouteTarget,
  '/*': 'beforeAll',
  '/surveys': {
    target: UsersRouteTarget,
    '/': 'list',
    '/:filter': 'list'
  }
});
```

For a URL like `'/surveys/active'` Aviator will call `AppRouteTarget.beforeAll` and then`UsersRouteTarget.list` - There can be any number of matched actions like that. You can provide exit functions for a route as well, which will call the chain of route actions in the reverse outer when the user navigates away from the matched route.

Read more about and download Aviator here: https://github.com/swipely/aviator.

react-router

React-router differs from the other routes in that it is completely made up of React components.

The routes are defined as components and the handlers of the routes are components.

This is how our router looks with react-router:

```
var appRouter = (
  <Routes location="history">
    <Route title="SurveyBuilder" handler={App}>
```

```
      <Route name="list" path="/" handler={ListSurveys} />
      <Route title="Add Survey to SurveyBuilder" name="add"
path="/add_survey" handler={AddSurvey} />
      <Route name="edit" path="/surveys/:surveyId/edit"
handler={EditSurvey} />
      <Route name="take" path="/surveys/:surveyId"
handler={TakeSurveyCtrl} />
      <NotFound title="Page Not Found"
handler={NotFoundHandler}/>
    </Route>
  </Routes>
);
```

Each handler is a component for that specific page. To get the router to run you render it as your top level component:

```
React.renderComponent(
  appRouter,
  document.querySelector('body')
);
```

Like the other Routers, react-router has a similar concept of params. For instance the route `'/surveys/:surveyId'` will pass in the surveyId property to the `TakeSurveyCtrl` component.

One of the cool features of react-router is that at provides a Link component which you can use for navigation and it will map it to the routes. It will even automatically mark the current page as active by adding the 'active' class to the link.

Here's how our `<MainNav/>` component looks with react-router's Link component:

```
var MainNav = React.createClass({
  render: function () {
    return (
      <nav className='main-nav' role='navigation'>
        <ul className='nav navbar-nav'>
          <li><Link to="list">All Surveys</Link></li>
```

```
            <li><Link to="add">Add Survey</Link></li>
          </ul>
        </nav>
      );
    }
  });
```

Read more about and download react-router here: https://github.com/rackt/react-router.

Om (ClojureScript)

Om is a very popular ClojureScript interface to React.
With persistent data structures from ClojureScript Om can re-render from the root of your application very fast. Each rendition is trivial to snapshot and use for something like undo.

Here's how an Om component looks:

```
(ns example
  (:require [om.core :as om :include-macros true]
            [om.dom :as dom :include-macros true]))

(defn App [data owner]
  (reify
    om/IRender
    (render [this]
      (dom/h1 nil (:text data)))))

(om/root App {:text "Survey Builder"}
  {:target (. js/document (querySelector "body"))})
```

This renders: `<h1>Survey Builder</h1>`

Flux

The previous architectural patterns have all emerged around React since it became open-source. However, from the beginning Flux has been the pattern React's original authors intended it to be used with.

Flux is an architectural pattern introduced by Facebook. It compliments React with a uni-directional data-flow that's easy to reason about and requires very little scaffolding to get up and running.

Flux is made of three main parts: the store, the dispatcher, and the views (which are React components.) Actions, which are helper methods to create a semantic interface to the dispatcher, can be thought of as a fourth part of the Flux pattern.

You can think of your top-most React component as a Controller-View. The Controller-View component interfaces with the store and facilitates communication with its children view components. This is not unlike a ViewController in the world of iOS.

Each node in the Flux pattern is independent, enforcing strict separation of concerns and keeping each easily testable in isolation.

Data Flow

A unique aspect of Flux is its one-way information flow. In a world of modern MVC frameworks this can seem strange, but it comes with some distinct benefits. Since Flux does not make use of two-way data binding your app becomes easier to reason about. State is easier to track since it is maintained by the stores (which own the data.) Stores deliver the data through their change methods. This triggers views to render. User input causes actions to dispatch through the dispatcher, which delivers the payload to the stores that care about the particular action.

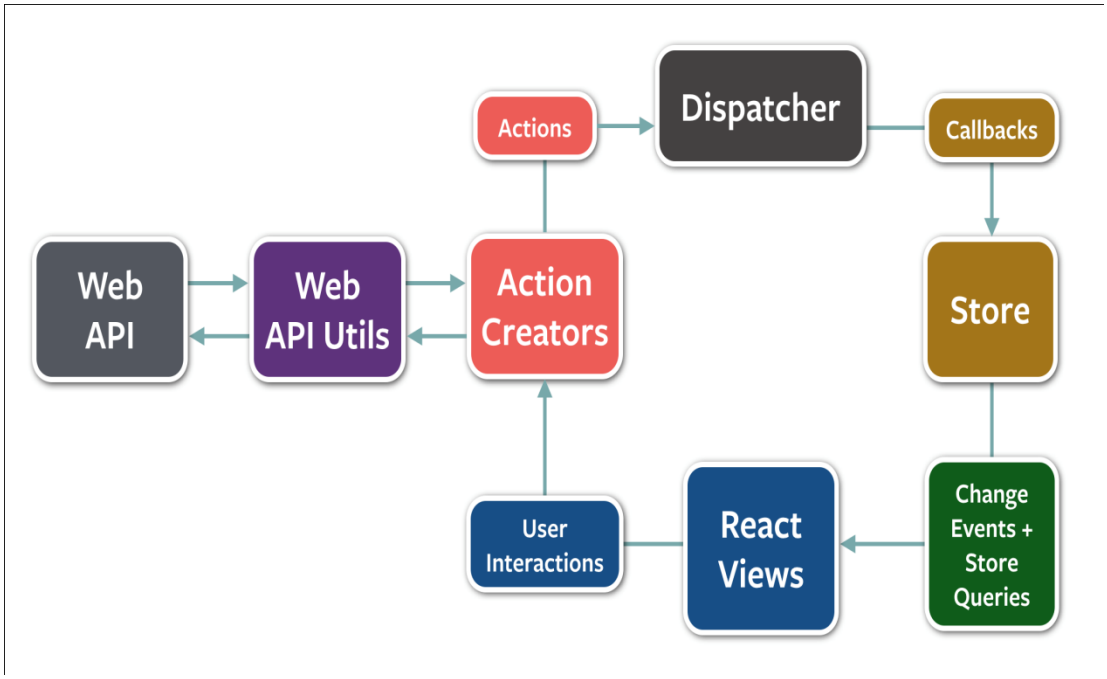


Figure 17-1. Flux enforces a one-way data flow

Flux Parts

Flux is comprised of distinct parts with specific responsibilities. Within the uni-directional flow of information each Flux part takes the input from downstream, processes it, and sends it's output upstream.

- Dispatcher — a central hub within your app
- Actions — the domain-specific-language of your app
- Store — business-logic and data-interactions
- Views — the component tree for rendering the app

Below we discuss each part, what it's responsible for, and how to use it effectively within your React apps.

Dispatcher

We've chosen to start with the dispatcher because it is the central hub through which all user interaction and data flows. The dispatcher is a singleton within the Flux pattern.

The dispatcher is responsible for registering callbacks on the stores and managing the dependencies between them. User actions flow into the dispatcher. A data payload flows out into the stores that have registered for it.

Our sample app contains a fairly simple but effective dispatcher for managing a single store. However, as you grow your app you'll inevitably run into situations where you need to manage multiple stores and their dependencies upon each other. We'll discuss this case below.

Actions

From your user's perspective this is where Flux starts. Every action they take through the UI creates an action that is sent to the dispatcher.

While actions are not a formal part of the Flux pattern, they do constitute the domain-specific-language of your app. User actions are translated into meaningful dispatcher actions — statements that the store can act on.

Our survey-taking app has three distinct actions the user can take:

1. Saving a new survey definition
2. Deleting an existing survey
3. Recording survey results

We've captured these actions within our `SurveyActions` object.

```
var SurveyActions = {
  save: function(survey) {...},
  delete: function(id) {...},
  record: function(results) {...}
};
```

For example, consider a user taking one of our surveys — after they've filled out all the required form fields they click "Save". This "click" action is translated into an "onSave" method call.

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

```
  handleClick: function() {
    this.props.onSave(this.state.results);
  },
  render:function(){
    return (
      <div className="survey">
        ...
        <button ... onClick={this.handleClick}>Save</button>
      </div>
    );
  }
});
```

The `TakeSurveyCtrl` Controller-View handles the `onSave` call, translating the "save" action from the user to the store's "record" action.

```
var TakeSurveyCtrl = React.createClass({
  ...
  handleSurveySave: function(results) {
    SurveyActions.record(results);
  },
  render:function () {
    var props = merge({}, this.state.survey, {
      onSave: this.handleSurveySave
    });
    return TakeSurvey(props);
  }
});
```

In this way user actions flow through the components into the dispatcher. In the process user actions are translated into the domain-specific-language of the Flux app.

Store

The store is responsible for encapsulating business logic and interactions with your app's data. The store chooses which actions to respond to from the dispatcher by registering for them. Stores send their data into the React component hierarchy through their change event.

It's important to keep strict separation of concerns with the store. No other part of your app should know how to interact with the data. All updates flow through the dispatcher into the store. Fresh data flows back into the app through the store's change event.

Following our example above of a user recording their survey results, the dispatcher sends the "record" action to the store.

```
Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    ...
    case SurveyConstants.RECORD_SURVEY:
      SurveyStore.recordSurvey(payload.results);
      break;
  }
});
```

And the store receives the action, performs the work of saving the results, and when done emits the change event.

```
SurveyStore.prototype.recordSurvey = function(results) {
  // handle saving the results here
  this.emitChange();
}
```

This change event is handled by the main Controller-View defined in `app.js`, with the new state flowing through the React component hierarchy, and the React components will re-render as needed.

Controller-View

Your app's component hierarchy will typically have a top-level component responsible for interacting with the store. Simple apps will only have one. More complex apps might have multiple controller-views.

Within our sample app the Controller View `App` is defined in the `app.js` file. The process of wiring up the store is straightforward.

1. When the component mounts, add the change listener
2. As the store changes, request the new data and process accordingly
3. When the component unmounts, clean up the change listener

Here is the snipped from `app.js` for handling store interactions.

```
var App = React.createClass({
  handleChange: function() {
    SurveyStore.listSurveys(function(surveys) {
      // handle the survey data
    });
  },
  componentDidMount: function() {
    SurveyStore.addChangeListener(this.handleChange);
  },
  componentWillUnmount: function() {
    SurveyStore.removeChangeListener(this.handleChange);
  },
  ...
});
```

Managing Multiple Stores

Our simple survey app only requires one store, but inevitably apps grow to need multiple stores. This becomes tricky when one store depends upon another,

requiring the second store to complete its actions before the first can perform its own actions.

For example, maybe we create a separate store to maintain a survey-results summary, tallying the results of all survey respondents. This summary store requires the main store to complete it's "record" action before we can safely update the summary store.

This will require a few changes:

- The Dispatcher must be updated so it can enforce the action queue.
- We need the ability to tell the Dispatcher to wait for an action to complete.
- The callbacks registered on the dispatcher must define which actions they wait upon.

Our stores should not be responsible for any of this. It's the dispatcher that enforces the action queue, and the methods we register with the dispatcher that control the flow of calls to our stores. Therefore it's the functions we register with the dispatcher who own this work.

While a full refactor of the Dispatcher is beyond the scope of this discussion, here we'll discuss the main points concerning multiple stores. A complete solution can be found at[*github.com/facebook/flux*](github.com/facebook/flux)*.*

Updating the Dispatcher

Our existing Dispatcher simply pushes new callbacks into an array. However, to enforce order we need to track each callback. Therefore we refactor our Dispatcher to assign each registered callback an id. This id is the token other callbacks can use when telling the dispatcher they need to wait on another store.

```
Dispatcher.prototype.register = function(callback) {
  var id = uniqueId('ID-');
  this.handlers[id] = {
    isPending: false,
    isHandled: false,
```

```
    callback: callback
  };
  return id;
};
```

Next we must add a `waitFor` method so we can tell the store to invoke certain callbacks before proceeding. An example of the `waitFor` function might look like

this, where we pass in a list of ids returned from the `register` function and make sure they are invoked before proceeding.

```
Dispatcher.prototype.waitFor = function(ids) {
  for (var i = 0; i < ids.length; i++) {
    var id = ids[i];
    if(!this.isPending[id] && !this.isHandled[id]) {
      this.invokeCallback(id);
    }
  }
};
```

Registering For Dependent Actions

Now we have two stores concerned about the `RECORD_SURVEY` action.

- The `SurveyStore` needs to record the survey results
- The `SurveySummaryStore` needs to re-tally all the results.

This means that `SurveySummaryStore` is dependent upon the `SurveyStore` completing the `RECORD_SURVEY` action before it can complete its work.

When we register the SurveyStore at the top of our App, we store a reference to the dispatcher token.

```
// Wire up the SurveyStore with the action dispatcher
SurveyStore.dispatchToken =
Dispatcher.register(function(payload) {
  switch(payload.actionType) {
```

```
      ...
      case SurveyConstants.RECORD_SURVEY:
        SurveyStore.recordSurvey(payload.results);
        break;
      ...
    }
 });
```

Then we register our new `SurveySummaryStore`, placing the call to `waitFor` before we access the`SurveyStore` data.

```
SurveySummaryStore.dispatchToken =
Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    case SurveyConstants.RECORD_SURVEY:
      Dispatcher.waitFor(SurveyStore.dispatchToken);
      // At this point it's guaranteed the `SurveyStore` callback
has been run
      // and we can safely access its data to summarize it.
      SurveySummaryStore.summarize(SurveyStore.listSurveys());
      break;
    }
  }
});
```

By now you've seen how React can be used with many modern architectural patterns. From integrating React into existing projects using traditional MVC to starting fresh with a new pattern like Flux, React has proven to be quite adaptable.

Beyond adapting to many architectural patterns there are other libraries and tools that work well with React. Up next you can read about other tools in the family that can both support and enhance your React p

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
- Manual 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.

This is how the test for `TakeSurveyItem#getSurveyItemClass` looks:

```
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', {
  getAnswerClass:
jest.genMockFn().mockReturnValue(TestUtils.mockComponent)
});

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 () {
      // ...
    });
  });
});
```

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

Immutability-js

Immutable Data Structures are data structures that cant 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 <span class="o" style="box-sizing: border-box;
font-weight: bold;">=</span> <span class="nx" style="box-sizing: border-
box;">Immutable</span><span class="p" style="box-sizing: border-
box;">.</span><span class="nx" style="box-sizing: border-
box;">Map</span><span class="p " style="box-sizing: border-
box;">({description</span><span class="o" style="box-sizing: border-box;
font-weight: bold;">:<font color="#009999"> 'who is your favorite
superhero?'</font></span><span class="p " style="box-sizing: border-
box;">});
// 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 third object.
// Once again none of the original objects are mutated.</span>
<span class="kd" style="box-sizing: border-box; font-weight:
bold;">var</span> <span class="nx" style="box-sizing: border-
box;">title</span> <span class="o" style="box-sizing: border-box; font-
weight: bold;">=</span> <span class="p " style="box-sizing: border-
box;">{title: 'Question #1'</span><span class="p" style="box-sizing:
border-box;">};</span>
<span class="kd" style="box-sizing: border-box; font-weight:
bold;">var</span> <span class="nx " style="box-sizing: border-
box;">question3</span> <span class="o" style="box-sizing: border-box; font-
weight: bold;">=</span> <span class="nx" style="box-sizing: border-
box;">question</span><span class="p" style="box-sizing: border-
box;">.</span><span class="nx" style="box-sizing: border-
box;">merge</span><span class="p" style="box-sizing: border-
box;">(question2</span><span class="p" style="box-sizing: border-
box;">,</span> <span class="nx" style="box-sizing: border-
box;">title</span><span class="p" style="box-sizing: border-box;">);
question3.toObject(); // { title: 'Question #1', description: 'who is your
favorite comicbook hero' }</span>
```

Immutable.Vector

Use Immutable.Vector for Arrays:

```
<span class="kd" style="box-sizing: border-box; font-weight:
bold;">var</span> options <span class="o" style="box-sizing:
border-box; font-weight: bold;">=</span> <span class="nx"
style="box-sizing: border-box;">Immutable</span><span
class="p" style="box-sizing: border-box;">.</span><span
class="nx" style="box-sizing: border-
box;">Vector</span><span class="p" style="box-sizing:
border-box;">(<font
color="#009999">'Superman'</font></span><span class="p"
style="box-sizing: border-box;">,</span> <font
color="#009999">'Batman'</font><span class="p" style="box-
sizing: border-box;">);</span>

<span class="kd" style="box-sizing: border-box; font-weight:
bold;">var</span> <span class="nx" style="box-sizing:
border-box;">options2</span> <span class="o" style="box-
sizing: border-box; font-weight: bold;">=</span> <span
class="nx" style="box-sizing: border-
box;">options</span><span class="p" style="box-sizing:
border-box;">.</span><span class="nx" style="box-sizing:
border-box;">push</span><span class="p" style="box-sizing:
border-box;">(<font
color="#009999">'Spiderman'</font></span><span class="p"
style="box-sizing: border-box;">);

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 class="o"
style="box-sizing: border-box; font-weight: bold;">=</span>
<span class="nx" style="box-sizing: border-
box;">Immutable</span><span class="p" style="box-sizing:
border-box;">.</span><span class="nx" style="box-sizing:
border-box;">Vector</span><span class="p" style="box-sizing:
border-box;">(<font
color="#009999">'Superman'</font></span><span class="p"
style="box-sizing: border-box;">,</span> <font
color="#009999">'Batman'</font><span class="p" style="box-
sizing: border-box;">);

</span>var question = Immutable.Map({
```

```
    description: 'who is your favorite superhero?',
    options: options
  });
```

Immutable-js has many more facets; For more information on Immutable-js go to https://github.com/facebook/immutable-js

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.

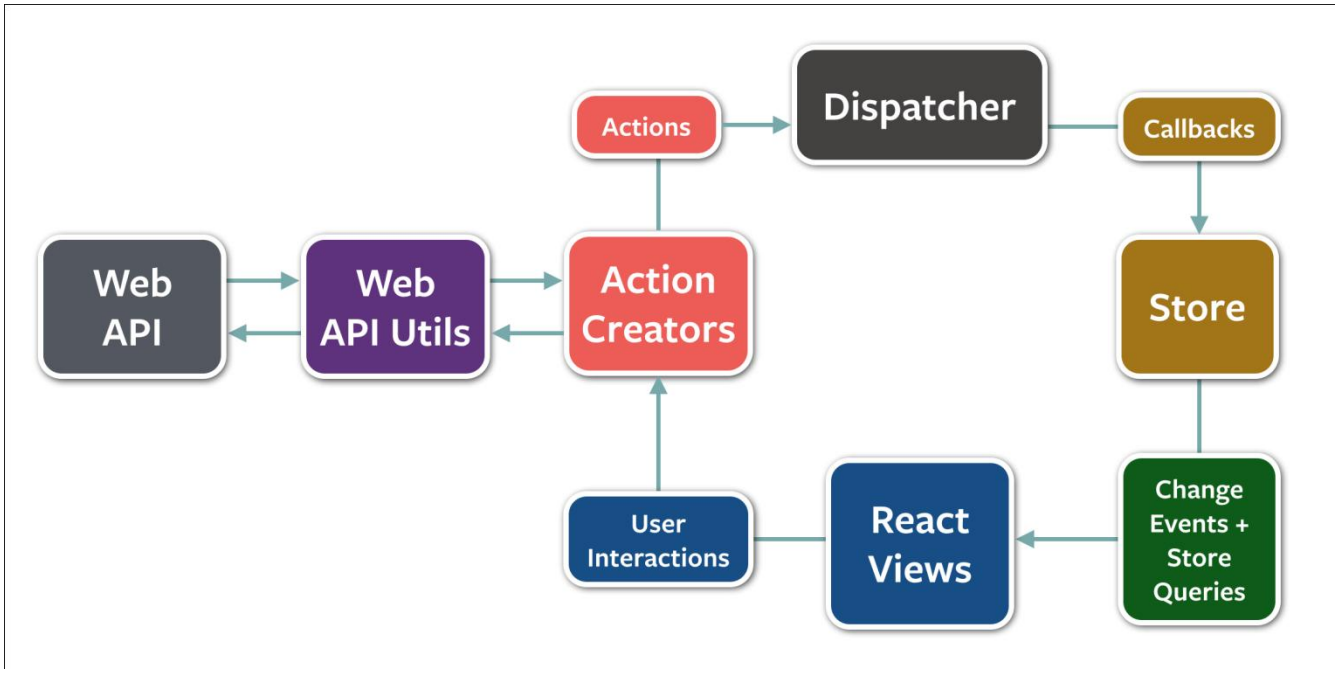


Figure 18-1. Flux enforces a uni-directional data flow

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 Bookand Instant Testing with CasperJS).

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 function for the test run. That integer argument is the number of assertions you will make in the test. OK, let's get started with our first test, `ROOT/test/functional/adding_a_survey.js` to test adding a survery 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.js
# 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, willNavigate=true,
isMainFrame=true
[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:

1. It's trying to load the homepage in the browser
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.js

# 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, willNavigate=true,
isMainFrame=true

[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 capserjs 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.js
# 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, willNavigate=true,
isMainFrame=true
[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)
PASS the title for the homepage is correct
[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.
```

If you want more information about the types of things CasperJS can do, read their documentation here: 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 correct");
  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 are inside of a new `then` callback. Why? Well, the answer is to handle the async nature of your 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 todo a few things:

1. load your nodejs server
2. start the server on part 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 stoping 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 recommend the following two resources to learn more:

- "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.htmls

Chapter 19. Uses

React is a powerful interactive UI rendering library, it provides a great way to handle data and user input. It encourages small components that are reusuable and easy to unit test. These are all great features that we can apply to other technologies than just the web.

In this chapter we'll look at how to use React for:

- Desktop applications
- Games
- Emails
- Charting

Desktop

With projects like atom-shell or node-webkit we can run a web application on the desktop. The Atom Editor from Github is built with atom-shell and also uses React.

Lets get our SurveyBuilder app working with atom-shell.

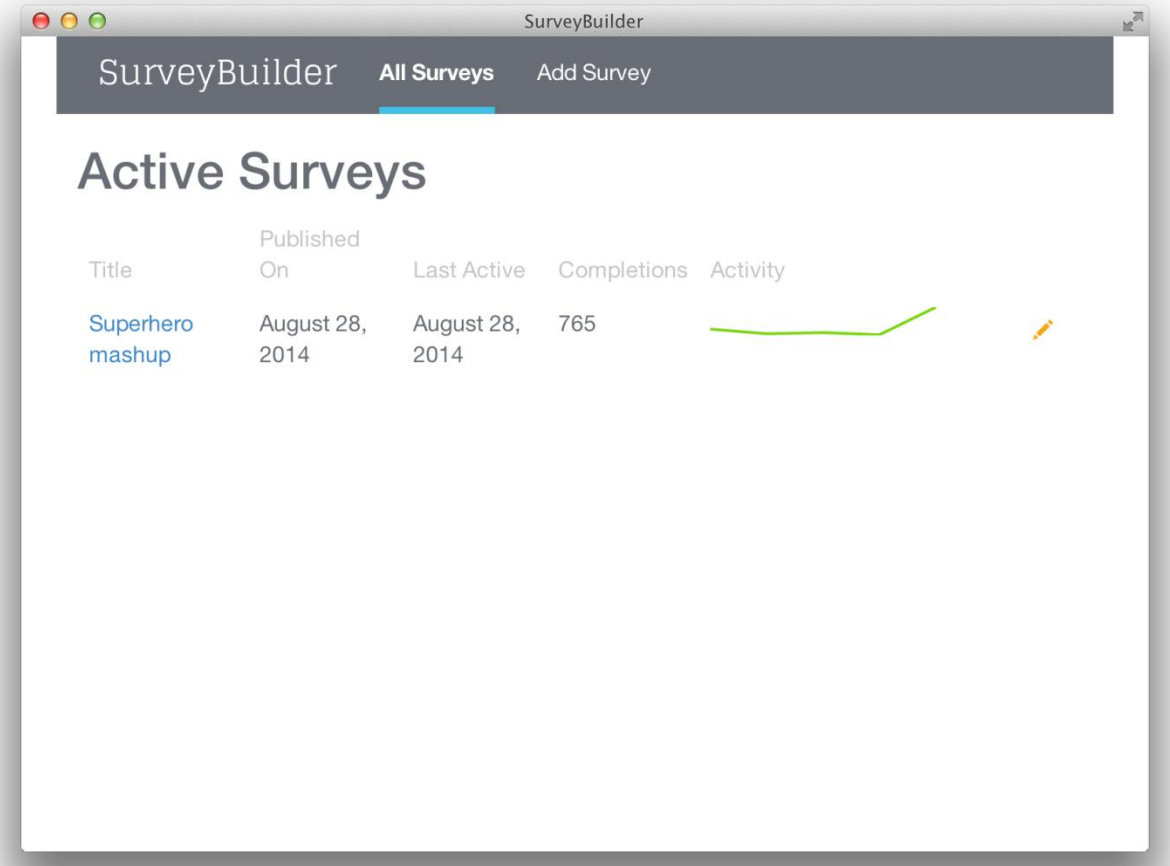
First we download and install from https://github.com/atom/atom-shell

Running the atom shell with this desktop script opens up the app in a window.

```
// desktop.js
```

```
var app = require('app');
var BrowserWindow = require('browser-window');
// require our SurveyBuilder server and start it.
var server = require('./server/server');
server.listen('8080');

// Report crashes to our server.
require('crash-reporter').start();

// Keep a global reference of the window object, if you
don't, the window will
// be closed automatically when the javascript object is
GCed.
var mainWindow = null;

// Quit when all windows are closed.
app.on('window-all-closed', function() {
  if (process.platform != 'darwin')
    app.quit();
});

// This method will be called when atom-shell has done
everything
// initialization and ready for creating browser windows.
app.on('ready', function() {
  // Create the browser window.
  mainWindow = new BrowserWindow({width: 800, height: 600});

  // and load the index.html of the app.
  //mainWindow.loadUrl('file://' + __dirname +
'/index.html');
```

```
  mainWindow.loadUrl('http://localhost:8080/');


  // Emitted when the window is closed.
  mainWindow.on('closed', function() {
    // Dereference the window object, usually you would
store windows
    // in an array if your app supports multi windows, this
is the time
    // when you should delete the corresponding element.
    mainWindow = null;
  });
});
```



With projects like atom-shell and node-webkit we can build desktop applications using the same technologies we use for the web. Just like for the web, React can help you build powerful interactive applications for the Desktop.

Email

Though React is optimized for building interactive UIs for the web, at it's core it renders HTML. This means we can get a lot of the same benefits we'd normally get from writing React application for something as terrible as writing HTML emails.

Building HTML emails requires a series of tables to render correctly in each email client. To write emails you need to turn back the clock a few years and write HTML as if it were 1999.

Successfully rendering emails in a range of email clients is no small feat. To build our design with React we will only touch on a number of challenges you will encounter when building emails, wether they are rendered using React or not.

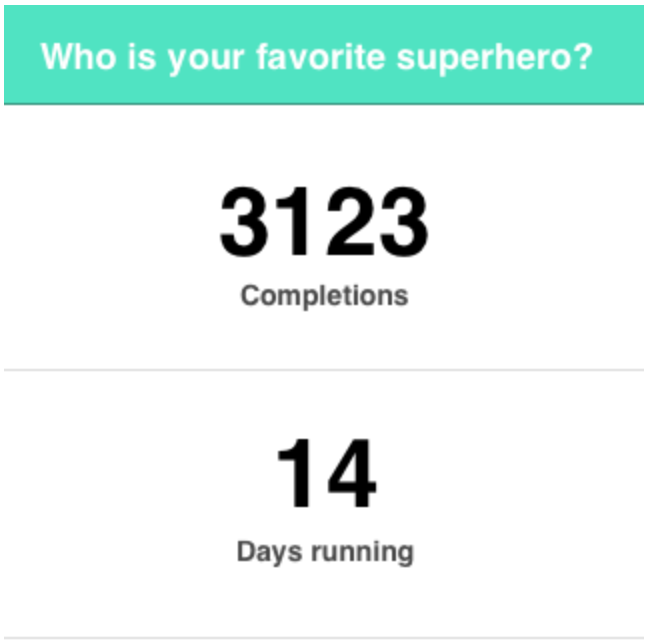
The core principle of rendering html for emails with React is `React.renderComponentToStaticMarkup`. This function returns an HTML string containing the full component tree, given 1 top level component. The only difference between `React.renderComponentToStaticMarkup` and `React.renderComponentToString` is that`React.renderComponentToStaticMarkup` doesn't create extra DOM attributes like `data-react-id` that React uses client side to keep track of the DOM. Since the email doesn't run client side in the browser - we have no need for those attributes.

Lets build an email with React given this design for desktop and mobile:

Who is your favorite superhero?

3123
Completions

14
Days running



To render our email we have made a small script that outputs HTML that can be used to send an email:

```
// render_email.js
var React = require('react');
var SurveyEmail = require('survey_email');
var survey = {};

console.log(
  React.renderComponentToStaticMarkup(
    <SurveyEmail survey={survey}/>
  )
```

```
  );
```

Lets get the core structure of SurveyEmail going

First lets build an Email component:

```
var Email = React.createClass({
  render: function () {
    return (
      <html>
        <body>
          {this.prop.children}
        </body>
      </html>
    );
  }
});
```

The `<SurveyEmail/>` component uses `<Email/>`

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },
  render: function () {
    var survey = this.props.survey;
    return (
      <Email>
        <h2>{survey.title}</h2>
      </Email>
    );
  }
});
```

Next, per the design we want to render 2 KPIs next to each other on desktop clients and stacked on a mobile device. Each KPI look similar in structure so they could share the same component:

```
var SurveyEmail = React.createClass({
  render: function () {
    return (
      <table className='kpi'>
        <tr>
          <td>{this.props.kpi}</td>
        </tr>
        <tr>
          <td>{this.props.label}</td>
        </tr>
      </table>
    );
  }
});
```

Let's add them to the `<SurveyEmail/>` component:

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },
  render: function () {
    var survey = this.props.survey;
    var completions = survey.activity.reduce(function (memo,
ac) {
      return memo + a;
    }, 0);
    var daysRunning = survey.activity.length;
```

```
    return (
      <Email>
       <h2>{survey.title}</h2>
       <KPI kpi={completions} label='Completions'/>
       <KPI kpi={daysRunning} label='Days running'/>
     </Email>
    );
  }
});
```

This stacks our KPIs, but our design had them next to each other for desktop. The challenge now is to both have this work for desktop and mobile, and to solve this there are a few gotchas we have to cover first.

Lets augment `<Email/>` with a way of adding a CSS file:

```
var fs = require('fs');
var Email = React.createClass({
  propTypes: {
    responsiveCSSFile: React.PropTypes.string
  },
  render: function () {
    var responsiveCSSFile = this.props.responsiveCSSFile;
    var styles;
    if (responsiveCSSFile) {
      styles =
<style>{fs.readFileSync(responsiveCSSFile)}</style>;
    }
    return (
      <html>
        <body>
          {styles}
          {this.prop.children}
```

```
        </body>
      </html>
    );
  }
});
```

The complete `<SurveyEmail/>` looks like this:

```
var SurveyEmail = React.createClass({
  propTypes: {
    survey: React.PropTypes.object.isRequired
  },

  render: function () {
    var survey = this.props.survey;
    var completions = survey.activity.reduce(function (memo,
ac) {
      return memo + a;
    }, 0);

    var daysRunning = survey.activity.length;

    return (
      <Email responsiveCSS='path/to/mobile.css'>
        <h2>{survey.title}</h2>
        <table className='for-desktop'>
          <tr>
            <td>
              <KPI kpi={completions} label='Completions'/>
            </td>
            <td>
```

```
              <KPI kpi={daysRunning} label='Days running'/>
            </td>
          </tr>
        </table>
        <div className='for-mobile'>
          <KPI kpi={completions} label='Completions'/>
          <KPI kpi={daysRunning} label='Days running'/>
        </div>
      </Email>
    );
  }
});
```

We grouped the Email into ‘for-desktop’ and ‘for-mobile’. Sadly we can’t use something like float: left in emails since that isn’t supported by most browsers and The HTML spec calls out the align and valign properties as being obsolete and therefor React doesn’t support those properties, thought they could have provided a similar implementation to floating 2 divs. Instead we are left with 2 groups which we can target with responsive style sheets to hide or show depending on the screen size.

Even thought we have to use tables it’s clear that using React for rendering emails gives us a lot of the same benefits from writing interactive UIs for a browser: Reusable, composable and testable components.

Charting

For our demo application we want to chart the number of completions of a survey pr day. We want this represented as a simple Sparkline in our table of surveys to at a glance show the attendance of our survey.

React has support for SVG tags and thus making a simple SVG becomes trivial.

To render a Sparkline, we need only a `<Path/>` with a set of instructions.

The complete example looks like this:

```
var Sparkline = React.createClass({
  propTypes: {
    points: React.PropTypes.arrayOf(React.PropTypes.number).isRequired
  },
  render: function () {
    var width = 200;
    var height = 20;
    var path = this.generatePath(width, height, this.props.points);

    return (
      <svg width={width} height={height}>
        <path d={path} stroke='#7ED321' strokeWidth='2' fill='none'/>
      </svg>
    );
  },

  generatePath: function (width, height, points) {
    var maxHeight = arrMax(points);
    var maxWidth = points.length;

    return points.map(function (p, i) {
      var xPct = i / maxWidth * 100;
      var x = (width / 100) * xPct;
      var yPct = 100 - (p / maxHeight * 100);
      var y = (height / 100) * yPct;

      if (i === 0) {
        return 'M0,' + y;
      }
      else {
        return 'L' + x + ',' + y;
      }
    }).join(' ');
  }
});
```

The Sparkline component above requires an array of numbers that represents the points. It then builds a simple svg with a path.

The interesting part is in the generatePath function which computes where each point should be rendered and returns an svg path description.

It returns a string like so “M0,30 L10,20 L20,50”. SVG paths translates this into drawing commands. Each command is separated by a blank space. “M0,30” means Move cursor to x0 and y30. Then “L10,20” means draw a line from the current cursor to x10 and y20 and so on.

It can be tedious to writing scale functions like this for larger charts, but its quite simple to drop in libraries like d3 and use the scale functions d3 provides instead of manual creating the path.