

webpack and React

 (<https://github.com/learn-co-curriculum/react-hooks-webpack-and-react>)  (<https://github.com/learn-co-curriculum/react-hooks-webpack-and-react/issues/new>)

Learning Goals

- Learn what webpack is
- Understand how webpack integrates with a Create React App project

Introduction

In this lesson, we'll unpack what **webpack** brings to the table when developing React applications.

The Problem

To best describe webpack, we will begin by describing the problem that it was created to solve.

Picture having a server that sends some JavaScript-using webpage to browsers. Let's imagine we have some `animateDiv.js` script we want browsers to receive that itself makes use of `d3js`  (<https://d3js.org/>), a data visualization library. The first file we send to a requesting client, `index.html`, may look like this:

```
<!-- index.html -->
<html>
  <head>
    <meta charset="utf-8" />
    <script src=".vendors/d3.js"></script>
    <script src=".lib/animateDiv.js"></script>
    <title>Discotek</title>
  </head>
  <body>
```

```
<div class="animat" onclick="animateDiv">  
    I'm going to animate if you click me!  
</div>  
</body>  
</html>
```

With this approach, we are actually making three http requests to the server for the application:

- We hit the base url and are returned the `index.html` file
- `index.html` tells the browser to request `d3.js` from the server
- `index.html` tells the browser to request `animateDiv.js` from the server

A quick and dirty way around this would be to combine our JavaScript files into one file on the server (bringing this to two requests):

```
<!-- index.html -->  
<script src="./lib/combinedD3AnimateDiv.js"></script>
```

We could go one step further and even in-line the JavaScript directly into our HTML in a `<script>` tag (sending everything at once in `index.html`):

```
<!-- index.html -->  
...  
<script>  
    // all the contents of d3.js and animateDiv.js written directly here!  
</script>  
...
```

Unfortunately, this is not very practical. We need to, by 'hand', combine JavaScript code from multiple files into one. Well...we're programmers! We automate the boring tasks like this! Introducing **webpack**!

[webpack ↗\(https://webpack.js.org/\)](https://webpack.js.org/) lets us combine different files automatically. This means that we can freely import external JS code in our JavaScript files (both local files as well as `node_modules` installed with `npm`). We trust that webpack, before we send clients our JS code over the internet, intelligently packages it up for us. In a simplified example:

- File `siliconOverlord.js` has space-age AI code in it
- File `enslaveHumanity.js` wants to make use of this other file and send it to browsers all over the internet.
- Instead of always sending both `enslaveHumanity.js` and `siliconOverlord.js` to browsers, one after the other, **webpack** bundles them together into a single file that can be sent instead: `singularity.js`

If you have been working with dependencies already (`gem` s in rails, `require` in Ruby, `import/export` in vanilla JS, etc.) you may have noticed we did not need any tool like webpack to work with code written in other files. While this is true, and we don't *need* webpack to do this, let's highlight the problem webpack solves before trying to understand it.

When compiling a React application with webpack, it'll check every file for dependencies that it needs to import, and also include that code. In more technical terms, it's traversing the dependency tree and inlining those dependencies in our application. What we'll end up with is one big JS file that includes *all* of our code, including any dependencies (like `react`, your components, your `npm` modules, etc.) in that file too. The convenience of this is not to be underestimated: one file, with *all* of our code, means we only need to transfer a single thing to our clients when they ask for our React applications!

Enough theory, let's take a look at a rudimentary example of how webpack does this. Let's assume we have the following application on our server that we want to share with the world:

Simplified webpack example

The files we want our client to have, which constitute one whole web application:

```
// reveal.js (pre webpack bundling)
function reveal(person, realIdentity) {
  person.identity = realIdentity;
}
```

```
export default reveal;
```

```
// main.js (pre webpack bundling)
import reveal from "./reveal.js";
```

```
const spidey = {
  name: "Spider-Man",
  identity: "Friendly Neighborhood Spider-Man",
};

reveal(spidey, "Peter Parker");
```

Without webpack, we would need to find some way to send both files to our client and ensure they are playing nicely together. We couldn't just send the `main.js` file to our client expecting it to make use of the `reveal` function: the client hasn't even received the `reveal.js` file in this case! While we have several ways we could make this work, most of them are headaches and someone else has already made an excellent solution: webpack.

The result after we unleash webpack on these files:

```
// bundle.js (post webpack bundling)
function reveal(person, realIdentity) {
  person.identity = realIdentity;
}

const spidey = {
  name: "Spider-Man",
  identity: "Friendly Neighborhood Spider-Man",
};

reveal(spidey, "Peter Parker");
```

After bundling our files with webpack, we have a single, all-encompassing, file that ensures our dependencies are right where they belong.

More than Just JavaScript

In addition to bundling up our JavaScript code, webpack also allows us to manage all kinds of other assets in our React projects. You may have noticed that we're able to `import` CSS files and images in addition to importing JavaScript files — this is something we can *only* do because of

webpack.

You can read more about how webpack is configured to work in our React projects in the [Create React App docs](https://create-react-app.dev/docs/adding-a-stylesheet) (https://create-react-app.dev/docs/adding-a-stylesheet).

Conclusion

You have just read a lot of information about a tool you likely have not worked directly with before. Luckily, it's straightforward to summarize:

In our React applications, **webpack** manages pesky dependency loading for us by **bundling** all the code — our own code plus the code for our dependences — and outputting it in a single file.

Looking Forward

After reading the previous lesson on Babel and now this one on webpack, you may, understandably, be asking yourself:

- "How important is this webpack/Babel jargon?"
- "How much do I need to learn about the different tools that improve React development experience vs. actual React programming?"

When introducing students to new material, we are constantly balancing an explanation of the fundamentals against practice on the real skills that will get you producing valuable applications quickly. We believe that, while learning React basics, it's important to know how these tools (webpack and Babel) work on a *high level*.

Most React code nowadays is being compiled one way or another — be it using **webpack**, an alternative such as [Browserify](http://browserify.org/) (http://browserify.org/), or something else. Thanks to tools like Create React App, we can use webpack without worrying about the complexities of how it works or setting up that configuration ourselves.

For the most part, Babel and webpack will be abstracted from you so you can focus on learning the primary React competencies. This will streamline the development process, and let you focus on React itself instead of the tooling.

Resources

- [webpack](https://webpack.js.org/) (https://webpack.js.org/)