

React 101

Introduction

This lesson assumes a basic knowledge of JavaScript. It will be assumed you have read and understand the concepts in the Level 1 and Level 2 JavaScript lessons. There are some concepts that are unique to React or Redux (or just heavily used in React and Redux and possibly not as much in vanilla JS) where a passable explanation will be given. If you want further enrichment, explore it! That's how you learn!

A few notes before we start:

1. You might have already noticed that I continually say "React and Redux" as if they must go together. This is not the case. React can stand alone, and there are other choices when it comes to state management apart from Redux. Flex is one, and now React comes out-of-the-box with the ability to use "context". Flex has largely lost the war to Redux, and context does not scale quite as well. Not using a state management library at all becomes very difficult as your app scales. There are as many opinions online as there are articles. Most agree that the downside of Redux is the learning curve and the upside is app scalability. Most rightly state that you can get by with a "small" app without a state manager. I don't see a reason to do this. Redux is light-weight enough to use in an app of any size. As for the learning curve, it's a "pay now or pay later" kind of thing. Invest the time and energy to figure it out and you will understand your app better, be able to separate concerns far better, and best of all, you will be able to access global state anywhere in your app. This is really awesome, and you'll see why as we go on.
2. This will – at times – be an attempt to distill down a Udemy course by Maximilian Schwarzmüller. His course is exhaustive and if you really want to understand React and Redux, you should take the course: <https://www.udemy.com/react-the-complete-guide-incl-redux/learn/v4/overview> That said, it's 36 hours of video and we need a more accessible primer.
3. This guide will assume you are building or working with a Single Page Application (SPA). This is the biggest usecase for React and is how much of React's benefits are realized. This is not the only way. When a legacy site is being migrated, or in a handful of other instances React can be "injected" into different subordinate parts of a web app. React is most useful when the power of its component architecture is leveraged from top to bottom. Having multiple injection sites breaks this continuity. Another disadvantage is that when React is mounted to different DOM elements they are essentially separate islands and cannot easily pass props nor reliably share state. This is only mentioned to explain that there are some use cases that prove an exception to the rule, but the SPA implementation is much more ubiquitous.
4. You will see syntax that you may not have seen when studying JS. This is EcmaScript 2015 (ES2015), (*also confusingly called ES6*). ES2015 goes hand-in-hand with React. It is not mandatory, but it is expected in the way that TypeScript is expected with Angular. You will just about always see them together in the wild. It's so expected that it's worth making a diversion to highlight a few of the differences. The next section will hit the changes you are most likely to see.

EcmaScript 2015 (ES2015/ES6) - The JS Features You Didn't Know You Wanted

ES2015 and ES6 are purposely used interchangeably in this lesson because you may see them both. They are two names for the same thing. Most of the changes are syntactic sugar that make life easier for developers. "Syntactic sugar" might make it sound like these features are window dressing and are unimportant. That's not the case, particularly in the case of React which makes use of things like ES2015 classes and modules as best practices.

Anyone who has played the browser compatibility game knows that one cannot simply start using an awesome new JavaScript version. It has to be transpiled into ES5 using something like Babel. That's important to keep in mind for troubleshooting. You are writing ES6 and the browser is running ES5. This will likely be the case until you retire because we are all slaves to IE even if they don't have any market share.

For a more thorough list and for more complicated examples, see Luke Hoban's write-up on ES6 features:

<https://github.com/lukehoban/es6features#readme>

We will cover the big changes, particularly the ones that are used a lot in React.

Arrow Functions:

```
// EcmaScript 5 (The JS you know):
function add(a, b){
  console.log(a + b);
}

// ES2015 (The JS you'll love):
const add = (a, b) => {
  console.log(a + b);
};
```

Ignore the `const` for now. This could have been a `var` instead, but would have been bad form, as you will see below. This looks like an academic difference at first glance, however unlike a normal function where `this` needs to be explicitly bound to its surrounding code, an arrow function shares the lexical `this` context with its surrounding code. Please review the JavaScript lessons for further information on lexical scope. In any case, this will become a major convenience in classes, which are also new.

Classes:

```
class QuantumLeapIntro extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: 'Dr. Sam Beckitt',
      machine: 'Quantum Leap Accelerator'
    }
  }

  render() {
```

```
        return <p>{this.state.name} stepped into the {this.state.machine} and  
        vanished!</p>;  
    }  
}
```

JS implements object orientation through "prototypes". Classes provide an abstraction that is syntactic sugar over the prototype paradigm. This allows the use of constructors, allows the component to have state, and allows the use of lifecycle methods which provide fine-grained control of the component. More on that later. All you need to know right now is that classes exist now and we're gonna use them.

String Templates:

```
var templates = 'String templates';  
var humble = 'Not as flashy as some other changes';  
var stillCool = 'one of my very favorites';  
  
console.log(`${templates}: ${humble}, but ${stillCool}!`);  
  
// Output:  
// "String templates: Not as flashy as some other changes, but one of my very  
// favorites!"
```

This is a convenience thing. Note the use of back ticks [`] surrounding the string.

Const and Let:

This one is kind of a big deal. As a best practice, the `var` keyword has been replaced. `var` will still work. In fact, not only will it still work, but once Babel is finished everything will be turned right back into `var` anyway. On the ES6 front, however, we want to use `const` and `let`.

`let` - used when we are going to change the contents of a variable. **This is the new `var`!**

`const` - use this any other time.

`const` cannot be reassigned. Rule of thumb: Use `const` unless you have to use `let`. I have found myself using `const` far more often than `let`. I'm "borrowing" that guy's example because it's a really good one:

```
function f() {  
  {  
    let x;  
    {  
      // this is ok since it's a block scoped name  
      const x = "sneaky";  
  
      // error, was just defined with `const` above  
      x = "foo";  
    }  
  }  
  // this is ok since it was declared with `let`  
  x = "bar";  
}
```

```
    // error, already declared above in this block
    let x = "inner";
  }
}
```

Default Parameters:

Just what it sounds like. If you pass an argument then it's whatever you passed in. If not, it falls back to the default.

```
function hiMyNameIs( noise, name = 'Slim Shady' ){
  console.log(`Hi! My name is (what?) My name is (Who?) My name is ${noise}
${name}!`);
}

hiMyNameIs('chicka chicka');
// Prints:
// "Hi! My name is (what?) My name is (Who?) My name is chicka chicka Slim Shady!"

hiMyNameIs('chicka chicka', 'Jeff Barras');
// Prints:
// "Hi! My name is (what?) My name is (Who?) My name is chicka chicka Jeff
Barras!"
```

Rest Parameters

In a function's parameter, using a rest parameter allows you to pass in any number of arguments to the function. They will be stored as an array with the name you put after the ellipses in the rest parameter.

```
function howManyArgs(firstArg, ...otherArgs){
  let numArgs = 0;
  // Add one for firstArg
  numArgs += 1;
  // Add however many other arguments you passed in
  numArgs += otherArgs.length;

  console.log('There are a total of ' + numArgs + ' args.');
```

```
}

howManyArgs('I', 'am', 'getting', 'tired', 42);
// Prints:
// There are a total of 5 args.
```

Spread Operators

Reverse of rest parameters. A spread operator is used to make a list of arguments out of an array.

```
let arr = [3, 5, 1];
let arr2 = [8, 9, 15];

let merged = [0, ...arr, 2, ...arr2];

alert(merged); // 0,3,5,1,2,8,9,15 (0, then arr, then 2, then arr2)
```

ES2015 Modules

There is now a built in Asynchronous module loader whereas we used to have to use patterns like AMD or CommonJS. AMD has a complicated syntax, but loads modules asynchronously. CommonJS has a compact syntax, but is synchronous. ES2015 modules take the best of both. The syntax is compact and the modules are loaded asynchronously.

There are two ways export. Either by name, where you can have several exports in a single module, or as a default export, which has one export per module. We will get some practice with these as the lessons go on, but for now we'll just take a look at the syntax.

Named exports/imports:

```
//-----lib.js-----
export const sqrt = Math.sqrt;
export function square(x){
  return x * x;
}
export function diag(x, y){
  return sqrt(square(x) + square(y));
}

//-----main.js-----
import {square, diag} from 'lib';
console.log(square(11)); // 121
console.log(diag(4, 3)); // 5
```

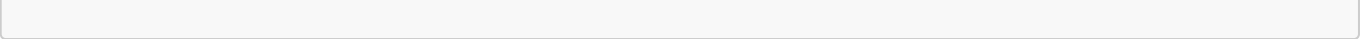
Note the names in curly braces in the import

Default export/import:

```
//-----myFunc.js-----
function foo(){
  console.log('Hello from myFunc.js');
}

export default foo;

//-----main1.js-----
import foo from 'myFunc';
foo();
```



Classes can exported and imported the same way!