**Webpack 101**

# **What is webpack?**

The basic definition of webpack is that it’s a module bundler. But what exactly is a module bundler?

Module bundlers are the way to organize and combine many files of JavaScript code into one file. Module bundlers allow you to organize your code by importing other modules, and can ultimately reduce redundant code blocks without having to manage the order in which code works. Before bundlers, people had to carefully manage script loading due to dependency conditions. A bundler allows you to write modules that import these functions and automatically orders these dependencies.

Javascript frameworks like React rely on a lot of modules. What you code is often not recognized by your web browser and there comes **webpack** - a simple way to manage your dependencies, get stuff working properly, automatically change your code into something any browser can comprehend. Webpack not only bundles but it also manages your code.

# **Why do we need module bundlers?**

Bundling is necessary on browsers because they don't support require or import

If you try to use ‘require()’ to add a particular dependency to your application, the browser will throw an error saying require is not defined as the browser doesn’t have access to the file system. This is where a module bundler comes in. A JavaScript module bundler is a tool that gets around the problem with a build step (which has access to the file system) to create a final output that is browser compatible (which doesn’t need access to the file system). In this case, we need a module bundler to find all require statements (which is invalid browser JavaScript syntax) and replace them with the actual contents of each required file. The final result is a single bundled JavaScript file (with no require statements)!

# **How CRA (create-react-app) uses Webpack**

We always need to configure webpack when setting up a react project but that could get a little confusing in the beginning. For that reason, a library called ‘create-react-app’ was built which handles all the basic configurations for you.

Now what exactly happens behind the scenes?

After running the command npx create-react-app app a package.json file is formed and a dependency list is created. This includes react, react-dom and react-scripts. react-scripts is responsible for most of the dependencies you find installed in the huge node\_modules folder. It basically installs all the dependencies required to configure your React app. Some famous ones are Babel, Eslint, Jest, PostCSS and **Webpack**. There are quite a few dependencies that come with webpack. Like the ones mentioned below.

* [@svgr/webpack](https://www.npmjs.com/package/@svgr/webpack) — Transforms SVGs to React components
* [case-sensitive-paths-webpack-plugin](https://www.npmjs.com/package/case-sensitive-paths-webpack-plugin) — Stops case sensitive operating systems and non case sensitive ones from causing issues for each other
* [css-loader](https://www.npmjs.com/package/css-loader) — Allows @import url() to work in CSS files
* [file-loader](https://www.npmjs.com/package/file-loader) — Resolves import/require urls and adds the file to the directory
* [html-webpack-plugin](https://www.npmjs.com/package/html-webpack-plugin) — Creates an HTML file of all your webpack bundles
* [mini-css-extract-plugin](https://www.npmjs.com/package/mini-css-extract-plugin) — Creates a CSS file per JS containing CSS
* [optimize-css-assets-webpack-plug](https://www.npmjs.com/package/optimize-css-assets-webpack-plugin) — Uses [cssnano](https://github.com/cssnano/cssnano) to minify and optimize CSS code
* [pnp-webpack-plugin](https://www.npmjs.com/package/pnp-webpack-plugin) — Yarn Plug’n’Play plugin for Webpack
* [sass-loader](https://www.npmjs.com/package/sass-loader) — This allows Webpack to turn Sass into normal CSS files. Works with css-loader above to output files
* [style-loader](https://www.npmjs.com/package/style-loader) — Adds CSS to style tags
* [terser-webpack-plugin](https://www.npmjs.com/package/terser-webpack-plugin) — Allows minification of Javascript
* [url-loader](https://www.npmjs.com/package/url-loader) — Transforms files into base64 URIs. This allows you to not have to request a second file, as the data is transferred in the HTML document
* [webpack](https://www.npmjs.com/package/webpack) — What everything else here plugs into. Allows transformation and bundling of JS files
* [webpack-dev-server](https://www.npmjs.com/package/webpack-dev-server) — Adds live reloading to Webpack. This means it will run everytime you save
* [webpack-manifest](https://www.npmjs.com/package/webpack-manifest-plugin) — Creates a [manifest.json](https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json) from your files
* [workbox-webpack-plugin](https://developers.google.com/web/tools/workbox/modules/workbox-webpack-plugin) — This is related to the service worker, adding tools to generate, and one that adds to it

As you can see the majority of what react-scripts does for you is related to setting up and running Webpack. Almost every other dependency listed here either plugs into Webpack, or plugs into something that plugs into Webpack.

# **Modifying create-react-app’s Webpack config**

In the above section we discussed how create-react-app handles all the nitty-gritties of Webpack for you but sometimes you might face issues related to certain technologies not being supported or some Webpack config causing the app to behave differently than expected. In those cases if you have full access to the Webpack configuration, the fix is only a matter of 2 or 3 lines. But in create-react-app you don’t have access to the Webpack configuration.

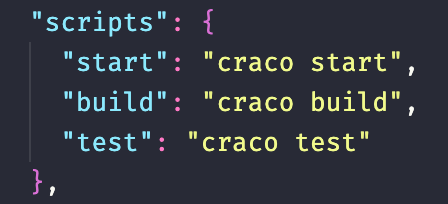
One option to modify create-react-app’s config is to eject it but that’s a bad idea as you won’t be able to benefit from the updates of create-react-app. So how do we modify the config without ejecting create-react-app?

There are some pre-existing solutions to this. The one we’re going to discuss is [CRACO](https://www.npmjs.com/package/@craco/craco) ([react-app-rewired](https://github.com/timarney/react-app-rewired) and [rewire](https://github.com/jhnns/rewire) are some alternatives).

1. Once you’ve got your CRA app ready, you can run the following command to install CRACO:

> ​​npm i @craco/craco -SE --force

1. Now go to your package.json file and modify the scripts as shown below



Notice that we have also removed the eject script as it’s no longer needed.

1. Now you just need to create a craco.config.js file in your root directory. You can add any webpack configs that you would like to use in this file.
2. Now you can just use npm start as you normally would and your application will run with the configs you’ve added.

# **Core concepts of webpack**

**Everything is a module** - Just like Js files can be “modules”, everything else (CSS, Images, HTML) can also be modules. This means that we can split any artifact into smaller manageable pieces, reuse them and so on.

**Load only “what” you need and “when” you need** - Typically module bundlers take all the modules and generate a large single output “bundle.js” file. But in many real world apps, this “bundle.js” file could be 10Mb-15Mb and could take forever to load. So Webpack has various features to split your code and generate multiple “bundle” files and also load parts of the app asynchronously so that you just load what you need and when you need it.

# **Custom Webpack configurations**

There are 4 basic configurations that are required for a project. These configurations are added in webpack.config.js

1. **entry**



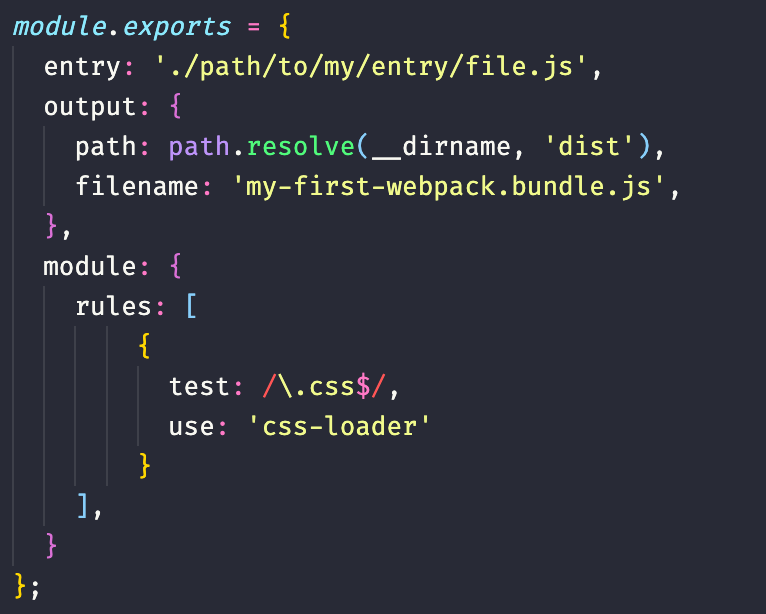
The snippet above is an example of an entry configuration. You are basically telling webpack the very first file it needs to look at when it starts creating the [dependency graph](https://webpack.js.org/concepts/dependency-graph/). The dependency graph starts from this entry file and then builds its dependencies and the dependencies of its dependencies and so on. webpack goes through all of these dependencies and creates modules then repeats this entire process all over the entire application.

1. **output**



The output configuration tells webpack how and where it should put the bundles and its format. For instance with the output value below, you are telling webpack to put the bundles in a javascript file named my-first-webpack.bundle.js, in a dist folder under the same directory to where the webpack.config.js is located.

1. **rules and loaders**

****

The rules and loaders config instruct webpack how it should process different file types and convert them into valid modules before they are added in the dependency graph.

loaders usually have two properties, namely:

1. **test** : The test property tells the type of file that will be processed.
2. **use** : The use property tells webpack what loader will be used in processing the file.

The modules property shown above is basically just telling webpack this:

"*Hey webpack compiler, when you come across a path that resolves to a .css file inside of a require()/import statement, use the css-loader to transform it before you add it to the bundle.*"

1. **plug-ins**

Plugins allow you to perform any kind of functionality. While loaders are used to transform certain types of modules, plugins can be leveraged to perform a wider range of tasks like bundle optimization, asset management, injection of environment variables, minifying files, etc. Webpack already has a rich collection of plugins, so developers can also check them out before reinventing the wheel.

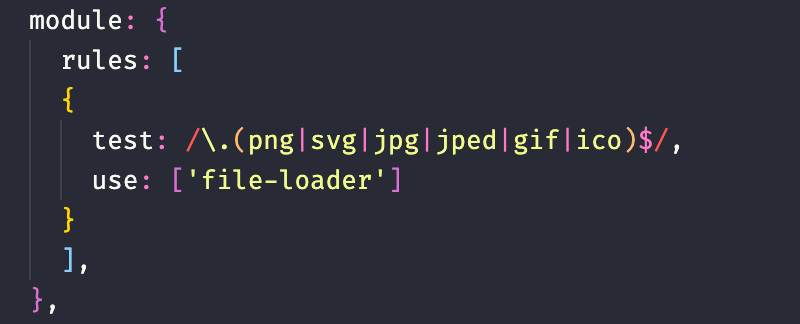
# **Asset management in webpack**

Most apps you build will have some images here and there. Here we will discuss how to handle such files. Prior to webpack, front-end developers would use tools like [grunt](https://gruntjs.com/) and [gulp](https://gulpjs.com/) to process these assets and move them from their /src folder into their /dist or /build directory. The same idea was used for JavaScript modules, but tools like webpack will dynamically bundle all dependencies (creating what's known as a [dependency graph](https://webpack.js.org/concepts/dependency-graph)). This is great because every module now explicitly states its dependencies and we'll avoid bundling modules that aren't in use. One of the coolest webpack features is that you can also include any other type of file, besides JavaScript, for which there is a loader or built-in [Asset Modules](https://webpack.js.org/guides/asset-modules/) support.

In this example we will show how to make webpack look for images:

1. Add the following code in your module.exports inside webpack.config.js, for this you will need a loader called file-loader , which you can get by simply running

yarn add -D file-loader



1. You can import any image from the image directory you’ve made.

const image = require(‘./images/img.jpg’)

1. Now if you run yarn webpack, it will create a build and in the dist folder in your root directory, you’ll be able to see a hashed version of the image, it will be served to the browser in that way.