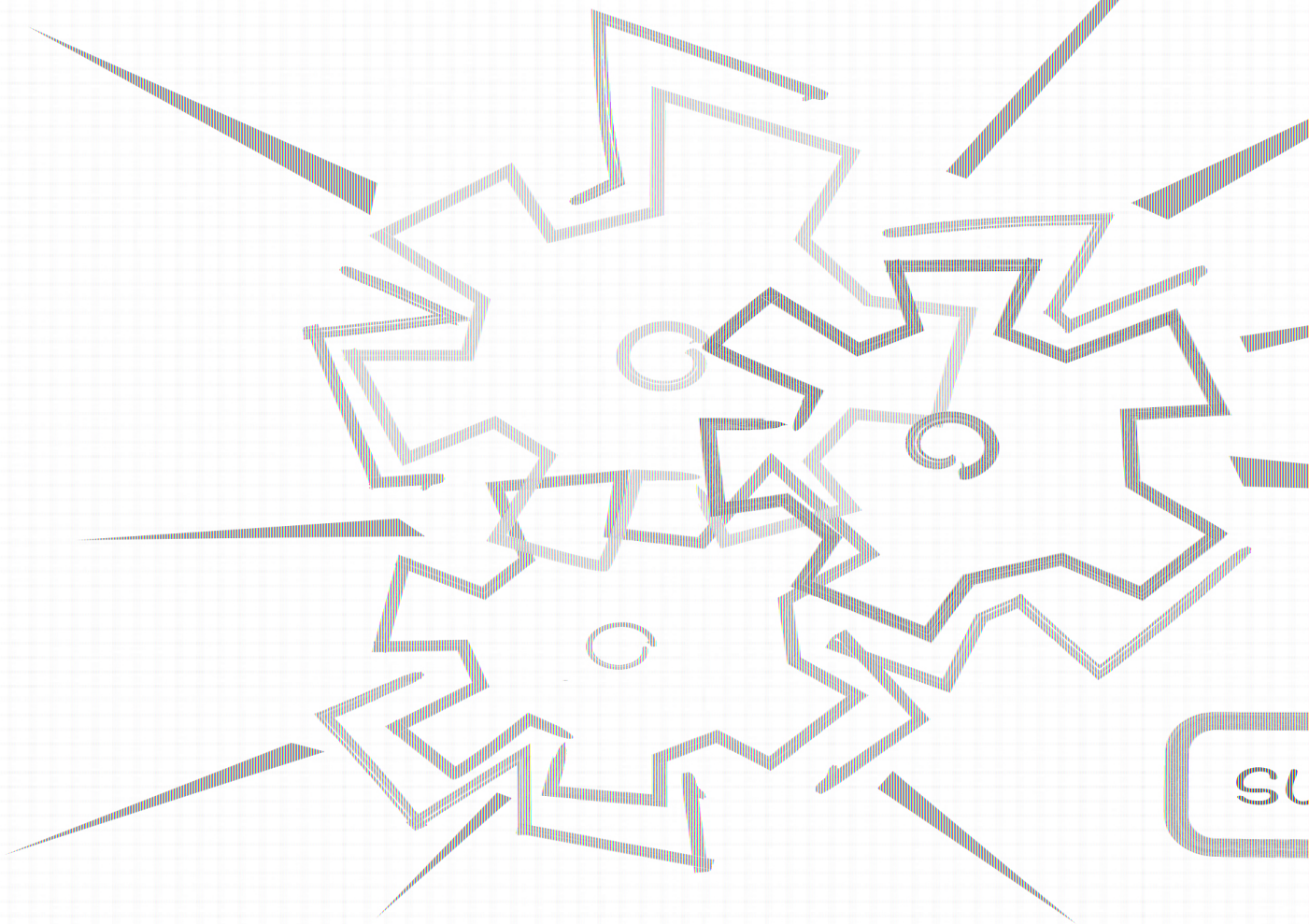
Developing a React.js Edge

Second Edition

THE JAVASCRIPT LIBRARY FOR USER INTERFACES



Richard Feldman, Frankie Bagnardi & Simon

BLEEDING

Developing a React Edge, Second Edition

The JavaScript Library for User Interfaces

Richard Feldman, Frankie Bagnardi, Simon Højberg, Jeremiah Hall

Developing a React Edge, Second Edition

Copyright (c) 2015 Bleeding Edge Press
All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.
This book expresses the authors views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Bleeding Edge Press, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.
ISBN 9781939902290
Published by: Bleeding Edge Press, Santa Rosa, CA 95404
Title: Developing a React Edge, Second Edition
Authors:Richard Feldman, Frankie Bagnardi, Simon Højberg, Jeremiah Hall
Editor: Troy Mott
Copy Editor: Christina Rudloff
Typesetter: Bob Herbstman
Cover Designer: Ellie Volckhausen
Website: bleedingedgepress.com

Preface

What is React and why should you use it?

React is a JavaScript library developed internally at Facebook and open sourced in 2013 for building interactive user interfaces for the web. It introduces a new way to deal with the Browser's DOM. Gone are the days of manually updating the DOM and laboriously keeping track of each piece of state that makes scalability and new feature development at best, a risky endeavor. Instead, React deals with the DOM in a very novel way. You declaratively define your user interface at any point in time. React removes the need to worry about which part of the DOM needs to update when data changes, and enables you to essentially re-render your entire application at any point in time with minimal DOM changes.

How this book helps

React introduces new and exciting concepts that challenge current practices. This book will enable you to navigate all of these concepts and help you understand why they are beneficial and can help you build scalable Single Page Applications (SPAs).

React focuses mainly on the “view” part of an application, and thus does not prescribe server communication or code organization. In this book we will cover the current best practices and complementary tools to help you build a complete application with React.

What do you need to know prior to reading the book?

To get the most out of this book you’ll need to be experienced with JavaScript and HTML. It’s beneficial if you have experience with writing SPAs (regardless of which framework like Backbone.js, AngularJS, or Ember.js), but it is not required.

Source code and sample application

Throughout this book we’ll be referencing bits of our example application: reddit clone. You can read the full source code at http://git.io/vlcpa and view an online demo at http://git.io/vlCUI

The writing process

This book was written as a focused virtual book sprint over the course of a month or two. This process helps create fresh and

current content, whereas conventional books often lag behind the coverage of cutting edge trends and technology.

In the second edition everything has been updated to React 0.14 and a new sample application has been created.

Authors

The book is written by a team of experienced and dedicated JavaScript developers:

Richard Feldman is the lead Front-End Engineer at NoRedInk, an education tech company in San Francisco. He is a functional programming enthusiast, a conference speaker, and the author of *seamless-immutable*, an open-source library that provides immutable data structures that are backward-compatible with normal JavaScript objects and arrays. He is @rtfeldman on both Twitter and GitHub.

Frankie Bagnardi is a Senior Front-end Developer creating user experiences for various clients. In his free time, he answers

questions on StackOverflow (FakeRainBrigand) and IRC (GreenJello), and enjoys small projects. You can reach him at f.bagnardi@gmail.com.

Simon Højberg is a Senior UI Engineer at Swipely in Providence, RI. He is the co-organizer of the Providence JS Meetup group and former JavaScript instructor at Startup Institute Boston. He spends his time building functional User Interfaces with JavaScript, and hacking on side projects like cssarrowplease.com. Simon tweets at @shojberg.

Jeremiah Hall is a Senior Software Engineer/Architect currently at OpenGov Inc. He is also the founder of Aspect Apps, building a journaling application that uses React Native and JavaScript for the UI. He can be found on Twitter @jeremiahrhall.

Chapter 1. Introduction to React

Background

In the early days of Web development, front-end code bases were small and JavaScript was underpowered. Thanks to years of browsers one-upping each other to push the envelope on JavaScript performance, today's web apps can deliver user experiences on par with those of native apps. As web apps continue to grow richer and more ambitious, scaling JavaScript code bases while maintaining strong performance is now a bigger challenge than ever.

Historically, many JavaScript libraries have prioritized performance or code organization—by making it easier to do high-performance DOM operations across browsers, or by offering organizational patterns that make code easier to scale. React has soared in popularity by making it easy to achieve both performance improvements and code scaling at the same time. This potent combination has made it one of the top ten most popular libraries across all of GitHub.

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

React is essentially a "state machine," helping you manage the complexity of state changing over time. It achieves this by having a very 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 due to the performance penalty of reading and updating the DOM. React has a very powerful rendering system that uses a virtual DOM, resulting in React only needing to update the DOM and not read from the DOM.

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 the process.

React uses a single event handler for the entire application and delegates all events to this event handler. This also gives React a performance boost, since having many event handlers can have significant performance penalties.

You can read the full source code of our example application used throughout this book, at http://git.io/vlcpa.

Book overview

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

Creating and composing components

The first seven chapters of the book are all about creating and composing React components. These chapters will provide an understanding of how to use React.

1) Introduction to React

This first chapter introduces React, covering the background and the book overview.

2) Using JSX and basic React components

JSX (JavaScript XML) is a way of writing declarative XML style syntax inside JavaScript. You will learn how to use JSX with React and how to build basic React Components. While not required to be used with React, most of the examples in this book, along with the example app, use JSX, since it is the recommended way to use React.

3) React component lifecycle

React is often creating and destroying components during the render process. React 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

It is important to know how data is passed down through the component tree and what data is safe to change. React has a very clear separation of `props` and `state`. This chapter will teach you what

props and state are and how to use props and state correctly in your React components.

5) Event handling

React implements event handling in a declarative manner. Event handling is an important part of any dynamic UI, and learning to master this is essential. Fortunately React makes event handling very simple.

6) Composing components

React 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) High order components and Mixins

This is a pattern that allows abstracting the 'how' of data dependencies. High order components wrap other components to provide data or functions via props. This chapter also covers mixins, which are still useful to use in certain circumstances.

Advanced topics

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

8) DOM manipulation

Even with all of the power of the virtual DOM available in React, sometimes you 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 React component lifecycle you can safely

access the DOM, and when to release your control on the DOM to avoid memory leaks.

9) Building forms with React

Using HTML form elements is one of the best ways of receiving input from your users. However, HTML form elements are very stateful. React provides a way to move most of the state from your form elements into your React 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 encourages the use of CSS for your animations. This chapter shows how React helps you leverage CSS for animating your React components.

11) Performance tuning your components

The virtual DOM in React gives you great performance right out of the box, but there is always room for improvement. React 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. Doing this can greatly improve the speed of your applications.

12) Server-side rendering

Many applications need SEO, fortunately React can be rendered to string in a non-browser environment like Node.js. Server-side rendering can also improve first page load times of your applications. Writing your application to support both server-side rendering and client-side rendering, however, 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.

Tooling for React

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

13) Development tools

As you write larger React applications you will start to require a way to automate packaging your code for deployment, and debugging your applications starts to get harder. In this chapter you will learn what tools are available to help you build and package your React applications and how to use the Google Chrome plugin available to visualize your React components, allowing for easier debugging.

14) Writing tests for React

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 components.

Working with React

The final chapters cover important aspects of working with React and different use cases you may not have thought about.

15) Architectural patterns

React provides only the "V" in "MVC", but it is very flexible in plugins in with other frameworks and systems. This chapter will help guide you in designing larger scale applications with React. We'll explore how our sample React app is structured and how this structure helps manage complexity as your project grows.

16) Immutability

React works well in conjunction with immutable data structures—that is, data structures that never change and are instantiated. In this chapter we cover the benefits and costs of immutability, and look at

three different libraries you can use to incorporate into your React application.

17) Other React uses

React is a powerful interactive UI rendering library, and it provides a great way to handle data and user input. In this chapter we'll look at how to use React for Desktop apps, games, emails and for charting.

Chapter 2. JSX

In React, components are used to separate concerns, not templates and display logic. When using React, you must embrace the idea that markup, and the code that generates it, are inherently tied together. This gives you all of the expressive power of JavaScript when building your markup, without having to resort to awkward or cumbersome templating languages.

React works well with an optional markup syntax, and is remarkably similar to HTML. But before proceeding, let's get one thing out of the way—for those put-off by the apparent awkwardness of markup in JavaScript, and for those not convinced of JSX's usefulness, consider the many benefits of this approach within React:

- It allows for familiar markup to define the element tree
- Provides a more semantic, easier to understand markup
- It is easier to visualize the structure of your application
- It abstracts the creation of React Elements
- It keeps markup and the code that generates it close at hand
- It's plain JavaScript

In this chapter we'll explore the many benefits of JSX, and how to use it, along with some of the gotchas separating 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.

In this book, the latest version of JavaScript, ES6 (or ES2015) will be used. You can find a good introduction to it here: https://babeljs.io/docs/learn-es2015/.

What is JSX?

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

For example, a function call in plain React to create a header might look like this.

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

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

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

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. Here we'll discuss the benefits of JSX and its purpose within an application, 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.

There are many benefits to using JSX, and they become more pertinent the larger your code base grows and the more complex your components become. Let's discuss some of these benefits now.

Familiarity

Many development teams include non-developers, from UI and 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 later), 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 offers you the benefit of declaring your component structure and information flow using an HTML-like syntax, knowing it will transform into plain JavaScript later.

JSX allows you to use all of the predefined HTML5 tag names alongside your own custom components within your application's markup. You'll read about defining custom components later, so here you'll simply see how JSX can make your JavaScript more readable.

As an example, let's consider a Divider element that renders a header on the left and a horizontal-divider that stretches to fill the right. The JSX for this divider looks like this:

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

After wrapping this in a `Divider` React component you can use it like you would any other HTML element, with the added benefit of markup with richer semantics.

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

Easy to visualize

JSX makes even small examples like these clearer and more concise. In larger projects with hundreds of components and deep markup trees, however, the benefit is magnified.

Here is the Divider component mentioned above. Notice that within the context of a JavaScript function the markup's intent is more clear and readable than the plain JavaScript version.

Here is the plain JavaScript:

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

Here is the JSX markup:

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

Most agree that the JSX markup is easier to understand and easier to debug.

Separation of concerns

Lastly, core to React, is the idea that markup and the code that generates it are inherently tied together. In React you do not separate the technologies of your application into a view and a template file. Rather, React encourages you to create discrete components, encapsulating all of the logic and the markup in one definition.

JSX provides a clean and concise way of keeping the markup of your component separate from the business logic. Not only does it provide a clean visual language for describing a component tree, but it makes your application easier to reason about.

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 assemble multiple components.

This section covers:

- Setting up a JavaScript file that contains JSX
- Walking you through the assembly of a component
- Discussing component ownership and the parent/child relationship

Let's look at each in turn.

Defining a custom component

Continuing with the aforementioned page divider, here again is the HTML we desire as output.

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

To express this HTML as a React component, just wrap it like the following so that the `render` function returns the markup (with some minor changes explained below).

```
class Divider extends React.Component {
  render() {
    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.

Dynamic values

JSX renders dynamic values between curly brackets `{...}` . The brackets signal a JavaScript context—anything you put between them will be evaluated and the results rendered as nodes in the markup.

For simple values, such as text or numbers, you can simply refer to the variable. You can render a dynamic h2 tag like this:

```
var text = 'Questions';
<h2>{text}</h2>

// <h2>Questions</h2>
```

For more complex logic you may wish to offload the calculations into a function. You can render the result of this function by calling it

within the brackets:

```
function dateToString(d) {
  return [
    d.getFullYear(),
    d.getMonth() + 1,
    d.getDate()
  ].join('-');
};

<h2>{dateToString(new Date())}</h2>

// <h2>2014-10-18</h2>
```

React will automatically evaluate arrays by rendering each item in the array as a node.

```
var text = ['hello', 'world'];
<h2>{text}</h2>

// <h2>helloworld</h2>
```

Often you may wish to render more than simple values. For example, you may want to render an array of data as `<li>` elements. This brings us to the idea of child nodes.

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, like this:

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

React captures all child nodes between the open and close tags in a special component prop, `this.props.children`. In this example, `this.props.children == "Questions"`. The actual structure of the children prop is undocumented, so don’t attempt to access its contents. Just pass it around as-is.

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.

```
class Divider extends React.Component {
  render() {
    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 (assuming the target is ES6).

```
class Divider extends React.Component {
  render() {
    return (
      React.createElement("div", {className: "divider"},
        React.createElement("h2", null, this.props.children),
        React.createElement("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 you 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.

You can read the full source code of our example application, a survey builder, at http://git.io/vlcpa.

Conditionals

In React, a component's markup and the logic that generate it are inherently tied together. This means that you have the full logical

power of JavaScript at your fingertips, such as with 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
- Use the `&&` operator

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

Using the ternary operator

```
...
render() {
  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() {
  return this.state.isComplete ? 'is-complete' : '';
}
```

```
render() {
  var isComplete = this.getIsComplete();
  return (
    <div className={isComplete}>...</div>
  );
}
...
```

Using a function call

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

Using the double-and (&&) operator

Because React does not output anything for null or false values you can use a boolean value and follow it with the desired output string. If the boolean evaluates to true then the following string will be used.

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

Non-DOM attributes

The following special attribute names exist in JSX:

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

Here we will explore them in more detail.

Keys

`key` is an optional unique identifier. Items and lists can change relative to their siblings, for example as the user performs a search or items are added, removed, or reordered in a list. When this happens, your component might be needlessly destroyed and recreated React may perform naive updates.

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, thus 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() {
  return <div>
    <input ref="myInput" ... />
  </div>;
}
...
```

And later you can access this ref by using `this.refs.myInput` anywhere in your component. For DOM components such as input, the ref is the DOM node in question. For composite components, the ref is the component instance. In practice, you will rarely use refs with React, they serve as an escape hatch from React's declarative nature.

For more detail see the discussion on parent/child relationships versus ownership in Chapter 6.

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

DANGEROUSLYSETINNERHTML

`dangerouslySetInnerHTML` may be changing soon, as described here:

https://github.com/facebook/react/issues/2134

https://github.com/facebook/react/pull/1515

Events

Event names are normalized across all browsers and are represented in camelCase. For example, `change` becomes `onChange`, and `click` becomes `onClick`. Capturing an event in JSX is as simple as assigning the property to a method on the component.

```
...
handleClick(event) {...}
render() {
  return (
```

```
      <div
        onClick={(e) => this.handleClick(e)}
      >
        ...
      </div>
    );
  }
  ...
```

Note that you use an arrow function here because it preserves this. Otherwise this would be undefined in `handleClick`. It also gives you easy control of which arguments are passed to the function.

For more details on the event system in React, reference Chapter 9.

Comments

JSX is JavaScript, so you can add plain JavaScript comments within your JSX markup. Comments can be added in two places:

1. As a child node of an element
2. Inline with a node's attributes

As a child node

Child-node comments are simply wrapped in curly brackets, and they can span multiple lines.

```
<div>
  {/* a comment about this input
      with multiple lines */}
  <input name="email" placeholder="Email Address" />
</div>
```

Inline with attributes

Inline comments can take two forms. First you can use a multi-line comment:

```
<div>
  <input
    /*
```

```
      a note about the input
    */
    name="email"
    placeholder="Email Address" />
</div>
```

Or you can use a single-line comment:

```
<div>
  <input
    name="email" // a single-line comment
    placeholder="Email Address" />
</div>
```

Special attributes

Because JSX transforms to plain JavaScript function calls, there are a few keywords you can't 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 you can access the class of an element using `elem.className`.

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

Styles

Lastly, let's examine 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 plain JavaScript. So JSX is not *necessary* for using React. However, JSX hides some complexity. If you're going to use React without it, you need to know the three parts to creating an element in React:

1. Defining the component class
2. Creating a factory for producing instances of the component class
3. Using the factory to create `ReactElement` instances

Creating React elements

Recall above how we defined a `Divider` component class. Here it has been renamed to `DividerClass` to clarify its purpose.

```
class DividerClass extends React.Component
  render() {
    return (
      React.createElement("div", {className: "divider"},
        React.createElement("h2", null, this.props.children),
        React.createElement("hr", null)
      )
    );
  }
});
```

To use this `DividerClass` without JSX, call `React.createElement` or `React.createFactory`.

To create the element directly simply call `createElement`.

```
var divider = React.createElement(DividerClass, null, 'Questions');
```

To create a factory, first use the `createFactory` function.

```
var Divider = React.createFactory(DividerClass);
```

Now that you have a factory function you're free to create a `ReactElement` from it.

```
var divider = Divider(null, 'Questions');
```

Further reading and references

Even if you don't like the idea of markup in your JavaScript, hopefully you can now appreciate how JSX offers a solution to the intimate relationship between your JavaScript and the markup it renders. Its growing popularity has earned JSX its own spec, offering a deep technical definition. There are a few tools to help you experiment with it if you're still uncertain or confused about how it works.

The official JSX spec

In September of 2014 Facebook released an official spec for JSX, stating its rationale for creating JSX, along with technical details of the syntax.

You can read more at http://facebook.github.io/jsx/.

In-browser experimenting

There are also a number of tools for experimenting with JSX. The React docs Getting Started page links to JSFiddle playgrounds with and without JSX.

http://facebook.github.io/react/docs/getting-started.html

For actual projects, the official tool is babel, which also provides nearly full ES6 support.

https://babeljs.io/

In the next chapter we will explore the component lifecycle in React.

[1] Retrieved from http://facebook.github.io/jsx/

Chapter 3. Component lifecycle

Throughout a component's lifecycle, as its props or state change, its DOM representation might change too. A component is a state machine; 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 the order of their appearance—first through instantiation, and then through the components life, and finally as the component is torn-down.

Lifecycle methods

React components have a minimal lifecycle API, offering only what you need without being overwhelming. Let's take a look at each method in the order they get called on your component.

Instantiation

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

- `constructor`
- `componentWillMount`
- `render`
- `componentDidMount`

Lifetime

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

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

Teardown & cleanup

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 in turn: instantiation, lifetime, and cleanup.

Instantiation

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

constructor(props, context)

The constructor allows you to set up instance properties and state for you component. A typical constructor looks like this:

```
class Foo extends React.Component {
  constructor(props){
    super(); // call the parent constructor, required
    this.state = {x: 'y'};
  }
  render(){ ... }
}
```

componentWillMount

Invoked immediately before the initial render. This is the last chance to affect the component state before the `render` method is called. This is just mentioned for completeness; it's a relic of `createClass`, which is now replaced by the constructor.

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 element.
- There can only be one top-level element (you cannot return an array of elements).
- It must be *pure*, meaning it does not change the state or 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 be made.

componentDidMount

After the render is successful and the actual DOM has been rendered, you can access it inside of `componentDidMount` via `act.findDOMNode(this)`, or by using a ref.

This is the lifecycle 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, since this is where you'd hook into.

For example, say you want to use the jQuery UI Autocomplete plugin on a React rendered input element. You can attach the plugin like this:

```
// A list of strings to autocomplete
var datasource = [...];

class MyComponent extends React.Component {
  render() {
    return (
      <input ... />
    );
  }
  componentDidMount() {
    $(React.findDOMNode(this)).autocomplete({
      sources: datasource
    });
  }
});
```

Note, the `componentDidMount` 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 application, 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, in the example application, when the user navigates between different boards, you can react to the change by fetching the relevant data from the server.

```
componentWillReceiveProps(nextProps) {
  var boardId = nextProps.params.boardId;
  if(boardId !== this.props.params.boardId) {
    Actions.State.changeBoard({type: 'board', name: boardId});
  }
}
```

You can read the full source code of our example application, a survey builder, at http://git.io/vlcpa.

---

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 the 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 you're 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. An example of using this will be covered later in this book.

componentWillUpdate

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

Note, you *can't* 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 & cleanup

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 that is 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. Failure to do so results in errors (if you `setState` on an unmounted component) and various kinds of leaks (memory, listeners, etc.).

Anti pattern: Calculated values as state

Given the possibility of creating state from `this.props` in the constructor, it's worth noting an anti-pattern here. You should be very concerned with maintaining a single source of truth. React's design makes duplicating the source of truth more obvious, which is 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.

```
class Day extends React.Component {
  constructor(props){
    super();
    // Anti-pattern. Calculated values should not be stored as state.
    this.state = {day: props.date.getDay()};
  }
  render(){
    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.

```
class Day extends React.Component {
  render(){
    return (
      <div>Day: {this.props.date.getDay()}</div>
    );
  }
```

Summary

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 its life.

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 application. How much does each child *really* need to know about? Who owns the application state? Which components, if any, need local state? This is the subject of our next chapter on 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 them. 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 of 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 outputs a virtual DOM representation.

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 comments = [{ author: 'Example', body: 'Hey' }];
<Comments comments={comments}/
```

You can read the full source code of our example application, a survey builder, at http://git.io/vlcpa.

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:

```
<Link to="/user/example">Example</Link>
```

It can also be set with the `{}` syntax, which is a regular JavaScript expression:

```
<Link to={'/user/' + comment.author}>{comment.author}</a>
```

It's possible to use the JSX spread syntax to use an object for props:

```
class Button extends React.Component {
  render() {
    // here we spread all of the props we're passed
    // but assert our version of className is used
    var className = ['Button', this.props.className].join(' ');
    return <button {...this.props} className={className} />;
  }
};
```

Props are useful for event handlers as well:

```
class SaveButton extends React.Component {
  render() {
    return (
      <Button
        onClick={() => this.handleClick()}
      >
        Save
      </Button>
    );
  }
  handleClick() {
    // ...
  }
};
```

Here we are passing the onClick prop to the Button component with a function that calls `this.handleClick`. Whenever the user clicks on the button, the `handleClick` method gets called.

PropTypes

React provides a way to validate your props, through a config object defined on your component. They also serve as a quick reference documentation.

```
class Post extends React.Component
  static propTypes = {
    data: PropTypes.shape({
      id: PropTypes.string,
      url: PropTypes.string,
      author: PropTypes.string,
      title: PropTypes.string,
      createdAt: PropTypes.number,
    }).isRequired,
  };
  // ...
};
```

If the requirements of the propTypes are not met when the component is instantiated, a `console.error` 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.

defaultProps

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

```
var Button = React.createClass({
  static propTypes = {
    which: PropTypes.oneOf(['primary', 'secondary', 'normal'])
  };
```

```
  static defaultProps = {
    which: 'normal'
  };
  // ...
};
```

State

Each component in 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. Note that this component includes the majority of what you need to know about React, so ×don't expect to understand it all right now.

```
class SimpleSelect extends React.Component {
  static propTypes = {
    options: PropTypes.arrayOf(PropTypes.shape({
      id: PropTypes.any,
      name: PropTypes.string,
    })).isRequired,
    value: PropTypes.any,
    valueText: PropTypes.any,
    onSelect: PropTypes.func,
  };

  constructor(){
    super();
    this.state = {open: false,};
  }

  toggleOpen(){
    this.setState({open: !this.state.open});
  }

  render(){
    return (
      <div>
        <Button onClick={() => this.toggleOpen()}>
          {this.props.valueText || 'Select an option'}
        </Button>
        {this.renderItems()}
      </div>
    );
  }

  renderItems(){
    if (!this.state.open) return null;
    return (
      <ul>
        {this.props.options.map((item, i) => {
          return (
            <li key={i} onClick={() => this.handleSelect(item)}>
```

```
                {item.name}
              </li>
                  );
          })}
        </ul>
          );

  }

  handleSelect(item){
    this.props.onSelect(item);
    this.setState({open: false});
  }
};
```

In the above example, state is used to track whether or not to show the options in the dropdown.

We set the initial state in the constructor with open set to false. When the dropdown button is clicked, you toggle the open state (`toggleOpen`). When an item is clicked, set the open state to false (`handleSelect`).

We can skim the code for `this.state.open` and see it's used in `renderItems`. If `this.state.open` is false, the options aren't rendered. Each time you `setState` or get new props from the parent, `render` (and `renderItems`) will run again. Causing changes in React is accomplished by impacting state or props rather than direct mutation.

Never update state directly via `this.state`. Instead, always go through the `this.setState` method.

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

What belongs in state and what belongs in props?

Don't store computed values or components in state, but instead focus on simple data that is directly required for the component to

function, like our open state for the dropdown. Without it you can't open or close the dropdown. This can also be the values of an input field, the user's mouse position, or the result of an AJAX request.

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

Stateless Functional Components

You can take this idea of a single source of truth and run with it to simplify your components. Functional components, introduced in React 0.14, are defined not classes with multiple methods, but rather as one single function. This function accepts props as an argument and returns the desired elements, much like a standalone `render` function in a traditional component.

Here is what our Day component looks like as a functional component:

```
function Day(props) {
  return (
    <div>Day: {props.day}</div>
  );
}
```

This style differs from traditional components in several ways. It is clearly more concise, but that is largely because several of a traditional component's features are no longer available. Specifically, functional components do not support the following features:

- state (see previous sections)
- lifecycle methods (see Chapter 3)
- refs or findDOMNode (see Chapter 8)

Functional components are not mere functions, however. They still support `propTypes` and `defaultProps`, which are used with the function's

props argument.

Given this feature set, functional components are best suited as drop-in replacements for components that have defined nothing more than a `render` method, with optional `propTypes` and `defaultProps` as well. Since typical React user interfaces are primarily comprised of components that fit that description, writing more components in this style can make a code base more concise without sacrificing clarity.

Summary

In this chapter you learned the following concepts:

1. Use props to pass data and settings through the component tree.
2. Never modify this.props inside of a component; consider props immutable.
3. Use props to for event handlers to communicate with child components.
4. Use state for storing simple view state like whether or not drop-down options are visible.
5. Never modify this.state directly, use this.setState instead.
6. When no local state, refs, or lifecycle methods are necessary, representing components as functional components can reduce verbosity and complexity.

We touched briefly on event handling in this chapter, which you'll learn about in greater detail in the next chapter.

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's 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 syntax. For example, in our example application, 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.

> You can read the full source code of our example application, a reddit clone, at http://git.io/vlcpa.

Note that although this code resembles the widely-discouraged practice of specifying event handlers in HTML using attributes such as `onclick`, it does not actually use the HTML `onclick` attribute under the hood. React uses this syntax only as a way to specify the handler, and internally takes care of efficiently managing event listeners as appropriate.

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.

Events and state

Suppose you have a markdown editor and you would like to display a live preview of the markdown.

In React this is done by listening to changes and updating state with the latest value.

```
class CommentEditor extends React.Component {
  constructor(){
    super();
    this.state = {text: ''};
  }
  render(){
    return (
      <div>
        <input
          value={this.state.text}
          onChange={(e) => this.setState({text: e.target.value})}
        />
      </div>
    );
  }
}
```

Rendering based on state

This is a basic two-way binding implementation. Because our data's in state we can render the markdown preview easily.

```
  render(){
    return (
      <div>
        <input
          value={this.state.text}
          onChange={(e) => this.setState({text: e.target.value})}
        />
        <Markdown content={this.state.text} />
      </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 way works just as well.

Updating state

Since updating a component's internal `state` causes the component to re-render, the markdown preview is always up to date. When we call `setState render` will run again, but it will read from the current value

of `this.state` . The Markdown component will also update, and the new content will be flushed to the DOM.

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 useful if you are using an immutable data structure to hold your state, but which is otherwise 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:

```
{
  title: "My Title",
  text: "Hello",
}
```

In this case, calling `this.setState({title: "Other Title"})` only affects the value of`this.state.title`, leaving `this.state.text` unaffected.

Calling `this.replaceState({title: "Other Title"})` will instead replace the entire state object with the new object `{title: "Fantastic Survey 2.0"}`, erasing `this.state.text` altogether. This will likely break the render function, which would expect `this.state.text` to be a string instead of `undefined`.

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 it might lead to surprising results the next time you call `setState`.

State isn't updating!

It's also worth noting that `setState` is async. So if your state is `{x: 1}` and you run this code, the console will show 1.

```
this.setState({x: 2})
console.log(this.state.x);
```

If you need to do something after state has been updated, render has been called, and changes have been flushed to the DOM, you can pass a callback as `setState`'s second argument.

```
this.setState({x: 2}, () => {
  console.log(this.state.x); // 2
})
```

This also means that multiple state updates in a single tick can cause issues.

```
this.state.xs // []
this.setState({xs: this.state.xs.concat([1])}
this.setState({xs: this.state.xs.concat([2])}
```

Eventually the state will be `[2]` instead of the desired `[1, 2]`.

You can avoid this problem by using the atomic `setState` variant. Normally you'd use an expression arrow function, but let's write it verbosely for clarity.

```
this.state.xs // []
this.setState((state) => {
  var xs = state.xs.concat([1]);
  return {xs: xs};
});
this.setState((state) => {
  var xs = state.xs.concat([2]);
  return {xs: xs};
});
```

Note, you wan to use the `state` argument rather than `this.state`. React queues these functions and calls each with the updated state of the previous. Often you know two updates won't occur in a single tick, or you may not care about the first one being hidden, so this feature is rarely used in practice.

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 `CommentEditor` class from the example application.

```
onChange={(e) => {
  this.props.onChange({text: e.target.value, type: 'comment'})
}}
```

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 event handler takes an event object, from which it extracts the current value of the input 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 of 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 built with only elements.

Since the entirety of a React application is built 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 application, so you can think of a React component as an extension of HTML.

In this chapter we'll be using code not found in the reddit clone sample application because a CRUD (create read update delete) application illustrates it better.

Extending HTML

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

React favors composition over inheritance, which means you combine small, simple components and data objects into larger, more complex components. With EcmaScript 6 you inherit from `React.Component`, but it does not encourage deeper levels of inheritance than that. Instead, building re-useable components that can be composed is encouraged.

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.

You can read the full source code of our example application, a reddit clone, at http://git.io/vlcpa.

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 some basic elements to help, namely the radio type inputs and input groups. Thinking of it from the top-down, the component hierarchy looks something like this:

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

You can think of these 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.

Assembling the HTML

Let's start by assembling the components from the bottom-up. The DOM components (starting with a lowercase letter) are provided by React. 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 its scope to behave like a radio button. Let's name it `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.

```
class AnswerRadioInput extends React.Component {
  render() {
    return (
      <div className="radio">
        <label>
          <input type="radio" />
          Label Text
        </label>
      </div>
    );
  }
}
```

Add dynamic properties

Nothing about our `input` is dynamic yet, so 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 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.

```
class AnswerRadioInput extends React.Component {
    ...
}

AnswerRadioInput.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 `defaultProps` static member. These values will be applied to each new instance when the parent component does not provide their value.

```
class AnswerRadioInput extends React.Component {
  ...
}

AnswerRadioInput.PropTypes = {...};
AnswerRadioInput.defaultProps = {
```

```
  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, let's define the initial state.

This happens in the `constructor` of the React component. We simply set the `this.state` hash with the default state.

Notice that we also call `super(props)` to use the default prop behavior.

```
class AnswerRadioInput extends React.Component {
  constructor(props) {
    super(props);
    var id = props.id ? props.id : uniqueId('radio-');
    this.state = {
      checked: !!props.checked,
      id: id,
      name: id
    };

  }
  ...
}

AnswerRadioInput.PropTypes = {...};
AnswerRadioInput.defaultProps = {...};
```

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

```
class AnswerRadioInput = React.createClass({

  constructor(props) {...}

  render() {
    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>
    );
  }
}

AnswerRadioInput.PropTypes = {...};
AnswerRadioInput.defaultProps = {...};
```

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.

```
class AnswerMultipleChoiceQuestion extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      id: uniqueId('multiple-choice-'),
      value:props.value
    };
  }
  render() {
    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>
    );
  }
}

AnswerMultipleChoiceQuestion.propTypes = {
  value: React.PropTypes.string,
  choices: React.PropTypes.array.isRequired,
  onCompleted: React.PropTypes.func.isRequired
};
```

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

```
class AnswerMultipleChoiceQuestion = React.createClass({

  ...
  renderChoices() {
    return this.props.choices.map((choice, i) => {
      return (
         <AnswerRadioInput
          id={"choice-" + i}
          name={this.state.id}
          label={choice}
          value={choice}
          checked={this.state.value === choice}
         />
      );
    });
  }

  render() {
    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 clearer. 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. This brings us to the parent/child relationship.

Parent / child relationship

At this point you should be able to render a form to the screen with our example, 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 when needed.

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

When we use EcmaScript 6 classes, methods aren't auto bound to the instance by default. You can set up this wiring in the `constructor` like below, or when you pass it in as the `onChanged` prop.

```
class AnswerMultipleChoiceQuestion extends React.Component {
  constructor(props) {
    ...
```

```
    this.handleChanged = this.handleChanged.bind(this);
  }

  handleChanged(value) {
    this.setState({value: value});
    this.props.onCompleted(value);
  }

  renderChoices() {
    return this.props.choices.map((choice, i) => {

        return (
         <AnswerRadioInput
           id={"choice-" + i}
           name={this.state.id}
             label={choice}
           value={choice}
          checked={this.state.value === choice}
           onChanged={this.handleChanged}
         />
       );
     });
  }
  ...
}
```

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.

```
class AnswerRadioInput extends React.Component {
  constructor(props) {
     ...
    this.handleChanged = this.handleChanged.bind(this);
  }

  handleChanged(e) {
    var checked = e.target.checked;
    this.setState({checked: checked});
    if (checked) {
      this.props.onChanged(this.props.value);
    }
  }

  render() {
    return (
      <div className="radio">
        <label htmlFor={this.state.id}>
          <input type="radio"
            ...
            onChange={this.handleChanged} />
          {this.props.label}
        </label>
      </div>
    );
  }
}

AnswerRadioInput.propTypes: {
  ...
  onChanged: React.PropTypes.func.isRequired
};
```

Wrap up

You've now 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 use components they become more specific and semantically meaningful. Thus, React takes generic inputs:

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

This transforms them into something a bit more meaningful:

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

You finally end 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. High Order Components and Mixins

There are two special kinds of abstraction with React. High order components are components that take a component as an argument and return another component.

Previously, the role of high order components was solved with mixins. A few things can still only be done with mixins, which we'll cover below.

Simple Example

Let's write a high order function that passes a random number to the child via props. Props are how high order components talk to child components. You also need to make sure you pass any extra props down using the `{...this.props}` syntax.

```
function providesRandomNumber(Component) {
  return class RandomNumberProvider extends React.Component {
    render(){
      return <Component {...this.props} randomNumber={4} />;
    }
  }
}
```

We can then use this function to wrap a component.

```
var MyComponent = providesRandomNumber(
  class MyComponent extends React.Component {
    render(){
      return (
        <div>
          The random number is {this.props.randomNumber}
        </div>
      );
    }
  }
);
```

This is a bit cleaner with the [es7 decorators] proposal.

```
@providesRandomNumber
class MyComponent extends React.Component {
  render(){
    return (
      <div>
        The random number is {this.props.randomNumber}
      </div>
    );
  }
}
```

Tips and Tricks

Often a high order component takes options. You can curry it, which means it can be called like `hoc(Component)`, `hoc({}, Component)`, or `hoc({})(Component)`.

```
function hoc(options, Component){
  if (typeof options === 'function') return hoc({}, options);
  if (arguments.length < 2) return hoc.bind(null, options || {});

  return class ...
}
```

If you have a mixin, you can create a high order component from the mixin and pass things down as props.

```
import {History} from 'react-router';

function providesRouter(Component){
  return React.createClass({
    mixins: [History],

    render(){
      return <Component
        {...this.props}
        router={{
          pushState: this.pushState,
        }}
      />
    }
  });
}
```

As a general rule, static properties should be passed down to the component you create.

```
function providesFoo(Component){
  return Object.assign(class Foo extends React.Component {
    render(){
      ...
    }
  }, Component);
}
```

Common Use Cases

High order components are typically used for global events, binding to flux stores, timers, and any other imperative APIs that you want to use declaratively.

Summary

High order components 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.

Even if you only intend to use a high order component in a single component, it allows you to describe a certain behavior or role, and provide that to the component. They reduce the amount of code you need to read before understanding a component, and allow you to do ugly things (such as managing __intervals) without making your component ugly.

When reading the next chapter about DOM, think about the behaviors and roles you could pull out of the components, or even try for yourself.

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 allows you to do this.

```
class DoodleArea extends React.Component
  render() {
    return <canvas ref="mainCanvas" />;
  }
}
```

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

Note, you can't reliably access the `<canvas>` node directly 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.

You can reliably access the `<canvas>` after your 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 not go well!
    doSomethingWith(this.refs.mainCanvas);

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

  componentDidMount: function() {
    doSomethingWith(this.refs.mainCanvas);
    // 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 use `refs`. 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;
    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.

Finding DOM Nodes by Component

Although refs are the preferred way of accessing underlying DOM nodes, sometimes you may want to write some DOM-altering logic that works with any component regardless of its refs, or perhaps you need to look one up the underlying DOM node for an imported third-party component that does not set any refs.

In cases like these, ReactDOM.findDOMNode is the answer.

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

  handleKeyDown: function() {
    // NOTE: It would be preferable to use a ref here
    // instead of using findDOMNode.
    var editor = ReactDOM.findDOMNode(this);
    var html = editor.innerHTML;

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

Note that the above is not the preferred way of doing this. It would be better to use a ref for this as we did in the previous section, as in that case the logic would not break if the `render` function were modified to introduce something like a wrapper element. `findDOMNode` should really only be used when a ref either will not work or would be excessively burdensome to introduce.

Keep in mind that although `refs` and `findDOMNode` are powerful, they should only be used when there is no other way to achieve the functionality you need. Using them hinders React's ability to optimize performance and increases the complexity of your application, so you should only reach for them when conventional techniques fall short of the capabilities you need.

Incorporating non-React libraries

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

Suppose you want to use an autocomplete library that 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 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.

```
class AutocompleteCities extends React.Component {
  render() {
    return <div ref="autocompleteTarget" />
  }

  static defaultProps = {
    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 that connects the two implementations via the underlying DOM node of the `autocompleteTarget` child component.

```
class AutocompleteCities extends React.Component {
  render() {
    return <div id="cities" ref="autocompleteTarget" />
  }

  static defaultProps = {...};

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

  componentDidMount() {
    autocomplete({
      target: this.refs.autocompleteTarget,
      data: this.props.data,
      events: {
        select: this.handleSelect
      }
    });
  }
};
```

Note that `componentDidMount` will only be called once per DOM node. As such, you do not need to worry whether calling `autocomplete` (in this example) twice on the same node will have any undesired effects.

That said, remember that this component may be removed and re-rendered later to a different DOM node, which can lead to memory leaks or other problems if your `componentDidMount` has effects that can survive the DOM node's removal. If this is a concern, make sure to specify a `componentWillUnmount` handler to have the component clean up after itself when its DOM node is going away.

Overreaching plugins

In our autocomplete example, we assume autocomplete is a well-behaved plugin that only modifies its children. In reality, this is often not the case.

To deal with these plugins, you need to hide them from React, to avoid errors about the DOM being unexpectedly mutated. You may also

need to do additional cleanup.

In this example, we're dealing with a fictional jQuery plugin that emits custom events, and modifies the element it's attached to. If you have a very bad plugin that modifies parent elements, there's nothing you can do, and it's incompatible with React. The best course of action is to find another plugin or modify its source.

The way to protect React from this overreaching plugin is to manage its DOM node completely by yourself. React thinks the component renders a single `div` with no children or props.

```
class SuperSelect extends React.Component {
  render: function(){
    return <div ref="holder"/>;
  }
};
```

Let's do the dirty work of setting things up in `componentDidMount`.

```
class SuperSelect extends React.Component {
  render(){
    return <div ref="holder" />;
  }

  componentDidMount(){
    var el = document.createElement('div');
    this.refs.holder.appendChild(el);
    $(el).superSelect(this.props);
    $(el).on('superSelect', this.handleSuperSelectChange);
  }


  handleSuperSelectChange() {
    // Put handler logic in here as appropriate...
  }
};
```

Now you have a `div` inside the component's rendered div that you are managing. This means that you need to be responsible and clean up.

```
  componentWillUnmount(){
    // remove superSelect's listeners and our own
    $(this.refs.holder).children().off();

    // remove the node from the DOM
    $(this.refs.holder).empty()
  }
```

In addition to the clean up, check the plugin's docs for any additional requirements to clean up the node. It may set global event listeners, timers, or start AJAX requests, which need to be terminated.

There is one more step in this process. You need to handle updates. This can happen in one of two ways: simulating an unmount and remount, or using the plugin's update API. The former is more reliable, and the latter is more performant and clean.

This is the unmount/remount solution.

```
  componentDidUpdate(){
    this.componentWillUnmount();
    this.componentDidMount();
  }
```

And this is a hypothetical alternative.

```
  componentWillReceiveProps(nextProps){
    $(this.refs.holder).children().superSelect('update', nextProps);
  }
};
```

Wrapping other libraries and plugins ranges in difficulty depending upon how many variables there are. Sometimes it's a simple jQuery plugin, and sometimes it's a rich text editor with its own plugins. For some of the simpler plugins, the best answer is to rewrite it as a React component in the same time it'd take to wrap it, and for others that is completely unfeasible.

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 `findDOMNode` 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 to create and manage forms using React.

Chapter 9. Forms

Forms are an essential part of any application that requires even modest input from users. Traditionally, in Single Page Applications, 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 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 components. Given the same props and state, any React component should render exactly the same every time. Forms are no exception.

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

This chapter also covers:

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

In the last chapter you learned how to access the DOM of your React components. React helps you move the state out of the DOM

into your components. Even with this help, you will still need to access the DOM for some complex form components.

You can read the full source code of our example application, at http://git.io/vlcpa.

Uncontrolled components

In most non-trivial forms, you will *not* want to use 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.

In HTML, forms work differently to React 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 component.

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

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

```
//http://jsfiddle.net/pmsy5y2u/
class MyForm extends React.Component {
  render() {
    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 its 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 DOM node's value.

As you saw in Chapter 8, `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/opfktus4/
class MyForm extends React.Component {
  submitHandler(event) {
    event.preventDefault();
    //access the input by it's ref
    var helloTo = this.refs.helloTo.value;
    alert(helloTo);
  }
  render() {
    return (
      <form onSubmit={(e) => this.submitHandler(e)}>
        <input
          ref="helloTo"
          type="text"
          defaultValue="Hello World!" />
        <br />
        <button type="submit">Speak</button>
      </form>
    );
  }
};
```

Uncontrolled components are rarely good to use. They don't allow you to render based on the current value of inputs, and your code is written in a DOM-centric way rather than in a data-centric way. They also make composition more difficult because there's no standard way to get the value of an input in a composite component.

Controlled components

Controlled components follow the same pattern for other React components. The state of the form component is Controlled by your

React component, with it's value being stored in your React component's 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/1a8xr2z6/
class MyForm extends React.Component {
  constructor() {
    super();
    this.state = {
      helloTo: "Hello World!"
    };
  }
  handleChange(value) {
    this.setState({
      helloTo: value
    });
  }
  submitHandler(event) {
    event.preventDefault();
    alert(this.state.helloTo);
  }
  render() {
    return (
      <form onSubmit={(e) => this.submitHandler(e)}>
        <input
          type="text"
          value={this.state.helloTo}
          onChange={(e) => this.handleChange(e.target.value)} />
        <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.

- `constructor` now sets the `defaultValue`
- `<input/>` value is set during render

- `onChange` of the `<input/>` value the change handler is invoked
- change handler updates the state
- `<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 characters to uppercase as they are being entered.

```
handleChange(value) {
  this.setState({
    helloTo: 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 is intercepting the native browser change event. Then this component re-renders the input after `setState` has been called. React 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 auto complete options.
- Update other UI elements with the input value.

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. 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 node = syntheticEvent.target;
  var newValue = node.value;
}
```

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

For elements that have children you need to use `event.currentTarget`.

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. For compatibility with ES3 (IE8) reserved words aren't used as properties.

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

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

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

Textarea and Select

React 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 supports multi select. In order to use multi select you must pass an array to `value` and `defaultValue`.

```
//Uncontrolled
<select multiple="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/yddy2ep0/
class MyForm extends React.Component {
  constructor() {
    super();
    this.state = {
      options: ["B"]
    };
  }
  handleChange(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(event) {
    event.preventDefault();
    alert(this.state.options);
  },
  render: function () {
    return (
      <form onSubmit={(e) => this.submitHandler(e)}>
        <select multiple="true"
            value={this.state.options}
            onChange={(e) => this.handleChange(e)}
        >
          <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
class MyForm extends React.Component {
  submitHandler(event) {
    event.preventDefault();
    alert(this.refs.mycheckbox.checked);
  }
  render() {
    return (
      <form onSubmit={(e) => this.submitHandler(e)}>
        <input
          ref="mycheckbox"
          type="checkbox"
          value="A"
          defaultChecked="true" />
        <br />
        <button type="submit">Speak</button>
      </form>
    );
  }
};

// Controlled
var MyForm = React.createClass({
  constructor() {
    super();
    this.state = {
      checked: true
    };
  }
  handleChange(event) {
    this.setState({
      checked: event.target.checked
    });
  }
  submitHandler(event) {
    event.preventDefault();
    alert(this.state.checked);
  }
  render() {
    return (
      <form onSubmit={(e) => this.submitHandler(e)}>
        <input
          type="checkbox"
          value="A"
```

```
          checked={this.state.checked}
          onChange={this.handleChange}
        />
        <br />
        <button type="submit">Speak</button>
      </form>
    );
  }
};
```

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

Names on form elements

Names carry less importance on from elements in React 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.

However, names are a crucial aspect of form components.

- Names allow third party form serializers to still work within React.
- 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 behavior 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.

```
class MyForm extends React.Component {
  constructor(){
    super();
    this.state = {
      radio: "B"
    };
  }
  handleChange(event) {
    this.setState({
      radio: event.target.value
    });
  }
  submitHandler(event) {
    event.preventDefault();
    alert(this.state.radio);
  }
  render() {
    return (
      <form onSubmit={(e) => this.submitHandler(e)}>
        <input
          type="radio"
          value="A"
          checked={this.state.radio == "A"}
          onChange={(e) => this.handleChange(e)} /> A
        <br />
        <input
          type="radio"
          value="B"
          checked={this.state.radio == "B"}
          onChange={(e) => this.handleChange(e)} /> B
        <br />
        <input
          type="radio"
          value="C"
          checked={this.state.radio == "C"}
          onChange={(e) => this.handleChange(e)} /> 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.

An example: arrow functions.

```
class MyForm extends React.Component {
  constructor() {
    super();
```

```
    this.state = {
      given_name: "",
      family_name: ""
    };
  }
  handleChange(name, event) {
    var newState = {};
    newState[name] = event.target.value;
    this.setState(newState);

    // or the es6 version
    this.setState({
      [name]: event.target.value,
    });
  }
  submitHandler(event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.given_name,
      this.state.family_name
    ];
    alert(words.join(" "));
  }
  render() {
    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={(e) => this.handleChange('given_name', e)}/>
        <br />
        <label htmlFor="family_name">Family Name:</label>
        <br />
        <input
          type="text"
          name="family_name"
          value={this.state.family_name}
          onChange={(e) => this.handleChange('family_name', e)}/>
        <br />
        <button type="submit">Speak</button>
      </form>
    );
  }
};
```

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

```
//http://jsfiddle.net/q3g0sk84/
class MyForm extends React.Component {
  constructor() {
    super();
    this.state = {
      givenName: "",
      familyName: "",
```

```
    };
  }
  handleChange(event) {
    this.setState({
      [event.target.name]: event.target.value
    });
  }
  submitHandler(event) {
    event.preventDefault();
    var words = [
      "Hi",
      this.state.givenName,
      this.state.familyName,
    ];
    alert(words.join(" "));
  }
  render() {
    return (
      <form onSubmit={this.submitHandler}>
        <label htmlFor="givenName">Given Name:</label>
        <br />
        <input
          type="text"
          name="givenName"
          value={this.state.givenName}
          onChange={(e) => this.handleChange(e)}/>
        <br />
        <label htmlFor="familyName">Family Name:</label>
        <br />
        <input
          type="text"
          name="familyName"
          value={this.state.familyName}
          onChange={this.handleChange}/>
        <br />
        <button type="submit">Speak</button>
      </form>
    );
  }
};
```

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

predictability 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 select component. We won't implement multi select functionality, since radio components don't support multi select.

```
class Radio extends React.Component {
  static propTypes = {
    onChange: React.PropTypes.func
  };

  constructor(){
    super();
    this.state = {
      value: this.props.defaultValue
    };
  }
  handleChange(event) {
    if (this.props.onChange) {
      this.props.onChange(event);
    }
    this.setState({
      value: event.target.value
    });
  }
  render() {
    var value = this.props.value || this.state.value;

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

    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 store the `defaultValue` in state.

Each time the component renders, it creates new labels and radios based on the options that were supplied as children to the component. Be sure that the dynamic children have consistent keys rendered. This ensures that React will keep your `<input/>`'s in the DOM and maintain current focus when using keyboard controls.

Then set the value, the name, and the checked state. Attach your `onChange` handler, then render the new children.

```
//Uncontrolled
//http://jsfiddle.net/moyfLkfv/
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>;
  }
});
```

When storing the value in the state of your component, you don't need to access the DOMNode to know what the current value is. You can access the state directly.

```
//Controlled
//http://jsfiddle.net/cwabLksg/
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 to 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. The merits of this are discussed more in the following section.

Since React forms are not always rendered at browser load, the auto focus for form inputs needs to operate a little differently. React has implemented `autoFocus` so when the component is first mounted, if no other form input has focus, React 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 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 behavior 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.

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. The ideal time to give feedback for validation errors is on `blur` of the input.

It's 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 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. See Chapter 10 to learn how to leverage this in React.

Be fast

React 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. Long running transitions may frustrate your users as they are forced to wait for the transition to complete before continuing to use your application.

Other factors outside your application can also alter how fast your application performs, like long running AJAX calls and poor network performance. 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. In these cases it is more important to give feedback to your users about the state of their request.

It is important to remember that speed is relative. It is a perception of the user. It is more important to appear fast than to be fast. For 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.

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

With this in mind, you are left with two options. You can either conform to the default behaviors of the platform, or you can radically change your user interface so it no longer resembles the 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 overlooked by developers and designers when creating user interfaces. It is important to keep your users in mind when considering all aspects of your user interfaces. As already covered, your users have a predefined expectation of how things work, which 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 a 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 devices 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 are visually impaired. Screen readers are the eyes for visually impaired people.

HTML5 has an Accessible Rich Internet Applications (ARIA) specification that provides a way to add the missing semantics

needed by assistive technologies such as screen readers. This allows you to hint on the `role` of your UI components, and hide or show elements if a screen reader is active and much more.

There are some great tools to help with accessibility in your applications like the Accessibility Developer Tools for Google Chrome.

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, so 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. This is achieved by using standard names on your inputs.

Autocomplete can help guide your users as they are entering information, this ties in with giving constant feedback. When searching for a movie, for example, 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. For 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, and 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 a 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.

Summary

React helps you manage the state of your forms by bringing the state management out of the DOM into your components. This allows you to have tighter control over how the form elements operate and create complex components to use in your applications.

Forms are one of the most complex interactions your users will encounter in your applications. It is important to keep the usability of your forms in mind when creating and composing your form components.

Next up you will learn how to animate your React components to create more engaging applications.

Chapter 10. Animations

Now that you can compose a complex set of React components, 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 easy to do.

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 CSS 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

In this example we'll animate our example app comment editing pane in and out. `editingPane` will either be a react element or null.

```
<ReactCSSTransitionGroup
  transitionName='EditingPaneTransition'
  transitionEnterTimeout={300}
  transitionLeaveTimeout={1200}
>
```

```
    {editingPane}
  </ReactCSSTransitionGroup>
```

This `ReactCSSTransitionGroup` component comes from an addon, which is included near the top of the file with `var ReactCSSTransitionGroup = require('react-addons-css-transition-group')`.

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.

You can read the full source code of our example application at: http://git.io/vlcpa.

Styling transition classes

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

Here's an example of the CSS.

```
.EditingPaneTransition-enter {
  transform: scale(1.2);
  transition: transform 0.3s cubic-bezier(.97,.84,.5,1.21);
}

.EditingPaneTransition-enter-active {
  transform: scale(1);
}

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

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

The transition lifecycle

The difference between `EditingPaneTransition-enter` and `EditingPaneTransition-enter-active` is that the `EditingPaneTransition-enter` class is applied as soon as the component is added to the group, and the `EditingPaneTransition-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.

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='EditingPaneTransition'
  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 can 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 that 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 the less-predictable `setTimeout` can be the only 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.

```
class Positioner extends React.Component {
  constructor() { super(); this.state = {position: 0} }

  resolveAnimationFrame() {
    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() {
    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.

The open source [React Tween State] library provides a convenient layer of abstraction over this style of animation.

Spring Animation

CSS transitions are simple to use and sometimes good enough. They have many limitations from only supporting bezier curves to being unable to halt or transition to a new value mid-animation. The web has fallen behind the trend by not supporting spring animations as seen on mobile and other native applications.

This is where a tool like react-motion (https://github.com/chenglou/react-motion) comes in. We're using react-motion 0.3.1 here, so be sure to check the docs for your version.

First, we need to import react-motion's take on transition groups:

```
import {TransitionMotion, spring} from 'react-motion';
```

Then we can use it in render. It has an odd API, where we pass a function as children instead of the elements directly. It then passes the current progress of the style values to that function and render based on it.

```
<TransitionMotion
  willEnter={() => ({ opacity: spring(0) })}
  willLeave={() => ({ opacity: spring(0) })}
  styles={this.state.show ? {x: {opacity: spring(1)}} : {}}
>
  {() => {
    var {x} = styles;
    if (!x) return null;
    return <div styles={x}>Hello world</div>;
  }}
</TransitionMotion>
```

When `this.state.show` becomes true it'll animate in with a nice physics based transition, and animate out in the same way. If the transition isn't complete it'll smoothly animate to the new value.

You can completely replace CSS transitions with spring physics to improve user experience.

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.
4. Deliver spring animations.

Next, you'll be learning how to fine-tune your React performance!

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. There are, however, cases where the delicate tuning of a component should render a new virtual DOM representation that can help make your application faster. For instance, this is useful 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 receiving new props or `setState` being called, React will invoke the `render` method on each child component of that component. In most cases this performs without a hitch, but on pages with deeply nested component trees or a complex render method, you can experience some sluggishness.

Sometimes a component's render method is invoked when it doesn't need to be. This can happen when your component doesn't use state or props, or if the props or state don't change when the parent re-rendered. So, rendering the component again would yield the same exact virtual DOM representation that's already present, which would be unnecessary.

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

`shouldComponentUpdate` should return a boolean. `false` tells React to not invoke the component's render method and use the previously

rendered virtual DOM. Returning `true` will tell React to invoke the render method on the component to get a new virtual DOM representation. By default `shouldComponentUpdate` returns true and components will thus always re-render.
Note that `shouldComponentUpdate` is not called on the initial render 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. For example a component with one prop and no state can look like this:

```
shouldComponentUpdate(nextProps, nextState){
  if (this.props.data !== nextProps.data) {
    return true;
  }
  return false;
}
```

For pure components that always render the same given the same props and state, you can add the `react-addons-pure-render-mixin`. Because we're using ES6 classes, we need a library called react-mixin (https://github.com/brigand/react-mixin) that allows you to use the mixin. This is one of the few cases where mixins should be used instead of high order components.

You can read the full source code of our example application, a reddit clone, at http://git.io/vlcpa

---

The mixin will overwrite `shouldComponentUpdate` to compare the new props and state with the old, and return false if they are equal like in the above example.

A few of the components are this simple, like `EditEssayQuestion`, which you can use `React.addons.PureRenderMixin` with:

```
import reactMixin from 'react-mixin';
import PureRenderMixin from 'react-addons-pure-render-mixin';
```

```
class Foo extends React.Component {
  render(){
    ...
  }
}
reactMixin.onClass(Foo, PureRenderMixin);
```

In cases where you have deep complex props or state, the comparison can be slow. To help mitigate this, consider using immutable data structures like those discussed in Chapter 16.

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 that 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, you can simply use some string (or number) you 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 causes the `src` attribute of all following `<img>` tags.

It's also valuable to note that while you seemingly pass the key in as a prop, it is not accessible in anyway from within a component.

Summary

In this chapter you learned how to:

1. Modify `shouldComponentUpdate` to return true/false for added performance.
2. Using `key` to help inform React of the minimum changes to a list of child components.

So far we've focused on using React in the browser, next you'll learn about universal (also know as isomorphic) applications: JavaScript with React on the server.

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 is central to why React can be used for server side rendering. Each React component is first rendered to the virtual DOM, then React 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 to work in non browser environments like Node.js. Instead of updating the real DOM, React can generate a string from the virtual DOM. This allows you to use the same React components on the client and the server.

React provides two functions that can be used to render your components on the server side, `React.renderToString` and `React.renderToStaticMarkup`.

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

- What render function is best to use.
- How to support Asynchronous 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 your application.
- Your usage of Singletons, Instances, and Context.

Render functions

When rendering your React components on the server side, you can't use the standard `React.render` method, since on the server side there is no DOM. React enables server side rendering by providing two render functions that support a subset of the standard React component lifecycle methods.

React.renderToString

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

Unlike `React.render`, 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.renderToString(<MyComponent/>);

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

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

`data-reactid` is used by React 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 to reuse the DOM from the server when rendering the same component on the client. This is only added to the root element.

React.renderToStaticMarkup

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

This is the same as `React.renderToString`, except it doesn't include the React data attributes.

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

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

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

String or StaticMarkup

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

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

Here are some examples:

- Generating HTML emails.
- Generating PDFs by HTML to PDF conversion.
- Testing components.

In most cases you will want to use `React.renderToString`. This will allow React to use the `data-react-checksum` to make the initialization of the same React component on the client much faster. As React 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.

It is very important that your React components render *exactly* the same on the server and on the client. If the `data-react-checksum` doesn't match, React will destroy the DOM supplied by the server and generate new DOM nodes and attach them to the document. In this case you will lose 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. This is true especially when creating event listeners, since there is no component lifecycle method to let you know when the React component is finished.

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

The 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 performance benefits of server side rendering. This

means designing your components with server sider rendering in mind.

You should always design your React 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.renderToStaticMarkup(<MyComponent/>);
var result2 = React.renderToStaticMarkup(<MyComponent/>);

//result
<div>0.5820949131157249</div>

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

If you were to change your component it would receive 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.renderToString(<MyComponent number={num}/>);
```

```
//pass num to client side
React.render(<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 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.render(<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 component can show a loading view while waiting for the asynchronous data to return. On the server side, this behavior can’t be directly replicated with React since 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 to be 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 since `React.renderToString` is synchronous.

Solution: Use a statics function to fetch your asynchronous data, and 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.renderToString(Username(props));

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

Using the solution above, the 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 (such as HTML5, pushState or fragment change) should ignore any `initialState` from the server. You will also want to display loading messages as the data is fetched.

Isomorphic routing

Routing is an essential part of any non trivial application. To render a React application with a router on the server side, you need to ensure the router supports rendering without the DOM.

The fetching of asynchronous data is the job of a router and its 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 should be passed down to the deeply nested component. If it is not needed for SEO then it is okay 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, 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 an isolation bubble. Each instance of your application can't mix state with other instances, since 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, since 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 your application may mutate the state of a singleton, resulting in unwanted behavior.

React 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 Contextify allow you to run your code in isolation from one another on the server. It is similar to using webworkers on the client. Contextify works by taking your application code and running it in a separate Node.js V8 instance. Once the code is loaded you can invoke functions in the context. This approach will give you the freedom to use singletons but does come at a performance cost since a new Node.js V8 instance will need to be started for every request.

The core development team of React discourage the use of passing context and instances down your component tree. 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 of the components up the component hierarchy. This in turn increases the complexity of your application, reducing maintainability as your application grows.

There are trade-offs with each approach when deciding to use singletons or instances to control your context. You will need to assess your specific requirements before deciding what approach you will want to take. You will also need to consider how the third party libraries you use are architected.

Summary

Server side rendering is an important part of building search engine optimized web sites and web applications. React supports the rendering of the same React components on the server and the client browser. To do this effectively, you will need to ensure your entire application is architected in such a manner to support server side rendering.

Next up you will you learn about development and build tools in the React family.

Chapter 13. Development tools

React leverages a few layers of abstraction to help ease development of your components and reason about the state of your applications. These abstractions have a down side when it comes to debugging, building and shipping your applications.

Fortunately, there are some fantastic development tools available to help in the process of debugging and building your applications. In this chapter we will explore the build tools and the debugging tools that you can use effectively with React.

Build tools

Build tools help you streamline repetitive tasks to make the process of running your code much easier. One of the most repetitive tasks in React development is running the JSX parser over all of your React components. The other big task is bundling all of your modules into one or more bundles for shipping to the browser.

Let's examine how to use two popular JavaScript build tools, Browserify and Webpack, with React.

Browserify

Browserify is a JavaScript packaging tool that allows for Node.js style `require()` calls in the browser. Without getting into too much detail and getting lost in the woods, Browserify packages all of the requires into a bundled JavaScript file suitable for running in the browser environment. Running Browserify on any JavaScript file with `require` statements will automatically bundle the necessary JavaScript.

While powerful, Browserify is only for JavaScript, unlike Bower, Webpack, or other bundling solutions.

Setup a Browserify project

To get Browserify up and running, you must initialize a node project. Assuming you already have both node and npm installed, you can initialize a new project using the following command in your terminal. This will create a `package.json` file with the necessary assets.

```
npm init
# ... answer questions as necessary to complete init
npm install --save-dev browserify babelify \
babel-preset-es2015 babel-preset-react \
babel-preset-stage-1 \
react uglify-js
```

Add the following build script to the bottom of your `package.json` file so it will look similar to this:

```
  ...
  "devDependencies": {
    "browserify": "^5.11.2",
    "babelify": "^6",
    "react": "^0.11.1",
    "uglify-js": "^2.4.15",
     ...
  },
  "scripts": {
    "build": "browserify --debug index.js > bundle.js",
    "build-dist": "NODE_ENV=production browserify index.js | uglifyjs -m > bundle.min.js"
  },
  "browserify": {
    "transform": ["babelify"]
  }
}
```

Also create a file called .babelrc that tells babel which transforms to use.

```
{
  "presets": ["es2015", "react", "stage-1"]
}
```

The default build task can be run with `npm run build`, and will create a bundle with source maps. This lets you inspect error messages and place breakpoints as if each file was included individually. You'll also see the original code with JSX rather than the compiled output.

For production builds you need to specify that the environment is production. React uses a transform called envify, which when combined with a minifier like uglify, allows debug code and detailed error messages to be removed for better performance and smaller file sizes.

Now you're ready to write some React and package it.

Add some React content

Create the following React + JSX JavaScript file as `index.js`.

```
var React = require('react');
var ReactDOM = require('react-dom');

var root = document.getElementById('root');
ReactDOM.render(<h1>Hello World</h1>, root);
```

And add a basic `index.html` file:

```
<html>
  <head>
    <title>React + Browserify Demo</title>
  </head>
  <body>

    <div id="root"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

Now your project will have the following files and folders inside its root directory:

- index.html
- index.js
- node_modules/
- package.json

If you try to load the index.html file now, it will fail to render the JavaScript since you have yet to build the final package. Run `npm run build` and reload the page to see your hello world example.

Watchify

You may choose to add a watch task, which is good for development. Watchify is a wrapper around Browserify, which rebuilds your bundle when you change a file. It uses caching to speed up the rebuilds.

```
npm install --save-dev watchify
```

Add this to your scripts object in package.json:

```
"watch": "watchify --debug index.js -o bundle.js"
```

Now instead of `npm run build`, you can `npm run watch` for a smoother development experience.

Build

Now, simply run the build script to package your React + JSX file as a bundled script for the browser.

```
npm run-script build
```

You should see a new bundle.js file appear. If you open bundle.js you'll notice some minified JavaScript at the top, followed by your transformed React + JSX code. This file now contains all of the supporting code along with your index.js file, ready to run in the browser, and loading index.html will now succeed.

Webpack

Webpack is like Browserify, it bundles up your JavaScript into a single package. It isn't quite fair, however, to compare Browserify and Webpack, because Weback has many more features, and Browserify is not often used on its own.

Webpack can also:

- Bundle CSS, images and other assets into the same package.

- Pre-process files before bundling (less, coffee, jsx, etc).
- Split your bundle into multiple files based on entry locations.
- Support feature flags for development.
- Perform “hot” module replacement.
- Support asynchronous loading.

As a result, Webpack can do the job of Browserify mixed with other build tools like grunt or gulp.

Webpack is a module system, and is supported by adding and replacing plugins. By default it comes with a commonjs parser plugin enabled.

We won’t detail every aspect of Webpack here, but will cover the basics and what you need to get Webpack to work with React.

Webpack and React

React helps you to create components for your application. Webpack helps you bundle not only JavaScript, but all of your other assets required for your application. This allows you to create components that include all of their asset dependencies. This then makes your components more portable, given that they can bring their own dependencies along with them. As an added advantage, as your application grows and changes and when you remove components, their asset dependencies are also removed. This means no more dead CSS or orphaned images.

Let’s have a look at what a what a React component looks like when it is requiring it’s own asset dependencies.

```
//logo.js
require('./logo.css');

var React = require('react');

var Logo = React.createClass({
    render: function () {
        return <img className="Logo" src={require('./logo.png')} />
    }
});
```

```
module.exports = Logo;
```

To bundle this component up you need to have any entry point for your application.

```
//app.js
var React = require('react');
var ReactDOM = require('react-dom');
var Logo = require('./logo.js');

var root = document.getElementById('root');
React.render(<Logo/>, root);
```

You will now create a Webpack config file to tell Webpack what loaders to use for the different file types. And, we will show what your applications entry point is and where to place all of the bundled files.

```
//webpack.config.js
module.exports = {
    //starting point of your application
  entry: './app.js',
  output: {

    //location for all bundled assets
    path: './public/build',



    //prefix for all url-loader assets
    publicPath: './build/',



    //resulting name of bundled application
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {

        //regex for file type supported by the loader

        test: /\.(js)$/,
        loader: 'babel-loader'
      },

      {

        test: /\.(css)$/,



        //multiple loaders are chained together with an "!";

        loader: 'style-loader!css-loader'

      },
```

```
      {
        test: /\.(png|jpg)$/,

        //url-loader supports base64 encoding for inline assets
        loader: 'url-loader?size=8192'
      }
    ]
  }
};
```

You now want to install Webpack and all of the loaders. You can either use npm via the command line, or package.json.

Make sure you install the loaders locally, not globally (-g).

```
npm install webpack react
npm install url-loader jsx-loader style-loader css-loader
```

Once everything is installed, you can run Webpack:

```
//build once for development
webpack

//build with source maps
webpack -d

//build for production, minified, uglify dead-code removal
webpack -p

//fast incremental building, can be use with other options
webpack --watch
```

Debugging tools

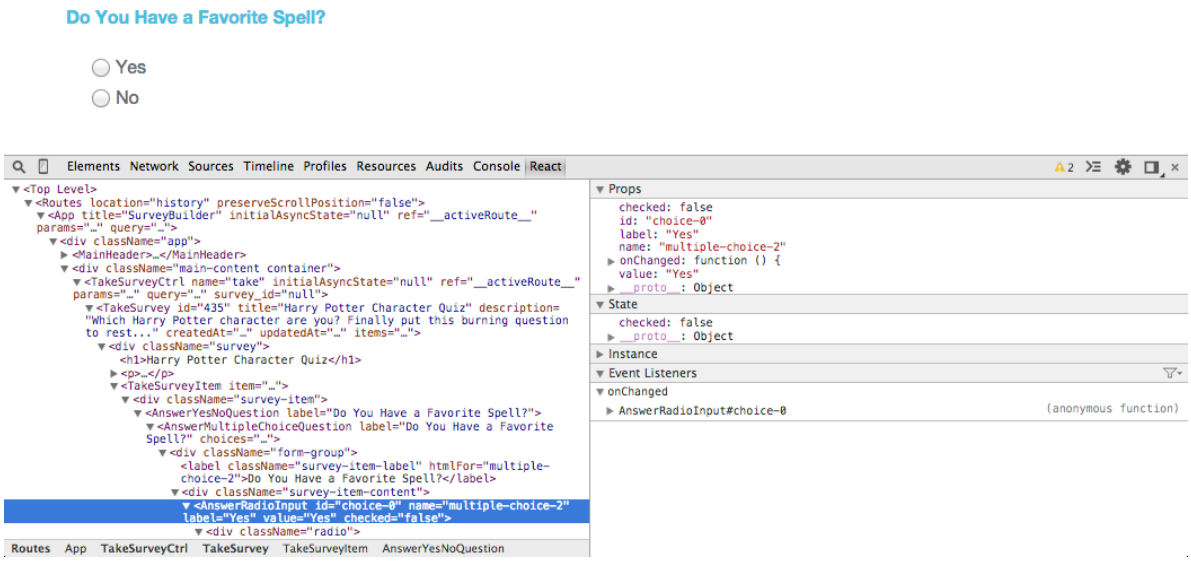
No matter how careful you are, mistakes will be made. We won't cover how to debug JavaScript, but we will talk about some tools to make debugging React applications easier.

Starting out

For this chapter, open Chrome and install the [React Developer Tools] addon.

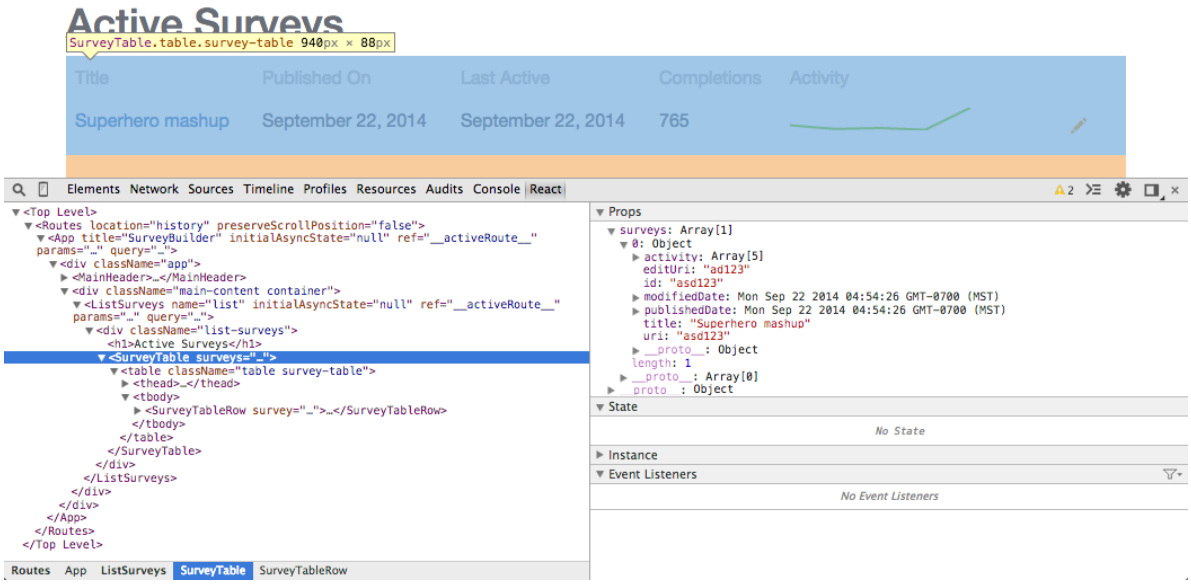
Right-click on an element, and choose Inspect Element. You get the familiar view of the DOM structure, which is the elements tab.

You're not here to look at the DOM, though. You want to see the components, their props, and their state. If completed, to take the next steps, you should see a tab on the far right named React.



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 your components in React. The developer tools allow you to do even more than this.



You can see your SurveyTable component get passed an array of surveys that contains 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 on your team find the components they need to edit and 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.

Here is a jsbin set up with Babel and React, ready for you to create things: http://jsbin.com/ledaqefuya/1/edit?js,output.

Summary

You should now see the benefits that debugging and building tools can provide you with when developing with React. In the next chapter you

will find details on how to use automated testing in React.

Chapter 14. Testing

Now that you've learned the ins and outs of how to build a web application using React, let's take a step back before learning about architectural patterns (that will happen in Chapter 15). 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 code base that is a twisted mess, which is very hard to change. That is why it is very important to have a powerful tool in your toolbox: automated testing. Automated testing, usually employed through a Test Driven Development (TDD) workflow, can help you achieve code that 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! If you've never done it before, automated testing can seem like a foreign concept that is always slightly out of reach. 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

You may be asking, "But I have an amazing QA team focused on testing. Why should I care about testing? Can't I just skip this chapter?" 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 that is poorly written is hard to test. So if you start writing your tests as you write your code, your tests will encourage you not to write sloppy code. You will be naturally pushed to follow the single responsibility principle [1], the law of demeter [2], and keeping code in a modular way.

When we refer to Test Driven Development, or TDD, we are referring to a style of testing that uses a "Red, Green, Refactor" process. First, you write a test that fails (the test is red), then you write the application code to make the test pass (the test is green), and then you refactor the application code to make it cleaner while keeping the test green. It is a process that allows you to work in small batches and each iteration gives a positive feeling when the test turns green.

Types of testing

Now that you've been convinced that automated testing is important, let's go over the type of testing: unit testing. There are many other types of automated testing (integration testing, functional testing, performance testing, security testing, and visual testing [3]), but those are outside the scope of this book.

- Unit Test: a test that 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.

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 them to get our test suites off the ground quickly. The tests for this book use Jest and Enzyme. Jest runs tests using Jasmine so, if you are familiar with Jasmine, you may find these tests familiar. Listed below are some popular alternatives.

- Karma
- Mocha
- Chai
- Sinon
- Casper.js
- Qunit

So let's start coding!

Unit Testing React Components with Jest and Enzyme

When writing a React component, the only requirement is that you define a render function. Let's begin writing a test for a new `<Comment />` component that verifies that a component renders the desired output.

Jest automatically scans for test files inside of directories named `__tests__`. Conventionally, these test directories are placed in the folder with the file being tested.

Making Assertions About the Content of Components

In keeping with that convention, create a new file for your test at `/src/molecules/__tests__/Comment.test.js` in the sample project.

We begin by importing React, a few helper functions from enzyme that we'll review as the test grows, and the Comment module itself.

```
import React from 'react';
import {shallow, mount, render} from 'enzyme';

import Comment from '../Comment';
jest.unmock('../Comment');
```

After importing Comment, we tell Jest to `unmock` it because we want to test it. This is part of what makes Jest unique. Jest automatically mocks all of the JavaScript modules your code depends on. This makes it simple to test the behavior of a single module or collection of modules without creating mock implementations manually.

CONFIGURING JEST

Jest is configured in the project's `package.json` file. It is configured to not mock a number of important libraries that support the app so those libraries still function as expected in your tests and to preprocess your source code to support new ES6 syntax.

Now, let's add the actual test.

```
...
describe('molecules/Comment', () => {
  it('<Comment />', () => {
    expect(render(<Comment />)).toBeDefined();
```

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

This test asserts that rendering the `Comment` actually returns a value. Let's see if it works. Save the file and run your tests by typing the following in your terminal at the root of the project:

```
npm test
```

When you run this command you should see output like the following:

```
Using Jest CLI v0.9.2, jasmine2
 PASS  src/utils/__tests__/shallowCompareWithChildrenId.test.js (0.347s)
Warning: React.createElement: type should not be null, undefined, boolean, or number. It should be a string (for DOM elements) c
FAIL  src/molecules/__tests__/Comment.test.js
● Runtime Error
Error: Cannot find module '../Comment' from 'Comment.test.js'
  at Loader._resolveNodeModule (/Users/youruser/bleeding-edge-react-sample-app.github.io/node_modules/jest-cli/src/HasteModuleLc
  at Object.<anonymous> (/Users/youruser/bleeding-edge-react-sample-app.github.io/src/molecules/__tests__/Comment.test.js:5:6)
 PASS  src/atoms/__tests__/Box.test.js (1.058s)
1 test failed, 7 tests passed (8 total in 3 test suites, run time 2.427s)
```

The test failed because we still haven't implemented our `Comment` component. Before we move on to fixing this, run the following in your terminal to run Jest in `watch` mode. It will watch for changes in your project and re-run your tests automatically, giving you quick feedback as you code!

```
npm test -- --watch
```

Now let's create a new file at `/src/molecules/Comment.js` and add the following implementation to start:

```
import React, {PropTypes} from 'react';
import Box from '../atoms/Box';

export default
class Comment extends React.Component {
  static propTypes = {

  };

  render(){
    return (
      <Box>
        A Comment!
      </Box>
    );
  }
}
```

This incomplete implementation simply renders a Box with static text. After saving your new Comment component, return to your terminal, where you should see:

```
 PASS  src/utils/__tests__/shallowCompareWithChildrenId.test.js (0.343s)
 PASS  src/atoms/__tests__/Box.test.js (0.987s)
 PASS  src/molecules/__tests__/Comment.test.js (1.092s)
8 tests passed (8 total in 3 test suites, run time 2.228s)
```

Your test passes! But our Comment component is not complete yet. The objective with the Comment component is to extract the logic inside of the map function in `Comments.js` and create a component out of it. The map function currently provides a `comment` object so our Comment component should take that as a required property. Let's modify our existing test so that it renders a Comment component with an example comment prop and verify that it contains some of the comment prop's content:

```
describe('molecules/Comment', () => {
  let comment = {
    id: 1,
    replies: [],
    score: 1,
    score_hidden: 1,
    body: "Test Driven Development, yeah!",
    author: "Jeremiah Hall"
  };

  it('<Comment comment={comment}/>', () => {
    let renderedComment = render(<Comment comment={comment} targetScore={1} />);
    expect(renderedComment.text()).toContain(comment.body);
    expect(renderedComment.text()).toContain(comment.author);
  });
});
```

Let's examine the new test:

1. First, an example `comment` object is defined.
2. Then a Comment component with some example props is rendered using Enzyme's render function.
3. You then assert that the text of the rendered comment contains the body and author of the comment.

The object returned by Enzyme's render function is a wrapper object with a number of helpful methods. You'll use the other two functions in later tests.

After saving the file you should see that the test failed:

```
PASS  src/utils/__tests__/shallowCompareWithChildrenId.test.js (0.342s)
PASS  src/atoms/__tests__/Box.test.js (0.99s)
FAIL  src/molecules/__tests__/Comment.test.js (1.112s)
● molecules/Comment › it <Comment comment={comment}/>
 - Expected 'A Comment!' to contain 'Test Driven Development, yeah!'.
       at Object.eval (src/molecules/__tests__/Comment.test.js:23:36)
       at handle (node_modules/worker-farm/lib/child/index.js:41:8)
       at process.<anonymous> (node_modules/worker-farm/lib/child/index.js:47:3)
       at emitTwo (events.js:87:13)
       at process.emit (events.js:172:7)
       at handleMessage (internal/child_process.js:686:10)
       at Pipe.channel.onread (internal/child_process.js:440:11)
 - Expected 'A Comment!' to contain 'Jeremiah Hall'.
       at Object.eval (src/molecules/__tests__/Comment.test.js:24:36)
       at handle (node_modules/worker-farm/lib/child/index.js:41:8)
       at process.<anonymous> (node_modules/worker-farm/lib/child/index.js:47:3)
       at emitTwo (events.js:87:13)
       at process.emit (events.js:172:7)
       at handleMessage (internal/child_process.js:686:10)
       at Pipe.channel.onread (internal/child_process.js:440:11)
1 test failed, 7 tests passed (8 total in 3 test suites, run time 2.192s)
```

Let's complete the Comment component. You'll need to add a few imports and implement the render method:

```
import React, {PropTypes} from 'react';
import Box from '../atoms/Box';
import Heading from '../atoms/Heading';
import Markdown from '../atoms/Markdown';
import Link from '../atoms/Link';
import {State} from '../utils/actions';

export default
class Comment extends React.Component {
  static propTypes = {
    comment: PropTypes.object.isRequired,
    targetScore: PropTypes.number.isRequired
  };

  render(){
      let comment = this.props.comment;
      let targetScore = this.props.targetScore;
      let replies = comment.replies || [];
      return (
        <Box
          key={comment.id}
          padding="0.5em"
          margin="0.5em"
          style={{
            background: comment.score >= targetScore ? '#ffffaa' : '#efefef'
          }}
        >
          <Box direction="row">
            <Heading level="title">{comment.score_hidden ? '?' : comment.score}</Heading>
            <Box margin={{right: "1em"}} />
            <Box style={{maxWidth: '80em', lineHeight: '1.5'}}>
              <Markdown content={comment.body} />
            </Box>
          </Box>
          <Box margin={{top: '0.5em'}} direction="row">
            <span>
              by <Link to={`/user/${comment.author}`}>{comment.author}</Link>
            </span>
            <Box
              id="replyBox"
              margin="0 1em"
              onClick={() => {
                State.setEditing({
```

```
                type: 'comment',
                id: comment.id,
              })
            }}
            style={{cursor: 'pointer'}}
          >
            Reply
          </Box>
        </Box>
        {replies.map((reply) => <Comment comment={reply} targetScore={targetScore} />)}
      </Box>
    );
  }
}
```

The render method of `Comment` is very similar to the function passed to the map call in the `Comments` component. When you return to your terminal after saving Comment.js, Jest should show that your test now passes:

```
PASS  src/utils/__tests__/shallowCompareWithChildrenId.test.js (0.366s)
PASS  src/atoms/__tests__/Box.test.js (1.079s)
PASS  src/molecules/__tests__/Comment.test.js (1.307s)
8 tests passed (8 total in 3 test suites, run time 2.335s)
```

Making Assertions About Functions and DOM Events

Let's add a new test to `Comment.test.js` that asserts that when the reply button is clicked, the state of the application is updated. This test should pass the first time without modification.

```
describe('molecules/Comment', () => {
  ...
  it('should should call State.setEditing when the reply button is clicked', () => {
    let wrapper = mount(<Comment comment={comment} targetScore={1} />);
    let replyBox = wrapper.find('#replyBox');
    State.setEditing = jest.fn();
    replyBox.simulate('click');
    expect(State.setEditing).toBeCalledWith({
      type: 'comment',
      id: comment.id,
    });
  });
});
```

There's a few new things in this test so, let's examine it:

1. First a Comment component is rendered to DOM using Enzyme's `mount`.
2. Using the returned Enzyme wrapper's `find` method, the element with id `replyBox` is selected.
3. A mock function is created with `jest.fn()` and assigned to `State.setEditing`.
4. Using the wrapper a `click` event is simulated on the `replyBox` element.
5. Finally, you assert that `State.setEditing` should have been called with the provided object.

It is common for a React component to execute a function in response to user interaction or other lifecycle events. Jest and Enzyme make it simple to test that your component behaves as expected.

Making Assertions About Child Components

Now that we've implemented our new `Comment` component, let's write a test to verify that the `Comments` component renders a `Comment` component for each object in its `comments` prop.

```
import React from 'react';
import {shallow, mount, render} from 'enzyme';

import Comments from '../Comments';
import Comment from '../Comment';

jest.unmock('../Comments');
jest.unmock('../Comment');

describe('molecules/Comments', () => {
  it('<Comments comments={comments}/>', () => {
    let comments = [{
```

```
        id: 1,
        replies: [],
        score: 1,
        score_hidden: 0,
        body: "Test Driven Development, yeah!",
        author: "Jeremiah Hall"
    }, {
        id: 2,
        score: 0.5,
        score_hidden: 0.25,
        body: "React is great!",
        author: "Jeremiah Hall"
    }];

    let wrapper = shallow(<Comments comments={comments} />);
    let childComments = wrapper.find(Comment);

    expect(childComments.length).toEqual(comments.length);
  });
});
```

This test defines an array of two comment objects, and shallow renders a `Comments` component with this array as a prop. It then uses the returned wrapper's `find` method to search for the child components of type Comment. You can search for any kind of React component this way. Enzyme supports supports searching with CSS selectors such as `.class-name` or `#replyBox`, using component constructors as shown in the above example, by the component's display name, or by component props.

You can review Enzyme's API documentation on GitHub:

https://github.com/airbnb/enzyme

SHALLOW, MOUNT, AND RENDER IN ENZYME

Enzyme's `shallow` function treats the component it is rendering as a single unit, so that your test is not affected by child components. The `mount` function renders to DOM and ensures that component lifecycle methods like `componentDidUpdate` and others are called when appropriate. The `render` function goes a step further to static HTML, which you can then analyze.

This Comments test will initially fail since we need to modify the component to use our new Comment:

```
...
import Comment from './Comment';

export default
class Comments extends React.Component {
  ...
  render(){
    // find the average comment score
    var targetScore = this.getGoodScoreThreshold();

    return (
      <Box>
        {this.props.comments.map((comment) => <Comment key={comment.id} comment={comment} targetScore={targetScore} />)}
      </Box>
    );
  }
}
```

An import statement for `Comment` is added to the top of the file and the map function is modified to return a `Comment` component for each comment in the props.

Let's expand our test to verify that the child `Comment` components received the right comment prop.

```
describe('molecules/Comments', () => {
  it('<Comments comments={comments}/>', () => {
    ...
    expect(childComments.length).toEqual(comments.length);
    expect(childComments.at(0).prop('comment')).toEqual(comments[0]);
    expect(childComments.at(1).prop('comment')).toEqual(comments[1]);
    ...
  });
});
```

The two new assertions retrieve the `comment` prop from each child component and assert that they are equal to the comment at the same index in the example comments array.

Summary

This chapter was a whirl-wind of testing concepts: rendering, auto-mocking, unmocking, wrappers, simulating events, element finders, and test driven development! You are now prepared to start testing your React components in your real world applications.

Now that we've covered how to unit test our React applications, let's learn about some architectural patterns we can use to structure our React applications.

[1] http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html

[2] http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf

[3] https://www.youtube.com/watch?v=1wHr-O6gEfc

Chapter 15. Architectural patterns

There are a good number of architectural patterns that can be used with React. From a standard Flux pattern, to Flux variants like Reflux, to an entirely different language and ecosystem like Om and ClojureScript, React has proven to be quite adaptable. In this chapter we'll explore how our sample React app is structured and how this structure helps manage complexity as your project grows. We'll also explore some popular alternatives.

It is commonly said that you can consider React to be the V in MVC. It could be used with inline AJAX requests as shown below but, this quickly becomes hard to reason about:

```
export default
class Reply extends React.Component {
  getInitialState() {
    return {
      text: ""
    };
  }
  submitReply() {
    $.ajax({ url: '/replies', type: 'POST',
      contentType: 'application/json',
      dataType: 'json',
      data: JSON.stringify(this.state.text)}, (json) => {
        this.setState({ text: "" });
      });
  }
  render() {
    return (
      <div>
        <textarea
          value={this.state.text}
          onChange={(e) => this.setState({text:e.target.value})}
        />
        <input value="Submit" type="button" onClick={() => this.submitReply()} />
      </div>
    );
  }
}
```

For example, how do you handle updating other components that may need to know about this new reply?

React is flexible enough to work well with many MVC frameworks. Let's begin by examining react-router.

Routing

Routing is an intuitive way to link the URL to the application's view hierarchy.

Routers direct URLs in a Single Page Application to specific views, or components.

You can imagine that for the URL/items you'd want to run a function that loads the items from the server and renders a `<ItemsList />` component.

There are many different kinds of Routers. They exist 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 provide a router. The explosive growth of the React community has supported the creation of a number of routing solutions. In this section we'll react-router, the routing solution used in the sample app.

react-router

The sample app for this book uses react-router. It differs from other routers in that the router is composed of JSX elements.

The routes are defined as `Route` components and each route has a `component` prop that is a React component.

This is the router in the sample app:

```
export default (
  <Route component={App}>
    <Route component={MainLayout}>
      <Route path="/" component={require('./pages/HomePage/HomePage')} />
      <Route path="/user/:id" component={require('./pages/UserPage/UserPage')} />
```

```
      <Route path="/r/:id" component={require('./pages/BoardPage/BoardPage')} />
      <Route path="/item/:id" component={require('./pages/DetailsPage/DetailsPage')} />
    </Route>
  </Route>
);
```

To get the router to run you render it as your top level component:

```
var root = document.getElementById('app-root');
ReactDOM.render(
  <Router history={createBrowserHistory()}>{routes}</Router>
, root);
```

Like the other Routers, react-router has a similar concept of params. For instance the route `'/user/:id'` will pass in the `id` property to the `UserPage` component.

One of the cool features of react-router is that it provides a Link component that you can use for navigation and it will map it to the routes. In the sample app, the react-router Link component has been encapsulated in the file `src/atom/Link.js`. but, the props are essentially just passed to the react-router component.

Here's how our `<MainNavigation/>` component looks with the Link component:

```
...
export default
class MainNavigation extends React.Component {
  render(){
    return (
      <Box>
        <Box direction="row">
          {tabs.map((tab) => {
            return (
              <Box margin={{left: "0.1em"}} key={tab.name}>
                <Link unstyled to={tab.to}>
                  <Button which={tab.which}>{tab.name}</Button>
                </Link>
              </Box>
            );
          })}
        </Box>
      </Box>
    );
  }
}
```

Read more about and download react-router here: https://github.com/reactjs/react-router.

Flux

Flux is an architectural pattern for client-side applications introduced by Facebook. It complements 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 four main parts: the store, the dispatcher, the action creators, and the views (React components.) Action creators are helper methods to create a semantic interface to the dispatcher.

You can think of your top-most React component as a View-Controller. The View-Controller component interfaces with the store and facilitates communication with its child 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 key aspect of Flux is the one-way flow of information. Compared to other approaches with less structured event handling or complicated two-way data binding this may seem very different.

Flux has a well defined flow between its components.

1. When your application starts your Stores will register callbacks or event listeners with the Dispatcher.
2. When React components in your application are rendered, they'll register their own callbacks or event listeners with the Stores.
3. When a user interacts with a View and clicks on something that may trigger an Action Creator.
4. The Action Creator might perform an AJAX request and then dispatch an Action with the Dispatcher.

5. The Dispatcher will then deliver that Action to all of the Stores that registered callbacks with it.
6. The Stores will update their state as necessary, depending on the content of the Action, and if their state changed they'll notify all the components that are listening for change events.
7. The notified Views will then request the updated state from the stores and re-render if necessary, bringing the application to a new, updated state.

This uni-directional flow and well defined separation of responsibilities among the components makes it much easier to reason about state changes within your application.

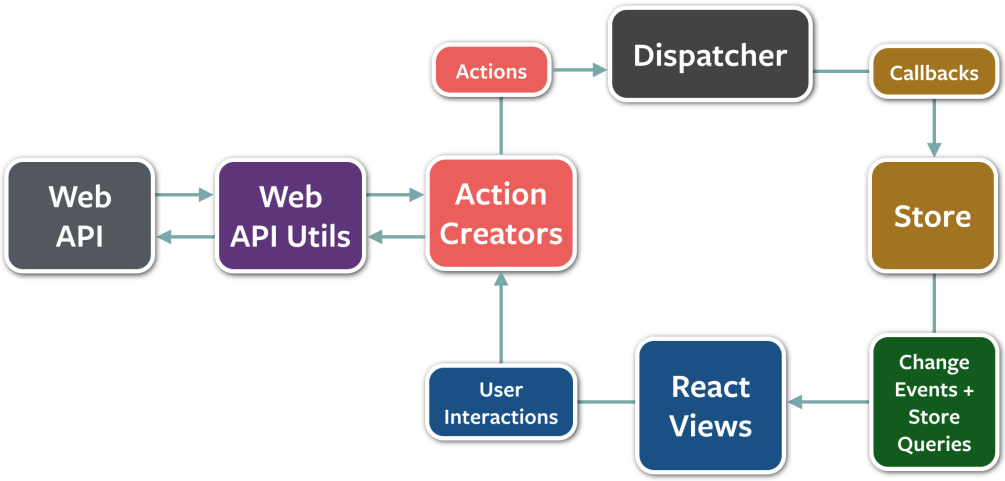


Image credit: Facebook

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.

Let's 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. Actions flow through the dispatcher and out into the stores that have registered for it.

Our sample application contains multiple stores. As you grow your application, 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 constitute the domain-specific-language of your application. Action Creators translate user interaction into meaningful dispatcher Actions —statements that a store can act on.

Store

The store is responsible for encapsulating business logic and interactions with your application'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.

- Stores hold *all* your application's data.
- No other part of your application should know how to interact with the data — stores are the only place in your application where data mutation occurs.
- Stores don't have setters—all updates flow through the dispatcher into the store. Fresh data flows back into the application through the store's change event.

Below is an example of a store registering with the Dispatcher to receive comments for a post.

```
Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    ...
    case CommentConstants.RECEIVE_COMMENTS:
      CommentStore.setComments(payload.comments, payload.parentPostId);
      break;
  }
});
```

The store receives the action, performs the work of saving the comments, and when done emits the change event.

```
CommentStore.prototype.setComments = function(comments, parentPostId) {
  // handle saving the results here
  this.emitChange();
}
```

This change event is then handled by your React Views. The new state will flow through your React component hierarchy and the components will re-render as needed.

View-Controller

Your application's component hierarchy will commonly have a high-level component responsible for interacting with stores. Simple applications may only have one. More complex applications might have many.

The process of wiring up a 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 an example of handling store interactions.

```
export default
class App extends React.Component {
  handleChange() {
    PostStore.getPosts((posts) => {
      this.setState({ posts: posts });
    });
  }
  componentDidMount() {
    PostStore.addChangeListener(this.handleChange);
  }
  componentWillUnmount() {
    PostStore.removeChangeListener(this.handleChange);
  }
  ...
}
```

Managing multiple stores

Inevitably, applications 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 its record action before we can safely update the summary store.

The standard Flux implementation from Facebook includes a method named `waitFor`, which will help us instruct stores to wait for other stores to finish processing an event before proceeding.

Registering for dependent actions

Suppose we have two stores concerned about a `CREATE_COMMENT` action.

- The `CommentStore` needs to store the comment.
- The `PostStore` needs to know re-tally the total comments for the post.

This means that `PostStore` is dependent upon the `CommentStore` completing the `CREATE_COMMENT` action before it can complete its work.

When we register the `CommentStore` at the top of our App, we store a reference to the dispatcher token.

```
// Wire up the CommentStore with the action dispatcher
CommentStore.dispatchToken = Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    ...
    case CommentConstants.CREATE_COMMENT:
      CommentStore.saveComment(payload.comment, payload.parentPostId);
      break;
    ...
  }
});
```

Then we register our new `PostStore`, placing the call to `waitFor` before we access the `CommentStore` data.

```
PostStore.dispatchToken = Dispatcher.register(function(payload) {
  switch(payload.actionType) {
    case CommentConstants.CREATE_COMMENT:
      Dispatcher.waitFor(CommentStore.dispatchToken);
      // At this point it's guaranteed the `CommentStore` callback has been run
      // and we can safely access its data to summarize it.
      var commentCount = CommentStore.getCommentsFor(payload.parentPostId);
      PostStore.setCommentCount(commentCount, payload.parentPostId);
      break;
    }
  }
});
```

Reflux

The sample app uses a popular Flux-inspired library called Reflux. Reflux differs from Flux in that Stores subscribe directly to Actions. This eliminates the need for large switch statements that check an Action's type. Stores may similarly listen to other stores, eliminating the need for the `waitFor` call described in the Flux section. Action Creators are eliminated because Reflux Actions are simply functions that pass the payload the receive to listeners.

The sample app defines a number of stores in the directory `src/stores`. All of the actions for the app are defined in `src/utils/actions.js`.

You can find learn more about Reflux here: https://github.com/reflux/refluxjs.

Redux

Influenced by ClojureScript's Om and functional reactive programming, Redux is another Flux-inspired library for React applications. Redux is designed to help you write applications that are easy to reason about and to provide an excellent developer experience.

Similar to Om, Redux collects the entire application's state into a single object as a large tree of data. Redux is designed so that all of your code, including Stores, can be hot-loaded, or updated while your app is running.

Redux has a single Store that is updated by Reducers that respond to Actions. Redux Reducers are pure functions that return an updated piece of the state tree. Reducers are often very similar to the function registered with the Dispatcher by a Store. Instead of using getter methods on a Store as in Flux, Redux has a concept of Selectors. Selectors are functions that retrieve a piece of data from the state tree. Components then use a function called `connect` to register for updates

from the Store and use Selectors to retrieve the desired data. These selected pieces of data are given to the connected component as props. Action Creators in Redux are similar to those in Flux but, may be enhanced with Middleware.

You can learn more about Redux here: http://redux.js.org.

Relay

Relay is a data-driven framework for React applications created by Facebook. Relay makes it possible to specify the data a component needs as a GraphQL fragment, or query, within the component itself. When Relay renders your React app it builds a large composite query from all of the GraphQL fragments on components to be rendered and then makes one request to the server. Co-locating the data requirements and the component make keeping up with shifting data requirements simple, and batching network requests improves performance. Using Relay requires that you have a server that supports responding to GraphQL queries but, for applications with complex data requirements or view hierarchies it can be a great fit.

You can learn more about Relay here: https://facebook.github.io/relay/.

Om and Om Next (ClojureScript)

Om, and it's successor Om Next, are very popular ClojureScript interfaces to React. ClojureScript's persistent data structures make implementing otherwise complex features like undo very simple. Om Next incorporates what the Om community has learned to provide an even better experience. Similar to Relay, Om Next allows you to specify a component's data requirements directly on the component. In Om Next, these queries are specified using Datomic Pull Syntax which is a query syntax for Datomic, a database commonly used with ClojureScript projects.

Below is an entire Om Next app!

```
(ns om-tutorial.core
  (:require [goog.dom :as gdom]
            [om.next :as om :refer-macros [defui]]
            [om.dom :as dom]
            [datascript.core :as d]))

(def app-state
  (atom
    {:app/title "Animals"
     :animals/list
     [[1 "Crow"] [2 "Antelope"] [3 "Bird"] [4 "Cat"] [5 "Dog"]
      [6 "Lion"] [7 "Mouse"] [8 "Monkey"] [9 "Snake"] [10 "Zebra"]]}))

(defmulti read (fn [env key params] key))

(defmethod read :default
  [{:keys [state] :as env} key params]
  (let [st @state]
    (if-let [[_ value] (find st key)]
      {:value value}
      {:value :not-found})))

(defmethod read :animals/list
  [{:keys [state] :as env} key {:keys [start end]}]
  {:value (subvec (:animals/list @state) start end)})

(defui AnimalsList
  static om/IQueryParams
  (params [this]
    {:start 0 :end 10})
  static om/IQuery
  (query [this]
    '[:app/title (:animals/list {:start ?start :end ?end})])
  Object
  (render [this]
    (let [{:keys [app/title animals/list]} (om/props this)]
      (dom/div nil
               (dom/h2 nil title)
               (apply dom/ul nil
                      (map
                        (fn [[i name]]
                          (dom/li nil (str i ". " name)))
                        list))))))

(def reconciler
  (om/reconciler
    {:state app-state
     :parser (om/parser {:read read})}))

(om/add-root! reconciler
              AnimalsList (gdom/getElement "app"))
```

This renders an unordered list of the animal names defined as the `app-state` near the top of the file.

You can learn more about Om and ClojureScript here: https://github.com/omcljs/om.

Summary

By now you've seen a number of architectural patterns that can be used with React. From a standard Flux pattern, to Flux variants like Reflux, to an entirely different language and ecosystem like Om and ClojureScript, React has proven to be quite adaptable.

Up next you can read about using Immutability in your React apps, a concept central to both Redux and Om.

Chapter 16. Immutability

An increasingly popular way to use React is in conjunction with immutable data structures—that is, data structures that never change after they are instantiated.

Whether immutability is right for your application, and if so, which immutability library to adopt, depends on the specifics of your use case. In this chapter we will cover the benefits and costs of immutability, and look at three different libraries you can use to incorporate it into your React application.

Performance Benefits

The most clear-cut benefit of immutability in a React application is what it does for `shouldComponentUpdate`. Although `shouldComponentUpdate` can provide great performance improvements, those improvements are often eroded or even erased as the `shouldComponentUpdate` handler itself increases in complexity.

Suppose you have a `shouldComponentUpdate`, which only needs to check one field in `props` to determine whether it will return `true` or `false`. That's going to run quickly! Now suppose instead it needs to check a dozen fields instead, and some of those fields contain object which each require multiple comparisons, since all that checking can add up fast.

The root of the problem here is that it is time-consuming to determine whether two mutable objects represent equivalent values. But what if instead that were a quick check?

With immutable `props` and `state`, there is indeed a quick check you can use. When your `props` are represented as an immutable object, then they can no longer be updated in place; to get new props, you must instantiate a new immutable object and replace the old object with the new. As such, using `oldProps !== newProps` as your `shouldComponentUpdate` check will very quickly tell you if you definitely need to update.

It's worth noting that although this check is very fast, it can generate false positives. For example, if your old state were `{ username: "Don Jacko", active: true }` and you invoked `replaceState` passing a freshly-instantiated object `{ username: "Don Jacko", active: true }`, the old `state` would clearly be identical to the new `state`, , but a

`shouldComponentUpdate` function using `oldState !== newState` as its test would still consider them different, and would re-render unnecessarily.

In practice, this rarely seems to come up, and if it does, the only cost would be an unnecessary re-render. False negatives (in which a component actually needed to re-render but determined that it did not) would be a much greater concern, but fortunately that problem does not manifest with this approach.

Performance Costs

Although immutable data structures can get you dead-simple and lightning-fast `shouldComponentUpdate` implementations, which save considerable rendering performance, they are not without their performance costs. Whereas, mutable objects are quick to update but slower to compare, immutable objects are quick to compare but slower to update.

Instead of making the change in place, the immutability library you're using must instantiate at least one new object, and modify it as necessary until it matches the old object in every way except for the one changed field. This costs time both during the update process and during garbage collection, when the additional objects are inevitably cleaned up.

For the vast majority of React applications, this is an excellent tradeoff to make. Remember that a single update typically results in a call to many render functions, and that render functions instantiate quite a few objects themselves in the course of building up their return values. Sacrificing a bit of update speed to save numerous render function calls is almost always going to be a performance win.

Separate from performance, another cost to consider is that you lose the convenience of `setState` and `setProp`. Since both of these rely on being able to mutate your `state` and `props` objects, respectively, when you switch those to immutable objects, your only options become `replaceState` and `replaceProps`.

Architectural Benefits

Besides performance benefits, there are additional architectural benefits that come from using immutable data more. These benefits extend beyond `props` and `state`, and apply more broadly to your application as a whole.

Immutable data is generally less error-prone to work with than mutable data. When passing mutable data from one function to another, and expecting that it will come back unchanged, you have two options: one is to clone the object ahead of time and pass the clone, and the other is to cross your fingers and hope the other function doesn't modify it. (Because if it does, you're in for some bug hunting!)

Defensive cloning is effective in situations like these, but generally less efficient than what an immutable library would do in the same spot. Knowing that your initial data is already assumed to be immutable can allow for faster cloning, than when it's mutable the whole way through.

Embracing immutability means you do not have to remember to defensively clone on a case-by-case basis; instead, you will get the same safe-to-pass characteristics by virtue of using immutable data in the usual way. This makes immutable data less error-prone to work with.

Using the Immutability Helpers Addon

The easiest way to introduce immutability to your React application is to use the Immutability Helpers Addon. It lets you continue using your existing mutable data structures as though they were immutable, by making it easy to create new (also mutable) objects instead of performing in-place updates.

While this does not actually give you any new guarantees, it does allow you to write a quick `shouldComponentUpdate` function as described above.

Let's update our example to use the Immutability Helpers Addon:

```
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 });
  }

  // ...
});
```

Next we'll try a different library, one which actually does make guarantees about immutability.

Using seamless-immutable

The seamless-immutable library is not part of the official React family of libraries, but was designed to be used with React. It creates immutable versions of regular Objects and Arrays, which can be passed around, accessed, and iterated over just like their mutable counterparts, but which block any operations that would mutate them.

Like the Immutability Helpers Addon, seamless-immutable provides convenience functions for working with immutable data. In the case of seamless-immutable, however, these are implemented as methods on the objects themselves. For example, seamless-immutable objects gain a `.merge()` method which works like the `$merge` option in the Immutable Helpers Addon.

```
var update = React.addons.update;

var SurveyEditor = React.createClass({
  // ...

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

    this.replaceState(this.state.merge({
      questions: questions,
      dropZoneEntered: false
    }));
  },

  handleQuestionChange: function (key, newQuestion) {
    var questions = this.state.questions.map(
        function(question, index) {
            return index === key ? newQuestion : question;
        }
    );

    this.setState({ questions: questions });
  },

  handleQuestionRemove: function (key) {
    var questions = this.state.questions.filter(
        function(question, index) {
```

```
            return index !== key;
        }
    );

    this.setState({ questions: questions });
  }

  // ...

});
```

Using Immutable.js

Whereas the previous two libraries provided tools around normal JavaScript objects and arrays, Immutable.js takes a different approach: it provides alternative data structures to objects and arrays.

The most commonly-used of these data structures are `Immutable.Map` (analogous to Object) and `Immutable.Vector` (analogous to Array). Neither of these can be freely substituted for JavaScript objects or arrays in the general case, although you can easily convert to and from the Immutable.js data structures and their mutable JavaScript counterparts.

Immutable.Map

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

```
var question = Immutable.Map({description: 'who is your favorite superhero?'});
// get values from the Map with .get
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?');

// merge 2 objects with .merge to get a third object.
// Once again none of the original objects are mutated.
var title = { title: 'Question #1' };
var question3 = question.merge(question2, title);
question3.toObject(); // { title: 'Question #1', description: 'who is your favorite comicbook hero' }
```

Immutable.Vector

Use `Immutable.Vector` for arrays:

```
var options = Immutable.Vector('Superman' , 'Batman');
var options2 = options.push('Spiderman');
options2.toArray(); // ['Superman', 'Batman', 'Spiderman']
```

You can also nest the data structures:

```
var options = Immutable.Vector('Superman' , 'Batman');
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.

Summary

In this chapter we learned about the benefits and costs of using immutability in a React application. We covered its impact on `shouldComponentUpdate`, on what it means for update times, and three different ways to introduce immutability into a React code base.

Next we'll look into other uses of React beyond traditional web applications.

Chapter 17. Other uses

React is a powerful interactive UI rendering library, but it provides a great way to handle data and user input. It encourages small components that are reusable and easy to unit test. These are all great features that you 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

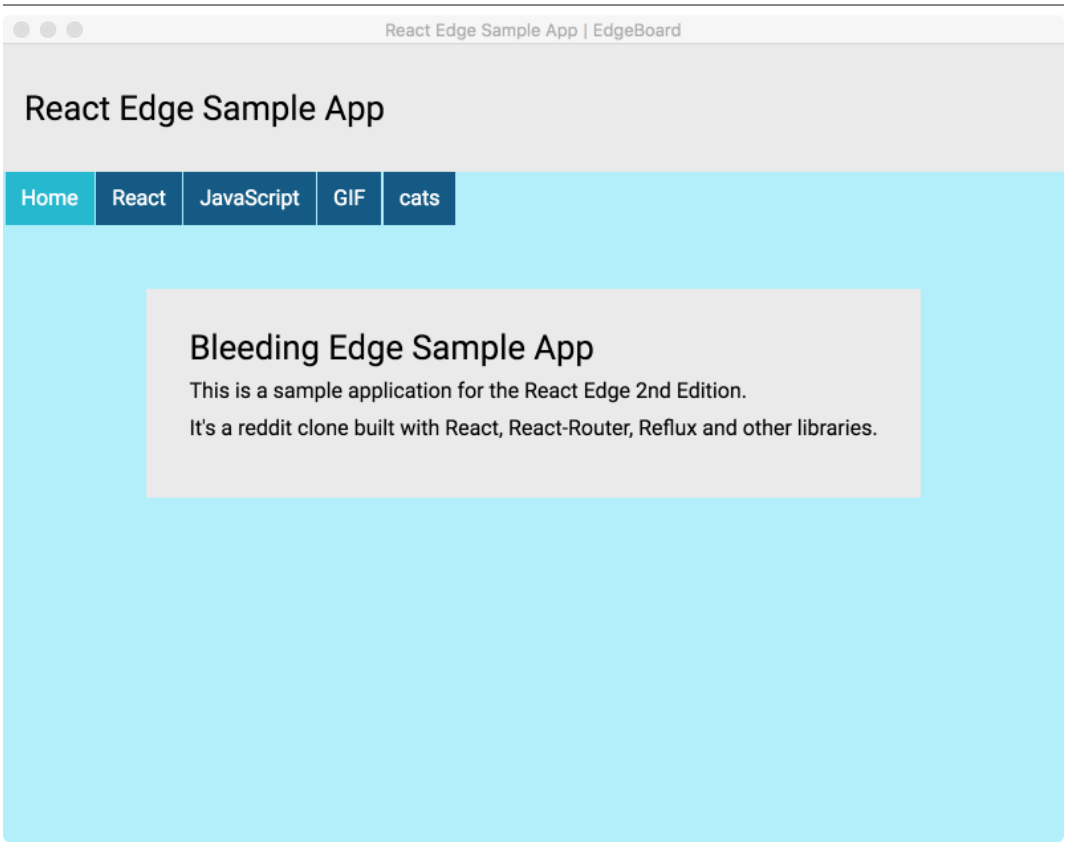
Desktop

With projects like Electron or node-webkit you can run a web application on the desktop. The Atom Editor from Github is built with Electron, which you can also use with React.

The sample app already includes Electron, and to launch it, run this command in the root of the sample app:

```
./node_modules/.bin/electron desktop.html
```

Running Electron with `desktop.html` opens up the application in a window.



You can read more about Electron at http://electron.atom.io.

With projects like Electron and node-webkit, you can build desktop applications using the same technologies you use for the web. Just like for the web, React can help you build powerful interactive applications for the desktop.

Games

Games generally require a high level of user interaction, where the user is responding to the changing state of the game. This is in contrast to most web applications where the user is either consuming content or producing content. Games are essentially a state machine, having two basic roles:

1. Updating the View

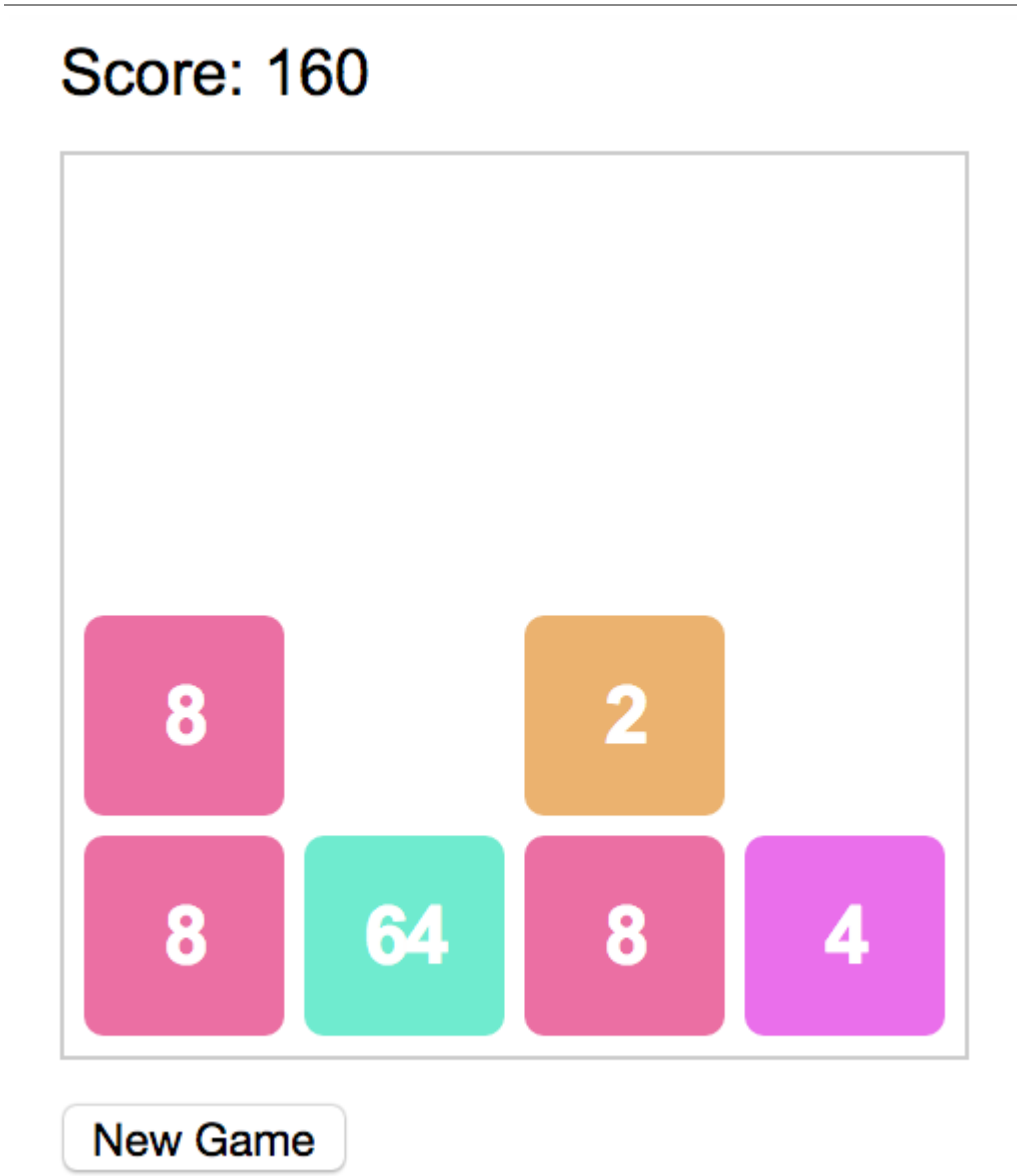
2. Responding to events

In the introduction to this book, you read that React has a very narrow scope that is concerned with only two things:

1. Updating the DOM
2. Responding to events

The similarities between React and Games don't just stop at this. React owes much of its Virtual DOM architecture to high performance 3D gaming engines, where given the intended view state, the rendering engine ensures an efficient update of view/DOM.

As an example of using React with gaming, let's take a look at an implementation of the game 2048. The object of this game is to combine matching numbers on the board until you reach 2048.



Let’s dive in and take a look at the implementation (https://jsfiddle.net/jeremiahrhall/6wa8djju/).

The source is divided into two parts. The first section is the game logic implemented as functions in the global namespace, and the second section is the React components. The first thing you will see is the initial data structure of the board.

```
var initial_board = {
    a1:null, a2:null, a3:null, a4:null,
    b1:null, b2:null, b3:null, b4:null,
    c1:null, c2:null, c3:null, c4:null,
    d1:null, d2:null, d3:null, d4:null
};
```

The data structure of the board is an object where the `keys` directly relate to a visual grid position defined in CSS. Following the initial data structure, you will see a series of functions that operate on this given data structure. All of these functions operate in an immutable way, thus returning a new board and not mutating the input. This gives the game logic some leverage, since it will be able to compare boards before and after moves are performed, and speculatively perform moves without changing the game state.

Another interesting property about the data structure is the structural sharing of boards. All boards share reference to unchanged tiles on the boards. This makes creating a new board very fast and allows the comparison of boards to use reference equality.

The game is comprised of two React components: `Tiles` and `GameBoard`.

`Tiles` is a basic React component. When given a board as `props` it will consistently render all tiles. This allows you to leverage CSS3 transitions for animation.

```
class Tiles extends React.Component {
  render() {
    var board = this.props.board;
    //sort board keys first to stop re-ordering of DOM elements
    var tiles = used_spaces(board).sort((a, b) => {
      return board[a].id - board[b].id;
    });
    return <div className="board">{
      tiles.map((key) => {
        var tile = board[key];
        var val = tile_value(tile);
        return <span key={tile.id} className={key + " value" + val}>
          {val}
        </span>;
      })}
    </div>
```

```
  }
}

<!-- Example output from render of Tiles -->
<div class="board" data-reactid=".0.1">
  <span class="d2 value64" data-reactid=".0.1.$2">64</span>
  <span class="d1 value8" data-reactid=".0.1.$27">8</span>
  <span class="c1 value8" data-reactid=".0.1.$28">8</span>
  <span class="d3 value8" data-reactid=".0.1.$32">8</span>
</div>

/* CSS transistion applied to animate tiles */
.board span{
  /* ... */
  transition: all 100ms linear;
}
```

`GameBoard` is the state machine, responding to user events `Arrow Keys` and `Button Press`, and interacting with game logic functions to then update the state with a new board.

```
class GameBoard extends React.Component {
    constructor(props) {
      super(props);
      this.state = this.addTile(this.addTile(initial_board));
    }

    keyHandler(e) {
        var directions = {
            37: left,
            38: up,
            39: right,
            40: down
        };
        if (directions[e.keyCode]
        && this.setBoard(fold_board(this.state, directions[e.keyCode]))
        && Math.floor(Math.random() * 30, 0) > 0){
            setTimeout(() => {
                 this.setBoard(this.addTile(this.state));
            }, 100);
        }
    }

    setBoard(new_board) {
        if (!same_board(this.state, new_board)){
            this.setState(new_board);
            return true;
        }
        return false;
    }

    addTile(board) {
        var location = available_spaces(board).sort(() => {
          return .5 - Math.random();
        }).pop();
        if (location) {
            var two_or_four = Math.floor(Math.random() * 2, 0) ? 2 : 4;
            return set_tile(board, location, new_tile(two_or_four));
        }
```

```
        return board;
    }

    newGame() {
        this.setState(this.addTile(this.addTile(initial_board)));
    }

    componentDidMount() {
        window.addEventListener("keydown", this.keyHandler.bind(this),
          false);
    }

    render() {
        var status = !can_move(this.state)?" - Game Over!":"";
        return <div className="app">
            <span className="score">
                Score: {score_board(this.state)}{status}
            </span>
            <Tiles board={this.state}/>
            <button onClick={this.newGame.bind(this)}>New Game</button>
        </div>
    }
}
```

In the `GameBoard` component we setup the keyboard handlers for interacting with the board. Every time an arrow key is pressed we call `setBoard` with the newly created board from the game logic. If the new board is different than the old, we update the state of the `GameBoard` component. This avoids unnecessary cycles and improves performance.

In the render function you render the Tiles component given the current board, and render the score, calculated by the current board in the game logic.

The `addTile` function keeps adding new tiles to the board when you use the arrow keys to keep the game going until the game is over when the board is full and no numbers can be combined.

Given the above implementation, it is trivial to extend the game with undo functionality. You can keep a history of all board changes in state within the `GameBoard` component and with an undo button change the current board: https://jsfiddle.net/jeremiahrhall/9fbjgnt6/.

The implementation of this game is quite simple, since React allows you to concentrate on the game logic and user interactions without having to worry about keeping the view in sync.

Email

Though React is optimized for building interactive UIs for the web, at it's core it renders HTML. This means you can get a lot of the same benefits you'd normally get from writing a 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, whether they are rendered using React or not.

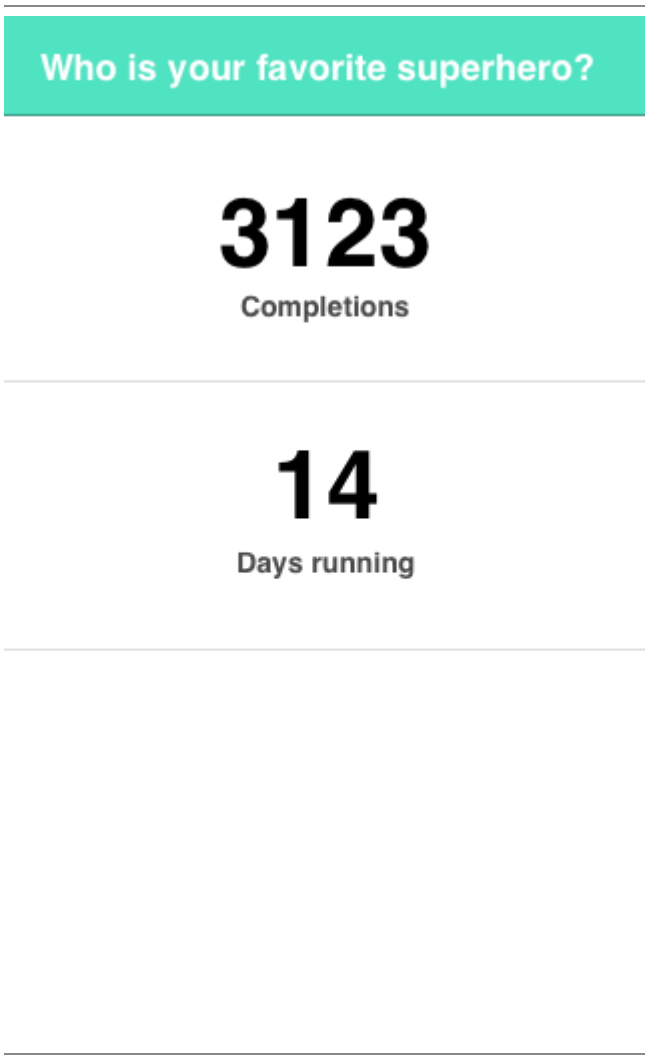
The core principle of rendering HTML for emails with React is `React.renderToStaticMarkup`. This function returns an HTML string containing the full component tree, given 1 top level component. The only difference between `React.renderToStaticMarkup` and `React.renderToString` is that `React.renderToStaticMarkup` 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 the email we have made a small script that outputs HTML that can be used to send an email:

```
// render_email.js
var ReactDOMServer = require('react-dom/server');
var SurveyEmail = require('survey_email');
var survey = {};

console.log(
  ReactDOMServer.renderToStaticMarkup(<SurveyEmail survey={survey}/>)
);
```

Let's get the core structure of `SurveyEmail` going. First, let's build an `Email` component:

```
class Email extends React.Component {
  render() {
```

```
    return (
      <html>
        <body>
          {this.prop.children}
        </body>
      </html>
    );
  }
}
```

The `<SurveyEmail/>` component uses `<Email/>`:

```
class SurveyEmail extends React.Component {
  render() {
    var survey = this.props.survey;
    return (
      <Email>
        <h2>{survey.title}</h2>
      </Email>
    );
  }
}

SurveyEmail.propTypes = {
  survey: React.PropTypes.object.isRequired
};
```

Next, per the design, render two KPIs next to each other on desktop clients and stacked on a mobile device. Each KPI looks similar in structure so they can share the same component:

```
class SurveyEmail extends React.Component {
  render() {
    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:

```
class SurveyEmail extends React.Component {
  render() {
    var survey = this.props.survey;
    var completions = survey.activity.reduce((memo, ac) => {
      return memo + ac;
    }, 0);
    var daysRunning = survey.activity.length;
```

```
    return (
      <Email>
        <h2>{survey.title}</h2>
        <KPI kpi={completions} label='Completions'/>
        <KPI kpi={daysRunning} label='Days running'/>
      </Email>
    );
  }
}

SurveyEmail.propTypes = {
  survey: React.PropTypes.object.isRequired
};
```

This stacks our KPIs, but the design had them next to each other for the desktop. The challenge now is to have this work for both desktop and mobile. 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');
class Email extends React.Component {
  render() {
    var responsiveCSSFile = this.props.responsiveCSSFile;
    var styles;
    if (responsiveCSSFile) {
      styles = <style>{fs.readFileSync(responsiveCSSFile)}</style>;
    }
    return (
      <html>
        <body>
          {styles}
          {this.prop.children}
       </body>
     </html>
    );
  }
}

Email.propTypes = {
  responsiveCSSFile: React.PropTypes.string
};
```

The complete `<SurveyEmail/>` looks like this:

```
class SurveyEmail extends React.Component {
  render() {
    var survey = this.props.survey;
    var completions = survey.activity.reduce((memo, ac) => {
      return memo + ac;
    }, 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>
    );
  }
}

SurveyEmail.propTypes: {
  survey: React.PropTypes.object.isRequired
};
```

We grouped the Email into for-desktop and for-mobile. Sadly you 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 therefore React doesn't support those properties. They could have, however, provided a similar implementation to floating two `div`s. Instead we are left with two groups to target with responsive style sheets to hide or show depending on the screen size.

Even though you have to use tables, it's clear that using React for rendering emails gives you a lot of the same benefits from writing interactive UIs for a browser's Reusable, composable, and testable components.

Charting

React has support for SVG tags and thus making a simple SVG becomes trivial.

To render a Sparkline (that we'll use as an example), you need only a `<Path/>` with a set of instructions.

The complete example looks like this:

```
class Sparkline extends React.Component {
  render() {
    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(width, height, points) {
    var maxHeight = arrMax(points);
    var maxWidth = points.length;

    return points.map((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(' ');
  }
}

Sparkline.propTypes = {
  points: React.PropTypes.arrayOf(React.PropTypes.number).isRequired
};
```

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 that 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 translate 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 like so:

```
class Sparkline extends React.Component {
  render() {
    var width = 200;
    var height = 20;
    var points = this.props.points.map((p, i) => {
      return { y: p, x: i };
    });

    var xScale = d3.scale.linear()
      .domain([0, points.length])
      .range([0, width]);

    var yScale = d3.scale.linear()
      .domain([0, arrMax(this.props.points)])
      .range([height, 0]);

    var line = d3.svg.line()
      .x(function (d) { return xScale(d.x) })
      .y(function (d) { return yScale(d.y) })
       .interpolate('linear');

    return (
      <svg width={width} height={height}>
        <path d={line(points)} stroke='#7ED321' strokeWidth='2' fill='none'/>
      </svg>
    );
  }
}

Sparkline.propTypes = {
  points: React.PropTypes.arrayOf(React.PropTypes.number).isRequired
};
```

Summary

In this chapter we learned:

1. React isn’t just about the browser, and it can be used to build desktop applications and emails.
2. How to use React to aid in game development.
3. React is a great choice for charting and works very well with libraries like D3.

You have now completed this book, and should be able to use React to create all types of interesting applications.