

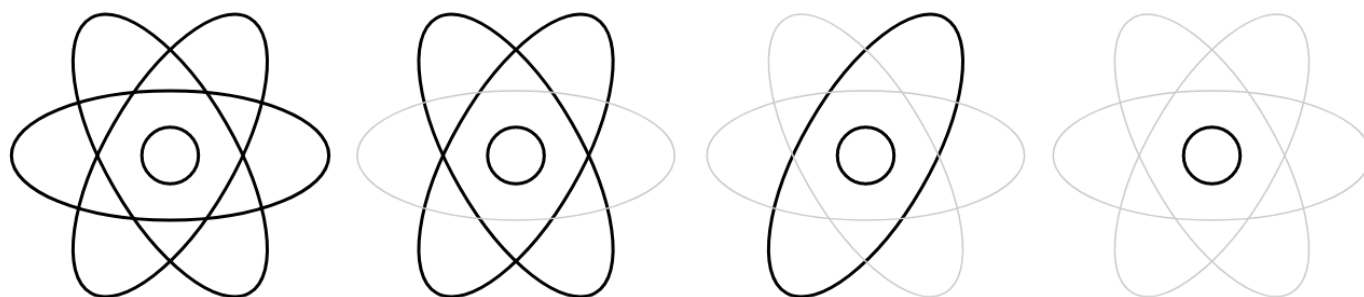
React: Under the Hood



Clayton Marshall

Follow

May 30, 2020 · 10 min read



This post is based off of a talk I gave during college. I thought it covered a lot of good material, but it wasn't totally smoothed out. So, this is my attempt to fix some of those road bumps, and condense it into a single article.

All source code from this post can be found [here](#).

Prerequisites

To get the most out of this post, you should have a solid understanding of basic web development, Javascript, and the React library.

Introduction

When I first learned React I felt like so much of what was happening was “magic”. A lot of under-the-hood work was taking place just to write a simple UI. Although I think abstraction is great, it's important that as developers, we know how our tools work. In this post, we'll be learning about the “magic” of React, by building it.

What are we building?

React as a library offers a ton of functionality. To build the entire library would take a lot of time, and honestly, expertise I don't have. This post will focus on the core bits of React. We won't be building efficient state management, component life-cycle methods, or the context API. Instead, we'll be building the two parts of React that all those extra things were created to support.

1. A way to declare UI elements

2. A way to render those elements to a webpage

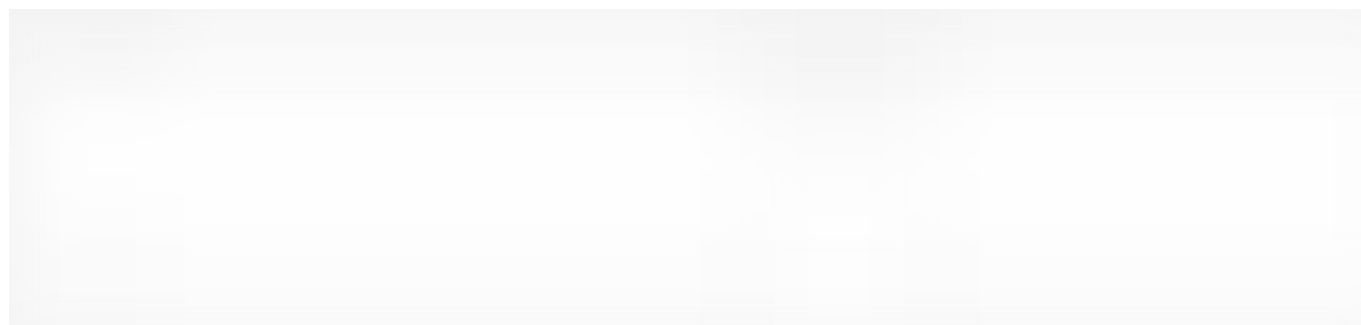
Demystifying JSX

To start, let's focus on creating UI elements. Before we do this though, we need to understand what that HTML syntax really is that we've been using inside our React apps. I'm gonna rip the band-aid off quick — it's not HTML. It's actually a syntax called JSX, and it's totally separate from React. A big misconception to newer users of React is that JSX is a part of the React library, but this isn't the case. You can actually use everything React has to offer without ever typing a line of JSX.

So why do we use it? Two reasons, convenience and familiarity. The JSX syntax compiles down to plain Javascript code (we'll see examples soon) that make references to React. Using JSX instead of directly invoking the React library makes our code more readable, and saves a lot of redundant typing. Plus, the syntax is familiar to web developers of any background, which really helped boost React's adoption.

To get a better look at how all this works, we need to get our hands on a JSX compiler. The most popular JSX compiler you'll find is called Babel. To get a quick look at how it works, you can try out the REPL on Babel's site [here](#).

When opening that link, you should see a page with two text boxes. The box on the left you can type JSX code and on the right side you'll see the compiled Javascript code.



With this tool, we can see what JSX code becomes after it's compiled. Each tag is translated into a function invocation of `React.createElement`. Taking a closer look we can see that the tag name, props, and all children of our component are being passed as arguments to the `React.createElement`.

Very soon, we'll implement that `createElement` function, but before we do that, we need to create a new project. Our first task will be setting up JSX support inside a project on our local machine, and then we'll focus on building our "React-like"

library. If you're not interested in building the library alongside the article feel free to skip to the **Building createElement** section.

. . .

Project Setup

Let's do some setup and start building! First, you will need to have node and npm installed on your machine. I've built this demo on node v13 and npm v6. These exact versions likely aren't necessary, but if you want to avoid as many issues as possible, I'd recommend using them.

Side note: If you have any trouble with the project setup instructions below, remember you can always just clone the code [here](#)!

```
$ mkdir react
```

```
$ cd react && npm init
```

```
$ touch src/index.js
```

```
$ npm install --save-dev @babel/core @babel/cli @babel/plugin-transform-react-  
jsx
```

Create a `.babelrc` file in your project's root directory, and add the following:

Babel is installed and configured for our project, now let's use it! To do this, we'll add a build script that will compile all of our JSX files with Babel in one easy command. Open your `package.json` file, and add the following to `scripts`:

```
"build": "babel src -d lib"
```

By adding this build script to your `package.json` we will be able to quickly run our `src` folder files through the babel compiler and see the output in a `lib` folder. Let's test it out!

Add some JSX code into `src/index.js`

To compile this JSX we need to use the build command we just added to our `package.json` file. Inside the root of your project run:

```
$ npm run build
```

Once the build script completes, you should see a `lib` folder created in your project's root directory. Open it to find the compiled version of our `src` folder files.

Quick fix-up:

The JSX tags are currently being compiled into a function call to `React.createElement`. To avoid confusion about using the React library (which we aren't), we're going to call our function `createElement`. We'll need to add something called a pragma spec to our JSX babel plugin. Update your `.babelrc` to look like this:

If you run the `npm run build` command again you should see the `lib` folder has been updated, and no longer contains references to `React.createElement`. Instead you'll just see calls to `createElement`.

Webpage setup (using our lib/index.js file)

One last setup step! We've successfully added support to compile JSX code into Javascript, but we still aren't using that Javascript anywhere. Like every React project, we need a root HTML document that we can render our UI onto. At the end of the day, React applications still boil down to web dev 101 — we create an `index.html` file and we load some Javascript onto the page.

Inside the root directory of your project create an `index.html` file, and inside, add this boilerplate:

The only thing to take note of here is that we're loading our `lib/index.js` file. Once we have a project built, this `lib/index.js` file will be the starting point of our React app.

Putting it all together

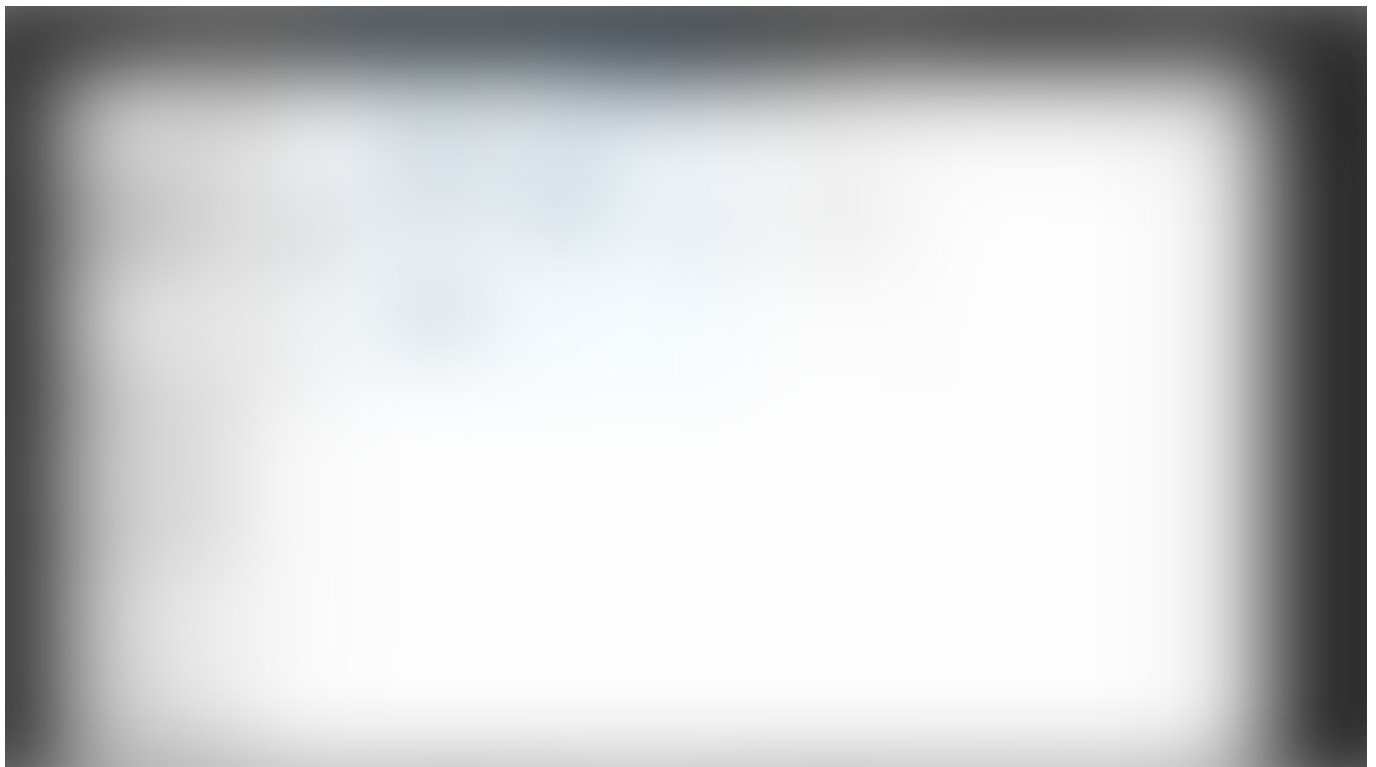
In an attempt to put any lasting confusion to rest, I want to go through an example workflow using all the tools and files we've tinkered with thus far. For starters, let's add a log statement into our `src/index.js` for testing purposes. If you still have the JSX code in there from earlier that's fine feel free to leave it.

Somewhere inside `src/index.js`:

Next, we need to run our build command. Since that `console.log` statement isn't JSX, we technically don't need to compile the file, but again, this is a workflow example, so just pretend.

Run: `$ npm run build`

If you take a look at `lib/index.js` you should see our log statement. The final step is to ensure `lib/index.js` is being loaded inside our HTML document. To test this, just open the `index.html` file in the browser, and check the developer console for the log output.



If you're seeing the log statement inside the console, your setup is complete 🎉

• • •

Building createElement

With our project ready to go, we can move on to building the React functionality. If you remember, we only have two major steps to take care of. First, we'll implement a function that creates an element (`createElement`) — mimicking `React.createElement`. Then, we'll implement a `render` function that takes our UI elements and writes them to our webpage in a similar fashion to `ReactDOM.render`.

Before we implement our `createElement` function, let's dissect its usage once more.

We see the first argument to `createElement` is the name of an HTML tag represented as a string, or in the case of using our own self-written component, a function. The second argument is an object of the props passed to the element. And all remaining arguments are the children of the UI element.

With this info we can start scaffolding out what our `createElement` function might look like.

What is createElement actually creating?

A function definition is only half the battle. We need to know what `createElement` actually does. All we really want in return from calling `createElement` is an object that contains specifications on what our UI element should look like. You may have heard this referred to as a virtual DOM by React documentation.

Unfamiliar with the DOM?

The DOM or document object model is a Javascript object that is used to represent the current webpage. It's the object your browser uses to properly render the page, and you can access it using the global `document` variable.

So, with some background knowledge on what the DOM is, the goal of `createElement` should be a bit more clear. `createElement` is going to be passed in a bunch of information about our UI element (tag name, properties, etc), then it will return to us the same information in object form (a virtual DOM). Once we have our virtual DOM created, we'll use our `render` function to update our browser's DOM to look like our virtual DOM.

First implementation of createElement

Here is our first implementation of `createElement`. If you're following along, add this at the top of your `src/index.js` file.

Our function accepts an element type, props, and a list of children. Then we just return those things inside a plain JS object. That's it. This is our virtual DOM!

So, what's left to do? We have our UI element "created", we just need to render it to the webpage by updating our browser's DOM.

We're going to write our own version of `ReactDOM.render` which will complete the core functionality of React. Below is some boilerplate for what our `render` function will do. Very similar to the `ReactDOM.render` function, our `render` function will

accept as arguments a virtual DOM , and a container of our HTML document to append our app to.

I'm going to pull the main workload of `render` out into another function, `buildDOM`. Our render function is in charge of two logically separate steps, and pulling out one of those tasks into another function will keep us organized. Add the following function definitions inside your `src/index.js` file.

The `buildDOM` function will accept a few different argument types as we'll see later, but for now, let's just assume that our `vnode` argument will be a plain virtual dom object (the return value from `createElement`).

Our first step in `buildDOM` is to create a new DOM node using the document API. The document API is our channel of communication between our virtual DOM and our browser's DOM. For more information on the document API functionality we'll be using, you can check out the MDN documentation [here](#).

Next, we'll assign our component's props as HTML attributes on the new DOM node we've created.

The following step is where things get a little tricky. We need to recursively build any child nodes the component has, and append them to our current node. The code itself looks nearly identical to setting our element's attributes, but there are a few non-obvious pitfalls we'll have to deal with soon.


And lastly, we returned the node.

`buildDOM` Edge Cases

Our `buildDOM` function is off to a good start, but there are a few edge cases it fails to address. I'm going to walk through each of the edge cases and their solutions, but I suggest you run these code snippets on your machine to see the errors first-hand and try to make more sense of them.


1.) Child elements that contain plain text






Imagine the case when our child element is just plain text. Our `buildDOM` function recursively calls itself on each child, and in this example, that child element is the string `"Hello, world"`. `buildDOM` can't treat this as a regular virtual DOM object, and instead should just create a new text node from the string.

2.) Function components (self-defined components)



The `elementType` property in App's virtual DOM is a function, not a plain HTML tag name like `'div'` or `'p'`. If we try to pass a function to `document.createElement` we'll get errors instead of our desired DOM element. To avoid this, we need to invoke the function attached to `elementType`, and call `buildDOM` on its return value.

3.) Arrays



This issue is difficult to make sense of, and it took me a bit of time to understand the problem when I was first debugging it. In short, the nested array is created because of the way we collect children inside our `createElement` function. So, we either need to support arrays as children inside our render function or avoid creating them in the first place. Since I don't want to render children in nested arrays any differently than regular child elements, I'm going to go with the latter solution. The simplest way to do this is by flattening our children array inside `createElement`. Luckily, JS arrays have a helper for us, [flat](#). With this strategy we're just treating items inside inner arrays as if they were top level elements.

To wrap things up, we'll complete our implementation of `render`. With our `buildDOM` helper function, we can break our `render` function down into two simple steps. First, use our `buildDOM` function to translate our virtual DOM into a real DOM element, and second, append the DOM element to the container.

Now, we get to see if all this really works. Inside the same file you've implemented `createElement`, `buildDOM`, and `render`, add some example code to mimic a React application.

If you rebuild your project and open the `index.html` file inside your browser, you should see that we have a working web app using our React library!

Summary

To review, we've implemented a way to define UI elements, and a way to render them. No, this isn't everything the React library offers, but it is the core of what React does, and we implemented it in less than 50 lines of code!

Although there is still a lot happening behind the scenes, I hope this made React feel less “magical”.

• • •

Source code: <https://github.com/AnvilDeveloperNetwork/ReactJS-Under-the-Hood>

Follow me on github: <https://github.com/claytn>

Reactjs

JavaScript

Reactjs Developers



[About](#) [Help](#) [Legal](#)

Get the Medium app

