

# JavaScript fundamentals before learning React

JULY 14, 2019 BY ROBIN WIERUCH - [EDIT THIS POST](#)

 Follow on Twitter

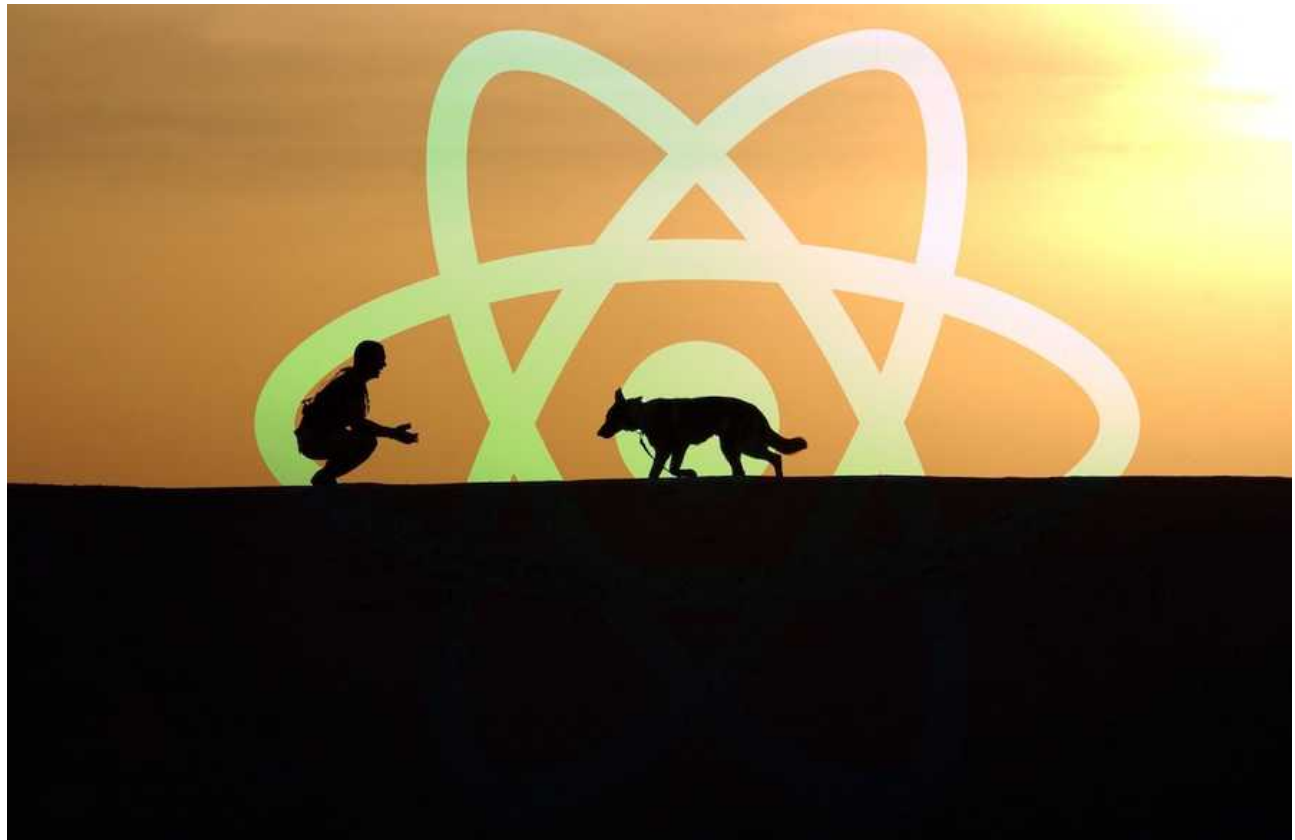
17k

 Follow on Facebook

f



in



After all my teachings about React, be it online for a larger audience or on-site for companies transitioning to web development and React, I always come to the conclusion that React is all about JavaScript. Newcomers to React but also myself see it as an [advantage](#), because you carry your JavaScript knowledge for a longer time around compared to your React skills.

During my workshops, the larger part of the material is about JavaScript and not React. Most of it boils down to JavaScript ES6 and beyond -- features and syntax -- but also ternary operators, shorthand versions in the language, the `this` object, JavaScript built-in functions ([map](#), [reduce](#), [filter](#)) or more general concepts such as [composability](#), [reusability](#), immutability, [closures](#), truth tables, or higher-order functions. These are the fundamentals, which you don't need necessarily to master before starting with React, but which will definitely come up while learning or practicing it.

The following walkthrough is my attempt giving you an almost extensive yet concise list about all the different JavaScript functionalities that complement your React knowledge. If you have any other things which are not in the list, just leave a comment for this article and I will keep it up to date.

---

## TABLE OF CONTENTS

- Entering React after learning JavaScript
- React and JavaScript Classes
- Arrow Functions in React
- Functions as Components in React
- React Class Component Syntax
- Template Literals in React
- Map, Reduce and Filter in React
- var, let, and const in React
- Ternary Operator in React
- Import and Export Statements in React
- Libraries in React
- Async/Await in React
- Higher-Order Functions in React
- Shorthand Object Assignment
- Destructuring in React
- Spread Operator in React
- There is more JavaScript than React



{{% package\_box "The Road to React" "Build a Hacker News App along the way. No setup configuration. No tooling. No Redux. Plain React in 200+ pages of learning material. Pay what you want like **50.000+ readers**." "Get the Book" "img/page/cover.png" "https://roadtoreact.com/" %}}

---

## ENTERING REACT AFTER LEARNING JAVASCRIPT

When you enter the world of React, you are often confronted with a React Class Component:

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

```
import './App.css';

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <img src={logo} className="App-logo" alt="logo" />
          <h1>
            Hello React
          </h1>
          <a href="https://reactjs.org">
            Learn React
          </a>
        </header>
      </div>
    );
  }
}

export default App;
```



In a React class component, there are lots of things to digest for beginners which are not necessarily React: class statements, class methods and inheritance due to being a class. Also



JavaScript import statements are only adding complexity when learning React. Even though the main focus point should be JSX (React's syntax) -- everything in the return statement -- in



the very beginning, often all the things around demand explanations as well. This article is supposed to shed some light into all the things around, most of it JavaScript, without worrying too much about React.

---

## REACT AND JAVASCRIPT CLASSES

Being confronted with a React class component, requires the prior knowledge about JavaScript classes. One would assume that this is given knowledge, but it isn't, because JavaScript classes are fairly new in the language. Previously, there was only *JavaScript's prototype chain* which has been used for inheritance too. JavaScript classes build up on top of the prototypical inheritance giving the whole thing a simpler representation with syntactic sugar.

In order to understand JavaScript classes, you can take some time learning about them without React:

```
class Developer {
```

```
constructor(firstname, lastname) {
  this.firstname = firstname;
  this.lastname = lastname;
}

getName() {
  return this.firstname + ' ' + this.lastname;
}

var me = new Developer('Robin', 'Wieruch');

console.log(me.getName());
```

A class describes an **entity** which is used as a blueprint to create an **instance** of this entity. Once an instance of the class gets created with the new statement, the constructor of the class is called which instantiates the instance of the class. Therefore, a class can have properties which are usually located in its constructor. In addition, class methods (e.g. `getName()`) are used to read (or write) data of the instance. The instance of the class is represented as the `this` object within the class, but outside the instance is just assigned to a JavaScript variable.

**f****tw****in**

Usually classes are used for inheritance in object-oriented programming. They are used for the same in JavaScript whereas the `extends` statement can be used to inherit with one class from another class. The more specialized class inherits all the abilities from the more general class with the `extends` statement, and can add its specialized abilities to it:

```
class Developer {
  constructor(firstname, lastname) {
    this.firstname = firstname;
    this.lastname = lastname;
  }

  getName() {
    return this.firstname + ' ' + this.lastname;
  }
}

class ReactDeveloper extends Developer {
  getJob() {
    return 'React Developer';
  }
}

var me = new ReactDeveloper('Robin', 'Wieruch');

console.log(me.getName());
console.log(me.getJob());
```

Basically that's all it needs to fully understand React class components. A JavaScript class is used for defining a React component, but as you can see, the React component is only a "React component" because it inherits all the abilities from the actual React Component class which is imported from the React package:

```
import React, { Component } from 'react';

class App extends Component {
  render() {
    return (
      <div>
        <h1>Welcome to React</h1>
      </div>
    );
  }
}

export default App;
```

**f** That's why the `render()` method is mandatory in React class components: The React Component from the imported React package instructs you to use it for displaying something in the browser. Furthermore, without extending from the React Component, you wouldn't be able to use other **lifecycle methods**. For instance, there wouldn't be a `componentDidMount()` lifecycle method, because the component would be an instance of a vanilla JavaScript class. **in** And not only the lifecycle methods would go away, React's API methods such as `this.setState()` for local state management wouldn't be available as well.

However, as you have seen, using a JavaScript class is beneficial for extending the general class with your specialized behavior. Thus you can introduce your own class methods or properties.

```
import React, { Component } from 'react';

class App extends Component {
  getGreeting() {
    return 'Welcome to React';
  }

  render() {
    return (
      <div>
        <h1>{this.getGreeting()}</h1>
      </div>
    );
  }
}
```

```
export default App;
```

Now you know why React uses JavaScript classes for defining React class components. They are used when you need access to React's API (lifecycle methods, `this.state` and `this.setState()`). In the following, you will see how React components can be defined in a different way without using a JavaScript class.

After all, JavaScript classes welcome one using inheritance in React, which isn't a desired outcome for React, *because React favors composition over inheritance*. So the only class you should extend from your React components should be the official React Component.

## ARROW FUNCTIONS IN REACT

When teaching someone about React, I explain *JavaScript arrow functions* pretty early. They are one of JavaScript's language additions in ES6 which pushed JavaScript forward in functional programming.



```
// JavaScript ES5 function
function getGreeting() {
  return 'Welcome to JavaScript';
}

// JavaScript ES6 arrow function with body
const getGreeting = () => {
  return 'Welcome to JavaScript';
}

// JavaScript ES6 arrow function without body and implicit return
const getGreeting = () =>
  'Welcome to JavaScript';
```

JavaScript arrow functions are often used in React applications for keeping the code concise and readable. I love them, teach them early, but always try to refactor my functions from JavaScript ES5 to ES6 functions along the way. At some point, when the differences between JavaScript ES5 functions and JavaScript ES6 functions become clear, I stick to the JavaScript ES6 way of doing it with arrow functions. However, I always see that too many different syntaxes can be overwhelming for React beginners. So I try to make the different characteristics of JavaScript functions clear before going all-in using them in React. In the following sections, you will see how JavaScript arrow functions are commonly used in React.

# FUNCTIONS AS COMPONENTS IN REACT

React uses the best of different programming paradigms. That's only possible because JavaScript is a many-sided programming language. On the object-oriented programming side, React's class components are a great way of leveraging the abilities of JavaScript classes (inheritance for the React component API, class methods and class properties such as `this.setState()` and `this.state`). On the other side, there are lots of concepts from functional programming used in React (and its ecosystem) too. For instance, **React's function components** are another way of defining components in React. The question which led to function components in React: What if components could be used like functions?

```
function (props) {  
  return view;  
}
```



It's a function which receives an input (e.g. `props`) and returns the displayed HTML elements (view). Under the hood, the function only needs to use the rendering mechanism of the `render()` method from React components:



```
function Greeting(props) {  
  return <h1>{props.greeting}</h1>;  
}
```

Function components are the preferred way of defining components in React. They have less boilerplate, add less complexity, and are simpler to maintain than React class components. You can easily migrate your class components to function components with React Hooks.

Previously, the article mentioned JavaScript arrow functions and how they improve your React code. Let's apply these kind of functions to your function components. The previous Greeting component has two different looks with JavaScript ES5 and ES6:

```
// JavaScript ES5 function  
function Greeting(props) {  
  return <h1>{props.greeting}</h1>;  
}
```

```
// JavaScript ES6 arrow function  
const Greeting = (props) => {  
  return <h1>{props.greeting}</h1>;  
}
```

```
// JavaScript ES6 arrow function
// without body and implicit return
const Greeting = (props) =>
  <h1>{props.greeting}</h1>;
```

JavaScript arrow functions are a great way of keeping your function components in React concise. Even more when there is no computation in between and thus the function body and return statement can be left out.

## REACT CLASS COMPONENT SYNTAX

React's way of defining components evolved over time. In its early stages, the `React.createClass()` method was the default way of creating a React class component. Nowadays, it isn't used anymore, because with the rise of JavaScript ES6, the previously used React class component syntax became the default (only before React function components were introduced).



However, JavaScript is evolving constantly and thus JavaScript enthusiasts pick up new ways of doing things all the time. That's why you will find often different syntaxes for React class components. One way of defining a React class component, with state and class methods, is the following:

```
class Counter extends Component {
  constructor(props) {
    super(props);

    this.state = {
      counter: 0,
    };

    this.onIncrement = this.onIncrement.bind(this);
    this.onDecrement = this.onDecrement.bind(this);
  }

  onIncrement() {
    this.setState(state => ({ counter: state.counter + 1 }));
  }

  onDecrement() {
    this.setState(state => ({ counter: state.counter - 1 }));
  }

  render() {
```

```

    return (
      <div>
        <p>{this.state.counter}</p>

        <button
          onClick={this.onIncrement}
          type="button"
        >
          Increment
        </button>
        <button
          onClick={this.onDecrement}
          type="button"
        >
          Decrement
        </button>
      </div>
    );
  }
}

```



However, when implementing lots of React class components, the **binding of class methods** in the constructor -- and having a constructor in the first place -- becomes a tedious implementation detail. Fortunately, there is a shorthand syntax for getting rid of both:

```

class Counter extends Component {
  state = {
    counter: 0,
  };

  onIncrement = () => {
    this.setState(state => ({ counter: state.counter + 1 }));
  }

  onDecrement = () => {
    this.setState(state => ({ counter: state.counter - 1 }));
  }

  render() {
    return (
      <div>
        <p>{this.state.counter}</p>

        <button
          onClick={this.onIncrement}
          type="button"
        >
          Increment
        </button>
        <button
          onClick={this.onDecrement}

```

```
        type="button"  
      >  
        Decrement  
      </button>  
    </div>  
  );  
}  
}
```

By using JavaScript arrow functions, you can auto-bind class methods without having to bind them in the constructor. Also the constructor can be left out, when not using the props, by defining the state directly as a class property.

*Note: Be aware that **class properties** are not in the JavaScript language yet.) Therefore you can say that this way of defining a React class component is way more concise than the other version.*

## f

## TEMPLATE LITERALS IN REACT



**Template literals** are another JavaScript language specific feature that came with JavaScript ES6. It is worth to mention it shortly, because when people new to JavaScript and React see them, they can be confusing as well. When learning JavaScript, it's the following syntax that you grow up with for concatenating a string:

```
function getGreeting(what) {  
  return 'Welcome to ' + what;  
}  
  
const greeting = getGreeting('JavaScript');  
console.log(greeting);  
// Welcome to JavaScript
```

Template literals can be used for the same which is called string interpolation:

```
function getGreeting(what) {  
  return `Welcome to ${what}`;  
}
```

You only have to use backticks and the `${ }` notation for inserting JavaScript primitives. However, string literals are not only used for string interpolation, but also for multiline strings

in JavaScript:

```
function getGreeting(what) {  
  return `  
    Welcome  
    to  
    ${what}  
  `;  
}
```

Basically that's how larger text blocks can be formatted on multiple lines. For instance, it can be seen with the recent introduction of GraphQL in JavaScript, because GraphQL queries are composed with template literals. Also React Styled Components makes use of template literals.

---

## MAP, REDUCE AND FILTER IN REACT

f



What's the best approach teaching the JSX syntax for React beginners? Usually I start out with defining a variable in the render() method and using it as JavaScript in HTML in the return block.

in

```
import React from 'react';  
  
const App = () => {  
  var greeting = 'Welcome to React';  
  return (  
    <div>  
      <h1>{greeting}</h1>  
    </div>  
  );  
};  
  
export default App;
```

You only have to use the curly braces to get your JavaScript in HTML. Going from rendering a string to a complex object isn't any different.

```
import React from 'react';  
  
const App = () => {  
  var user = { name: 'Robin' };  
  return (  

```

```
    <div>
      <h1>{user.name}</h1>
    </div>
  );
}

export default App;
```

Usually the next question then is: **How to render a list of items?** That's one of the best parts about explaining React in my opinion. There is no React specific API such as a custom attribute on a HTML tag which enables you to render multiple items in React. You can use plain JavaScript for iterating over the list of items and returning HTML for each item.

```
import React from 'react';

const App = () => {
  var users = [
    { name: 'Robin' },
    { name: 'Markus' },
  ];

  return (
    <ul>
      {users.map(function (user) {
        return <li>{user.name}</li>;
      })}
    </ul>
  );
};

export default App;
```

Having used the JavaScript arrow function before, you can get rid of the arrow function body and the return statement which leaves your rendered output way more concise.

```
import React from 'react';

const App = () => {
  var users = [
    { name: 'Robin' },
    { name: 'Markus' },
  ];

  return (
    <ul>
      {users.map(user => <li>{user.name}</li>)}
    </ul>
  );
};
```

```
}  
  
export default App;
```

Pretty soon, every React developer becomes used to the built-in JavaScript `map()` methods for arrays. It just makes so much sense to map over an array and return the rendered output for each item. The same can be applied for custom tailored cases where `filter()` or `reduce()` make more sense rather than rendering an output for each mapped item.

```
import React from 'react';  
  
const App = () => {  
  var users = [  
    { name: 'Robin', isDeveloper: true },  
    { name: 'Markus', isDeveloper: false },  
  ];  
  
  return (  
    <ul>  
      {users  
        .filter(user => user.isDeveloper)  
        .map(user => <li>{user.name}</li>)  
      }  
    </ul>  
  );  
};  
  
export default App;
```



In general, that's how React developers are getting used to these JavaScript built-in functions without having to use a React specific API for it. It is just JavaScript in HTML.

---

## VAR, LET, AND CONST IN REACT

Also the different variable declarations with `var`, `let` and `const` can be confusing for beginners to React even though they are not React specific. Maybe it is because JavaScript ES6 was introduced when React became popular. In general, I try to introduce `let` and `const` very early in my workshops. It simply starts with exchanging `var` with `const` in a React component:

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

```
const App = () => {  
  const users = [  
    { name: 'Robin' },  
    { name: 'Markus' },  
  ];  
  
  return (  
    <ul>  
      {users.map(user => <li>{user.name}</li>)}  
    </ul>  
  );  
};  
  
export default App;
```

Then I give the rules of thumb when to use which variable declaration:

- (1) don't use var anymore, because let and const are more specific
- (2) default to const, because it cannot be re-assigned or re-declared
- (3) use let when re-assigning the variable

**f****in**

While let is usually used in a for loop for incrementing the iterator, const is normally used for keeping JavaScript variables unchanged. Even though it is possible to change the inner properties of objects and arrays when using const, the variable declaration shows the intent of keeping the variable unchanged though.

## TERNARY OPERATOR IN REACT

But it doesn't end with displaying JavaScript strings, objects, and arrays in React. What about an if-else statement for enabling conditional rendering? You cannot use an if-else statement directly in JSX, but you can return early from the rendering function. Returning null is valid in React when displaying nothing.

```
import React from 'react';  
  
const App = () => {  
  const users = [  
    { name: 'Robin' },  
    { name: 'Markus' },  
  ];  
  
  const showUsers = false;  
  
  if (!showUsers) {
```

```
    return null;
  }

  return (
    <ul>
      {users.map(user => <li>{user.name}</li>)}
    </ul>
  );
};

export default App;
```

However, if you want to use an if-else statement within the returned JSX, you can do it by using a JavaScripts ternary operator:

```
import React from 'react';

const App = () => {
  const users = [
    { name: 'Robin' },
    { name: 'Markus' },
  ];

  const showUsers = false;

  return (
    <div>
      {showUsers ? (
        <ul>
          {users.map(user => (
            <li>{user.name}</li>
          ))}
        </ul>
      ) : null}
    </div>
  );
};

export default App;
```

Another way of doing it, if you only return one side of the conditional rendering anyway, is using the && operator:

```
import React from 'react';

const App = () => {
  const users = [
    { name: 'Robin' },
  ];
```

```
    { name: 'Markus' },  
  ];  
  
  const showUsers = false;  
  
  return (  
    <div>  
      {showUsers && (  
        <ul>  
          {users.map(user => (  
            <li>{user.name}</li>  
          ))}  
        </ul>  
      )}  
    </div>  
  );  
};  
  
export default App;
```

I will not go into detail why this works, but if you are curious, you can learn about it and other techniques for conditional rendering over here: [All the conditional renderings in React](#). After all, the conditional rendering in React only shows again that most of React is only JavaScript in JSX and not anything React specific.



## IMPORT AND EXPORT STATEMENTS IN REACT

Fortunately, the JavaScript community settled on one way to import and export functionalities from files with JavaScript ES6 with `import` and `export` statements.

However, being new to React and JavaScript ES6, these import and export statements are just another topic which requires explanation when getting started with your first React application. Pretty early you will have your first imports for CSS, SVG or other JavaScript files. The `create-react-app` project already starts with those import statements:

```
import React from 'react';  
import logo from './logo.svg';  
import './App.css';  
  
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <h1>
```

```

    Hello React
  </h1>
  <a href="https://reactjs.org">
    Learn React
  </a>
</header>
</div>
);
}

export default App;

```

It's great for the starter project, because it offers you a well-rounded experience how other files can be imported and (exported). Also the App component gets imported in the `src/index.js` file. However, when doing your first steps in React, I try to avoid those imports in the beginning. Instead, I try to focus on JSX and React components. Only later the import and export statements are introduced when separating the first React component or JavaScript function in another file.



So how do these import and export statements work? Let's say in one file you want to export the following variables:



```

const firstname = 'Robin';
const lastname = 'Wieruch';

export { firstname, lastname };

```

Then you can import them in another file with a relative path to the first file:

```

import { firstname, lastname } from './file1.js';

console.log(firstname);
// output: Robin

```

So it's not necessarily about importing/exporting components or functions, it's about sharing everything that is assignable to a variable (leaving out CSS or SVG imports/exports, but speaking only about JS). You can also import all exported variables from another file as one object:

```

import * as person from './file1.js';

console.log(person.firstname);
// output: Robin

```

Imports can have an alias. It can happen that you import functionalities from multiple files that have the same named export. That's why you can use an alias:

```
import { firstname as username } from './file1.js';  
  
console.log(username);  
// output: Robin
```

All the previous cases are named imports and exports. But there exists the default statement too. It can be used for a few use cases:

- to export and import a single functionality
- to highlight the main functionality of the exported API of a module
- to have a fallback import functionality



```
const robin = {  
  firstname: 'Robin',  
  lastname: 'Wieruch',  
};  
  
export default robin;
```

Leave out the curly braces for the import to import the default export:

```
import developer from './file1.js';  
  
console.log(developer);  
// output: { firstname: 'Robin', lastname: 'Wieruch' }
```

Furthermore, the import name can differ from the exported default name. You can also use it in conjunction with the named export and import statements:

```
const firstname = 'Robin';  
const lastname = 'Wieruch';  
  
const person = {  
  firstname,  
  lastname,  
};  
  
export {
```

```
    firstname,  
    lastname,  
  };  
  
  export default person;
```

And import the default or the named exports in another file:

```
import developer, { firstname, lastname } from './file1.js';  
  
console.log(developer);  
// output: { firstname: 'Robin', lastname: 'Wieruch' }  
console.log(firstname, lastname);  
// output: Robin Wieruch
```

You can also spare additional lines and export the variables directly for named exports:

```
export const firstname = 'Robin';  
export const lastname = 'Wieruch';
```



These are the main functionalities for ES6 modules. They help you to organize your code, to maintain your code, and to design reusable module APIs. You can also export and import functionalities to test them.

## LIBRARIES IN REACT

React offers state management and side-effect features, but apart from this, it is only a component library which renders HTML for your browser. Everything else can be added from APIs (e.g. browser API, DOM API), JavaScript functionalities (e.g. map, filter, reduce) or external libraries. It's not always simple to choose the right library for complementing your React application, but [once you have a good overview of the different options](#), you can pick the one which fits best to your tech stack.

For instance, [fetching data in React](#) can be done with the native fetch API:

```
import React, { Component } from 'react';  
  
class App extends Component {  
  state = {  
    data: null,  
  },
```

```
};

componentDidMount() {
  fetch('https://api.mydomain.com')
    .then(response => response.json())
    .then(data => this.setState({ data }));
}

render() {
  ...
}
}

export default App;
```

But it is up to you to use another library to fetch data in React. **Axios** is one popular choice for React applications:

```
import React, { Component } from 'react';
import axios from 'axios';

class App extends Component {
  state = {
    data: null,
  };

  componentDidMount() {
    axios.get('https://api.mydomain.com')
      .then(response => this.setState({ data: response.data }));
  }

  render() {
    ...
  }
}

export default App;
```

So once you know about your problem which needs to be solved, **React's extensive and innovative ecosystem** should give you plenty of options solving it. There again it's not about React, but knowing about all the different JavaScript libraries which can be used to complement your application.

## ASYNCR/AWAIT IN REACT

In a React Function Component, **fetching data** looks slightly different with **React Hooks**:

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

const App = () => {
  const [data, setData] = React.useState(null);

  React.useEffect(() => {
    const fetchData = () => {
      axios.get('https://api.mydomain.com')
        .then(response => setData(response.data));
    };

    fetchData();
  }, []);

  return (
    ...
  );
};

export default App;
```



In the previous code snippet, we have used the most common way to resolve a promise with a then-block. The catch-block for error handling is missing for keeping the example simple. Please read one of the referenced tutorials to learn more about fetching data in React with error handling.

Anyway, you can also use async/await which got introduced to JavaScript not long ago:

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

const App = () => {
  const [data, setData] = React.useState(null);

  React.useEffect(() => {
    const fetchData = async () => {
      const response = await axios.get('https://api.mydomain.com');

      setData(response.data);
    };

    fetchData();
  }, []);

  return (
    ...
  );
};
```

```
    );  
  };  
  
  export default App;
```

In the end, async/await is just another way of resolving promises in asynchronous JavaScript.

## HIGHER-ORDER FUNCTIONS IN REACT

Higher-order functions are a great programming concept especially when moving towards functional programming. In React, it makes total sense to know about these kind of functions, because at some point you have to deal with higher-order components which can be explained best when knowing about higher-order functions in the first place.

Higher-order functions can be showcased in React early on without introducing higher-order components. For instance, let's say a rendered list of users can be filtered based on the value of an input field.



```
import React from 'react';  
  
const App = () => {  
  const users = [{ name: 'Robin' }, { name: 'Markus' }];  
  const [query, setQuery] = React.useState('');  
  
  const handleChange = event => {  
    setQuery(event.target.value);  
  };  
  
  return (  
    <div>  
      <ul>  
        {users  
          .filter(user => user.name.includes(query))  
          .map(user => (  
            <li>{user.name}</li>  
          ))}  
      </ul>  
  
      <input type="text" onChange={handleChange} />  
    </div>  
  );  
};  
  
export default App;
```

It's not always desired to extract functions, because it can add unnecessary complexity, but on the other side, it can have beneficial learning effects for JavaScript. In addition, by extracting a function you make it testable in isolation from the React component. So let's showcase it with the function which is provided to the built-in filter function.



```
import React from 'react';

function doFilter(user) {
  return user.name.includes(query);
}

const App = () => {
  const users = [{ name: 'Robin' }, { name: 'Markus' }];
  const [query, setQuery] = React.useState('');

  const handleChange = event => {
    setQuery(event.target.value);
  };

  return (
    <div>
      <ul>
        {users.filter(doFilter).map(user => (
          <li>{user.name}</li>
        )))}
      </ul>

      <input type="text" onChange={handleChange} />
    </div>
  );
};

export default App;
```

The previous implementation doesn't work because the `doFilter()` function needs to know about the query property from the state. So you can pass it to the function by wrapping it with another function which leads to a higher-order function.

```
import React from 'react';

function doFilter(query) {
  return function(user) {
    return user.name.includes(query);
  };
}

const App = () => {
  const users = [{ name: 'Robin' }, { name: 'Markus' }];
```

```
const [query, setQuery] = React.useState('');

const handleChange = event => {
  setQuery(event.target.value);
};

return (
  <div>
    <ul>
      {users.filter(doFilter(query)).map(user => (
        <li>{user.name}</li>
      ))}
    </ul>

    <input type="text" onChange={handleChange} />
  </div>
);
};

export default App;
```



Basically a higher-order function is a function which returns a function. By using JavaScript ES6 arrow functions, you can make a higher-order function more concise. Furthermore, this shorthand version makes it more attractive composing functions into functions.



```
const doFilter = query => user =>
  user.name.includes(query);
```

Now, the `doFilter()` function can be exported from the file and tested in isolation as pure (higher-order) function. After learning about higher-order functions, all the fundamental knowledge is established to learn more about [React's higher-order components](#), if you want to learn about this advanced technique in React. Moving functions around your code base is a great way to learn about the benefits of having functions as first class citizens in JavaScript. It's powerful when moving your code towards functional programming.

---

## SHORTHAND OBJECT ASSIGNMENT

There is one little addition in the JavaScript language which leaves beginners confused. In JavaScript ES6, you can use a shorthand property syntax to initialize your objects more concisely, like following object initialization:

```
const name = 'Robin';
```

```
const user = {  
  name: name,  
};
```

When the property name in your object is the same as your variable name, you can do the following:

```
const name = 'Robin';  
  
const user = {  
  name,  
};
```

Shorthand method names are also useful. In JavaScript ES6, you can initialize methods in an object more concisely:



```
// without shorthand method names  
var userService = {  
  getUsername: function (user) {  
    return user.firstname + ' ' + user.lastname;  
  },  
};  
  
// shorthand method names  
const userService = {  
  getUsername(user) {  
    return user.firstname + ' ' + user.lastname;  
  },  
};
```

Finally, you are allowed to use computed property names in JavaScript ES6:

```
// normal usage of key property in an object  
var user = {  
  name: 'Robin',  
};  
  
// computed key property for dynamic naming  
const key = 'name';  
const user = {  
  [key]: 'Robin',  
};
```

You are able to use computed property names to allocate values by key in an object dynamically, a handy way to generate lookup tables (also called dictionaries) in JavaScript.

## DESTRUCTURING IN REACT

Another language feature introduced in JavaScript is called **destructuring**. It's often the case that you have to access plenty of properties from your state or props in your component. Rather than assigning them to a variable one by one, you can use destructuring assignment in JavaScript.

```
const state = { counter: 1, list: ['a', 'b'] };

// no object destructuring
const list = state.list;
const counter = state.counter;

// object destructuring
const { list, counter } = state;
```



That's especially beneficial for React's Function Components, because they always receive the props object in their function signature. Often you will not use the props but only its content, so you can destructure the content in the function signature.

```
// no destructuring
function Greeting(props) {
  return <h1>{props.greeting}</h1>;
}

// destructuring
function Greeting({ greeting }) {
  return <h1>{greeting}</h1>;
}
```

The destructuring works for JavaScript arrays too:

```
const list = ['a', 'b'];

// no array destructuring
const itemOne = list[0];
const itemTwo = list[1];

// array destructuring
```

```
const [itemOne, itemTwo] = list;
```

As you have already seen, React Hooks are using the **array destructuring** to access state and state updater function.

```
import React from 'react';

const Counter = () => {
  const [count, setCount] = React.useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
};

export default Counter;
```



Another great feature is the **rest destructuring**. It is often used for splitting out a part of an object, but keeping the remaining properties in another object.



```
const state = { counter: 1, list: ['a', 'b'] };

// rest destructuring
const { list, ...rest } = state;

console.log(rest);
// output: { counter: 1 }
console.log(list);
// output: ['a', 'b']
```

Afterward, the list can be used to be rendered, for instance in a React component, whereas the remaining state (here counter) is used somewhere else. That's where the **JavaScript spread operator** comes into play to forward the rest object to the next component. In the next section, you will see this operator in action.

---

## SPREAD OPERATOR IN REACT

The spread operator comes with three ..., but shouldn't be mistaken for the rest operator. It depends on the context where it is used. Used within a destructuring (see above), it is as rest operator. Used somewhere else it is a spread operator.

```
const userCredentials = { firstname: 'Robin' };
const userDetails = { nationality: 'German' };

const user = {
  ...userCredentials,
  ...userDetails,
};

console.log(user);
// output: { firstname: 'Robin', nationality: 'German' }
```

The spread operator literally spreads all the key value pairs of an object. In React, it comes in handy when props are just being passed down to the next component.



```
import React from 'react';

const App = () => {
  const users = [
    { name: 'Robin', nationality: 'German' },
    { name: 'Markus', nationality: 'American' },
  ];

  return (
    <ul>
      {users.map(user => <li>
        <User
          name={user.name}
          nationality={user.nationality}
        />
      </li>)}
    </ul>
  );
};

const User = ({ name, nationality }) =>
  <span>{name} from {nationality}</span>;

export default App;
```

Rather than passing all properties of an object property by property, you can use the spread operator to pass all key value pairs to the next component.

```
import React from 'react';

const App = () => {
  const users = [
    { name: 'Robin', nationality: 'German' },
    { name: 'Markus', nationality: 'American' },
  ];

  return (
    <ul>
      {users.map(user => <li>
        <User {...user} />
      </li>)}
    </ul>
  );
};

const User = ({ name, nationality }) =>
  <span>{name} from {nationality}</span>;

export default App;
```



Also you don't need to worry about the object's structure beforehand, because the operator simply passes *everything* to the next component.



## THERE IS MORE JAVASCRIPT THAN REACT

In conclusion, there is lots of JavaScript which can be harnessed in React. Whereas React has only a slim API surface area, developers have to get used to all the functionalities JavaScript has to offer. The saying is not without any reason: *"being a React developer makes you a better JavaScript developer"*. Let's recap some of the learned aspects of JavaScript in React by refactoring a higher-order component.

```
function withLoading(Component) {
  return class WithLoading extends React.Component {
    render() {
      const { isLoading, ...rest } = this.props;

      if (isLoading) {
        return <p>Loading</p>;
      }

      return <Component { ...rest } />;
    }
  }
}
```

}

This higher-order component is only used for showing a conditional loading indicator when the `isLoading` prop is set to true. Otherwise it renders the input component. You can already see the (rest) destructuring from the props and the spread operator in for the next Component. The latter can be seen for the rendered Component, because all the remaining properties from the props object are passed to the Component as key value pairs.

The first step for making the higher-order component more concise is refactoring the returned React Class Component to a Function Component:

```
function withLoading(Component) {  
  return function ({ isLoading, ...rest }) {  
    if (isLoading) {  
      return <p>Loading</p>;  
    }  
  
    return <Component { ...rest } />;  
  };  
}
```

f



in

You can see that the rest destructuring can be used in the function's signature too. Next, using JavaScript ES6 arrow functions makes the higher-order component more concise again:

```
const withLoading = Component => ({ isLoading, ...rest }) => {  
  if (isLoading) {  
    return <p>Loading</p>;  
  }  
  
  return <Component { ...rest } />;  
}
```

And adding the ternary operator shortens the function body into one line of code. Thus the function body can be left out and the return statement can be omitted.

```
const withLoading = Component => ({ isLoading, ...rest }) =>  
  isLoading  
    ? <p>Loading</p>  
    : <Component { ...rest } />
```

As you can see, the higher-order component uses various JavaScript and not React relevant techniques: arrow functions, higher-order functions, a ternary operator, destructuring and the

spread operator. Basically that's how JavaScript's functionalities can be used in React applications in a nutshell.

. . .

Often people say that learning React has a steep learning curve. But it hasn't when only leaving React in the equation and leaving all the JavaScript out of it. React doesn't add any foreign abstraction layer on top as other web frameworks are doing it. Instead you have to use JavaScript. So hone your JavaScript skills and you will become a great React developer.

---

Show Comments

---

KEEP READING ABOUT [REACT](#)➤

## HOW TO MANAGE REACT STATE WITH ARRAYS

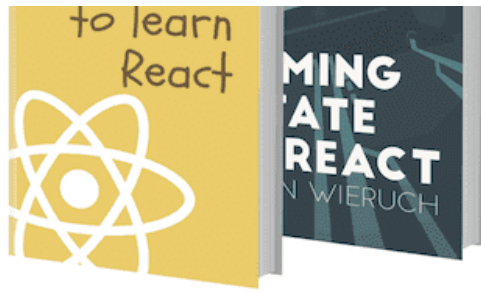


After learning how to pass props in React , the next thing you will learn is often state in React . Managing JavaScript primitives such as strings, booleans, and integers in React State are the...

## HOW TO USE CONTEXT IN REACT?

React's Function Components come with React Hooks these days. Not only can React Hooks be used for State in React but also for using React's Context in a more convenient way. This tutorial shows...





## THE ROAD TO REACT

Learn React by building real world applications. No setup configuration. No tooling. Plain React in 200+ pages of learning material. Learn React like **50.000+ readers.**

GET THE BOOK >

Get it on Amazon.



## TAKE PART

NEVER MISS AN ARTICLE ABOUT WEB DEVELOPMENT AND JAVASCRIPT.

- ✓ Join 50.000+ Developers
- ✓ Learn Web Development with JavaScript
- ✓ Tips and Tricks
- ✓ Access Tutorials, eBooks and Courses
- ✓ Personal Development as a Software Engineer

[SUBSCRIBE](#)[View our Privacy Policy.](#)

---

## PORTFOLIO

[Online Courses](#)[Open Source](#)[Tutorials](#)

## ABOUT

[About me](#)[What I use](#)[How to work with me](#)[How to support me](#)

---

© Robin Wieruch



[Contact Me](#) [Privacy & Terms](#)