npx webpack-cli init

* At its core, webpack is a static module bundler for modern JavaScript applications. When webpack processes your application, it internally builds a dependency graph which maps every module your project needs and generates one or more bundles.
* An **entry** point indicates which module webpack should use to begin building out its internal dependency graph. webpack will figure out which other modules and libraries that entry point depends on (directly and indirectly).

Usage: entry: {[entryChunkName: string]: string|Array<string>}

**webpack.config.js**

module.exports = {

entry: {

app: './src/app.js',

adminApp: './src/adminApp.js'

}

};

**Multi Page Application**

**webpack.config.js**

module.exports = {

entry: {

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

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

pageThree: './src/pageThree/index.js'

}

};

**What does this do?**We are telling webpack that we would like 3 separate dependency graphs (like the above example).

**Why?** In a multi-page application, the server is going to fetch a new HTML document for you. The page reloads this new document and assets are redownloaded. However, this gives us the unique opportunity to do multiple things:

* Use optimization.splitChunks to create bundles of shared application code between each page. Multi-page applications that reuse a lot of code/modules between entry points can greatly benefit from these techniques, as the amount of entry points increase.

As a rule of thumb: for each HTML document use exactly one entry point.

* The **output** property tells webpack where to emit the bundles it creates and how to name these files. It defaults to *./dist/main.js* for the main output file and to the *./dist* folder for any other generated file.

## Multiple Entry Points

If your configuration creates more than a single "chunk" (as with multiple entry points or when using plugins like CommonsChunkPlugin), you should use [substitutions](https://webpack.js.org/configuration/output#output-filename) to ensure that each file has a unique name.

module.exports = {

entry: {

app: './src/app.js',

search: './src/search.js'

},

output: {

filename: '[name].js',

path: \_\_dirname + '/dist'

}

};

// writes to disk: ./dist/app.js, ./dist/search.js

* **Out of the box, webpack only understands JavaScript and JSON files.** **Loaders** allow webpack to process other types of files and convert them into valid modules that can be consumed by your application and added to the dependency graph.
* [*module.rules*](https://webpack.js.org/configuration/module/#module-rules) allows you to specify several loaders within your webpack configuration. This is a concise way to display loaders and helps to maintain clean code. It also offers you a full overview of each respective loader.
* Loaders are evaluated/executed from right to left (or from bottom to top). In the example below execution starts with sass-loader, continues with css-loader and finally ends with style-loader. See ["Loader Features"](https://webpack.js.org/concepts/loaders/#loader-features) for more information about loaders order.

module.exports = {

module: {

rules: [

{

test: /\.css$/,

use: [

// [style-loader](https://webpack.js.org/loaders/style-loader)

{ loader: 'style-loader' },

// [css-loader](https://webpack.js.org/loaders/css-loader)

{

loader: 'css-loader',

options: {

modules: true

}

},

// [sass-loader](https://webpack.js.org/loaders/sass-loader)

{ loader: 'sass-loader' }

]

}

]

}

};

* While loaders are used to transform certain types of modules, **plugins** can be leveraged to perform a wider range of tasks like bundle optimization, asset management and injection of environment variables.

They also serve the purpose of doing **anything else** that a [loader](https://webpack.js.org/concepts/loaders) cannot do.

A webpack **plugin** is a JavaScript object that has an [apply](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply) method. This apply method is called by the webpack compiler, giving access to the **entire** compilation lifecycle.

The first parameter of the tap method of the compiler hook should be a camelized version of the plugin name. It is advisable to use a constant for this so it can be reused in all hooks.

**ConsoleLogOnBuildWebpackPlugin.js**

const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {

apply(compiler) {

compiler.hooks.run.tap(pluginName, compilation => {

console.log('The webpack build process is starting!!!');

});

}

}

**webpack.config.js**

const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm

const webpack = require('webpack'); //to access built-in plugins

const path = require('path');

module.exports = {

entry: './path/to/my/entry/file.js',

output: {

filename: 'my-first-webpack.bundle.js',

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

},

module: {

rules: [

{

test: /\.(js|jsx)$/,

use: 'babel-loader'

}

]

},

plugins: [

new webpack.ProgressPlugin(),

new HtmlWebpackPlugin({template: './src/index.html'})

]

};

When using the Node API, you can also pass plugins via the plugins property in the configuration.

**some-node-script.js**

const webpack = require('webpack'); //to access webpack runtime

const configuration = require('./webpack.config.js');

let compiler = webpack(configuration);

new webpack.ProgressPlugin().apply(compiler);

compiler.run(function(err, stats) {

// ...

});

* By setting the **mode** parameter to either development, production or none, you can enable webpack's built-in optimizations that correspond to each environment. The default value is production.

*The most important part to take away from this document is that there are many different ways to format and style your webpack configuration. The key is to stick with something consistent that you and your team can understand and maintain.*

## Use different config file

If for some reason you want to use different config file depending on certain situations you can change this via command line by using the --config flag.

**package.json**

"scripts": {

"build": "webpack --config prod.config.js"

}

| **Template** | **Description** |
| --- | --- |
| [absolute-resource-path] | The absolute filename |
| [all-loaders] | Automatic and explicit loaders and params up to the name of the first loader |
| [hash] | The hash of the module identifier |
| [id] | The module identifier |
| [loaders] | Explicit loaders and params up to the name of the first loader |
| [resource] | The path used to resolve the file and any query params used on the first loader |
| [resource-path] | The path used to resolve the file without any query params |
| [namespace] | The modules namespace. This is usually the library name when building as a library, empty otherwise |