Printable 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 from one or more *entry points* and then combines every module your project needs into one or more *bundles*, which are static assets to serve your content from.

> **Tip**
>
> 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 Basic Module Bundler
- Detailed Explanation of a Basic 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',
};
```

> **Tip**
>
> 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.

> **Tip**
>
> 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.

> ### Warning
>
> One of webpack's specific features is the ability to `import` any type of module, e.g. `.css` files, which 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."*

> ### Warning
>
> 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.

> ### Warning

> 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.

> **Tip**
>
> 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 configuration 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');
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 and automatically injects all your generated bundles into this file.

> **Tip**
>
> There are many plugins that webpack provides out of the box! Check out the list of plugins.

Using plugins in your webpack configuration 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.

Environment

Webpack 5 runs on Node.js version 10.13.0+.

Entry Points

As mentioned in Getting Started, there are multiple ways to define the `entry` property in your webpack configuration. We will show you the ways you **can** configure the `entry` property, in addition to explaining why it may be useful to you.

Single Entry (Shorthand) Syntax

Usage: `entry: string | [string]`

webpack.config.js

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

The single entry syntax for the `entry` property is a shorthand for:

webpack.config.js

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

We can also pass an array of file paths to the `entry` property which creates what is known as a **"multi-main entry"**. This is useful when you would like to inject multiple dependent files together and graph their dependencies into one "chunk".

webpack.config.js

```
module.exports = {
  entry: ['./src/file_1.js', './src/file_2.js'],
  output: {
    filename: 'bundle.js',
  },
};
```

Single Entry Syntax is a great choice when you are looking to quickly set up a webpack configuration for an application or tool with one entry point (i.e. a library). However, there is not much flexibility in extending or scaling your configuration with this syntax.

Object Syntax

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

webpack.config.js

```
module.exports = {
  entry: {
    app: './src/app.js',
    adminApp: './src/adminApp.js',
  },
};
```

The object syntax is more verbose. However, this is the most scalable way of defining entry/entries in your application.

> **Tip**

> **"Scalable webpack configurations"** are ones that can be reused and combined with other partial configurations. This is a popular technique used to separate concerns by environment, build target, and runtime. They are then merged using specialized tools like webpack-merge.

> **Tip**
>
> You can pass empty object `{}` to `entry` when you have only entry points generated by plugins.

EntryDescription object

An object of entry point description. You can specify the following properties.

- `dependOn` : The entry points that the current entry point depends on. They must be loaded before this entry point is loaded.
- `filename` : Specifies the name of each output file on disk.
- `import` : Module(s) that are loaded upon startup.
- `library` : Specify library options to bundle a library from current entry.
- `runtime` : The name of the runtime chunk. When set, a new runtime chunk will be created. It can be set to `false` to avoid a new runtime chunk since webpack 5.43.0.
- `publicPath` : Specify a public URL address for the output files of this entry when they are referenced in a browser. Also, see output.publicPath.

webpack.config.js

```
module.exports = {
  entry: {
    a2: 'dependingfile.js',
    b2: {
      dependOn: 'a2',
      import: './src/app.js',
    },
  },
};
```

`runtime` and `dependOn` should not be used together on a single entry, so the following config is invalid and would throw an error:

webpack.config.js

```
module.exports = {
  entry: {
    a2: './a',
```

```
    b2: {
      runtime: 'x2',
      dependOn: 'a2',
      import: './b',
    },
  },
};
```

Make sure `runtime` must not point to an existing entry point name, for example the below config would throw an error:

```
module.exports = {
  entry: {
    a1: './a',
    b1: {
      runtime: 'a1',
      import: './b',
    },
  },
};
```

Also `dependOn` must not be circular, the following example again would throw an error:

```
module.exports = {
  entry: {
    a3: {
      import: './a',
      dependOn: 'b3',
    },
    b3: {
      import: './b',
      dependOn: 'a3',
    },
  },
};
```

Scenarios

Below is a list of entry configurations and their real-world use cases:

Separate App and Vendor Entries

webpack.config.js

```
module.exports = {
  entry: {
    main: './src/app.js',
```

```
    vendor: './src/vendor.js',
  },
};
```

webpack.prod.js

```
module.exports = {
  output: {
    filename: '[name].[contenthash].bundle.js',
  },
};
```

webpack.dev.js

```
module.exports = {
  output: {
    filename: '[name].bundle.js',
  },
};
```

What does this do? We are telling webpack that we would like 2 separate entry points (like the above example).

Why? With this, you can import required libraries or files that aren't modified (e.g. Bootstrap, jQuery, images, etc) inside `vendor.js` and they will be bundled together into their own chunk. Content hash remains the same, which allows the browser to cache them separately thereby reducing load time.

> ### Tip
>
> In webpack version < 4 it was common to add vendors as a separate entry point to compile it as a separate file (in combination with the `CommonsChunkPlugin`).
>
> This is discouraged in webpack 4. Instead, the `optimization.splitChunks` option takes care of separating vendors and app modules and creating a separate file. **Do not** create an entry for vendors or other stuff that is not the starting point of execution.

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 things like using `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 number of entry points increases.

> **Tip**
>
> As a rule of thumb: Use exactly one entry point for each HTML document. See the issue described here for more details.

Output

Configuring the `output` configuration options tells webpack how to write the compiled files to disk. Note that, while there can be multiple `entry` points, only one `output` configuration is specified.

Usage

The minimum requirement for the `output` property in your webpack configuration is to set its value to an object and provide an `output.filename` to use for the output file(s):

webpack.config.js

```
module.exports = {
  output: {
    filename: 'bundle.js',
  },
};
```

This configuration would output a single `bundle.js` file into the `dist` directory.

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 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
```

Advanced

Here's a more complicated example of using a CDN and hashes for assets:

config.js

```
module.exports = {
  //...
  output: {
    path: '/home/proj/cdn/assets/[fullhash]',
    publicPath: 'https://cdn.example.com/assets/[fullhash]/',
  },
};
```

In cases where the eventual `publicPath` of output files isn't known at compile time, it can be left blank and set dynamically at runtime via the `__webpack_public_path__` variable in the entry point file:

```
__webpack_public_path__ = myRuntimePublicPath;

// rest of your application entry
```

Loaders

Loaders are transformations that are applied to the source code of a module. They allow you to pre-process files as you `import` or “load” them. Thus, loaders are kind of like “tasks” in other build tools and provide a powerful way to handle front-end build steps. Loaders can transform files from a different language (like TypeScript) to JavaScript or load inline images as data URLs. Loaders even allow you to do things like `import` CSS files directly from your JavaScript modules!

Example

For example, you can use loaders to tell webpack to load a CSS file or to convert TypeScript to JavaScript. To do this, you would start by installing the loaders you need:

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

And then instruct webpack to use the `css-loader` for every `.css` file and the `ts-loader` for all `.ts` files:

webpack.config.js

```
module.exports = {
  module: {
    rules: [
      { test: /\.css$/, use: 'css-loader' },
      { test: /\.ts$/, use: 'ts-loader' },
    ],
  },
};
```

Using Loaders

There are two ways to use loaders in your application:

- Configuration (recommended): Specify them in your **webpack.config.js** file.
- Inline: Specify them explicitly in each `import` statement.

Note that loaders can be used from CLI under webpack v4, but the feature was deprecated in webpack v5.

Configuration

`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" for more information about loaders order.

```
module.exports = {
  module: {
    rules: [
```

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

Inline

It's possible to specify loaders in an `import` statement, or any equivalent "importing" method. Separate loaders from the resource with `!`. Each part is resolved relative to the current directory.

```
import Styles from 'style-loader!css-loader?modules!./styles.css';
```

It's possible to override any loaders, preLoaders and postLoaders from the configuration by prefixing the inline `import` statement:

- Prefixing with `!` will disable all configured normal loaders

  ```
  import Styles from '!style-loader!css-loader?modules!./styles.css';
  ```

- Prefixing with `!!` will disable all configured loaders (preLoaders, loaders, postLoaders)

  ```
  import Styles from '!!style-loader!css-loader?modules!./styles.css';
  ```

- Prefixing with `-!` will disable all configured preLoaders and loaders but not postLoaders

  ```
  import Styles from '-!style-loader!css-loader?modules!./styles.css';
  ```

Options can be passed with a query parameter, e.g. `?key=value&foo=bar`, or a JSON object, e.g. `?{"key":"value","foo":"bar"}`.

> ### Tip
>
> Use `module.rules` whenever possible, as this will reduce boilerplate in your source code and allow you to debug or locate a loader faster if something goes south.

Loader Features

- 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.
- Loaders can be synchronous or asynchronous.
- Loaders run in Node.js and can do everything that's possible there.
- Loaders can be configured with an `options` object (using `query` parameters to set options is still supported but has been deprecated).
- Normal modules can export a loader in addition to the normal `main` via `package.json` with the `loader` field.
- Plugins can give loaders more features.
- Loaders can emit additional arbitrary files.

Loaders provide a way to customize the output through their preprocessing functions. Users now have more flexibility to include fine-grained logic such as compression, packaging, language translations and more.

Resolving Loaders

Loaders follow the standard module resolution. In most cases it will be loaded from the module path (think `npm install`, `node_modules`).

A loader module is expected to export a function and be written in Node.js compatible JavaScript. They are most commonly managed with npm, but you can also have custom loaders as files within your application. By convention, loaders are usually named `xxx-loader` (e.g. `json-loader`). See "Writing a Loader" for more information.

Plugins

Plugins are the backbone of webpack. Webpack itself is built on the **same plugin system** that you use in your webpack configuration!

They also serve the purpose of doing **anything else** that a loader cannot do. Webpack provides many such plugins out of the box.

> **Tip**

> When consuming `webpack-sources` package in plugins, use `require('webpack').sources` instead of `require('webpack-sources')` to avoid version conflicts for persistent caching.

Anatomy

A webpack **plugin** is a JavaScript object that has an `apply` method. This `apply` method is called by the webpack compiler, giving access to the **entire** compilation lifecycle.

ConsoleLogOnBuildWebpackPlugin.js

```
const pluginName = 'ConsoleLogOnBuildWebpackPlugin';

class ConsoleLogOnBuildWebpackPlugin {
  apply(compiler) {
    compiler.hooks.run.tap(pluginName, (compilation) => {
      console.log('The webpack build process is starting!');
    });
  }
}

module.exports = ConsoleLogOnBuildWebpackPlugin;
```

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

Usage

Since **plugins** can take arguments/options, you must pass a `new` instance to the `plugins` property in your webpack configuration.

Depending on how you are using webpack, there are multiple ways to use plugins.

Configuration

webpack.config.js

```
const HtmlWebpackPlugin = require('html-webpack-plugin');
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' }),
  ],
};
```

The `ProgressPlugin` is used to customize how progress should be reported during compilation, and `HtmlWebpackPlugin` will generate a HTML file including the `my-first-webpack.bundle.js` file using a `script` tag.

Node API

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) {
  // ...
});
```

> ### Tip
>
> Did you know: The example seen above is extremely similar to the webpack runtime itself! There are lots of great usage examples hiding in the webpack source code that you can apply to your own configurations and scripts!

Configuration

You may have noticed that few webpack configurations look exactly alike. This is because **webpack's configuration file is a JavaScript file that exports a webpack configuration.** This configuration is then processed by webpack based upon its defined properties.

Because it's a standard Node.js CommonJS module, you **can do the following**:

- Import other files via `require(...)`
- use utilities on npm via `require(...)`
- use JavaScript control flow expressions, e.g. the `?:` operator
- use constants or variables for often used values
- write and execute functions to generate a part of the configuration

Use these features when appropriate.

While they are technically feasible, **the following practices should be avoided**:

- Access CLI arguments, when using the webpack CLI (instead write your own CLI, or use `--env`)
- Export non-deterministic values (calling webpack twice should result in the same output files)
- Write long configurations (instead split the configuration into multiple files)

> **Tip**
>
> 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.

The examples below describe how webpack's configuration can be both expressive and configurable because *it is code*:

Introductory Configuration

webpack.config.js

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

module.exports = {
  mode: 'development',
  entry: './foo.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'foo.bundle.js',
  },
};
```

See: Configuration section for all supported configuration options

Multiple Targets

Along with exporting a single configuration as an object, function or Promise, you can export multiple configurations.

See: Exporting multiple configurations

Using other Configuration Languages

Webpack accepts configuration files written in multiple programming and data languages.

See: Configuration Languages

Modules

In modular programming, developers break programs up into discrete chunks of functionality called a *module*.

Each module has a smaller surface area than a full program, making verification, debugging, and testing trivial. Well-written *modules* provide solid abstractions and encapsulation boundaries, so that each module has a coherent design and a clear purpose within the overall application.

Node.js has supported modular programming almost since its inception. On the web, however, support for *modules* has been slow to arrive. Multiple tools exist that support modular JavaScript on the web, with a variety of benefits and limitations. Webpack builds on lessons learned from these systems and applies the concept of *modules* to any file in your project.

What is a webpack Module

In contrast to Node.js modules, webpack *modules* can express their *dependencies* in a variety of ways. A few examples are:

- An ES2015 `import` statement
- A CommonJS `require()` statement
- An AMD `define` and `require` statement
- An `@import` statement inside of a css/sass/less file.

- An image url in a stylesheet `url(...)` or HTML `<img src=...>` file.

Supported Module Types

Webpack supports the following module types natively:

- ECMAScript modules
- CommonJS modules
- AMD modules
- Assets
- WebAssembly modules

In addition to that webpack supports modules written in a variety of languages and preprocessors via *loaders*. *Loaders* describe to webpack **how** to process non-native *modules* and include these *dependencies* into your *bundles*. The webpack community has built *loaders* for a wide variety of popular languages and language processors, including:

- CoffeeScript
- TypeScript
- ESNext (Babel)
- Sass
- Less
- Stylus
- Elm

And many others! Overall, webpack provides a powerful and rich API for customization that allows one to use webpack for **any stack**, while staying **non-opinionated** about your development, testing, and production workflows.

For a full list, see **the list of loaders** or **write your own**.

Module Resolution

A resolver is a library which helps in locating a module by its absolute path. A module can be required as a dependency from another module as:

```
import foo from 'path/to/module';
// or
require('path/to/module');
```

The dependency module can be from the application code or a third-party library. The resolver helps webpack find the module code that needs to be included in the bundle for every such `require` / `import` statement. webpack uses enhanced-resolve to resolve file paths while bundling modules.

Resolving rules in webpack

Using `enhanced-resolve`, webpack can resolve three kinds of file paths:

Absolute paths

```
import '/home/me/file';

import 'C:\\Users\\me\\file';
```

Since we already have the absolute path to the file, no further resolution is required.

Relative paths

```
import '../src/file1';
import './file2';
```

In this case, the directory of the source file where the `import` or `require` occurs is taken to be the context directory. The relative path specified in the `import/require` is joined to this context path to produce the absolute path to the module.

Module paths

```
import 'module';
import 'module/lib/file';
```

Modules are searched for inside all directories specified in `resolve.modules`. You can replace the original module path by an alternate path by creating an alias for it using the `resolve.alias` configuration option.

- If the package contains a `package.json` file, then fields specified in `resolve.exportsFields` configuration options are looked up in order, and the first such field in `package.json` determines the available exports from the package according to the package exports guideline.

Once the path is resolved based on the above rule, the resolver checks to see if the path points to a file or a directory. If the path points to a file:

- If the path has a file extension, then the file is bundled straightaway.
- Otherwise, the file extension is resolved using the `resolve.extensions` option, which tells the resolver which extensions are acceptable for resolution e.g. `.js , .jsx .`

If the path points to a folder, then the following steps are taken to find the right file with the right extension:

- If the folder contains a `package.json` file, then fields specified in `resolve.mainFields` configuration option are looked up in order, and the first such field in `package.json` determines the file path.
- If there is no `package.json` or if the `resolve.mainFields` do not return a valid path, file names specified in the `resolve.mainFiles` configuration option are looked for in order, to see if a matching filename exists in the imported/required directory.
- The file extension is then resolved in a similar way using the `resolve.extensions` option.

Webpack provides reasonable defaults for these options depending on your build target.

Resolving Loaders

This follows the same rules as those specified for file resolution. But the `resolveLoader` configuration option can be used to have separate resolution rules for loaders.

Caching

Every filesystem access is cached so that multiple parallel or serial requests to the same file occur faster. In watch mode, only modified files are evicted from the cache. If watch mode is off, then the cache gets purged before every compilation.

See Resolve API to learn more about the configuration options mentioned above.

Module Federation

Motivation

Multiple separate builds should form a single application. These separate builds act like containers and can expose and consume code between builds, creating a single, unified application.

This is often known as Micro-Frontends, but is not limited to that.

> **Live Preview**
>
> Check out this live module federation example on StackBlitz.
>
> Open in StackBlitz

Low-level concepts

We distinguish between local and remote modules. Local modules are regular modules that are part of the current build. Remote modules are modules that are not part of the current build but are loaded at runtime from a remote container.

Loading remote modules is considered an asynchronous operation. When using a remote module, these asynchronous operations will be placed in the next chunk loading operation(s) that are between the remote module and the entrypoint. It's not possible to use a remote module without a chunk loading operation.

A chunk loading operation is usually an `import()` call, but older constructs like `require.ensure` or `require([...])` are supported as well.

A container is created through a container entry, which exposes asynchronous access to the specific modules. The exposed access is separated into two steps:

1. loading the module (asynchronous)
2. evaluating the module (synchronous).

Step 1 will be done during the chunk loading. Step 2 will be done during the module evaluation interleaved with other (local and remote) modules. This way, evaluation order is unaffected by converting a module from local to remote or the other way around.

It is possible to nest containers. Containers can use modules from other containers. Circular dependencies between containers are also possible.

High-level concepts

Each build acts as a container and also consumes other builds as containers. This way, each build is able to access any other exposed module by loading it from its container.

Shared modules are modules that are both overridable and provided as overrides to nested containers. They usually point to the same module in each build, e.g., the same library.

The `packageName` option allows setting a package name to look for a `requiredVersion`. It is automatically inferred for the module requests by default, set `requiredVersion` to `false` when automatic infer should be disabled.

Building blocks

ContainerPlugin (low level)

This plugin creates an additional container entry with the specified exposed modules.

ContainerReferencePlugin (low level)

This plugin adds specific references to containers as externals and allows to import remote modules from these containers. It also calls the `override` API of these containers to provide overrides to them. Local overrides (via `__webpack_override__` or `override` API when build is also a container) and specified overrides are provided to all referenced containers.

ModuleFederationPlugin (high level)

`ModuleFederationPlugin` combines `ContainerPlugin` and `ContainerReferencePlugin`.

Concept goals

- It should be possible to expose and consume any module type that webpack supports.
- Chunk loading should load everything needed in parallel (web: single round-trip to server).
- Control from consumer to container
  - Overriding modules is a one-directional operation.
  - Sibling containers cannot override each other's modules.
- Concept should be environment-independent.
  - Usable in web, Node.js, etc.
- Relative and absolute request in shared:
  - Will always be provided, even if not used.
  - Will resolve relative to `config.context`.
  - Does not use a `requiredVersion` by default.
- Module requests in shared:
  - Are only provided when they are used.

    - Will match all used equal module requests in your build.
    - Will provide all matching modules.
    - Will extract `requiredVersion` from package.json at this position in the graph.
    - Could provide and consume multiple different versions when you have nested node_modules.
- Module requests with trailing `/` in shared will match all module requests with this prefix.

Use cases

Separate builds per page

Each page of a Single Page Application is exposed from container build in a separate build. The application shell is also a separate build referencing all pages as remote modules. This way each page can be separately deployed. The application shell is deployed when routes are updated or new routes are added. The application shell defines commonly used libraries as shared modules to avoid duplication of them in the page builds.

Components library as container

Many applications share a common components library which could be built as a container with each component exposed. Each application consumes components from the components library container. Changes to the components library can be separately deployed without the need to re-deploy all applications. The application automatically uses the up-to-date version of the components library.

Dynamic Remote Containers

The container interface supports `get` and `init` methods. `init` is an `async` compatible method that is called with one argument: the shared scope object. This object is used as a shared scope in the remote container and is filled with the provided modules from a host. It can be leveraged to connect remote containers to a host container dynamically at runtime.

init.js

```
(async () => {
  // Initializes the shared scope. Fills it with known provided modules from this build and all
  await __webpack_init_sharing__('default');
  const container = window.someContainer; // or get the container somewhere else
  // Initialize the container, it may provide shared modules
  await container.init(__webpack_share_scopes__.default);
```

```
  const module = await container.get('./module');
})();
```

The container tries to provide shared modules, but if the shared module has already been used, a warning and the provided shared module will be ignored. The container might still use it as a fallback.

This way you could dynamically load an A/B test which provides a different version of a shared module.

> **Tip**
>
> Ensure you have loaded the container before attempting to dynamically connect a remote container.

Example:

init.js

```
function loadComponent(scope, module) {
  return async () => {
    // Initializes the shared scope. Fills it with known provided modules from this build and al
    await __webpack_init_sharing__('default');
    const container = window[scope]; // or get the container somewhere else
    // Initialize the container, it may provide shared modules
    await container.init(__webpack_share_scopes__.default);
    const factory = await window[scope].get(module);
    const Module = factory();
    return Module;
  };
}

loadComponent('abtests', 'test123');
```

See full implementation

Promise Based Dynamic Remotes

Generally, remotes are configured using URL's like in this example:

```
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'host',
      remotes: {
        app1: 'app1@http://localhost:3001/remoteEntry.js',
      },
    }),
```

```
  ],
};
```

But you can also pass in a promise to this remote, which will be resolved at runtime. You should resolve this promise with any module that fits the `get/init` interface described above. For example, if you wanted to pass in which version of a federated module you should use, via a query parameter you could do something like the following:

```
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'host',
      remotes: {
        app1: `promise new Promise(resolve => {
      const urlParams = new URLSearchParams(window.location.search)
      const version = urlParams.get('app1VersionParam')
      // This part depends on how you plan on hosting and versioning your federated modules
      const remoteUrlWithVersion = 'http://localhost:3001/' + version + '/remoteEntry.js'
      const script = document.createElement('script')
      script.src = remoteUrlWithVersion
      script.onload = () => {
        // the injected script has loaded and is available on window
        // we can now resolve this Promise
        const proxy = {
          get: (request) => window.app1.get(request),
          init: (arg) => {
            try {
              return window.app1.init(arg)
            } catch(e) {
              console.log('remote container already initialized')
            }
          }
        }
        resolve(proxy)
      }
      // inject this script with the src set to the versioned remoteEntry.js
      document.head.appendChild(script);
    })
    `,
      },
      // ...
    }),
  ],
};
```

Note that when using this API you *have* to resolve an object which contains the get/init API.

Dynamic Public Path

Offer a host API to set the publicPath

One could allow the host to set the publicPath of a remote module at runtime by exposing a method from that remote module.

This approach is particularly helpful when you mount independently deployed child applications on the sub path of the host domain.

Scenario:

You have a host app hosted on `https://my-host.com/app/*` and a child app hosted on `https://foo-app.com` . The child app is also mounted on the host domain, hence, `https://foo-app.com` is expected to be accessible via `https://my-host.com/app/foo-app` and `https://my-host.com/app/foo-app/*` requests are redirected to `https://foo-app.com/*` via a proxy.

Example:

webpack.config.js (remote)

```
module.exports = {
  entry: {
    remote: './public-path',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'remote', // this name needs to match with the entry name
      exposes: ['./public-path'],
      // ...
    }),
  ],
};
```

public-path.js (remote)

```
export function set(value) {
  __webpack_public_path__ = value;
}
```

src/index.js (host)

```
const publicPath = await import('remote/public-path');
publicPath.set('/your-public-path');

//bootstrap app  e.g. import('./bootstrap.js')
```

Infer publicPath from script

One could infer the publicPath from the script tag from `document.currentScript.src` and set it with the `__webpack_public_path__` module variable at runtime.

Example:

webpack.config.js (remote)

```
module.exports = {
  entry: {
    remote: './setup-public-path',
  },
  plugins: [
    new ModuleFederationPlugin({
      name: 'remote', // this name needs to match with the entry name
      // ...
    }),
  ],
};
```

setup-public-path.js (remote)

```
// derive the publicPath with your own logic and set it with the __webpack_public_path__ API
__webpack_public_path__ = document.currentScript.src + '/../';
```

> **Tip**
>
> There is also an `'auto'` value available to `output.publicPath` which automatically determines the publicPath for you.

Troubleshooting

`Uncaught Error: Shared module is not available for eager consumption`

The application is eagerly executing an application that is operating as an omnidirectional host. There are options to choose from:

You can set the dependency as eager inside the advanced API of Module Federation, which doesn't put the modules in an async chunk, but provides them synchronously. This allows us to use these shared modules in the initial chunk. But be careful as all provided and fallback modules will always be downloaded. It's recommended to provide it only at one point of your application, e.g. the shell.

We strongly recommend using an asynchronous boundary. It will split out the initialization code of a larger chunk to avoid any additional round trips and improve performance in general.

For example, your entry looked like this:

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';
ReactDOM.render(<App />, document.getElementById('root'));
```

Let's create `bootstrap.js` file and move contents of the entry into it, and import that bootstrap into the entry:

index.js

```
+ import('./bootstrap');
- import React from 'react';
- import ReactDOM from 'react-dom';
- import App from './App';
- ReactDOM.render(<App />, document.getElementById('root'));
```

bootstrap.js

```
+ import React from 'react';
+ import ReactDOM from 'react-dom';
+ import App from './App';
+ ReactDOM.render(<App />, document.getElementById('root'));
```

This method works but can have limitations or drawbacks.

Setting `eager: true` for dependency via the `ModuleFederationPlugin`

webpack.config.js

```
// ...
new ModuleFederationPlugin({
  shared: {
    ...deps,
    react: {
      eager: true,
    },
  },
});
```

Uncaught Error: Module "./Button" does not exist in container.

It likely does not say `"./Button"`, but the error message will look similar. This issue is typically seen if you are upgrading from webpack beta.16 to webpack beta.17.

Within ModuleFederationPlugin. Change the exposes from:

```
new ModuleFederationPlugin({
  exposes: {
-   'Button': './src/Button'
+   './Button':'./src/Button'
  }
});
```

`Uncaught TypeError: fn is not a function`

You are likely missing the remote container, make sure it's added. If you have the container loaded for the remote you are trying to consume, but still see this error, add the host container's remote container file to the HTML as well.

Collision between modules from different remotes

If you're going to load multiple modules from different remotes, it's advised to set the `output.uniqueName` option for your remote builds to avoid collisions between multiple webpack runtimes.

Dependency Graph

Any time one file depends on another, webpack treats this as a *dependency*. This allows webpack to take non-code assets, such as images or web fonts, and also provide them as *dependencies* for your application.

When webpack processes your application, it starts from a list of modules defined on the command line or in its configuration file. Starting from these *entry points*, webpack recursively builds a *dependency graph* that includes every module your application needs, then bundles all of those modules into a small number of *bundles* - often, only one - to be loaded by the browser.

> **Tip**
>
> Bundling your application is especially powerful for *HTTP/1.1* clients, as it minimizes the number of times your app has to wait while the browser starts a new request. For *HTTP/2*, you can also use Code Splitting to achieve best results.

Targets

Because JavaScript can be written for both server and browser, webpack offers multiple deployment *targets* that you can set in your webpack configuration.

> **Warning**
>
> The webpack `target` property is not to be confused with the `output.libraryTarget` property. For more information see our guide on the `output` property.

Usage

To set the `target` property, you set the target value in your webpack config:

webpack.config.js

```
module.exports = {
  target: 'node',
};
```

In the example above, using `node` webpack will compile for usage in a Node.js-like environment (uses Node.js `require` to load chunks and not touch any built in modules like `fs` or `path`).

Each *target* has a variety of deployment/environment specific additions, support to fit its needs. See what targets are available.

> **Todo**
>
> Further expansion for other popular target values

Multiple Targets

Although webpack does **not** support multiple strings being passed into the `target` property, you can create an isomorphic library by bundling two separate configurations:

webpack.config.js

```
const path = require('path');
const serverConfig = {
  target: 'node',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'lib.node.js',
  },
  //…
};

const clientConfig = {
```

```
  target: 'web', // <=== can be omitted as default is 'web'
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'lib.js',
  },
  //…
};

module.exports = [serverConfig, clientConfig];
```

The example above will create a `lib.js` and `lib.node.js` file in your `dist` folder.

Resources

As seen from the options above, there are multiple deployment *targets* that you can choose from. Below is a list of examples and resources that you can refer to.

- **compare-webpack-target-bundles**: A great resource for testing and viewing different webpack *targets*. Also great for bug reporting.
- **Boilerplate of Electron-React Application**: A good example of a build process for electron's main process and renderer process.

> **Todo**
>
> Need to find up to date examples of these webpack targets being used in live code or boilerplates.

The Manifest

In a typical application or site built with webpack, there are three main types of code:

1. The source code you, and maybe your team, have written.
2. Any third-party library or "vendor" code your source is dependent on.
3. A webpack runtime and **manifest** that conducts the interaction of all modules.

This article will focus on the last of these three parts: the runtime and, in particular, the manifest.

Runtime

The runtime, along with the manifest data, is all the code webpack needs to connect your modularized application while it's running in the browser. It contains the loading and resolving logic needed to

connect your modules as they interact. This includes connecting modules that have already been loaded into the browser as well as logic to lazy-load the ones that haven't.

Manifest

Once your application hits the browser in the form of `index.html` file, some bundles and a variety of other assets required by your application must be loaded and linked somehow. That `/src` directory you meticulously laid out is now bundled, minified and maybe even split into smaller chunks for lazy-loading by webpack's `optimization` . So how does webpack manage the interaction between all of your required modules? This is where the manifest data comes in...

As the compiler enters, resolves, and maps out your application, it keeps detailed notes on all your modules. This collection of data is called the "Manifest," and it's what the runtime will use to resolve and load modules once they've been bundled and shipped to the browser. No matter which module syntax you have chosen, those `import` or `require` statements have now become `__webpack_require__` methods that point to module identifiers. Using the data in the manifest, the runtime will be able to find out where to retrieve the modules behind the identifiers.

The Problem

So now you have a little bit of insight about how webpack works behind the scenes. "But, how does this affect me?", you might ask. Most of the time, it doesn't. The runtime will do its thing, utilizing the manifest, and everything will appear to magically work once your application hits the browser. However, if you decide to improve the performance of your projects by utilizing browser caching, this process will all of a sudden become an important thing to understand.

By using content hashes within your bundle file names, you can indicate to the browser when the content of a file has changed, thus invalidating the cache. Once you start doing this though, you'll immediately notice some funny behavior. Certain hashes change even when their content apparently does not. This is caused by the injection of the runtime and manifest, which changes every build.

See the manifest section of our *Output management* guide to learn how to extract the manifest, and read the guides below to learn more about the intricacies of long term caching.

Hot Module Replacement

Hot Module Replacement (HMR) exchanges, adds, or removes modules while an application is running, without a full reload. This can significantly speed up development in a few ways:

- Retain application state which is lost during a full reload.
- Save valuable development time by only updating what's changed.

- Instantly update the browser when modifications are made to CSS/JS in the source code, which is almost comparable to changing styles directly in the browser's dev tools.

How It Works

Let's go through some different viewpoints to understand exactly how HMR works...

In the Application

The following steps allow modules to be swapped in and out of an application:

1. The application asks the HMR runtime to check for updates.
2. The runtime asynchronously downloads the updates and notifies the application.
3. The application then asks the runtime to apply the updates.
4. The runtime synchronously applies the updates.

You can set up HMR so that this process happens automatically, or you can choose to require user interaction for updates to occur.

In the Compiler

In addition to normal assets, the compiler needs to emit an "update" to allow updating from the previous version to the new version. The "update" consists of two parts:

1. The updated manifest (JSON)
2. One or more updated chunks (JavaScript)

The manifest contains the new compilation hash and a list of all updated chunks. Each of these chunks contains the new code for all updated modules (or a flag indicating that the module was removed).

The compiler ensures that module IDs and chunk IDs are consistent between these builds. It typically stores these IDs in memory (e.g. with webpack-dev-server), but it's also possible to store them in a JSON file.

In a Module

HMR is an opt-in feature that only affects modules containing HMR code. One example would be patching styling through the `style-loader`. In order for patching to work, the `style-loader` implements the HMR interface; when it receives an update through HMR, it replaces the old styles with the new ones.

Similarly, when implementing the HMR interface in a module, you can describe what should happen when the module is updated. However, in most cases, it's not mandatory to write HMR code in every module. If a module has no HMR handlers, the update bubbles up. This means that a single handler can update a complete module tree. If a single module from the tree is updated, the entire set of dependencies is reloaded.

See the HMR API page for details on the `module.hot` interface.

In the Runtime

Here things get a bit more technical... if you're not interested in the internals, feel free to jump to the HMR API page or HMR guide.

For the module system runtime, additional code is emitted to track module `parents` and `children`. On the management side, the runtime supports two methods: `check` and `apply`.

A `check` makes an HTTP request to the update manifest. If this request fails, there is no update available. If it succeeds, the list of updated chunks is compared to the list of currently loaded chunks. For each loaded chunk, the corresponding update chunk is downloaded. All module updates are stored in the runtime. When all update chunks have been downloaded and are ready to be applied, the runtime switches into the `ready` state.

The `apply` method flags all updated modules as invalid. For each invalid module, there needs to be an update handler in the module or in its parent(s). Otherwise, the invalid flag bubbles up and invalidates parent(s) as well. Each bubble continues until the app's entry point or a module with an update handler is reached (whichever comes first). If it bubbles up from an entry point, the process fails.

Afterwards, all invalid modules are disposed (via the dispose handler) and unloaded. The current hash is then updated and all `accept` handlers are called. The runtime switches back to the `idle` state and everything continues as normal.

Get Started

HMR can be used in development as a LiveReload replacement. webpack-dev-server supports a `hot` mode in which it tries to update with HMR before trying to reload the whole page. See the Hot Module Replacement guide for details.

> **Tip**

> As with many other features, webpack's power lies in its customizability. There are *many* ways of configuring HMR depending on the needs of a particular project. However, for most purposes, `webpack-dev-server` is a good fit and will allow you to get started with HMR quickly.

Why webpack

To understand why you should use webpack, let's recap how we used JavaScript on the web before bundlers were a thing.

There are two ways to run JavaScript in a browser. First, include a script for each functionality; this solution is hard to scale because loading too many scripts can cause a network bottleneck. The second option is to use a big `.js` file containing all your project code, but this leads to problems in scope, size, readability and maintainability.

IIFEs - Immediately invoked function expressions

IIFEs solve scoping issues for large projects; when script files are wrapped by an IIFE, you can safely concatenate or safely combine files without worrying about scope collision.

The use of IIFEs led to tools like Make, Gulp, Grunt, Broccoli or Brunch. These tools are known as task runners, and they concatenate all your project files together.

However, changing one file means you have to rebuild the whole thing. Concatenating makes it easier to reuse scripts across files but makes build optimizations more difficult. How can you find out if code is actually being used or not?

Even if you only use a single function from lodash, you have to add the entire library and then squish it together. How do you treeshake the dependencies on your code? Lazy loading chunks of code can be hard to do at scale and requires a lot of manual work from the developer.

Birth of JavaScript Modules happened thanks to Node.js

Webpack runs on Node.js, a JavaScript runtime that can be used in computers and servers outside a browser environment.

When Node.js was released a new era started, and it came with new challenges. Now that JavaScript is not running in a browser, how are Node applications supposed to load new chunks of code? There are no html files and script tags that can be added to it.

CommonJS came out and introduced `require`, which allows you to load and use a module in the current file. This solved scope issues out of the box by importing each module as it was needed.

npm + Node.js + modules – mass distribution

JavaScript is taking over the world as a language, as a platform and as a way to rapidly develop and create fast applications.

But there is no browser support for CommonJS. There are no live bindings. There are problems with circular references. Synchronous module resolution and loading is slow. While CommonJS was a great solution for Node.js projects, browsers didn't support modules, so bundlers and tools like Browserify, RequireJS and SystemJS were created, allowing us to write CommonJS modules that run in a browser.

ESM - ECMAScript Modules

The good news for web projects is that modules are becoming an official feature in the ECMAScript standard. However, browser support is incomplete and bundling is still faster and currently recommended over these early module implementations.

Automatic Dependency Collection

Old school Task Runners and even Google Closure Compiler requires you to manually declare all dependencies upfront. While bundlers like webpack automatically build and infer your dependency graph based on what is imported and exported. This along with other plugins and loaders make for a great developer experience.

Wouldn't it be nice...

...to have something that will not only let us write modules but also support any module format (at least until we get to ESM) and handle resources and assets at the same time?

This is why webpack exists. It's a tool that lets you bundle your JavaScript applications (supporting both ESM and CommonJS), and it can be extended to support many different assets such as images, fonts and stylesheets.

Webpack cares about performance and load times; it's always improving or adding new features, such as async chunk loading and prefetching, to deliver the best possible experience for your project and your users.

Under The Hood

> *This section describes webpack internals and can be useful for plugin developers*

The bundling is a function that takes some files and emits others.

But between input and output, it also has modules, entry points, chunks, chunk groups, and many other intermediate parts.

The main parts

Every file used in your project is a Module

./index.js

```
import app from './app.js';
```

./app.js

```
export default 'the app';
```

By using each other, the modules form a graph (`ModuleGraph`).

During the bundling process, modules are combined into chunks. Chunks combine into chunk groups and form a graph (`ChunkGraph`) interconnected through modules. When you describe an entry point - under the hood, you create a chunk group with one chunk.

./webpack.config.js

```
module.exports = {
  entry: './index.js',
};
```

One chunk group with the `main` name created (`main` is the default name for an entry point). This chunk group contains `./index.js` module. As the parser handles imports inside `./index.js` new modules are added into this chunk.

Another example:

./webpack.config.js

```
module.exports = {
  entry: {
    home: './home.js',
```

```
    about: './about.js',
  },
};
```

Two chunk groups with names `home` and `about` are created. Each of them has a chunk with a module - `./home.js` for `home` and `./about.js` for `about`

> *There might be more than one chunk in a chunk group. For example SplitChunksPlugin splits a chunk into one or more chunks.*

Chunks

Chunks come in two forms:

- `initial` is the main chunk for the entry point. This chunk contains all the modules and their dependencies that you specify for an entry point.
- `non-initial` is a chunk that may be lazy-loaded. It may appear when dynamic import or SplitChunksPlugin is being used.

Each chunk has a corresponding **asset**. The assets are the output files - the result of bundling.

webpack.config.js

```
module.exports = {
  entry: './src/index.jsx',
};
```

./src/index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

import('./app.jsx').then((App) => {
  ReactDOM.render(<App />, root);
});
```

Initial chunk with name `main` is created. It contains:

- `./src/index.jsx`
- `react`
- `react-dom`

and all their dependencies, except `./app.jsx`

Non-initial chunk for `./app.jsx` is created as this module is imported dynamically.

Output:

- `/dist/main.js` - an `initial` chunk
- `/dist/394.js` - `non-initial` chunk

By default, there is no name for `non-initial` chunks so that a unique ID is used instead of a name. When using dynamic import we may specify a chunk name explicitly by using a "magic" comment:

```
import(
  /* webpackChunkName: "app" */
  './app.jsx'
).then((App) => {
  ReactDOM.render(<App />, root);
});
```

Output:

- `/dist/main.js` - an `initial` chunk
- `/dist/app.js` - `non-initial` chunk

Output

The names of the output files are affected by the two fields in the config:

- `output.filename` - for `initial` chunk files
- `output.chunkFilename` - for `non-initial` chunk files
- In some cases chunks are used `initial` and `non-initial`. In those cases `output.filename` is used.

A few placeholders are available in these fields. Most often:

- `[id]` - chunk id (e.g. `[id].js -> 485.js`)
- `[name]` - chunk name (e.g. `[name].js -> app.js`). If a chunk has no name, then its id will be used
- `[contenthash]` - md4-hash of the output file content (e.g. `[contenthash].js -> 4ea6ff1de66c537eb9b2.js`)