Lesson Weekend

Intermediate JavaScript (/intermediate-javascript) / Test-Driven Development and Environments with JavaScript (/intermediate-javascript/test-driven-development-and-environments-with-javascript) / Introduction to webpack

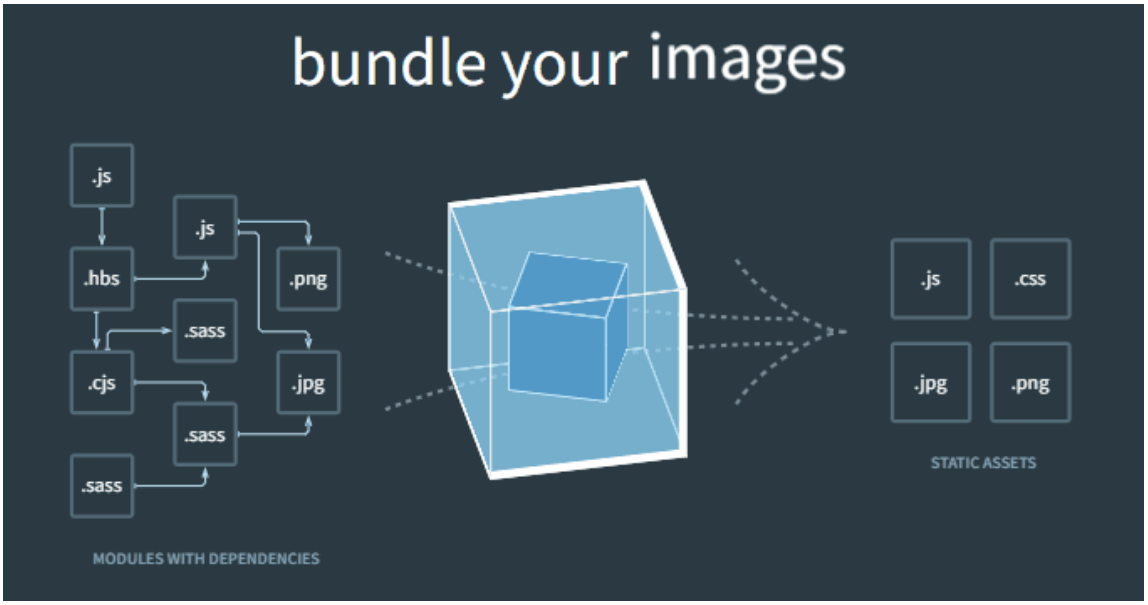
Text

webpack is a **module bundler** that bundles our code. A **module** is a single JavaScript file. As a bundler, webpack stitches together separate modules (JavaScript files) into a single file called a bundle. By bundling our code, we can make our website more efficient at loading and processing its resources, because the source code has been optimized.

That may not seem like much, but imagine a large, complex application with hundreds of files using many different libraries and dependencies. We need all these files and libraries to work together. That's where webpack comes in.

Introduction to webpack

The following gif from webpack's homepage (https://webpack.js.org/) is a very helpful visualization of what it does as a module bundler.



As the gif suggests, other than JavaScript, webpack can also bundle styles, images, and other assets (like fonts). These files are also considered modules. To bundle files types other than JavaScript, webpack uses loaders. A webpack **loader** is a tool that enables webpack to work with a certain type of file, so it can process it and bundle it. Loaders don't come installed with webpack — we always need to install packages to use loaders in our projects.

Through webpack configurations, we can also optimize our code *and* improve our development experience. We use plugins and built-in webpack configurations to do this. A **built-in webpack configuration** is just that — tooling options that come out of the box with webpack.

Just like loaders, plugins don't come installed with webpack — we always need to install packages to use plugins in our projects. A webpack **plugin** is a tool that works on the entire bundle to optimize its performance or improve our developer experience.

There are several other popular tools for achieving the same goals, including Gulp.js and Grunt.js. These tools are known as **task runners**. A task runner is exactly what it sounds like: a tool to run tasks such as concatenation (combining multiple files into one) and

minification (changing variable names and removing blank spaces to make the file smaller). webpack does the same things, but it does so through loaders and plugins.

webpack is the most popular solution today and it's used with major frameworks such as React and Angular. If you're interested, you can explore Gulp.js or Grunt.js in your own time — though they aren't as widely used now that module bundlers have taken over. You're not expected to know the fine points of the differences between module bundlers and task runners while you're at Epicodus, but you're encouraged to do some additional reading on your own.

Over the remaining weekend homework, we'll make incremental additions to our webpack configuration. We will also provide a basic explanation of what webpack is doing. These lessons are not designed to be exhaustive. For exhaustive details, the webpack documentation is excellent. We recommend referring back to the documentation (https://webpack.js.org/) if you have further questions or need clarification about webpack.

How Does webpack Work?

So how does webpack work and why is it so useful?

webpack uses a **dependency graph** to recursively manage an application's assets. A graph is a data structure that describes complex and non-linear relationships between objects. **Recursive** just means to perform the same action over and over again until a goal is met. For webpack, the goal is working through each module in a project and identifying its dependencies (files it depends on to function). The result is a dependency graph.

This all sounds complicated, and it truthfully is, but the good news is that we don't need to understand how webpack works under the hood in order to use it. We'll let webpack do most of the heavy lifting for us. Let's take a look at an example to better understand how webpack creates a dependency graph.

Imagine that we're building an application that makes very complex peanut butter and jelly sandwiches. As a result, we have multiple JavaScript files for managing the creation of these sandwiches: `peanut-butter.js`, `jelly.js` and `bread.js`.

As a module bundler, webpack is perfect for this job, but how does it know where to start its bundling process? Well, in addition to our typical source code, we need to identify an **entry point**.

Think of an **entry point** as a door leading into our application. webpack needs this entry point in order to recursively gather all the other files the application needs. A bigger application may have multiple entry points but we'll only be working with one.

The entry point for our applications is the file `index.js`, where we keep our user interface logic. Why `index.js`? it's simply a common naming convention for an entry point file.

Here's what the first few lines of `index.js` might look like:

index.js

```
import { PeanutButter } from './peanut-butter.js'
import { Jelly } from './jelly.js'
import { Bread } from './bread.js'
import '../css/styles.css'

...
```

We haven't covered `import` statements just yet — we'll do so in a few lessons. For now, just be aware that an `import` statement is exactly what it sounds like: a way to import code that's in one file into another file.

When we tell webpack to load `index.js`, webpack will recursively load and concatenate all the code from `index.js` as well as any required code from other files, in this case `peanut-butter.js`,

`jelly.js`, and `bread.js`.

And if `jelly.js` imports code from yet another file called `blueberry.js`, webpack would gather that code, too.

In the end, webpack will gather all of this code into a single file with a name like `bundle.js`. In fact, this is exactly what we'll call our bundled code. Remember how we mentioned that our finished project will have a `dist` directory with a file named `bundle.js` inside it? Well, webpack automatically creates that folder and file for us!

And just like that, our code is bundled into one file.

As we mentioned previously, webpack will not just load our project's JavaScript files, but also many other types of assets as long as we're using the right loaders and plugins. That's why we'll ultimately store all our source code (CSS, HTML, JS, and other assets) in our `src` directory — so that webpack can process all of these files.

In general, we don't really need to worry about how webpack is gathering its resources. This is one of those things where the tool we're using will take care of things for us and we don't need to dig too much deeper. However, it's good to have a general sense of what webpack is actually up to behind the scenes. Ultimately, as long as you correctly set up your webpack configuration file and use `import` statements, webpack will take care of the rest for you.

Previous (/intermediate-javascript/test-driven-development-and-environments-with-javascript/installing-dependencies-with-npm-webpack-and-webpack-cli)
Next (/intermediate-javascript/test-driven-development-and-environments-with-javascript/configuring-webpack-and-using-npm-scripts)

(http://www.epicodus.com)