Effective

# React

*Ian Chursky*

# *Effective React*

Ian Chursky

# *Contents*

## *Acknowledgements*

Big thanks to all of my family and friends for their continued love and support over the years.
Big thanks to all of my teachers and colleagues who have invested their time, effort (and patience) with me in so many ways. I would not be where I am today without all of your help.
Biggest thanks of all to my wife Lauren and my 2 daughters: Mia and Calista. Love you all so much!

# Preface

Have you ever fired up your favorite web browser and gone to purchase airline tickets? Perhaps there was a visual model of a plane where you could click on the exact seat and aisle that you wanted (of those that were available). Or maybe you've signed in to your favorite photo-sharing or music-streaming website and you used a UI to create a photo album or a playlist of songs by dragging and dropping elements on to different containers. If you have done so at any time in the second decade of 21st century (the 2010s) you have likely noticed that a lot of these services incorporate a high degree of interactivity. You click things, you drag and drop things and a lot of interesting things happen in response. What's more, things that you do in the application in the present will remain reflected in the application the next time you return to it in the future. Your data is somehow saved and persisted on the server behind the scenes.

However you use the web, chances are you have used an application built with technology that implements this interactive functionality. React, the view rendering library developed by a team of engineers at Facebook, is part of the technologies that make these sorts of applications possible. That's what this book is about. But first, a little bit of history on how we got here.

## History

Beginning at around 2009 - 2010 web pages on the modern Internet started undergoing rapid evolution in many significant ways. While the web has always been a dynamic, ever-changing environment, one of the more noticeable developments was the rise of websites and web applications making use of more open technologies such as HTML5 and JavaScript to bring interactivity and 2-way communication between clients (usually browsers) and servers. In the early 2000s, when it came to interactive applications within browsers, Adobe Flash dominated the Internet landscape. Flash, however, was ultimately a problematic implementation for the open nature of the web in a number of ways. Many of the problems that Flash brought about were summarized by the co-founder and CEO of Apple Steve Jobs in an open letter called "Thoughts on Flash" published on April 29, 2010. Jobs explained why Apple would not allow Flash on the iPhone, iPod touch and iPad. He cited the rapid energy consumption (Flash had always been a bit of a resource hog), poor performance on mobile devices, abysmal security, lack of touch support, and desire to avoid "a third party layer of software coming between the platform and the developer" (Flash runs as a browser plugin). He touched on the idea of Flash being "open", claiming that "By almost any definition, Flash is a closed system". This spelled the beginning of the end of the dominance of Flash on the web.

So with Flash on the way out, something had to fill the void. A lot of organizations returned to just the regular old HTML website, but there was definitely a place for web applications especially with the ever accelerating growth of the mobile web. And this is where the "single page application" (SPA) built using open technologies (HTML5 and JavaScript) entered the fray. A single-page application, could be described as web application or web site that fits on a single page. All the necessary code -- usually HTML, JavaScript, and CSS -- is retrieved with a single page load and any additional resources that are needed are dynamically loaded and added to the page, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page. Interaction with the single page application often involves dynamic communication with the web server behind the scenes. There is a diverse and flourishing ecosystem of JavaScript libraries that drive the functionality of these single page applications. React is part of this ecosystem.

Circa 2010 onward there were a number of JavaScript application libraries that were emerging around this time in the web/JavaScript development community. These frameworks included AngularJS, Ember.js, Backbone.js, Knockout.js, and a number of others. These JavaScript libraries could be loosely described as client-side "model-view-whatever" frameworks for creating single-page applications. They all loosely follow the popular model-view-controller design pattern and they all seem to have model component for storing data and a view component for displaying data, but the component in between varies from framework to framework... Backbone has collections, Angular has directives, Knockout has view models, and so on. This is where the "whatever" term comes from. It is sometimes abbreviated MVW or MV*, with the * being a wildcard.

Fast forward a few years after this, Facebook open sources their React JavaScript library and it comes on to the scene with a fairly positive response from the JavaScript world.

## What is React?

React is a JavaScript library that is primarily responsible for handling the "view" component of a JavaScript single-page application. As it describes itself of the React homepage...

> *Lots of people use React as the V in MVC. Since React makes no assumptions about the rest of your technology stack, it's easy to try it out on a small feature in an existing project.*

React's focus, first and foremost, has to do with rendering dynamic UIs in a manner that is visually appealing to the user. It is flexible enough that it can be used in a diversity of capacities and contexts. We mentioned earlier that there are a number of different MVW/MV* JavaScript frameworks and all of them have different implementations in how they handle the various aspects of a JavaScript application. Naturally they all have different ways that they handle rendering visual representations of the data that the application is handling. Backbone.js natively uses Underscore.js templates, Ember users Handlebars.js, and Knockout.js and Angular have their own syntax. With a little bit of work you can swap view engines within these frameworks if you so choose. And it would be here where we could incorporate React to handle the view rendering portion of an application. But, as mentioned above, React is not dependent on any JavaScript framework being in place. It can be used as a standalone aspect of an application almost entirely independent from any other library or environment.

So why go to the trouble to incorporate React? Why not just use a framework's native rendering engine (if you are using a framework). If you are not using a framework, why not just use something like jQuery to handle your view layer? Again, looking at the React homepage, the following description is provided...

> *React abstracts away the DOM from you, giving a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native.*

At the end of the day, React is a library for building large applications with data that changes over time and React will automatically manage all your UI updates when the underlying data changes. But, as was mentioned, it does not do this by interacting entirely with the DOM as other JavaScript libraries and applications do. That aspect is abstracted away and only engages the the DOM when it is absolutely necessary. React does this abstraction by using something known as the virtual DOM. What is the virtual DOM? Without getting too deep into the technical details of it right away, you could perhaps think of the virtual DOM it as a regular JavaScript object in object literal notation. This object in its keys and values will contain a reference to every single node in the actual DOM somewhere within it. React will constantly be watching some data object associated with a virtual DOM component on a loop. When that data changes React can quickly get that element (because all the references are stored in the virtual DOM object) and instantly update the actual associated DOM node. There is no need to engage any kind of lookup in the DOM by crawling through it with the regular JavaScript APIs such as document.getElementById or document.getElementsByClassName. All of that is done ahead of time and React does this using a very efficient algorithm that makes it much faster than JavaScript libraries and/or applications that interact with the actual DOM. For smaller applications, the difference between these two approaches will not be tremendously noticeable. But as applications grow and mature and start handling larger and larger sets of data, there are a number of different positive aspects in terms of performance, scalability, and simplicity that React can provide to application development as it progresses along in both size and complexity. We will explore these in detail as we go forward.

As many have noticed, React is a little bit unorthodox from "traditional" JavaScript application development (if there were such a thing) and it can take a little bit of working with it and a little bit of trial and error before many of the benefits of it are fully realized. Indeed, many developers admit that they did not quite "get it" at first and some have even described a substantial initial aversion to some of its implementations. So if you experience this on any level, you are definitely not alone and I would merely invite you to "stick with it" through some of the initial

tutorials before you run for the hills screaming in terror. Who knows? You just might be surprised at the end of the day.

...Or maybe you won't. Regardless, I believe it will still be worthwhile to explore a different approach to implementing a solution to common problems. That, really, is how you learn and grow as a developer and how you can decide what you like and what you don't. Like with so many things in technology and software development, there are pros and cons to everything. But even if React is not your cup of tea in the end, it is not outside the realm of possibility that you may one day inherit a project that uses React in some capacity and you will (hopefully) be glad that you read through the discussions and tutorials that are to follow should that day ever come. So at least on that level it could be worth your while.

As of mid 2015, there are many different well known websites that use React including Facebook, Imgur, Bleacher Report, Feedly, Airbnb, SeatGeek, and HelloSign. So there might be something to this library after all.

## Who This Book Is For

This book is for the developer who maybe has some experience with writing a bit of JavaScript or jQuery code but has not yet ventured into the waters of single page application MVW (model-view-whatever) frameworks or just wants to see how React might be able to provide some benefit to new and/or existing technology stacks. This book will provide you the essential basics of creating a diversity of applications in React and in the process you hopefully will be able to get a grasp of the concepts of building within this sort of framework. This knowledge is definitely helpful when moving to other frameworks of similar type.

This book is also for the intermediate or seasoned JavaScript application developer who has maybe worked with other frameworks but wants to get a grasp of the concepts involved with working on a React application and needs a quick reference of working code and a high-level overview of the common components therein. This book will give you the essentials to get up and going (hopefully as fast as possible).

Please note that this book will not cover a basic introduction to HTML and JavaScript. The content in this book assumes that you have had at least some experience working with HTML, JavaScript, and jQuery – though, honestly, you really do not need all that much. If you do not really know these technologies at all, it would be of great benefit for you to take a few online tutorial courses before returning to what is presented here. W3Schools is a very popular choice, but there are many others as well. All you really have to do these days is go to Google or YouTube and type in "learn X," where X is whatever subject you want to learn.

This book also assumes that you have a basic understanding of how the web operates and that web pages and resources (CSS files, images, cookies, etc.) are sent from server to client over HTTP(S) and the client (i.e. browser) can send data back to the server (e.g. submitting forms) via the same protocol.

It is also assumed that you have at least a very basic understanding of how AJAX works using JSON as a data-exchange format.

## Required Tools

In working with React there are a number of different tools that we will make use of to accomplish some of the tasks we are setting out to do. This section describes some of the tools we will be using as we go along.

### *Node.js*

We will utilize Node.js in a few different contexts in our work with React. Having been first published by Ryan Dahl in 2009, Node.js -- the open source software that allows the common web language of JavaScript to run outside the browser -- has absolutely grabbed ahold of imaginations in the world of technology in its first half-decade of life. The explosion of Node.js applications and the exponentially increasing number of published modules of npm, the repository where Node.js modules are stored, are a huge indicator of its success as a platform. Circa midway through the 2010s decade, so much of the web development world uses Node.js in some capacity. React is primarily a front-end development framework, and Node.js very prevalent even in front-end development with task runners like Grunt and Gulp, JS Lint, LESS and SASS compilers and a host of other tools. All of these run on Node.js.

So be sure head over to the Node.js website and install it for your OS if you have not done so already. You will definitely be needing it as we go along. When you install Node.js you automatically get npm installed along with it, so there is no further setup that you need to do. We will use npm for installation of the various dependencies and modules we will use. If you are not entirely familiar with Node.js and npm, don't worry! We will provide instructions on how to work with it when the time comes. Though it is not required at this point, if you wanted to familiarize yourself with Node.js and get an idea of what it is and why it has become so widespread within the web development community, it might be worthwhile to take a look at a few basic introductory tutorials or videos out on the web.

### *Command Line Shell/Terminal*

In our work with Node.js we will be utilizing the command line, so be sure you have a basic idea of how to work with your favorite terminal for your OS. I like BASH.

### *Local Web Server*

We will also want to run our various projects inside of a localhost HTTP web server. If you have a local Apache or IIS setup you may wish to use this by putting the files that you create in the upcoming tutorials in a directory created inside the www folder. Another quick easy implementation is to just pull down the Node.js http-server module by running the following from your command line...

```
$ npm install -g http-server
```

From there you can create a directory anywhere you like and run the following...

```
$ http-server
```

This will run an HTTP server at localhost:8080. If you got to http://localhost:8080 you should see something there that tells you a server is running.

### *JSX Compilation with Babel & Browserify*

As we will see, React has its own programming language called JSX. It has been described as sort of like a mixture of JavaScript and XML and it is React's own special easy-to-read abstract syntax. JSX is not required to create React applications and you can write your React components in native JavaScript if you prefer. However, our discussions will utilize JSX substantially because of the current reality that JSX is the preferred implementation embraced and utilized by much of the React community. Many of the examples you see in tutorials and code you see within React based libraries rely on JSX. Because it is so ubiquitous, we will use it for our discussions as well.

Because browsers do not natively understand JSX, in order for your React application to run JSX within your browser it needs get converted to native JavaScript. The React website suggests that you utilize tools like Babel and/or Browserify (both of which run on Node.js). Babel is what is known as a transpiler. It mostly functions as a tool that takes code written in ECMAScript 6 (ES6) and converts it to ES5. As of early 2016 this is still necessary because support for ES6 across all browsers is not 100% supported yet. Until support for ES6 is implemented across all browsers, tools like Babel will still be needed. Although many people think of Babel primarily as an ES6 to ES5 transpiler, Babel also comes with JSX support as well. You can merely run the Babel transpiler on your JSX files and they will be converted to native JavaScript that will be recognized by any browser circa 2015.

Browserify is a tool that allows for Node.js style module loading in your client-side JavaScript. Like with Node, it allows you to import code in certain JavaScript files into other JavaScript files for better organization of your code.

### *Task Runners / Module Bundlers*

For our sample code, we will be using task runners to handle our JSX conversion to native JavaScript. You may have heard of task runners like Grunt or Gulp, or module bundlers like webpack. We will be using webpack because it has, at least for the time being, become the utility of choice among the React community. If you are not entirely familiar with JavaScript task runners it would definitely be worthwhile to read the webpack documentation (or

watch a few videos) on what it is and why you would want to use one. The more recent versions of React essentially require you to set up your tools before you work with JSX in React. In earlier versions of React there was an easy extra <script> tag that you could add in your HTML to make it easy to implement JSX right within your web application. However this has since been deprecated. If this seems daunting and you are already feeling a bit overwhelmed, not to worry! We will walk through how to set everything up before we dive into React coding. And code samples are provided in Appendix A if you need a visual example of how things should look.

### *ECMAScript 6*

React is a modern library that uses technology that is on the cutting edge of approaches used within the JavaScript community. ECMAScript 6, or ES6 for short, also known as ECMAScript 2015, is for lack of better terminology "the next version of JavaScript." ECMAScript 6 involves a significant number of functional and syntactical refinements for the JavaScript language since the previous version ECMAScript 5. Almost exactly halfway through the second decade of 21st century (June 2015), specifications around ECMAScript 6 have more-or-less been finalized. However, it is important to note that this does not mean that support for ECMAScript 6 has been entirely implemented across all browsers (at least not as of early 2016). Different bowsers (Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, and Opera) have been gradually been adding support for different pieces of ECMAScript 6 with each successive release. But as of the time of this writing, to use ECMAScript 6 the general approach is still to use tools to convert your ES6 code to ES5 via transpilers like Babel so that your code will run and function properly in all browsers that may not fully support all of the components of ES6. Things will change in time and eventually the day will come when transpiling tools may no longer be needed.

The important thing to take away from all of this is that ECMAScript 6 is coming and in many ways it is already here. Thus, we will be writing our code in ECMAScript 6 because it will be the syntax that nearly all JavaScript code will be written in before too long. If you are not entirely familiar with ES6, again, like with JSX, not to worry. We will cover the aspects of ES6 that to will be relevant to React as we go along. In that sense, you will receive a fairly solid introduction to the "next version of JavaScript" and if you are a web developer by any stretch it is material that you are going to have to be familiar with at some point before too long as the technology moves forward.

## Errata

While I and others have done our best to have proofread the content that is to follow, it is likely that there are a fair number of mistakes that still exist within. You can send me any notifications of such errors via e-mail: books@9bitstudios.com. Errors found will be fixed with future updates of the book, which I will try to churn out as fast as possible.

## Work in Progress

At the time of this writing in mid-2015, this book is still a work in progress and is not 100% complete. But fortunately if you have purchased this book you will receive all future updates to this book at no additional cost! You should be notified of future updates via e-mail through Leanpub. Also note that React is currently pre version 1.0. This means that many of the concepts, code samples, subject and chapter order and other components presented in this book are subject to change. I will do my best to keep up to date on the latest changes as they roll out, but please note that there might be some sensitivity involved with functionality between different versions of React and the surrounding tools. Just be sure to be on the lookout for updates to this book as things continuously move and evolve. Also of note, if you would like to see a particular subject related to React covered in a future chapter, please let me know by dropping me a line at books@9bitstudios.com.

## Feedback

Also, feel free to send additional feedback of any kind via email: books@9bitstudios.com. Just make sure to reference in the subject line your e-mail has to do with this book. I will do my best to respond to all those who make the effort to contact me. I am also on the Twitters @ianchursky and @9bitStudios so if you want to leave 140 characters of feedback (or just say hi) you can contact me there as well.

So, without further ado, let's jump right in with an introduction to React.

# Chapter 1: Getting Started With React

React is a view rendering JavaScript library that is created and maintained by a team of engineers at Facebook. It is responsible for dynamically handling the visual display of various components within a web site or web application in a flexible, efficient, and interactive manner. What this often means is that React handles the displaying of the data that underlies an application and will dynamically update everything that it needs to in a performance conscious manner when that data changes. In other words, it will ***react*** to when data changes and events occur. We will look at all of React's various components that come together to make this happen in greater detail as we go along in the upcoming tutorials. But before we dive into React, we will need to set up the tools we will need for our React development workflow.

## Setting Up: Local Web Server, Task Runners & JSX Transpiler

In this section in what follows we will set up the files and utilities we will need to start developing in React. Remember that we want to run our React examples inside of a localhost web server. So create a new directory where you are going to be doing your React development. You can name the folder whatever you want. We will need to have a way to set up a local web server to serve the files in this folder. Install the "http-server" module globally from npm with $ npm install -g http-server (if you have not yet done so in the preface).

As was mentioned earlier we will need to turn to Node.js based build tooling to handle the process of converting our JSX to JavaScript. To set up our Node.js tooling there are number of different modules that we will want to install. What are Node.js modules? Modules are essentially pieces of code that you can import into your own code, tools, or projects that will do some sort of utility function for you. Modules can be as simple as a function that will do simple math calculations like addition and subtraction for you all the way up to something as complex as an entire web application framework (like Express.js). In short, modules consist of standalone independent pieces code that is designed to be used together with other code. By breaking code up into smaller pieces it's easier to make changes and scale

To install Node.js modules we use a utility called "npm" that is part of the Node.js installation (so you already have it installed because you have Node installed). We are going to specify which modules we want to install by creating a package.json file and putting it in a new folder. In this package.json file, add the following...

```
{
    "name": "React",
    "version": "0.0.1",
    "description": "A React application...",
    "license": "MIT",
    "repository": {
        "type": "",
        "url": ""
    },
    "homepage": "http://www.9bitstudios.com",
    "dependencies": {
        "react": "15.4.0"
    },
    "devDependencies": {
        "webpack": "2.2.1",
        "babel-core": "6.13.0",
        "babel-loader": "6.2.4",
        "babel-preset-es2015": "6.13.0",
        "babel-preset-react": "6.11.1"
    }
}
```

Basically the first part of this file defines some informational metadata that is not tremendously important for our purposes. This information starts to matter a bit more if you ever want to publish your own modules on npm. It is how other people will be able to find your file and install the version that they want (you can specify the version of the module that you want to install when you are using npm). The important info for us are the listings under the "devDependencies" object. From this we can see that we are going to install a number of modules: "webpack",

"babel-core", "babel-loader", "babel-preset-es2015", and "babel-preset-react" (all of which will be used to assist in JSX compilation).

Notice that we have specified versions of each of these modules that we want to install. npm actually has ways of allowing us to set wildcards so we can install different latest major and minor versions. For example, if we just wanted to install the latest version of all of these modules we could have used the "*" character like so...

```
{
    "name": "React",
    "version": "0.0.1",
    "description": "A React application...",
    "license": "MIT",
    "repository": {
        "type": "",
        "url": ""
    },
    "homepage": "http://www.9bitstudios.com",
    "dependencies": {
        "react": "*"
    },
    "devDependencies": {
        "webpack": "*",
        "babel-core": "*",
        "babel-loader": "*",
        "babel-preset-es2015": "*",
        "babel-preset-react": "*"
    }
}
```

While we certainly could do things this way it is important to emphasize that **setting specific versions is the preferred way of doing things**. Why is this? Sometimes the version you install matters as certain versions might only be compatible with certain versions of Node.js and/or other modules. It is a good idea to read the docs of each individual module to familiarize yourself with the current version you are installing. If you decide to put all * indicators for every module, it might work fine at one particular point in time, but subsequent updates to any number of modules may cause certain versions to not maintain compatibility between each other. If you run into errors, you have to spend time tracking down the root cause/problematic module(s). It is far more preferable to gradually update modules to the latest version one by one (if you do not have the latest version) so that can know where the problem first arises and work toward a solution. Managing versions and updates is something that is important in any sort of software development and it is important to approach things in a manner that will cause you the least amount of headache down the road. So for the remainder of our discussions and examples we will be using specific versions. These will be periodically updated as the versions of these tools progress in development.

To install these modules all we have to do now is open a terminal/command window in the same directory as our package.json file and type the following...

```
$ npm install
```

Note: You may have to type $ sudo npm install if you are on Linux.

Node.js will look for the package.json file in the same directory and pull down all the dependencies you have specified and place them all in the node_modules folder!

So judging from the modules we are installing, to do our ES6/JSX compilation we are going to utilize a utility that runs on Node.js called webpack which is a JavaScript module bundler that can also take on some task running capabilities as well. Essentially, it is something that automates common development processes for you. For example, one of the common tasks that a developer will do before deploying their code to production is to minify all of his or her CSS and JavaScript to reduce the file size in an attempt to decrease the page load time as much as possible. You can configure webpack to create a minified version of your CSS and/or your JavaScript. We will look at how to do this later on. It can do a lot of other tasks as well such as concatenating files, running your files through code checking-tools like JSLint, compiling files, copying files, deleting files, and a whole number of other things.

You could configure things to do pretty much whatever you wanted to provided that you had the right modules installed. There are other task runners out there such as Grunt and Gulp but we will be using webpack because it has become a very common utility among the React development community. You will see it used in many different React projects and demos so it is definitely worthwhile to become familiar with it.

The way that webpack works is primarily through a configuration file where you specify what you want webpack to do. We can use webpack in combination with the Babel transpiler because Babel has support for JSX. So to set this up, create another file called "webpack.config.js" in our directory. In the file, place the following code...

```
module.exports = {

    entry: __dirname + "/src/App.js",
    output: {
        path: __dirname + "/dist",
        filename: "App.js"
    },
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                loader: 'babel-loader',
                query: {
                    presets: ['es2015', 'react']
                }
            }
        ]
    }
};
```

Without getting too bogged down in the details, if we look at the "entry" property what this is basically saying is "look in the "src" folder for a file called App.js." The "loaders" setting under the "modules" property instructs webpack to run Babel on it (convert React JSX and our ES6 to native ES5 JavaScript) and put the output in a file called "App.js" in the "dist" directory which is specified in the "output" property. There are a number of different loaders that you can use with webpack to preprocess files apply different changes and transformations to them. Also note that the webpack utility will create any needed files for you if they do not already exist. The "App.js" file, now being converted from ES6 to ES5, can now run in all browsers and it is the file we will reference in our HTML.

Now that we have gotten our tooling all set up, we will take a look at some of the elements of ECMAScript 6 that will be important for us when we start writing our React applications.

## ECMAScript 6 Introduction

As we have discussed earlier ECMAScript 6, or ES6 for short, also known as ECMAScript 2015, is for lack of better terminology "the next version of JavaScript." ECMAScript 6 involves a significant number of functional and syntactical refinements for the JavaScript language since the previous version ECMAScript 5. React, being on the front end of incorporating new technologies when they emerge, can be utilized with ECMAScript 6. It is outside the scope of this discussion to look at *all* of the new features of ES6 up from ES5, but it is still worthwhile to cover a few of the parts of ES6 that are important to writing code for React applications. So let's take a brief look at some of these new pieces that will be important to writing React applications.

### *Class Syntax*

One of the more visible changes that ECMAScript 6 implements differently than ECMAScript 6 is a more syntactical conceptualization of "classes" in JavaScript. While JavaScript "classes" (air quotes) are not and never have been classes in the strictest technical sense as they are in the traditional object-oriented languages Java, C++, and C#, many JavaScript applications can still be written in a way that incorporates a very class-like implementation using functions, constructors and prototypal inheritance. For example, you will often see something the following in an ES5 JavaScript application:

```
function Animal() {
```

```
    this.color = 'brown';
    this.size = 'medium';
    this.age = 1;
}
Animal.prototype.sound = function () {
    console.log('Grunt');
};
Animal.prototype.move = function () {
    console.log('Running');
};

var beast = new Animal();
```

Inheritance can even be implemented in this approach...

```
function Dog() {
    Animal.call(this);
}
Dog.prototype = new Animal();
Dog.prototype.constructor = Dog

var fido = new Dog();
fido.color // brown;
```

By calling the Animal function and passing in the Dog function as the context, and new instance of a Dog will get all of the properties and methods found on Animal. We then do a little bit of additional cleanup with some of the prototype to make sure that the pointers are pointing at the right place.

This is all fine, but it seems a little bit convoluted. Fortunately for us, ES6 implements these same objects as follows using the "class" key word and the constructor function. What is shown below is exactly the same as the above.

```
class Animal {

    constructor(){
        this.color = 'brown';
        this.size = 'medium';
        this.age = 1;
    }

    sound() {
        console.log('Grunt');
    }
    move() {
        console.log('Running');
    }
}
```

We can implement inheritance using the "extends" keyword. We override some of the default properties that we found on Animal.

```
class Dog extends Animal {
    constructor(color, size, age) {
        this.color = color;
        this.size = size;
        this.age = age;
    }

    sound() {
        console.log('Woof');
    }
}

var d = new Dog('Black', 'medium', 5);
console.log(d.size); // medium
d.sound(); // Woof
```

```
d.move(); // Running
```

We can even go further than this and extend the Dog class. The implementation below makes use of the "super" keyword to call the constructor of the parent object...

```
class Pomeranian extends Dog{
    constructor(color, size, age) {
        super(color, size, age)
    }

    sound() {
        console.log('Yap');
    }
}

var p = new Pomeranian('Brown', 'small', 5);
p.sound(); // Yap
p.move(); // Running
```

When we call super(), our child class will automatically get all of the properties and methods of the parent class. We can then, of course, override these properties and methods in the child class to implement our inheritance.

## *Module Loading*

Another component of ES6 that you will find prevalent is module loading. These are basically the same conceptual implementation of modules that you will find in Node.js. Modules in ES6 solve the problem of scope and dependency management. In the past with ES5 a variable or function just declared in the open -- whether in a script block in an HTML file or in an external .js file -- like so....

```
function sayHi(name) {
    return "Hi, " + name;
}
```

would be considered to be bad practice because you were polluting the global object (which is often the window object in the context of the web). As the size and complexity of your application grew the maintainability of it would become a lot more difficult because you would have to worry about potentially overwriting other variables and functions later on in your application if you happened to name them the same or accidentally assign them a value that you did not intend to. Because of this, you often saw applications "namespace" the various methods and variables of an application to objects like so...

```
var app = app || {};
app.greeting = {
    sayHi: function (name) {
        return "Hi, " + name;
    }
};
```

to avoid polluting the global object.

With ES6 modules, however, all of this becomes unnecessary. You can declare functions and variables freely and the scope is always local to that file unless explicitly specified as a global object. This makes managing scope and maintaining your application a lot easier.

If you have ever done any development with Node.js, you should be somewhat familiar with modules as it follows a particular form and structure known as the CommonJS module loading syntax. Basically the idea is that you can import various functions and methods from other JavaScript files into your own JavaScript file by using the "import" keyword. You reference the path to the file you are loading as a module from the location of the file you are importing your module(s) into. So let's say in your project you had a file called "greeting.js" that lived in a folder named "lib" with the following code in it

```
export function sayHi(name) {
```

```
    return "Hi, " + name;
}
export function sayHello(name) {
    return "Hello, " + name;
}
export function sayHey(name) {
    return "Hey, " + name;
}
```

The "exports" keyword means that these functions are exported and can be imported into other JavaScript files. For example, in the root of your project (one level up from the "lib" directory) let's say you had a file called main.js. In that file you could do the following...

```
import * as greet from 'lib/greeting';

console.log(greet.sayHi('Joe')); // "Hi, Joe"
console.log(greet.sayHello('Sally')); // "Hello, Sally"
```

Here we are importing all of the exported functions in the greeting.js file into a variable called "greet" using the * wildcard locally to our main.js file. If there were any other exported functions or properties in our greeting.js file, this would be available in main.js as well.

There are other syntaxes that can be utilized in module importing. For example, if we only wanted the sayHi and sayHello methods imported from our greeting.js file (and we did not want or need the sayHey method for some reason) we could do the following in main.js...

```
import { sayHi, sayHello } from 'lib/greeting';

console.log(sayHi('Joe')); // "Hi, Joe"
console.log(sayHello('Sally')); // "Hello, Sally"
```

There are other variations on the importing syntax and implementation as well that we might see as we go along. The main point is that this gives you a lot of flexibility in how you structure your application. Doing all of this is not too dissimilar from the importing that you would so in Java...

```
import com.domain.package;
```

or C#

```
using System.Net;
```

The difference being that in JavaScript you import from the path to the file and in Java and C# the packages are compiled into memory and imported from there. But in all cases the approach serves the same purpose.

You could also use the require syntax (again familiar from Node.js) to import a module...

```
var greet = require('lib/greeting.js')

console.log(greet.sayHi('Joe')); // "Hi, Joe"
console.log(greet.sayHello('Sally')); // "Hello, Sally"
```

The choice is yours depending on which you prefer.

### *Arrow Functions*

Arrow functions, also sometimes called "fat arrow" functions, are a new ES6 syntax for writing anonymous function expressions. A function expression in JavaScript refers to the notation where a developer sets a variable equal to a function like so...

```
var sum = function (a, b) { return a + b; }
```

Note that a function expression is different from a function declaration which would look like the following...

```
function sum(a, b) { return a + b }
```

So arrow functions are primarily centered around function expressions. They have 2 main purposes. For one, they provide a more concise syntax which is a nice thing to have and can make code more readable. If were to take our function expression above and rewrite it as an arrow function, it would look like the following...

```
var sum = (a, b) => a + b;
```

As can be seen here, this syntax looks a bit different from our original function expression. We have the arrow pointer pointing a statement that has omitted the { } from encapsulating the function body and we have omitted the "return" statement. In the above arrow function these things are implied. If we wanted to we could include these things and the meaning of the function would be the same.

```
var sum = (a, b) => { return a + b };
```

There are few rules around the syntax of arrow functions to watch out for. If there is only one argument in an arrow function the parenthesis do not need to be included. So an arrow function that simply returns "a" could be written like so...

```
var getVal = a => a;
```

However if the arrow function takes no arguments, parentheses must be used...

```
var getVal = () => "Hello world!";
```

Another interesting case of arrow function syntax is returning an empty object. The following results in an error...

```
var getVal = () => { };
```

The compiler sees this as having nothing in the function body and does not know what to return. As a result when returning an empty object you have to do the following...

```
var getVal = () => ({ });
```

So as we can see, arrow functions give us a much more terse way of writing code, allowing for a more readable concise syntax. This becomes useful in cases when using functions that have callbacks involved as arguments. Many functions that that perform asynchronous operations with data have this format. The following asynchronous promise style function...

```
getData().then(function (data) {
    return data;
});
```

Could be rewritten as the following using arrow functions...

```
getData().then(data => data);
```

It is pretty apparent that the second example is a lot more compact. As the number of functions like these get chained together, it can make the code a lot easier to read once you get used to the syntax.

Another and perhaps a more important aspect arrow functions is that they handle the scope of the "this" pointer within the function in a different manner. In other words, in arrow functions "this" points to the scope of the parent environment that the arrow function is contained within. This is a very important distinction. Arrow functions do not create their own "this" as normal functions do. A side-effect of this is that you cannot instantiate an arrow function with the "new" keyword as you can with normal functions because arrow functions have no internal prototype property.

Why were things done this way in ES6 and why is this useful? This can be very useful because you now no longer have to use apply, call, or bind to bind to the parent scope. One of the more common uses of the bind method within JavaScript applications is to pass the parent scope downward as you go multiple levels deep within methods within in object literal. For example, let's say we had the following very general "actionClass" object literal. A lot of JavaScript applications use this pattern as an implementation...

```
<div class="button">Click me!</div>

<script type="javascript">
    var actionClass = {

        init: function(){
            this.setEvents();
        },
        setEvents: function(){
            // we will set events here...

        },
        doSomething: function(){
            console.log('We are doing something');
        },

        doSomethingElse: function() {
            console.log('We are doing something else');
        }

    };

    actionClass.init();

</script>
```

As we can see we are calling the actionClass.init() method which will set up some event handlers. So let's add an event (using jQuery) where when we click an element we call another one of the methods within our actionClass object.

```
<div class="button">Click me!</div>

<script type="javascript">
    var actionClass = {

        init: function(){
            this.setEvents();
        },
        setEvents: function(){

            jQuery('.button').on('click', function(){
                this.doSomething();
            });

        },
        doSomething: function(){
            console.log('We are doing something');
        },

        doSomethingElse: function() {
            console.log('We are doing something else');
        }

    };

    actionClass.init();

</script>
```

This code does not work. When we try to click the div we get an error in our console...

```
Uncaught TypeError: this.doSomething is not a function
```

Why is that? Well it is because the callback function for the click event has its own scope. "this" is now pointing at the function it is being called from. One way that developers solve this is by setting a variable to the parent scope. The following code works...

```
<div class="button">Click me!</div>

<script type="javascript">
    var actionClass = {

        init: function(){
            this.setEvents();
        },
        setEvents: function(){

            var self = this;

            jQuery('.button').on('click', function(){
                self.doSomething();
            });

        },
        doSomething: function(){
            console.log('We are doing something');
        },

        doSomethingElse: function() {
            console.log('We are doing something else');
        }

    };

    actionClass.init();

</script>
```

This is certainly a viable option and works fine for our simple example. However, as we go deeper and get more complex into multiple levels of nested functions with events or getting data asynchronously via AJAX requests or any number of other possibilities the process of constantly assigning variables to the scope that you want can get kind of messy. So what other approach can we take? This is where bind comes into the picture. We could also do something like the following...

```
<div class="button">Click me!</div>

<script type="javascript">
    var actionClass = {

        init: function(){
            this.setEvents();
        },
        setEvents: function(){

            jQuery('.button').on('click', function(){
                this.doSomething();
            }.bind(this));

        },
        doSomething: function(){
            console.log('We are doing something');
        },

        doSomethingElse: function() {
```

```
            console.log('We are doing something else');
        }

    };

    actionClass.init();

</script>
```

Simply by attaching .bind(this) after our function closing bracket } we are changing the scope of the function. We longer need to say var self = this;

But with arrow functions we can accomplish the same thing without having to resort to using "bind." We can rewrite the above in the following manner (assuming we had ES6 compatibility in the browser or we transpiled the code)...

```
<div class="button">Click me!</div>

<script type="javascript">
    var actionClass = {

        init: function(){
            this.setEvents();
        },
        setEvents: function(){

            jQuery('.button').on('click', () => {
                this.doSomething();
            });

        },
        doSomething: function(){
            console.log('We are doing something');
        },

        doSomethingElse: function() {
            console.log('We are doing something else');
        }

    };

    actionClass.init();

</script>
```

Now because we have used an arrow function, the "this" pointer is set to the parent scope because of how arrow functions bind the lexical "this" to the parent. It may take a bit of getting used to but overall the differing scope of the "this" pointer combined with the concise syntax can make code that makes use of arrow functions a lot more readable and maintainable. We will be making use of them to some degree in upcoming examples, so it was necessary to at least introduce what arrow functions are and what they do.

This is just a brief look at a couple of the newer features found in ES6. There are many other parts of ES6 that will be incorporated into our React applications that we will discuss as we go along.

## "Hello World" in React

So now our tooling set up and we have gotten ourselves at least a bit more familiar with ES6, let's take a look at utilizing React's JSX syntax. In the same directory as our package.json and webpack.config.js file create a file called "index.html" and add it to the folder. In this file add the following code...

```
<!DOCTYPE html>
<html>
<head>
    <title>React</title>
    <meta charset="UTF-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.0/react-dom.js"></script>
    <script type="text/javascript" src="dist/App.js"></script>
</body>
</html>
```

In this same folder, create another folder called "src" (short for "source") and create a file called "App.js" within this folder. Remember from our webpack.config.js configuration that we are going to use this file as the source to build the "dist/App.js" file that we are referencing in the script above. So after you create this "App.js" file inside of it let's add the following JSX code...

```
import {react} from 'react';

class Hello extends React.Component {
    render() {
        return(<h1>Hello World</h1>)
    }
}

React.render(<Hello />, document.body);
```

Now we can open a command prompt window and run webpack to build our JavaScript file from JSX. To run webpack you just need to type the following into a command window...

```
$ webpack
```

If all goes well we should get a "js" directory with an "App.js" file inside of it created.

Note: If this does not work for you or you get an error you might need to globally install webpack and babel from npm. The way that you do this is open a command window/terminal anywhere. To install webpack globally type...

```
$ npm install -g webpack
```

To install Babel globally type...

```
$ npm install -g babel
```

The -g flag specifies that you are installing the tool globally on your system and that you can use those utilities from anywhere.

Once we have everything sorted out we can open another command prompt in the directory with our package.json, webpack.config.js file and index.html file and run the following...

```
$ http-server
```

If you go to http://localhost:8080 you should see the message "Hello World" displayed.

## Properties & Displaying Data With this.props

Let's try to create a React application that is a little bit more complex... one that shows how we can render data within our components. We will be displaying a list of notes in the form of a notepad application. To accomplish this we will pass a set of data into our React component and iterate through it. In our "src" folder change the code in "App.js" file to the following code...

```
import {react} from 'react';

var notepad = [
    {
```

```
        id: 1,
        title: "Eat Breakfast",
        content: "Toast and scrambled eggs would be nice."
    },
    {
        id: 2,
        title: "Walk Dog",
        content: "Dress for wet weather."
    },
    {
        id: 3,
        title: "Learn React",
        content: "I am making great progress!"
    }
];

class NotesList extends React.Component {

    render() {

        var notes = this.props.notes.map(function (note) {
            return (
                <li>
                    <h2>{note.title}</h2>
                    <p>{note.content}</p>
                </li>
            );
        });

        return (<ul>{notes}</ul>);
    }
};
```

So here we can see our "NotesList" component. The data that gets passed in to any React component is available on the "props" object. The format goes this.props.attribute where "attribute" is the name of the attribute. So in our example it is called "notes". So when we call our render function, it will look like the following...

```
ReactDOM.render(<NotesList notes={notepad} />, document.getElementById('content'));
```

The value "notepad" can be any JavaScript type... a string, object literal, array, or whatever else. We access it inside our component with this.props. In this case it is an array, but it could be anything. In a production grade application it would probably be best practice to do some checking/error handling against this, but for this simple example we will omit it for now.

Get used to seeing this.props. Along with this.state these are among 2 of the most commonly seen code segments that you will see in React. Since React is all about rendering the data in components as a UI, this.props is how we are able to access data coming in from outside our component when it is passed in. We saw this above with our "NotesList" component receiving the array stored in the "notepad" variable passed into our component. This data was available to us by using this.props.notes which was the slot where we passed the array into our component. So this.props is very important for how we pass data around between components.

In general, this.state is the container object that stores the current state of a React component. Whenever anything in this.state changes a "re-render" is triggered by the React library to update the UI.

We are calling the map function on the notes property because we are expecting this value passed into the notes prop to be an array. We will iterate through this array and render out a title and some content. We iterate through the array and display the various values set in each object. For the second one, because there is no title, we just use that as the title. If we run this in our local web server, we can see the results.

## Configuring webpack to Watch Our Files for Changes

It is somewhat inefficient to have to go to the command window and run $ webpack every single time we make a change to our files. Fortunately there is a way for us to configure webpack to "watch" for changes in our files and every time we save a file webpack will automatically recompile the JSX to JavaScript for us. All we need to do for this is add the following to our webpack.config.js file...

```
module.exports = {

    entry: __dirname + "/src/App.js",
    output: {
        path: __dirname + "/dist",
        filename: "App.js"
    },
    watch: true,
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                loader: 'babel-loader',
                query:{
                    presets: ['es2015', 'react']
                }
            }
        ]
    }
};
```

So we have in our "watch" property we have basically said, "Watch all the .js files in the project excluding the "node_modules" directory. When a change occurs, run the "babel" on our files." You can test this out by changing the App.js file (try adding a /* test comment */ or something) and see webpack run the task in the console. If it says that the task was/is completed successfully without errors then you should be able to open the App.js file in the "dist" folder and see your changes reflected.

You can quit the current running "watch" task at any time by hitting Ctrl + C in your terminal.

## Configuring webpack to Minify JavaScript

There is one final piece that we can add as an additional step of polish on our application. That has to do with minifying our JavaScript. This step is not required and is basically just a brief demonstration of how you can configure our webpack to run different tasks. If you have worked with JavaScript in the past you will likely know that we will want to minify our JavaScript for performance reasons. A smaller file to download over HTTP will result in a faster load time.

To minify our scripts on build we have to add the webpack UglifyJS plugin. This is a module that "uglifies" (i.e. minifies) JavaScript to reduce the file size. So we can change our "webpack.config.js" file to the following...

```
var webpack = require('webpack');

module.exports = {
    entry: __dirname + "/src/App.js",
    output: {
        path: __dirname + "/dist",
        filename: "App.js"
    },
    watch: true,
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                loader: 'babel-loader',
                query:{
```

```
                    presets: ['es2015', 'react']
                }
            }
        ]
    },
    plugins:[
        new webpack.optimize.UglifyJsPlugin()
    ]
};
```

Now, when we change the App.js file and web pack runs and builds, the App.js file will be minified. If you want to do the minify step on build but you also want to debug using your dev tools you can generate a source map by adding the following property to your webpack configuration file...

```
var webpack = require('webpack');

module.exports = {
    entry: __dirname + "/src/App.js",
    output: {
        path: __dirname + "/dist",
        filename: "App.js"
    },
    watch: true,
    module: {
        loaders: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                loader: 'babel',
                query:{
                    presets: ['es2015', 'react']
                }
            }
        ]
    },
    devtool: 'source-map',
    plugins:[
        new webpack.optimize.UglifyJsPlugin({ sourceMap: true })
    ]
};
```

And that is all there is to it!

## Using webpack as a Web Server

We have been using another node module, http-server, to run our React application. As it turns out, however, we can actually utilize a webpack utility to do the same thing for us. To install this, we need to change out package.json file...

```
{
    "name": "React",
    "version": "0.0.1",
    "description": "A React application...",
    "license": "MIT",
    "repository": {
        "type": "",
        "url": ""
    },
    "homepage": "http://www.9bitstudios.com",
    "dependencies": {
        "react": "15.4.0"
    },
    "devDependencies": {
        "webpack": "2.2.1",
        "webpack-dev-server": "2.4.1",
        "babel-core": "6.13.0",
```

```
        "babel-loader": "6.2.4",
        "babel-preset-es2015": "6.13.0",
        "babel-preset-react": "6.11.1"
    }
}
```

See how we have added the "webpack-dev-server" module? Now that we have added it to our package.json file we can open a terminal run

```
$ npm install
```

to install the module. After the "webpack-dev-server" module finishes installing we now need to configure our webpack.config.js file. So change the webpack.config.js file to the following...

```
var webpack = require('webpack');

module.exports = {
    entry: __dirname + "/src/App.js",
    output: {
        path: __dirname + "/dist",
        filename: "App.js"
    },
    watch: true,
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                loader: 'babel-loader',
                query:{
                    presets: ['es2015', 'react']
                }
            }
        ]
    },
    devServer: {
        stats: {
            colors: true
        },
        inline: true
    },
    devtool: 'source-map',
    plugins:[
        new webpack.optimize.UglifyJsPlugin({ sourceMap: true })
    ]
};
```

Notice that we have added a "devServer" property in our file with a couple of options set. There are a number of other options you can configure for the "webpack-dev-server" module. To learn more about what these are, please refer to the documentation here.

Now that we have added this to our "webpack.config.js" file we can run it from out command line/terminal by typing in the following...

```
$ node_modules/.bin/webpack-dev-server
```

This will start a web server and serve up the root directory of our project at http://localhost:8080 just as the "http-server" module had done.

Alternatively, you can add the following start script to your package.json file...

```
{
    "name": "React",
    "version": "0.0.1",
```

```
    "description": "A React application...",
    "license": "MIT",
    "repository": {
        "type": "",
        "url": ""
    },
    "homepage": "http://www.9bitstudios.com",
    "scripts": {
        "start": "webpack-dev-server --progress"
    },
    "dependencies": {
        "react": "15.4.0"
    },
    "devDependencies": {
        "webpack": "2.2.1",
        "webpack-dev-server": "2.4.1",
        "babel-core": "6.13.0",
        "babel-loader": "6.2.4",
        "babel-preset-es2015": "6.13.0",
        "babel-preset-react": "6.11.1"
    }
}
```

We have added a "scripts" property and specified a "start" command. The --progress flag will show a percentage progress of webpack building your file(s). Now that you have added this, all you need to do to run the development server is type the following...

```
$ npm start
```

and the server will be started up. You can press Ctrl + C to quit the server at any time.

These are just some examples of how you can utilize the various components that webpack offers to aid your development workflow. There are many other ways that you can configure webpack to do different things for you. How you want to use these various different tools is really up to you.

## Summary

So this concludes our very brief and succinct introduction to React. We have only just scratched the surface with some basic rendering of data coming from our React component and we also looked at how to set up some of the React tools to work with JSX and compile it to plain JavaScript using webpack. In upcoming discussions, we will explore the more complex and more dynamic features that React can give us and we will see where the value of this library really emerges. But our journey has started out with a few small steps that we can build upon from here.

# Chapter 2: To Do: Create Your First React Application

One of the more ubiquitous tutorial applications that developers build when learning a new language or framework is the TO DO (To do) app. In terms of prevalence, it is probably second only to the "Hello, World" app which has been around since the dawn of programming. But the "To Do" application is popular because it covers a lot of the basics of how a language or framework handles a number of things. There must be a way to add a new "to do" so handling user input is involved. The same is true for displaying data, editing and saving data, and changing display state when data updates. All of these things are going to be important implementations in any application.

In what follows we will create the iconic "To Do" application in React and discuss some of the inner workings as we go along. We will take a deeper look into React components and how we can add some dynamic interactivity to our React applications in response to user actions -- things like clicks and entering text in a text box -- and updating our views accordingly.

## Setting Up

To start we need to set up our package.json and webpack.config.js files so that we can set up our tools to do JSXcompilation to native JavaScript. Create a new directory and then create a package.json file and add the following...

```
{
    "name": "React",
    "version": "0.0.1",
    "description": "A React application...",
    "license": "MIT",
    "repository": {
        "type": "",
        "url": ""
    },
    "homepage": "http://www.9bitstudios.com",
    "scripts": {
        "start": "webpack-dev-server --progress"
    },
    "dependencies": {
        "react": "15.4.0"
    },
    "devDependencies": {
        "webpack": "2.2.1",
        "webpack-dev-server": "2.4.1",
        "babel-core": "6.13.0",
        "babel-loader": "6.2.4",
        "babel-preset-es2015": "6.13.0",
        "babel-preset-react": "6.11.1"
    }
}
```

Next, in this same directory, create a webpack.config.js file and add the following...

```
var webpack = require('webpack');

module.exports = {
    entry: __dirname + "/src/App.js",
    output: {
        path: __dirname + "/dist",
        filename: "App.js"
    },
    watch: true,
    module: {
        rules: [
            {
                test: /\.jsx?$/,
                exclude: /node_modules/,
                loader: 'babel-loader',
                query:{
```

```
                    presets: ['es2015', 'react']
                }
            }
        ]
    },
    devServer: {
        stats: {
            colors: true
        },
        inline: true
    },
    devtool: 'source-map',
    plugins:[
        new webpack.optimize.UglifyJsPlugin({ sourceMap: true })
    ]
};
```

After this, let's open a command prompt and run the following...

```
$ npm install
```

to install all of our modules that we will need for compiling ES6 and JSX. And that is all there is to it. Now that we have our tools installed and configured, we can dive into writing our application.

## Writing a "To Do" Application

So let's get started creating our "To Do" app. We will start out by creating our index.html file...

```
<!DOCTYPE html>
<html>
<head>
    <title>React</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
    <div id="content"></div>

    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.4.0/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.4.0/react-dom.js"></script>
    <script type="text/javascript" src="dist/App.js"></script>

</body>
</html>
```

As we can see, we are going to include App.js, which will be a file that we generate in our task runner. Note too we are creating a  with the id of "content."

Next, create a folder called "src" and create a file called App.js in this folder. For this tutorial we are going to be breaking our components up into different modules so in our App.js file all we need to add for now is the following.

```
import {react} from 'react';
```

In the same directory as your App.js file, next create a file called "Todo.js". In this file, we are going to need to create a React component/class that will represent each individual "to do." Add the following code to this file...

```
import {react} from 'react';

class Todo extends React.Component {

    render() {
        return(<div>I am a to do</div>)
    }
};
```

Pretty straightforward. This should look familiar as it is similar to our other earlier examples.

Now what we're going to do is import this module into our App.js file. How do we do this? In our App.js file we just need to add the following...

```
import {react} from 'react';
import {Todo} from './Todo';
```

This will tell our code to import the "Todo" module from the "Todo" file relative to where "App.js" is located. With module loading in ES6 you can omit the ".js" from the file when you are importing modules.

So we are definitely well on our way but if we were to run this code as it is right now we would get an error. Why is this? Well, because we are **importing** a module from our "Todo.js" file we need to specify what this file **exports**. To do this we need to add the "export" keyword in front of what we want to make available from the "Todo.js" file. Thus, we need to change our code in our "Todo.js" file to the following...

```
import {react} from 'react';

export class Todo extends React.Component {

    render() {
        return(<div>I am a to do</div>)
    }
};
```

Notice how we have added the "export" keyword in front of our class. Now there is something that can be imported from the "Todo.js" file. We will get into more complex combinations of what various files can export and import later on, but for now this simple example shows the basic setup of how it is done.

Now in our App.js file add the "render" method so we can confirm that everything is working...

```
import {react} from 'react';
import {Todo} from './Todo';

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

Let's build our App.js file by running "webpack" by opening up a terminal window typing the following...

```
$ webpack
```

Doing the above will build the App.js file and place it in our "dist" folder. Now if we open up another terminal window and run our server with...

```
$ npm start
```

and we go to http://localhost:8080 in our browser, if all has gone well we should be able to see the text "I am a to do."