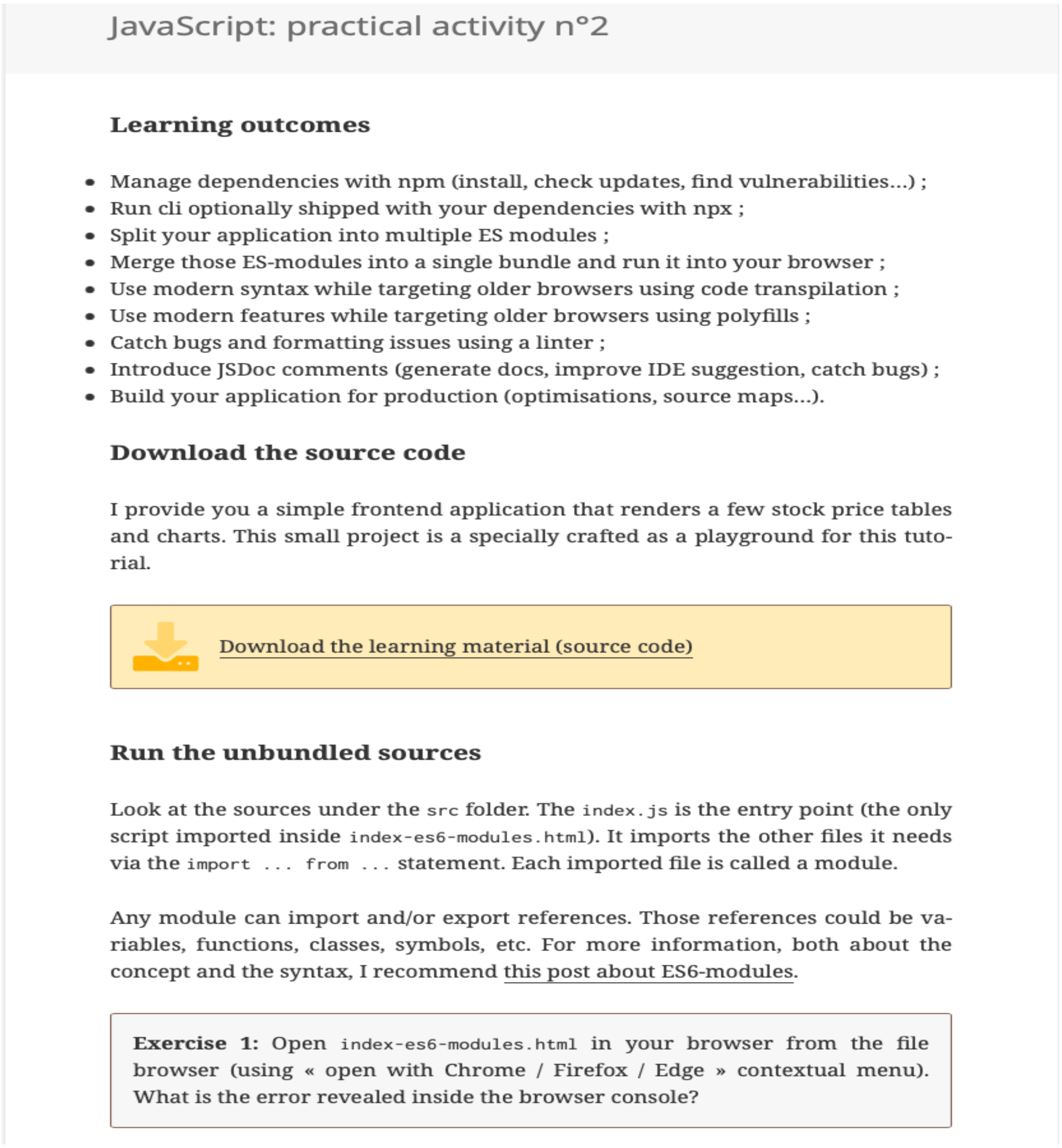
Web Development

Lab2 : Advanced JS

- Observer l'architecture de l'application (Front vs Back).
- Prenez le temps de faire le lien entre les différents fichiers Js et Html.



JavaScript: practical activity n°2

Learning outcomes

- Manage dependencies with npm (install, check updates, find vulnerabilities...) ;
- Run cli optionally shipped with your dependencies with npx ;
- Split your application into multiple ES modules ;
- Merge those ES-modules into a single bundle and run it into your browser ;
- Use modern syntax while targeting older browsers using code transpilation ;
- Use modern features while targeting older browsers using polyfills ;
- Catch bugs and formatting issues using a linter ;
- Introduce JSDoc comments (generate docs, improve IDE suggestion, catch bugs) ;
- Build your application for production (optimisations, source maps...).

Download the source code

I provide you a simple frontend application that renders a few stock price tables and charts. This small project is a specially crafted as a playground for this tutorial.

Download the learning material (source code)

Run the unbundled sources

Look at the sources under the `src` folder. The `index.js` is the entry point (the only script imported inside `index-es6-modules.html`). It imports the other files it needs via the `import ... from ...` statement. Each imported file is called a module.

Any module can import and/or export references. Those references could be variables, functions, classes, symbols, etc. For more information, both about the concept and the syntax, I recommend this post about ES6-modules.

Exercise 1: Open `index-es6-modules.html` in your browser from the file browser (using « open with Chrome / Firefox / Edge » contextual menu). What is the error revealed inside the browser console?

1) Compléter le fichier « serveur.js » dans le dossier « Back ».
2) Lancer le serveur avec « node serveur.js ».
3) Ensuite pour lancer l'application en « Front », la cmd est « npm run serve-es6 ».

Exercise 2: Launch a local HTTP server (seen in the lecture) and run the application from it. Then, inspect the HTTP traffic and record some statistics:

- How many JS files were loaded ?
- How long does it take on average to load one ES6-module ?
- What is the total completion time to load all JS files ?
- Did the browser download JS files sequentially or in parallel ?

Tips: your development tools record traffic network only when opened. You may refresh your tab with development tools opened to capture the whole network activity.

Exercise 3: Discuss the performance penalty of serving ES6-modules as independent files to the browser. How to mitigate this performance issue?

Getting familiar with npm features

npm stands for « Node Package Manager ». This utility is shipped with the node.js installer and is a foundation of modern JavaScript ecosystem.

The primary goal of a package manager is making the installation of dependencies easy. Those dependencies may be libraries, frameworks, command line software, etc.

- A library can be imported like any ES6-module in your code.
- A command line software is run from the terminal as a standalone software.
- A framework is generally a set of multiple libraries, cli scripts and guidelines, that comes with strong opinions on how to use it.

***

First of all, npm is tightly coupled to the `package.json` file, which is both a manifest of the project and a configuration file for npm. Both your project using npm, and any dependency you install this way have it. A sample is included into the historical rates explorer app.

Any `package.json` contains basic information about the project, such as its `name`, `version` (which should follow semantic versioning standard), `author` and `license`. Such a basic `package.json` can be created with the command `npm init`.

Then, you may specify dependencies (each one is identified with a name and an assertion about accepted versions). Just with that information, npm can pull dependencies from the public registry.

The `package.json` of the sample app contains 2 groups of dependencies:

- those under the `dependencies` key, that are used by your code at runtime ;
- those under the `devDependencies` key, that are part of the developer experience (for building your app, check code quality, execute automated testing, etc.).

Depending on the npm's command you use, you can install each or all groups.

```
# Installs all dependencies from package.json (dependencies and devDependenci
npm install

# Installs only production dependencies from package.json (ignores devDepende
npm install --prod

# Installs a new package (called webpack-cli) and appends it to package.json
npm install webpack-cli --save-dev
```

Exercise 4: Install dependencies using the `npm install` command. Dependencies are locally stored inside the `node_modules` folder. Why is there far more many modules into `node_modules` that those declared in `package.json` ?

***

Because dependencies and versions are listed in a single place and installable through a single command, you can ensure every replica of the installation use consistent dependencies.

Another great benefit is automated tasks and analysis you can run on it:

- track updates and install them easily (using `npm outdated`) ;
- be alerted in case of security advice about one of your dependencies (find the command yourself).

Exercise 5: The rate history app is potentially vulnerable to a high-level security issue through one of its dependencies. Using npm, find all vulnerable packages and associated version. In which release those bug have been fixed?

Tips: Before npm exists, developers frequently used CDN (Content Delivery Network) to include libraries. A CDN hosts a JS script you may include just before your own script (ex. by appending `<script src="https://CDN_HOST/jquery.min.js" />` to the HTML source). As a result, the library exposes its functions by populating the global scope. Using CDN generally leads to outdated and inconsistent dependency versions across the project, and finally to security issues and technical dept. The bundler (kind of tool we discuss later) cannot optimise your app. For those and various other reasons, **prefer npm over CDN**.

***

Did you notice the `scripts` key in my `package`.json? It points to an object whose keys are user-defined names and values are the bash commands to execute. The value can make reference to cli software installed inside `node_modules`.

For example, running `npm run serve-es6-sources` will start a local http server (using the `http-server` package).

Very common script's names are:

- `build`: to produce a distributable bundle, sometime with many transformations ;
- `watch`: mostly the same than `build`, but efficiently re-emits bundle on change ;
- `test`: to execute a suite of automated tests you wrote yourself.

The `build` script can target both the production (by generating more optimised bundles) and development (by preferring fastly emitted bundles and easier debugging in the browser). The `watch` script is intended for development, of course.

Bundle the sources using webpack

Exercise 6: Look at the webpack documentation (both home page and the concepts). What is its primary purpose and how does it fix our performance issue previously encountered with ES6-modules?

Exercise 7: The `webpack.config.js` file is ready to build the aggregated bundle in development mode. The entry point is `src/index.js`. Bundle should be output at `dist/index.js`. You can run webpack using `npm run build`.

Tips: look at the emitted `dist/index.js` and be sure to understand what webpack did. Remember to question the teacher when necessary.

Exercise 8: Open `index-bundled.html` in your browser (could be from the disc or from HTTP server). Does it actually fix the ES-module fetching performance issue ? Please provide some evidence in your answer.

Exercise 9: Compare webpack's output in development and in production mode.

The browser compatibility issue

The 2 major kinds of compatibility issues are **unsupported syntax** and **missing features**.

- Fixing an **unsupported syntax** error imply transforming your code not to use that syntax anymore. For example, you could write classes without the `class` keyword.

```
// ES6 code (using the class keyword)

class User {
  constructor (id, firstName, lastName) {
    this.id = id
    this.firstName = firstName
    this.lastName = lastName
  }

  getName () {
    return this.firstName + ' ' + this.lastName.toUpperCase()
  }
}
```

```
// Isofunctional ES5 code (defining the same class without using the `class`

var User = (function () {
    function User(id, firstName, lastName) {
        this.id = id
        this.firstName = firstName
        this.lastName = lastName
    }

    User.prototype.getName = function () {
        return this.firstName + ' ' + this.lastName.toUpperCase()
    }

    return User
}())
```

But instead of writing old, verbose, ugly and potentially buggy code, it is common to **transpile** from recent syntax to older one. That the purpose of babeljs, which can work as a plugin for webpack. This will be one of your next job 🤗.

- Fixing a **missing feature** issue is more complicated depending on your use case. You cannot implement yourself the support of NFC hardware if the browser does not have this feature. But you can implement yourself some utility methods introduced recently.

  Let's suppose your code uses the `array.includes(item)` methods, which returns `true` if `item` is in `array`. This method has been introduced in ES7 (released in 2016).

  That method can easily be implemented with a one-liner snippet. Then, you can add it to the Array's prototype and make it callable on any array instance.

  ```
  // ⛔ you CANNOT use [1, 2, 3].includes(2) on old browser

  Array.prototype.includes = function (search) {
    return this.indexOf(search) !== -1
  }

  // 🚀 you CAN use [1, 2, 3].includes(2) (if array.indexOf() is supported)
  ```

  Tips: Extending prototypes is generally a bad practice as it leads to conflicts with someone else's code using the same name. Nevertheless, it is allowed to backport missing standard methods (as long as it strictly behaves the same way). Those backported methods are called **polyfills**. Also, the community shares npm package with ready to import polyfills, like the popular core-js.

***

For the next questions, you need an obsolete browser. More exactly, we will use Internet Explorer 11. If not available on your system, you can:

1. run a free IE 11 instance on the cloud with browserstack.com
2. forward your local HTTP server with ngrok.com.

Tips: If you need to set up browserling + ngrok combo before the teacher demonstrated their usage, notify him!

***

Exercice 10: Try the bundled app with Internet Explorer 11. Why IE does it fail to run the bundled app?

Exercice 11: The following webpack's configuration sends every `.js` to `babel-loader`. Babel transforms any syntax not supported by IE 11. Is it enough to ensure compatibility with IE? Why?

```
// updated webpack.config.js that integrates babel to assets pipeline.
const path = require('path')

module.exports = {
  entry: './src/index.js',
  mode: 'development',
  target: ['web', 'es5'],
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'index.js'
  },
  module: {
    rules: [
      {
        test: /\.m?js$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            targets: 'ie >= 11',
            presets: ['@babel/preset-env']
          }
        }
      }
    ]
  }
}
```

You also need in import the babel utilties (transformed code uses them):

```
// add this at the beginning of index.js
import 'regenerator-runtime/runtime'
```

Exercice 12: Import the required polyfills from core-js and whatwg-fetch in `index.js`. Now, the app should work on IE 11.

Exercise 13: List at least 2 issues of transpiling the code and adding polyfills like we did. Try to mention a mitigation or a solution for each issue.

Hey! Still alive?

Congratulations for going so far! Despite there is still a lot uncovered learning outcomes, you have completed the mandatory part. The rest of the topics will be explained by the teacher at the end of the activity.

In case you are ahead of time, feel free to reach yourself the above goals:

1. Catch some code smells and improved code formatting with standard.js.
2. Generate documentation using JSDoc.
3. Test the quality of autocomplete with vscode, intellisense and JSDoc.
4. Catch errors using types described in JSDoc comments with the typescript compiler in JS-compatible mode.
5. Improve debugging experience for transformed code with source-maps.

Breaking change! Vue moved from Vue-CLI to `vue-create`.

My initial intent while writing this tutorial was to explain the tools behind `Vue-CLI`, the official command line tool to create and manage vue applications. As of September 2022, the `vue-cli` is now deprecated in favour of `vue-create`. It does not use webpack under the hook anymore. Still, the background exposed in the current tutorial still helps to understand the benefits of Vite and Rollup internally used by the new tool chain.