**Basic Setup**

First let's create a directory, initialize npm, [install webpack locally](https://webpack.js.org/guides/installation/#local-installation), and install the [webpack-cli](https://github.com/webpack/webpack-cli) (the tool used to run webpack on the command line):

mkdir webpack-demo

cd webpack-demo

npm init -y

npm install webpack webpack-cli --save-dev

Throughout the Guides we will use **diff** blocks to show you what changes we're making to directories, files, and code. For instance:

+ this is a new line you shall copy into your code

- and this is a line to be removed from your code

and this is a line not to touch.

Now we'll create the following directory structure, files and their contents:

**project**

webpack-demo

|- package.json

+ |- index.html

+ |- /src

+ |- index.js

**src/index.js**

function component() {

const element = document.createElement('div');

// Lodash, currently included via a script, is required for this line to work

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

}

document.body.appendChild(component());

**index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Getting Started</title>

<script src="https://unpkg.com/lodash@4.17.20"></script>

</head>

<body>

<script src="./src/index.js"></script>

</body>

</html>

We also need to adjust our package.json file in order to make sure we mark our package as private, as well as removing the main entry. This is to prevent an accidental publish of your code.

**Tip**

If you want to learn more about the inner workings of package.json, then we recommend reading the [npm documentation](https://docs.npmjs.com/files/package.json).

**package.json**

{

"name": "webpack-demo",

"version": "1.0.0",

"description": "",

- "main": "index.js",

+ "private": true,

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"keywords": [],

"author": "",

"license": "MIT",

"devDependencies": {

"webpack": "^5.38.1",

"webpack-cli": "^4.7.2",

}

}

In this example, there are implicit dependencies between the <script> tags. Our index.js file depends on lodash being included in the page before it runs. This is because index.js never explicitly declared a need for lodash; it assumes that the global variable \_ exists.

There are problems with managing JavaScript projects this way:

* It is not immediately apparent that the script depends on an external library.
* If a dependency is missing, or included in the wrong order, the application will not function properly.
* If a dependency is included but not used, the browser will be forced to download unnecessary code.

Let's use webpack to manage these scripts instead.

**Creating a Bundle**

First we'll tweak our directory structure slightly, separating the "source" code (./src) from our "distribution" code (./dist). The "source" code is the code that we'll write and edit. The "distribution" code is the minimized and optimized output of our build process that will eventually be loaded in the browser. Tweak the directory structure as follows:

**project**

webpack-demo

|- package.json

+ |- /dist

+ |- index.html

- |- index.html

|- /src

|- index.js

**Tip**

You may have noticed that index.html was created manually, even though it is now placed in the dist directory. Later on in [another guide](https://webpack.js.org/guides/output-management/#setting-up-htmlwebpackplugin), we will generate index.html rather than edit it manually. Once this is done, it should be safe to empty the dist directory and to regenerate all the files within it.

To bundle the lodash dependency with index.js, we'll need to install the library locally:

npm install --save lodash

**Tip**

When installing a package that will be bundled into your production bundle, you should use npm install --save. If you're installing a package for development purposes (e.g. a linter, testing libraries, etc.) then you should use npm install --save-dev. More information can be found in the [npm documentation](https://docs.npmjs.com/cli/install).

Now, let's import lodash in our script:

**src/index.js**

+import \_ from 'lodash';

+

function component() {

const element = document.createElement('div');

- // Lodash, currently included via a script, is required for this line to work

+ // Lodash, now imported by this script

element.innerHTML = \_.join(['Hello', 'webpack'], ' ');

return element;

}

document.body.appendChild(component());

Now, since we'll be bundling our scripts, we have to update our index.html file. Let's remove the lodash <script>, as we now import it, and modify the other <script> tag to load the bundle, instead of the raw ./src file:

**dist/index.html**

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Getting Started</title>

- <script src="https://unpkg.com/lodash@4.17.20"></script>

</head>

<body>

- <script src="./src/index.js"></script>

+ <script src="main.js"></script>

</body>

</html>

In this setup, index.js explicitly requires lodash to be present, and binds it as \_ (no global scope pollution). By stating what dependencies a module needs, webpack can use this information to build a dependency graph. It then uses the graph to generate an optimized bundle where scripts will be executed in the correct order.

With that said, let's run npx webpack, which will take our script at src/index.js as the [entry point](https://webpack.js.org/concepts/entry-points), and will generate dist/main.js as the [output](https://webpack.js.org/concepts/output). The npx command, which ships with Node 8.2/npm 5.2.0 or higher, runs the webpack binary (./node\_modules/.bin/webpack) of the webpack package we installed in the beginning:

$ npx webpack

[webpack-cli] Compilation finished

asset main.js 69.3 KiB [emitted] [minimized] (name: main) 1 related asset

runtime modules 1000 bytes 5 modules

cacheable modules 530 KiB

./src/index.js 257 bytes [built] [code generated]

./node\_modules/lodash/lodash.js 530 KiB [built] [code generated]

webpack 5.4.0 compiled successfully in 1851 ms

**Tip**

Your output may vary a bit, but if the build is successful then you are good to go.

Open index.html from the dist directory in your browser and, if everything went right, you should see the following text: 'Hello webpack'.

**Modules**

The [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) and [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) statements have been standardized in [ES2015](https://babeljs.io/docs/en/learn/). They are supported in most of the browsers at this moment, however there are some browsers that don't recognize the new syntax. But don't worry, webpack does support them out of the box.

Behind the scenes, webpack actually "**transpiles**" the code so that older browsers can also run it. If you inspect dist/main.js, you might be able to see how webpack does this, it's quite ingenious! Besides import and export, webpack supports various other module syntaxes as well, see [Module API](https://webpack.js.org/api/module-methods) for more information.

Note that webpack will not alter any code other than import and export statements. If you are using other [ES2015 features](http://es6-features.org/), make sure to [use a transpiler](https://webpack.js.org/loaders/#transpiling) such as [Babel](https://babeljs.io/) or [Bublé](https://buble.surge.sh/guide/) via webpack's [loader system](https://webpack.js.org/concepts/loaders/).

**Using a Configuration**

As of version 4, webpack doesn't require any configuration, but most projects will need a more complex setup, which is why webpack supports a [configuration file](https://webpack.js.org/concepts/configuration). This is much more efficient than having to manually type in a lot of commands in the terminal, so let's create one:

**project**

webpack-demo

|- package.json

+ |- webpack.config.js

|- /dist

|- index.html

|- /src

|- index.js

**webpack.config.js**

const path = require('path');

module.exports = {

entry: './src/index.js',

output: {

filename: 'main.js',

path: path.resolve(\_\_dirname, 'dist'),

},

};

Now, let's run the build again but instead using our new configuration file:

$ npx webpack --config webpack.config.js

[webpack-cli] Compilation finished

asset main.js 69.3 KiB [compared for emit] [minimized] (name: main) 1 related asset

runtime modules 1000 bytes 5 modules

cacheable modules 530 KiB

./src/index.js 257 bytes [built] [code generated]

./node\_modules/lodash/lodash.js 530 KiB [built] [code generated]

webpack 5.4.0 compiled successfully in 1934 ms

**Tip**

If a webpack.config.js is present, the webpack command picks it up by default. We use the --config option here only to show that you can pass a configuration of any name. This will be useful for more complex configurations that need to be split into multiple files.

A configuration file allows far more flexibility than CLI usage. We can specify loader rules, plugins, resolve options and many other enhancements this way. See the [configuration documentation](https://webpack.js.org/configuration) to learn more.

**NPM Scripts**

Given it's not particularly fun to run a local copy of webpack from the CLI, we can set up a little shortcut. Let's adjust our *package.json* by adding an [npm script](https://docs.npmjs.com/misc/scripts):

**package.json**

{

"name": "webpack-demo",

"version": "1.0.0",

"description": "",

"private": true,

"scripts": {

- "test": "echo \"Error: no test specified\" && exit 1"

+ "test": "echo \"Error: no test specified\" && exit 1",

+ "build": "webpack"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"webpack": "^5.4.0",

"webpack-cli": "^4.2.0"

},

"dependencies": {

"lodash": "^4.17.20"

}

}

Now the npm run build command can be used in place of the npx command we used earlier. Note that within scripts we can reference locally installed npm packages by name the same way we did with npx. This convention is the standard in most npm-based projects because it allows all contributors to use the same set of common scripts.

Now run the following command and see if your script alias works:

$ npm run build

...

[webpack-cli] Compilation finished

asset main.js 69.3 KiB [compared for emit] [minimized] (name: main) 1 related asset

runtime modules 1000 bytes 5 modules

cacheable modules 530 KiB

./src/index.js 257 bytes [built] [code generated]

./node\_modules/lodash/lodash.js 530 KiB [built] [code generated]

webpack 5.4.0 compiled successfully in 1940 ms

**Tip**

Custom parameters can be passed to webpack by adding two dashes between the npm run build command and your parameters, e.g. npm run build -- --color.

**Conclusion**

Now that you have a basic build together you should move on to the next guide [Asset Management](https://webpack.js.org/guides/asset-management) to learn how to manage assets like images and fonts with webpack. At this point, your project should look like this:

**project**

webpack-demo

|- package.json

|- webpack.config.js

|- /dist

|- main.js

|- index.html

|- /src

|- index.js

|- /node\_modules