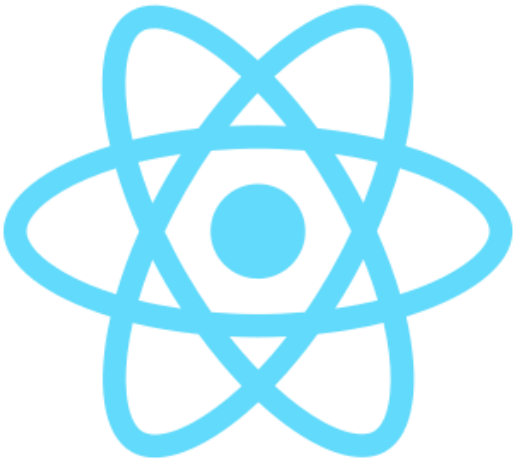
All the fundamental React.js concepts, jammed into this single Medium article

Samer Buna Follow
Aug 18, 2017 · 15 min read



> ***Update:*** *This article is now part of my book "React.js Beyond The Basics".*

> *Read the updated version of this content and more about React at* **jscomplete.com/react-beyond-basics**.

. . .

This article is not going to cover what React is or why you should learn it. Instead, this is a practical introduction to the fundamentals of React.js for those who are already familiar with JavaScript and know the basics of the DOM API.

All code examples below are labeled for reference. They are purely intended to provide examples of concepts. Most of them can be written in a much better way.

Fundamental #1: React is all about components

React is designed around the concept of reusable components. You define small components and you put them together to form bigger components.

All components small or big are reusable, even across different projects.

A React component—in its simplest form—is a plain-old JavaScript function:

```
// Example 1
// https://jscomplete.com/repl?j=Sy3QAdKHW

function Button (props) {
  // Returns a DOM element here. For example:
  return <button type="submit">{props.label}</button>;
}

// To render the Button component to the browser
ReactDOM.render(<Button label="Save" />, mountNode)
```

The curly braces used for the button label are explained below. Don't worry about them now. `ReactDOM` will also be explained later, but if you want to test this example and all upcoming code examples, the above `render` function is what you need.

The second argument to `ReactDOM.render` is the destination DOM element which React is going to take over and control. In the jsComplete React Playground, you can just use the special variable `mountNode`.

Note the following about Example 1:

- The component name starts with a capital letter. This is required since we will be dealing with a mix of HTML elements and React elements. Lowercase names are reserved for HTML elements. In fact, go ahead and try to name the React component just "button" and see how ReactDOM will ignore the function and renders a regular empty HTML button.
- Every component receives a list of attributes, just like HTML elements. In React, this list is called **props**. With a function component, you can name it anything though.
- We weirdly wrote what looks like HTML in the returned output of the `Button` function component above. This is neither JavaScript nor HTML, and it's not even React.js. But, it's so popular that it became the default in React applications. It's called *JSX* and it's a JavaScript extension. JSX is also a **compromise**! Go ahead and try and return any other HTML element inside the function above and see how they are all supported (for example, return a text input element).

Fundamental #2: What the flux is JSX?

Example 1 above can be written in pure React.js without JSX as follows:

```
// Example 2 -  React component without JSX
// https://jscomplete.com/repl?j=HyiEwoYB-
```

```
function Button (props) {
  return React.createElement(
    "button",
    { type: "submit" },
    props.label
  );
}
```

```
// To use Button, you would do something like
ReactDOM.render(
  React.createElement(Button, { label: "Save" }),
  mountNode
);
```

The `createElement` function is the main function in the React top-level API. It's 1 of a total of 8 things in that level that you need to learn. That's how small the React API is.

Much like the DOM itself having a `document.createElement` function to create an element specified by a tag name, React's `createElement` function is a higher-level function that can do what `document.createElement` does, but it can also be used to create an element to represent a React component. We did the latter when we used the `Button` component in Example 2 above.

Unlike `document.createElement`, React's `createElement` accepts a dynamic number of arguments after the second one to represent the **children** of the created element. So `createElement` actually creates a **tree**.

Here's an example of that:

```
// Example 3 -  React's createElement API
// https://jscomplete.com/repl?j=r1GNoiFBb

const InputForm = React.createElement(
  "form",
  { target: "_blank", action: "https://google.com/search" },
  React.createElement("div", null, "Enter input and click
Search"),
  React.createElement("input", { name: "q", className:
"input" }),
  React.createElement(Button, { label: "Search" })
);

// InputForm uses the Button component, so we need that too:
function Button (props) {
  return React.createElement(
    "button",
    { type: "submit" },
    props.label
  );
}

// Then we can use InputForm directly with .render
ReactDOM.render(InputForm, mountNode);
```

Note a few things about the example above:

- `InputForm` is not a React component; it's just a React **element**. This is why we used it directly in the `ReactDOM.render` call and not with `<InputForm />`.
- The `React.createElement` function accepted multiple arguments after the first two. Its list of arguments starting from the 3rd one comprises the list of children for the created element.
- We were able to nest `React.createElement` calls because it's all JavaScript.
- The second argument to `React.createElement` can be null or an empty object when no attributes or props are needed for the element.
- We can mix HTML element with React elements.
- React's API tries to be as close to the DOM API as possible, that's why we use `className` instead of `class` for the input element. Secretly, we all wish the React's API would become part of the DOM API itself. Because, you know, it's much much better.

The code above is what the browser understands when you include the React library. The browser does not deal with any JSX business. However, we humans like to see and work with HTML instead of these `createElement` calls (imagine building a website with just `document.createElement`, which you can!). This is why the JSX compromise exists. Instead of writing the form above with `React.createElement` calls, we can write it with a syntax very similar to HTML:

```
// Example 4 - JSX (compare with Example 3)
// https://jscomplete.com/repl?j=SJWy3otHW

const InputForm =
  <form target="_blank" action="https://google.com/search">
    <div>Enter input and click Search</div>
    <input name="q" className="input" />
    <Button label="Search" />
  </form>;

// InputForm "still" uses the Button component, so we need
that too.
// Either JSX or normal form would do
function Button (props) {
```

```
  // Returns a DOM element here. For example:
  return <button type="submit">{props.label}</button>;
}


// Then we can use InputForm directly with .render
ReactDOM.render(InputForm, mountNode);
```

Note a few things about the above:

- It's not HTML. For example, we're still doing `className` instead of `class`.
- We're still considering what looks like HTML above as JavaScript. See how I added a semicolon at the end.

What we wrote above (Example 4) is JSX. Yet, what we took to the browser is the compiled version of it (Example 3). To make that happen, we need to use a pre-processor to convert the JSX version into the `React.createElement` version.

That is JSX. It's a compromise that allows us to write our React components in a syntax similar to HTML, which is a pretty good deal.

> *The word "Flux" in the header above was chosen to rhyme, but it's also the name of a very popular application architecture popularized by Facebook. The most famous implementation of which is Redux. Flux fits the React reactive pattern perfectly.*

JSX, by the way, can be used on its own. It's not a React-only thing.

Fundamental #3: You can use JavaScript expressions anywhere in JSX

Inside a JSX section, you can use any JavaScript expression within a pair of curly braces.

```
// Example 5 -  Using JavaScript expressions in JSX
// https://jscomplete.com/repl?j=SkNN3oYSW

const RandomValue = () =>
  <div>
```

```
    { Math.floor(Math.random() * 100) }
  </div>;

// To use it:
ReactDOM.render(<RandomValue />, mountNode);
```

Any JavaScript expression can go inside those curly braces. This is equivalent to the `${}` interpolation syntax in JavaScript template literals.

This is the only constraint inside JSX: only expressions. So, for example, you can't use a regular `if` statement, but a ternary expression is ok.

JavaScript variables are also expressions, so when the component receives a list of props (the `RandomValue` component didn't, `props` are optional), you can use these props inside curly braces. We did this in the `Button` component above (Example 1).

JavaScript objects are also expressions. Sometimes we use a JavaScript object inside curly braces, which makes it look like double curly braces, but it's really just an object inside curly braces. One use case of that is to pass a CSS style object to the special `style` attribute in React:

```
// Example 6 - An object passed to the special React style
prop
// https://jscomplete.com/repl?j=S1Kw2sFHb

const ErrorDisplay = ({message}) =>
  <div style={ { color: 'red', backgroundColor: 'yellow' }
}>
    {message}
  </div>;

// Use it:
ReactDOM.render(
  <ErrorDisplay
    message="These aren't the droids you're looking for"
  />,
  mountNode
);
```

Note how I **destructured** only the message out of the props argument. Also note how the `style` attribute above is a special one (again, it's not HTML, it's closer to the DOM API). We use an object as the value of the `style` attribute. That object defines the styles as if we're doing so with JavaScript (because we are).

You can even use a React element inside JSX, because that too is an expression. Remember, a React element is essentially a function call:

```
// Example 7 - Using a React element within {}
// https://jscomplete.com/repl?j=SkTLpjYr-
```

```
const MaybeError = ({errorMessage}) =>
  <div>
    {errorMessage && <ErrorDisplay message={errorMessage}
/>}
  </div>;

// The MaybeError component uses the ErrorDisplay component:
const ErrorDisplay = ({message}) =>
  <div style={ { color: 'red', backgroundColor: 'yellow' }
}>
    {message}
  </div>;
```

```
// Now we can use the MaybeError component:
ReactDOM.render(
  <MaybeError
    errorMessage={Math.random() > 0.5 ? 'Not good' : ''}
  />,
  mountNode
);
```

The `MaybeError` component above would only display the `ErrorDisplay` component if there is an `errorMessage` string passed to it and an empty `div`. React considers `{true}`, `{false}`, `{undefined}`, and `{null}` to be valid element children, which do not render anything.

You can also use all of JavaScript functional methods on collections (`map`, `reduce`, `filter`, `concat`, and so on) inside JSX. Again, because they return expressions:

```
// Example 8 - Using an array map inside {}
// https://jscomplete.com/repl?j=SJ29aiYH-

const Doubler = ({value=[1, 2, 3]}) =>
  <div>
    {value.map(e => e * 2)}
  </div>;

// Use it
ReactDOM.render(<Doubler />, mountNode);
```

Note how I gave the `value` prop a default value above, because it's all just Javascript. Note also that I outputted an array expression inside the `div`. React is okay with that; It will place every doubled value in a text node.

Fundamental #4: You can write React components with JavaScript classes

Simple function components are great for simple needs, but sometimes we need more. React supports creating components through the JavaScript class syntax as well. Here's the `Button` component (in Example 1) written with the class syntax:

```
// Example 9 - Creating components using JavaScript classes
// https://jscomplete.com/repl?j=ryjk0iKHb

class Button extends React.Component {
  render() {
    return <button>{this.props.label}</button>;
  }
}

// Use it (same syntax)
ReactDOM.render(<Button label="Save" />, mountNode);
```

The class syntax is simple. Define a class that extends `React.Component` (another top-level React API thing that you need to learn). The class defines a single instance function `render()`, and that render function returns the virtual DOM element. Every time we use the `Button` class-based component above (for example, by doing `<Button ... />`),

React will instantiate an object from this class-based component and use that object to render a DOM element in the DOM tree.

This is the reason why we used `this.props.label` inside the JSX in the rendered output above. Because every element rendered through a class component gets a special **instance** property called `props` that holds all values passed to that element when it was created.

Since we have an instance associated with a single use of the component, we can customize that instance as we wish. We can, for example, customize it after it gets constructed by using the regular JavaScript `constructor` function:

```
// Example 10 -  Customizing a component instance
// https://jscomplete.com/repl?j=rko7RsKS-

class Button extends React.Component {
  constructor(props) {
    super(props);
    this.id = Date.now();
  }
  render() {
    return <button id={this.id}>{this.props.label}</button>;
  }
}

// Use it
ReactDOM.render(<Button label="Save" />, mountNode);
```

We can also define class functions and use them anywhere we wish, including inside the returned JSX output:

```
// Example 11 - Using class properties
// https://jscomplete.com/repl?j=H1YDCoFSb

class Button extends React.Component {
  clickCounter = 0;

  handleClick = () => {
    console.log(`Clicked: ${++this.clickCounter}`);
  };

  render() {
    return (
```

```
        <button id={this.id} onClick={this.handleClick}>
          {this.props.label}
        </button>
      );
    }
  }

// Use it
ReactDOM.render(<Button label="Save" />, mountNode);
```

Note a few things about Example 11 above:

- The `handleClick` function is written using the new proposed class-field syntax in JavaScript. This is still at stage-2, but for many reasons it's the best option to access the component mounted instance (thanks to arrow functions). But, you need to use a compiler like Babel configured to understand stage-2 (or the class-field syntax) to get the code above to work. The jsComplete REPL has that pre-configured.
- We've also defined the `clickCounter` instance variables using the same class-field syntax. This allows us to skip using a class constructor call altogether.
- When we specified the `handleClick` function as the value of the special `onClick` React attribute, we did not call it. We passed in the **reference** to the `handleClick` function. Calling the function on that level is one of the most common mistakes when working with React.

```
// Wrong:
onClick={this.handleClick()}

// Right:
onClick={this.handleClick}
```

Fundamental #5: Events in React: Two Important Differences

When handling events inside React elements, there are two very important differences from the way we do so with the DOM API:

- All React elements attributes (events included) are named using **camelCase**, rather than **lowercase**. It's `onClick`, not `onclick`.
- We pass an actual JavaScript function reference as the event handler, rather than a string. It's `onClick={handleClick}`, not `onClick="handleClick"`.

React wraps the DOM event object with an object of its own to optimize the performance of events handling. But inside an event handler, we can still access all methods available on the DOM event object. React passes that wrapped event object to every handle call. For example, to prevent a form from the default submission action, you can do:

```
// Example 12 - Working with wrapped events
// https://jscomplete.com/repl?j=HkIhRoKBb

class Form extends React.Component {
  handleSubmit = (event) => {
    event.preventDefault();
    console.log('Form submitted');
  };

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <button type="submit">Submit</button>
      </form>
    );
  }
}

// Use it
ReactDOM.render(<Form />, mountNode);
```

Fundamental #6: Every React component has a story

The following applies to the class component only (those that extend `React.Component`). Function components have a slightly different story.

1. First, we define a template for React to create elements from the component.

2. Then, we instruct React to use it somewhere. For example, inside a `render` call of another component, or with `ReactDOM.render`.
3. Then, React instantiates an element and gives it a set of **props** that we can access with `this.props`. Those props are exactly what we passed in step 2 above.
4. Since it's all JavaScript, the `constructor` method will be called (if defined). This is the first of what we call: **component lifecycle methods**.
5. React then computes the output of the render method (the virtual DOM node).
6. Since this is the first time React is rendering the element, React will communicate with the browser (on our behalf, using the DOM API) to display the element there. This process is commonly known as **mounting**.
7. React then invokes another lifecycle method, called `componentDidMount`. We can use this method to, for example, do something on the DOM that we now know exists in the browser. Prior to this lifecycle method, the DOM we work with was all virtual.
8. Some components stories end here. Other components get unmounted from the browser DOM for various reasons. Right before the latter happens, React invokes another lifecycle method, `componentWillUnmount`.
9. The **state** of any mounted element might change. The parent of that element might re-render. In either case, the mounted element might receive a different set of props. React magic happens here and we actually start **needing** React at this point! Prior to this point, we did not need React at all, honestly.

The story of this component continues, but before it does, we need to understand this **state** thing that I speak of.

Fundamental #7: React components can have a private state

The following is also only applicable to class components. Did I mention that some people call presentational-only components **dumb**?

The `state` property is a special one in any React class component. React monitors every component state for changes. But for React to do so efficiently, we have to change the state field through another React API thing that we need to learn, `this.setState`:

```
// Example 13 -  the setState API
// https://jscomplete.com/repl?j=H1fek2KH-

class CounterButton extends React.Component {
  state = {
    clickCounter: 0,
    currentTimestamp: new Date(),
  };

  handleClick = () => {
    this.setState((prevState) => {
     return { clickCounter: prevState.clickCounter + 1 };
    });
  };

  componentDidMount() {
   setInterval(() => {
     this.setState({ currentTimestamp: new Date() })
    }, 1000);
  }

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Click</button>
        <p>Clicked: {this.state.clickCounter}</p>
        <p>Time:
{this.state.currentTimestamp.toLocaleString()}</p>
      </div>
    );
  }
}

// Use it
ReactDOM.render(<CounterButton />, mountNode);
```

This is the most important example to understand. It will basically complete your fundamental knowledge of the React way. After this example, there are a few other small things that you need to learn, but it's mostly you and your JavaScript skills from that point.

Let's walk through Example 13, starting with class fields. It has two of them. The special `state` field is initialized with an object that holds a `clickCounter` that starts with `0`, and a `currentTimestamp` that starts with `new Date()`.

The second class field is a `handleClick` function, which we passed to the `onClick` event for the button element inside the render method. The `handleClick` method modifies this component instance state using `setState`. Take notice of that.

The other place we're modifying the state is inside an interval timer that we started inside the `componentDidMount` lifecycle method. It ticks every second and executes another call to `this.setState`.

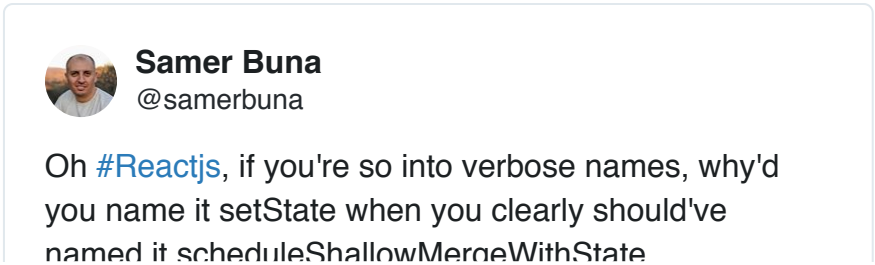
In the render method, we used the two properties we have on the state with a normal read syntax. There is no special API for that.

Now, notice that we updated the state using two different ways:

1. By passing a function that returned an object. We did that inside the `handleClick` function.
2. By passing a regular object. We did that inside the interval callback.

Both ways are acceptable, but the first one is preferred when you read and write to the state at the same time (which we do). Inside the interval callback, we're only writing to the state and not reading it. When in doubt, always use the first function-as-argument syntax. It's safer with race conditions because `setState` should always be treated as an asynchronous method.

How do we update the state? We return an object with the new value of what we want to update. Notice how in both calls to `setState`, we're only passing one property from the state field and not both. This is completely okay because `setState` actually **merges** what you pass it (the returned value of the function argument) with the existing state. So, not specifying a property while calling `setState` means that we wish to not change that property (but not delete it).



Fundamental #8: React will react

React gets its name from the fact that it **reacts** to state changes (although not reactively, but on a schedule). There was a joke that React should have been named **Schedule**!

However, what we witness with the naked eye when the state of any component gets updated is that React reacts to that update and automatically reflects the update in the browser DOM (if needed).

Think of the render function's input as both:

- The props that get passed by the parent
- The internal private state that can be updated anytime

When the input of the render function changes, its output might change.

React keeps a record of the history of renders and when it sees that one render is different than the previous one, it'll compute the difference between them and efficiently translate it into actual DOM operations that get executed in the DOM.

Fundamental #9: React is your agent

You can think of React as the agent we hired to communicate with the browser. Take the current timestamp display above as an example. Instead of us manually going to the browser and invoking DOM API operations to find and update the `p#timestamp` element every second, we just changed a property on the state of the component and React did its job of communicating with the browser on our behalf. I believe this is the true reason why React is popular. We hate talking to Mr.

Browser (and the so many dialects of the DOM language that it speaks) and React volunteered to do all the talking for us, for free.

Fundamental #10: Every React component has a story (part 2)

Now that we know about the state of a component and how when that state changes some magic happens, let's learn the last few concepts about that process.

1. A component might need to re-render when its state gets updated or when its parent decides to change the props that it passed to the component
2. If the latter happens, React invokes another lifecycle method, `componentWillReceiveProps`.
3. If either the state object or the passed-in props are changed, React has an important decision to do. Should the component be updated in the DOM? This is why it invokes another important lifecycle method here, `shouldComponentUpdate`. This method is an actual question, so if you need to customize or optimize the render process on your own, you have to answer that question by returning **either** true or false.
4. If there is no custom `shouldComponentUpdate` specified, React defaults to a very smart thing that's actually good enough in most situations.
5. First, React invokes another lifecycle method at this point, `componentWillUpdate`. React will then compute the new rendered output and compare it with the last rendered output.
6. If the rendered output is exactly the same, React does nothing (no need to talk to Mr. Browser).
7. If there is a difference, React takes that difference to the browser, as we've seen before.
8. In any case, since an update process happened anyway (even if the output was exactly the same), React invokes the final lifecycle method, `componentDidUpdate`.

Lifecycle methods are actually escape hatches. If you're not doing anything special, you can create full applications without them. They're very handy for analyzing what is going on in the application and for further optimizing the performance of React updates.

That's it. Believe it or not, with what you learned above (or parts of it, really), you can start creating some interesting React applications. If you're hungry for more, check out my Learn React.js by Building Games book!

Thanks to the many readers who reviewed and improved this article, Łukasz Szewczak, Tim Broyles, Kyle Holden, Robert Axelse, Bruce Lane, Irvin Waldman, and Amie Wilt.

. . .

Learning React or Node? Checkout my books:

- Learn React.js by Building Games
- Node.js Beyond the Basics