GSAtrain

Supplemental Labs

# ReactJS Building Modern Web Applications

GSAtrain

## Table of Contents

1. Create-React-App
2. ES6
3. JavaScript Array map() Method
4. JavaScript Promises
5. Setting Up a React Project

# create-react-app

*Edit this page on Github*

Today, we're going to add a build process to store common build actions so we can easily develop and deploy our applications.

The React team noticed that there is a lot of configuration required (and the community helped bloat -- us included) to run a React app. Luckily, some smart folks in the React team/community got together and built/released an official generator app that makes it much easier to get up and running quickly.

# create-react-app

The create-react-app project is released through Facebook helps us get up and running quickly with a React app on our system with no custom configuring required on our part.

The package is released as a Node Package and can be installed using `npm`.

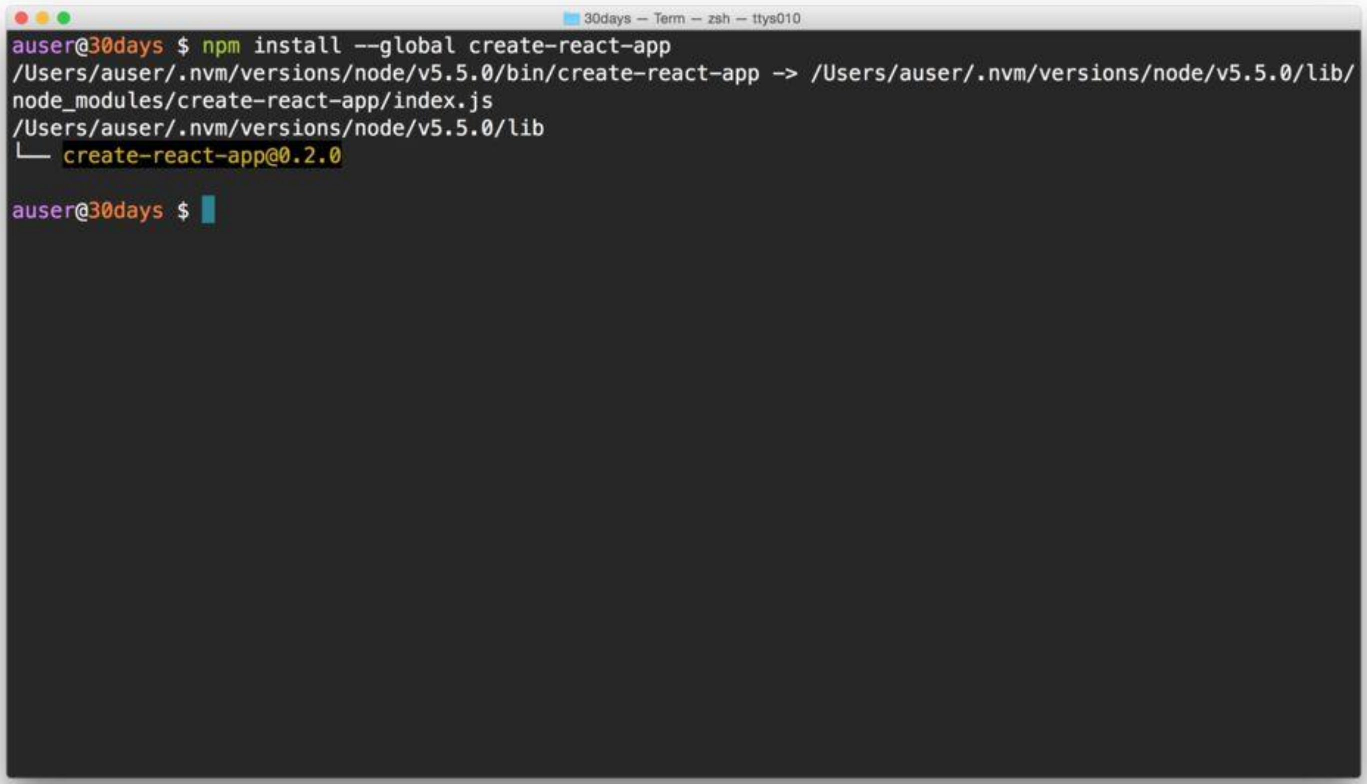
## A plug for `nvm` and `n`

The Node homepage has simple documentation on how to install node, if you don't already have it installed on your system.

We recommend using the nvm or the n version management tools. These tools make it incredibly easy to install/use multiple versions of node on your system at any point.

With `node` installed on our system, we can install the `create-react-app` package:

```
npm install --global create-react-app
```



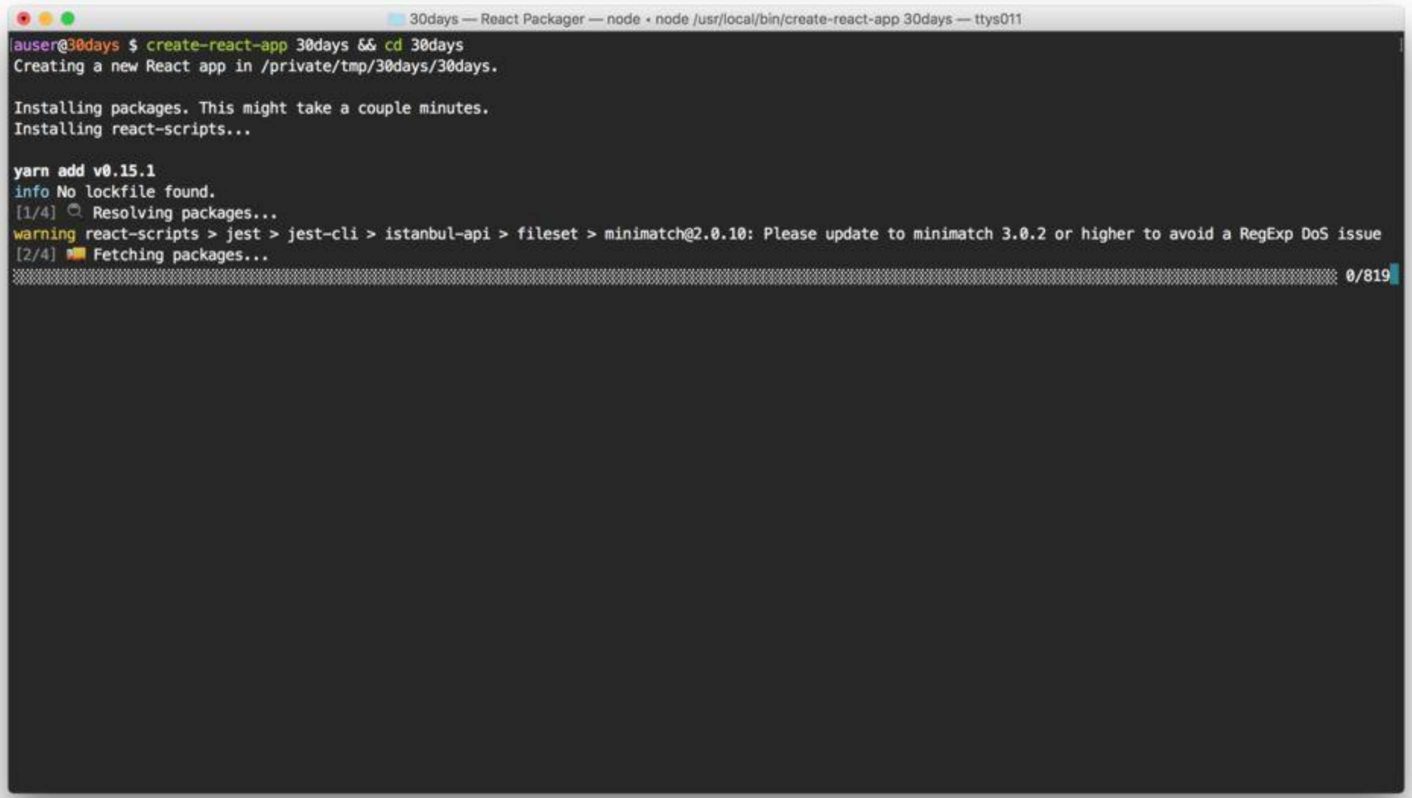
```
auser@30days $ npm install --global create-react-app
/Users/auser/.nvm/versions/node/v5.5.0/bin/create-react-app -> /Users/auser/.nvm/versions/node/v5.5.0/lib/
node_modules/create-react-app/index.js
/Users/auser/.nvm/versions/node/v5.5.0/lib
└── create-react-app@0.2.0

auser@30days $
```

With `create-react-app` installed globally, we'll be able to use the `create-react-app`command anywhere in our terminal.

Let's create a new app we'll call 30days using the `create-react-app` command we just installed. Open a Terminal window in a directory where you want to create your app.

In terminal, we can create a new React application using the command and adding a name to the app we want to create.

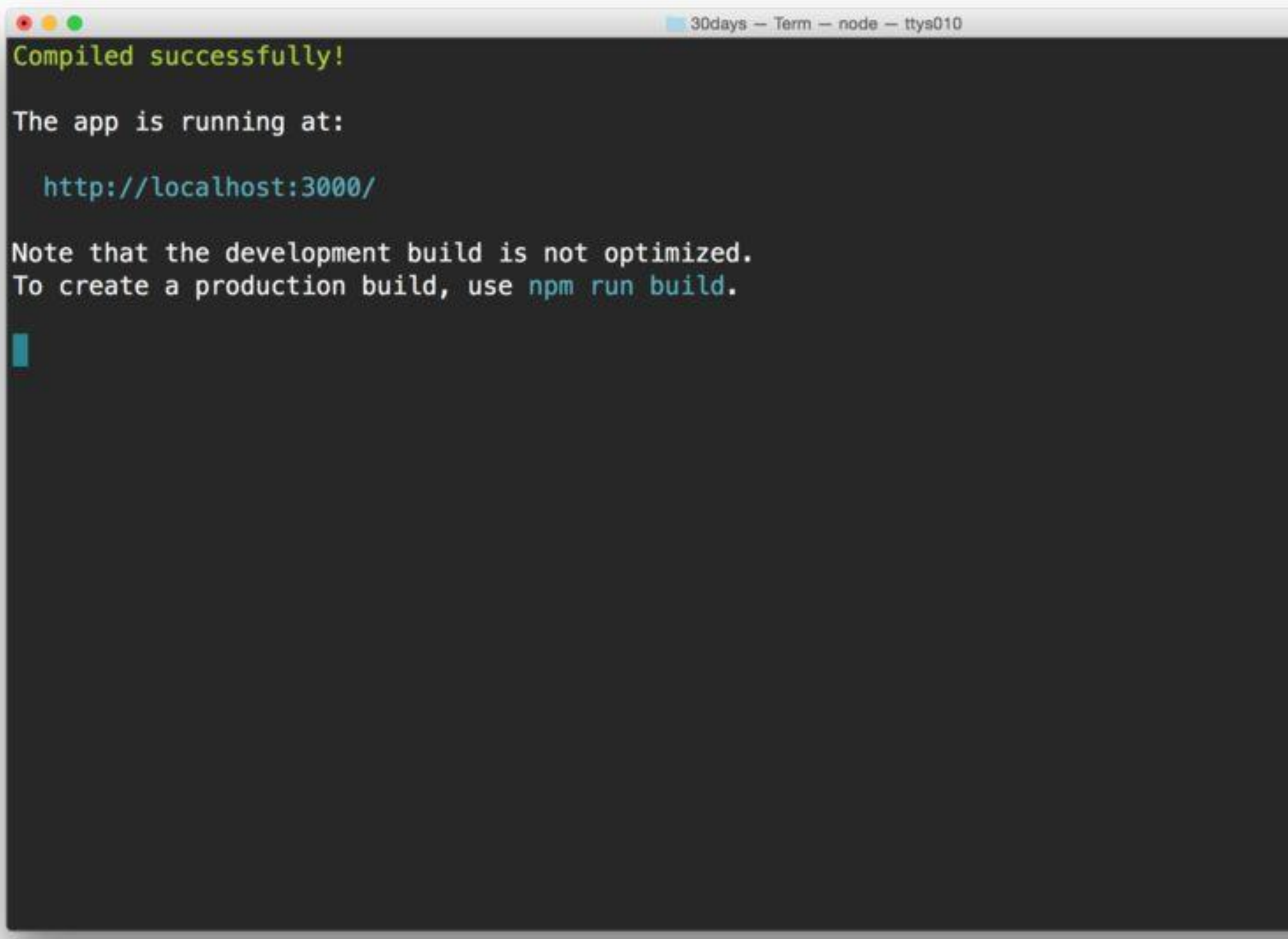
```
create-react-app 30days && cd 30days
```


30days — React Packager — node • node /usr/local/bin/create-react-app 30days — ttys011

```
auser@30days $ create-react-app 30days && cd 30days
Creating a new React app in /private/tmp/30days/30days.

Installing packages. This might take a couple minutes.
Installing react-scripts...

yarn add v0.15.1
info No lockfile found.
[1/4] 🔍 Resolving packages...
warning react-scripts > jest > jest-cli > istanbul-api > fileset > minimatch@2.0.10: Please update to minimatch 3.0.2 or higher to avoid a RegExp DoS issue
[2/4] 🚚 Fetching packages...
░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░░ 0/819
```

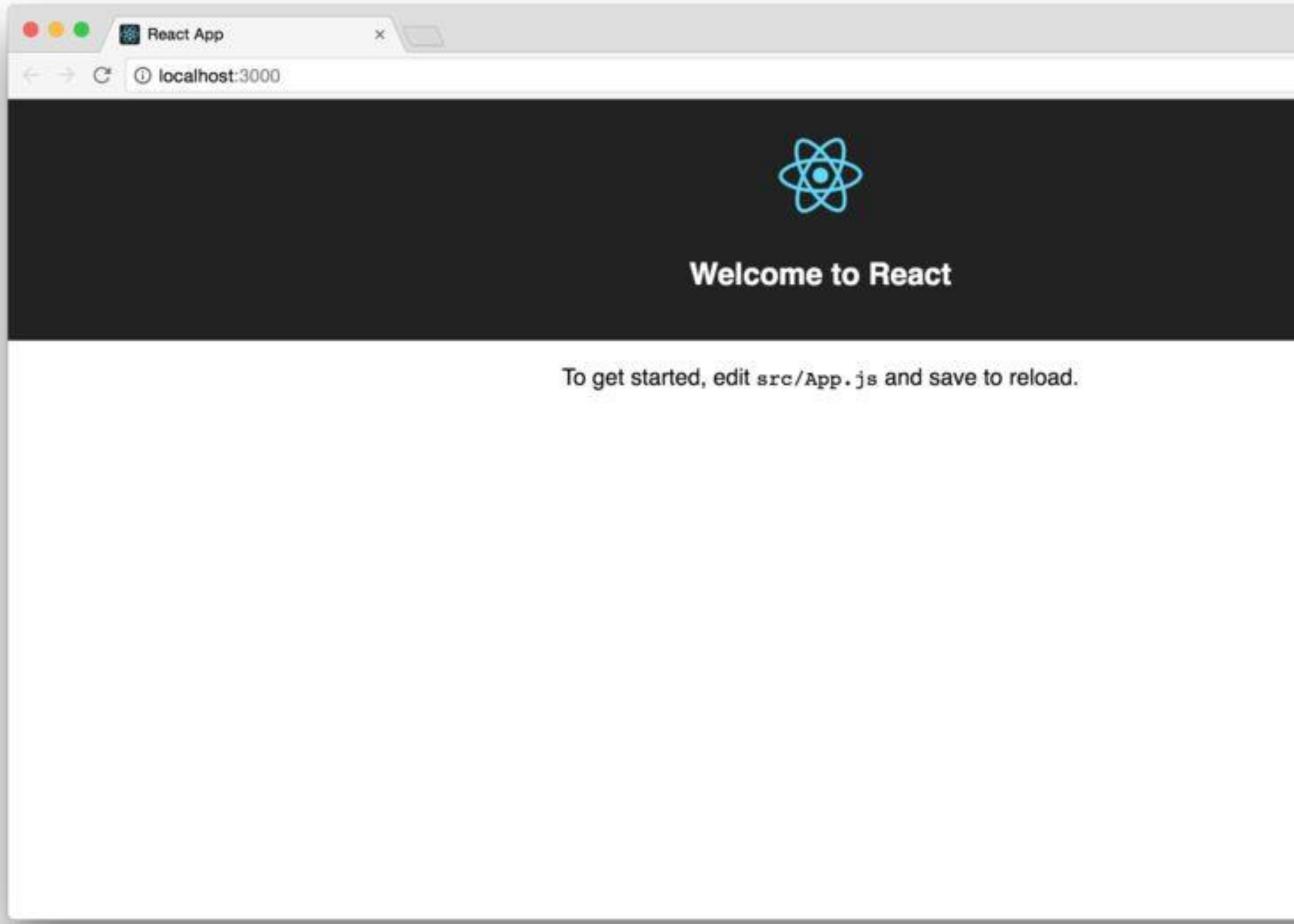
Let's start our app in the browser. The `create-react-app` package comes with a few built-in scripts it created for us (in the `package.json` file). We can *start* editing our app using the built-in webserver using the `npm start` command:

```
npm start
```



```
Compiled successfully!

The app is running at:

  http://localhost:3000/

Note that the development build is not optimized.
To create a production build, use npm run build.
```

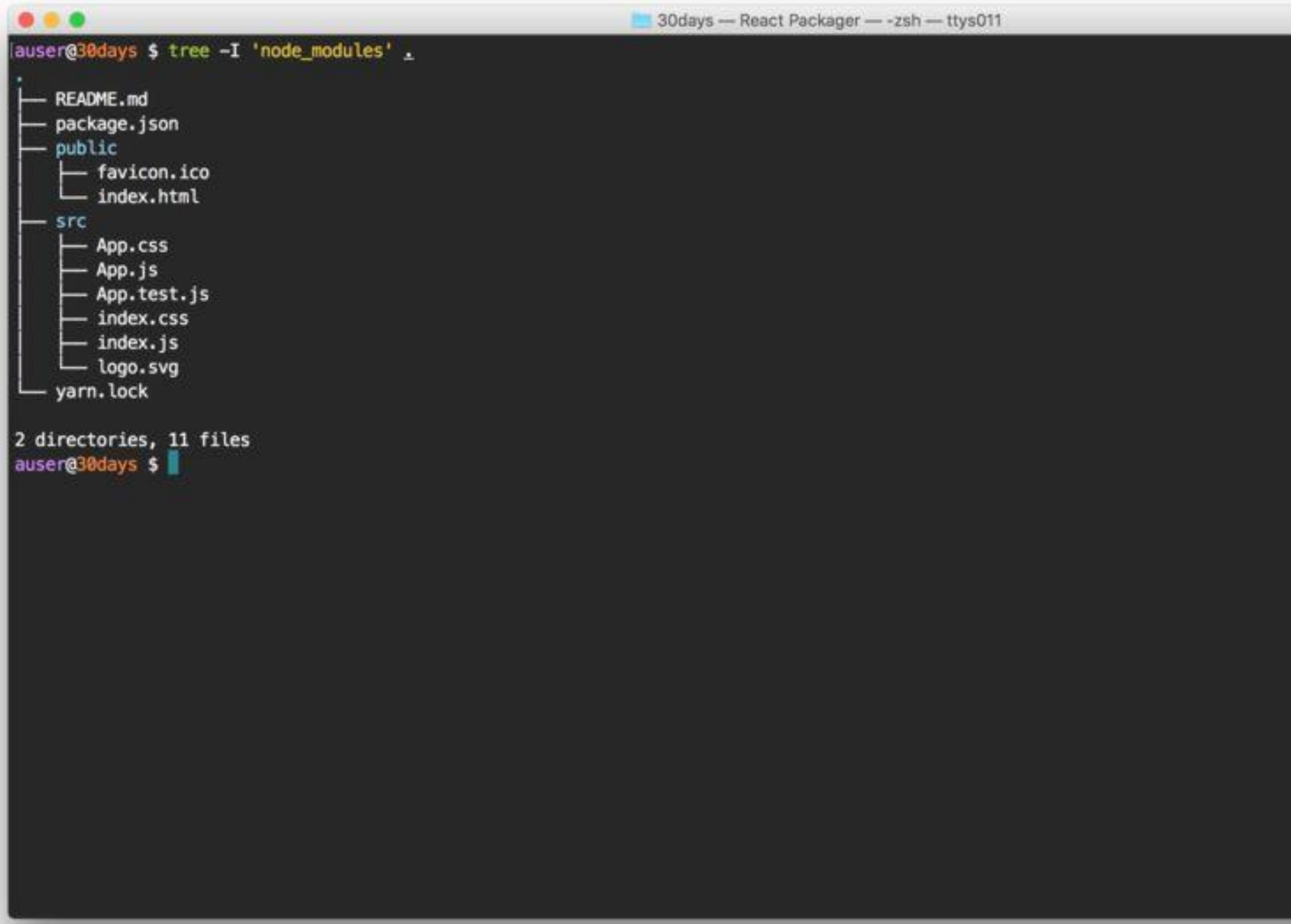
This command will open a window in Chrome to the default app it created for us running at the url: [http://localhost:3000/](http://localhost:3000/).



Welcome to React

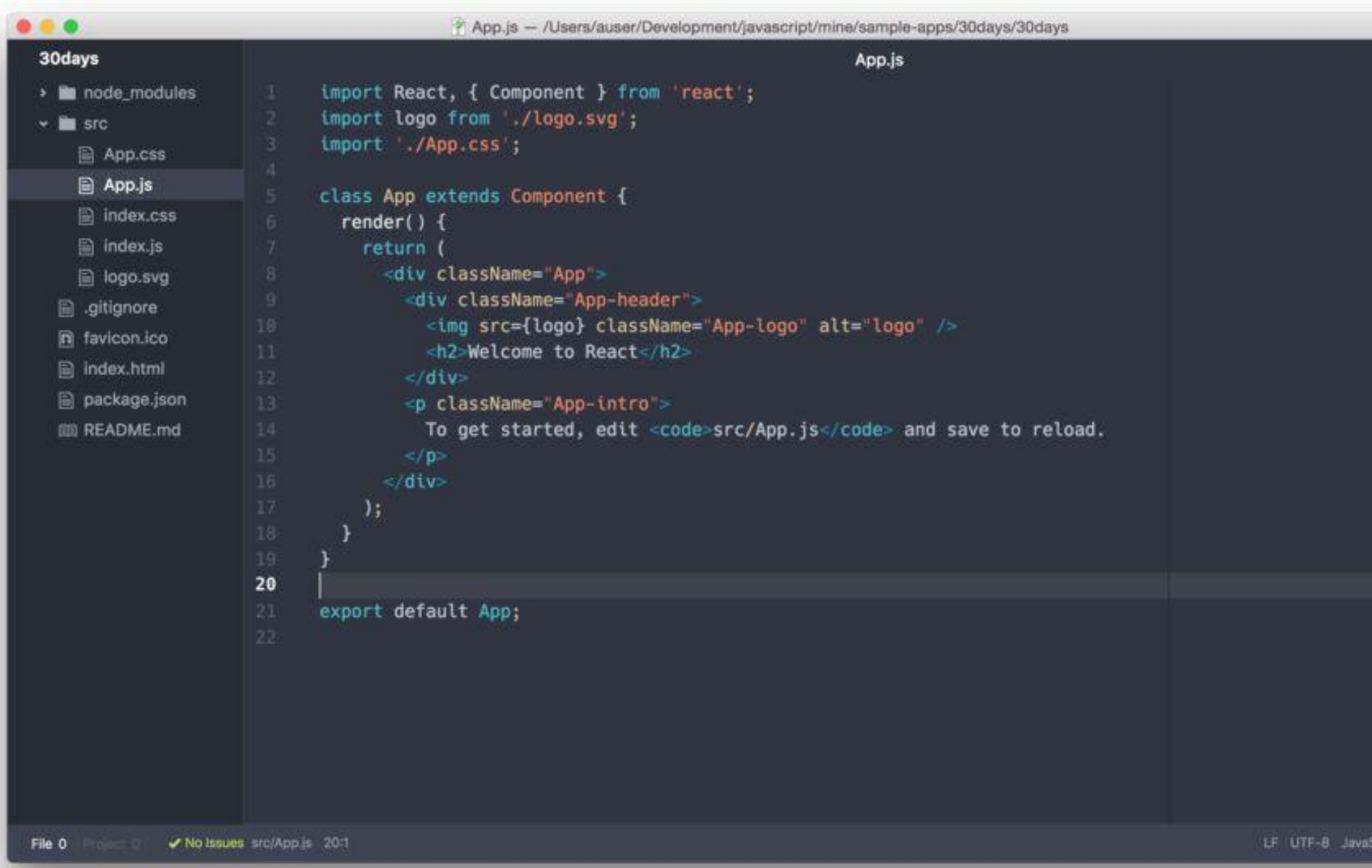
To get started, edit `src/App.js` and save to reload.

Let's edit the newly created app. Looking at the directory structure it created, we'll see we have a basic node app running with a `public/index.html` and a few files in the `src/`directory that comprise our running app.



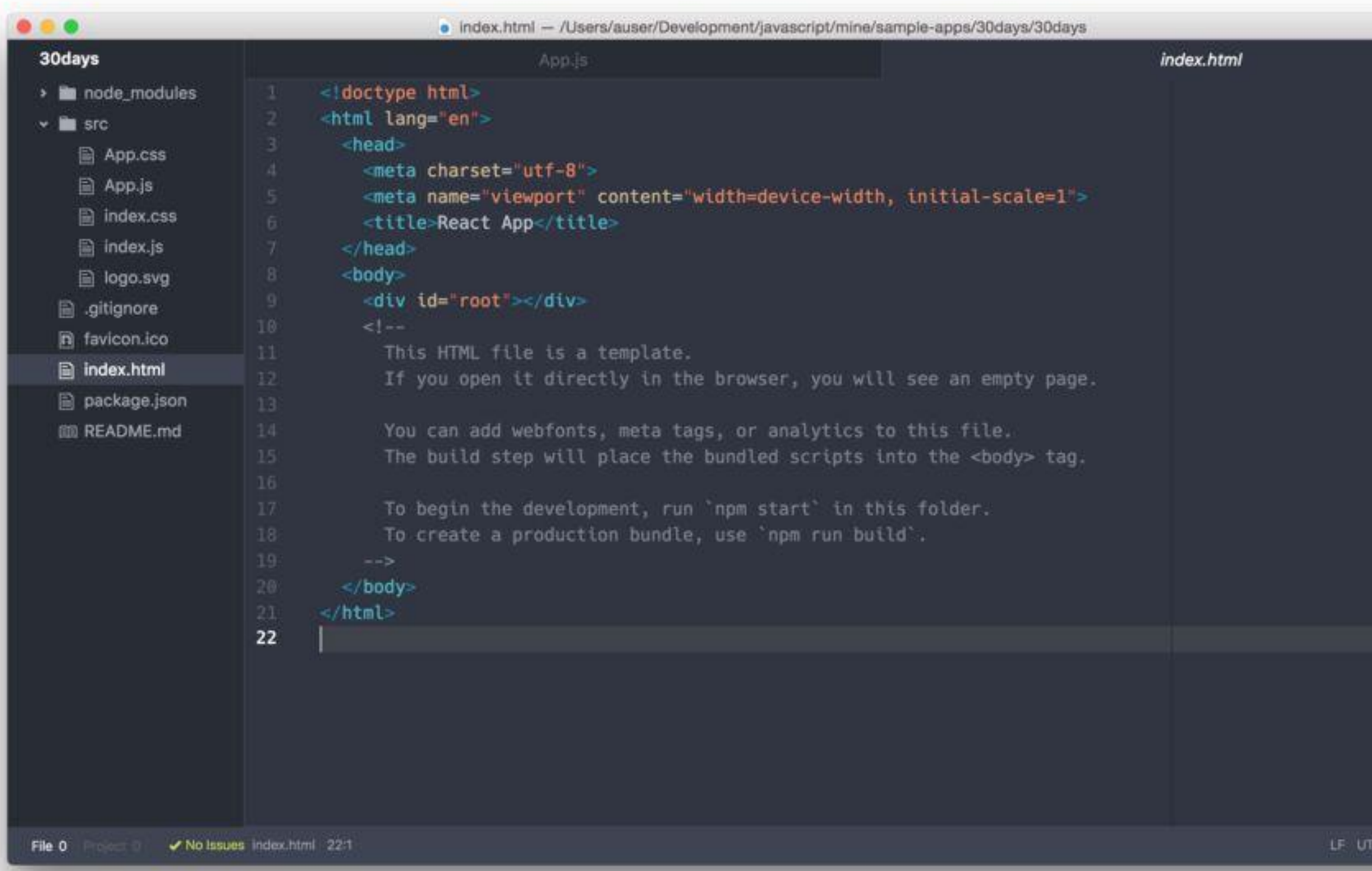
```
auser@30days $ tree -I 'node_modules' .
.
├── README.md
├── package.json
├── public
│   ├── favicon.ico
│   └── index.html
├── src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   └── logo.svg
└── yarn.lock

2 directories, 11 files
auser@30days $
```

Let's open up the `src/App.js` file and we'll see we have a very basic component that should all look familiar. It has a simple render function which returns the result we see in the Chrome window.



```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <div className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h2>Welcome to React</h2>
        </div>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    );
  }
}

export default App;
```

The `index.html` file has a single `<div />` node with the id of `#root`, where the app itself will be mounted for us automatically (this is handled in the `src/index.js` file). Anytime we want to add webfonts, style tags, etc. we can load them in the `index.html`file.


index.html — /Users/auser/Development/javascript/mine/sample-apps/30days/30days

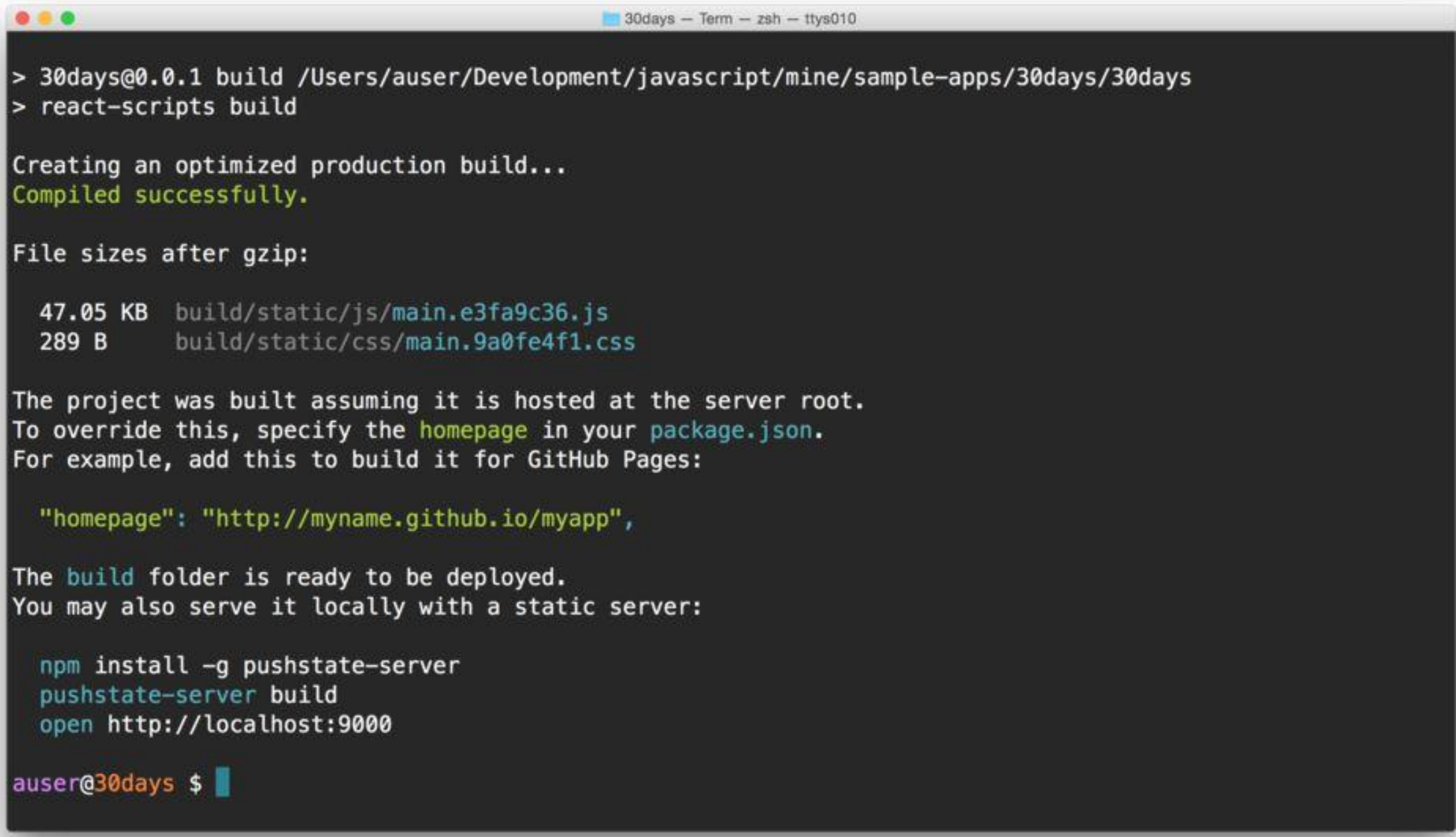
```html
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>React App</title>
  </head>
  <body>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.

      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.

      To begin the development, run `npm start` in this folder.
      To create a production bundle, use `npm run build`.
    -->
  </body>
</html>
```

## Shipping

We'll get to deployment in a few weeks, but for the time being know that the generator created a build command so we can create minified, optimize versions of our app that we can upload to a server.

We can build our app using the `npm run build` command in the root of our project:

```
npm run build
```



```
> 30days@0.0.1 build /Users/auser/Development/javascript/mine/sample-apps/30days/30days
> react-scripts build

Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  47.05 KB  build/static/js/main.e3fa9c36.js
  289 B     build/static/css/main.9a0fe4f1.css

The project was built assuming it is hosted at the server root.
To override this, specify the homepage in your package.json.
For example, add this to build it for GitHub Pages:

  "homepage": "http://myname.github.io/myapp",

The build folder is ready to be deployed.
You may also serve it locally with a static server:

  npm install -g pushstate-server
  pushstate-server build
  open http://localhost:9000

auser@30days $
```

With that, we now have a live-reloading single-page app (SPA) ready for development. Tomorrow, we'll use this new app we built diving into rendering multiple components at run-time.

# Appendix B: ES6

This appendix is a non-exhaustive list of new syntactic features and methods that were added to JavaScript in ES6. These features are the most commonly used and most helpful.

While this appendix doesn't cover ES6 classes, we go over the basics while learning about components in the book. In addition, this appendix doesn't include descriptions of some larger new features like promises and generators. If you'd like more info on those or on any topic below, we encourage you to reference the Mozilla Developer Network's website[196] (MDN).

## Prefer `const` and `let` over `var`

If you've worked with ES5 JavaScript before, you're likely used to seeing variables declared with `var`:

**appendix/es6/const_let.js**

```
var myVariable = 5;
```

Both the `const` and `let` statements also declare variables. They were introduced in ES6.

Use `const` in cases where a variable is never re-assigned. Using `const` makes this clear to whoever is reading your code. It refers to the "constant" state of the variable in the context it is defined within.

If the variable will be re-assigned, use `let`.

We encourage the use of `const` and `let` instead of `var`. In addition to the restriction introduced by `const`, both `const` and `let` are *block scoped* as opposed to *function scoped*. This scoping can help avoid unexpected bugs.

## Arrow functions

There are three ways to write arrow function bodies. For the examples below, let's say we have an array of city objects:

[196] https://developer.mozilla.org

**appendix/es6/arrow_funcs.js**

```
const cities = [
  { name: 'Cairo', pop: 7764700 },
  { name: 'Lagos', pop: 8029200 },
];
```

If we write an arrow function that spans multiple lines, we must use braces to delimit the function body like this:

**appendix/es6/arrow_funcs.js**

```
const formattedPopulations = cities.map((city) => {
  const popMM = (city.pop / 1000000).toFixed(2);
  return popMM + ' million';
});
console.log(formattedPopulations);
// -> [ "7.76 million", "8.03 million" ]
```

Note that we must also explicitly specify a `return` for the function.

However, if we write a function body that is only a single line (or single expression) we can use parentheses to delimit it:

**appendix/es6/arrow_funcs.js**

```
const formattedPopulations2 = cities.map((city) => (
  (city.pop / 1000000).toFixed(2) + ' million'
));
```

Notably, we don't use `return` as it's implied.

Furthermore, if your function body is terse you can write it like so:

**appendix/es6/arrow_funcs.js**

```
const pops = cities.map(city => city.pop);
console.log(pops);
// [ 7764700, 8029200 ]
```

The terseness of arrow functions is one of two reasons that we use them. Compare the one-liner above to this:

**appendix/es6/arrow_funcs.js**

```
const popsNoArrow = cities.map(function(city) { return city.pop });
```

Of greater benefit, though, is how arrow functions bind the `this` object.

The traditional JavaScript function declaration syntax (`function () {}`) will bind `this` in anonymous functions to the global object. To illustrate the confusion this causes, consider the following example:

**appendix/es6/arrow_funcs_jukebox_1.js**

```
function printSong() {
  console.log("Oops - The Global Object");
}

const jukebox = {
  songs: [
    {
      title: "Wanna Be Startin' Somethin'",
      artist: "Michael Jackson",
    },
    {
      title: "Superstar",
      artist: "Madonna",
    },
  ],
  printSong: function (song) {
    console.log(song.title + " - " + song.artist);
  },
  printSongs: function () {
    // `this` bound to the object (OK)
    this.songs.forEach(function(song) {
      // `this` bound to global object (bad)
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Oops - The Global Object"
// > "Oops - The Global Object"
```

The method `printSongs()` iterates over `this.songs` with `forEach()`. In this context, `this` is bound to the object (`jukebox`) as expected. However, the anonymous function passed to `forEach()` binds its internal `this` to the global object. As such, `this.printSong(song)` calls the function declared at the top of the example, *not* the method on `jukebox`.

JavaScript developers have traditionally used workarounds for this behavior, but arrow functions solve the problem by **capturing the `this` value of the enclosing context**. Using an arrow function for `printSongs()` has the expected result:

**appendix/es6/arrow_funcs_jukebox_2.js**

```
  printSongs: function () {
    this.songs.forEach((song) => {
      // `this` bound to same `this` as `printSongs()` (`jukebox`)
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Wanna Be Startin' Somethin' - Michael Jackson"
// > "Superstar - Madonna"
```

For this reason, throughout the book we use arrow functions for all anonymous functions.

## Modules

ES6 formally supports modules using the `import`/`export` syntax.

### Named exports

Inside any file, you can use `export` to specify a variable the module should expose. Here's an example of a file that exports two functions:

```
// greetings.js

export const sayHi = () => (console.log('Hi!'));
export const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));
```

Now, anywhere we wanted to use these functions we could use `import`. We need to specify which functions we want to import. A common way of doing this is using ES6's destructuring assignment syntax to list them out like this:

```
// app.js

import { sayHi, sayBye } from './greetings';

sayHi(); // -> Hi!
sayBye(); // => Bye!
```

Importantly, the function that was *not* exported (`saySomething`) is unavailable outside of the module.

Also note that we supply a **relative path** to `from`, indicating that the ES6 module is a local file as opposed to an npm package.

Instead of inserting an `export` before each variable you'd like to export, you can use this syntax to list off all the exposed variables in one area:

```
// greetings.js

const sayHi = () => (console.log('Hi!'));
const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));

export { sayHi, sayBye };
```

We can also specify that we'd like to import all of a module's functionality underneath a given namespace with the `import * as <Namespace>` syntax:

```
// app.js

import * as Greetings from './greetings';

Greetings.sayHi();
  // -> Hi!
Greetings.sayBye();
  // => Bye!
Greetings.saySomething();
  // => TypeError: Greetings.saySomething is not a function
```

### Default export

The other type of export is a default export. A module can only contain one default export:

```
// greetings.js

const sayHi = () => (console.log('Hi!'));
const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));

const Greetings = { sayHi, sayBye };

export default Greetings;
```

This is a common pattern for libraries. It means you can easily import the library wholesale without specifying what individual functions you want:

```
// app.js

import Greetings from './greetings';

Greetings.sayHi(); // -> Hi!
Greetings.sayBye(); // => Bye!
```

It's not uncommon for a module to use a mix of both named exports and default exports. For instance, with `react-dom`, you can import `ReactDOM` (a default export) like this:

```
import ReactDOM from 'react-dom';

ReactDOM.render(
  // ...
);
```

Or, if you're only going to use the `render()` function, you can import the named `render()` function like this:

```
import { render } from 'react-dom';

render(
  // ...
);
```

To achieve this flexibility, the export implementation for `react-dom` looks something like this:

```
// a fake react-dom.js

export const render = (component, target) => {
  // ...
};

const ReactDOM = {
  render,
  // ... other functions
};

export default ReactDOM;
```

If you want to play around with the module syntax, check out the folder `code/webpack/es6-modules`.

For more reading on ES6 modules, see this article from Mozilla: "ES6 in Depth: Modules[197]".

## `Object.assign()`

We use `Object.assign()` often throughout the book. We use it in areas where we want to create a modified version of an existing object.

`Object.assign()` accepts any number of objects as arguments. When the function receives two arguments, it *copies* the properties of the second object onto the first, like so:

**appendix/es6/object_assign.js**

```
const coffee = { };
const noCream = { cream: false };
const noMilk = { milk: false };
Object.assign(coffee, noCream);
// coffee is now: `{ cream: false }`
```

It is idiomatic to pass in three arguments to `Object.assign()`. The first argument is a new JavaScript object, the one that `Object.assign()` will ultimately return. The second is the object whose properties we'd like to build off of. The last is the changes we'd like to apply:

[197] https://hacks.mozilla.org/2015/08/es6-in-depth-modules/

**appendix/es6/object_assign.js**

```
const coffeeWithMilk = Object.assign({}, coffee, { milk: true });
// coffeeWithMilk is: `{ cream: false, milk: true }`
// coffee was not modified: `{ cream: false, milk: false }`
```

`Object.assign()` is a handy method for working with “immutable” JavaScript objects.

## Template literals

In ES5 JavaScript, you’d interpolate variables into strings like this:

**appendix/es6/template_literals_1.js**

```
var greeting = 'Hello, ' + user + '! It is ' + degF + ' degrees outside.';
```

With ES6 template literals, we can create the same string like this:

**appendix/es6/template_literals_2.js**

```
const greeting = `Hello, ${user}! It is ${degF} degrees outside.`;
```

## The spread operator (`...`)

In arrays, the ellipsis `...` operator will *expand* the array that follows into the parent array. The spread operator enables us to succinctly construct new arrays as a composite of existing arrays.

Here is an example:

**appendix/es6/spread_operator_arrays.js**

```
const a = [ 1, 2, 3 ];
const b = [ 4, 5, 6 ];
const c = [ ...a, ...b, 7, 8, 9 ];

console.log(c); // -> [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

Notice how this is different than if we wrote:

**appendix/es6/spread_operator_arrays.js**

```
const d = [ a, b, 7, 8, 9 ];
console.log(d); // -> [ [ 1, 2, 3 ], [ 4, 5, 6 ], 7, 8, 9 ]
```

## Enhanced object literals

In ES5, all objects were required to have explicit key and value declarations:

**appendix/es6/enhanced_object_literals.js**

```
const explicit = {
  getState: getState,
  dispatch: dispatch,
};
```

In ES6, you can use this terser syntax whenever the property name and variable name are the same:

**appendix/es6/enhanced_object_literals.js**

```
const implicit = {
  getState,
  dispatch,
};
```

Lots of open source libraries use this syntax, so it's good to be familiar with it. But whether you use it in your own code is a matter of stylistic preference.

## Default arguments

With ES6, you can specify a default value for an argument in the case that it is `undefined` when the function is called.

This:

**appendix/es6/default_args.js**

```
function divide(a, b) {
  // Default divisor to `1`
  const divisor = typeof b === 'undefined' ? 1 : b;

  return a / divisor;
}
```

Can be written as this:

**appendix/es6/default_args.js**

```
function divide(a, b = 1) {
  return a / b;
}
```

In both cases, using the function looks like this:

**appendix/es6/default_args.js**

```
divide(14, 2);
// => 7
divide(14, undefined);
// => 14
divide(14);
// => 14
```

Whenever the argument `b` in the example above is `undefined`, the default argument is used. Note that `null` will not use the default argument:

**appendix/es6/default_args.js**

```
divide(14, null); // `null` is used as divisor
// => Infinity    // 14 / null
```

## Destructuring assignments

### For arrays

In ES5, extracting and assigning multiple elements from an array looked like this:

**appendix/es6/destructuring_assignments.js**

```
var fruits = [ 'apples', 'bananas', 'oranges' ];
var fruit1 = fruits[0];
var fruit2 = fruits[1];
```

In ES6, we can use the destructuring syntax to accomplish the same task like this:

**appendix/es6/destructuring_assignments.js**

```
const [ veg1, veg2 ] = [ 'asparagus', 'broccoli', 'onion' ];
console.log(veg1); // -> 'asparagus'
console.log(veg2); // -> 'broccoli'
```

The variables in the array on the left are "matched" and assigned to the corresponding elements in the array on the right. Note that `'onion'` is ignored and has no variable bound to it.

### For objects

We can do something similar for extracting object properties into variables:

**appendix/es6/destructuring_assignments.js**

```
const smoothie = {
  fats: [ 'avocado', 'peanut butter', 'greek yogurt' ],
  liquids: [ 'almond milk' ],
  greens: [ 'spinach' ],
  fruits: [ 'blueberry', 'banana' ],
};

const { liquids, fruits } = smoothie;

console.log(liquids); // -> [ 'almond milk' ]
console.log(fruits); // -> [ 'blueberry', 'banana' ]
```

### Parameter context matching

We can use these same principles to bind arguments inside a function to properties of an object supplied as an argument:

**appendix/es6/destructuring_assignments.js**

```
const containsSpinach = ({ greens }) => {
  if (greens.find(g => g === 'spinach')) {
    return true;
  } else {
    return false;
  }
};

containsSpinach(smoothie); // -> true
```

We do this often with functional React components:

**appendix/es6/destructuring_assignments.js**

```
const IngredientList = ({ ingredients, onClick }) => (
  <ul className='IngredientList'>
    {
      ingredients.map(i => (
        <li
          key={i.id}
          onClick={() => onClick(i.id)}
          className='item'
        >
          {i.name}
        </li>
      ))
    }
  </ul>
)
```

Here, we use destructuring to extract the props into variables (`ingredients` and `onClick`) that we then use inside the component's function body.

# JavaScript Array map() Method

❮ JavaScript Array Reference

## Example

Return an array with the square root of all the values in the original array:

```
var numbers = [4, 9, 16, 25];

function myFunction() {
    x = document.getElementById("demo")
    x.innerHTML = numbers.map(Math.sqrt);
}
```

The result will be:

```
2,3,4,5
```

Try it Yourself »

More "Try it Yourself" examples below.

---

## Definition and Usage

The map() method creates a new array with the results of calling a function for every array element.

The map() method calls the provided function once for each element in an array, in order.

**Note:** map() does not execute the function for array elements without values.

**Note:** map() does not change the original array.

---

## Syntax

```
array.map(function(currentValue, index, arr), thisValue)
```

## Parameter Values

| Parameter | Description |
|---|---|

| | |
|---|---|
| *function(currentValue, index, arr)* | Required. A function to be run for each element in the array.<br>Function arguments: |

| **Argument** | **Description** |
|---|---|
| *currentValue* | Required. The value of the current element |
| *index* | Optional. The array index of the current element |
| *arr* | Optional. The array object the current element belongs to |
| *thisValue* | Optional. A value to be passed to the function to be used as its "this" value. If this parameter is empty, the value "undefined" will be passed as its "this" value |

## Technical Details

| | |
|---|---|
| **Return Value:** | An Array containing the results of calling the provided function for each element in the original array. |
| **JavaScript Version:** | ECMAScript 3 |

## More Examples

### Example

Multiply all the values in array with a specific number:

```
var numbers = [65, 44, 12, 4];

function multiplyArrayElement(num) {
    return num * document.getElementById("multiplyWith").value;
}

function myFunction() {
    document.getElementById("demo").innerHTML = numbers.map(multiplyArrayElement);
}
```

### Example

Get the full name for each person in the array:

```
var persons = [
    {firstname : "Malcom", lastname: "Reynolds"},
    {firstname : "Kaylee", lastname: "Frye"},
    {firstname : "Jayne", lastname: "Cobb"}
];


function getFullName(item, index) {
    var fullname = [item.firstname,item.lastname].join(" ");
    return fullname;
}

function myFunction() {
    document.getElementById("demo").innerHTML = persons.map(getFullName);
}
```

# Javascript Promises

**Javascript Promises** are not difficult. However, lots of people find it a little bit hard to understand at the beginning. Therefore, I would like to write down the way I understand promises, in a dummy way.

## # A Promise in short:

"Imagine you are a **kid**. Your mom **promises** you that she'll get you a **new phone** next week."
You *don't know* if you will get that phone until next week. Your mom can either *really buy*you a brand new phone, or *stand you up* and withhold the phone if she is not happy :(.
That is a **promise**. A promise has 3 states. They are:

1. Promise is **pending**: You don't know if you will get that phone until next week.
2. Promise is **resolved**: Your mom really buy you a brand new phone.
3. Promise is **rejected**: You don't get a new phone because your mom is not happy.

## # Creating a Promise

Let's convert this to JavaScript.

```
/* ES5 */
var isMomHappy = false;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone); // fulfilled
        } else {
            var reason = new Error('mom is not happy');
            reject(reason); // reject
        }

    }
);
```

The code is quite expressive in itself.

1. We have a boolean `isMomHappy`, to define if mom is happy.
2. We have a promise `willIGetNewPhone`. The promise can be either `resolved` (if mom get you a new phone) or `rejected`(mom is not happy, she doesn't buy you one).
3. There is a standard syntax to define a new `Promise`, refer to MDN documentation, a promise syntax look like this.

```
// promise syntax look like this
new Promise(/* executor*/ function (resolve, reject) { ... } );
```

4. What you need to remember is, when the result is successful, call `resolve(your_success_value)`, if the result fails, call `reject(your_fail_value)` in your promise. In our example, if mom is happy, we will get a phone. Therefore, we call `resolve` function with `phone` variable. If mom is not happy, we will call `reject`function with a reason `reject(reason)`;

## # Consuming Promises

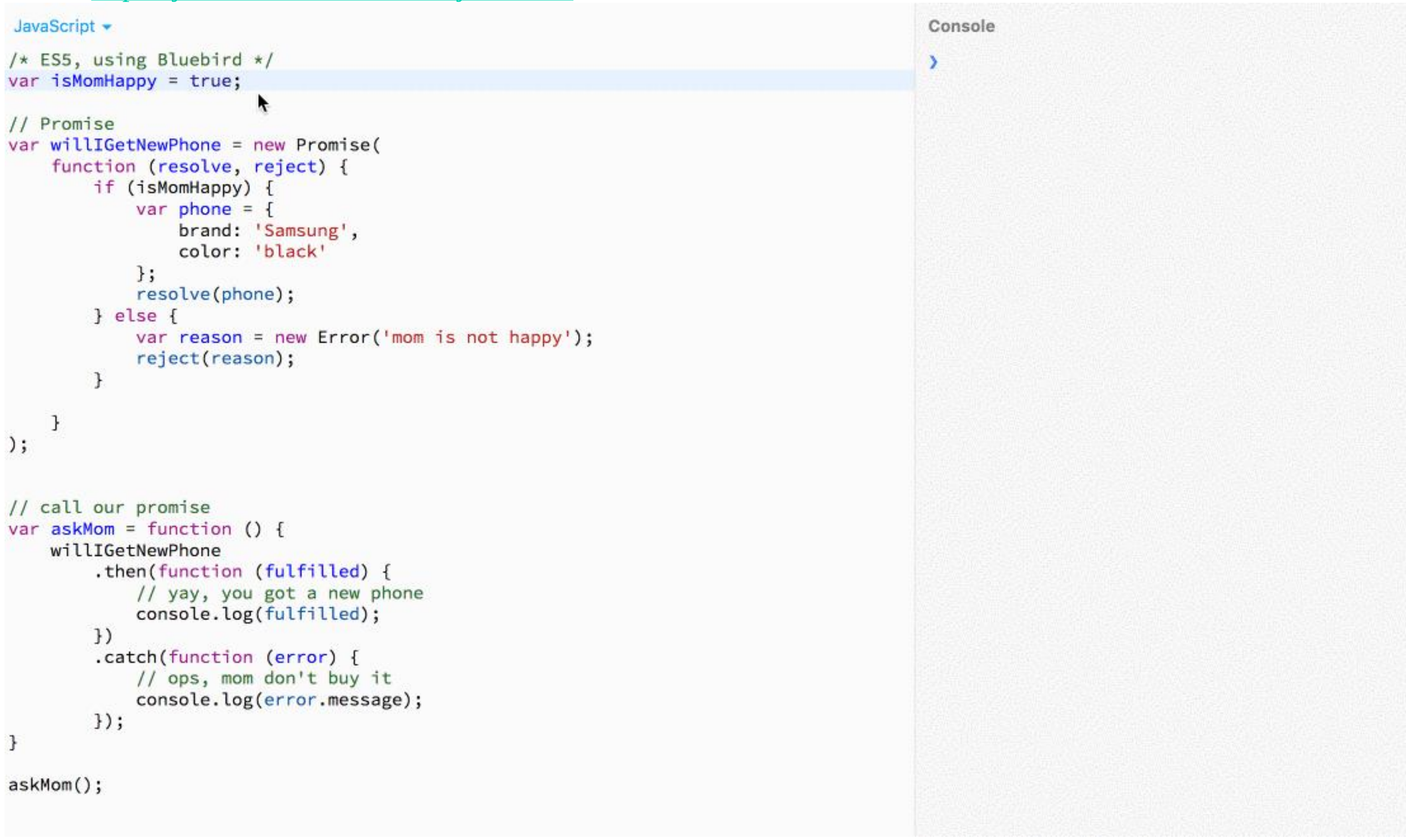
Now that we have the promise, let's consume it.

```
/* ES5 */
...

// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
         // output: { brand: 'Samsung', color: 'black' }
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};

askMom();
```

1. We have a function call `askMom`. In this function, we will consume our promise `willIGetNewPhone`.
2. We want to take some action once the promise is resolved or rejected, we use `.then`and `.catch` to handle our action.
3. In our example, we have `function(fulfilled) { ... }` in `.then`. What is the value of `fulfilled`? The `fulfilled` value is exactly the value you pass in your promise `resolve(your_success_value)`. Therefore, it will be `phone` in our case.
4. We have `function(error){ ... }` in `.catch`. What is the value of `error`? As you can guess, the `error` value is exactly the value you pass in your promise `reject(your_fail_value)`. Therefore, it will be `reason` in our case.

Let's run the example and see the result!

Demo: https://jsbin.com/nifocu/1/edit?js,console



```
/* ES5, using Bluebird */
var isMomHappy = true;

// Promise
var willIGetNewPhone = new Promise(
    function (resolve, reject) {
        if (isMomHappy) {
            var phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            var reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);


// call our promise
var askMom = function () {
    willIGetNewPhone
        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
        })
        .catch(function (error) {
            // ops, mom don't buy it
            console.log(error.message);
        });
}

askMom();
```

## # Chaining Promises

Promises are chainable.
Let's say, you, the kid, **promise** your friend that you will **show them** the new phone when your mom buy you one.
That is another promise. Let's write it!

```
...

// 2nd promise
var showOff = function (phone) {
    return new Promise(
        function (resolve, reject) {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};
```

Notes:

- In this example, you might realize we didn't call the `reject`. It's optional.
- We can shorten this sample like using `Promise.resolve` instead.

```
// shorten it
...

// 2nd promise
var showOff = function (phone) {
    var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

    return Promise.resolve(message);
};
```

Let's chain the promises. You, the kid can only start the `showOff` promise after the `willIGetNewPhone` promise.

```
...

// call our promise
var askMom = function () {
    willIGetNewPhone
    .then(showOff) // chain it here
    .then(function (fulfilled) {
            console.log(fulfilled);
         // output: 'Hey friend, I have a new black Samsung phone.'
        })
        .catch(function (error) {
            // oops, mom don't buy it
            console.log(error.message);
         // output: 'mom is not happy'
        });
};
```

That's how easy to chain the promise.

## # Promises are Asynchronous

Promises are asynchronous. Let's log a message before and after we call the promise.

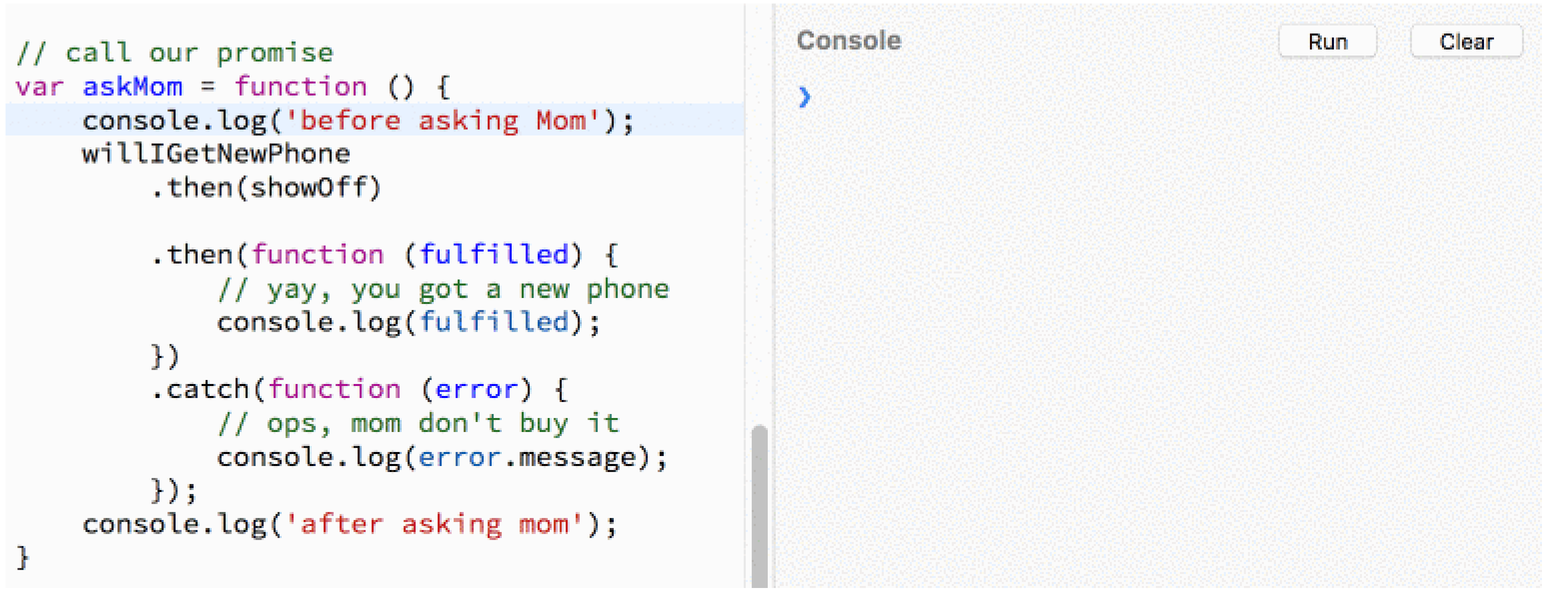
```
// call our promise
var askMom = function () {
    console.log('before asking Mom'); // log before
    willIGetNewPhone
        .then(showOff)
        .then(function (fulfilled) {
            console.log(fulfilled);
        })
        .catch(function (error) {
            console.log(error.message);
        });
    console.log('after asking mom'); // log after
}
```

What is the sequence of expected output? Probably you expect:

```
1. before asking Mom
2. Hey friend, I have a new black Samsung phone.
3. after asking mom
```

However, the actual output sequence is:

```
1. before asking Mom
2. after asking mom
3. Hey friend, I have a new black Samsung phone.
```



```
// call our promise
var askMom = function () {
    console.log('before asking Mom');
    willIGetNewPhone
        .then(showOff)

        .then(function (fulfilled) {
            // yay, you got a new phone
            console.log(fulfilled);
        })
        .catch(function (error) {
            // ops, mom don't buy it
            console.log(error.message);
        });
    console.log('after asking mom');
}
```

### Why? Because life (or JS) waits for no man.

You, the kid, wouldn't stop playing while waiting for your mom promise (the new phone). Don't you? That's something we call **asynchronous**, the code will run without blocking or waiting for the result. Anything that need to wait for promise to proceed, you put that in `.then`.

## # Promises in ES5, ES6/2015, ES7/Next

### ES5 - Majority browsers

The demo code is workable in ES5 environments (all major browsers + NodeJs) if you include Bluebird promise library. It's because ES5 doesn't support promises out of the box. Another famous promise library is Q by Kris Kowal.

### ES6 / ES2015 - Modern browsers, NodeJs v6

The demo code works out of the box because ES6 supports promises natively. In addition, with ES6 functions, we can further simplify the code with **fat arrow** => and use `const` and `let`.

Here is an example of ES6 code:

```
/* ES6 */
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => { // fat arrow
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);

const showOff = function (phone) {
    const message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';
    return Promise.resolve(message);
};
```

```
// call our promise
const askMom = function () {
    willIGetNewPhone
        .then(showOff)
        .then(fulfilled => console.log(fulfilled)) // fat arrow
        .catch(error => console.log(error.message)); // fat arrow
};

askMom();
```

Notes that all the `var` are replaced with `const`. All the `function(resolve, reject)` has been simplified to `(resolve, reject) =>`. There are a few benefits come with these changes. Read more on:-

- JavaScript ES6 Variable Declarations with let and const
- An introduction to Javascript ES6 arrow functions

## ES7 - Async Await make the syntax look prettier

ES7 introduce `async` and `await` syntax. It makes the asynchronous syntax look prettier and easier to understand, without the `.then` and `.catch`.
Rewrite our example with ES7 syntax.

```
/* ES7 */
const isMomHappy = true;

// Promise
const willIGetNewPhone = new Promise(
    (resolve, reject) => {
        if (isMomHappy) {
            const phone = {
                brand: 'Samsung',
                color: 'black'
            };
            resolve(phone);
        } else {
            const reason = new Error('mom is not happy');
            reject(reason);
        }

    }
);
```

```
// 2nd promise
async function showOff(phone) {
    return new Promise(
        (resolve, reject) => {
            var message = 'Hey friend, I have a new ' +
                phone.color + ' ' + phone.brand + ' phone';

            resolve(message);
        }
    );
};

// call our promise
async function askMom() {
    try {
        console.log('before asking Mom');

        let phone = await willIGetNewPhone;
        let message = await showOff(phone);

        console.log(message);
        console.log('after asking mom');
    }
    catch (error) {
        console.log(error.message);
    }
}

(async () => {
    await askMom();
})();
```

1. Whenever you need to return a promise in a function, you prepend `async` to that function. E.g. `async function showOff(phone)`
2. Whenever you need to call a promise, you prepend with `await`. E.g. `let phone = await willIGetNewPhone;` and `let message = await showOff(phone);`.
3. Use `try { ... } catch(error) { ... }` to catch promise error, the **rejected** promise.

# # Why Promises and When to Use Them?
Why do we need promises? How's the world look like before promise? Before answering these questions, let's go back to the fundamental.

## Normal Function vs Async Function
Let's take a look at these two example, both example perform addition of two number, one add using normal function, the other add remotely.

## Normal Function to Add Two Numbers
```
// add two numbers normally


function add (num1, num2) {
    return num1 + num2;
}


const result = add(1, 2); // you get result = 3 immediately
```

## Async Function to Add Two numbers
```
// add two numbers remotely


// get the result by calling an API
const result = getAddResultFromServer('http://www.example.com?num1=1&num2=2');
// you get result  = "undefined"
```

If you add the numbers with normal function, you get the result immediately. However when you issue a remote call to get result, you need to wait, you can't get the result immediately.
Or put it this way, you don't know if you will get the result because the server might be down, slow in response, etc. You don't want your entire process to be blocked while waiting for the result.
Calling APIs, downloading files, reading files are among some of the usual async operations that you'll perform.

## World Before Promises: Callback
```
Must we use promise for asynchronous call? Nope. Prior to Promise, we use callback.
Callback is just a function you call when you get the return result. Let's modify the
previous example to accept a callback.
// add two numbers remotely
// get the result by calling an API


function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}
```

```
addAsync(1, 2, success => {
    // callback
    const result = success; // you get result = 3 here
});
```

The syntax looks ok, why do we need promises then?

## What if You Want to Perform Subsequent Async Action?

Let's say, instead of just add the numbers one time, we want to add 3 times. In a normal function, we do this:-

```
// add two numbers normally

let resultA, resultB, resultC;

 function add (num1, num2) {
    return num1 + num2;
}

resultA = add(1, 2); // you get resultA = 3 immediately
resultB = add(resultA, 3); // you get resultB = 6 immediately
resultC = add(resultB, 4); // you get resultC = 10 immediately

console.log('total' + resultC);
console.log(resultA, resultB, resultC);
```

How it looks like with callbacks?

```
// add two numbers remotely
// get the result by calling an API

let resultA, resultB, resultC;

function addAsync (num1, num2, callback) {
    // use the famous jQuery getJSON callback API
    return $.getJSON('http://www.example.com', {
        num1: num1,
        num2: num2
    }, callback);
}
```

```
addAsync(1, 2, success => {
    // callback 1
    resultA = success; // you get result = 3 here

    addAsync(resultA, 3, success => {
        // callback 2
        resultB = success; // you get result = 6 here

        addAsync(resultB, 4, success => {
            // callback 3
            resultC = success; // you get result = 10 here

            console.log('total' + resultC);
            console.log(resultA, resultB, resultC);
        });
    });
});
```

Demo: https://jsbin.com/barimo/edit?html,js,console
The syntax is less user friendly. In a nicer term, It looks like a pyramid, but people usually refer this as "callback hell", because the callback nested into another callback. Imagine you have 10 callbacks, your code will nested 10 times!

## Escape From Callback Hell

Promises come in to rescue. Let's look at the promise version of the same example.

```
// add two numbers remotely using observable

let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    return fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());
}

addAsync(1, 2)
    .then(success => {
        resultA = success;
        return resultA;
    })
    .then(success => addAsync(success, 3))
    .then(success => {
        resultB = success;
        return resultB;
    })
    .then(success => addAsync(success, 4))
    .then(success => {
        resultC = success;
        return resultC;
    })
    .then(success => {
        console.log('total: ' + success)
        console.log(resultA, resultB, resultC)
    });
```

Demo: https://jsbin.com/qafane/edit?js,console
With promises, we flatten the callback with `.then`. In a way, it looks cleaner because of no callback nesting. Of course, with ES7 `async` syntax, we can even further enhance this example, but I leave that to you. :)

## # New Kid On the Block: Observables

Before you settle down with promises, there is something that has come about to make it even easier to deal with async data called `Observables`.
Observables are lazy event streams which can emit zero or more events, and may or may not finish.

- source

Some key differences between promises and observable are:

- Observables are cancellable
- Observable are lazy

Fear not, let look at the same demo written with Observables. In this example, I am using RxJS for the observables.

```
let Observable = Rx.Observable;
let resultA, resultB, resultC;

function addAsync(num1, num2) {
    // use ES6 fetch API, which return a promise
    const promise = fetch(`http://www.example.com?num1=${num1}&num2=${num2}`)
        .then(x => x.json());

    return Observable.fromPromise(promise);
}
```

```
addAsync(1,2)
  .do(x => resultA = x)
  .flatMap(x => addAsync(x, 3))
  .do(x => resultB = x)
  .flatMap(x => addAsync(x, 4))
  .do(x => resultC = x)
  .subscribe(x => {
    console.log('total: ' + x)
    console.log(resultA, resultB, resultC)
  });
```
Demo: https://jsbin.com/dosaviwalu/edit?js,console
Notes:

- `Observable.fromPromise` converts a promise to observable stream.
- `.do` and `.flatMap` are among some of the operators available for Observables
- Streams are lazy. Our `addAsync` runs when we `.subscribe` to it.

Observables can do more funky stuff easily. For example, `delay` add function by `3 seconds`with just one line of code or retry so you can retry a call a certain number of times.
```
...

addAsync(1,2)
  .delay(3000) // delay 3 seconds
  .do(x => resultA = x)
  ...
```
Well, let's talk about Observables in future post!

## # Summary
Get yourself familiar with callbacks and promises. Understand them and use them. Don't worry about Observables, just yet. All three can factor into your development depending on the situation.
Here are the demo code for all `mom promise to buy phone` examples:

- Demo (ES5): https://jsbin.com/habuwuyeqo/edit?html,js,console
- Demo (ES6): https://jsbin.com/cezedu/edit?js,console
- Demo (ES7): https://goo.gl/U3fPmh
- Github example (ES7): https://github.com/chybie/js-async-await-promise

That's it. Hopefully this article smoothen your path to tame the JavaScript promises. Happy coding!

# Setting Up a React Project

In this unit, you set up a development environment to develop and run an ECMAScript 6 application using Babel and Webpack.

## Step 1: Install the Sample Application

1. Clone the es6-tutorial-react (https://github.com/ccoenraets/es6-tutorial-react/) repository:

   ```
   git clone https://github.com/ccoenraets/es6-tutorial-react
   ```

   Alternatively, you can just download and unzip **this file (https://github.com/ccoenraets/es6-tutorial-react/archive/master.zip)** instead of cloning the repository.

2. Open `app.js` and examine the React implementation of the mortgage calculator application. This version of the application is written with ECMAScript 5.

## Step 2: Set Up Babel and Webpack

1. Open a command prompt, and navigate ( `cd` ) to the `es6-tutorial-react` directory.
2. Type the following command to create a `package.json` file:

   ```
   npm init
   ```

   Press the **Return** key in response to all the questions to accept the default values.
3. Type the following command to install the **react** and **react-dom** modules:

   ```
   npm install react react-dom --save-dev
   ```

4. Type the following command to install the babel and webpack modules:

   ```
   npm install babel-core babel-loader webpack --save-dev
   ```

5. Type the following command to install the ECMAScript 2015 and React presets:

   ```
   npm install babel-preset-es2015 babel-preset-react --save-dev
   ```

6. In the `es6-tutorial-react` directory, create a new file named `webpack.config.js` defined as follows:

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

module.exports = {
    entry: './js/app.js',
    output: {
        path: path.resolve(__dirname, 'build'),
        filename: 'app.bundle.js'
    },
    module: {
        loaders: [
            {
                test: /\.js$/,
                loader: 'babel-loader',
                query: {
                    presets: ['es2015', 'react']
                }
            }
        ]
    },
    stats: {
        colors: true
    },
    devtool: 'source-map'
};
```

7. Open `package.json` in your favorite code editor. In the `scripts` section, remove the **test** script, and add a script named **webpack** that compiles app.js. The `scripts` section should now look like this:

```
"scripts": {
    "webpack": "webpack"
},
```

8. In the `es6-tutorial-react` directory, create a `build` directory to host the compiled version of the application.

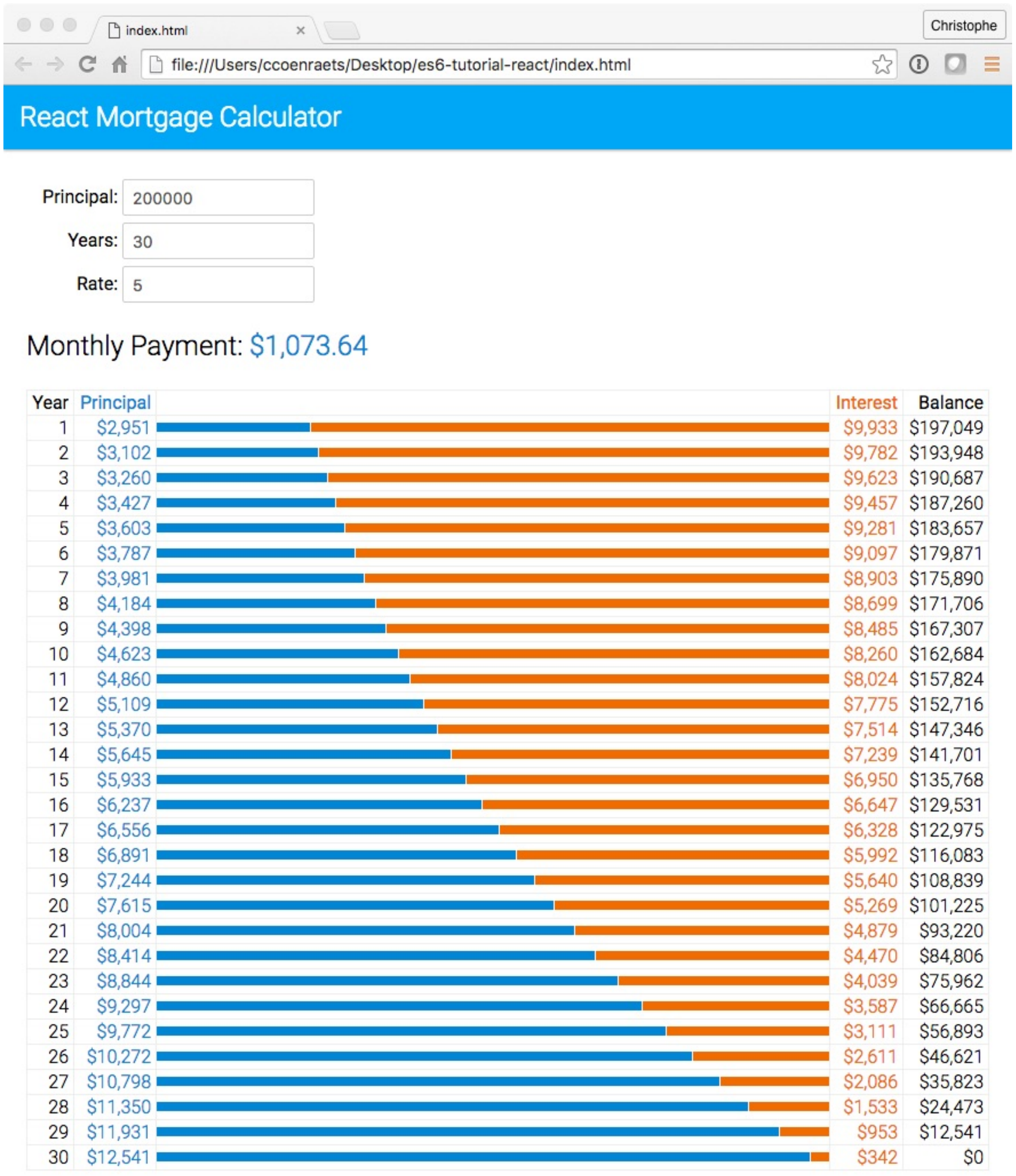
   The build process will fail if you don't create the build directory

## Step 3: Build and Run

1. On the command line, make sure you are in the `es6-tutorial-react` directory, and type the following command to run the **webpack** script and compile app.js:

```
npm run webpack
```

2. Open **index.html** in your browser and test the application


index.html
Christophe
file:///Users/ccoenraets/Desktop/es6-tutorial-react/index.html

React Mortgage Calculator

Principal: 200000

Years: 30

Rate: 5

Monthly Payment: $1,073.64

| Year | Principal | Interest | Balance |
|---|---|---|---|
| 1 | $2,951 | $9,933 | $197,049 |
| 2 | $3,102 | $9,782 | $193,948 |
| 3 | $3,260 | $9,623 | $190,687 |
| 4 | $3,427 | $9,457 | $187,260 |
| 5 | $3,603 | $9,281 | $183,657 |
| 6 | $3,787 | $9,097 | $179,871 |
| 7 | $3,981 | $8,903 | $175,890 |
| 8 | $4,184 | $8,699 | $171,706 |
| 9 | $4,398 | $8,485 | $167,307 |
| 10 | $4,623 | $8,260 | $162,684 |
| 11 | $4,860 | $8,024 | $157,824 |
| 12 | $5,109 | $7,775 | $152,716 |
| 13 | $5,370 | $7,514 | $147,346 |
| 14 | $5,645 | $7,239 | $141,701 |
| 15 | $5,933 | $6,950 | $135,768 |
| 16 | $6,237 | $6,647 | $129,531 |
| 17 | $6,556 | $6,328 | $122,975 |
| 18 | $6,891 | $5,992 | $116,083 |
| 19 | $7,244 | $5,640 | $108,839 |
| 20 | $7,615 | $5,269 | $101,225 |
| 21 | $8,004 | $4,879 | $93,220 |
| 22 | $8,414 | $4,470 | $84,806 |
| 23 | $8,844 | $4,039 | $75,962 |
| 24 | $9,297 | $3,587 | $66,665 |
| 25 | $9,772 | $3,111 | $56,893 |
| 26 | $10,272 | $2,611 | $46,621 |
| 27 | $10,798 | $2,086 | $35,823 |
| 28 | $11,350 | $1,533 | $24,473 |
| 29 | $11,931 | $953 | $12,541 |
| 30 | $12,541 | $342 | $0 |

**Topics**

Setting Up a React Project (/es6-tutorial-react/setup/)

Using ECMAScript 6 Features (/es6-tutorial-react/es6-features/)

Using Modules (/es6-tutorial-react/modules/)

Creating the MortgageCalculator Module (/es6-tutorial-react/modules-more/)

Tweet Follow @ccoenraets

Next (/es6-tutorial-react/es6-features/)

# Using ECMAScript 6 Features

## Step 1: Use New ECMAScript 6 Features

1. Open `js/app.js` in your favorite code editor
2. Replace all `var` definitions with `let`
3. Replace all `React.createClass()` definitions with the new ECMAScript 6 class syntax. For example:

```
class Header extends React.Component {

};
```

4. Modify all class functions to use the ECMAScript 6 syntax for class function definitions. Use the `render()` function below as an example:

```
class Header extends React.Component {

    render() {
        return (
            <header>
                <h1>{this.props.title}</h1>
            </header>
        )
    }

};
```

5. In MortgageCalculator, replace `getInitialState()` with a constructor implemented as follows:

```
constructor(props) {
    super(props);
    this.state = {
        principal: this.props.principal,
        years: this.props.years,
        rate: this.props.rate
    };
}
```

6. In the `render()` function of the MortgageCalculator class, bind the call to the change event handlers as follows:

```
 <input type="text" value={this.state.principal}
        onChange={this.principalChange.bind(this)}/>
```

7. Replace all remaining `function()` definitions with arrow functions
8. Use Object Destructuring syntax when appropriate

## Step 2: Build and Run

1. Build the app:

```
npm run webpack
```

2. Open **index.html** in your browser and test the application

## Solution

The final code for this step is available in this branch (https://github.com/ccoenraets/es6-tutorial-react/tree/step2).

**Topics**

Setting Up a React Project (/es6-tutorial-react/setup/)

Using ECMAScript 6 Features (/es6-tutorial-react/es6-features/)

Using Modules (/es6-tutorial-react/modules/)

Creating the MortgageCalculator Module (/es6-tutorial-react/modules-more/)

Tweet Follow @ccoenraets

Previous (/es6-tutorial-react/setup/)

Next (/es6-tutorial-react/modules/)

We were unable to load Disqus. If you are a moderator please see our troubleshooting guide.

Written by Christophe Coenraets (@ccoenraets (https://twitter.com/ccoenraets)), Developer Evangelist at Salesforce (http://developer.salesforce.com).

# Using Modules

In this unit, you externalize the Application Header in an ECMAScript 6 module.

## Step 1: Create the Header Module

1. Create a new file named `Header.js` in the `js` directory.
2. Import the `react` module:

    ```
    import React from 'react';
    ```

3. Move the `Header` class definition from `app.js` into `Header.js`.
4. At the bottom of the file, export the class as the default export:

    ```
    export default Header;
    ```

## Step 2: Use the Header Module

1. In `app.js` make sure you removed the `Header` class definition
2. Add an `import` statement at the top of the file to import the newly created Header module:

    ```
    import Header from './Header';
    ```

## Step 3: Build and Run

1. Build the app:

    ```
    npm run webpack
    ```

2. Open **index.html** in your browser and test the application

## Solution

The final code for this step is available in this branch (https://github.com/ccoenraets/es6-tutorial-react/tree/step3).

**Topics**

Setting Up a React Project (/es6-tutorial-react/setup/)

Using ECMAScript 6 Features (/es6-tutorial-react/es6-features/)

Using Modules (/es6-tutorial-react/modules/)

Creating the MortgageCalculator Module (/es6-tutorial-react/modules-more/)

Tweet Follow @ccoenraets

Previous (/es6-tutorial-react/es6-features/) Next (/es6-tutorial-react/modules-more/)

# Creating the MortgageCalculator Module

In this unit, you create the MortgageCalculator module as well as the modules it depends on.

## Step 1: Create the mortgage Module

1. Create a new file named `mortgage.js` in the `js` directory.
2. Move the `calculatePayment()` function from `app.js` into `mortgage.js` .
3. Add `export` in front of the function definition to make it available as part of the module public API:

```
export let calculatePayment = (principal, years, rate) => {
```

## Step 2: Create the AmortizationChart Module

1. Create a new file named `AmortizationChart.js` in the `js` directory.
2. Import the react module:

```
import React from 'react';
```

3. Move the `AmortizationChart` class definition from `app.js` into `AmortizationChart.js` .
4. At the bottom of the file, export the class as the default export:

```
export default AmortizationChart;
```

## Step 3: Create the MortgageCalculator Module

1. Create a new file named `MortgageCalculator.js` in the `js` directory.
2. Add the following import statements at the top of the file:

```
import React from 'react';
import AmortizationChart from './AmortizationChart';
import * as mortgage from './mortgage';
```

3. Move the `MortgageCalculator` class definition from `app.js` into `MortgageCalculator.js` .
4. In the `render()` function, change `calculatePayment()` to `mortgage.calculatePayment()`
5. At the bottom of the file, export the class as the default export:

```
export default MortgageCalculator;
```

6. Open `app.js` and add an import statement for the MortgageCalculator module:

```
import MortgageCalculator from './MortgageCalculator';
```

`app.js` should now look like this:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Header from './Header';
import MortgageCalculator from './MortgageCalculator';

class App extends React.Component {

    render() {
        return (
            <div>
                <Header title="React ES6 Mortgage Calculator"/>
                <MortgageCalculator principal="200000" years="30" rate="5"/>
            </div>
        );
    }

};

ReactDOM.render(<App/>, document.getElementById("app"));
```

## Step 3: Build and Run

1. Build the application:

```
npm run webpack
```

2. Open **index.html** in your browser and test the application

## Solution

The final code for this step is available in this branch (https://github.com/ccoenraets/es6-tutorial-react/tree/step4).

**Topics**

Setting Up a React Project (/es6-tutorial-react/setup/)

Using ECMAScript 6 Features (/es6-tutorial-react/es6-features/)

Using Modules (/es6-tutorial-react/modules/)

Creating the MortgageCalculator Module (/es6-tutorial-react/modules-more/)

Tweet Follow @ccoenraets

Previous (/es6-tutorial-react/modules/)

We were unable to load Disqus. If you are a moderator please see our troubleshooting guide.

Written by Christophe Coenraets (@ccoenraets (https://twitter.com/ccoenraets)), Developer Evangelist at Salesforce (http://developer.salesforce.com).