Tip 50: Use Build Tools to Combine Components

In this tip, you'll learn how to compile JavaScript code and assets with build tools.

We'll cover the following

In the previous tip, you saw the advantages of the component architecture. You also learned about the one big problem with component architecture: *It won't work natively in browsers*.

Also in the previous tip, you used the tools provided by create-react-app to get your project compiled and running. That's great. You should always take advantage of predesigned build tools. Every project has one. Sometimes there are official projects—angular-cli and EmberCLI are examples—and if there are no official projects, search code repos such as github for Starter Packs. Eventually, however, you'll need to customize your build.

Build tools

In this tip, you're going to make a basic build process. Build tools can be exhausting, and it can be difficult to keep up with the latest trends and tools. Don't get discouraged. A **build tool** is merely a way for you to *process* the code one piece at a time.

To begin, take a simplified version of your components from the previous tip. Start by removing everything except for some HTML, in the form of React JSX, and some JavaScript. It'll be easier to make build tools when you have fewer assets. Here's a basic container component:

```
import React from 'react';
import Copyright from './components/Copyright/Copyright';
export default function App() {
    return (
        <div className="main">
            <footer>
                <Copyright />
            </footer>
        </div>
    );
}
```

Here's a stripped-down version of your *copyright* component:

```
import React from 'react';
export default function CopyrightStatement() {
    const year = new Date().getFullYear();
    return (
        <div className="copyright">
            Copyright {year}
        </div>
    );
}
```

Even though these files are simple, you couldn't run them in a browser. And even if you could, you certainly wouldn't be able to run them in older browsers. You need a tool to convert ES6 syntax—`import` and `export`—and JSX into compatible code.

Working with Babel

Fortunately, there's an amazing tool called Babel that can convert bleedingedge JavaScript to browser-friendly code. Babel is the single-most-important tool you have for working with modern JavaScript. Not only does it convert your ES6+ JavaScript, but you can even configure Babel to use syntax that's still in committee.

To get started, you need to install the **Babel command-line interface** (**cli**) along with the `preset-env` to convert ES6+ and `babel-present-react` to convert react code.

The installation command should look familiar. This time, you're installing three packages with a single command.

```
npm install --save-dev babel-cli babel-preset-env babel-preset-react
```

The next thing you need to do is set up a `.babelrc` file to hold your configuration

information. This file tells Babel what kind of code you have and how Babel will

need to convert it. In this case, you have ES6 code—signified with `env`—and react code.

```
{ "presets": ["env", "react"] }
```

Now add a script to your `package.json` file and you'll be ready to compile. Notice that you're outputting the compiled information to a single file, `bundle.js`, in the build directory. Here's your final `package.json`.

```
{
    "name": "initial",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "build": "babel src/index.js -o build/bundle.js"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
        "babel-cli": "^6.26.0",
        "babel-preset-env": "^1.6.1",
        "babel-preset-react": "^6.24.1"
    },
    "dependencies": {
        "react": "^16.1.1",
        "react-dom": "^16.1.1"
    }
}
```

Finally, update your `index.html` to use the compiled code:

```
< !DOCTYPE html >
    <html lang="en">
        <head>
            <title>Sample</title>
        </head>
        <body>
            <div id="root">
            </div>
            <script src="./build/bundle.js"></script>
        </body>
    </html>
```

If you try to open that file in a browser, you'll encounter a problem. The console will display an *error*: `Uncaught ReferenceError: require isn't defined`.

Installing webpack

Installing webpack

Babel converts the code, but it doesn't include a module loader, which handles the compiled imports and exports. You have a few options for module loaders. Currently, the most popular module loaders are webpack and rollup.js. In this example, you'll use *webpack*.

Webpack is a project that can handle everything from combining your JavaScript to processing your CSS or SASS, to image conversion. Webpack can handle so many file types because you declare different actions—*referred to as loaders in webpack—based on file extension*.

To get webpack working, you'll need to install it. You'll also need to install a loader for Babel. The webpack documentation encourages you to think of loaders as a task in another build tool. Because compiling the code with Babel is just a step in getting usable JavaScript, you'll need the babel-loader. You can install them both in the same command:

```
npm install --save-dev babel-loader webpack.
```

You'll also need to create a `webpack.config.js` file. Inside the file, declare an *entry* point and an *output* path. After that, you need to tell webpack what to do with the code it encounters. This is where the loaders come in.

At this point, you're probably getting overwhelmed. So remember: Don't think of the whole system—*just think about each step*. You first needed to convert ES6 and React code, so you installed Babel. Next, you wanted to combine everything together, so you installed webpack. Now, you need to declare what you want webpack to do with JavaScript specifically. Next, you'll make similar declarations for style and assets.

How does webpack work with JS?

Webpack uses *regular expressions* to decide which loader to use on each file. Because you're working with JavaScript, you only want files that match `.js`. When webpack encounters a file with a `.js` extension—such as `Copyright.js`—you need to tell it which loader to use. In this case, it needs to run the `babel-loader`.

```
const path = require('path');
module.exports = {
    entry: './src/index.js',
```

```
    module: {
        loaders: [
            {
                test: /\.js?/,
                use: 'babel-loader',
            },
        ],
    },
    output: {
        filename: 'build/bundle.js',
        path: path.resolve(__dirname),
    },
};
```

The last step is to update your `package.json` script to call *webpack*. Webpack will look for your *config* file, so you don't need any other flags or arguments. All you need to do is change

```
"scripts": {
  "build": "babel src/index.js -o build/bundle.js"
}
```

to

```
"scripts": {
  "build": "webpack"
}
```

If you run this, you'll finally be able to see your code in the browser. Try it out.

Compiling CSS

Now that you have the JavaScript working, it's time for things to get interesting. Remember, the goal is to have components that import all their dependencies. You need webpack to compile your JavaScript, but also to compile your CSS and load your images.

Start with CSS. Go back to your `Copyright.js` file and import your CSS. It should look exactly like it did in the previous tip.

```
import React from 'react';
import './Copyright.css';
export default function CopyrightStatement() {
    const year = new Date().getFullYear();
    return (
        <div className="copyright">
            Copyright {year}
        </div>
```

```
    );
}
```

Now you'll need to install a CSS loader and update your `webpack.config.js` file. There are lots of tools for handling CSS, but in this case, keep it simple. Install and add *two* loaders—a *CSS loader* to interpret the CSS file and a *style loader* to inject the styles into the `<head>` element on your page.

```
npm install --save-dev css-loader style-loader.
```

Now that you've installed your loaders, update your webpack config by adding a test for files that end in `css`. This time, you won't use a single loader. You'll use two loaders—*css-loader and style-loader*—so you'll need an array of strings instead of a single string. Add the style-loader first and then the css-loader.

```
module: {
    loaders: [
        {
            test: /\.css$/,
            use: [
                'style-loader',
                'css-loader',
            ],
        },
        {
            test: /\.js?/,
            use: 'babel-loader',
        },
    ],
},
```

When you run the build script and open `index.html`, your components will have the correct styles.

Impressive, huh? This is why developers fell in love with webpack. You can keep all your assets batched together and you can call different actions, or series of actions, on each file type.

Handling images

The final step is to handle your image. This time you aren't compiling an image. Instead, you're going to use webpack to move the file and *rename* it to a *unique* name. Webpack will automatically update the `src` link in your markup.

As a reminder, here's your component with an imported image:

```
import React from 'react';
import './IdeaButton.css';
import idea from './idea.svg';

export default function IdeaButton({ handleClick, message }) {
    return (
        <button
            className="idea-button"
            onClick={handleClick}
        >
            <img
                className="idea-button__icon"
                src={idea}
                alt="idea icon"
            />
            {message}
        </button>
    );
}
```

Because you aren't doing any specific image manipulation, use *file-loader* to move and update your `src` path. In your webpack config, you'll test to see if the file is an **SVG**.

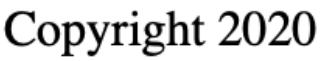
This time, you aren't just declaring a loader; you're also passing options to the loader. This means you'll pass an array containing a single object. The object will include your loader and configuration options. The only option you need to pass is the directory for your images. This directory will be where the browser looks for images, so it's best to reuse your `build` directory.

Set the `outputPath` to the `build` directory:

```
module: {
    loaders: [
        {
            test: /\.svg?/,
            use: [
                {
                    loader: 'file-loader',
                    options: {
                        outputPath: 'build/',
                    },
                },
            ],
        },
        {
            test: /\.css$/,
            use: [
                'style-loader',
                'css-loader',
            ],
        },
        {
            test: /\.js?/,
            use: 'babel-loader',
```

```
        },
    ],
},
```

Run the build script. Open up `index.html` and you have your components.



index.html

See that wasn't so bad! Of course, if this were an enterprise application, you'd want a server. You'll probably have more images than just SVGs. You might want the CSS to go to a style sheet instead of `<style>` tags. Build tools can handle all that for you.

The key is to take it slow and add one piece at a time. It's much harder to add a configuration to a large project than it is to add it piece by piece. Webpack and rollup.js can be complex projects. Webpack has put a lot of work into updating its documentation, and it's worth reading as you explore more on your own.

At this point, you have all the tools you need to write modern JavaScript applications. The final tip is a little different. CSS and HTML are also growing and evolving—actions that used to require JavaScript can now be handled by CSS. In this case, you should happily abandon JavaScript and use other tools.

In the next tip, you'll see how to animate page elements with CSS.