

# Concepts

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*.

*Learn more about JavaScript modules and webpack modules [here](#).*

Since version 4.0.0, **webpack does not require a configuration file** to bundle your project. Nevertheless, it is *incredibly configurable* to better fit your needs.

To get started you only need to understand its **Core Concepts**:

- [Entry](#)
- [Output](#)
- [Loaders](#)
- [Plugins](#)
- [Mode](#)
- [Browser Compatibility](#)

This document is intended to give a **high-level** overview of these concepts, while providing links to detailed concept-specific use cases.

For a better understanding of the ideas behind module bundlers and how they work under the hood, consult these resources:

- [Manually Bundling an Application](#)
- [Live Coding a Simple Module Bundler](#)
- [Detailed Explanation of a Simple Module Bundler](#)

## Entry


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).

By default its value is `./src/index.js`, but you can specify a different (or multiple entry points) by setting an `entry` [property in the webpack configuration](#). For example:

**webpack.config.js**

```
module.exports = {  
  entry: './path/to/my/entry/file.js'  
};
```



*Learn more in the [entry points](#) section.*

## Output

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.

You can configure this part of the process by specifying an `output` field in your configuration:

**webpack.config.js**

```
const path = require('path');  
  
module.exports = {  
  entry: './path/to/my/entry/file.js',  
  output: {  
    path: path.resolve(__dirname, 'dist'),  
    filename: 'my-first-webpack.bundle.js'  
  }  
};
```



In the example above, we use the `output.filename` and the `output.path` properties to tell webpack the name of our bundle and where we want it to be emitted to. In case you're wondering about the `path` module being imported at the top, it is a core [Node.js module](#) that gets used to manipulate file paths.

*The `output` property has [many more configurable features](#). If you want to learn about the concepts behind it, you can [read more in the output section](#).*

# Loaders

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.

*Note that the ability to `import` any type of module, e.g. `.css` files, is a feature specific to webpack and may not be supported by other bundlers or task runners. We feel this extension of the language is warranted as it allows developers to build a more accurate dependency graph.*

At a high level, **loaders** have two properties in your webpack configuration:

1. The `test` property identifies which file or files should be transformed.
2. The `use` property indicates which loader should be used to do the transforming.

## webpack.config.js

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

module.exports = {
  output: {
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  }
};
```

The configuration above has defined a `rules` property for a single module with two required properties: `test` and `use`. This tells webpack's compiler the following:

*"Hey webpack compiler, when you come across a path that resolves to a `.txt` file inside of a `require()` / `import` statement, **use** the `raw-loader` to transform it before you add it to the bundle."*

*It is important to remember that when defining rules in your webpack config, you are defining them under `module.rules` and not `rules`. For your benefit, webpack will warn you if this is done incorrectly.*

Keep in mind that when using regex to match files, you may not quote it. i.e `/\.txt$/` is not the same as `' /\.txt$/'` or `" /\.txt$/"`. The former instructs webpack to match any file that ends with `.txt` and the latter instructs webpack to match a single file with an absolute path `'txt'`; this is likely not your intention.

You can check further customization when including loaders in the [loaders section](#).

## Plugins

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.

*Check out the [plugin interface](#) and how to use it to extend webpack's capabilities.*

In order to use a plugin, you need to `require()` it and add it to the `plugins` array. Most plugins are customizable through options. Since you can use a plugin multiple times in a config for different purposes, you need to create an instance of it by calling it with the `new` operator.

### webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin'); //installed via npm
const webpack = require('webpack'); //to access built-in plugins

module.exports = {
  module: {
    rules: [
      { test: /\.txt$/, use: 'raw-loader' }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

In the example above, the `html-webpack-plugin` generates an HTML file for your application by injecting automatically all your generated bundles.

*There are many plugins that webpack provides out of the box! Check out the [list of plugins](#).*

Using plugins in your webpack config is straightforward. However, there are many use cases that are

worth further exploration. [Learn more about them here.](#)

## Mode



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`.

```
module.exports = {  
  mode: 'production'  
};
```

Learn more about the [mode configuration here](#) and what optimizations take place on each value.

## Browser Compatibility

webpack supports all browsers that are [ES5-compliant](#) (IE8 and below are not supported). webpack needs `Promise` for `import()` and `require.ensure()`. If you want to support older browsers, you will need to [load a polyfill](#) before using these expressions.