Supercharged Live Stream Blog: Code Splitting

By Surma

Surma is a contributor to Web**Fundamentals**

Note: As always – this is not production-ready code. I have simplified the code at the cost of generality. Our main goal is to convey concepts and demystify buzzwords.

In our most recent Supercharged Live Stream we implemented code splitting and route-based chunking. With HTTP/2 and native ES6 modules, these techniques will become essential to enabling efficient loading and caching of script resources.

Miscellaneous tips & tricks in this episode

- `asyncFunction().catch()` with `error.stack`: 9:55
- Modules and `nomodule` attribute on `<script>` tags: 7:30
- `promisify()` in Node 8: 17:20

TL;DR:

How to do code splitting via route-based chunking:

1. Obtain a list of your entry points.
2. Extract the module dependencies of all these entry points.
3. Find shared dependencies between all entry points.
4. Bundle the shared dependencies.
5. Rewrite the entry points.

Code splitting vs. route-based chunking

Time stamp: 1:50

Code splitting and route-based chunking are closely related and are often used interchangably. This has caused some confusion. Let's try to clear this up:

- **Code splitting**: Code splitting is the process of splitting your code into multiple bundles. If you are *not* shipping one big bundle with all of your JavaScript to the client, you are doing code splitting. One specific way of splitting your code is to use route-based chunking.
- **Route-based chunking**: Route-based chunking creates bundles that are related your app's routes. By analyzing your routes and their dependencies, we can change what modules go into which bundle.

Why do code splitting?

Loose modules

With native ES6 modules, every JavaScript module can import its own dependencies. When the browser receives a module, all `import` statements will trigger additional fetches to get ahold of the modules that are necessary to run the code. However, all these modules can have dependencies of their own. The danger is that the browser ends up with a cascade of fetches that last for multiple round trips before the code can finally be executed.

Bundling

Bundling, which is inlining all your modules into one single bundle will make sure the browser has all the code it needs after 1 round trip and can start running the code more

quickly. This, however, forces the user to download a lot of code that is not needed, so bandwidth and time have been wasted. Additionally, every change to one of our original modules will result in a change in the bundle, invalidating any cached version of the bundle. Users will have to re-download the entire thing.

Code splitting

Code splitting is the middle ground. We are willing to invest additional round trips to get network efficiency by only downloading what we need, and better caching efficiency by making the number of modules per bundle much smaller. If the bundling is done right, the total number of round trips will be much lower than with loose modules. Finally, we could make use of pushing mechanisms like `link[rel=preload]` or HTTP/2 Push to save additional round trio times if needed.

Step 1: Obtain a list of your entry points

> Time stamp: 16:02 16:02

This is only one of many approaches, but in the episode we parsed the website's `sitemap.xml` to get the entry points to our website. Usually, a dedicated JSON file listing all entry points is used.

Using babel to process JavaScript

> Time stamp: 25:25 to 28:25

Babel is commonly used for "transpiling": consuming bleeding-edge JavaScript code and turning it into an older version of JavaScript so that more browsers are able to execute the code. The first step here is to parse the new JavaScript with a parser (Babel uses babylon) that turns the code into a so-called "Abstrac Syntax Tree" (AST). Once the AST has been generated, a series of plugins analyze and mangle the AST.

Note: I am not very experienced with Babel. The plugins I built work, but they are probably neither efficient nor idiomatically babel. For a more in-depth guide to authoring Babel plugins, I recommend the Babel Handbook.

We are going to make heavy use of babel to detect (and later manipulated) the imports of a JavaScript module. You might be tempted to resort to regular expressions, but regular

expressions are not powerful enough to properly parse a language and are hard to maintain. Relying on tried-and-tested tools like Babel will save you many headaches.

Here's a simple example of running Babel with a custom plugin:

```
const plugin = {
  visitor: {
    ImportDeclaration(decl) {
      /* ... */
    }
  }
}
const {code} = babel.transform(inputCode, {plugins: [plugin]});
```

A plugin can provide a **`visitor`** object. The visitor contains a function for any node type that the plugin wants to handle. When a node of that type is encountered while traversing the AST the corresponding function in the **`visitor`** object will be invoked with that node as a parameter. In the example above, the **`ImportDeclaration()`** method will be called for every **`import`** declaration in the file. To get more of a feeling for node types and the AST, take a look at astexplorer.net.

Step 2: Extract the module dependencies

Time stamp: 28:25 to 34:57

To build the dependency tree of a module, we will parse that module and create a list of all the modules it imports. We also need to parse those dependencies, as they in turn might have dependencies as well. A classic case for recursion!

```
async function buildDependencyTree(file) {
  let code = await readFile(file);
  code = code.toString('utf-8');

  // `dep` will collect all dependencies of `file`
  let dep = [];
  const plugin = {
    visitor: {
      ImportDeclaration(decl) {
        const importedFile = decl.node.source.value;
        // Recursion: Push an array of the dependency's dependencies onto the lis
        dep.push((async function() {
          return await buildDependencyTree(`./app/${importedFile}`);
        })());
        // Push the dependency itself onto the list
```

```
        dep.push(importedFile);
      }
    }
  }
  // Run the plugin
  babel.transform(code, {plugins: [plugin]});
  // Wait for all promises to resolve and then flatten the array
  return flatten(await Promise.all(dep));
}
```

Step 3: Find shared dependencies between all entry points

Time stamp: 34:57 to 38:30

Since we have a set of dependency trees – a dependency forest if you will – we can find the shared dependencies by looking for nodes that appear in *every* tree. We will flatten and deduplicate our forest and filter to only keep the elements that appear in all trees.

```
function findCommonDeps(depTrees) {
  const depSet = new Set();
  // Flatten
  depTrees.forEach(depTree => {
    depTree.forEach(dep => depSet.add(dep));
  });
  // Filter
  return Array.from(depSet)
    .filter(dep => depTrees.every(depTree => depTree.includes(dep)));
}
```

Step 4: Bundle shared dependencies

Time stamp: 39:20 to 46:43

To bundle our set of shared dependencies, we could just concatenate all the module files. Two problems arise when using that approach: The first problem is that the bundle will still contain `import` statements which will make the browser attempt to fetch resources. The second problem is that the dependencies' dependencies have not been bundled. Because we have done it before, we are going to write *another* babel plugin.

Note: I am convinced someone with more experience with Babel and its APIs would be able to do all of this in one pass. For the sake of brevity and clarity, I chose to write multiple plugins and parse some files

multiple times so that the steps are truly separate.

The code is fairly similar to our first plugin, but instead of just extracting the imports, we will also be removing them and inserting a bundled version of the imported file:

```
async function bundle(oldCode) {
  // `newCode` will be filled with code fragments that eventually form the bundle
  let newCode = [];
  const plugin = {
    visitor: {
      ImportDeclaration(decl) {
        const importedFile = decl.node.source.value;
        newCode.push((async function() {
          // Bundle the imported file and add it to the output.
          return await bundle(await readFile(`./app/${importedFile}`));
        })());
        // Remove the import declaration from the AST.
        decl.remove();
      }
    }
  };
  // Save the stringified, transformed AST. This code is the same as `oldCode`
  // but without any import statements.
  const {code} = babel.transform(oldCode, {plugins: [plugin]});
  newCode.push(code);
  // `newCode` contains all the bundled dependencies as well as the
  // import-less version of the code itself. Concatenate to generate the code
  // for the bundle.
  return flatten(await Promise.all(newCode)).join('\n');
}
```

Step 5: Rewrite entry points

Time stamp: 46:43

For the last step we will write yet another Babel plugin. Its job is to remove all imports of modules that are in the shared bundle.

```
async function rewrite(section, sharedBundle) {
  let oldCode = await readFile(`./app/static/${section}.js`);
  oldCode = oldCode.toString('utf-8');
  const plugin = {
    visitor: {
      ImportDeclaration(decl) {
```

```
        const importedFile = decl.node.source.value;
        // If this import statement imports a file that is in the shared bundle,
        if(sharedBundle.includes(importedFile))
          decl.remove();
      }
    }
  };
  let {code} = babel.transform(oldCode, {plugins: [plugin]});
  // Prepend an import statement for the shared bundle.
  code = `import '/static/_shared.js';\n${code}`;
  await writeFile(`./app/static/_${section}.js`, code);
}
```

End

This was quite the ride, wasn't it? Please remember that our goal for this episode was to *explain and demystify* code splitting. The result works – but it's specific to our demo site and will fail horribly in the generic case. For production, I'd recommend relying on established tools like WebPack, RollUp, etc..

You can find our code in the GitHub repository.

See you next time!

Last updated July 2, 2018.