Getting Started

webpack is used to compile JavaScript modules. Once installed, you can interface with webpack either from its CLI or API. If you're still new to webpack, please read through the core concepts and this comparison to learn why you might use it over the other tools that are out in the community.

Basic Setup

First let's create a directory, initialize npm, install webpack locally, and install the 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.*

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>
    <title>Getting Started</title>
    <script src="https://unpkg.com/lodash@4.16.6"></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.

If you want to learn more about the inner workings of `package.json`, then we recommend reading the npm documentation.

package.json

```
  {
    "name": "webpack-demo",
    "version": "1.0.0",
    "description": "",
+   "private": true,
-   "main": "index.js",
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "webpack": "^4.20.2",
      "webpack-cli": "^3.1.2"
    },
    "dependencies": {}
  }
```

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 just 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
```

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

```
npm install --save lodash
```

> *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.*

Now, lets 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
    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>
     <title>Getting Started</title>
-    <script src="https://unpkg.com/lodash@4.16.6"></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, and will generate `dist/main.js` as the 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

...
Built at: 13/06/2018 11:52:07
```

```
  Asset      Size  Chunks             Chunk Names
main.js  70.4 KiB       0  [emitted]  main
...

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value. S
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack
```

> *Your output may vary a bit, but if the build is successful then you are good to go. Also, don't worry about the warning, we'll tackle that later.*

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

> *If you are getting a syntax error in the middle of minified JavaScript when opening `index.html` in the browser, set `development mode` and run `npx webpack` again. This is related to running `npx webpack` on latest Node.js (v12.5+) instead of LTS version.*

Modules

The `import` and `export` statements have been standardized in ES2015. Although they are not supported in most browsers yet, 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 for more information.

Note that webpack will not alter any code other than `import` and `export` statements. If you are using other ES2015 features, make sure to use a transpiler such as Babel or Bublé via webpack's loader system.

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. 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

...
  Asset      Size  Chunks             Chunk Names
main.js  70.4 KiB       0  [emitted]  main
...

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value. S
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack
```

> *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 config 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 simple CLI usage. We can specify loader rules, plugins, resolve options and many other enhancements this way. See the configuration documentation 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:

package.json

```
  {
    "name": "webpack-demo",
    "version": "1.0.0",
    "description": "",
    "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": "^4.20.2",
      "webpack-cli": "^3.1.2"
    },
    "dependencies": {
      "lodash": "^4.17.5"
    }
  }
```

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 (each with flags like `--config` if necessary).

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

```
npm run build

...
  Asset      Size  Chunks             Chunk Names
main.js  70.4 KiB       0  [emitted]  main
...

WARNING in configuration
The 'mode' option has not been set, webpack will fallback to 'production' for this value. S
You can also set it to 'none' to disable any default behavior. Learn more: https://webpack
```

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 -- --colors` .

Conclusion

Now that you have a basic build together you should move on to the next guide `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
```

If you're using npm 5, you'll probably also see a `package-lock.json` file in your directory.

If you want to learn more about webpack's design, you can check out the basic concepts and configuration pages. Furthermore, the API section digs into the various interfaces webpack offers.