Code Splitting

> *This guide extends the examples provided in Getting Started and Output Management. Please make sure you are at least familiar with the examples provided in them.*

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

There are three general approaches to code splitting available:

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

Entry Points

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

project

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

another-module.js

```
import _ from 'lodash';

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

webpack.config.js

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

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

This will yield the following build result:

```
...
            Asset     Size   Chunks             Chunk Names
another.bundle.js  550 KiB  another  [emitted]  another
  index.bundle.js  550 KiB    index  [emitted]  index
Entrypoint index = index.bundle.js
Entrypoint another = another.bundle.js
...
```

As mentioned there are some pitfalls to this approach:

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

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

Prevent Duplication

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

> *The `CommonsChunkPlugin` has been removed in webpack v4 legato. To learn how chunks are treated in the latest version, check out the `SplitChunksPlugin`.*

webpack.config.js

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

module.exports = {
  mode: 'development',
  entry: {
```

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

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

```
...
                         Asset      Size                  Chunks             Chunk Names
             another.bundle.js  5.95 KiB                 another  [emitted]  another
               index.bundle.js  5.89 KiB                   index  [emitted]  index
vendors~another~index.bundle.js   547 KiB  vendors~another~index  [emitted]  vendors~another~ind
Entrypoint index = vendors~another~index.bundle.js index.bundle.js
Entrypoint another = vendors~another~index.bundle.js another.bundle.js
...
```

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

- `mini-css-extract-plugin`: Useful for splitting CSS out from the main application.
- `bundle-loader`: Used to split code and lazy load the resulting bundles.
- `promise-loader`: Similar to the `bundle-loader` but uses promises.

Dynamic Imports

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

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

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

webpack.config.js

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

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

Note the use of `chunkFilename`, which determines the name of non-entry chunk files. For more information on `chunkFilename`, see output documentation. We'll also update our project to remove the now unused files:

project

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

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

src/index.js

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

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

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

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

Note the use of `webpackChunkName` in the comment. This will cause our separate bundle to be named `lodash.bundle.js` instead of just `[id].bundle.js`. For more information on `webpackChunkName` and the other available options, see the `import()` documentation. Let's run webpack to see `lodash` separated out to a separate bundle:

```
...
                  Asset      Size          Chunks             Chunk Names
        index.bundle.js  7.88 KiB           index  [emitted]  index
vendors~lodash.bundle.js   547 KiB  vendors~lodash  [emitted]  vendors~lodash
Entrypoint index = index.bundle.js
...
```

As `import()` returns a promise, it can be used with `async` functions. However, this requires using a pre-processor like Babel and the Syntax Dynamic Import Babel Plugin. Here's how it would simplify the code:

src/index.js

```
- function getComponent() {
+ async function getComponent() {
-   return import(/* webpackChunkName: "lodash" */ 'lodash').then({ default: _ } => {
-     const element = document.createElement('div');
-
-     element.innerHTML = _.join(['Hello', 'webpack'], ' ');
-
-     return element;
-
-   }).catch(error => 'An error occurred while loading the component');
+   const element = document.createElement('div');
+   const { default: _ } = await import(/* webpackChunkName: "lodash" */ 'lodash');
+
+   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+
```

```
+   return element;
  }

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

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

Prefetching/Preloading modules

webpack 4.6.0+ adds support for prefetching and preloading.

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

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

Simple prefetch example can be having a `HomePage` component, which renders a `LoginButton` component which then on demand loads a `LoginModal` component after being clicked.

LoginButton.js

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

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

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

Preload directive has a bunch of differences compared to prefetch:

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

Simple preload example can be having a `Component` which always depends on a big library that should be in a separate chunk.

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

ChartComponent.js

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

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

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

Bundle Analysis

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

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

Next Steps

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