## Should you learn webpack?

Today **CLI tools** as create-react-app or Vue cli abstract away most of the configuration, and **provide sane defaults**.

Even then, understanding how things work under the hood is beneficial because sooner or later you'll need to make some adjustment to the defaults.

In this guide we'll see what webpack can do, and how to configure it to suit your needs.

## Terminology. What is webpack?

As a JavaScript developer you should be familiar with the term **module**.

**webpack is a module bundler** and **has a broader definition of what a module is**, specifically, for webpack, modules are:

* Common JS modules
* AMD modules
* CSS import
* Images url
* ES modules

That is, webpack is able to ingest **dependencies** from any of these sources.

The ultimate goal of webpack is to **unify all these different sources and module types** in a way that's possible to **import everything in your JavaScript code**, and finally produce a shippable output.

### Entry point

An entry point for webpack is **the starting point** from which all the dependencies of a frontend project are collected. In practice, it's a simple JavaScript file.

These dependencies form a **dependency graph**.

The default entry point for webpack (since version 4) is src/index.js, and it's configurable. webpack can have multiple entry points.

### Output

The output is where the resulting JavaScript and static files are collected during the build process.

The default output folder for webpack (since version 4) is dist/, configurable as well.

The resulting JavaScript files are part of the so called **bundle**.

### Loaders

Loaders are **third-party extensions that help webpack deal with various file extensions**. For example there are loaders for CSS, for images, or for txt files.

The goal of a loader is to transform files (other than JavaScript) in modules. Once the file becomes a module, webpack can use it as a dependency in your project.

### Plugins

Plugins are **third-party extensions that can alter how webpack works**. For example there are plugins for extracting HTML, CSS, or for setting up environment variables.

### Mode

webpack has two modes of operations: **development** and **production**. The main difference between them is that **production** mode automatically applies **minification** and other optimizations to your JavaScript code.

### Code splitting

**Code splitting, or lazy loading** is an optimization technique for avoiding larger bundles.

With code splitting, developers can decide to load whole blocks of JavaScript only in response to some user interaction, like clicks or route changes (or other conditions).

A piece of code that's splitted becomes a **chunk**.

## Getting started with webpack

To start off with webpack create a new folder and move into it to initialize an NPM project:

mkdir webpack-tutorial && cd $\_

npm init -y

Once inside install **webpack, webpack-cli, and the webpack-dev-server**:

npm i webpack webpack-cli webpack-dev-server --save-dev

To run webpack easily from an NPM script, open up package.json and configure a "dev" script:

"scripts": {

"dev": "webpack --mode development"

},

With this script we instruct webpack to work in **development mode**, convenient for working locally.

## First steps with webpack

To run webpack in development mode:

npm run dev

You should see the following error.

ERROR in Entry module not found: Error: Can't resolve './src'

Here webpack is looking for the default entry point, src/index.js. Create the folder, and inside the same folder create a simple JavaScript file:

mkdir src

echo 'console.log("Hello webpack!")' > src/index.js

Now run again npm run dev and you should see no more errors. The result of this run is a **new folder named** dist/, with a JavaScript file in it named main.js:

dist

└── main.js

This is your first **webpack bundle**, also called output.

## Configuring webpack

For simpler tasks webpack could work without a configuration, but you'll hit the limit pretty soon. To configure webpack through a file create a webpack.config.js in the project folder:

touch webpack.config.js

Webpack is written in JavaScript, and runs on top on a headless JavaScript environment such as **Node.js**. In this file you'll need at least a module.exports, which is the Common JS export for Node.js:

module.exports = {

//

};

In webpack.config.js we can change how webpack behaves by adding or altering:

* entry point
* output
* loaders
* plugins
* code splitting

For example, to **change the entry point** path we can do:

const path = require("path");

module.exports = {

entry: { index: path.resolve(\_\_dirname, "source", "index.js") }

};

Now webpack will look in source/index.js for the first file to load. To change instead the **output** of our bundle we can do:

const path = require("path");

module.exports = {

output: {

path: path.resolve(\_\_dirname, "build")

}

};

With this configuration webpack will put the bundle in build instead of dist. (To keep things simple we'll stick to the default in this guide).

## Working with HTML

A web application without an HTML page is almost useless. **To work with HTML in webpack** we need to install a plugin, **html-webpack-plugin**:

npm i html-webpack-plugin --save-dev

Once the plugin is installed we can configure it:

const HtmlWebpackPlugin = require("html-webpack-plugin");

const path = require("path");

module.exports = {

plugins: [

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname, "src", "index.html")

})

]

};

Here we say to webpack, load an HTML template from src/index.html.

The ultimate goal of **html-webpack-plugin** is twofold:

* it loads our HTML files
* it **injects the bundle(s) in the same file**

Before moving on create a simple HTML file in src/index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Webpack tutorial</title>

</head>

<body>

</body>

</html>

In a second we'll run this "app" with webpack's development server.

## webpack's development server

In the first section of this guide we installed **webpack-dev-server**. If you forgot to add it, install the package now:

npm i webpack-dev-server --save-dev

**webpack-dev-server** is a convenient package for development. Once configured, we can launch a local server to serve our files.

To configure **webpack-dev-server**, open up package.json and add a "start" script:

"scripts": {

"dev": "webpack --mode development",

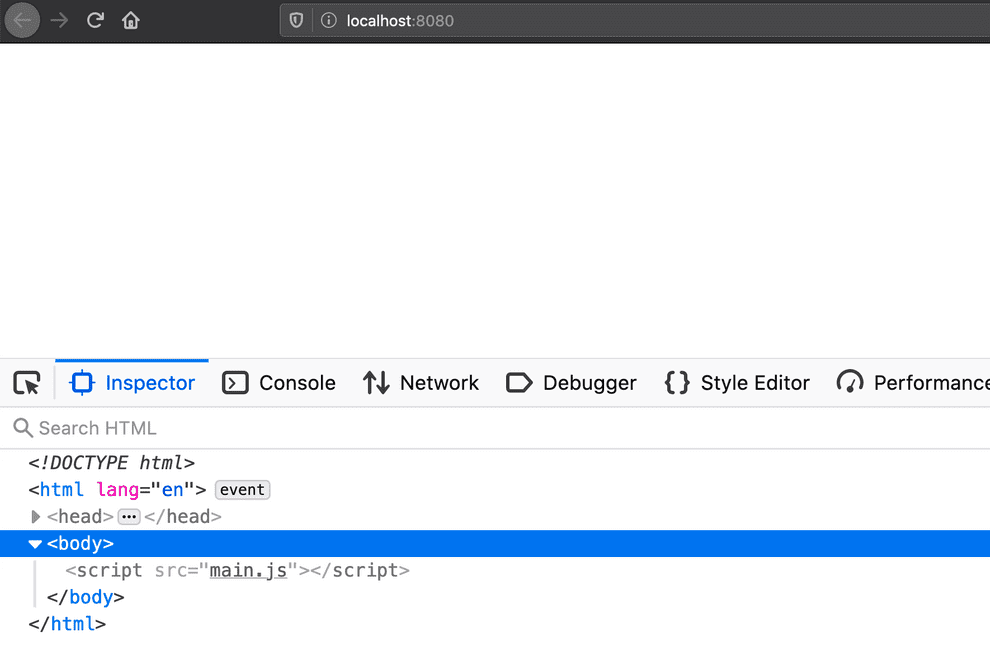
"start": "webpack serve --mode development --open 'Chrome'"

},

With this script we can run the server easily. Now run:

npm run start

Your default browser should open, or you have to go to http://localhost:8080/. In the browser's console you should also see a script tag, with our main JavaScript bundle injected:

[](https://www.valentinog.com/blog/static/06dee181263809dde67ecf0e4c4d8d01/61016/webpack-dev-server.png)

## Working with webpack's loaders

Loaders are **third-party extensions that help webpack deal with various file extensions**. For example there are loaders for CSS, for images, or for txt files.

The anatomy of a webpack loader, configuration wise, is the following:

module.exports = {

module: {

rules: [

{

test: /\.filename$/,

use: ["loader-b", "loader-a"]

}

]

},

//

};

The relevant configuration starts with the module key. Inside this key we configure each loaders group, or a single loader, inside rules.

For **each file we want to treat as a module** we configure an object with a test key, and with use:

{

test: /\.filename$/,

use: ["loader-b", "loader-a"]

}

test tells webpack "hey, treat this filename as a module". use instead, defines what loaders are applied to the file.

## Working with CSS

**To work with CSS in webpack** we need to install at least two **loaders**.

Loaders here are necessary for helping webpack to understand how to deal with .css files.

To test CSS in webpack create a simple stylesheet in src/style.css:

h1 {

color: orange;

}

Also, add an HTML element to our HTML template in src/index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Webpack tutorial</title>

</head>

<body>

<h1>Hello webpack!</h1>

</body>

</html>

Finally, load the CSS in src/index.js:

import "./style.css";

console.log("Hello webpack!");

Before testing the page we need to install the loaders:

* **css-loader** for loading CSS files with import
* **style-loader** for loading the stylesheet in the DOM

Install the loaders:

npm i css-loader style-loader --save-dev

Then configure them in webpack.config.js:

const HtmlWebpackPlugin = require("html-webpack-plugin");

const path = require("path");

module.exports = {

module: {

rules: [

{

test: /\.css$/,

use: ["style-loader", "css-loader"]

}

]

},

plugins: [

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname, "src", "index.html")

})

]

};

The relevant configuration starts with the module key:

const HtmlWebpackPlugin = require("html-webpack-plugin");

const path = require("path");

module.exports = {

module: {

rules: [

{

test: /\.css$/,

use: ["style-loader", "css-loader"]

}

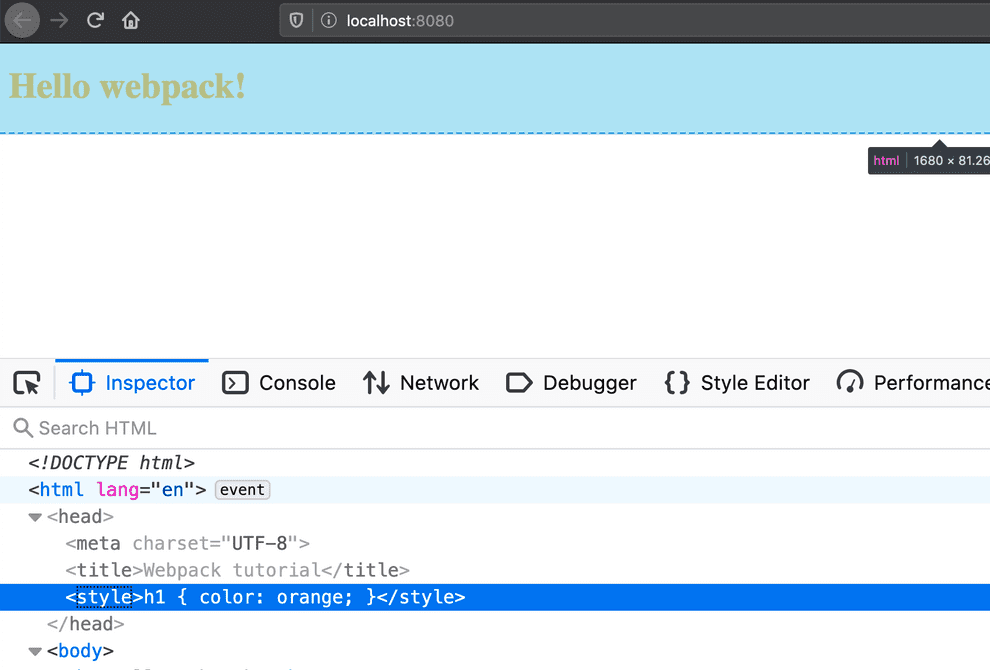
]

},

//

};

Now if you run npm run start you should see the stylesheet loaded in the HTML's head:

[](https://www.valentinog.com/blog/static/107120eb641df48da9625bf147954004/61016/webpack-css-loader.png)

Once CSS loaders are in place you can extract CSS files with [MiniCssExtractPlugin](https://webpack.js.org/plugins/mini-css-extract-plugin/" \t "_blank).

## Order of webpack loaders matters!

In webpack, **the order in which loaders appear in the configuration is of high importance**. The following configuration is invalid:

//

module.exports = {

module: {

rules: [

{

test: /\.css$/,

use: ["css-loader", "style-loader"]

}

]

},

//

};

Here "style-loader" appears before "css-loader". But style-loader is for injecting the style in the page, not for loading the actual CSS file.

The following configuration instead is valid:

//

module.exports = {

module: {

rules: [

{

test: /\.css$/,

use: ["style-loader", "css-loader"]

}

]

},

//

};

**webpack loaders are loaded from right to left, (or think as of top to bottom).**

## Working with SASS

**To work with SASS in webpack** we need to install at least the appropriate loaders.

Loaders here are necessary for helping webpack to understand how to deal with .scss files.

To test SASS in webpack create a simple stylesheet in src/style.scss:

@import url("https://fonts.googleapis.com/css?family=Karla:weight@400;700&display=swap");

$font: "Karla", sans-serif;

$primary-color: #3e6f9e;

body {

font-family: $font;

color: $primary-color;

}

Also, add some more HTML elements to our HTML template in src/index.html:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Webpack tutorial</title>

</head>

<body>

<h1>Hello webpack!</h1>

<p>Hello sass!</p>

</body>

</html>

Finally, load the SASS file in src/index.js:

import "./style.scss";

console.log("Hello webpack!");

Before testing the page we need to install the loaders (and the sass package for Node.js):

* **sass-loader** for loading SASS files with import
* **css-loader** for loading CSS files as modules
* **style-loader** for loading the stylesheet in the DOM

Install the loaders:

npm i css-loader style-loader sass-loader sass --save-dev

Then configure them in webpack.config.js:

const HtmlWebpackPlugin = require("html-webpack-plugin");

const path = require("path");

module.exports = {

module: {

rules: [

{

test: /\.scss$/,

use: ["style-loader", "css-loader", "sass-loader"]

}

]

},

plugins: [

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname, "src", "index.html")

})

]

};

Again, the relevant configuration starts with the module key:

const HtmlWebpackPlugin = require("html-webpack-plugin");

const path = require("path");

module.exports = {

module: {

rules: [

{

test: /\.scss$/,

use: ["style-loader", "css-loader", "sass-loader"]

}

]

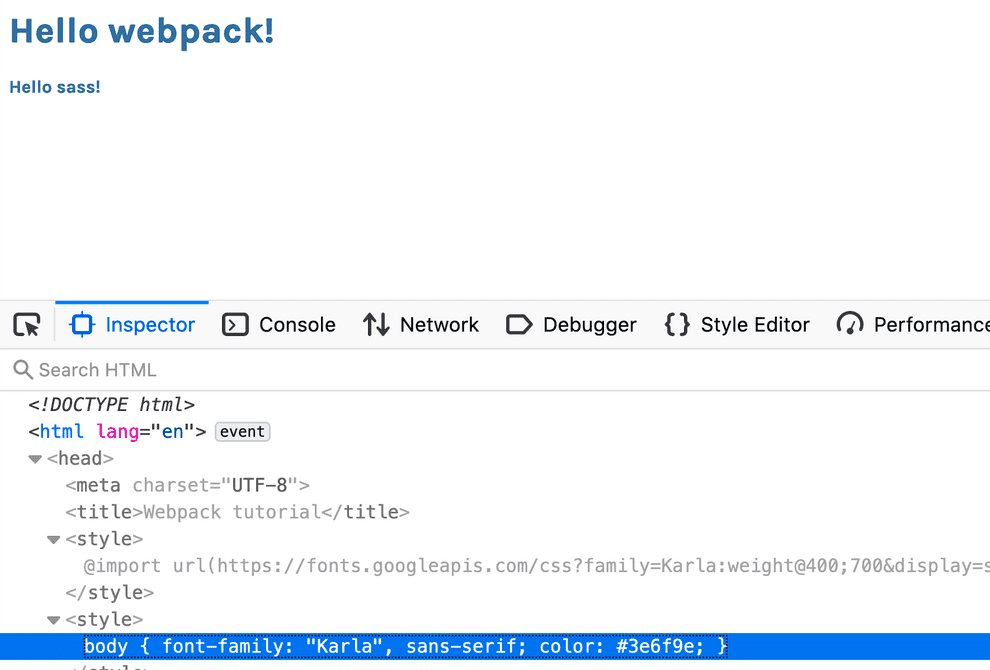
},

//

};

Notice **the order in which loaders appear**: first, sass-loader, then css-loader, finally style-loader.

Now if you run npm start you should see the stylesheet loaded in the HTML's head:

[](https://www.valentinog.com/blog/static/28f360788e76a4b957456842642c32c8/61016/webpack-sass.png)

Once SASS and CSS loaders are in place you can extract CSS files with [MiniCssExtractPlugin](https://webpack.js.org/plugins/mini-css-extract-plugin/" \t "_blank).

## Working with modern JavaScript

**webpack doesn't know on its own how to transform JavaScript code.** This task is outsourced to a third-party loader, specifically **babel-loader**, with **babel**.

babel is a JavaScript compiler and "transpiler". **Given modern JavaScript syntax as input, babel is able to transform it to compatible code** that can run in (almost) any browser.

Before moving forward we need to install a bunch of packages:

* **babel core**, the actual engine
* **babel preset env** for compiling modern Javascript down to ES5
* **babel loader** for webpack

Let's pull in the dependencies:

npm i @babel/core babel-loader @babel/preset-env --save-dev

Then configure babel by creating a new file, babel.config.json. Here we configure babel to use [preset-env](https://www.valentinog.com/blog/preset-env/" \t "_blank):

{

"presets": [

"@babel/preset-env"

]

}

Finally, configure webpack to use the loader for transforming JavaScript files (I left the SASS too loader for a bit more context):

const HtmlWebpackPlugin = require("html-webpack-plugin");

const path = require("path");

module.exports = {

module: {

rules: [

{

test: /\.scss$/,

use: ["style-loader", "css-loader", "sass-loader"]

},

{

test: /\.js$/,

exclude: /node\_modules/,

use: ["babel-loader"]

}

]

},

plugins: [

new HtmlWebpackPlugin({

template: path.resolve(\_\_dirname, "src", "index.html")

})

]

};

To test the transformation, write some modern syntax in src/index.js:

import "./style.scss";

console.log("Hello webpack!");

const fancyFunc = () => {

return [1, 2];

};

const [a, b] = fancyFunc();

Now run npm run dev to see the transformed code in dist. Open up dist/main.js and search for "fancyFunc":

\n\nvar fancyFunc = function fancyFunc() {\n return [1, 2];\n};\n\nvar \_fancyFunc = fancyFunc(),\n \_fancyFunc2 = \_slicedToArray(\_fancyFunc, 2),\n a = \_fancyFunc2[0],\n b = \_fancyFunc2[1];\n\n//# sourceURL=webpack:///./src/index.js?"

Without babel, the code wouldn't be transpiled:

\n\nconsole.log(\"Hello webpack!\");\n\nconst fancyFunc = () => {\n return [1, 2];\n};\n\nconst [a, b] = fancyFunc();\n\n\n//# sourceURL=webpack:///./src/index.js?");

**Note**: webpack works totally fine even without babel. The transpiling process is only necessary for shipping ES5.

## How to set up React, webpack 5, and Babel from scratch

To use React components with webpack, alongside with babel loader you should also install the babel preset for React:

npm i @babel/core babel-loader @babel/preset-env @babel/preset-react --save-dev

Once done, configure babel to use the React preset in babel.config.json:

{

"presets": ["@babel/preset-env", "@babel/preset-react"]

}

At this point you can install React with:

npm i react react-dom

Finally, you can write your components in the project. To test things out you can create a component in src/index.js:

import React, { useState } from "react";

import { render } from "react-dom";

function App() {

const [state, setState] = useState("CLICK ME");

return <button onClick={() => setState("CLICKED")}>{state}</button>;

}

render(<App />, document.getElementById("root"));

Now when running webpack's dev server with npm start you should see the component in the browser. (Don't forget to add a <div> in the page with the relevant id).

## Working with JavaScript's modules in webpack

webpack treats a whole range of files as modules. But, let's not forget its main purpose: **loading ES modules**.

Up until 2015 JavaScript didn't have a standard mechanism for code reuse. There had been a lot of attempts to standardize this aspect, which resulted in a messy fragmentation during the years.

You might have heard about AMD modules, UMD, or Common JS. There was no clear winner. Finally, with ECMAScript 2015, ES modules landed in the language. We now have an "official" module system.

**webpack makes a joy working with ES modules and modularized code.**

To try out ES modules in webpack let's create a module in a new file at src/common/usersAPI.js with the following code:

const ENDPOINT = "https://jsonplaceholder.typicode.com/users/";

export function getUsers() {

return fetch(ENDPOINT)

.then(response => {

if (!response.ok) throw Error(response.statusText);

return response.json();

})

.then(json => json);

}

Now in src/index.js you can load the module and use the function:

import { getUsers } from "./common/usersAPI";

import "./style.scss";

console.log("Hello webpack!");

getUsers().then(json => console.log(json));

For a refresher on ES modules: [All I need to know about ECMAScript modules](https://www.valentinog.com/blog/es-modules/).

## Production mode

As introduced earlier, webpack has two modes of operation: **development** and **production**. So far we worked only in development mode.

In **development** mode, webpack takes all the JavaScript code we write, almost pristine, and loads it in the browser.

No **minification** is applied. This makes reloading the application in development faster.

In **production** mode instead, webpack applies a number of optimizations:

* minification with TerserWebpackPlugin to reduce the bundle size
* scope hoisting with ModuleConcatenationPlugin

It also set process.env.NODE\_ENV to "production". This environment variable is useful for doing things conditionally in production or in development.

To configure webpack in production mode, open up package.json and add a "build" script:

"scripts": {

"dev": "webpack --mode development",

"start": "webpack serve --open 'Firefox'",

"build": "webpack --mode production"

},

Now when running npm run build webpack will produce a minified bundle.

## Code splitting with webpack

**Code splitting refers to an optimization technique** aiming at:

* avoid big bundles
* avoid dependencies duplication

There is a limit that the webpack community considers the maximum size for the initial bundle of an application: **200KB**. To understand why keeping bundles small is paramount, search for "The Cost of JavaScript" on Google.

There are three main ways to activate code splitting in webpack:

* with multiple entry points
* with optimization.splitChunks
* with dynamic imports

The first technique based on multiple entry points works well for smaller projects, but it's not scalable in the long run. Here we'll focus only on optimization.splitChunks and dynamic imports.

### Code splitting with optimization.splitChunks

Consider a JavaScript application using **Moment.js**, the popular JS library for times and dates. There are better alternatives to it, but for a moment (no pun intended) let's prove my point.

Install the library in your project folder:

npm i moment

Now wipe out the content of src/index.js and import the library there:

import moment from "moment";

Run a build with npm run build and look at the output:

main.js 350 KiB 0 [emitted] [big] main

The **whole library is bundled in the main entry point of our app**. Not good. With optimization.splitChunks we can move out moment.js from the main bundle.

To configure code splitting open up webpack.config.js and add the optimization key to your configuration, configured as follows:

const HtmlWebpackPlugin = require("html-webpack-plugin");

const path = require("path");

module.exports = {

module: {

// omitted for brevity

},

optimization: {

splitChunks: { chunks: "all" }

},

// omitted for brevity

};

Run a build with npm run build and look at the output:

main.js 5.05 KiB 0 [emitted] main

vendors~main.js 346 KiB 1 [emitted] [big] vendors~main

We now have a vendors~main.js with moment.js, while the main entry point has a more reasonable size.

**Note**: even with code splitting moment.js remains a gigantic library. There are better alternatives like luxon or date-fns.

### Code splitting with dynamic imports

A more powerful technique for code splitting uses **dynamic imports** to load code conditionally. webpack offered dynamic imports long before this feature shipped in ECMAScript 2020.

This approach is widely used in modern frontend library like Vue and React (React has its own way, but the concept is the same).

Code splitting might be used:

* at the module level
* at the route level

For example, you can load conditionally some JavaScript module in response to a user interaction, like a click, or a mouse move. Or, you can load relevant portions of your code on response to **route changes**.

To get started with dynamic imports, wipe out the content of src/index.html, and place in the following HTML instead:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<title>Dynamic imports</title>

</head>

<body>

<button id="btn">Load!</button>

</body>

</html>

Make sure to still have the fetch module in src/common/usersAPI.js:

const ENDPOINT = "https://jsonplaceholder.typicode.com/users/";

export function getUsers() {

return fetch(ENDPOINT)

.then(response => {

if (!response.ok) throw Error(response.statusText);

return response.json();

})

.then(json => json);

}

Now in src/index.js create the following logic:

const btn = document.getElementById("btn");

btn.addEventListener("click", () => {

//

});

Nothing happens if you run npm run start to see and click the button in the interface.

Now imagine we want to load a list of users after someone clicks the button. A "naive" approach can use a static import to load the function from src/common/usersAPI.js:

import { getUsers } from "./common/usersAPI";

const btn = document.getElementById("btn");

btn.addEventListener("click", () => {

getUsers().then(json => console.log(json));

});

The problem is that **ES modules are static, meaning we cannot change imports at runtime.**

With a **dynamic import** instead we can choose **when to load our code**:

const getUserModule = () => import("./common/usersAPI");

const btn = document.getElementById("btn");

btn.addEventListener("click", () => {

getUserModule().then(({ getUsers }) => {

getUsers().then(json => console.log(json));

});

});

Here we create a function to load the module dynamically:

const getUserModule = () => import("./common/usersAPI");

Then in the event listener we chain then() to the dynamic import:

btn.addEventListener("click", () => {

getUserModule().then(/\*\*/);

});

This gives the ability to extract our getUsers function with object destructuring:

btn.addEventListener("click", () => {

getUserModule().then(({ getUsers }) => {

//

});

});

Finally, we use our function as usual:

//

btn.addEventListener("click", () => {

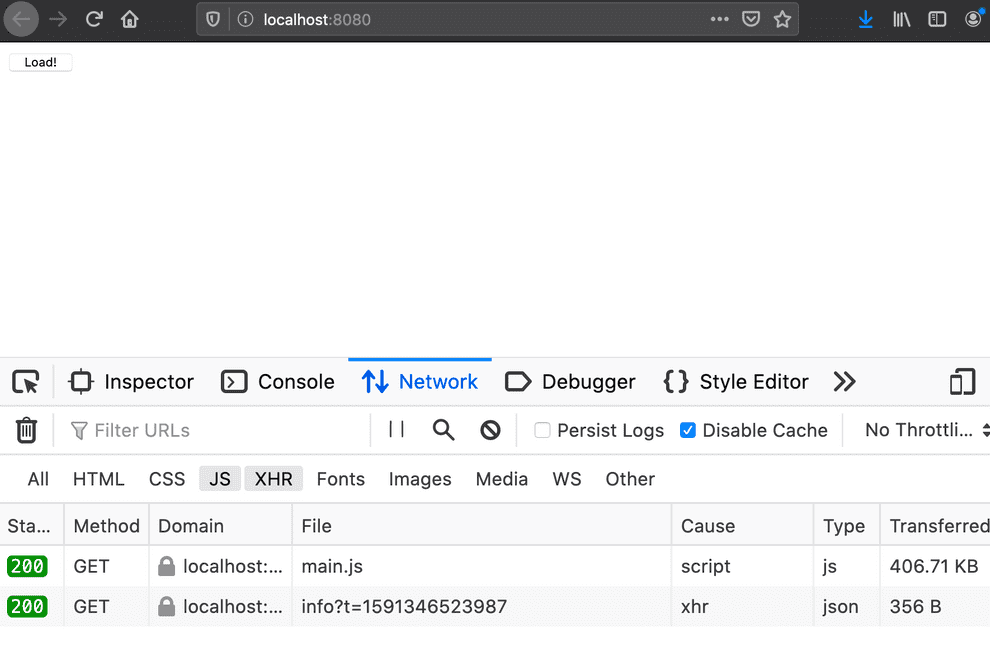
getUserModule().then(({ getUsers }) => {

getUsers().then(json => console.log(json));

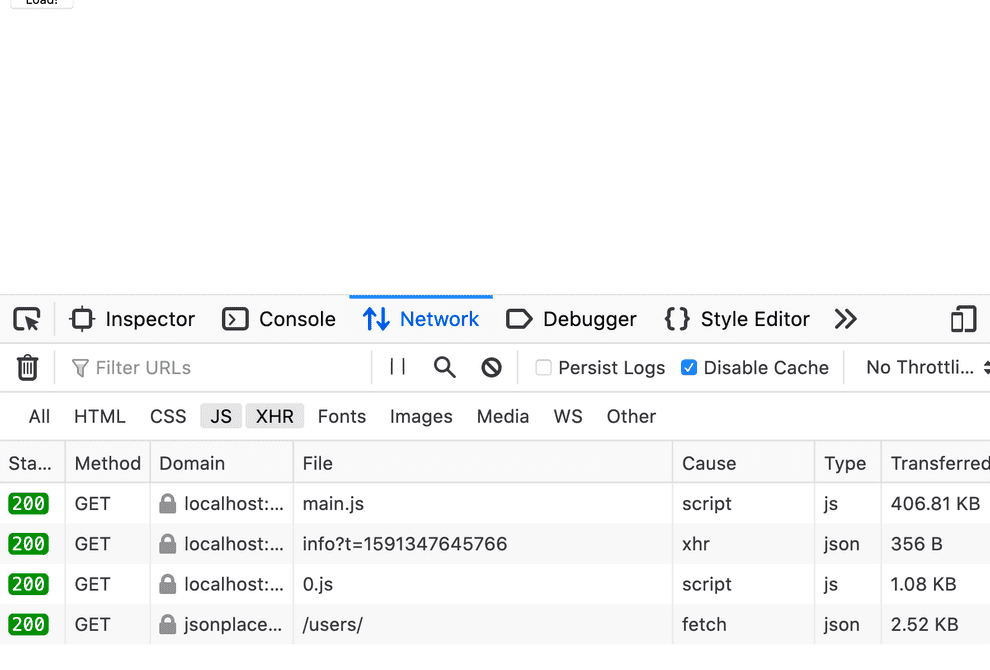
});

});

When you now load the page for the first time with npm run start you see the main bundle loaded in the console:

[](https://www.valentinog.com/blog/static/dfbcf49bdf273505b85d3f9264406d59/61016/main-bundle-loaded.png)

Now **"./common/usersAPI" loads only when clicking the button**:

[](https://www.valentinog.com/blog/static/dfbe402ce1ad36d9e1fad066b7f0060b/61016/dynamic-import-webpack.png)

The lazy "chunk" is 0.js.

By prefixing the import path with /\* webpackChunkName: "name\_here" \*/ we can also control the chunk name:

const getUserModule = () =>

import(/\* webpackChunkName: "usersAPI" \*/ "./common/usersAPI");

const btn = document.getElementById("btn");

btn.addEventListener("click", () => {

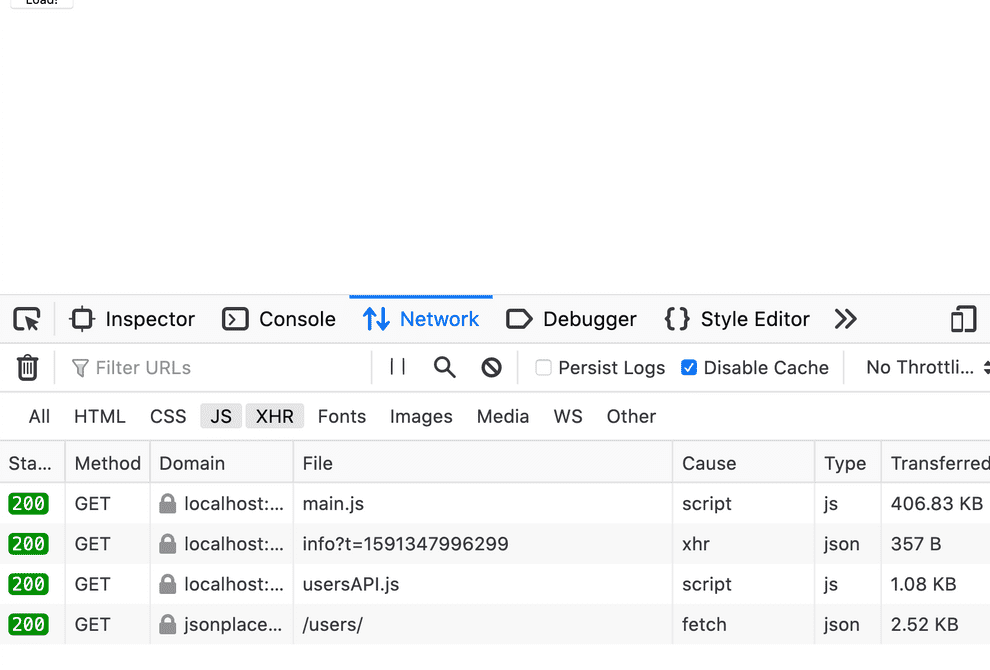
getUserModule().then(({ getUsers }) => {

getUsers().then(json => console.log(json));

});

});

The chunk now will have the desired name:

[](https://www.valentinog.com/blog/static/a8ad13d04b27f3986b40fba271dda85e/61016/webpackChunkName.png)