Printable Guides

This section contains guides for understanding and mastering the wide variety of tools and features that webpack offers. The first is a guide that takes you through getting started.

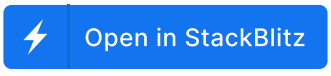
The guides get more advanced as you go on. Most serve as a starting point, and once completed you should feel more comfortable diving into the actual documentation.

> ### Warning
>
> The output shown from running webpack in the guides may differ slightly from the output of newer versions. This is to be expected. As long as the bundles look similar and run correctly, then there shouldn't be any issues. If you do come across an example that seems to be broken by a new version, please create an issue and we will do our best to resolve the discrepancy.

Getting Started

Webpack is used to compile JavaScript modules. Once installed, you can interact 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.

> ### Warning
>
> The minimum supported Node.js version to run webpack 5 is 10.13.0 (LTS)

> ### Live Preview
>
> Check out this guide live on StackBlitz.
>
> 
>

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. 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
  |- package-lock.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.

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
  |- package-lock.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, 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.

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, 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
[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` and `export` statements have been standardized in ES2015. 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 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 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
  |- package-lock.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 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": "",
   "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` 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
|- package-lock.json
|- webpack.config.js
|- /dist
  |- main.js
  |- index.html
|- /src
  |- index.js
|- /node_modules
```

> **Warning**
>
> Do not compile untrusted code with webpack. It could lead to execution of malicious code on your computer, remote servers, or in the Web browsers of the end users of your application.

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.

Asset Management

If you've been following the guides from the start, you will now have a small project that shows "Hello webpack". Now let's try to incorporate some other assets, like images, to see how they can be handled.

Prior to webpack, front-end developers would use tools like grunt and gulp 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). 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 support. This means that the same benefits listed above for JavaScript (e.g. explicit dependencies) can be applied to everything used in building a website or web app. Let's start with CSS, as you may already be familiar with that setup.

Setup

Let's make a minor change to our project before we get started:

dist/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
-   <title>Getting Started</title>
+   <title>Asset Management</title>
  </head>
  <body>
-   <script src="main.js"></script>
+   <script src="bundle.js"></script>
  </body>
</html>
```

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
-   filename: 'main.js',
+   filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

Loading CSS

In order to `import` a CSS file from within a JavaScript module, you need to install and add the style-loader and css-loader to your `module` configuration:

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

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
+ module: {
+   rules: [
+     {
```

```
+        test: /\.css$/i,
+        use: ['style-loader', 'css-loader'],
+      },
+    ],
+  },
 };
```

Module loaders can be chained. Each loader in the chain applies transformations to the processed resource. A chain is executed in reverse order. The first loader passes its result (resource with applied transformations) to the next one, and so forth. Finally, webpack expects JavaScript to be returned by the last loader in the chain.

The above order of loaders should be maintained: `'style-loader'` comes first and followed by `'css-loader'` . If this convention is not followed, webpack is likely to throw errors.

> **Tip**
>
> webpack uses a regular expression to determine which files it should look for and serve to a specific loader. In this case, any file that ends with `.css` will be served to the `style-loader` and the `css-loader` .

This enables you to `import './style.css'` into the file that depends on that styling. Now, when that module is run, a `<style>` tag with the stringified css will be inserted into the `<head>` of your html file.

Let's try it out by adding a new `style.css` file to our project and import it in our `index.js` :

project

```
  webpack-demo
  |- package.json
  |- package-lock.json
  |- webpack.config.js
  |- /dist
    |- bundle.js
    |- index.html
  |- /src
+   |- style.css
    |- index.js
  |- /node_modules
```

src/style.css

```
.hello {
  color: red;
}
```

src/index.js

```
 import _ from 'lodash';
+import './style.css';

 function component() {
   const element = document.createElement('div');

   // Lodash, now imported by this script
   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+  element.classList.add('hello');

   return element;
 }

 document.body.appendChild(component());
```

Now run your build command:

```
$ npm run build

...
[webpack-cli] Compilation finished
asset bundle.js 72.6 KiB [emitted] [minimized] (name: main) 1 related asset
runtime modules 1000 bytes 5 modules
orphan modules 326 bytes [orphan] 1 module
cacheable modules 539 KiB
  modules by path ./node_modules/ 538 KiB
    ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
    ./node_modules/style-loader/dist/runtime/injectStylesIntoStyleTag.js 6.67 KiB [built] [cc
    ./node_modules/css-loader/dist/runtime/api.js 1.57 KiB [built] [code generated]
  modules by path ./src/ 965 bytes
    ./src/index.js + 1 modules 639 bytes [built] [code generated]
    ./node_modules/css-loader/dist/cjs.js!./src/style.css 326 bytes [built] [code generated]
webpack 5.4.0 compiled successfully in 2231 ms
```

Open up `dist/index.html` in your browser again and you should see that `Hello webpack` is now styled in red. To see what webpack did, inspect the page (don't view the page source, as it won't show you the result, because the `<style>` tag is dynamically created by JavaScript) and look at the page's head tags. It should contain the style block that we imported in `index.js`.

Note that you can, and in most cases should, minimize css for better load times in production. On top of that, loaders exist for pretty much any flavor of CSS you can think of – postcss, sass, and less to name a few.

Loading Images

So now we're pulling in our CSS, but what about our images like backgrounds and icons? As of webpack 5, using the built-in Asset Modules we can easily incorporate those in our system as well:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: ['style-loader', 'css-loader'],
      },
+     {
+       test: /\.(png|svg|jpg|jpeg|gif)$/i,
+       type: 'asset/resource',
+     },
    ],
  },
};
```

Now, when you `import MyImage from './my-image.png'`, that image will be processed and added to your `output` directory *and* the `MyImage` variable will contain the final url of that image after processing. When using the css-loader, as shown above, a similar process will occur for `url('./my-image.png')` within your CSS. The loader will recognize this is a local file, and replace the `'./my-image.png'` path with the final path to the image in your `output` directory. The html-loader handles `<img src="./my-image.png" />` in the same manner.

Let's add an image to our project and see how this works, you can use any image you like:

project

```
 webpack-demo
 |- package.json
 |- package-lock.json
 |- webpack.config.js
 |- /dist
   |- bundle.js
   |- index.html
 |- /src
+  |- icon.png
   |- style.css
   |- index.js
```

```
  |- /node_modules
```

src/index.js

```
 import _ from 'lodash';
 import './style.css';
+import Icon from './icon.png';

 function component() {
   const element = document.createElement('div');

   // Lodash, now imported by this script
   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
   element.classList.add('hello');

+  // Add the image to our existing div.
+  const myIcon = new Image();
+  myIcon.src = Icon;
+
+  element.appendChild(myIcon);
+
   return element;
 }

 document.body.appendChild(component());
```

src/style.css

```
 .hello {
   color: red;
+  background: url('./icon.png');
 }
```

Let's create a new build and open up the `index.html` file again:

```
$ npm run build

...
[webpack-cli] Compilation finished
assets by status 9.88 KiB [cached] 1 asset
asset bundle.js 73.4 KiB [emitted] [minimized] (name: main) 1 related asset
runtime modules 1.82 KiB 6 modules
orphan modules 326 bytes [orphan] 1 module
cacheable modules 540 KiB (javascript) 9.88 KiB (asset)
  modules by path ./node_modules/ 539 KiB
    modules by path ./node_modules/css-loader/dist/runtime/*.js 2.38 KiB
      ./node_modules/css-loader/dist/runtime/api.js 1.57 KiB [built] [code generated]
      ./node_modules/css-loader/dist/runtime/getUrl.js 830 bytes [built] [code generated]
    ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
    ./node_modules/style-loader/dist/runtime/injectStylesIntoStyleTag.js 6.67 KiB [built] [cc
```

```
  modules by path ./src/ 1.45 KiB (javascript) 9.88 KiB (asset)
    ./src/index.js + 1 modules 794 bytes [built] [code generated]
    ./src/icon.png 42 bytes (javascript) 9.88 KiB (asset) [built] [code generated]
    ./node_modules/css-loader/dist/cjs.js!./src/style.css 648 bytes [built] [code generated]
webpack 5.4.0 compiled successfully in 1972 ms
```

If all went well, you should now see your icon as a repeating background, as well as an `img` element beside our `Hello webpack` text. If you inspect this element, you'll see that the actual filename has changed to something like `29822eaa871e8eadeaa4.png`. This means webpack found our file in the `src` folder and processed it!

Loading Fonts

So what about other assets like fonts? The Asset Modules will take any file you load through them and output it to your build directory. This means we can use them for any kind of file, including fonts. Let's update our `webpack.config.js` to handle font files:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.(png|svg|jpg|jpeg|gif)$/i,
        type: 'asset/resource',
      },
+     {
+       test: /\.(woff|woff2|eot|ttf|otf)$/i,
+       type: 'asset/resource',
+     },
    ],
  },
};
```

Add some font files to your project:

project

```
  webpack-demo
  |- package.json
  |- package-lock.json
  |- webpack.config.js
  |- /dist
    |- bundle.js
    |- index.html
  |- /src
+   |- my-font.woff
+   |- my-font.woff2
    |- icon.png
    |- style.css
    |- index.js
  |- /node_modules
```

With the loader configured and fonts in place, you can incorporate them via an `@font-face` declaration. The local `url(...)` directive will be picked up by webpack, as it was with the image:

src/style.css

```
+@font-face {
+  font-family: 'MyFont';
+  src: url('./my-font.woff2') format('woff2'),
+    url('./my-font.woff') format('woff');
+  font-weight: 600;
+  font-style: normal;
+}
+
 .hello {
    color: red;
+   font-family: 'MyFont';
    background: url('./icon.png');
 }
```

Now run a new build and let's see if webpack handled our fonts:

```
$ npm run build

...
[webpack-cli] Compilation finished
assets by status 9.88 KiB [cached] 1 asset
assets by info 33.2 KiB [immutable]
  asset 55055dbfc7c6a83f60ba.woff 18.8 KiB [emitted] [immutable] [from: src/my-font.woff] (au
  asset 8f717b802eaab4d7fb94.woff2 14.5 KiB [emitted] [immutable] [from: src/my-font.woff2] (
asset bundle.js 73.7 KiB [emitted] [minimized] (name: main) 1 related asset
runtime modules 1.82 KiB 6 modules
orphan modules 326 bytes [orphan] 1 module
cacheable modules 541 KiB (javascript) 43.1 KiB (asset)
```

```
  javascript modules 541 KiB
    modules by path ./node_modules/ 539 KiB
      modules by path ./node_modules/css-loader/dist/runtime/*.js 2.38 KiB 2 modules
      ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
      ./node_modules/style-loader/dist/runtime/injectStylesIntoStyleTag.js 6.67 KiB [built] [
    modules by path ./src/ 1.98 KiB
      ./src/index.js + 1 modules 794 bytes [built] [code generated]
      ./node_modules/css-loader/dist/cjs.js!./src/style.css 1.21 KiB [built] [code generated]
  asset modules 126 bytes (javascript) 43.1 KiB (asset)
    ./src/icon.png 42 bytes (javascript) 9.88 KiB (asset) [built] [code generated]
    ./src/my-font.woff2 42 bytes (javascript) 14.5 KiB (asset) [built] [code generated]
    ./src/my-font.woff 42 bytes (javascript) 18.8 KiB (asset) [built] [code generated]
webpack 5.4.0 compiled successfully in 2142 ms
```

Open up `dist/index.html` again and see if our `Hello webpack` text has changed to the new font. If all is well, you should see the changes.

Loading Data

Another useful asset that can be loaded is data, like JSON files, CSVs, TSVs, and XML. Support for JSON is actually built-in, similar to NodeJS, meaning `import Data from './data.json'` will work by default. To import CSVs, TSVs, and XML you could use the csv-loader and xml-loader. Let's handle loading all three:

```
npm install --save-dev csv-loader xml-loader
```

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  module: {
    rules: [
      {
        test: /\.css$/i,
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.(png|svg|jpg|jpeg|gif)$/i,
        type: 'asset/resource',
      },
      {
```

```
          test: /\.(woff|woff2|eot|ttf|otf)$/i,
          type: 'asset/resource',
        },
+       {
+         test: /\.(csv|tsv)$/i,
+         use: ['csv-loader'],
+       },
+       {
+         test: /\.xml$/i,
+         use: ['xml-loader'],
+       },
      ],
    },
  };
```

Add some data files to your project:

project

```
  webpack-demo
  |- package.json
  |- package-lock.json
  |- webpack.config.js
  |- /dist
    |- bundle.js
    |- index.html
  |- /src
+   |- data.xml
+   |- data.csv
    |- my-font.woff
    |- my-font.woff2
    |- icon.png
    |- style.css
    |- index.js
  |- /node_modules
```

src/data.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Mary</to>
  <from>John</from>
  <heading>Reminder</heading>
  <body>Call Cindy on Tuesday</body>
</note>
```

src/data.csv

```
to,from,heading,body
Mary,John,Reminder,Call Cindy on Tuesday
```

```
Zoe,Bill,Reminder,Buy orange juice
Autumn,Lindsey,Letter,I miss you
```

Now you can `import` any one of those four types of data (JSON, CSV, TSV, XML) and the `Data` variable you import, will contain parsed JSON for consumption:

src/index.js

```
 import _ from 'lodash';
 import './style.css';
 import Icon from './icon.png';
+import Data from './data.xml';
+import Notes from './data.csv';

 function component() {
   const element = document.createElement('div');

   // Lodash, now imported by this script
   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
   element.classList.add('hello');

   // Add the image to our existing div.
   const myIcon = new Image();
   myIcon.src = Icon;

   element.appendChild(myIcon);

+  console.log(Data);
+  console.log(Notes);
+
   return element;
 }

 document.body.appendChild(component());
```

Re-run the `npm run build` command and open `dist/index.html`. If you look at the console in your developer tools, you should be able to see your imported data being logged to the console!

> **Tip**
>
> This can be especially helpful when implementing some sort of data visualization using a tool like d3. Instead of making an ajax request and parsing the data at runtime you can load it into your module during the build process so that the parsed data is ready to go as soon as the module hits the browser.

> **Warning**

> Only the default export of JSON modules can be used without warning.

```
// No warning
import data from './data.json';

// Warning shown, this is not allowed by the spec.
import { foo } from './data.json';
```

Customize parser of JSON modules

It's possible to import any `toml`, `yaml` or `json5` files as a JSON module by using a custom parser instead of a specific webpack loader.

Let's say you have a `data.toml`, a `data.yaml` and a `data.json5` files under `src` folder:

src/data.toml

```
title = "TOML Example"

[owner]
name = "Tom Preston-Werner"
organization = "GitHub"
bio = "GitHub Cofounder & CEO\nLikes tater tots and beer."
dob = 1979-05-27T07:32:00Z
```

src/data.yaml

```
title: YAML Example
owner:
  name: Tom Preston-Werner
  organization: GitHub
  bio: |-
    GitHub Cofounder & CEO
    Likes tater tots and beer.
  dob: 1979-05-27T07:32:00.000Z
```

src/data.json5

```
{
  // comment
  title: 'JSON5 Example',
  owner: {
    name: 'Tom Preston-Werner',
    organization: 'GitHub',
    bio: 'GitHub Cofounder & CEO\n\
Likes tater tots and beer.',
    dob: '1979-05-27T07:32:00.000Z',
```

```
  },
}
```

Install `toml`, `yamljs` and `json5` packages first:

```
npm install toml yamljs json5 --save-dev
```

And configure them in your webpack configuration:

webpack.config.js

```
 const path = require('path');
+const toml = require('toml');
+const yaml = require('yamljs');
+const json5 = require('json5');

 module.exports = {
   entry: './src/index.js',
   output: {
     filename: 'bundle.js',
     path: path.resolve(__dirname, 'dist'),
   },
   module: {
     rules: [
       {
         test: /\.css$/i,
         use: ['style-loader', 'css-loader'],
       },
       {
         test: /\.(png|svg|jpg|jpeg|gif)$/i,
         type: 'asset/resource',
       },
       {
         test: /\.(woff|woff2|eot|ttf|otf)$/i,
         type: 'asset/resource',
       },
       {
         test: /\.(csv|tsv)$/i,
         use: ['csv-loader'],
       },
       {
         test: /\.xml$/i,
         use: ['xml-loader'],
       },
+      {
+        test: /\.toml$/i,
+        type: 'json',
+        parser: {
+          parse: toml.parse,
+        },
```

```
+      },
+      {
+        test: /\.yaml$/i,
+        type: 'json',
+        parser: {
+          parse: yaml.parse,
+        },
+      },
+      {
+        test: /\.json5$/i,
+        type: 'json',
+        parser: {
+          parse: json5.parse,
+        },
+      },
     ],
   },
 };
```

src/index.js

```
 import _ from 'lodash';
 import './style.css';
 import Icon from './icon.png';
 import Data from './data.xml';
 import Notes from './data.csv';
+import toml from './data.toml';
+import yaml from './data.yaml';
+import json from './data.json5';
+
+console.log(toml.title); // output `TOML Example`
+console.log(toml.owner.name); // output `Tom Preston-Werner`
+
+console.log(yaml.title); // output `YAML Example`
+console.log(yaml.owner.name); // output `Tom Preston-Werner`
+
+console.log(json.title); // output `JSON5 Example`
+console.log(json.owner.name); // output `Tom Preston-Werner`

 function component() {
   const element = document.createElement('div');

   // Lodash, now imported by this script
   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
   element.classList.add('hello');

   // Add the image to our existing div.
   const myIcon = new Image();
   myIcon.src = Icon;

   element.appendChild(myIcon);
```

```
  console.log(Data);
  console.log(Notes);

  return element;
}

document.body.appendChild(component());
```

Re-run the `npm run build` command and open `dist/index.html`. You should be able to see your imported data being logged to the console!

Global Assets

The coolest part of everything mentioned above, is that loading assets this way allows you to group modules and assets in a more intuitive way. Instead of relying on a global `/assets` directory that contains everything, you can group assets with the code that uses them. For example, a structure like this can be useful:

```
- |- /assets
+ |- /components
+ |  |- /my-component
+ |  |  |- index.jsx
+ |  |  |- index.css
+ |  |  |- icon.svg
+ |  |  |- img.png
```

This setup makes your code a lot more portable as everything that is closely coupled now lives together. Let's say you want to use `/my-component` in another project, copy or move it into the `/components` directory over there. As long as you've installed any *external dependencies* and your *configuration has the same loaders* defined, you should be good to go.

However, let's say you're locked into your old ways or you have some assets that are shared between multiple components (views, templates, modules, etc.). It's still possible to store these assets in a base directory and even use aliasing to make them easier to `import`.

Wrapping up

For the next guides we won't be using all the different assets we've used in this guide, so let's do some cleanup so we're prepared for the next piece of the guides Output Management:

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
-   |- data.csv
-   |- data.json5
-   |- data.toml
-   |- data.xml
-   |- data.yaml
-   |- icon.png
-   |- my-font.woff
-   |- my-font.woff2
-   |- style.css
    |- index.js
|- /node_modules
```

webpack.config.js

```
const path = require('path');
-const toml = require('toml');
-const yaml = require('yamljs');
-const json5 = require('json5');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
-  module: {
-    rules: [
-      {
-        test: /\.css$/i,
-        use: ['style-loader', 'css-loader'],
-      },
-      {
-        test: /\.(png|svg|jpg|jpeg|gif)$/i,
-        type: 'asset/resource',
-      },
-      {
-        test: /\.(woff|woff2|eot|ttf|otf)$/i,
-        type: 'asset/resource',
-      },
-      {
-        test: /\.(csv|tsv)$/i,
-        use: ['csv-loader'],
-      },
```

```
-      {
-        test: /\.xml$/i,
-        use: ['xml-loader'],
-      },
-      {
-        test: /\.toml$/i,
-        type: 'json',
-        parser: {
-          parse: toml.parse,
-        },
-      },
-      {
-        test: /\.yaml$/i,
-        type: 'json',
-        parser: {
-          parse: yaml.parse,
-        },
-      },
-      {
-        test: /\.json5$/i,
-        type: 'json',
-        parser: {
-          parse: json5.parse,
-        },
-      },
-    ],
-  },
 };
```

src/index.js

```
 import _ from 'lodash';
-import './style.css';
-import Icon from './icon.png';
-import Data from './data.xml';
-import Notes from './data.csv';
-import toml from './data.toml';
-import yaml from './data.yaml';
-import json from './data.json5';
-
-console.log(toml.title); // output `TOML Example`
-console.log(toml.owner.name); // output `Tom Preston-Werner`
-
-console.log(yaml.title); // output `YAML Example`
-console.log(yaml.owner.name); // output `Tom Preston-Werner`
-
-console.log(json.title); // output `JSON5 Example`
-console.log(json.owner.name); // output `Tom Preston-Werner`

 function component() {
   const element = document.createElement('div');
```

```
-  // Lodash, now imported by this script
   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-  element.classList.add('hello');
-
-  // Add the image to our existing div.
-  const myIcon = new Image();
-  myIcon.src = Icon;
-
-  element.appendChild(myIcon);
-
-  console.log(Data);
-  console.log(Notes);

   return element;
 }

 document.body.appendChild(component());
```

And remove those dependencies we added before:

```
npm uninstall css-loader csv-loader json5 style-loader toml xml-loader yamljs
```

Next guide

Let's move on to Output Management

Further Reading

- Loading Fonts on SurviveJS

Output Management

> **Tip**
>
> This guide extends on code examples found in the `Asset Management` guide.

So far we've manually included all our assets in our `index.html` file, but as your application grows and once you start using hashes in filenames and outputting multiple bundles, it will be difficult to keep managing your `index.html` file manually. However, a few plugins exist that will make this process much easier to manage.

Preparation

First, let's adjust our project a little bit:

project

```
  webpack-demo
  |- package.json
  |- package-lock.json
  |- webpack.config.js
  |- /dist
  |- /src
    |- index.js
+   |- print.js
  |- /node_modules
```

Let's add some logic to our `src/print.js` file:

src/print.js

```
export default function printMe() {
  console.log('I get called from print.js!');
}
```

And use that function in our `src/index.js` file:

src/index.js

```
 import _ from 'lodash';
+import printMe from './print.js';

 function component() {
   const element = document.createElement('div');
+  const btn = document.createElement('button');

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

+  btn.innerHTML = 'Click me and check the console!';
+  btn.onclick = printMe;
+
+  element.appendChild(btn);
+
   return element;
 }

 document.body.appendChild(component());
```

Let's also update our `dist/index.html` file, in preparation for webpack to split out entries:

dist/index.html

```
 <!DOCTYPE html>
 <html>
   <head>
     <meta charset="utf-8" />
-    <title>Asset Management</title>
+    <title>Output Management</title>
+    <script src="./print.bundle.js"></script>
   </head>
   <body>
-    <script src="bundle.js"></script>
+    <script src="./index.bundle.js"></script>
   </body>
 </html>
```

Now adjust the config. We'll be adding our `src/print.js` as a new entry point (`print`) and we'll change the output as well, so that it will dynamically generate bundle names, based on the entry point names:

webpack.config.js

```
 const path = require('path');

 module.exports = {
-  entry: './src/index.js',
+  entry: {
+    index: './src/index.js',
+    print: './src/print.js',
+  },
   output: {
-    filename: 'bundle.js',
+    filename: '[name].bundle.js',
     path: path.resolve(__dirname, 'dist'),
   },
 };
```

Let's run `npm run build` and see what this generates:

```
...
[webpack-cli] Compilation finished
asset index.bundle.js 69.5 KiB [emitted] [minimized] (name: index) 1 related asset
asset print.bundle.js 316 bytes [emitted] [minimized] (name: print)
runtime modules 1.36 KiB 7 modules
cacheable modules 530 KiB
  ./src/index.js 406 bytes [built] [code generated]
  ./src/print.js 83 bytes [built] [code generated]
  ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
webpack 5.4.0 compiled successfully in 1996 ms
```

We can see that webpack generates our `print.bundle.js` and `index.bundle.js` files, which we also specified in our `index.html` file. if you open `index.html` in your browser, you can see what happens when you click the button.

But what would happen if we changed the name of one of our entry points, or even added a new one? The generated bundles would be renamed on a build, but our `index.html` file would still reference the old names. Let's fix that with the `HtmlWebpackPlugin` .

Setting up HtmlWebpackPlugin

First install the plugin and adjust the `webpack.config.js` file:

```
npm install --save-dev html-webpack-plugin
```

webpack.config.js

```
 const path = require('path');
+const HtmlWebpackPlugin = require('html-webpack-plugin');

 module.exports = {
   entry: {
     index: './src/index.js',
     print: './src/print.js',
   },
+  plugins: [
+    new HtmlWebpackPlugin({
+      title: 'Output Management',
+    }),
+  ],
   output: {
     filename: '[name].bundle.js',
     path: path.resolve(__dirname, 'dist'),
   },
 };
```

Before we do a build, you should know that the `HtmlWebpackPlugin` by default will generate its own `index.html` file, even though we already have one in the `dist/` folder. This means that it will replace our `index.html` file with a newly generated one. Let's see what happens when we do an `npm run build` :

```
...
[webpack-cli] Compilation finished
asset index.bundle.js 69.5 KiB [compared for emit] [minimized] (name: index) 1 related asset
asset print.bundle.js 316 bytes [compared for emit] [minimized] (name: print)
asset index.html 253 bytes [emitted]
runtime modules 1.36 KiB 7 modules
```

```
  cacheable modules 530 KiB
    ./src/index.js 406 bytes [built] [code generated]
    ./src/print.js 83 bytes [built] [code generated]
    ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
  webpack 5.4.0 compiled successfully in 2189 ms
```

If you open `index.html` in your code editor, you'll see that the `HtmlWebpackPlugin` has created an entirely new file for you and that all the bundles are automatically added.

If you want to learn more about all the features and options that the `HtmlWebpackPlugin` provides, then you should read up on it on the `HtmlWebpackPlugin` repo.

Cleaning up the `/dist` folder

As you might have noticed over the past guides and code example, our `/dist` folder has become quite cluttered. Webpack will generate the files and put them in the `/dist` folder for you, but it doesn't keep track of which files are actually in use by your project.

In general it's good practice to clean the `/dist` folder before each build, so that only used files will be generated. Let's take care of that with `output.clean` option.

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');

  module.exports = {
    entry: {
      index: './src/index.js',
      print: './src/print.js',
    },
    plugins: [
      new HtmlWebpackPlugin({
        title: 'Output Management',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
+     clean: true,
    },
  };
```

Now run an `npm run build` and inspect the `/dist` folder. If everything went well you should now only see the files generated from the build and no more old files!

The Manifest

You might be wondering how webpack and its plugins seem to "know" what files are being generated. The answer is in the manifest that webpack keeps to track how all the modules map to the output bundles. If you're interested in managing webpack's `output` in other ways, the manifest would be a good place to start.

The manifest data can be extracted into a json file for consumption using the `WebpackManifestPlugin`.

We won't go through a full example of how to use this plugin within your projects, but you can read up on the concept page and the caching guide to find out how this ties into long term caching.

Conclusion

Now that you've learned about dynamically adding bundles to your HTML, let's dive into the development guide. Or, if you want to dig into more advanced topics, we would recommend heading over to the code splitting guide.

Development

> **Tip**
>
> This guide extends on code examples found in the Output Management guide.

If you've been following the guides, you should have a solid understanding of some of the webpack basics. Before we continue, let's look into setting up a development environment to make our lives a little easier.

> **Warning**
>
> The tools in this guide are **only meant for development**, please **avoid** using them in production!

Let's start by setting `mode to 'development'` and `title to 'Development'`.

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
```

```
 module.exports = {
+  mode: 'development',
   entry: {
     index: './src/index.js',
     print: './src/print.js',
   },
   plugins: [
     new HtmlWebpackPlugin({
-      title: 'Output Management',
+      title: 'Development',
     }),
   ],
   output: {
     filename: '[name].bundle.js',
     path: path.resolve(__dirname, 'dist'),
     clean: true,
   },
 };
```

Using source maps

When webpack bundles your source code, it can become difficult to track down errors and warnings to their original location. For example, if you bundle three source files (`a.js` , `b.js` , and `c.js`) into one bundle (`bundle.js`) and one of the source files contains an error, the stack trace will point to `bundle.js` . This isn't always helpful as you probably want to know exactly which source file the error came from.

In order to make it easier to track down errors and warnings, JavaScript offers source maps, which map your compiled code back to your original source code. If an error originates from `b.js` , the source map will tell you exactly that.

There are a lot of different options available when it comes to source maps. Be sure to check them out so you can configure them to your needs.

For this guide, let's use the `inline-source-map` option, which is good for illustrative purposes (though not for production):

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  mode: 'development',
  entry: {
    index: './src/index.js',
```

```
    print: './src/print.js',
  },
+ devtool: 'inline-source-map',
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Development',
    }),
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
};
```

Now let's make sure we have something to debug, so let's create an error in our `print.js` file:

src/print.js

```
 export default function printMe() {
-  console.log('I get called from print.js!');
+  cosnole.log('I get called from print.js!');
 }
```

Run an `npm run build`, it should compile to something like this:

```
...
[webpack-cli] Compilation finished
asset index.bundle.js 1.38 MiB [emitted] (name: index)
asset print.bundle.js 6.25 KiB [emitted] (name: print)
asset index.html 272 bytes [emitted]
runtime modules 1.9 KiB 9 modules
cacheable modules 530 KiB
  ./src/index.js 406 bytes [built] [code generated]
  ./src/print.js 83 bytes [built] [code generated]
  ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
webpack 5.4.0 compiled successfully in 706 ms
```

Now open the resulting `index.html` file in your browser. Click the button and look in your console where the error is displayed. The error should say something like this:

```
Uncaught ReferenceError: cosnole is not defined
   at HTMLButtonElement.printMe (print.js:2)
```

We can see that the error also contains a reference to the file (`print.js`) and line number (2) where the error occurred. This is great because now we know exactly where to look in order to fix the issue.

Choosing a Development Tool

> **Warning**
>
> Some text editors have a "safe write" function that might interfere with some of the following tools. Read Adjusting Your Text Editor for a solution to these issues.

It quickly becomes a hassle to manually run `npm run build` every time you want to compile your code.

There are a couple of different options available in webpack that help you automatically compile your code whenever it changes:

1. webpack's Watch Mode
2. webpack-dev-server
3. webpack-dev-middleware

In most cases, you probably would want to use `webpack-dev-server`, but let's explore all of the above options.

Using Watch Mode

You can instruct webpack to "watch" all files within your dependency graph for changes. If one of these files is updated, the code will be recompiled so you don't have to run the full build manually.

Let's add an npm script that will start webpack's Watch Mode:

package.json

```
 {
   "name": "webpack-demo",
   "version": "1.0.0",
   "description": "",
   "private": true,
   "scripts": {
     "test": "echo \"Error: no test specified\" && exit 1",
+    "watch": "webpack --watch",
     "build": "webpack"
   },
   "keywords": [],
   "author": "",
   "license": "ISC",
   "devDependencies": {
     "html-webpack-plugin": "^4.5.0",
     "webpack": "^5.4.0",
```

```
    "webpack-cli": "^4.2.0"
  },
  "dependencies": {
    "lodash": "^4.17.20"
  }
}
```

Now run `npm run watch` from the command line and see how webpack compiles your code. You can see that it doesn't exit the command line because the script is currently watching your files.

Now, while webpack is watching your files, let's remove the error we introduced earlier:

src/print.js

```
export default function printMe() {
-  cosnole.log('I get called from print.js!');
+  console.log('I get called from print.js!');
}
```

Now save your file and check the terminal window. You should see that webpack automatically recompiles the changed module!

The only downside is that you have to refresh your browser in order to see the changes. It would be much nicer if that would happen automatically as well, so let's try `webpack-dev-server` which will do exactly that.

Using webpack-dev-server

The `webpack-dev-server` provides you with a rudimentary web server and the ability to use live reloading. Let's set it up:

```
npm install --save-dev webpack-dev-server
```

Change your configuration file to tell the dev server where to look for files:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  mode: 'development',
  entry: {
    index: './src/index.js',
    print: './src/print.js',
  },
  devtool: 'inline-source-map',
```

```
+  devServer: {
+    static: './dist',
+  },
   plugins: [
     new HtmlWebpackPlugin({
       title: 'Development',
     }),
   ],
   output: {
     filename: '[name].bundle.js',
     path: path.resolve(__dirname, 'dist'),
     clean: true,
   },
+  optimization: {
+    runtimeChunk: 'single',
+  },
 };
```

This tells `webpack-dev-server` to serve the files from the `dist` directory on `localhost:8080`.

> **Tip**
>
> The `optimization.runtimeChunk: 'single'` was added because in this example we have more than one entrypoint on a single HTML page. Without this, we could get into trouble described here. Read the Code Splitting chapter for more details.

> **Tip**
>
> `webpack-dev-server` serves bundled files from the directory defined in `output.path`, i.e., files will be available under `http://[devServer.host]:[devServer.port]/[output.publicPath]/[output.filename]`.

> **Warning**
>
> webpack-dev-server doesn't write any output files after compiling. Instead, it keeps bundle files in memory and serves them as if they were real files mounted at the server's root path. If your page expects to find the bundle files on a different path, you can change this with the `devMiddleware.publicPath` option in the dev server's configuration.

Let's add a script to easily run the dev server as well:

package.json

```
{
```

```
    "name": "webpack-demo",
    "version": "1.0.0",
    "description": "",
    "private": true,
    "scripts": {
      "test": "echo \"Error: no test specified\" && exit 1",
      "watch": "webpack --watch",
+     "start": "webpack serve --open",
      "build": "webpack"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "html-webpack-plugin": "^4.5.0",
      "webpack": "^5.4.0",
      "webpack-cli": "^4.2.0",
      "webpack-dev-server": "^3.11.0"
    },
    "dependencies": {
      "lodash": "^4.17.20"
    }
  }
```

Now we can run `npm start` from the command line and we will see our browser automatically loading up our page. If you now change any of the source files and save them, the web server will automatically reload after the code has been compiled. Give it a try!

The `webpack-dev-server` comes with many configurable options. Head over to the documentation to learn more.

> **Tip**
>
> Now that your server is working, you might want to give Hot Module Replacement a try!

Using webpack-dev-middleware

`webpack-dev-middleware` is a wrapper that will emit files processed by webpack to a server. This is used in `webpack-dev-server` internally, however it's available as a separate package to allow more custom setups if desired. We'll take a look at an example that combines `webpack-dev-middleware` with an express server.

Let's install `express` and `webpack-dev-middleware` so we can get started:

```
npm install --save-dev express webpack-dev-middleware
```

Now we need to make some adjustments to our webpack configuration file in order to make sure the middleware will function correctly:

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  mode: 'development',
  entry: {
    index: './src/index.js',
    print: './src/print.js',
  },
  devtool: 'inline-source-map',
  devServer: {
    static: './dist',
  },
  plugins: [
    new HtmlWebpackPlugin({
      title: 'Development',
    }),
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
+   publicPath: '/',
  },
};
```

The `publicPath` will be used within our server script as well in order to make sure files are served correctly on `http://localhost:3000`. We'll specify the port number later. The next step is setting up our custom `express` server:

project

```
 webpack-demo
 |- package.json
 |- package-lock.json
 |- webpack.config.js
+|- server.js
 |- /dist
 |- /src
   |- index.js
   |- print.js
 |- /node_modules
```

server.js

```
const express = require('express');
const webpack = require('webpack');
const webpackDevMiddleware = require('webpack-dev-middleware');

const app = express();
const config = require('./webpack.config.js');
const compiler = webpack(config);

// Tell express to use the webpack-dev-middleware and use the webpack.config.js
// configuration file as a base.
app.use(
  webpackDevMiddleware(compiler, {
    publicPath: config.output.publicPath,
  })
);

// Serve the files on port 3000.
app.listen(3000, function () {
  console.log('Example app listening on port 3000!\n');
});
```

Now add an npm script to make it a little easier to run the server:

package.json

```
 {
   "name": "webpack-demo",
   "version": "1.0.0",
   "description": "",
   "private": true,
   "scripts": {
     "test": "echo \"Error: no test specified\" && exit 1",
     "watch": "webpack --watch",
     "start": "webpack serve --open",
+    "server": "node server.js",
     "build": "webpack"
   },
   "keywords": [],
   "author": "",
   "license": "ISC",
   "devDependencies": {
     "express": "^4.17.1",
     "html-webpack-plugin": "^4.5.0",
     "webpack": "^5.4.0",
     "webpack-cli": "^4.2.0",
     "webpack-dev-middleware": "^4.0.2",
     "webpack-dev-server": "^3.11.0"
   },
   "dependencies": {
     "lodash": "^4.17.20"
   }
```

```
}
```

Now in your terminal run `npm run server`, it should give you an output similar to this:

```
Example app listening on port 3000!
...
<i> [webpack-dev-middleware] asset index.bundle.js 1.38 MiB [emitted] (name: index)
<i> asset print.bundle.js 6.25 KiB [emitted] (name: print)
<i> asset index.html 274 bytes [emitted]
<i> runtime modules 1.9 KiB 9 modules
<i> cacheable modules 530 KiB
<i>   ./src/index.js 406 bytes [built] [code generated]
<i>   ./src/print.js 83 bytes [built] [code generated]
<i>   ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
<i> webpack 5.4.0 compiled successfully in 709 ms
<i> [webpack-dev-middleware] Compiled successfully.
<i> [webpack-dev-middleware] Compiling...
<i> [webpack-dev-middleware] assets by status 1.38 MiB [cached] 2 assets
<i> cached modules 530 KiB (javascript) 1.9 KiB (runtime) [cached] 12 modules
<i> webpack 5.4.0 compiled successfully in 19 ms
<i> [webpack-dev-middleware] Compiled successfully.
```

Now fire up your browser and go to `http://localhost:3000`. You should see your webpack app running and functioning!

> **Tip**
>
> If you would like to know more about how Hot Module Replacement works, we recommend you read the Hot Module Replacement guide.

Adjusting Your Text Editor

When using automatic compilation of your code, you could run into issues when saving your files. Some editors have a "safe write" feature that can potentially interfere with recompilation.

To disable this feature in some common editors, see the list below:

- **Sublime Text 3**: Add `atomic_save: 'false'` to your user preferences.
- **JetBrains IDEs (e.g. WebStorm)**: Uncheck "Use safe write" in `Preferences > Appearance & Behavior > System Settings`.
- **Vim**: Add `:set backupcopy=yes` to your settings.

Conclusion

Now that you've learned how to automatically compile your code and run a development server, you can check out the next guide, which will cover Code Splitting.

Code Splitting

> **Tip**
>
> This guide extends the example provided in Getting Started. Please make sure you are at least familiar with the example provided there and the Output Management chapter.

Code splitting is one of the most compelling features of webpack. This feature allows you to split your code into various bundles which can then be loaded on demand or in parallel. It can be used to achieve smaller bundles and control resource load prioritization which, if used correctly, can have a major impact on load time.

There are three general approaches to code splitting available:

- **Entry Points**: Manually split code using `entry` configuration.
- **Prevent Duplication**: Use Entry dependencies or `SplitChunksPlugin` to dedupe and split chunks.
- **Dynamic Imports**: Split code via inline function calls within modules.

Entry Points

This is by far the easiest and most intuitive way to split code. However, it is more manual and has some pitfalls we will go over. Let's take a look at how we might split another module from the main bundle:

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+ |- another-module.js
|- /node_modules
```

another-module.js

```
import _ from 'lodash';

console.log(_.join(['Another', 'module', 'loaded!'], ' '));
```

webpack.config.js

```
 const path = require('path');

 module.exports = {
-  entry: './src/index.js',
+  mode: 'development',
+  entry: {
+    index: './src/index.js',
+    another: './src/another-module.js',
+  },
   output: {
-    filename: 'main.js',
+    filename: '[name].bundle.js',
     path: path.resolve(__dirname, 'dist'),
   },
 };
```

This will yield the following build result:

```
...
[webpack-cli] Compilation finished
asset index.bundle.js 553 KiB [emitted] (name: index)
asset another.bundle.js 553 KiB [emitted] (name: another)
runtime modules 2.49 KiB 12 modules
cacheable modules 530 KiB
  ./src/index.js 257 bytes [built] [code generated]
  ./src/another-module.js 84 bytes [built] [code generated]
  ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
webpack 5.4.0 compiled successfully in 245 ms
```

As mentioned there are some pitfalls to this approach:

- If there are any duplicated modules between entry chunks they will be included in both bundles.
- It isn't as flexible and can't be used to dynamically split code with the core application logic.

The first of these two points is definitely an issue for our example, as `lodash` is also imported within `./src/index.js` and will thus be duplicated in both bundles. Let's remove this duplication in next section.

Prevent Duplication

Entry dependencies

The `dependOn` option allows to share the modules between the chunks:

webpack.config.js

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: {
-   index: './src/index.js',
-   another: './src/another-module.js',
+   index: {
+     import: './src/index.js',
+     dependOn: 'shared',
+   },
+   another: {
+     import: './src/another-module.js',
+     dependOn: 'shared',
+   },
+   shared: 'lodash',
  },
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

If we're going to use multiple entry points on a single HTML page, `optimization.runtimeChunk: 'single'` is needed too, otherwise we could get into trouble described here.

webpack.config.js

```
const path = require('path');

module.exports = {
  mode: 'development',
  entry: {
    index: {
      import: './src/index.js',
      dependOn: 'shared',
    },
    another: {
      import: './src/another-module.js',
      dependOn: 'shared',
```

```
      },
      shared: 'lodash',
    },
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
    },
+   optimization: {
+     runtimeChunk: 'single',
+   },
  };
```

And here's the result of build:

```
...
[webpack-cli] Compilation finished
asset shared.bundle.js 549 KiB [compared for emit] (name: shared)
asset runtime.bundle.js 7.79 KiB [compared for emit] (name: runtime)
asset index.bundle.js 1.77 KiB [compared for emit] (name: index)
asset another.bundle.js 1.65 KiB [compared for emit] (name: another)
Entrypoint index 1.77 KiB = index.bundle.js
Entrypoint another 1.65 KiB = another.bundle.js
Entrypoint shared 557 KiB = runtime.bundle.js 7.79 KiB shared.bundle.js 549 KiB
runtime modules 3.76 KiB 7 modules
cacheable modules 530 KiB
  ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
  ./src/another-module.js 84 bytes [built] [code generated]
  ./src/index.js 257 bytes [built] [code generated]
webpack 5.4.0 compiled successfully in 249 ms
```

As you can see there's another `runtime.bundle.js` file generated besides `shared.bundle.js`, `index.bundle.js` and `another.bundle.js`.

Although using multiple entry points per page is allowed in webpack, it should be avoided when possible in favor of an entry point with multiple imports: `entry: { page: ['./analytics', './app'] }`. This results in a better optimization and consistent execution order when using `async` script tags.

SplitChunksPlugin

The `SplitChunksPlugin` allows us to extract common dependencies into an existing entry chunk or an entirely new chunk. Let's use this to de-duplicate the `lodash` dependency from the previous example:

webpack.config.js

```
const path = require('path');
```

```
  module.exports = {
    mode: 'development',
    entry: {
      index: './src/index.js',
      another: './src/another-module.js',
    },
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
    },
+   optimization: {
+     splitChunks: {
+       chunks: 'all',
+     },
+   },
  };
```

With the `optimization.splitChunks` configuration option in place, we should now see the duplicate dependency removed from our `index.bundle.js` and `another.bundle.js`. The plugin should notice that we've separated `lodash` out to a separate chunk and remove the dead weight from our main bundle. Let's do an `npm run build` to see if it worked:

```
...
[webpack-cli] Compilation finished
asset vendors-node_modules_lodash_lodash_js.bundle.js 549 KiB [compared for emit] (id hint: v
asset index.bundle.js 8.92 KiB [compared for emit] (name: index)
asset another.bundle.js 8.8 KiB [compared for emit] (name: another)
Entrypoint index 558 KiB = vendors-node_modules_lodash_lodash_js.bundle.js 549 KiB index.bund
Entrypoint another 558 KiB = vendors-node_modules_lodash_lodash_js.bundle.js 549 KiB another.
runtime modules 7.64 KiB 14 modules
cacheable modules 530 KiB
  ./src/index.js 257 bytes [built] [code generated]
  ./src/another-module.js 84 bytes [built] [code generated]
  ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
webpack 5.4.0 compiled successfully in 241 ms
```

Here are some other useful plugins and loaders provided by the community for splitting code:

- `mini-css-extract-plugin` : Useful for splitting CSS out from the main application.

Dynamic Imports

Two similar techniques are supported by webpack when it comes to dynamic code splitting. The first and recommended approach is to use the `import()` syntax that conforms to the ECMAScript proposal for dynamic imports. The legacy, webpack-specific approach is to use `require.ensure`. Let's try using the first of these two approaches...

> ### Warning
>
> `import()` calls use promises internally. If you use `import()` with older browsers (e.g., IE 11), remember to shim `Promise` using a polyfill such as es6-promise or promise-polyfill.

Before we start, let's remove the extra `entry` and `optimization.splitChunks` from our configuration in the above example as they won't be needed for this next demonstration:

webpack.config.js

```
 const path = require('path');

 module.exports = {
   mode: 'development',
   entry: {
     index: './src/index.js',
-    another: './src/another-module.js',
   },
   output: {
     filename: '[name].bundle.js',
     path: path.resolve(__dirname, 'dist'),
   },
-  optimization: {
-    splitChunks: {
-      chunks: 'all',
-    },
-  },
 };
```

We'll also update our project to remove the now unused files:

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
- |- another-module.js
|- /node_modules
```

Now, instead of statically importing `lodash`, we'll use dynamic importing to separate a chunk:

src/index.js

```
-import _ from 'lodash';
```

```
-
-function component() {
+function getComponent() {
   const element = document.createElement('div');

-  // Lodash, now imported by this script
-  element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+  return import('lodash')
+    .then(({ default: _ }) => {
+      const element = document.createElement('div');
+
+      element.innerHTML = _.join(['Hello', 'webpack'], ' ');

-  return element;
+      return element;
+    })
+    .catch((error) => 'An error occurred while loading the component');
 }

-document.body.appendChild(component());
+getComponent().then((component) => {
+  document.body.appendChild(component);
+});
```

The reason we need `default` is that since webpack 4, when importing a CommonJS module, the import will no longer resolve to the value of `module.exports`, it will instead create an artificial namespace object for the CommonJS module. For more information on the reason behind this, read webpack 4: import() and CommonJs.

Let's run webpack to see `lodash` separated out to a separate bundle:

```
...
[webpack-cli] Compilation finished
asset vendors-node_modules_lodash_lodash_js.bundle.js 549 KiB [compared for emit] (id hint: v
asset index.bundle.js 13.5 KiB [compared for emit] (name: index)
runtime modules 7.37 KiB 11 modules
cacheable modules 530 KiB
  ./src/index.js 434 bytes [built] [code generated]
  ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
webpack 5.4.0 compiled successfully in 268 ms
```

As `import()` returns a promise, it can be used with `async` functions. Here's how it would simplify the code:

src/index.js

```
-function getComponent() {
+async function getComponent() {
   const element = document.createElement('div');
```

```
+  const { default: _ } = await import('lodash');

-  return import('lodash')
-    .then(({ default: _ }) => {
-      const element = document.createElement('div');
+  element.innerHTML = _.join(['Hello', 'webpack'], ' ');

-      element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-
-      return element;
-    })
-    .catch((error) => 'An error occurred while loading the component');
+  return element;
 }

 getComponent().then((component) => {
   document.body.appendChild(component);
 });
```

> **Tip**
>
> It is possible to provide a dynamic expression to `import()` when you might need to import specific module based on a computed variable later.

Prefetching/Preloading modules

Webpack 4.6.0+ adds support for prefetching and preloading.

Using these inline directives while declaring your imports allows webpack to output "Resource Hint" which tells the browser that for:

- **prefetch**: resource is probably needed for some navigation in the future
- **preload**: resource will also be needed during the current navigation

An example of this is having a `HomePage` component, which renders a `LoginButton` component which then on demand loads a `LoginModal` component after being clicked.

LoginButton.js

```
//...
import(/* webpackPrefetch: true */ './path/to/LoginModal.js');
```

This will result in `<link rel="prefetch" href="login-modal-chunk.js">` being appended in the head of the page, which will instruct the browser to prefetch in idle time the `login-modal-`

`chunk.js` file.

> **Tip**
>
> webpack will add the prefetch hint once the parent chunk has been loaded.

Preload directive has a bunch of differences compared to prefetch:

- A preloaded chunk starts loading in parallel to the parent chunk. A prefetched chunk starts after the parent chunk finishes loading.
- A preloaded chunk has medium priority and is instantly downloaded. A prefetched chunk is downloaded while the browser is idle.
- A preloaded chunk should be instantly requested by the parent chunk. A prefetched chunk can be used anytime in the future.
- Browser support is different.

An example of this can be having a `Component` which always depends on a big library that should be in a separate chunk.

Let's imagine a component `ChartComponent` which needs a huge `ChartingLibrary`. It displays a `LoadingIndicator` when rendered and instantly does an on demand import of `ChartingLibrary`:

ChartComponent.js

```
//...
import(/* webpackPreload: true */ 'ChartingLibrary');
```

When a page which uses the `ChartComponent` is requested, the charting-library-chunk is also requested via `<link rel="preload">`. Assuming the page-chunk is smaller and finishes faster, the page will be displayed with a `LoadingIndicator`, until the already requested `charting-library-chunk` finishes. This will give a little load time boost since it only needs one round-trip instead of two. Especially in high-latency environments.

> **Tip**
>
> Using `webpackPreload` incorrectly can actually hurt performance, so be careful when using it.

Sometimes you need to have your own control over preload. For example, preload of any dynamic import can be done via async script. This can be useful in case of streamming server side rendering.

```
const lazyComp = () =>
```

```
import('DynamicComponent').catch((error) => {
  // Do something with the error.
  // For example, we can retry the request in case of any net error
});
```

If the script loading will fail before webpack starts loading of that script by itself (webpack just creates a script tag to load its code, if that script is not on a page), that catch handler won't start till chunkLoadTimeout is not passed. This behavior can be unexpected. But it's explainable — webpack can not throw any error, cause webpack doesn't know, that script failed. Webpack will add onerror handler to the script right after the error has happen.

To prevent such problem you can add your own onerror handler, which removes the script in case of any error:

```
<script
  src="https://example.com/dist/dynamicComponent.js"
  async
  onerror="this.remove()"
></script>
```

In that case, errored script will be removed. Webpack will create its own script and any error will be processed without any timeouts.

Bundle Analysis

Once you start splitting your code, it can be useful to analyze the output to check where modules have ended up. The official analyze tool is a good place to start. There are some other community-supported options out there as well:

- webpack-chart: Interactive pie chart for webpack stats.
- webpack-visualizer: Visualize and analyze your bundles to see which modules are taking up space and which might be duplicates.
- webpack-bundle-analyzer: A plugin and CLI utility that represents bundle content as a convenient interactive zoomable treemap.
- webpack bundle optimize helper: This tool will analyze your bundle and give you actionable suggestions on what to improve to reduce your bundle size.
- bundle-stats: Generate a bundle report(bundle size, assets, modules) and compare the results between different builds.

Next Steps

See Lazy Loading for a more concrete example of how `import()` can be used in a real application and Caching to learn how to split code more effectively.

Caching

> **Tip**
>
> The examples in this guide stem from getting started, output management and code splitting.

So we're using webpack to bundle our modular application which yields a deployable `/dist` directory. Once the contents of `/dist` have been deployed to a server, clients (typically browsers) will hit that server to grab the site and its assets. The last step can be time consuming, which is why browsers use a technique called caching. This allows sites to load faster with less unnecessary network traffic. However, it can also cause headaches when you need new code to be picked up.

This guide focuses on the configuration needed to ensure files produced by webpack compilation can remain cached unless their content has changed.

Output Filenames

We can use the `output.filename` substitutions setting to define the names of our output files. Webpack provides a method of templating the filenames using bracketed strings called **substitutions**. The `[contenthash]` substitution will add a unique hash based on the content of an asset. When the asset's content changes, `[contenthash]` will change as well.

Let's get our project set up using the example from getting started with the `plugins` from output management, so we don't have to deal with maintaining our `index.html` file manually:

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
|- /node_modules
```

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
```

```
  module.exports = {
    entry: './src/index.js',
    plugins: [
      new HtmlWebpackPlugin({
-       title: 'Output Management',
+       title: 'Caching',
      }),
    ],
    output: {
-     filename: 'bundle.js',
+     filename: '[name].[contenthash].js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
  };
```

Running our build script, `npm run build`, with this configuration should produce the following output:

```
...
                      Asset       Size  Chunks                    Chunk Names
main.7e2c49a622975ebd9b7e.js     544 kB       0  [emitted]  [big]  main
                 index.html  197 bytes          [emitted]
...
```

As you can see the bundle's name now reflects its content (via the hash). If we run another build without making any changes, we'd expect that filename to stay the same. However, if we were to run it again, we may find that this is not the case:

```
...
                      Asset       Size  Chunks                    Chunk Names
main.205199ab45963f6a62ec.js     544 kB       0  [emitted]  [big]  main
                 index.html  197 bytes          [emitted]
...
```

This is because webpack includes certain boilerplate, specifically the runtime and manifest, in the entry chunk.

> ### Warning
>
> Output may differ depending on your current webpack version. Newer versions may not have all the same issues with hashing as some older versions, but we still recommend the following steps to be safe.

Extracting Boilerplate

As we learned in code splitting, the `SplitChunksPlugin` can be used to split modules out into separate bundles. Webpack provides an optimization feature to split runtime code into a separate chunk using the `optimization.runtimeChunk` option. Set it to `single` to create a single runtime bundle for all chunks:

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');

  module.exports = {
    entry: './src/index.js',
    plugins: [
      new HtmlWebpackPlugin({
      title: 'Caching',
      }),
    ],
    output: {
      filename: '[name].[contenthash].js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
+   optimization: {
+     runtimeChunk: 'single',
+   },
  };
```

Let's run another build to see the extracted `runtime` bundle:

```
Hash: 82c9c385607b2150fab2
Version: webpack 4.12.0
Time: 3027ms
                          Asset       Size  Chunks             Chunk Names
runtime.cc17ae2a94ec771e9221.js    1.42 KiB       0  [emitted]  runtime
   main.e81de2cf758ada72f306.js    69.5 KiB       1  [emitted]  main
                     index.html  275 bytes          [emitted]
[1] (webpack)/buildin/module.js 497 bytes {1} [built]
[2] (webpack)/buildin/global.js 489 bytes {1} [built]
[3] ./src/index.js 309 bytes {1} [built]
    + 1 hidden module
```

It's also good practice to extract third-party libraries, such as `lodash` or `react`, to a separate `vendor` chunk as they are less likely to change than our local source code. This step will allow clients to request even less from the server to stay up to date. This can be done by using the `cacheGroups` option of the `SplitChunksPlugin` demonstrated in Example 2 of SplitChunksPlugin. Lets add `optimization.splitChunks` with `cacheGroups` with next

params and build:

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');

  module.exports = {
    entry: './src/index.js',
    plugins: [
      new HtmlWebpackPlugin({
      title: 'Caching',
      }),
    ],
    output: {
      filename: '[name].[contenthash].js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
    optimization: {
      runtimeChunk: 'single',
+     splitChunks: {
+       cacheGroups: {
+         vendor: {
+           test: /[\\/]node_modules[\\/]/,
+           name: 'vendors',
+           chunks: 'all',
+         },
+       },
+     },
    },
  };
```

Let's run another build to see our new `vendor` bundle:

```
...
                         Asset       Size  Chunks             Chunk Names
runtime.cc17ae2a94ec771e9221.js    1.42 KiB       0  [emitted]  runtime
vendors.a42c3ca0d742766d7a28.js    69.4 KiB       1  [emitted]  vendors
   main.abf44fedb7d11d4312d7.js  240 bytes       2  [emitted]  main
                     index.html  353 bytes          [emitted]
...
```

We can now see that our `main` bundle does not contain `vendor` code from `node_modules` directory and is down in size to `240 bytes` !

Module Identifiers

Let's add another module, `print.js`, to our project:

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+ |- print.js
|- /node_modules
```

print.js

```
+ export default function print(text) {
+   console.log(text);
+ };
```

src/index.js

```
  import _ from 'lodash';
+ import Print from './print';

  function component() {
    const element = document.createElement('div');

    // Lodash, now imported by this script
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   element.onclick = Print.bind(null, 'Hello webpack!');

    return element;
  }

  document.body.appendChild(component());
```

Running another build, we would expect only our `main` bundle's hash to change, however...

```
...
                          Asset       Size  Chunks                    Chunk Names
  runtime.1400d5af64fc1b7b3a45.js    5.85 kB       0  [emitted]         runtime
  vendor.a7561fb0e9a071baadb9.js     541 kB       1  [emitted]  [big]  vendor
    main.b746e3eb72875af2caa9.js    1.22 kB       2  [emitted]         main
                     index.html  352 bytes          [emitted]
...
```

... we can see that all three have. This is because each `module.id` is incremented based on resolving order by default. Meaning when the order of resolving is changed, the IDs will be changed

as well. To recap:

- The `main` bundle changed because of its new content.
- The `vendor` bundle changed because its `module.id` was changed.
- And, the `runtime` bundle changed because it now contains a reference to a new module.

The first and last are expected, it's the `vendor` hash we want to fix. Let's use `optimization.moduleIds` with `'deterministic'` option:

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');

  module.exports = {
    entry: './src/index.js',
    plugins: [
      new HtmlWebpackPlugin({
        title: 'Caching',
      }),
    ],
    output: {
      filename: '[name].[contenthash].js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
    optimization: {
+     moduleIds: 'deterministic',
      runtimeChunk: 'single',
      splitChunks: {
        cacheGroups: {
          vendor: {
            test: /[\\/]node_modules[\\/]/,
            name: 'vendors',
            chunks: 'all',
          },
        },
      },
    },
  };
```

Now, despite any new local dependencies, our `vendor` hash should stay consistent between builds:

```
...
                          Asset       Size  Chunks             Chunk Names
   main.216e852f60c8829c2289.js  340 bytes       0  [emitted]  main
vendors.55e79e5927a639d21a1b.js   69.5 KiB       1  [emitted]  vendors
runtime.725a1a51ede5ae0cfde0.js   1.42 KiB       2  [emitted]  runtime
```

```
                    index.html  353 bytes          [emitted]
Entrypoint main = runtime.725a1a51ede5ae0cfde0.js vendors.55e79e5927a639d21a1b.js main.216e85
...
```

And let's modify our `src/index.js` to temporarily remove that extra dependency:

src/index.js

```
  import _ from 'lodash';
- import Print from './print';
+ // import Print from './print';

  function component() {
    const element = document.createElement('div');

    // Lodash, now imported by this script
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-   element.onclick = Print.bind(null, 'Hello webpack!');
+   // element.onclick = Print.bind(null, 'Hello webpack!');

    return element;
  }

  document.body.appendChild(component());
```

And finally run our build again:

```
...
                         Asset       Size  Chunks             Chunk Names
   main.ad717f2466ce655fff5c.js  274 bytes       0  [emitted]  main
vendors.55e79e5927a639d21a1b.js   69.5 KiB       1  [emitted]  vendors
runtime.725a1a51ede5ae0cfde0.js   1.42 KiB       2  [emitted]  runtime
                    index.html  353 bytes          [emitted]
Entrypoint main = runtime.725a1a51ede5ae0cfde0.js vendors.55e79e5927a639d21a1b.js main.ad717f
...
```

We can see that both builds yielded `55e79e5927a639d21a1b` in the `vendor` bundle's filename.

Conclusion

Caching can be complicated, but the benefit to application or site users makes it worth the effort. See the *Further Reading* section below to learn more.

Authoring Libraries

Aside from applications, webpack can also be used to bundle JavaScript libraries. The following

guide is meant for library authors looking to streamline their bundling strategy.

Authoring a Library

Let's assume that we are writing a small library, `webpack-numbers`, that allows users to convert the numbers 1 through 5 from their numeric representation to a textual one and vice-versa, e.g. 2 to 'two'.

The basic project structure would look like this:

project

```
+  |- webpack.config.js
+  |- package.json
+  |- /src
+    |- index.js
+    |- ref.json
```

Initialize the project with npm, then install `webpack`, `webpack-cli` and `lodash`:

```
npm init -y
npm install --save-dev webpack webpack-cli lodash
```

We install `lodash` as `devDependencies` instead of `dependencies` because we don't want to bundle it into our library, or our library could be easily bloated.

src/ref.json

```
[
  {
    "num": 1,
    "word": "One"
  },
  {
    "num": 2,
    "word": "Two"
  },
  {
    "num": 3,
    "word": "Three"
  },
  {
    "num": 4,
    "word": "Four"
  },
  {
```

```
    "num": 5,
    "word": "Five"
  },
  {
    "num": 0,
    "word": "Zero"
  }
]
```

src/index.js

```
import _ from 'lodash';
import numRef from './ref.json';

export function numToWord(num) {
  return _.reduce(
    numRef,
    (accum, ref) => {
      return ref.num === num ? ref.word : accum;
    },
    ''
  );
}

export function wordToNum(word) {
  return _.reduce(
    numRef,
    (accum, ref) => {
      return ref.word === word && word.toLowerCase() ? ref.num : accum;
    },
    -1
  );
}
```

Webpack Configuration

Let's start with this basic webpack configuration:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
  },
```

```
};
```

In the above example, we're telling webpack to bundle `src/index.js` into `dist/webpack-numbers.js` .

Expose the Library

So far everything should be the same as bundling an application, and here comes the different part – we need to expose exports from the entry point through `output.library` option.

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
+     library: "webpackNumbers",
  },
};
```

We exposed the entry point as `webpackNumbers` so users can use it through script tag:

```
<script src="https://example.org/webpack-numbers.js"></script>
<script>
  window.webpackNumbers.wordToNum('Five');
</script>
```

However it only works when it's referenced through script tag, it can't be used in other environments like CommonJS, AMD, Node.js, etc.

As a library author, we want it to be compatible in different environments, i.e., users should be able to consume the bundled library in multiple ways listed below:

- **CommonJS module require**:

```
const webpackNumbers = require('webpack-numbers');
// ...
webpackNumbers.wordToNum('Two');
```

- **AMD module require**:

```
require(['webpackNumbers'], function (webpackNumbers) {
```

```
  // ...
  webpackNumbers.wordToNum('Two');
});
```

- **script tag:**

```
<!DOCTYPE html>
<html>
  ...
  <script src="https://example.org/webpack-numbers.js"></script>
  <script>
    // ...
    // Global variable
    webpackNumbers.wordToNum('Five');
    // Property in the window object
    window.webpackNumbers.wordToNum('Five');
    // ...
  </script>
</html>
```

Let's update the `output.library` option with its `type` set to `'umd'`:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'webpack-numbers.js',
-   library: 'webpackNumbers',
+   library: {
+     name: 'webpackNumbers',
+     type: 'umd',
+   },
  },
};
```

Now webpack will bundle a library that can work with CommonJS, AMD, and script tag.

> ### Tip
>
> Note that the `library` setup is tied to the `entry` configuration. For most libraries, specifying a single entry point is sufficient. While multi-part libraries are possible, it is more straightforward to expose partial exports through an index script that serves as a single entry point. Using an `array` as an `entry` point for a library is **not recommended**.

Externalize Lodash

Now, if you run `npx webpack`, you will find that a largish bundle is created. If you inspect the file, you'll see that lodash has been bundled along with your code. In this case, we'd prefer to treat `lodash` as a *peer dependency*. Meaning that the consumer should already have `lodash` installed. Hence you would want to give up control of this external library to the consumer of your library.

This can be done using the `externals` configuration:

webpack.config.js

```
    const path = require('path');

    module.exports = {
      entry: './src/index.js',
      output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'webpack-numbers.js',
        library: {
          name: "webpackNumbers",
          type: "umd"
        },
      },
+     externals: {
+       lodash: {
+         commonjs: 'lodash',
+         commonjs2: 'lodash',
+         amd: 'lodash',
+         root: '_',
+       },
+     },
    };
```

This means that your library expects a dependency named `lodash` to be available in the consumer's environment.

External Limitations

For libraries that use several files from a dependency:

```
import A from 'library/one';
import B from 'library/two';

// ...
```

You won't be able to exclude them from the bundle by specifying `library` in the externals. You'll

either need to exclude them one by one or by using a regular expression.

```
module.exports = {
  //...
  externals: [
    'library/one',
    'library/two',
    // Everything that starts with "library/"
    /^library\/.+$/,
  ],
};
```

Final Steps

Optimize your output for production by following the steps mentioned in the production guide. Let's also add the path to your generated bundle as the package's `main` field in with the `package.json`

package.json

```
{
  ...
  "main": "dist/webpack-numbers.js",
  ...
}
```

Or, to add it as a standard module as per this guide:

```
{
  ...
  "module": "src/index.js",
  ...
}
```

The key `main` refers to the standard from `package.json`, and `module` to a proposal to allow the JavaScript ecosystem upgrade to use ES2015 modules without breaking backwards compatibility.

> ### Warning
>
> The `module` property should point to a script that utilizes ES2015 module syntax but no other syntax features that aren't yet supported by browsers or node. This enables webpack to parse the module syntax itself, allowing for lighter bundles via tree shaking if users are only consuming certain parts of the library.

Now you can publish it as an npm package and find it at unpkg.com to distribute it to your users.

> **Tip**
>
> To expose stylesheets associated with your library, the `MiniCssExtractPlugin` should be used. Users can then consume and load these as they would any other stylesheet.

Environment Variables

To disambiguate in your `webpack.config.js` between development and production builds you may use environment variables.

> **Tip**
>
> webpack's environment variables are different from the environment variables of operating system shells like `bash` and `CMD.exe`

The webpack command line environment option `--env` allows you to pass in as many environment variables as you like. Environment variables will be made accessible in your `webpack.config.js`. For example, `--env production` or `--env goal=local`.

```
npx webpack --env goal=local --env production --progress
```

> **Tip**
>
> Setting up your `env` variable without assignment, `--env production` sets `env.production` to `true` by default. There are also other syntaxes that you can use. See the webpack CLI documentation for more information.

There is one change that you will have to make to your webpack config. Typically, `module.exports` points to the configuration object. To use the `env` variable, you must convert `module.exports` to a function:

webpack.config.js

```
const path = require('path');

module.exports = (env) => {
  // Use env.<YOUR VARIABLE> here:
  console.log('Goal: ', env.goal); // 'local'
  console.log('Production: ', env.production); // true

  return {
```

```
    entry: './src/index.js',
    output: {
      filename: 'bundle.js',
      path: path.resolve(__dirname, 'dist'),
    },
  };
};
```

> **Tip**
>
> Webpack CLI offers some built-in environment variables which you can access inside a webpack configuration.

Build Performance

This guide contains some useful tips for improving build/compilation performance.

General

The following best practices should help, whether you're running build scripts in development or production.

Stay Up to Date

Use the latest webpack version. We are always making performance improvements. The latest recommended version of webpack is:

Staying up-to-date with **Node.js** can also help with performance. On top of this, keeping your package manager (e.g. `npm` or `yarn`) up-to-date can also help. Newer versions create more efficient module trees and increase resolving speed.

Loaders

Apply loaders to the minimal number of modules necessary. Instead of:

```
module.exports = {
  //...
  module: {
```

```
    rules: [
      {
        test: /\.js$/,
        loader: 'babel-loader',
      },
    ],
  },
};
```

Use the `include` field to only apply the loader modules that actually need to be transformed by it:

```
const path = require('path');

module.exports = {
  //...
  module: {
    rules: [
      {
        test: /\.js$/,
        include: path.resolve(__dirname, 'src'),
        loader: 'babel-loader',
      },
    ],
  },
};
```

Bootstrap

Each additional loader/plugin has a bootup time. Try to use as few tools as possible.

Resolving

The following steps can increase resolving speed:

- Minimize the number of items in `resolve.modules`, `resolve.extensions`, `resolve.mainFiles`, `resolve.descriptionFiles`, as they increase the number of filesystem calls.
- Set `resolve.symlinks: false` if you don't use symlinks (e.g. `npm link` or `yarn link`).
- Set `resolve.cacheWithContext: false` if you use custom resolving plugins, that are not context specific.

Dlls

Use the `DllPlugin` to move code that is changed less often into a separate compilation. This will improve the application's compilation speed, although it does increase complexity of the build process.

Smaller = Faster

Decrease the total size of the compilation to increase build performance. Try to keep chunks small.

- Use fewer/smaller libraries.
- Use the `SplitChunksPlugin` in Multi-Page Applications.
- Use the `SplitChunksPlugin` in `async` mode in Multi-Page Applications.
- Remove unused code.
- Only compile the part of the code you are currently developing on.

Worker Pool

The `thread-loader` can be used to offload expensive loaders to a worker pool.

> **Warning**
>
> Don't use too many workers, as there is a boot overhead for the Node.js runtime and the loader. Minimize the module transfers between worker and main process. IPC is expensive.

Persistent cache

Use `cache` option in webpack configuration. Clear cache directory on `"postinstall"` in `package.json` .

> **Tip**
>
> We support yarn PnP version 3 `yarn 2 berry` for persistent caching.

Custom plugins/loaders

Profile them to not introduce a performance problem here.

Progress plugin

It is possible to shorten build times by removing `ProgressPlugin` from webpack's configuration. Keep in mind, `ProgressPlugin` might not provide as much value for fast builds as well, so make sure you are leveraging the benefits of using it.

Development

The following steps are especially useful in *development*.

Incremental Builds

Use webpack's watch mode. Don't use other tools to watch your files and invoke webpack. The built-in watch mode will keep track of timestamps and passes this information to the compilation for cache invalidation.

In some setups, watching falls back to polling mode. With many watched files, this can cause a lot of CPU load. In these cases, you can increase the polling interval with `watchOptions.poll`.

Compile in Memory

The following utilities improve performance by compiling and serving assets in memory rather than writing to disk:

- `webpack-dev-server`
- `webpack-hot-middleware`
- `webpack-dev-middleware`

stats.toJson speed

Webpack 4 outputs a large amount of data with its `stats.toJson()` by default. Avoid retrieving portions of the `stats` object unless necessary in the incremental step. `webpack-dev-server` after v3.1.3 contained a substantial performance fix to minimize the amount of data retrieved from the `stats` object per incremental build step.

Devtool

Be aware of the performance differences between the different `devtool` settings.

- `"eval"` has the best performance, but doesn't assist you for transpiled code.

- The `cheap-source-map` variants are more performant if you can live with the slightly worse mapping quality.
- Use a `eval-source-map` variant for incremental builds.

> **Tip**
>
> In most cases, `eval-cheap-module-source-map` is the best option.

Avoid Production Specific Tooling

Certain utilities, plugins, and loaders only make sense when building for production. For example, it usually doesn't make sense to minify and mangle your code with the `TerserPlugin` while in development. These tools should typically be excluded in development:

- `TerserPlugin`
- `[fullhash]` / `[chunkhash]` / `[contenthash]`
- `AggressiveSplittingPlugin`
- `AggressiveMergingPlugin`
- `ModuleConcatenationPlugin`

Minimal Entry Chunk

Webpack only emits updated chunks to the filesystem. For some configuration options, (HMR, `[name]` / `[chunkhash]` / `[contenthash]` in `output.chunkFilename`, `[fullhash]`) the entry chunk is invalidated in addition to the changed chunks.

Make sure the entry chunk is cheap to emit by keeping it small. The following configuration creates an additional chunk for the runtime code, so it's cheap to generate:

```
module.exports = {
  // ...
  optimization: {
    runtimeChunk: true,
  },
};
```

Avoid Extra Optimization Steps

Webpack does extra algorithmic work to optimize the output for size and load performance. These optimizations are performant for smaller codebases, but can be costly in larger ones:

```
module.exports = {
  // ...
  optimization: {
    removeAvailableModules: false,
    removeEmptyChunks: false,
    splitChunks: false,
  },
};
```

Output Without Path Info

Webpack has the ability to generate path info in the output bundle. However, this puts garbage collection pressure on projects that bundle thousands of modules. Turn this off in the `options.output.pathinfo` setting:

```
module.exports = {
  // ...
  output: {
    pathinfo: false,
  },
};
```

Node.js Versions 8.9.10-9.11.1

There was a performance regression in Node.js versions 8.9.10 - 9.11.1 in the ES2015 `Map` and `Set` implementations. Webpack uses those data structures liberally, so this regression affects compile times.

Earlier and later Node.js versions are not affected.

TypeScript Loader

To improve the build time when using `ts-loader`, use the `transpileOnly` loader option. On its own, this option turns off type checking. To gain type checking again, use the `ForkTsCheckerWebpackPlugin`. This speeds up TypeScript type checking and ESLint linting by moving each to a separate process.

```
module.exports = {
  // ...
  test: /\.tsx?$/,
  use: [
    {
      loader: 'ts-loader',
      options: {
```

```
          transpileOnly: true,
        },
      },
    ],
  };
```

> **Tip**
>
> There is a full example on the `ts-loader` GitHub repository.

Production

The following steps are especially useful in *production*.

> **Warning**
>
> **Don't sacrifice the quality of your application for small performance gains!** Keep in mind that optimization quality is, in most cases, more important than build performance.

Source Maps

Source maps are really expensive. Do you really need them?

Specific Tooling Issues

The following tools have certain problems that can degrade build performance:

Babel

- Minimize the number of preset/plugins

TypeScript

- Use the `fork-ts-checker-webpack-plugin` for typechecking in a separate process.
- Configure loaders to skip typechecking.

- Use the `ts-loader` in `happyPackMode: true / transpileOnly: true`.

Sass

- `node-sass` has a bug which blocks threads from the Node.js thread pool. When using it with the `thread-loader` set `workerParallelJobs: 2`.

Content Security Policies

Webpack is capable of adding `nonce` to all scripts that it loads. To activate the feature set a `__webpack_nonce__` variable needs to be included in your entry script. A unique hash based nonce should be generated and provided for each unique page view this is why `__webpack_nonce__` is specified in the entry file and not in the configuration. Please note that `nonce` should always be a base64-encoded string.

Examples

In the entry file:

```
// ...
__webpack_nonce__ = 'c29tZSBjb29sIHN0cmluZyB3aWxsIHBvcCB1cCAxMjM=';
// ...
```

Enabling CSP

Please note that CSPs are not enabled by default. A corresponding header `Content-Security-Policy` or meta tag `<meta http-equiv="Content-Security-Policy" ...>` needs to be sent with the document to instruct the browser to enable the CSP. Here's an example of what a CSP header including a CDN white-listed URL might look like:

```
Content-Security-Policy: default-src 'self'; script-src 'self'
https://trusted.cdn.com;
```

For more information on CSP and `nonce` attribute, please refer to **Further Reading** section at the bottom of this page.

Trusted Types

Webpack is also capable of using Trusted Types to load dynamically constructed scripts, to adhere to CSP `require-trusted-types-for` directive restrictions. See `output.trustedTypes` configuration option.

Development - Vagrant

If you have a more advanced project and use Vagrant to run your development environment in a Virtual Machine, you'll often want to also run webpack in the VM.

Configuring the Project

To start, make sure that the `Vagrantfile` has a static IP;

```
Vagrant.configure("2") do |config|
  config.vm.network :private_network, ip: "10.10.10.61"
end
```

Next, install `webpack` , `webpack-cli` , `@webpack-cli/serve` , and `webpack-dev-server` in your project;

```
npm install --save-dev webpack webpack-cli @webpack-cli/serve webpack-dev-server
```

Make sure to have a `webpack.config.js` file. If you haven't already, use this as a minimal example to get started:

```
module.exports = {
  context: __dirname,
  entry: './app.js',
};
```

And create an `index.html` file. The script tag should point to your bundle. If `output.filename` is not specified in the config, this will be `bundle.js` .

```
<!DOCTYPE html>
<html>
  <head>
    <script src="/bundle.js" charset="utf-8"></script>
  </head>
  <body>
    <h2>Hey!</h2>
  </body>
</html>
```

Note that you also need to create an `app.js` file.

Running the Server

Now, let's run the server:

```
webpack serve --host 0.0.0.0 --client-web-socket-url ws://10.10.10.61:8080/ws --watch-options
```

By default, the server will only be accessible from localhost. We'll be accessing it from our host PC, so we need to change `--host` to allow this.

`webpack-dev-server` will include a script in your bundle that connects to a WebSocket to reload when a change in any of your files occurs. The `--client-web-socket-url` flag makes sure the script knows where to look for the WebSocket. The server will use port `8080` by default, so we should also specify that here.

`--watch-options-poll` makes sure that webpack can detect changes in your files. By default, webpack listens to events triggered by the filesystem, but VirtualBox has many problems with this.

The server should be accessible on `http://10.10.10.61:8080` now. If you make a change in `app.js`, it should live reload.

Advanced Usage with nginx

To mimic a more production-like environment, it is also possible to proxy the `webpack-dev-server` with nginx.

In your nginx configuration file, add the following:

```
server {
  location / {
    proxy_pass http://127.0.0.1:8080;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    error_page 502 @start-webpack-dev-server;
  }

  location @start-webpack-dev-server {
    default_type text/plain;
    return 502 "Please start the webpack-dev-server first.";
  }
}
```

The `proxy_set_header` lines are important, because they allow the WebSockets to work correctly.

The command to start `webpack-dev-server` can then be changed to this:

```
webpack serve --client-web-socket-url ws://10.10.10.61:8080/ws --watch-options-poll
```

This makes the server only accessible on `127.0.0.1`, which is fine because nginx takes care of making it available on your host PC.

Conclusion

We made the Vagrant box accessible from a static IP, and then made `webpack-dev-server` publicly accessible so it is reachable from a browser. We then tackled a common problem that VirtualBox doesn't send out filesystem events, causing the server to not reload on file changes.

Dependency Management

> *es6 modules*

> *commonjs*

> *amd*

require with expression

A context is created if your request contains expressions, so the **exact** module is not known on compile time.

Example, given we have the following folder structure including `.ejs` files:

```
example_directory
|
└──template
|   |   table.ejs
|   |   table-row.ejs
|   |
|   └──directory
|       |   another.ejs
```

When following `require()` call is evaluated:

```
require('./template/' + name + '.ejs');
```

Webpack parses the `require()` call and extracts some information:

```
Directory: ./template
Regular expression: /^.*\.ejs$/
```

context module

A context module is generated. It contains references to **all modules in that directory** that can be required with a request matching the regular expression. The context module contains a map which translates requests to module ids.

Example map:

```
{
  "./table.ejs": 42,
  "./table-row.ejs": 43,
  "./directory/another.ejs": 44
}
```

The context module also contains some runtime logic to access the map.

This means dynamic requires are supported but will cause all matching modules to be included in the bundle.

require.context

You can create your own context with the `require.context()` function.

It allows you to pass in a directory to search, a flag indicating whether subdirectories should be searched too, and a regular expression to match files against.

Webpack parses for `require.context()` in the code while building.

The syntax is as follows:

```
require.context(
  directory,
  (useSubdirectories = true),
  (regExp = /^\.\/.*$/),
  (mode = 'sync')
);
```

Examples:

```
require.context('./test', false, /\.test\.js$/);
// a context with files from the test directory that can be required with a request ending wi

require.context('../', true, /\.stories\.js$/);
// a context with all files in the parent folder and descending folders ending with `.stories
```

> **Warning**
>
> The arguments passed to `require.context` must be literals!

context module API

A context module exports a (require) function that takes one argument: the request.

The exported function has 3 properties: `resolve`, `keys`, `id`.

- `resolve` is a function and returns the module id of the parsed request.
- `keys` is a function that returns an array of all possible requests that the context module can handle.

This can be useful if you want to require all files in a directory or matching a pattern, Example:

```
function importAll(r) {
  r.keys().forEach(r);
}

importAll(require.context('../components/', true, /\.js$/));


const cache = {};

function importAll(r) {
  r.keys().forEach((key) => (cache[key] = r(key)));
}
```

```
importAll(require.context('../components/', true, /\.js$/));
// At build-time cache will be populated with all required modules.
```

- `id` is the module id of the context module. This may be useful for `module.hot.accept`.

Installation

This guide goes through the various methods used to install webpack.

Prerequisites

Before we begin, make sure you have a fresh version of Node.js installed. The current Long Term Support (LTS) release is an ideal starting point. You may run into a variety of issues with the older versions as they may be missing functionality webpack and/or its related packages require.

Local Installation

The latest webpack release is:

To install the latest release or a specific version, run one of the following commands:

```
npm install --save-dev webpack
# or specific version
npm install --save-dev webpack@<version>
```

> **Tip**
>
> Whether to use `--save-dev` or not depends on your use cases. Say you're using webpack only for bundling, then it's suggested that you install it with `--save-dev` option since you're not going to include webpack in your production build. Otherwise you can ignore `--save-dev`.

If you're using webpack v4 or later and want to call `webpack` from the command line, you'll also need to install the CLI.

```
npm install --save-dev webpack-cli
```

Installing locally is what we recommend for most projects. This makes it easier to upgrade projects

individually when breaking changes are introduced. Typically webpack is run via one or more npm scripts which will look for a webpack installation in your local `node_modules` directory:

```
"scripts": {
  "build": "webpack --config webpack.config.js"
}
```

> **Tip**
>
> To run the local installation of webpack you can access its binary version as `node_modules/.bin/webpack`. Alternatively, if you are using npm v5.2.0 or greater, you can run `npx webpack` to do it.

Global Installation

The following NPM installation will make `webpack` available globally:

```
npm install --global webpack
```

> **Warning**
>
> Note that this is **not a recommended practice**. Installing globally locks you down to a specific version of webpack and could fail in projects that use a different version.

Bleeding Edge

If you are enthusiastic about using the latest that webpack has to offer, you can install beta versions or even directly from the webpack repository using the following commands:

```
npm install --save-dev webpack@next
# or a specific tag/branch
npm install --save-dev webpack/webpack#<tagname/branchname>
```

> **Warning**
>
> Take caution when installing these bleeding edge releases! They may still contain bugs and therefore should not be used in production.

Hot Module Replacement

> **Tip**
>
> This guide extends on code examples found in the Development guide.

Hot Module Replacement (or HMR) is one of the most useful features offered by webpack. It allows all kinds of modules to be updated at runtime without the need for a full refresh. This page focuses on **implementation** while the concepts page gives more details on how it works and why it's useful.

> **Warning**
>
> **HMR** is not intended for use in production, meaning it should only be used in development. See the building for production guide for more information.

Enabling HMR

This feature is great for productivity. All we need to do is update our webpack-dev-server configuration, and use webpack's built-in HMR plugin. We'll also remove the entry point for `print.js` as it will now be consumed by the `index.js` module.

Since `webpack-dev-server` v4.0.0, Hot Module Replacement is enabled by default.

> **Tip**
>
> If you took the route of using `webpack-dev-middleware` instead of `webpack-dev-server`, please use the `webpack-hot-middleware` package to enable HMR on your custom server or application.

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: {
     app: './src/index.js',
-    print: './src/print.js',
  },
  devtool: 'inline-source-map',
  devServer: {
```

```
      static: './dist',
+     hot: true,
    },
    plugins: [
      new HtmlWebpackPlugin({
        title: 'Hot Module Replacement',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
  };
```

you can also provide manual entry points for HMR:

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');
+ const webpack = require("webpack");

  module.exports = {
    entry: {
       app: './src/index.js',
-      print: './src/print.js',
+      // Runtime code for hot module replacement
+      hot: 'webpack/hot/dev-server.js',
+      // Dev server client for web socket transport, hot and live reload logic
+      client: 'webpack-dev-server/client/index.js?hot=true&live-reload=true',
    },
    devtool: 'inline-source-map',
    devServer: {
      static: './dist',
+     // Dev server client for web socket transport, hot and live reload logic
+     hot: false,
+     client: false,
    },
    plugins: [
      new HtmlWebpackPlugin({
        title: 'Hot Module Replacement',
      }),
+     // Plugin for hot module replacement
+     new webpack.HotModuleReplacementPlugin(),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
```

```
};
```

> **Tip**
>
> You can use the CLI to modify the webpack-dev-server configuration with the following command: `webpack serve --hot-only .`

Now let's update the `index.js` file so that when a change inside `print.js` is detected we tell webpack to accept the updated module.

index.js

```
import _ from 'lodash';
import printMe from './print.js';

function component() {
  const element = document.createElement('div');
  const btn = document.createElement('button');

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

  btn.innerHTML = 'Click me and check the console!';
  btn.onclick = printMe;

  element.appendChild(btn);

  return element;
}

document.body.appendChild(component());
+
+ if (module.hot) {
+   module.hot.accept('./print.js', function() {
+     console.log('Accepting the updated printMe module!');
+     printMe();
+   })
+ }
```

Start changing the `console.log` statement in `print.js` , and you should see the following output in the browser console (don't worry about that `button.onclick = printMe` output for now, we will also update that part later).

print.js

```
export default function printMe() {
-   console.log('I get called from print.js!');
+   console.log('Updating print.js...');
}
```

console

```
[HMR] Waiting for update signal from WDS...
main.js:4395 [WDS] Hot Module Replacement enabled.
+ 2main.js:4395 [WDS] App updated. Recompiling...
+ main.js:4395 [WDS] App hot update...
+ main.js:4330 [HMR] Checking for updates on the server...
+ main.js:10024 Accepting the updated printMe module!
+ 0.4b8ee77….hot-update.js:10 Updating print.js...
+ main.js:4330 [HMR] Updated modules:
+ main.js:4330 [HMR]  - 20
```

Via the Node.js API

When using Webpack Dev Server with the Node.js API, don't put the dev server options on the webpack configuration object. Instead, pass them as a second parameter upon creation. For example:

```
new WebpackDevServer(options, compiler)
```

To enable HMR, you also need to modify your webpack configuration object to include the HMR entry points. Here's a small example of how that might look:

dev-server.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

const webpack = require('webpack');
const webpackDevServer = require('webpack-dev-server');

const config = {
  mode: 'development',
  entry: [
    // Runtime code for hot module replacement
    'webpack/hot/dev-server.js',
    // Dev server client for web socket transport, hot and live reload logic
    'webpack-dev-server/client/index.js?hot=true&live-reload=true',
    // Your entry
    './src/index.js',
  ],
  devtool: 'inline-source-map',
  plugins: [
    // Plugin for hot module replacement
    new webpack.HotModuleReplacementPlugin(),
    new HtmlWebpackPlugin({
      title: 'Hot Module Replacement',
```

```
    }),
  ],
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, 'dist'),
    clean: true,
  },
};
const compiler = webpack(config);

// `hot` and `client` options are disabled because we added them manually
const server = new webpackDevServer({ hot: false, client: false }, compiler);

(async () => {
  await server.start();
  console.log('dev server is running');
})();
```

See the full documentation of `webpack-dev-server` Node.js API.

> **Tip**
>
> If you're using `webpack-dev-middleware` , check out the `webpack-hot-middleware` package to enable HMR on your custom dev server.

Gotchas

Hot Module Replacement can be tricky. To show this, let's go back to our working example. If you go ahead and click the button on the example page, you will realize the console is printing the old `printMe` function.

This is happening because the button's `onclick` event handler is still bound to the original `printMe` function.

To make this work with HMR we need to update that binding to the new `printMe` function using `module.hot.accept` :

index.js

```
import _ from 'lodash';
import printMe from './print.js';

function component() {
  const element = document.createElement('div');
  const btn = document.createElement('button');
```

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

    btn.innerHTML = 'Click me and check the console!';
    btn.onclick = printMe;  // onclick event is bind to the original printMe function

    element.appendChild(btn);

    return element;
  }

- document.body.appendChild(component());
+ let element = component(); // Store the element to re-render on print.js changes
+ document.body.appendChild(element);

  if (module.hot) {
    module.hot.accept('./print.js', function() {
      console.log('Accepting the updated printMe module!');
-     printMe();
+     document.body.removeChild(element);
+     element = component(); // Re-render the "component" to update the click handler
+     document.body.appendChild(element);
    })
  }
```

This is only one example, but there are many others that can easily trip people up. Luckily, there are a lot of loaders out there (some of which are mentioned below) that will make hot module replacement much easier.

HMR with Stylesheets

Hot Module Replacement with CSS is actually fairly straightforward with the help of the `style-loader`. This loader uses `module.hot.accept` behind the scenes to patch `<style>` tags when CSS dependencies are updated.

First let's install both loaders with the following command:

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

Now let's update the configuration file to make use of the loader.

webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
  entry: {
```

```
      app: './src/index.js',
    },
    devtool: 'inline-source-map',
    devServer: {
      static: './dist',
      hot: true,
    },
+   module: {
+     rules: [
+       {
+         test: /\.css$/,
+         use: ['style-loader', 'css-loader'],
+       },
+     ],
+   },
    plugins: [
      new HtmlWebpackPlugin({
        title: 'Hot Module Replacement',
      }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
  };
```

Hot loading stylesheets can be done by importing them into a module:

project

```
  webpack-demo
  | - package.json
  | - webpack.config.js
  | - /dist
    | - bundle.js
  | - /src
    | - index.js
    | - print.js
+   | - styles.css
```

styles.css

```
body {
  background: blue;
}
```

index.js

```
import _ from 'lodash';
```

```
  import printMe from './print.js';
+ import './styles.css';

  function component() {
    const element = document.createElement('div');
    const btn = document.createElement('button');

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

    btn.innerHTML = 'Click me and check the console!';
    btn.onclick = printMe;  // onclick event is bind to the original printMe function

    element.appendChild(btn);

    return element;
  }

  let element = component();
  document.body.appendChild(element);

  if (module.hot) {
    module.hot.accept('./print.js', function() {
      console.log('Accepting the updated printMe module!');
      document.body.removeChild(element);
      element = component(); // Re-render the "component" to update the click handler
      document.body.appendChild(element);
    })
  }
```

Change the style on `body` to `background: red;` and you should immediately see the page's background color change without a full refresh.

styles.css

```
  body {
-   background: blue;
+   background: red;
  }
```

Other Code and Frameworks

There are many other loaders and examples out in the community to make HMR interact smoothly with a variety of frameworks and libraries...

- React Hot Loader: Tweak react components in real time.
- Vue Loader: This loader supports HMR for vue components out of the box.

- Elm Hot webpack Loader: Supports HMR for the Elm programming language.
- Angular HMR: No loader necessary! A small change to your main NgModule file is all that's required to have full control over the HMR APIs.
- Svelte Loader: This loader supports HMR for Svelte components out of the box.

> **Tip**
>
> If you know of any other loaders or plugins that help with or enhance HMR, please submit a pull request to add them to this list!

Tree Shaking

Tree shaking is a term commonly used in the JavaScript context for dead-code elimination. It relies on the static structure of ES2015 module syntax, i.e. `import` and `export`. The name and concept have been popularized by the ES2015 module bundler rollup.

The webpack 2 release came with built-in support for ES2015 modules (alias *harmony modules*) as well as unused module export detection. The new webpack 4 release expands on this capability with a way to provide hints to the compiler via the `"sideEffects"` `package.json` property to denote which files in your project are "pure" and therefore safe to prune if unused.

> **Tip**
>
> The remainder of this guide will stem from Getting Started. If you haven't read through that guide already, please do so now.

Add a Utility

Let's add a new utility file to our project, `src/math.js`, that exports two functions:

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- index.js
```

```
+ |- math.js
|- /node_modules
```

src/math.js

```
export function square(x) {
  return x * x;
}

export function cube(x) {
  return x * x * x;
}
```

Set the `mode` configuration option to development to make sure that the bundle is not minified:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
+ mode: 'development',
+ optimization: {
+   usedExports: true,
+ },
};
```

With that in place, let's update our entry script to utilize one of these new methods and remove `lodash` for simplicity:

src/index.js

```
- import _ from 'lodash';
+ import { cube } from './math.js';

  function component() {
-   const element = document.createElement('div');
+   const element = document.createElement('pre');

-   // Lodash, now imported by this script
-   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   element.innerHTML = [
+     'Hello webpack!',
+     '5 cubed is equal to ' + cube(5)
+   ].join('\n\n');

    return element;
  }

  document.body.appendChild(component());
```

Note that we **did not `import` the `square` method** from the `src/math.js` module. That function is what's known as "dead code", meaning an unused `export` that should be dropped. Now let's run our npm script, `npm run build`, and inspect the output bundle:

dist/bundle.js (around lines 90 - 100)

```
/* 1 */
/***/ (function (module, __webpack_exports__, __webpack_require__) {
  'use strict';
  /* unused harmony export square */
  /* harmony export (immutable) */ __webpack_exports__['a'] = cube;
  function square(x) {
    return x * x;
  }

  function cube(x) {
    return x * x * x;
  }
});
```

Note the `unused harmony export square` comment above. If you look at the code below it, you'll notice that `square` is not being imported, however, it is still included in the bundle. We'll fix that in the next section.

Mark the file as side-effect-free

In a 100% ESM module world, identifying side effects is straightforward. However, we aren't there

quite yet, so in the mean time it's necessary to provide hints to webpack's compiler on the "pureness" of your code.

The way this is accomplished is the `"sideEffects"` package.json property.

```
{
  "name": "your-project",
  "sideEffects": false
}
```

All the code noted above does not contain side effects, so we can mark the property as `false` to inform webpack that it can safely prune unused exports.

> **Tip**
>
> A "side effect" is defined as code that performs a special behavior when imported, other than exposing one or more exports. An example of this are polyfills, which affect the global scope and usually do not provide an export.

If your code did have some side effects though, an array can be provided instead:

```
{
  "name": "your-project",
  "sideEffects": ["./src/some-side-effectful-file.js"]
}
```

The array accepts simple glob patterns to the relevant files. It uses glob-to-regexp under the hood (Supports: `*` , `**` , `{a,b}` , `[a-z]`). Patterns like `*.css` , which do not include a `/` , will be treated like `**/*.css` .

> **Tip**
>
> Note that any imported file is subject to tree shaking. This means if you use something like `css-loader` in your project and import a CSS file, it needs to be added to the side effect list so it will not be unintentionally dropped in production mode:

```
{
  "name": "your-project",
  "sideEffects": ["./src/some-side-effectful-file.js", "*.css"]
}
```

Finally, `"sideEffects"` can also be set from the `module.rules` configuration option.

Clarifying tree shaking and `sideEffects`

The `sideEffects` and `usedExports` (more known as tree shaking) optimizations are two different things.

`sideEffects` is much more effective since it allows to skip whole modules/files and the complete subtree.

`usedExports` relies on terser to detect side effects in statements. It is a difficult task in JavaScript and not as effective as straightforward `sideEffects` flag. It also can't skip subtree/dependencies since the spec says that side effects need to be evaluated. While exporting function works fine, React's Higher Order Components (HOC) are problematic in this regard.

Let's make an example:

```
import { Button } from '@shopify/polaris';
```

The pre-bundled version looks like this:

```
import hoistStatics from 'hoist-non-react-statics';

function Button(_ref) {
  // ...
}

function merge() {
  var _final = {};

  for (
    var _len = arguments.length, objs = new Array(_len), _key = 0;
    _key < _len;
    _key++
  ) {
    objs[_key] = arguments[_key];
  }

  for (var _i = 0, _objs = objs; _i < _objs.length; _i++) {
    var obj = _objs[_i];
    mergeRecursively(_final, obj);
  }

  return _final;
}

function withAppProvider() {
  return function addProvider(WrappedComponent) {
    var WithProvider =
      /*#__PURE__*/
      (function (_React$Component) {
```

```
        // ...
        return WithProvider;
      })(Component);

    WithProvider.contextTypes = WrappedComponent.contextTypes
      ? merge(WrappedComponent.contextTypes, polarisAppProviderContextTypes)
      : polarisAppProviderContextTypes;
    var FinalComponent = hoistStatics(WithProvider, WrappedComponent);
    return FinalComponent;
  };
}

var Button$1 = withAppProvider()(Button);

export {
  // ...,
  Button$1,
};
```

When `Button` is unused you can effectively remove the `export { Button$1 };` which leaves all the remaining code. So the question is "Does this code have any side effects or can it be safely removed?". Difficult to say, especially because of this line `withAppProvider()(Button)`. `withAppProvider` is called and the return value is also called. Are there any side effects when calling `merge` or `hoistStatics`? Are there side effects when assigning `WithProvider.contextTypes` (Setter?) or when reading `WrappedComponent.contextTypes` (Getter?).

Terser actually tries to figure it out, but it doesn't know for sure in many cases. This doesn't mean that terser is not doing its job well because it can't figure it out. It's too difficult to determine it reliably in a dynamic language like JavaScript.

But we can help terser by using the `/*#__PURE__*/` annotation. It flags a statement as side effect free. So a small change would make it possible to tree-shake the code:

```
var Button$1 = /*#__PURE__*/ withAppProvider()(Button);
```

This would allow to remove this piece of code. But there are still questions with the imports which need to be included/evaluated because they could contain side effects.

To tackle this, we use the `"sideEffects"` property in `package.json`.

It's similar to `/*#__PURE__*/` but on a module level instead of a statement level. It says (`"sideEffects"` property): "If no direct export from a module flagged with no-sideEffects is used, the bundler can skip evaluating the module for side effects.".

In the Shopify's Polaris example, original modules look like this:

index.js

```
import './configure';
export * from './types';
export * from './components';
```

components/index.js

```
// ...
export { default as Breadcrumbs } from './Breadcrumbs';
export { default as Button, buttonFrom, buttonsFrom } from './Button';
export { default as ButtonGroup } from './ButtonGroup';
// ...
```

package.json

```
// ...
"sideEffects": [
  "**/*.css",
  "**/*.scss",
  "./esnext/index.js",
  "./esnext/configure.js"
],
// ...
```

For `import { Button } from "@shopify/polaris";` this has the following implications:

- include it: include the module, evaluate it and continue analysing dependencies
- skip over: don't include it, don't evaluate it but continue analysing dependencies
- exclude it: don't include it, don't evaluate it and don't analyse dependencies

Specifically per matching resource(s):

- `index.js` : No direct export is used, but flagged with sideEffects -> include it
- `configure.js` : No export is used, but flagged with sideEffects -> include it
- `types/index.js` : No export is used, not flagged with sideEffects -> exclude it
- `components/index.js` : No direct export is used, not flagged with sideEffects, but reexported exports are used -> skip over
- `components/Breadcrumbs.js` : No export is used, not flagged with sideEffects -> exclude it. This also excluded all dependencies like `components/Breadcrumbs.css` even if they are flagged with sideEffects.
- `components/Button.js` : Direct export is used, not flagged with sideEffects -> include it
- `components/Button.css` : No export is used, but flagged with sideEffects -> include it

In this case only 4 modules are included into the bundle:

- `index.js` : pretty much empty
- `configure.js`
- `components/Button.js`
- `components/Button.css`

After this optimization, other optimizations can still apply. For example: `buttonFrom` and `buttonsFrom` exports from `Button.js` are unused too. `usedExports` optimization will pick it up and terser may be able to drop some statements from the module.

Module Concatenation also applies. So that these 4 modules plus the entry module (and probably more dependencies) can be concatenated. **`index.js` has no code generated in the end**.

Mark a function call as side-effect-free

It is possible to tell webpack that a function call is side-effect-free (pure) by using the `/*#__PURE__*/` annotation. It can be put in front of function calls to mark them as side-effect-free. Arguments passed to the function are not being marked by the annotation and may need to be marked individually. When the initial value in a variable declaration of an unused variable is considered as side-effect-free (pure), it is getting marked as dead code, not executed and dropped by the minimizer. This behavior is enabled when `optimization.innerGraph` is set to `true` .

file.js

```
/*#__PURE__*/ double(55);
```

Minify the Output

So we've cued up our "dead code" to be dropped by using the `import` and `export` syntax, but we still need to drop it from the bundle. To do that, set the `mode` configuration option to `production` .

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
```

```
- mode: 'development',
- optimization: {
-   usedExports: true,
- }
+ mode: 'production',
};
```

> **Tip**
>
> Note that the `--optimize-minimize` flag can be used to enable `TerserPlugin` as well.

With that squared away, we can run another `npm run build` and see if anything has changed.

Notice anything different about `dist/bundle.js`? The whole bundle is now minified and mangled, but, if you look carefully, you won't see the `square` function included but will see a mangled version of the `cube` function (`function r(e){return e*e*e}n.a=r`). With minification and tree shaking, our bundle is now a few bytes smaller! While that may not seem like much in this contrived example, tree shaking can yield a significant decrease in bundle size when working on larger applications with complex dependency trees.

> **Tip**
>
> `ModuleConcatenationPlugin` is needed for the tree shaking to work. It is added by `mode: 'production'`. If you are not using it, remember to add the `ModuleConcatenationPlugin` manually.

Conclusion

What we've learned is that in order to take advantage of *tree shaking*, you must...

- Use ES2015 module syntax (i.e. `import` and `export`).
- Ensure no compilers transform your ES2015 module syntax into CommonJS modules (this is the default behavior of the popular Babel preset @babel/preset-env - see the documentation for more details).
- Add a `"sideEffects"` property to your project's `package.json` file.
- Use the `production` `mode` configuration option to enable various optimizations including minification and tree shaking.

You can imagine your application as a tree. The source code and libraries you actually use represent the green, living leaves of the tree. Dead code represents the brown, dead leaves of the tree that are consumed by autumn. In order to get rid of the dead leaves, you have to shake the tree, causing

them to fall.

If you are interested in more ways to optimize your output, please jump to the next guide for details on building for production.

Production

In this guide, we'll dive into some of the best practices and utilities for building a production site or application.

> **Tip**
>
> This walkthrough stems from Tree Shaking and Development. Please ensure you are familiar with the concepts/setup introduced in those guides before continuing on.

Setup

The goals of *development* and *production* builds differ greatly. In *development*, we want strong source mapping and a localhost server with live reloading or hot module replacement. In *production*, our goals shift to a focus on minified bundles, lighter weight source maps, and optimized assets to improve load time. With this logical separation at hand, we typically recommend writing **separate webpack configurations** for each environment.

While we will separate the *production* and *development* specific bits out, note that we'll still maintain a "common" configuration to keep things DRY. In order to merge these configurations together, we'll use a utility called `webpack-merge`. With the "common" configuration in place, we won't have to duplicate code within the environment-specific configurations.

Let's start by installing `webpack-merge` and splitting out the bits we've already worked on in previous guides:

```
npm install --save-dev webpack-merge
```

project

```
  webpack-demo
  |- package.json
  |- package-lock.json
- |- webpack.config.js
+ |- webpack.common.js
+ |- webpack.dev.js
+ |- webpack.prod.js
  |- /dist
```

```
  |- /src
    |- index.js
    |- math.js
  |- /node_modules
```

webpack.common.js

```
+ const path = require('path');
+ const HtmlWebpackPlugin = require('html-webpack-plugin');
+
+ module.exports = {
+   entry: {
+     app: './src/index.js',
+   },
+   plugins: [
+     new HtmlWebpackPlugin({
+       title: 'Production',
+     }),
+   ],
+   output: {
+     filename: '[name].bundle.js',
+     path: path.resolve(__dirname, 'dist'),
+     clean: true,
+   },
+ };
```

webpack.dev.js

```
+ const { merge } = require('webpack-merge');
+ const common = require('./webpack.common.js');
+
+ module.exports = merge(common, {
+   mode: 'development',
+   devtool: 'inline-source-map',
+   devServer: {
+     static: './dist',
+   },
+ });
```

webpack.prod.js

```
+ const { merge } = require('webpack-merge');
+ const common = require('./webpack.common.js');
+
+ module.exports = merge(common, {
+   mode: 'production',
+ });
```

In `webpack.common.js`, we now have setup our `entry` and `output` configuration and we've

included any plugins that are required for both environments. In `webpack.dev.js`, we've set `mode` to `development`. Also, we've added the recommended `devtool` for that environment (strong source mapping), as well as our `devServer` configuration. Finally, in `webpack.prod.js`, `mode` is set to `production` which loads `TerserPlugin`, which was first introduced by the tree shaking guide.

Note the use of `merge()` calls in the environment-specific configurations to include our common configuration in `webpack.dev.js` and `webpack.prod.js`. The `webpack-merge` tool offers a variety of advanced features for merging but for our use case we won't need any of that.

NPM Scripts

Now, let's modify our npm scripts to use the new configuration files. For the `start` script, which runs `webpack-dev-server`, we will use `webpack.dev.js`, and for the `build` script, which runs `webpack` to create a production build, we will use `webpack.prod.js`:

package.json

```
  {
    "name": "development",
    "version": "1.0.0",
    "description": "",
    "main": "src/index.js",
    "scripts": {
-     "start": "webpack serve --open",
+     "start": "webpack serve --open --config webpack.dev.js",
-     "build": "webpack"
+     "build": "webpack --config webpack.prod.js"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "devDependencies": {
      "css-loader": "^0.28.4",
      "csv-loader": "^2.1.1",
      "express": "^4.15.3",
      "file-loader": "^0.11.2",
      "html-webpack-plugin": "^2.29.0",
      "style-loader": "^0.18.2",
      "webpack": "^4.30.0",
      "webpack-dev-middleware": "^1.12.0",
      "webpack-dev-server": "^2.9.1",
      "webpack-merge": "^4.1.0",
      "xml-loader": "^1.2.1"
    }
  }
```

Feel free to run those scripts and see how the output changes as we continue adding to our *production* configuration.

Specify the Mode

Many libraries will key off the `process.env.NODE_ENV` variable to determine what should be included in the library. For example, when `process.env.NODE_ENV` is not set to `'production'` some libraries may add additional logging and testing to make debugging easier. However, with `process.env.NODE_ENV` set to `'production'` they might drop or add significant portions of code to optimize how things run for your actual users. Since webpack v4, specifying `mode` automatically configures `DefinePlugin` for you:

webpack.prod.js

```
const { merge } = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'production',
});
```

> **Tip**
>
> Technically, `NODE_ENV` is a system environment variable that Node.js exposes into running scripts. It is used by convention to determine dev-vs-prod behavior by server tools, build scripts, and client-side libraries. Contrary to expectations, `process.env.NODE_ENV` is not set to `'production'` **within** the build script `webpack.config.js`, see #2537. Thus, conditionals like `process.env.NODE_ENV === 'production' ? '[name].[contenthash].bundle.js' : '[name].bundle.js'` within webpack configurations do not work as expected.

If you're using a library like `react`, you should actually see a significant drop in bundle size after adding `DefinePlugin`. Also, note that any of our local `/src` code can key off of this as well, so the following check would be valid:

src/index.js

```
  import { cube } from './math.js';
+
+ if (process.env.NODE_ENV !== 'production') {
+   console.log('Looks like we are in development mode!');
+ }
```

```
function component() {
  const element = document.createElement('pre');

  element.innerHTML = [
    'Hello webpack!',
    '5 cubed is equal to ' + cube(5)
  ].join('\n\n');

  return element;
}

document.body.appendChild(component());
```

Minification

Webpack v4+ will minify your code by default in `production mode`.

Note that while the `TerserPlugin` is a great place to start for minification and being used by default, there are other options out there:

- `ClosureWebpackPlugin`

If you decide to try another minification plugin, make sure your new choice also drops dead code as described in the tree shaking guide and provide it as the `optimization.minimizer`.

Source Mapping

We encourage you to have source maps enabled in production, as they are useful for debugging as well as running benchmark tests. That said, you should choose one with a fairly quick build speed that's recommended for production use (see `devtool`). For this guide, we'll use the `source-map` option in the *production* as opposed to the `inline-source-map` we used in the *development*:

webpack.prod.js

```
const { merge } = require('webpack-merge');
const common = require('./webpack.common.js');

module.exports = merge(common, {
  mode: 'production',
+   devtool: 'source-map',
});
```

> **Tip**

> Avoid `inline-***` and `eval-***` use in production as they can increase bundle size and reduce the overall performance.

Minimize CSS

It is crucial to minimize your CSS for production. Please see the Minimizing for Production section.

CLI Alternatives

Many of the options described above can be set as command line arguments. For example, `optimization.minimize` can be set with `--optimization-minimize`, and `mode` can be set with `--mode`. Run `npx webpack --help=verbose` for a full list of CLI arguments.

While these shorthand methods are useful, we recommend setting these options in a webpack configuration file for more configurability.

Lazy Loading

> **Tip**
>
> This guide is a small follow-up to Code Splitting. If you have not yet read through that guide, please do so now.

Lazy, or "on demand", loading is a great way to optimize your site or application. This practice essentially involves splitting your code at logical breakpoints, and then loading it once the user has done something that requires, or will require, a new block of code. This speeds up the initial load of the application and lightens its overall weight as some blocks may never even be loaded.

Example

Let's take the example from Code Splitting and tweak it a bit to demonstrate this concept even more. The code there does cause a separate chunk, `lodash.bundle.js`, to be generated and technically "lazy-loads" it as soon as the script is run. The trouble is that no user interaction is required to load the bundle – meaning that every time the page is loaded, the request will fire. This doesn't help us too much and will impact performance negatively.

Let's try something different. We'll add an interaction to log some text to the console when the user

clicks a button. However, we'll wait to load that code (`print.js`) until the interaction occurs for the first time. To do this we'll go back and rework the final *Dynamic Imports* example from *Code Splitting* and leave `lodash` in the main chunk.

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
+ |- print.js
|- /node_modules
```

src/print.js

```
console.log(
  'The print.js module has loaded! See the network tab in dev tools...'
);

export default () => {
  console.log('Button Clicked: Here\'s "some text"!');
};
```

src/index.js

```
+ import _ from 'lodash';
+
- async function getComponent() {
+ function component() {
    const element = document.createElement('div');
-   const _ = await import(/* webpackChunkName: "lodash" */ 'lodash');
+   const button = document.createElement('button');
+   const br = document.createElement('br');

+   button.innerHTML = 'Click me and look at the console!';
    element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   element.appendChild(br);
+   element.appendChild(button);
+
+   // Note that because a network request is involved, some indication
+   // of loading would need to be shown in a production-level site/app.
+   button.onclick = e => import(/* webpackChunkName: "print" */ './print').then(module => {
+     const print = module.default;
+
+     print();
+   });

    return element;
  }

- getComponent().then(component => {
-   document.body.appendChild(component);
- });
+ document.body.appendChild(component());
```

> **Warning**
>
> Note that when using `import()` on ES6 modules you must reference the `.default` property as it's the actual `module` object that will be returned when the promise is resolved.

Now let's run webpack and check out our new lazy-loading functionality:

```
...
          Asset       Size  Chunks                    Chunk Names
print.bundle.js  417 bytes       0  [emitted]         print
index.bundle.js     548 kB       1  [emitted]  [big]  index
     index.html  189 bytes          [emitted]
...
```

Frameworks

Many frameworks and libraries have their own recommendations on how this should be accomplished within their methodologies. Here are a few examples:

- React: Code Splitting and Lazy Loading
- Vue: Dynamic Imports in Vue.js for better performance
- Angular: Lazy Loading route configuration and AngularJS + webpack = lazyLoad

ECMAScript Modules

ECMAScript Modules (ESM) is a specification for using Modules in the Web. It's supported by all modern browsers and the recommended way of writing modular code for the Web.

Webpack supported processing ECMAScript Modules to optimize them.

Exporting

The `export` keyword allows to expose things from an ESM to other modules:

```
export const CONSTANT = 42;

export let variable = 42;
// only reading is exposed
// it's not possible to modify the variable from outside

export function fun() {
  console.log('fun');
}

export class C extends Super {
  method() {
    console.log('method');
  }
}

let a, b, other;
export { a, b, other as c };

export default 1 + 2 + 3 + more();
```

Importing

The `import` keyword allows to get references to things from other modules into an ESM:

```
import { CONSTANT, variable } from './module.js';
// import "bindings" to exports from another module
// these bindings are live. The values are not copied,
// instead accessing "variable" will get the current value
// in the imported module

import * as module from './module.js';
module.fun();
// import the "namespace object" which contains all exports

import theDefaultValue from './module.js';
// shortcut to import the "default" export
```

Flagging modules as ESM

By default webpack will automatically detect whether a file is an ESM or a different module system.

Node.js established a way of explicitly setting the module type of files by using a property in the `package.json`. Setting `"type": "module"` in a package.json does force all files below this package.json to be ECMAScript Modules. Setting `"type": "commonjs"` will instead force them to be CommonJS Modules.

```
{
  "type": "module"
}
```

In addition to that, files can set the module type by using `.mjs` or `.cjs` extension. `.mjs` will force them to be ESM, `.cjs` force them to be CommonJs.

In DataURIs using the `text/javascript` or `application/javascript` mime type will also force module type to ESM.

In addition to the module format, flagging modules as ESM also affect the resolving logic, interop logic and the available symbols in modules.

Imports in ESM are resolved more strictly. Relative requests must include a filename and file extension.

> **Tip**
>
> Requests to packages e.g. `import "lodash"` are still supported.

Only the "default" export can be imported from non-ESM. Named exports are not available.

CommonJs Syntax is not available: `require` , `module` , `exports` , `__filename` , `__dirname` .

> **Tip**
>
> HMR can be used with `import.meta.webpackHot` instead of `module.hot` .

Shimming

The `webpack` compiler can understand modules written as ES2015 modules, CommonJS or AMD. However, some third party libraries may expect global dependencies (e.g. `$` for `jQuery`). The libraries might also create globals which need to be exported. These "broken modules" are one instance where *shimming* comes into play.

> **Warning**
>
> **We don't recommend using globals!** The whole concept behind webpack is to allow more modular front-end development. This means writing isolated modules that are well contained and do not rely on hidden dependencies (e.g. globals). Please use these features only when necessary.

Another instance where *shimming* can be useful is when you want to polyfill browser functionality to support more users. In this case, you may only want to deliver those polyfills to the browsers that need patching (i.e. load them on demand).

The following article will walk through both of these use cases.

> **Tip**
>
> For simplicity, this guide stems from the examples in Getting Started. Please make sure you are familiar with the setup there before moving on.

Shimming Globals

Let's start with the first use case of shimming global variables. Before we do anything let's take another look at our project:

project

```
webpack-demo
|- package.json
|- package-lock.json
```

```
|- webpack.config.js
|- /dist
  |- index.html
|- /src
  |- index.js
|- /node_modules
```

Remember that `lodash` package we were using? For demonstration purposes, let's say we wanted to instead provide this as a global throughout our application. To do this, we can use `ProvidePlugin`.

The `ProvidePlugin` makes a package available as a variable in every module compiled through webpack. If webpack sees that variable used, it will include the given package in the final bundle. Let's go ahead by removing the `import` statement for `lodash` and instead provide it via the plugin:

src/index.js

```
-import _ from 'lodash';
-
 function component() {
   const element = document.createElement('div');

-  // Lodash, now imported by this script
   element.innerHTML = _.join(['Hello', 'webpack'], ' ');

   return element;
 }

 document.body.appendChild(component());
```

webpack.config.js

```
 const path = require('path');
+const webpack = require('webpack');

 module.exports = {
   entry: './src/index.js',
   output: {
     filename: 'main.js',
     path: path.resolve(__dirname, 'dist'),
   },
+  plugins: [
+    new webpack.ProvidePlugin({
+      _: 'lodash',
+    }),
+  ],
 };
```

What we've essentially done here is tell webpack...

> *If you encounter at least one instance of the variable `_`, include the `lodash` package and provide it to the modules that need it.*

If we run a build, we should still see the same output:

```
$ npm run build

..

[webpack-cli] Compilation finished
asset main.js 69.1 KiB [emitted] [minimized] (name: main) 1 related asset
runtime modules 344 bytes 2 modules
cacheable modules 530 KiB
  ./src/index.js 191 bytes [built] [code generated]
  ./node_modules/lodash/lodash.js 530 KiB [built] [code generated]
webpack 5.4.0 compiled successfully in 2910 ms
```

We can also use the `ProvidePlugin` to expose a single export of a module by configuring it with an "array path" (e.g. `[module, child, ...children?]`). So let's imagine we only wanted to provide the `join` method from `lodash` wherever it's invoked:

src/index.js

```
 function component() {
   const element = document.createElement('div');

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

   return element;
 }

 document.body.appendChild(component());
```

webpack.config.js

```
 const path = require('path');
 const webpack = require('webpack');

 module.exports = {
   entry: './src/index.js',
   output: {
     filename: 'main.js',
     path: path.resolve(__dirname, 'dist'),
   },
   plugins: [
```

```
      new webpack.ProvidePlugin({
-       _: 'lodash',
+       join: ['lodash', 'join'],
      }),
    ],
  };
```

This would go nicely with Tree Shaking as the rest of the `lodash` library should get dropped.

Granular Shimming

Some legacy modules rely on `this` being the `window` object. Let's update our `index.js` so this is the case:

```
  function component() {
    const element = document.createElement('div');

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

+   // Assume we are in the context of `window`
+   this.alert("Hmmm, this probably isn't a great idea...");
+
    return element;
  }

  document.body.appendChild(component());
```

This becomes a problem when the module is executed in a CommonJS context where `this` is equal to `module.exports`. In this case you can override `this` using the `imports-loader`:

webpack.config.js

```
  const path = require('path');
  const webpack = require('webpack');

  module.exports = {
    entry: './src/index.js',
    output: {
      filename: 'main.js',
      path: path.resolve(__dirname, 'dist'),
    },
+   module: {
+     rules: [
+       {
+         test: require.resolve('./src/index.js'),
+         use: 'imports-loader?wrapper=window',
+       },
```

```
+    ],
+  },
   plugins: [
     new webpack.ProvidePlugin({
       join: ['lodash', 'join'],
     }),
   ],
 };
```

Global Exports

Let's say a library creates a global variable that it expects its consumers to use. We can add a small module to our setup to demonstrate this:

project

```
  webpack-demo
  |- package.json
  |- package-lock.json
  |- webpack.config.js
  |- /dist
  |- /src
    |- index.js
+   |- globals.js
  |- /node_modules
```

src/globals.js

```
const file = 'blah.txt';
const helpers = {
  test: function () {
    console.log('test something');
  },
  parse: function () {
    console.log('parse something');
  },
};
```

Now, while you'd likely never do this in your own source code, you may encounter a dated library you'd like to use that contains similar code to what's shown above. In this case, we can use `exports-loader`, to export that global variable as a normal module export. For instance, in order to export `file as file` and `helpers.parse as parse`:

webpack.config.js

```
const path = require('path');
```

```
 const webpack = require('webpack');

 module.exports = {
   entry: './src/index.js',
   output: {
     filename: 'main.js',
     path: path.resolve(__dirname, 'dist'),
   },
   module: {
     rules: [
       {
         test: require.resolve('./src/index.js'),
         use: 'imports-loader?wrapper=window',
       },
+      {
+        test: require.resolve('./src/globals.js'),
+        use:
+          'exports-loader?type=commonjs&exports=file,multiple|helpers.parse|parse',
+      },
     ],
   },
   plugins: [
     new webpack.ProvidePlugin({
       join: ['lodash', 'join'],
     }),
   ],
 };
```

Now from within our entry script (i.e. `src/index.js`), we could use `const { file, parse } = require('./globals.js');` and all should work smoothly.

Loading Polyfills

Almost everything we've discussed thus far has been in relation to handling legacy packages. Let's move on to our second topic: **polyfills**.

There's a lot of ways to load polyfills. For example, to include the `babel-polyfill` we might:

```
npm install --save babel-polyfill
```

and `import` it so as to include it in our main bundle:

src/index.js

```
+import 'babel-polyfill';
+
 function component() {
```

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

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

  // Assume we are in the context of `window`
  this.alert("Hmmm, this probably isn't a great idea...");

  return element;
}

document.body.appendChild(component());
```

> **Tip**
>
> Note that we aren't binding the `import` to a variable. This is because polyfills simply run on their own, prior to the rest of the code base, allowing us to then assume certain native functionality exists.

Note that this approach prioritizes correctness over bundle size. To be safe and robust, polyfills/shims must run **before all other code**, and thus either need to load synchronously, or, all app code needs to load after all polyfills/shims load. There are many misconceptions in the community, as well, that modern browsers "don't need" polyfills, or that polyfills/shims merely serve to add missing features - in fact, they often *repair broken implementations*, even in the most modern of browsers. The best practice thus remains to unconditionally and synchronously load all polyfills/shims, despite the bundle size cost this incurs.

If you feel that you have mitigated these concerns and wish to incur the risk of brokenness, here's one way you might do it: Let's move our `import` to a new file and add the `whatwg-fetch` polyfill:

```
npm install --save whatwg-fetch
```

src/index.js

```
-import 'babel-polyfill';
-
 function component() {
   const element = document.createElement('div');

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

   // Assume we are in the context of `window`
   this.alert("Hmmm, this probably isn't a great idea...");

   return element;
 }
```

```
document.body.appendChild(component());
```

project

```
webpack-demo
|- package.json
|- package-lock.json
|- webpack.config.js
|- /dist
|- /src
  |- index.js
  |- globals.js
+   |- polyfills.js
  |- /node_modules
```

src/polyfills.js

```
import 'babel-polyfill';
import 'whatwg-fetch';
```

webpack.config.js

```
const path = require('path');
const webpack = require('webpack');

module.exports = {
-  entry: './src/index.js',
+  entry: {
+    polyfills: './src/polyfills',
+    index: './src/index.js',
+  },
   output: {
-    filename: 'main.js',
+    filename: '[name].bundle.js',
     path: path.resolve(__dirname, 'dist'),
   },
   module: {
     rules: [
       {
         test: require.resolve('./src/index.js'),
         use: 'imports-loader?wrapper=window',
       },
       {
         test: require.resolve('./src/globals.js'),
         use:
           'exports-loader?type=commonjs&exports[]=file&exports[]=multiple|helpers.parse|pars
       },
     ],
   },
   plugins: [
```

```
    new webpack.ProvidePlugin({
      join: ['lodash', 'join'],
    }),
  ],
};
```

With that in place, we can add the logic to conditionally load our new `polyfills.bundle.js` file. How you make this decision depends on the technologies and browsers you need to support. We'll do some testing to determine whether our polyfills are needed:

dist/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Getting Started</title>
+   <script>
+     const modernBrowser = 'fetch' in window && 'assign' in Object;
+
+     if (!modernBrowser) {
+       const scriptElement = document.createElement('script');
+
+       scriptElement.async = false;
+       scriptElement.src = '/polyfills.bundle.js';
+       document.head.appendChild(scriptElement);
+     }
+   </script>
  </head>
  <body>
-   <script src="main.js"></script>
+   <script src="index.bundle.js"></script>
  </body>
</html>
```

Now we can `fetch` some data within our entry script:

src/index.js

```
function component() {
  const element = document.createElement('div');

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

  // Assume we are in the context of `window`
  this.alert("Hmmm, this probably isn't a great idea...");

  return element;
}
```

```
 document.body.appendChild(component());
+
+fetch('https://jsonplaceholder.typicode.com/users')
+  .then((response) => response.json())
+  .then((json) => {
+    console.log(
+      "We retrieved some data! AND we're confident it will work on a variety of browser dist
+    );
+    console.log(json);
+  })
+  .catch((error) =>
+    console.error('Something went wrong when fetching this data: ', error)
+  );
```

If we run our build, another `polyfills.bundle.js` file will be emitted and everything should still run smoothly in the browser. Note that this set up could likely be improved upon but it should give you a good idea of how you can provide polyfills only to the users that actually need them.

Further Optimizations

The `babel-preset-env` package uses browserslist to transpile only what is not supported in your browsers matrix. This preset comes with the `useBuiltIns` option, `false` by default, which converts your global `babel-polyfill` import to a more granular feature by feature `import` pattern:

```
import 'core-js/modules/es7.string.pad-start';
import 'core-js/modules/es7.string.pad-end';
import 'core-js/modules/web.timers';
import 'core-js/modules/web.immediate';
import 'core-js/modules/web.dom.iterable';
```

See the babel-preset-env documentation for more information.

Node Built-Ins

Node built-ins, like `process`, can be polyfilled right directly from your configuration file without the use of any special loaders or plugins. See the node configuration page for more information and examples.

Other Utilities

There are a few other tools that can help when dealing with legacy modules.

When there is no AMD/CommonJS version of the module and you want to include the `dist`, you can flag this module in `noParse`. This will cause webpack to include the module without parsing it or resolving `require()` and `import` statements. This practice is also used to improve the build performance.

> **Warning**
>
> Any feature requiring the AST, like the `ProvidePlugin`, will not work.

Lastly, there are some modules that support multiple module styles; e.g. a combination of AMD, CommonJS, and legacy. In most of these cases, they first check for `define` and then use some quirky code to export properties. In these cases, it could help to force the CommonJS path by setting `additionalCode=var%20define%20=%20false;` via the `imports-loader`.

TypeScript

> **Tip**
>
> This guide stems from the *Getting Started* guide.

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. In this guide we will learn how to integrate TypeScript with webpack.

Basic Setup

First install the TypeScript compiler and loader by running:

```
npm install --save-dev typescript ts-loader
```

Now we'll modify the directory structure & the configuration files:

project

```
  webpack-demo
  |- package.json
  |- package-lock.json
+ |- tsconfig.json
  |- webpack.config.js
  |- /dist
    |- bundle.js
```

```
    |- index.html
  |- /src
    |- index.js
+   |- index.ts
  |- /node_modules
```

tsconfig.json

Let's set up a configuration to support JSX and compile TypeScript down to ES5...

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "noImplicitAny": true,
    "module": "es6",
    "target": "es5",
    "jsx": "react",
    "allowJs": true,
    "moduleResolution": "node"
  }
}
```

See TypeScript's documentation to learn more about `tsconfig.json` configuration options.

To learn more about webpack configuration, see the configuration concepts.

Now let's configure webpack to handle TypeScript:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.ts',
  module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
```

```
};
```

This will direct webpack to *enter* through `./index.ts`, *load* all `.ts` and `.tsx` files through the `ts-loader`, and *output* a `bundle.js` file in our current directory.

Now lets change the import of `lodash` in our `./index.ts` due to the fact that there is no default export present in `lodash` definitions.

./index.ts

```
- import _ from 'lodash';
+ import * as _ from 'lodash';

  function component() {
    const element = document.createElement('div');

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

    return element;
  }

  document.body.appendChild(component());
```

> ### Tip
>
> To make imports do this by default and keep `import _ from 'lodash';` syntax in TypeScript, set `"allowSyntheticDefaultImports" : true` and `"esModuleInterop" : true` in your **tsconfig.json** file. This is related to TypeScript configuration and mentioned in our guide only for your information.

Loader

`ts-loader`

We use `ts-loader` in this guide as it makes enabling additional webpack features, such as importing other web assets, a bit easier.

> ### Warning
>
> `ts-loader` uses `tsc`, the TypeScript compiler, and relies on your `tsconfig.json` configuration. Make sure to avoid setting `module` to "CommonJS", or webpack won't be able to tree-shake your code.

Note that if you're already using `babel-loader` to transpile your code, you can use `@babel/preset-typescript` and let Babel handle both your JavaScript and TypeScript files instead of using an additional loader. Keep in mind that, contrary to `ts-loader`, the underlying `@babel/plugin-transform-typescript` plugin does not perform any type checking.

Source Maps

To learn more about source maps, see the development guide.

To enable source maps, we must configure TypeScript to output inline source maps to our compiled JavaScript files. The following line must be added to our TypeScript configuration:

tsconfig.json

```
  {
    "compilerOptions": {
      "outDir": "./dist/",
+     "sourceMap": true,
      "noImplicitAny": true,
      "module": "commonjs",
      "target": "es5",
      "jsx": "react",
      "allowJs": true,
      "moduleResolution": "node",
    }
  }
```

Now we need to tell webpack to extract these source maps and include in our final bundle:

webpack.config.js

```
  const path = require('path');

  module.exports = {
    entry: './src/index.ts',
+   devtool: 'inline-source-map',
    module: {
      rules: [
        {
          test: /\.tsx?$/,
          use: 'ts-loader',
          exclude: /node_modules/,
        },
      ],
    },
    resolve: {
      extensions: [ '.tsx', '.ts', '.js' ],
```

```
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
};
```

See the devtool documentation for more information.

Client types

It's possible to use webpack specific features in your TypeScript code, such as `import.meta.webpack`. And webpack provides types for them as well, just add a TypeScript `reference` directive to declare it:

```
/// <reference types="webpack/module" />
console.log(import.meta.webpack); // without reference declared above, TypeScript will throw
```

Using Third Party Libraries

When installing third party libraries from npm, it is important to remember to install the typing definition for that library. These definitions can be found at TypeSearch.

For example if we want to install lodash we can run the following command to get the typings for it:

```
npm install --save-dev @types/lodash
```

For more information see this blog post.

Importing Other Assets

To use non-code assets with TypeScript, we need to defer the type for these imports. This requires a `custom.d.ts` file which signifies custom definitions for TypeScript in our project. Let's set up a declaration for `.svg` files:

custom.d.ts

```
declare module '*.svg' {
  const content: any;
  export default content;
```

```
}
```

Here we declare a new module for SVGs by specifying any import that ends in `.svg` and defining the module's `content` as `any`. We could be more explicit about it being a url by defining the type as string. The same concept applies to other assets including CSS, SCSS, JSON and more.

Build Performance

> **Warning**
>
> This may degrade build performance.

See the Build Performance guide on build tooling.

Web Workers

As of webpack 5, you can use Web Workers without `worker-loader`.

Syntax

```
new Worker(new URL('./worker.js', import.meta.url));
```

The syntax was chosen to allow running code without bundler, it is also available in native ECMAScript modules in the browser.

Example

src/index.js

```
const worker = new Worker(new URL('./deep-thought.js', import.meta.url));
worker.postMessage({
  question:
    'The Answer to the Ultimate Question of Life, The Universe, and Everything.',
});
worker.onmessage = ({ data: { answer } }) => {
  console.log(answer);
};
```

src/deep-thought.js

```
self.onmessage = ({ data: { question } }) => {
  self.postMessage({
    answer: 42,
  });
};
```

Node.js

Similar syntax is supported in Node.js (>= 12.17.0):

```
import { Worker } from 'worker_threads';

new Worker(new URL('./worker.js', import.meta.url));
```

Note that this is only available in ESM. `Worker` in CommonJS syntax is not supported by either webpack or Node.js.

Progressive Web Application

> **Tip**
>
> This guide extends on code examples found in the Output Management guide.

Progressive Web Applications (or PWAs) are web apps that deliver an experience similar to native applications. There are many things that can contribute to that. Of these, the most significant is the ability for an app to be able to function when **offline**. This is achieved through the use of a web technology called Service Workers.

This section will focus on adding an offline experience to our app. We'll achieve this using a Google project called Workbox which provides tools that help make offline support for web apps easier to setup.

We Don't Work Offline Now

So far, we've been viewing the output by going directly to the local file system. Typically though, a real user accesses a web app over a network; their browser talking to a **server** which will serve up the required assets (e.g. `.html` , `.js` , and `.css` files).

So let's test what the current experience is like using a server with more basic features. Let's use the http-server package: `npm install http-server --save-dev`. We'll also amend the `scripts` section of our `package.json` to add in a `start` script:

package.json

```
{
  ...
  "scripts": {
-   "build": "webpack"
+   "build": "webpack",
+   "start": "http-server dist"
  },
  ...
}
```

Note: webpack DevServer writes in-memory by default. We'll need to enable devserverdevmiddleware.writeToDisk option in order for http-server to be able to serve files from `./dist` directory.

If you haven't previously done so, run the command `npm run build` to build your project. Then run the command `npm start`. This should produce the following output:

```
> http-server dist

Starting up http-server, serving dist
Available on:
  http://xx.x.x.x:8080
  http://127.0.0.1:8080
  http://xxx.xxx.x.x:8080
Hit CTRL-C to stop the server
```

If you open your browser to `http://localhost:8080` (i.e. `http://127.0.0.1`) you should see your webpack application being served from the `dist` directory. If you stop the server and refresh, the webpack application is no longer available.

This is what we aim to change. Once we reach the end of this module we should be able to stop the server, hit refresh and still see our application.

Adding Workbox

Let's add the Workbox webpack plugin and adjust the `webpack.config.js` file:

```
npm install workbox-webpack-plugin --save-dev
```

webpack.config.js

```
  const path = require('path');
  const HtmlWebpackPlugin = require('html-webpack-plugin');
+ const WorkboxPlugin = require('workbox-webpack-plugin');

  module.exports = {
    entry: {
      app: './src/index.js',
      print: './src/print.js',
    },
    plugins: [
      new HtmlWebpackPlugin({
-       title: 'Output Management',
+       title: 'Progressive Web Application',
      }),
+     new WorkboxPlugin.GenerateSW({
+       // these options encourage the ServiceWorkers to get in there fast
+       // and not allow any straggling "old" SWs to hang around
+       clientsClaim: true,
+       skipWaiting: true,
+     }),
    ],
    output: {
      filename: '[name].bundle.js',
      path: path.resolve(__dirname, 'dist'),
      clean: true,
    },
  };
```

With that in place, let's see what happens when we do an `npm run build`:

```
...
                  Asset       Size  Chunks                    Chunk Names
          app.bundle.js     545 kB    0, 1  [emitted]  [big]  app
        print.bundle.js    2.74 kB       1  [emitted]         print
             index.html  254 bytes          [emitted]
precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js  268 bytes          [emitted]
      service-worker.js       1 kB          [emitted]
...
```

As you can see, we now have 2 extra files being generated; `service-worker.js` and the more verbose `precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js`. `service-worker.js` is the Service Worker file and `precache-manifest.b5ca1c555e832d6fbf9462efd29d27eb.js` is a file that `service-worker.js` requires so it can run. Your own generated files will likely be different; but you should have a `service-worker.js` file there.

So we're now at the happy point of having produced a Service Worker. What's next?

Registering Our Service Worker

Let's allow our Service Worker to come out and play by registering it. We'll do that by adding the registration code below:

index.js

```
  import _ from 'lodash';
  import printMe from './print.js';

+ if ('serviceWorker' in navigator) {
+   window.addEventListener('load', () => {
+     navigator.serviceWorker.register('/service-worker.js').then(registration => {
+       console.log('SW registered: ', registration);
+     }).catch(registrationError => {
+       console.log('SW registration failed: ', registrationError);
+     });
+   });
+ }
```

Once more `npm run build` to build a version of the app including the registration code. Then serve it with `npm start`. Navigate to `http://localhost:8080` and take a look at the console. Somewhere in there you should see:

```
SW registered
```

Now to test it. Stop your server and refresh your page. If your browser supports Service Workers then you should still be looking at your application. However, it has been served up by your Service Worker and **not** by the server.

Conclusion

You have built an offline app using the Workbox project. You've started the journey of turning your web app into a PWA. You may now want to think about taking things further. A good resource to help you with that can be found here.

Public Path

The `publicPath` configuration option can be quite useful in a variety of scenarios. It allows you to specify the base path for all the assets within your application.

Use Cases

There are a few use cases in real applications where this feature becomes especially neat. Essentially, every file emitted to your `output.path` directory will be referenced from the `output.publicPath` location. This includes child chunks (created via code splitting) and any other assets (e.g. images, fonts, etc.) that are a part of your dependency graph.

Environment Based

In development for example, we might have an `assets/` folder that lives on the same level of our index page. This is fine, but what if we wanted to host all these static assets on a CDN in production?

To approach this problem you can easily use a good old environment variable. Let's say we have a variable `ASSET_PATH`:

```
import webpack from 'webpack';

// Try the environment variable, otherwise use root
const ASSET_PATH = process.env.ASSET_PATH || '/';

export default {
  output: {
    publicPath: ASSET_PATH,
  },

  plugins: [
    // This makes it possible for us to safely use env vars on our code
    new webpack.DefinePlugin({
      'process.env.ASSET_PATH': JSON.stringify(ASSET_PATH),
    }),
  ],
};
```

On The Fly

Another possible use case is to set the `publicPath` on the fly. Webpack exposes a global variable called `__webpack_public_path__` that allows you to do that. In your application's entry point, you can do this:

```
__webpack_public_path__ = process.env.ASSET_PATH;
```

That's all you need. Since we're already using the `DefinePlugin` on our configuration, `process.env.ASSET_PATH` will always be defined so we can safely do that.

> **Warning**
>
> Be aware that if you use ES6 module imports in your entry file the `__webpack_public_path__` assignment will be done after the imports. In such cases, you'll have to move the public path assignment to its own dedicated module and then import it on top of your entry.js:

```
// entry.js
import './public-path';
import './app';
```

Automatic publicPath

There are chances that you don't know what the publicPath will be in advance, and webpack can handle it automatically for you by determining the public path from variables like `import.meta.url`, `document.currentScript`, `script.src` or `self.location`. What you need is to set `output.publicPath` to `'auto'`:

webpack.config.js

```
module.exports = {
  output: {
    publicPath: 'auto',
  },
};
```

Note that in cases where `document.currentScript` is not supported, e.g., IE browser, you will have to include a polyfill like `currentScript Polyfill`.

Integrations

Let's start by clearing up a common misconception. Webpack is a module bundler like Browserify or Brunch. It is *not a task runner* like Make, Grunt, or Gulp. Task runners handle automation of common development tasks such as linting, building, or testing your project. Compared to bundlers, task runners have a higher level focus. You can still benefit from their higher level tooling while leaving the problem of bundling to webpack.

Bundlers help you get your JavaScript and stylesheets ready for deployment, transforming them into a format that's suitable for the browser. For example, JavaScript can be minified or split into chunks and lazy-loaded to improve performance. Bundling is one of the most important challenges in web development, and solving it well can remove a lot of pain from the process.

The good news is that, while there is some overlap, task runners and bundlers can play well together

if approached in the right way. This guide provides a high-level overview of how webpack can be integrated into some of the more popular task runners.

NPM Scripts

Often webpack users use npm `scripts` as their task runner. This is a good starting point. Cross-platform support can become a problem, but there are several workarounds for that. Many, if not most users, get by with npm `scripts` and various levels of webpack configuration and tooling.

So while webpack's core focus is bundling, there are a variety of extensions that can enable you to use it for jobs typical of a task runner. Integrating a separate tool adds complexity, so be sure to weigh the pros and cons before going forward.

Grunt

For those using Grunt, we recommend the `grunt-webpack` package. With `grunt-webpack` you can run webpack or webpack-dev-server as a task, get access to stats within template tags, split development and production configurations and more. Start by installing `grunt-webpack` as well as `webpack` itself if you haven't already:

```
npm install --save-dev grunt-webpack webpack
```

Then register a configuration and load the task:

Gruntfile.js

```
const webpackConfig = require('./webpack.config.js');

module.exports = function (grunt) {
  grunt.initConfig({
    webpack: {
      options: {
        stats: !process.env.NODE_ENV || process.env.NODE_ENV === 'development',
      },
      prod: webpackConfig,
      dev: Object.assign({ watch: true }, webpackConfig),
    },
  });

  grunt.loadNpmTasks('grunt-webpack');
};
```

For more information, please visit the repository.

Gulp

Gulp is also a fairly straightforward integration with the help of the `webpack-stream` package (a.k.a. `gulp-webpack`). In this case, it is unnecessary to install `webpack` separately as it is a direct dependency of `webpack-stream`:

```
npm install --save-dev webpack-stream
```

You can `require('webpack-stream')` instead of `webpack` and optionally pass it an configuration:

gulpfile.js

```
const gulp = require('gulp');
const webpack = require('webpack-stream');
gulp.task('default', function () {
  return gulp
    .src('src/entry.js')
    .pipe(
      webpack({
        // Any configuration options...
      })
    )
    .pipe(gulp.dest('dist/'));
});
```

For more information, please visit the repository.

Mocha

The `mocha-webpack` utility can be used for a clean integration with Mocha. The repository offers more details on the pros and cons but essentially `mocha-webpack` is a simple wrapper that provides almost the same CLI as Mocha itself and provides various webpack functionality like an improved watch mode and improved path resolution. Here is a small example of how you would install it and use it to run a test suite (found within `./test`):

```
npm install --save-dev webpack mocha mocha-webpack
mocha-webpack 'test/**/*.js'
```

For more information, please visit the repository.

Karma

The `karma-webpack` package allows you to use webpack to pre-process files in Karma.

```
npm install --save-dev webpack karma karma-webpack
```

karma.conf.js

```
module.exports = function (config) {
  config.set({
    frameworks: ['webpack'],
    files: [
      { pattern: 'test/*_test.js', watched: false },
      { pattern: 'test/**/*_test.js', watched: false },
    ],
    preprocessors: {
      'test/*_test.js': ['webpack'],
      'test/**/*_test.js': ['webpack'],
    },
    webpack: {
      // Any custom webpack configuration...
    },
    plugins: ['karma-webpack'],
  });
};
```

For more information, please visit the repository.

Advanced entry

Multiple file types per entry

It is possible to provide different types of files when using an array of values for entry to achieve separate bundles for CSS and JavaScript (and other) files in applications that are not using `import` for styles in JavaScript (pre Single Page Applications or different reasons).

Let's make an example. We have a PHP application with two page types: home and account. The home page has different layout and non-sharable JavaScript with the rest of the application (account page). We want to output `home.js` and `home.css` from our application files for the home page and `account.js` and `account.css` for account page.

home.js

```
console.log('home page type');
```

home.scss

```
// home page individual styles
```

account.js

```
console.log('account page type');
```

account.scss

```
// account page individual styles
```

We will use `MiniCssExtractPlugin` in `production` mode for css as a best practice.

webpack.config.js

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  mode: process.env.NODE_ENV,
  entry: {
    home: ['./home.js', './home.scss'],
    account: ['./account.js', './account.scss'],
  },
  output: {
    filename: '[name].js',
  },
  module: {
    rules: [
      {
        test: /\.scss$/,
        use: [
          // fallback to style-loader in development
          process.env.NODE_ENV !== 'production'
            ? 'style-loader'
            : MiniCssExtractPlugin.loader,
          'css-loader',
          'sass-loader',
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: '[name].css',
    }),
  ],
};
```

Running webpack with above configuration will output into `./dist` as we did not specify different output path. `./dist` directory will now contain four files:

- home.js
- home.css
- account.js
- account.css

Asset Modules

Asset Modules is a type of module that allows one to use asset files (fonts, icons, etc) without configuring additional loaders.

Prior to webpack 5 it was common to use:

- `raw-loader` to import a file as a string
- `url-loader` to inline a file into the bundle as a data URI
- `file-loader` to emit a file into the output directory

Asset Modules type replaces all of these loaders by adding 4 new module types:

- `asset/resource` emits a separate file and exports the URL. Previously achievable by using `file-loader`.
- `asset/inline` exports a data URI of the asset. Previously achievable by using `url-loader`.
- `asset/source` exports the source code of the asset. Previously achievable by using `raw-loader`.
- `asset` automatically chooses between exporting a data URI and emitting a separate file. Previously achievable by using `url-loader` with asset size limit.

When using the old assets loaders (i.e. `file-loader` / `url-loader` / `raw-loader`) along with Asset Module in webpack 5, you might want to stop Asset Module from processing your assets again as that would result in asset duplication. This can be done by setting asset's module type to `'javascript/auto'`.

webpack.config.js

```
module.exports = {
  module: {
   rules: [
      {
        test: /\.(png|jpg|gif)$/i,
        use: [
          {
            loader: 'url-loader',
```

```
            options: {
              limit: 8192,
            }
          },
        ],
+       type: 'javascript/auto'
      },
    ]
  },
}
```

To exclude assets that came from new URL calls from the asset loaders add `dependency: { not: ['url'] }` to the loader configuration.

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      {
        test: /\.(png|jpg|gif)$/i,
+       dependency: { not: ['url'] },
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 8192,
            },
          },
        ],
      },
    ],
  }
}
```

Resource assets

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist')
  },
```

```
+ module: {
+   rules: [
+     {
+       test: /\.png/,
+       type: 'asset/resource'
+     }
+   ]
+ },
};
```

src/index.js

```
import mainImage from './images/main.png';

img.src = mainImage; // '/dist/151cfcfa1bd74779aadb.png'
```

All `.png` files will be emitted to the output directory and their paths will be injected into the bundles, besides, you can customize `outputPath` and `publicPath` for them.

Custom output filename

By default, `asset/resource` modules are emitting with `[hash][ext][query]` filename into output directory.

You can modify this template by setting `output.assetModuleFilename` in your webpack configuration:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
+   assetModuleFilename: 'images/[hash][ext][query]'
  },
  module: {
    rules: [
      {
        test: /\.png/,
        type: 'asset/resource'
      }
    ]
  },
};
```

Another case to customize output filename is to emit some kind of assets to a specified directory:

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
+   assetModuleFilename: 'images/[hash][ext][query]'
  },
  module: {
    rules: [
      {
        test: /\.png/,
        type: 'asset/resource'
-     }
+     },
+     {
+       test: /\.html/,
+       type: 'asset/resource',
+       generator: {
+         filename: 'static/[hash][ext][query]'
+       }
+     }
    ]
  },
};
```

With this configuration all the `html` files will be emitted into a `static` directory within the output directory.

`Rule.generator.filename` is the same as `output.assetModuleFilename` and works only with `asset` and `asset/resource` module types.

Inlining assets

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist'),
-   assetModuleFilename: 'images/[hash][ext][query]'
```

```
  },
  module: {
    rules: [
      {
-       test: /\.png/,
-       type: 'asset/resource'
+       test: /\.svg/,
+       type: 'asset/inline'
-     },
+     }
-     {
-       test: /\.html/,
-       type: 'asset/resource',
-       generator: {
-         filename: 'static/[hash][ext][query]'
-       }
-     }
    ]
  }
};
```

src/index.js

```
- import mainImage from './images/main.png';
+ import metroMap from './images/metro.svg';

- img.src = mainImage; // '/dist/151cfcfa1bd74779aadb.png'
+ block.style.background = `url(${metroMap})`; // url(
```

All `.svg` files will be injected into the bundles as data URI.

Custom data URI generator

By default, data URI emitted by webpack represents file contents encoded by using Base64 algorithm.

If you want to use a custom encoding algorithm, you may specify a custom function to encode a file content:

webpack.config.js

```
const path = require('path');
+ const svgToMiniDataURI = require('mini-svg-data-uri');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
```

```
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
        test: /\.svg/,
        type: 'asset/inline',
+       generator: {
+         dataUrl: content => {
+           content = content.toString();
+           return svgToMiniDataURI(content);
+         }
+       }
      }
    ]
  },
};
```

Now all `.svg` files will be encoded by `mini-svg-data-uri` package.

Source assets

webpack.config.js

```
const path = require('path');
- const svgToMiniDataURI = require('mini-svg-data-uri');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
-       test: /\.svg/,
-       type: 'asset/inline',
-       generator: {
-         dataUrl: content => {
-           content = content.toString();
-           return svgToMiniDataURI(content);
-         }
-       }
+       test: /\.txt/,
+       type: 'asset/source',
      }
    ]
```

```
  },
};
```

src/example.txt

```
Hello world
```

src/index.js

```
- import metroMap from './images/metro.svg';
+ import exampleText from './example.txt';

- block.style.background = `url(${metroMap}); // url(
+ block.textContent = exampleText; // 'Hello world'
```

All `.txt` files will be injected into the bundles as is.

URL assets

When using `new URL('./path/to/asset', import.meta.url)`, webpack creates an asset module too.

src/index.js

```
const logo = new URL('./logo.svg', import.meta.url);
```

Depending on the `target` in your configuration, webpack would compile the above code into a different result:

```
// target: web
new URL(
  __webpack_public_path__ + 'logo.svg',
  document.baseURI || self.location.href
);

// target: webworker
new URL(__webpack_public_path__ + 'logo.svg', self.location);

// target: node, node-webkit, nwjs, electron-main, electron-renderer, electron-preload, async
new URL(
  __webpack_public_path__ + 'logo.svg',
  require('url').pathToFileUrl(__filename)
);
```

As of webpack 5.38.0, Data URLs are supported in `new URL()` as well:

src/index.js

```
const url = new URL('data:,', import.meta.url);
console.log(url.href === 'data:,');
console.log(url.protocol === 'data:');
console.log(url.pathname === ',');
```

General asset type

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist')
  },
  module: {
    rules: [
      {
+       test: /\.txt/,
+       type: 'asset',
      }
    ]
  },
};
```

Now webpack will automatically choose between `resource` and `inline` by following a default condition: a file with size less than 8kb will be treated as a `inline` module type and `resource` module type otherwise.

You can change this condition by setting a `Rule.parser.dataUrlCondition.maxSize` option on the module rule level of your webpack configuration:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'main.js',
    path: path.resolve(__dirname, 'dist')
  },
```

```
  module: {
    rules: [
      {
        test: /\.txt/,
        type: 'asset',
+       parser: {
+         dataUrlCondition: {
+           maxSize: 4 * 1024 // 4kb
+         }
+       }
      }
    ]
  },
};
```

Also you can specify a function to decide to inlining a module or not.

Replacing Inline Loader Syntax

Before Asset Modules and Webpack 5, it was possible to use inline syntax with the legacy loaders mentioned above.

It is now recommended to remove all inline loader syntax and use a resourceQuery condition to mimic the functionality of the inline syntax.

For example, in the case of replacing `raw-loader` with `asset/source` type:

```
- import myModule from 'raw-loader!my-module';
+ import myModule from 'my-module?raw';
```

and in the webpack configuration:

```
  module: {
    rules: [
    // ...
+     {
+       resourceQuery: /raw/,
+       type: 'asset/source',
+     }
    ]
  },
```

and if you'd like to exclude raw assets from being processed by other loaders, use a negative condition:

```
  module: {
```

```
    rules: [
    // ...
+     {
+       test: /\.m?js$/,
+       resourceQuery: { not: [/raw/] },
+       use: [ ... ]
+     },
      {
        resourceQuery: /raw/,
        type: 'asset/source',
      }
    ]
  },
```

or a `oneOf` list of rules. Here only the first matching rule will be applied:

```
module: {
    rules: [
    // ...
+     { oneOf: [
        {
          resourceQuery: /raw/,
          type: 'asset/source',
        },
+       {
+         test: /\.m?js$/,
+         use: [ ... ]
+       },
+     ] }
    ]
  },
```

Package exports

The `exports` field in the `package.json` of a package allows to declare which module should be used when using module requests like `import "package"` or `import "package/sub/path"`. It replaces the default implementation that returns `main` field resp. `index.js` files for `"package"` and the file system lookup for `"package/sub/path"`.

When the `exports` field is specified, only these module requests are available. Any other requests will lead to a ModuleNotFound Error.

General syntax

In general the `exports` field should contain an object where each properties specifies a sub path of the module request. For the examples above the following properties could be used: `"."` for

`import "package"` and `"./sub/path"` for `import "package/sub/path"`. Properties ending with a `/` will forward a request with this prefix to the old file system lookup algorithm. For properties ending with `*`, `*` may take any value and any `*` in the property value is replaced with the taken value.

An example:

```
{
  "exports": {
    ".": "./main.js",
    "./sub/path": "./secondary.js",
    "./prefix/": "./directory/",
    "./prefix/deep/": "./other-directory/",
    "./other-prefix/*": "./yet-another/*/*.js"
  }
}
```

Module request	`package`
Result	`.../package/main.js`

Module request	`package/sub/path`
Result	`.../package/secondary.js`

Module request	`package/prefix/some/file.js`
Result	`.../package/directory/some/file.js`

Module request	`package/prefix/deep/file.js`
Result	`.../package/other-directory/file.js`

Module request	`package/other-prefix/deep/file.js`
Result	`.../package/yet-another/deep/file/deep/file.js`

Module request	`package/main.js`
Result	Error

Alternatives

Instead of providing a single result, the package author may provide a list of results. In such a scenario this list is tried in order and the first valid result will be used.

Note: Only the first valid result will be used, not all valid results.

Example:

```
{
  "exports": {
    "./things/": ["./good-things/", "./bad-things/"]
  }
}
```

Here `package/things/apple` might be found in `.../package/good-things/apple` or in `.../package/bad-things/apple`.

Conditional syntax

Instead of providing results directly in the `exports` field, the package author may let the module system choose one based on conditions about the environment.

In this case an object mapping conditions to results should be used. Conditions are tried in object order. Conditions that contain invalid results are skipped. Conditions might be nested to create a logical AND. The last condition in the object might be the special `"default"` condition, which is always matched.

Example:

```
{
  "exports": {
    ".": {
      "red": "./stop.js",
      "yellow": "./stop.js",
      "green": {
        "free": "./drive.js",
        "default": "./wait.js"
      },
      "default": "./drive-carefully.js"
    }
  }
}
```

This translates to something like:

```
if (red && valid('./stop.js')) return './stop.js';
if (yellow && valid('./stop.js')) return './stop.js';
if (green) {
  if (free && valid('./drive.js')) return './drive.js';
  if (valid('./wait.js')) return './wait.js';
```

```
}
if (valid('./drive-carefully.js')) return './drive-carefully.js';
throw new ModuleNotFoundError();
```

The available conditions vary depending on the module system and tool used.

Abbreviation

When only a single entry (`"."`) into the package should be supported the `{ ".": ... }` object nesting can be omitted:

```
{
  "exports": "./index.mjs"
}
```

```
{
  "exports": {
    "red": "./stop.js",
    "green": "./drive.js"
  }
}
```

Notes about ordering

In an object where each key is a condition, order of properties is significant. Conditions are handled in the order they are specified.

Example: `{ "red": "./stop.js", "green": "./drive.js" }` != `{ "green": "./drive.js", "red": "./stop.js" }` (when both `red` and `green` conditions are set, first property will be used)

In an object where each key is a subpath, order of properties (subpaths) is not significant. More specific paths are preferred over less specific ones.

Example: `{ "./a/": "./x/", "./a/b/": "./y/", "./a/b/c": "./z" }` == `{ "./a/b/c": "./z", "./a/b/": "./y/", "./a/": "./x/" }` (order will always be: `./a/b/c` > `./a/b/` > `./a/`)

`exports` field is preferred over other package entry fields like `main` , `module` , `browser` or custom ones.

Support

Feature	`"."` property
Supported by	Node.js, webpack, rollup, esinstall, wmr

Feature	normal property
Supported by	Node.js, webpack, rollup, esinstall, wmr

Feature	property ending with `/`
Supported by	Node.js(1), webpack, rollup, esinstall(2), wmr(3)

Feature	property ending with `*`
Supported by	Node.js, webpack, rollup, esinstall

Feature	Alternatives
Supported by	Node.js, webpack, rollup, ~~esinstall~~(4)

Feature	Abbreviation only path
Supported by	Node.js, webpack, rollup, esinstall, wmr

Feature	Abbreviation only conditions
Supported by	Node.js, webpack, rollup, esinstall, wmr

Feature	Conditional syntax
Supported by	Node.js, webpack, rollup, esinstall, wmr

Feature	Nested conditional syntax
Supported by	Node.js, webpack, rollup, wmr(5)

Feature	Conditions Order
Supported by	Node.js, webpack, rollup, wmr(6)

Feature	`"default"` condition
Supported by	Node.js, webpack, rollup, esinstall, wmr

Feature	Path Order
Supported by	Node.js, webpack, rollup

Feature	Error when not mapped

Supported by	Node.js, webpack, rollup, esinstall, wmr(7)

Feature	Error when mixing conditions and paths
Supported by	Node.js, webpack, rollup

(1) deprecated in Node.js, `*` should be preferred.

(2) `"./"` is intentionally ignored as key.

(3) The property value is ignored and property key is used as target. Effectively only allowing mappings with key and value are identical.

(4) The syntax is supported, but always the first entry is used, which makes it unusable for any practical use case.

(5) Fallback to alternative sibling parent conditions is handling incorrectly.

(6) For the `require` condition object order is handled incorrectly. This is intentionally as wmr doesn't differ between referencing syntax.

(7) When using `"exports": "./file.js"` abbreviation, any request e. g. `package/not-existing` will resolve to that. When not using the abbreviation, direct file access e. g. `package/file.js` will not lead to an error.

Conditions

Reference syntax

One of these conditions is set depending on the syntax used to reference the module:

Condition	`import`
Description	Request is issued from ESM syntax or similar.
Supported by	Node.js, webpack, rollup, esinstall(1), wmr(1)

Condition	`require`
Description	Request is issued from CommonJs/AMD syntax or similar.
Supported by	Node.js, webpack, rollup, esinstall(1), wmr(1)

Condition	`style`
Description	Request is issued from a stylesheet reference.

Condition	`sass`
Description	Request is issued from a sass stylesheet reference.

Condition	`asset`
Description	Request is issued from a asset reference.

Condition	`script`
Description	Request is issued from a normal script tag without module system.

These conditions might also be set additionally:

Condition	`module`
Description	All module syntax that allows to reference javascript supports ESM. (only combined with `import` or `require`)
Supported by	webpack, rollup, wmr

Condition	`esmodules`
Description	Always set by supported tools.
Supported by	wmr

Condition	`types`
Description	Request is issued from typescript that is interested in type declarations.

(1) `import` and `require` are both set independent of referencing syntax. `require` has always lower priority.

import

The following syntax will set the `import` condition:

- ESM `import` declarations in ESM
- JS `import()` expression
- HTML `<script type="module">` in HTML

- HTML `<link rel="preload/prefetch">` in HTML
- JS `new Worker(..., { type: "module" })`
- WASM `import` section
- ESM HMR (webpack) `import.hot.accept/decline([...])`
- JS `Worklet.addModule`
- Using javascript as entrypoint

require

The following syntax will set the `require` condition:

- CommonJs `require(...)`
- AMD `define()`
- AMD `require([...])`
- CommonJs `require.resolve()`
- CommonJs (webpack) `require.ensure([...])`
- CommonJs (webpack) `require.context`
- CommonJs HMR (webpack) `module.hot.accept/decline([...])`
- HTML `<script src="...">`

style

The following syntax will set the `style` condition:

- CSS `@import`
- HTML `<link rel="stylesheet">`

asset

The following syntax will set the `asset` condition:

- CSS `url()`
- ESM `new URL(..., import.meta.url)`
- HTML `<img src="...">`

script

The following syntax will set the `script` condition:

- HTML `<script src="...">`

`script` should only be set when no module system is supported. When the script is preprocessed by a system supporting CommonJs it should set `require` instead.

This condition should be used when looking for a javascript file that can be injected as script tag in a HTML page without additional preprocessing.

Optimizations

The following conditions are set for various optimizations:

Condition	`production`
Description	In a production environment. No devtooling should be included.
Supported by	webpack

Condition	`development`
Description	In a development environment. Devtooling should be included.
Supported by	webpack

Note: Since `production` and `development` is not supported by everyone, no assumption should be made when none of these is set.

Target environment

The following conditions are set depending on the target environment:

Condition	`browser`
Description	Code will run in a browser.
Supported by	webpack, esinstall, wmr

Condition	`electron`
Description	Code will run in electron.(1)
Supported by	webpack

Condition	`worker`
Description	Code will run in a (Web)Worker.(1)

Supported by	webpack

Condition	`worklet`
Description	Code will run in a Worklet.(1)
Supported by	-

Condition	`node`
Description	Code will run in Node.js.
Supported by	Node.js, webpack, wmr(2)

Condition	`deno`
Description	Code will run in Deno.
Supported by	-

Condition	`react-native`
Description	Code will run in react-native.
Supported by	-

(1) `electron` , `worker` and `worklet` comes combined with either `node` or `browser` , depending on the context.

(2) This is set for browser target environment.

Since there are multiple versions of each environment the following guidelines apply:

- `node` : See `engines` field for compatibility.
- `browser` : Compatible with current Spec and stage 4 proposals at time of publishing the package. Polyfilling resp. transpiling must be handled on consumer side.
  - Features that are not possible to polyfill or transpile should be used carefully as it limits the possible usage.
- `deno` : TBD
- `react-native` : TBD

Conditions: Preprocessor and runtimes

The following conditions are set depending on which tool preprocesses the source code.

Condition	`webpack`

Description	Processed by webpack.
Supported by	webpack

Sadly there is no `node-js` condition for Node.js as runtime. This would simplify creating exceptions for Node.js.

Conditions: Custom

The following tools support custom conditions:

Tool	Node.js
Supported	yes
Notes	Use `--conditions` CLI argument.

Tool	webpack
Supported	yes
Notes	Use `resolve.conditionNames` configuration option.

Tool	rollup
Supported	yes
Notes	Use `exportConditions` option for `@rollup/plugin-node-resolve`

Tool	esinstall
Supported	no

Tool	wmr
Supported	no

For custom conditions the following naming schema is recommended:

```
<company-name>:<condition-name>
```

Examples: `example-corp:beta` , `google:internal` .

Common patterns

All patterns are explained with a single `"."` entry into the package, but they can be extended from multiple entries too, by repeating the pattern for each entry.

These pattern should be used as guide not as strict ruleset. They can be adapted to the individual packages.

These pattern are based on the following list of goals/assumptions:

- Packages are rotting.
  - We assume at some point packages are no longer being maintained, but they are continued to be used.
  - `exports` should be written to use fallbacks for unknown future cases. `default` condition can be used for that.
  - As the future is unknown we assume an environment similar to browsers and module system similar to ESM.
- Not all conditions are supported by every tool.
  - Fallbacks should be used to handled these cases.
  - We assume the following fallback make sense in general:
    - ESM > CommonJs
    - Production > Development
    - Browser > node.js

Depending on the package intention maybe something else makes sense and in this case the patterns should be adopted to that. Example: For a command line tool a browser-like future and fallback doesn't make a lot of sense, and in this case node.js-like environments and fallbacks should be used instead.

For complex use cases multiple patterns need to be combined by nesting these conditions.

Target environment independent packages

These patterns make sense for packages that do not use environment specific APIs.

Providing only an ESM version

```
{
  "type": "module",
  "exports": "./index.js"
}
```

Note: Providing only a ESM comes with restrictions for node.js. Such a package would only work in

Node.js >= 14 and only when using `import`. It won't work with `require()`.

Providing CommonJs and ESM version (stateless)

```
{
  "type": "module",
  "exports": {
    "node": {
      "module": "./index.js",
      "require": "./index.cjs"
    },
    "default": "./index.js"
  }
}
```

Most tools get the ESM version. Node.js is an exception here. It gets a CommonJs version when using `require()`. This will lead to two instances of these package when referencing it with `require()` and `import`, but that doesn't hurt as the package doesn't have state.

The `module` condition is used as optimization when preprocessing node-targeted code with a tool that supports ESM for `require()` (like a bundler, when bundling for Node.js). For such a tool the exception is skipped. This is technically optional, but bundlers would include the package source code twice otherwise.

You can also use the stateless pattern if you are able to isolate your package state in JSON files. JSON is consumable from CommonJs and ESM without polluting the graph with the other module system.

Note that here stateless also means class instances are not tested with `instanceof` as there can be two different classes because of the double module instantiation.

Providing CommonJs and ESM version (stateful)

```
{
  "type": "module",
  "exports": {
    "node": {
      "module": "./index.js",
      "import": "./wrapper.js",
      "require": "./index.cjs"
    },
    "default": "./index.js"
  }
}

// wrapper.js
import cjs from './index.cjs';
```

```
export const A = cjs.A;
export const B = cjs.B;
```

In a stateful package we must ensure that the package is never instantiated twice.

This isn't a problem for most tools, but Node.js is again an exception here. For Node.js we always use the CommonJs version and expose named exports in the ESM with a ESM wrapper.

We use the `module` condition as optimization again.

Providing only a CommonJs version

```
{
  "type": "commonjs",
  "exports": "./index.js"
}
```

Providing `"type": "commonjs"` helps to statically detect CommonJs files.

Providing a bundled script version for direct browser consumption

```
{
  "type": "module",
  "exports": {
    "script": "./dist-bundle.js",
    "default": "./index.js"
  }
}
```

Note that despite using `"type": "module"` and `.js` for `dist-bundle.js` this file is not in ESM format. It should use globals to allow direct consumption as script tag.

Providing devtools or production optimizations

These patterns make sense when a package contains two versions, one for development and one for production. E. g. the development version could include additional code for better error message or additional warnings.

Without Node.js runtime detection

```
{
  "type": "module",
  "exports": {
    "development": "./index-with-devtools.js",
    "default": "./index-optimized.js"
  }
}
```

When the `development` condition is supported we use the version enhanced for development. Otherwise, in production or when mode is unknown, we use the optimized version.

With Node.js runtime detection

```
{
  "type": "module",
  "exports": {
    "development": "./index-with-devtools.js",
    "production": "./index-optimized.js",
    "node": "./wrapper-process-env.cjs",
    "default": "./index-optimized.js"
  }
}
```

```
// wrapper-process-env.cjs
if (process.env.NODE_ENV !== 'development') {
  module.exports = require('./index-optimized.cjs');
} else {
  module.exports = require('./index-with-devtools.cjs');
}
```

We prefer static detection of production/development mode via the `production` or `development` condition.

Node.js allows to detection production/development mode at runtime via `process.env.NODE_ENV`, so we use that as fallback in Node.js. Sync conditional importing ESM is not possible and we don't want to load the package twice, so we have to use CommonJs for the runtime detection.

When it's not possible to detect mode we fallback to the production version.

Providing different versions depending on target environment

A fallback environment should be chosen that makes sense for the package to support future environments. In general a browser-like environment should be assumed.

Providing Node.js, WebWorker and browser versions

```
{
  "type": "module",
  "exports": {
    "node": "./index-node.js",
    "worker": "./index-worker.js",
    "default": "./index.js"
  }
}
```

Providing Node.js, browser and electron versions

```
{
  "type": "module",
  "exports": {
    "electron": {
      "node": "./index-electron-node.js",
      "default": "./index-electron.js"
    },
    "node": "./index-node.js",
    "default": "./index.js"
  }
}
```

Combining patterns

Example 1