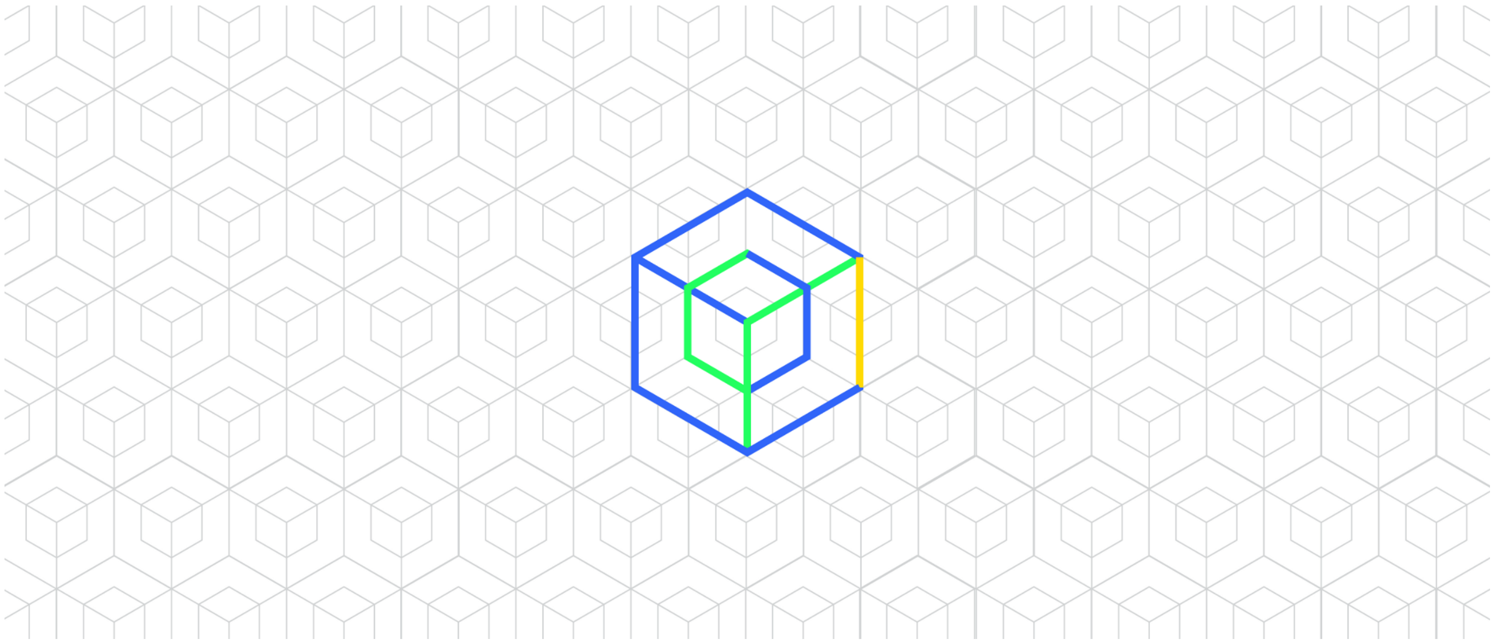
Drew Powers Follow
I know Photoshop @envylabs
Oct 24, 2016 · 12 min read

Getting Started with webpack 2

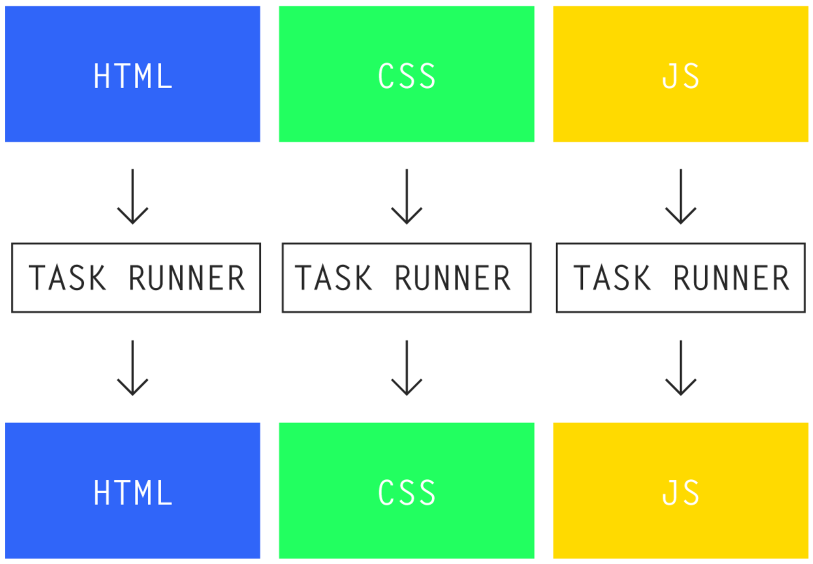


webpack 2 will be out of beta* once the documentation has been finished. But that doesn't mean you can't start using version 2 now if you know how to configure it.

- *Edit: 2 is released! Now* `npm install webpack` *grabs v2.*

What is webpack?

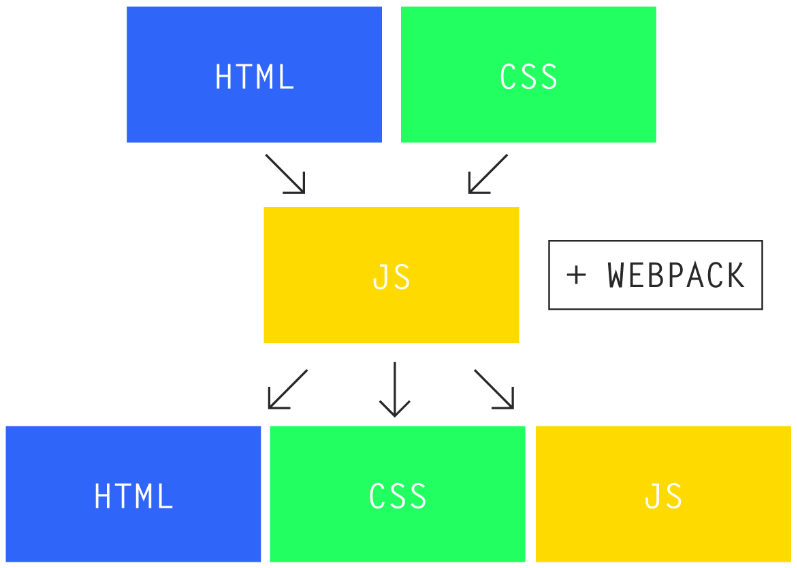
At its simplest, webpack is a module bundler for your JavaScript. However, since its release it's evolved into a manager of all your front-end code (either intentionally or by the community's will).



The old task runner way: your markup, styles, and JavaScript are isolated. You must manage each separately, and it's your job to make sure everything gets to production properly.

A task runner such as *Gulp* can handle many different preprocessers and transpilers, but in all cases, it will take a source *input* and crunch it into a compiled *output*. However, it does this on a case-by-case basis with no concern for the system at large. That is the burden of the developer: to pick up where the task runner left off and find the proper way for all these moving parts to mesh together in production.

webpack attempts to lighten the developer load a bit by asking a bold question: *what if there were a part of the development process that handled dependencies on its own? What if we could simply write code in such a way that the build process managed itself, based on only what was necessary in the end?*



The webpack way: if webpack knows about it, it bundles only what you're *actually* using to production.

If you've been a part of the web community for the past few years, you already know the preferred method of solving a problem: *build this with JavaScript*. And so webpack attempts to make the build process easier by passing dependencies through JavaScript. But the true power of its design isn't simply the code *management* part; it's that this management layer is 100% valid JavaScript (with Node features). webpack gives you the ability to write valid JavaScript that has a better sense of the system at large.

In other words: *you don't write code for webpack. You write code for your projec*t. And webpack keeps up (with some config, of course).

In a nutshell, if you've ever struggled with any of the following:

- Loading dependencies out of order
- Including unused CSS or JS in production
- Accidentally double-loading (or triple-loading) libraries
- Encountering scoping issues—both from CSS and JavaScript
- Finding a good system for using Node/Bower modules in your JavaScript, or relying on a crazy backend configuration to properly utilize those modules
- Needing to optimize asset delivery better but fearing you'll break something

…then you could benefit from webpack. It handles all the above effortlessly by letting JavaScript worry about your dependencies and load order instead of your developer brain. The best part? webpack can even run purely on the server side, meaning you can still build progressively-enhanced websites using webpack.

First Steps

We'll use Yarn (`brew install yarn`) in this tutorial instead of `npm` , but it's totally up to you; they do the same thing. From our project folder, we'll run the following in a terminal window to add webpack to both our global packages and our local project:

```
npm i -g webpack webpack-dev-server@2
yarn add --dev webpack webpack-dev-server@2
```

Note: we're installing it globally for simplicity here, instead of using NPM scripts as recommended. Either way is fine; the docs explain the difference.

We'll then declare a webpack configuration with a `webpack.config.js` file in the root of our project directory:

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

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

Note: `__dirname` refers to the directory where this `webpack.config.js` lives, which in this blogpost is the project root.

Remember that webpack "knows" what's going in your project? It *knows* by reading your code (don't worry; it signed an NDA). webpack basically does the following:

1. Starting from the `context` folder, …
2. … it looks for `entry` filenames …
3. … and reads the content. Every `import` (ES6) or `require()` (Node) dependency it finds as it parses the code, it bundles for the final build. It then searches *those* dependencies, and those dependencies' dependencies, until it reaches the very end of the "tree"—only bundling what it needed to, and nothing else.
4. From there, webpack bundles everything to the `output.path` folder, naming it using the `output.filename` naming template (`[name]` gets replaced with the object key from `entry`)

So if our `src/app.js` file looked something like this (assuming we ran `yarn add moment` beforehand):

```
import moment from 'moment';

var rightNow = moment().format('MMMM Do YYYY, h:mm:ss a');
console.log(rightNow);

// "October 23rd 2016, 9:30:24 pm"
```

We'd run

```
webpack -p
```

Note: The `p` *flag is "production" mode and uglifies/minifies output.*

And it would output a `dist/app.bundle.js` that logged the current date & time to the console. Note that webpack automatically knew what `'moment'` referred to (although if you had a `moment.js` file in your directory, by default webpack would have prioritized this over your `moment` Node module).

Working with Multiple Files

You can specify any number of entry/output points you wish by modifying only the `entry` object.

Multiple files, bundled together

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


module.exports = {
  context: path.resolve(__dirname, './src'),
  entry: {
    app: ['./home.js', './events.js', './vendor.js'],
  },
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: '[name].bundle.js',
  },
};
```

Will all be bundled together as one `dist/app.bundle.js` file, in array order.

Multiple files, multiple outputs

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


module.exports = {
  context: path.resolve(__dirname, './src'),
  entry: {
    home: './home.js',
    events: './events.js',
    contact: './contact.js',
  },
  output: {
    path: path.resolve(__dirname, './dist'),
    filename: '[name].bundle.js',
  },
};
```

Alternately, you may choose to bundle multiple JS files to break up parts of your app. This will be bundled as 3 files: `dist/home.bundle.js` , `dist/events.bundle.js` , and `dist/contact.bundle.js` .

Automatic vendor bundling

If you're breaking up your application into multiple `output` bundles (useful if one part of your app has a ton of JS you don't need to load up front), there's a likelihood you may be duplicating code across those files (usually vendor libraries), because it will resolve each dependency separately from one another. Fortunately, webpack has a built-in *CommonsChunk* plugin to handle this:

```
module.exports = {
  // …

  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      name: 'commons',
      filename: 'commons.js',
      minChunks: 2,
    }),
  ],

// …
};
```

Now, across your `output` files, if you have any modules that get loaded `2` or more times (set by `minChunks`), it will bundle that into a `commons.js` file which you can then cache on the client side. This will result in an additional header request, sure, but you prevent the client from downloading the same libraries more than once. So there are many scenarios where this is a net gain for speed.

Manual vendor bundling

If you'd prefer to do more work yourself, you could alternately take a more manual approach:

```
module.exports = {
  entry: {
    index: './index.js',
    vendor: ['react', 'react-dom', 'rxjs'],
  },
  // …
}
```

In this, you're explicitly telling webpack to export a `vendor` bundle containing your `react`, `react-dom`, and `rxjs` Node modules, rather than relying on CommonsChunkPlugin to do its thing automatically.

Developing

webpack actually has its own development server, so whether you're developing a static site or are just prototyping your front-end, it's perfect for either. To get that running, just add a `devServer` object to `webpack.config.js`:

```
module.exports = {
  context: path.resolve(__dirname, './src'),
  entry: {
    app: './app.js',
  },
  output: {
    filename: '[name].bundle.js',
    path: path.resolve(__dirname, './dist/assets'),
    publicPath: '/assets',                          // New
  },
  devServer: {
    contentBase: path.resolve(__dirname, './src'),  // New
  },
};
```

Now make a `src/index.html` file that has:

```
<script src="/assets/app.bundle.js"></script>
```

… and from your terminal, run:

```
webpack-dev-server
```

Your server is now running at `localhost:8080`. *Note how `/assets` in the script tag matches `output.publicPath`—this prefixes all asset URLs, so you can load assets from anywhere you need to (useful if you use a CDN).*

webpack will hotload any JavaScript changes as you make them without the need to refresh your browser. However, **any changes to the `webpack.config.js` file will require a server restart** to take effect.

Globally-accessible methods

Need to use some of your functions from a global namespace? Simply set `output.library` within `webpack.config.js`:

```
module.exports = {
  output: {
    library: 'myClassName',
  }
};
```

… and it will attach your bundle to a `window.myClassName` instance. So using that name scope, you could call methods available to that entry point (you can read more about this setting on the documentation).

Loaders

Up until now, we've only covered working with JavaScript. It's important to start with JavaScript because *that's the only language webpack speaks*. We can work with virtually any file type, as long as we pass it into JavaScript. We do that with *Loaders*.

A loader can refer to a preprocessor such as Sass, or a transpiler such as Babel. On NPM, they're usually named `*-loader` such as `sass-loader` or `babel-loader`.

Babel + ES6

If we wanted to use ES6 via Babel in our project, we'd first install the appropriate loaders locally:

```
yarn add --dev babel-loader babel-core babel-preset-es2015
```

… and then add it to `webpack.config.js` so webpack knows where to use it.

```
module.exports = {
  // …

  module: {
    rules: [
      {
        test: /\.js$/,
        exclude: [/node_modules/],
        use: [{
          loader: 'babel-loader',
          options: { presets: ['es2015'] },
        }],
      },

      // Loaders for other file types can go here


    ],
  },

  // …
};
```

A note for webpack 1.x users: the core concept for Loaders remains the same, but the syntax has improved.

This looks for the `/\.js$/` RegEx search for any files that end in `.js` to be loaded via Babel. webpack relies on regex tests to give you complete control, not limiting you to only file extensions or assume your code must be organized in a certain way.

If you find a loader mangling files, or otherwise processing things it shouldn't, you can specify an `exclude` option to skip certain files.

Here, we excluded our `node_modules` folder from being processed by Babel—we don't need it. But we also could apply this to any of our own project files, for example if we had a `my_legacy_code` folder. This doesn't prevent you from loading these files; rather, you're just letting webpack know it's OK to import as-is and not process them. You can use `include` as well to make the opposite exception (but you usually won't need this option).

CSS + Style Loader

If we wanted to only load CSS as our application needed, we could do that as well. Let's say we have an `index.js` file. We'll import it from there:

```
import styles from './assets/stylesheets/application.css';
```

We'll get the following error: `You may need an appropriate loader to handle this file type`. Remember that webpack can only understand JavaScript, so we'll have to install the appropriate loader:

```
yarn add --dev css-loader style-loader
```

… and then add a rule to `webpack.config.js`:

```
module.exports = {
  // …

  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },


      // …
    ],
  },
};
```

Loaders are processed in ***reverse array order****. That means* `css-loader` *will run before* `style-loader`.

You may notice that even in production builds, this actually bundles your CSS in with your bundled JavaScript, and `style-loader` manually writes your styles to the `<head>`. At first glance it may seem a little kooky, but slowly starts to make more sense the more you think about it. You've saved a header request—saving valuable time on some connections—and if you're loading your DOM with JavaScript anyway, this essentially eliminates FOUC on its own.

You'll also notice that—out of the box—webpack has automatically resolved all of your `@import` queries by packaging those files together as one (rather than relying on CSS's default import which can result in gratuitious header requests and slow-loading assets).

Loading CSS from your JS is pretty amazing, **because you now can modularize your CSS in powerful new ways.** Say you loaded `button.css` only through `button.js`. This would mean if `button.js` is never actually used, its CSS wouldn't bloat out our production build. If you adhere to component-oriented CSS practices such as SMACSS or BEM, you see the value in pairing your CSS more closely with your markup + JavaScript.

CSS + Node modules

We can use webpack to take advantage of importing Node modules using Node's `~` prefix. If we ran `yarn add normalize.css`, we could use:

```
@import "~normalize.css";
```

… and take full advantage of NPM managing our third party styles for us—versioning and all—without any copy + pasting on our part. Further, getting webpack to bundle CSS for us has obvious advantages over using CSS's default import, saving the client from gratuitous header requests and slow load times.

Update: this and the following section have been updated for accuracy, no longer confusing using CSS Modules to simply import Node modules.

Thanks to Albert Fernández for the help!

CSS Modules

You may have heard of CSS Modules, which takes the *C* out of *CSS*. It typically works best only if you're building the DOM with JavaScript, but in essence, it magically scopes your CSS classes to the JavaScript file that loaded it (learn more about it here). If you plan on using it, CSS Modules comes packaged with `css-loader` (`yarn add --dev css-loader`):

```
module.exports = {
  // …

  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          {
            loader: 'css-loader',
            options: { modules: true },
          },
        ],
      },

      // …
    ],
  },
};
```

*Note: for `css-loader` we're now using the **expanded object syntax** to pass an option to it. You can use a string instead as shorthand to use the default options, as we're still doing with `style-loader`.*

· · ·

It's worth noting that you can actually drop the `~` when importing Node Modules with CSS Modules enabled (e.g.: `@import "normalize.css";`). However, you may encounter build errors now when you `@import` your own CSS. If you're getting "can't find ___" errors, try adding a `resolve` object to `webpack.config.js` to give webpack a better understanding of your intended module order.

```
module.exports = {
  //…

  resolve: {
    modules: [path.resolve(__dirname, './src'),
'node_modules']
  },
};
```

We specified our source directory first, and then `node_modules`. So webpack will handle resolution a little better, first looking through our source directory and then the installed Node modules, in that order (replace `"src"` and `"node_modules"` with your source and Node module directories, respectively).

Sass

Need to use Sass? No problem. Install:

```
yarn add --dev sass-loader node-sass
```

And add another rule:

```
module.exports = {
  // …

  module: {
    rules: [
      {
        test: /\.(sass|scss)$/,
        use: [
          'style-loader',
          'css-loader',
          'sass-loader',
        ]
      }


      // …
    ],
  },
};
```

Then when your Javascript calls for an `import` on a `.scss` or `.sass` file, webpack will do its thing. Remember: the order of `use` is backward, so we're loading Sass first, followed by our CSS parser, and finally Style loader to load our parsed CSS into the `<head>` of our page.

CSS bundled separately

Maybe you're dealing with progressive enhancement; maybe you need a separate CSS file for some other reason. We can do that easily by swapping out `style-loader` with `extract-text-webpack-plugin` in our config without having to change any code. Take our example `app.js` file:

```
import styles from './assets/stylesheets/application.css';
```

Let's install the plugin locally (we need the beta version for this)…

```
yarn add --dev extract-text-webpack-plugin@2.0.0-beta.5
```

… and add to `webpack.config.js`:

```
const ExtractTextPlugin = require('extract-text-webpack-
plugin');

module.exports = {
  // …

  module: {
    rules: [
      {
        test: /\.css$/,
        loader:  ExtractTextPlugin.extract({
          loader: 'css-loader?importLoaders=1',
        }),
      },

      // …
    ]
  },
  plugins: [
    new ExtractTextPlugin({
      filename: '[name].bundle.css',
      allChunks: true,
    }),
  ],
};
```

Note: the latest version of the extract text plugin currently only supports 1.x loader syntax, hence `loader`. This will support `rules` / `use` in an upcoming release.

Now when running `webpack -p` you'll also notice an `app.bundle.css` file in your `output` directory. Simply add a `<link>` tag to that file in your HTML as you would normally.

HTML

As you might have guessed, there's also an `html-loader` plugin for webpack. However, when we get to loading HTML with JavaScript, this is about the point where we branch off into a myriad of differing approaches, and I can't think of one single example that would set you up for whatever you're planning on doing next. Typically, you'd load HTML for the purpose of using JavaScript-flavored markup such as JSX or Mustache or Handlebars to be used within a larger system such as React, Vue, or Angular. Or you're using a pre-processor such as Pug (formerly Jade). Or you could just be literally pushing the same HTML from your source directory into your build directory. Whatever you're doing, I won't assume.

So I'll end the tutorial here: you *can* load markup with webpack, but by this point you'll be making your own decisions about your architecture that neither I nor webpack can make for you. But using the above examples for reference and searching for the right loaders on NPM should be enough to get you going.

Thinking in Modules

In order to get the most out of webpack, you'll have to think in modules—small, reusable, self-contained processes that do one thing and one thing well. That means taking something like this:

```
└── js/
    └── application.js   // 300KB of spaghetti code
```

… and turning it into this:

```
└── js/
    ├── components/
    │   ├── button.js
    │   ├── calendar.js
    │   ├── comment.js
    │   ├── modal.js
    │   ├── tab.js
    │   ├── timer.js
    │   ├── video.js
    │   └── wysiwyg.js
    │
    └── index.js  // ~ 1KB of code; imports from
./components/
```

The result is clean, reusable code. Each individual component depends on `import` -ing its own dependencies, and `export` -ing what it wants to make public to other modules. Pair this with Babel + ES6, and you can utilize JavaScript Classes for great modularity, and *don't-think-about-it* scoping that just works.

For more on modules, see this excellent article by Preethi Kasreddy.

. . .

Further Reading

- What's New in webpack 2
- Webpack + PostCSS + cssnext
- Webpack Config docs
- Webpack Examples
- React + Webpack Starter Kit
- Webpack How-to