Make use of long-term caching

By Ivan Akulov
Contracting software engineer · Blogging, open source, performance, UX

The next thing (after optimizing the app size) that improves the app loading time is caching. Use it to keep parts of the app on the client and avoid re-downloading them every time.

Use bundle versioning and cache headers

The common approach of doing caching is to:

1. tell the browser to cache a file for a very long time (e.g., a year):

   ```
   # Server header
   Cache-Control: max-age=31536000
   ```

   Note: If you aren't familiar what `Cache-Control` does, see Jake Archibald's excellent post on caching best practices.

2. and rename the file when it's changed to force the re-download:

   ```
   <!-- Before the change -->
   <script src="./index-v15.js"></script>

   <!-- After the change -->
   <script src="./index-v16.js"></script>
   ```

This approach tells the browser to download the JS file, cache it and use the cached copy. The browser will only hit the network only if the file name changes (or if a year passes).

With webpack, you do the same, but instead of a version number, you specify the file hash. To include the hash into the file name, use [chunkhash]:

```
// webpack.config.js
module.exports = {
  entry: './index.js',
  output: {
    filename: 'bundle.[chunkhash].js',
        // → bundle.8e0d62a03.js
```

```
  },
};
```

Note: Webpack could generate a different hash even if the bundle stays the same – e.g. if you rename a file or compile the bundle under a different OS. This is a bug, and there's no clear solution yet. See the discussion on GitHub

If you need the file name to send it to the client, use either the `HtmlWebpackPlugin` or the `WebpackManifestPlugin`.

The `HtmlWebpackPlugin` is a simple, but less flexible approach. During compilation, this plugin generates an HTML file which includes all compiled resources. If your server logic isn't complex, then it should be enough for you:

```
<!-- index.html -->
<!doctype html>
<!-- ... -->
<script src="bundle.8e0d62a03.js"></script>
```

The `WebpackManifestPlugin` is a more flexible approach which is useful if you have a complex server part. During the build, it generates a JSON file with a mapping between file names without hash and file names with hash. Use this JSON on the server to find out which file to work with:

```
// manifest.json
{
  "bundle.js": "bundle.8e0d62a03.js"
}
```

Further reading

- Jake Archibald about caching best practices

Extract dependencies and runtime into a separate file

Dependencies

App dependencies tend to change less often than the actual app code. If you move them into a separate file, the browser will be able to cache them separately – and won't re-download them each time the app code changes.

Key Term: In webpack terminology, separate files with the app code are called *chunks*. We'll use this name later.

To extract dependencies into a separate chunk, perform three steps:

1. Replace the output filename with `[name].[chunkname].js`:

   ```
   // webpack.config.js
   module.exports = {
     output: {
       // Before
       filename: 'bundle.[chunkhash].js',
       // After
       filename: '[name].[chunkhash].js',
     },
   };
   ```

   When webpack builds the app, it replaces `[name]` with a name of a chunk. If we don't add the `[name]` part, we'll have to differentiate between chunks by their hash – which is pretty hard!

2. Convert the `entry` field into an object:

   ```
   // webpack.config.js
   module.exports = {
     // Before
     entry: './index.js',
     // After
     entry: {
       main: './index.js',
     },
   };
   ```

   In this snippet, "main" is a name of a chunk. This name will be substituted in place of `[name]` from step 1.

   By now, if you build the app, this chunk will include the whole app code – just like we haven't done these steps. But this will change in a sec.

3. **In webpack 4,** add the `optimization.splitChunks.chunks: 'all'` option into your webpack config:

   ```
   // webpack.config.js (for webpack 4)
   module.exports = {
     optimization: {
       splitChunks: {
   ```

```
        chunks: 'all',
      }
    },
  };
```

This option enables smart code splitting. With it, webpack would extract the vendor code if it gets larger than 30 kB (before minification and gzip). It would also extract the common code – this is useful if your build produces several bundles (e.g. if you split your app into routes).

In webpack 3, add the `CommonsChunkPlugin`:

```
// webpack.config.js (for webpack 3)
module.exports = {
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      // A name of the chunk that will include the dependencies.
      // This name is substituted in place of [name] from step 1
      name: 'vendor',

      // A function that determines which modules to include into this chunk
      minChunks: module => module.context &&
        module.context.includes('node_modules'),
    }),
  ],
};
```

This plugin takes all modules which paths include `node_modules` and moves them into a separate file called `vendor.[chunkhash].js`.

After these changes, each build will generate two files instead of one: `main.[chunkhash].js` and `vendor.[chunkhash].js` (`vendors~main.[chunkhash].js` for webpack 4). In case of webpack 4, the vendor bundle might not be generated if dependencies are small – and that's fine:

```
$ webpack
Hash: ac01483e8fec1fa70676
Version: webpack 3.8.1
Time: 3816ms
                           Asset   Size  Chunks             Chunk Names
  ./main.00bab6fd3100008a42b0.js  82 kB       0  [emitted]  main
./vendor.d9e134771799ecdf9483.js  47 kB       1  [emitted]  vendor
```

The browser would cache these files separately – and redownload only code that changes.

Webpack runtime code

Unfortunately, extracting just the vendor code is not enough. If you try to change something in the app code:

```
// index.js
…
…

// E.g. add this:
console.log('Wat');
```

you'll notice that the `vendor` hash also changes:

```
                            Asset   Size  Chunks             Chunk Names
./vendor.d9e134771799ecdf9483.js  47 kB       1  [emitted]  vendor
```

↓

```
                            Asset   Size  Chunks             Chunk Names
./vendor.e6ea4504d61a1cc1c60b.js  47 kB       1  [emitted]  vendor
```

This happens because the webpack bundle, apart from the code of modules, has *a runtime* – a small piece of code that manages the module execution. When you split the code into multiple files, this piece of code starts including a mapping between chunk ids and corresponding files:

```
// vendor.e6ea4504d61a1cc1c60b.js
script.src = __webpack_require__.p + chunkId + "." + {
  "0": "2f2269c7f0a55a5c1871"
}[chunkId] + ".js";
```

Webpack includes this runtime into the last generated chunk, which is `vendor` in our case. And every time any chunk changes, this piece of code changes too, causing the whole `vendor` chunk to change.

To solve this, let's move the runtime into a separate file. **In webpack 4,** this is achieved by enabling the `optimization.runtimeChunk` option:

```
// webpack.config.js (for webpack 4)
module.exports = {
  optimization: {
    runtimeChunk: true,
  },
};
```

In webpack 3, do this by creating an extra empty chunk with the `CommonsChunkPlugin`:

```
// webpack.config.js (for webpack 3)
module.exports = {
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      name: 'vendor',

      minChunks: module => module.context &&
        module.context.includes('node_modules'),
    }),

    // This plugin must come after the vendor one (because webpack
    // includes runtime into the last chunk)
    new webpack.optimize.CommonsChunkPlugin({
      name: 'runtime',

      // minChunks: Infinity means that no app modules
      // will be included into this chunk
      minChunks: Infinity,
    }),
  ],
};
```

After these changes, each build will be generating three files:

```
$ webpack
Hash: ac01483e8fec1fa70676
Version: webpack 3.8.1
Time: 3816ms
                            Asset     Size  Chunks             Chunk Names
   ./main.00bab6fd3100008a42b0.js    82 kB       0  [emitted]  main
 ./vendor.26886caf15818fa82dfa.js    46 kB       1  [emitted]  vendor
./runtime.79f17c27b335abc7aaf4.js  1.45 kB       3  [emitted]  runtime
```

Include them into `index.html` in the reverse order – and you're done:

```
<!-- index.html -->
<script src="./runtime.79f17c27b335abc7aaf4.js"></script>
<script src="./vendor.26886caf15818fa82dfa.js"></script>
<script src="./main.00bab6fd3100008a42b0.js"></script>
```

Further reading

- Webpack guide on long term caching
- Webpack docs about webpack runtime and manifest
- “Getting the most out of the CommonsChunkPlugin”
- How `optimization.splitChunks` and `optimization.runtimeChunk` work

Inline webpack runtime to save an extra HTTP request

To make things even better, try inlining the webpack runtime into the HTML response. I.e., instead of this:

```
<!-- index.html -->
<script src="./runtime.79f17c27b335abc7aaf4.js"></script>
```

do this:

```
<!-- index.html -->
<script>
!function(e){function n(r){if(t[r])return t[r].exports;…}} ([]);
</script>
```

The runtime is small, and inlining it will help you save an HTTP request (pretty important with HTTP/1; less important with HTTP/2 but might still play an effect).

Here’s how to do it.

If you generate HTML with the HtmlWebpackPlugin

If you use the HtmlWebpackPlugin to generate an HTML file, the InlineSourcePlugin is all you need:

```
// webpack.config.js
const HtmlWebpackPlugin = require('html-webpack-plugin');
const InlineSourcePlugin = require('html-webpack-inline-source-plugin');

module.exports = {
  plugins: [
    new HtmlWebpackPlugin({
      // Inline all files which names start with “runtime~” and end with “.js”.
      // That’s the default naming of runtime chunks
      inlineSource: 'runtime~.+\\.js',
    }),
    // This plugin enables the “inlineSource” option
    new InlineSourcePlugin(),
  ],
};
```

If you generate HTML using a custom server logic

With webpack 4:

1. Add the `WebpackManifestPlugin` to know the generated name of the runtume chunk:

```
// webpack.config.js (for webpack 4)
const ManifestPlugin = require('webpack-manifest-plugin');

module.exports = {
  plugins: [
    new ManifestPlugin(),
  ],
};
```

   A build with this plugin would create a file that looks like this:

```
// manifest.json
{
  "runtime~main.js": "runtime~main.8e0d62a03.js"
}
```

2. Inline the content of the runtime chunk in a convenient way. E.g. with Node.js and Express:

```
// server.js
const fs = require('fs');
const manifest = require('./manifest.json');

const runtimeContent = fs.readFileSync(manifest['runtime~main.js'], 'utf-8')

app.get('/', (req, res) => {
  res.send(`
    …
    <script>${runtimeContent}</script>
    …
  `);
});
```

Or with webpack 3:

1. Make the runtime name static by specifying **`filename`**:

   ```
   // webpack.config.js (for webpack 3)
   module.exports = {
     plugins: [
       new webpack.optimize.CommonsChunkPlugin({
         name: 'runtime',
         minChunks: Infinity,
         filename: 'runtime.js',
           // → Now the runtime file will be called
           // "runtime.js", not "runtime.79f17c27b335abc7aaf4.js"
       }),
     ],
   };
   ```

2. Inline the **`runtime.js`** content in a convenient way. E.g. with Node.js and Express:

   ```
   // server.js
   const fs = require('fs');
   const runtimeContent = fs.readFileSync('./runtime.js', 'utf-8');

   app.get('/', (req, res) => {
     res.send(`
       …
       <script>${runtimeContent}</script>
       …
     `);
   });
   ```

Lazy-load code that you don't need right now

Sometimes, a page has more and less important parts:

- If you load a video page on YouTube, you care more about the video than about comments. Here, the video is more important than comments.
- If you open an article on a news site, you care more about the text of the article than about ads. Here, the text is more important than ads.

In such cases, improve the initial loading performance by downloading only the most important stuff first, and lazy-loading the remaining parts later. Use the **`import()`** function and code-splitting for this:

```
// videoPlayer.js
export function renderVideoPlayer() { … }

// comments.js
export function renderComments() { … }

// index.js
import {renderVideoPlayer} from './videoPlayer';
renderVideoPlayer();

// …Custom event listener
onShowCommentsClick(() => {
  import('./comments').then((comments) => {
    comments.renderComments();
  });
});
```

`import()` specifies that you want to load a specific module dynamically. When webpack sees **`import('./module.js')`**, it moves this module into a separate chunk:

```
$ webpack
Hash: 39b2a53cb4e73f0dc5b2
Version: webpack 3.8.1
Time: 4273ms
                            Asset      Size  Chunks             Chunk Names
      ./0.8ecaf182f5c85b7a8199.js  22.5 kB       0  [emitted]
   ./main.f7e53d8e13e9a2745d6d.js    60 kB       1  [emitted]  main
 ./vendor.4f14b6326a80f4752a98.js    46 kB       2  [emitted]  vendor
./runtime.79f17c27b335abc7aaf4.js  1.45 kB       3  [emitted]  runtime
```

and downloads it only when execution reaches the **`import()`** function.

This will make the **`main`** bundle smaller, improving the initial loading time. Even more, it will improve caching – if you change the code in the main chunk, comments chunk won't get affected.

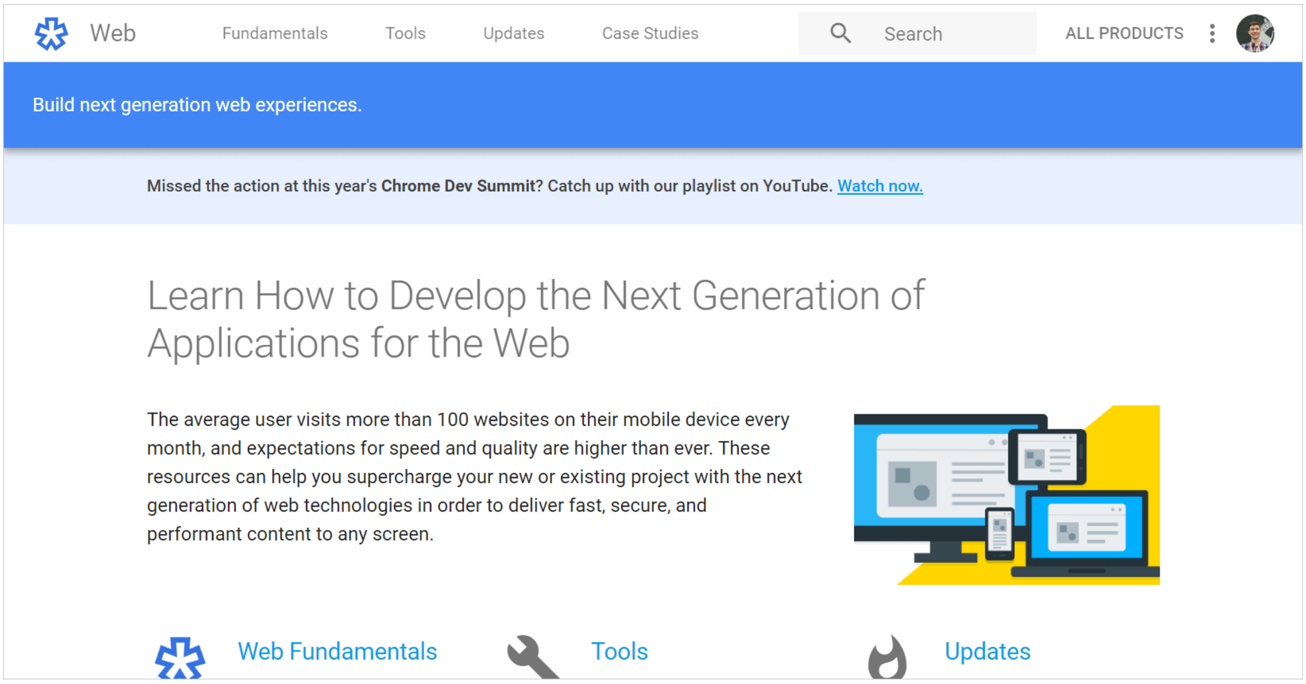
Note: If you compile this code with Babel, you'll have a syntax error because Babel doesn't understand `import()` out of the box. To avoid the error, add the `syntax-dynamic-import` plugin.

Further reading

- Webpack docs for the **`import()`** function
- The JavaScript proposal for implementing the **`import()`** syntax

Split the code into routes and pages

If your app has multiple routes or pages, but there's only a single JS file with the code (a single `main` chunk), it's likely that you're serving extra bytes on each request. For example, when a user visits a home page of your site:



Build next generation web experiences.

Missed the action at this year's **Chrome Dev Summit**? Catch up with our playlist on YouTube. Watch now.

Learn How to Develop the Next Generation of Applications for the Web

The average user visits more than 100 websites on their mobile device every month, and expectations for speed and quality are higher than ever. These resources can help you supercharge your new or existing project with the next generation of web technologies in order to deliver fast, secure, and performant content to any screen.

they don't need to load the code for rendering an article that's on a different page – but they will load it. Moreover, if the user always visits only the home page, and you make a change in the article code, webpack will invalidate the whole bundle – and the user will have to re-download the whole app.

If we split the app into pages (or routes, if it's a single-page app), the user will download only the relevant code. Plus, the browser will cache the app code better: if you change the home page code, webpack will invalidate only the corresponding chunk.

For single-page apps

To split single-page apps by routes, use `import()` (see the "Lazy-load code that you don't need right now" section). If you use a framework, it might have an existing solution for this:

- "Code Splitting" in `react-router`'s docs (for React)
- "Lazy Loading Routes" in `vue-router`'s docs (for Vue.js)

For traditional multi-page apps

To split traditional apps by pages, use webpack's entry points. If your app has three kinds of pages: the home page, the article page and the user account page, – it should have three entries:

```
// webpack.config.js
module.exports = {
  entry: {
    home: './src/Home/index.js',
    article: './src/Article/index.js',
    profile: './src/Profile/index.js'
  },
};
```

For each entry file, webpack will build a separate dependency tree and generate a bundle that includes only modules that are used by that entry:

```
$ webpack
Hash: 318d7b8490a7382bf23b
Version: webpack 3.8.1
Time: 4273ms
                            Asset     Size  Chunks             Chunk Names
      ./0.8ecaf182f5c85b7a8199.js  22.5 kB       0  [emitted]
   ./home.91b9ed27366fe7e33d6a.js    18 kB       1  [emitted]  home
./article.87a128755b16ac3294fd.js    32 kB       2  [emitted]  article
./profile.de945dc02685f6166781.js    24 kB       3  [emitted]  profile
 ./vendor.4f14b6326a80f4752a98.js    46 kB       4  [emitted]  vendor
./runtime.318d7b8490a7382bf23b.js  1.45 kB       5  [emitted]  runtime
```

So, if only the article page uses Lodash, the `home` and the `profile` bundles won't include it – and the user won't have to download this library when visiting the home page.

Separate dependency trees have their drawbacks though. If two entry points use Lodash, and you haven't moved your dependencies into a vendor bundle, both entry points will include a copy of Lodash. To solve this, **in webpack 4,** add the `optimization.splitChunks.chunks: 'all'` option into your webpack config:

```
// webpack.config.js (for webpack 4)
module.exports = {
  optimization: {
    splitChunks: {
      chunks: 'all',
    }
  },
};
```

This option enables smart code splitting. With this option, webpack would automatically look for common code and extract it into separate files.

Or, **in webpack 3,** use the `CommonsChunkPlugin` – it will move common dependencies into a new specified file:

```
// webpack.config.js (for webpack 3)
module.exports = {
  plugins: [
    new webpack.optimize.CommonsChunkPlugin({
      // A name of the chunk that will include the common dependencies
      name: 'common',

      // The plugin will move a module into a common file
      // only if it's included into `minChunks` chunks
      // (Note that the plugin analyzes all chunks, not only entries)
      minChunks: 2,    // 2 is the default value
    }),
  ],
};
```

Feel free to play with the `minChunks` value to find the best one. Generally, you want to keep it small, but increase if the number of chunks grows. For example, for 3 chunks, `minChunks` might be 2, but for 30 chunks, it might be 8 – because if you keep it at 2, too many modules will get into the common file, inflating it too much.

Further reading

- Webpack docs about the concept of entry points
- Webpack docs about the CommonsChunkPlugin
- "Getting the most out of the CommonsChunkPlugin"
- How `optimization.splitChunks` and `optimization.runtimeChunk` work

Make module ids more stable

When building the code, webpack assigns each module an ID. Later, these IDs are used in `require()`s inside the bundle. You usually see IDs in the build output right before the module paths:

```
$ webpack
Hash: df3474e4f76528e3bbc9
```

```
Version: webpack 3.8.1
Time: 2150ms
                           Asset      Size  Chunks             Chunk Names
      ./0.8ecaf182f5c85b7a8199.js  22.5 kB       0  [emitted]
   ./main.4e50a16675574df6a9e9.js    60 kB       1  [emitted]  main
 ./vendor.26886caf15818fa82dfa.js    46 kB       2  [emitted]  vendor
./runtime.79f17c27b335abc7aaf4.js  1.45 kB       3  [emitted]  runtime
```

↓ Here

```
   [0] ./index.js 29 kB {1} [built]
   [2] (webpack)/buildin/global.js 488 bytes {2} [built]
   [3] (webpack)/buildin/module.js 495 bytes {2} [built]
   [4] ./comments.js 58 kB {0} [built]
   [5] ./ads.js 74 kB {1} [built]
    + 1 hidden module
```

By default, IDs are calculated using a counter (i.e. the first module has ID 0, the second one has ID 1, and so on). The problem with this is that when you add a new module, it might appear in the middle of the module list, changing all the next modules' IDs:

```
$ webpack
Hash: df3474e4f76528e3bbc9
Version: webpack 3.8.1
Time: 2150ms
                           Asset      Size  Chunks             Chunk Names
      ./0.5c82c0f337fcb22672b5.js    22 kB       0  [emitted]
   ./main.0c8b617dfc40c2827ae3.js    82 kB       1  [emitted]  main
 ./vendor.26886caf15818fa82dfa.js    46 kB       2  [emitted]  vendor
./runtime.79f17c27b335abc7aaf4.js  1.45 kB       3  [emitted]  runtime
   [0] ./index.js 29 kB {1} [built]
   [2] (webpack)/buildin/global.js 488 bytes {2} [built]
   [3] (webpack)/buildin/module.js 495 bytes {2} [built]
```

↓ We've added a new module...

```
   [4] ./webPlayer.js 24 kB {1} [built]
```

↓ And look what it has done! `comments.js` now has ID 5 instead of 4

```
   [5] ./comments.js 58 kB {0} [built]
```

↓ `ads.js` now has ID 6 instead of 5

```
   [6] ./ads.js 74 kB {1} [built]
       + 1 hidden module
```

This invalidates all chunks that include or depend on modules with changed IDs – even if their actual code hasn't changed. In our case, the `0` chunk (the chunk with `comments.js`) and the `main` chunk (the chunk with the other app code) get invalidated – whereas only the `main` one should've been.

To solve this, change how module IDs are calculated using the `HashedModuleIdsPlugin`. It replaces counter-based IDs with hashes of module paths:

```
$ webpack
Hash: df3474e4f76528e3bbc9
Version: webpack 3.8.1
Time: 2150ms
                           Asset      Size  Chunks             Chunk Names
      ./0.6168aaac8461862eab7a.js  22.5 kB       0  [emitted]
   ./main.a2e49a279552980e3b91.js    60 kB       1  [emitted]  main
 ./vendor.ff9f7ea865884e6a84c8.js    46 kB       2  [emitted]  vendor
./runtime.25f5d0204e4f77fa57a1.js  1.45 kB       3  [emitted]  runtime
```

↓ Here

```
[3IRH] ./index.js 29 kB {1} [built]
[DuR2] (webpack)/buildin/global.js 488 bytes {2} [built]
[JkW7] (webpack)/buildin/module.js 495 bytes {2} [built]
[LbCc] ./webPlayer.js 24 kB {1} [built]
[lebJ] ./comments.js 58 kB {0} [built]
[02Tr] ./ads.js 74 kB {1} [built]
    + 1 hidden module
```

With this approach, the ID of a module only changes if you rename or move that module. New modules won't affect other modules' IDs.

To enable the plugin, add it to the `plugins` section of the config:

```
// webpack.config.js
module.exports = {
  plugins: [
    new webpack.HashedModuleIdsPlugin(),
  ],
};
```

Further reading

- Webpack docs about the HashedModuleIdsPlugin

Summing up

- Cache the bundle and differentiate between versions by changing the bundle name
- Split the bundle into app code, vendor code and runtime
- Inline the runtime to save an HTTP request
- Lazy-load non-critical code with `import`
- Split code by routes/pages to avoid loading unnecessary stuff

← Previous
Decrease Front-end Size

Next
Monitor and analyze the app →

Last updated July 2, 2018.