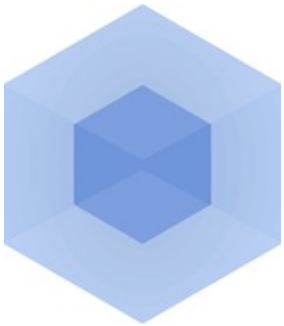
UNIT 1

JavaScript



Part 4
Node Package Manager. Webpack.

Client-side Web Development
2nd course – DAW
IES San Vicente 2020/2021
Author: Arturo Bernal Mayordomo

Index

Introduction

We could say that NPM (Node Package Manager) is a way to reuse code from other developers in our own projects. This code is distributed in what we call **packages** or modules. Among these packages we find popular libraries, frameworks and tools like JQuery, Angular, Express, JSHint, ESLint, Sass, Browserify, Bootstrap, Cordova, Ionic, and many others.

NPM also provides a powerful script based task automation. For most projects, NPM makes other package and task managers like Gulp, Grunt or Bower not necessary, although some developers combine them in their projects, but it has the disadvantage that you need to learn how 2, 3 or even 4 different tools work. We will try to do everything using only NPM.

NPM Installation

In order to install NPM, we also need to install Node.js (NPM is part of it), a JavaScript runtime based on Chrome's JavaScript engine that allows us to build **server side** applications using only this language, making our web applications 100% JavaScript code.

You can read the Windows and Mac installation instructions from here. If you're using Linux, many distributions include Node in their repositories. If you want to use an external repository that has the latest version, you can follow instructions here. The recommended version to install is the latest 10.x, because it will be the Long Term Support version (very stable) from October 2018 until April 2021.

Once Node is installed, you can check Node and NPM versions from the command line with these two commands:

```
arturo@arturo-desktop:~$ node -v
v10.11.0
arturo@arturo-desktop:~$ npm -v
6.4.1
```

You can update NPM to the latest version with **npm install npm -g** (run as administrator, in Linux use **sudo** or root).

NPM Packages

A package (library, framework, tool, ...) is a directory containing many files and a **package.json** file that holds information about the author, version, other packages on which this package depends, etc. In this section, we're going to see how to install, update, or remove packages in our project (or globally in the system).

We can search for help with the **npm** command like this:

- **npm -h** → Quick help and a list of the typical npm commands.
- **npm command -h** → Quick help with specific information about a npm command (npm install -h).
- **npm help command** → It will open a browser page or a man page (Linux) with detailed help about a command.
- **npm help-search words** → Will return a list of help topics that contain the searched words (separated by space).

Creating our package.json

There are two types of projects we can start. A web application for users to interact with, or a library/tool package to be included in other developer's projects. Both of them will usually have dependencies on other packages and we also need to perform task on them like testing or minifying.

To create a **package.json** file in your project, simply move to the project's main directory and type **npm init**. This will ask you some questions about your project and create a **package.json** file based on them. We can leave most of them with the default values or empty by pressing enter. Before creating the file, it will show you what is going to write into it.

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "A project created to learn NPM",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Arturo",
  "license": "ISC"
}
```

Installing packages

The command to install a package in our project is **npm install package-name**. If we execute this command for the first time, we'll see that a directory named **node_modules** will be created inside your project's main directory. This directory will contain all the packages installed by npm (packages that we installed and also their

dependencies). Let's try with Jquery:

```
arturo@arturo-desktop:~/Documentos/example$ npm install jquery
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN example@1.0.0 No repository field.

+ jquery@3.3.1
added 1 package from 1 contributor and audited 1 package in 1.369s
found 0 vulnerabilities
```

As you can see it will install JQuery (latest available version) inside the node_modules directory (the files to use in our project will be inside jquery/dist/ directory).

With previous NPM versions (< 5.0), we needed to include the option **--save** or **-S** to store the dependency in package.json. However, that's done automatically now.

```
{
    "name": "project",
    "version": "1.0.0",
    "description": "A project created to learn NPM",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Arturo",
    "license": "ISC",
    "dependencies": {
        "jquery": "^3.3.1"
    }
}
```

This will ensure that NPM installs and updates to the latest "3.x.x" version of the JQuery library in our project. By default, this will be considered a production package (our application will need it to run). There are some other packages that are not needed to run our application but for other purposes like testing (ex: karma), or javascript minifying (ex: uglify). They're called development dependencies, and usually will not be included when our application running is in production mode.

To include a package in our **development** dependencies list, you should use the option --**save-dev**. Example:

npm i uglify --save-dev

```
{
    "name": "project",
    "version": "1.0.0",
    "description": "A project created to learn NPM",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Arturo",
    "license": "ISC",
    "dependencies": {
        "jquery": "^3.1.0"
    },
    "devDependencies": {
        "uglify": "^0.1.5"
    }
```

```
}
```

Installing project dependencies

If you download an existing NPM project (or clone a git repository), it usually won't have the node_modules directory with the packages in it. To install both production and development dependencies listed in package.json, just type **npm install** (or **npm i**).

You can install only production or development dependencies with **npm install --only=prod**, or **npm install --only=dev**.

Executing local commands instead of global

If your project has a different version of a global package installed locally, like handlebars, Angular, Ionic, etc. You can execute the local version instead of the global one using the npx command.

npx handlebars ... → Execute the local project version of Handlebars

Installing global packages

There are JavaScript tools like handlebars, gulp, grunt, etc. that can be installed globally in the system. This way, we can have access to the command that the installed tool uses anywhere in any project.

In order to install a dependency globally, just include the option **-g** at the end of the install command. You'll also need to run the command as an administrator user (or with sudo in Linux). Example:

sudo npm i handlebars -g → Installs handlebars command globally

```
arturo@arturo-desktop:~/Documentos/example$ handlebars -v
4.0.12
```

Listing installed packages

In order to list the installed packages in the npm_modules folder type **npm list** command. This will show in a tree view, the packages you installed along their dependencies in a tree view. To list only the packages that we have installed (dependencies of those packages not include), type **npm list --depth=0**.

```
arturo@arturo-desktop:~/Documentos/example$ npm list --depth 0
example@1.0.0 /home/arturo/Documentos/example
├── jquery@3.3.1
└── uglify@0.1.5
```

If we want to list global installed packages instead, we need to add the option **--global=true** to the command.

```
arturo@arturo-desktop:~/Documentos/example$ npm list --depth 0 --global true
/usr/lib
├── handlebars@4.0.12
└── npm@6.4.1
```

Other useful options include **--dev=true** (list only development packages), **--prod=true** (only production packages), or **--long=true** (include a description of the packages).

```
arturo@arturo-desktop:~/Documentos/example$ npm list --depth 0 --long true
example@1.0.0
├ /home/arturo/Documentos/example
├ Example
├── jquery@3.3.1
│   JavaScript library for DOM operations
│   git+https://github.com/jquery/jquery.git
│   https://jquery.com
└── uglify@0.1.5
    A simple tool to uglify javascript & css files
    git://github.com/nanjingboy/uglify.git
    https://github.com/nanjingboy/uglify#readme
```

Installing specific package versions

By default, when we install a package, it will install the latest stable version. Sometimes, our project needs to include a previous version of a package (If we want to support IE8, we would need JQuery 1.x version, for example).

Package versions usually have three numbers **X.Y.Z**. Maybe not every package (their developers) don't follow exactly the rules that apply to those numbers, but they're usually followed:

- When number **Z** is incremented, it means that a bug or performance fix have been released, but there won't be any new functionality.
- When number **Y** is incremented, it means that new functionality have been added, but it won't break any code from previous versions (anything else have been removed or changed). For example, an application made with Angular 1.2 should continue to work with Angular 1.5.
- Number **X** should only be incremented if there have been enough changes that it doesn't guarantee that older code will work with the new version. You should be very careful and usually make a migration process of your code when you change to a library or framework new version that has incremented the first number. Ex: JQuery 2.x doesn't support IE 8 and older, while 1.x does.

If we want to install a specific version of a library instead of the latest, when you execute the command to install, just type package@"x.y.z" instead of just the name.

npm i jquery@"1.12.3" → Will install this JQuery version that's still compatible with IE8.

We can specify that we want to install a version newer than other (x.y.z) version by typing **package@">x.y.z"**, or the opposite (the latest version previous to this one) **pakage@"<x.y.z"**. We even can install the latest version between two versions: **package@">1.5.0 <2.0.0"**, and even make it more complicated with || (or) operator.

Other possibilities (examples):

- **"*" or "x"** → Install the latest version of the package (be careful with major version changes…).

- **"3" or "3.x", or "3.x.x"** → This will install the latest version of a package whose major version is 3 (it will never update to version 4.x.x).

- **"^3.3.5"** → This will install the latest major version 3 (3.x.x) of a package, but at least has to be 3.3.5 (character **^** means stick to this major version). This is NPM default behaviour.

- **"3.1" or "3.1.x"** → This will install the latest 3.1 minor version of a package (will never upgrade to 3.2.x or higher).

- **"~3.3.5"** → This will install the latest 3.3 minor version (won't upgrade to 3.4 or higher), but at least has to be version 3.3.5 (character ~ means stick to latest minor version).

Updating packages

To update all the dependencies in a project just execute **npm update** (it will look at your package.json file to see which versions are allowed). To update only one package type **npm update package**. You can use **--prod** or **--dev** options to update only production or development dependecies. Or **-g** to update a global package.

Removing packages

To remove a package from your project, just type **npm uninstall package**. Instead of **uninstall** you can also use **remove**, **rm**, **un**, **r** or **unlink** to do the same. To uninstall a global package, just put the **-g** option.

npm r jquery → Removes JQuery (and its dependencies) from the project and also from the package.json file.

npm prune --production → Removes development dependencies leaving only **production** dependencies packages.

Automating tasks with NPM scripts

In our **package.json** file, apart from managing package dependencies, we can create some useful scripts for our project (run a web server, testing, minifying, etc.). Those scripts are commands and they all have a name that identifies them. You have to put those scripts inside the "script" section (which is an array). A script syntax can be described as: "**script-name**": "**Command to run**". To execute any script just type **npm run script-name**.

```
"scripts": {
    "hello": "echo 'hello'"
}
```

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm run hello

> project@1.0.0 hello /home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project
> echo 'hello'

hello
```

Start scripts

There are some common scripts like **start**, **stop** or **test** which are so common that can be executed without typing **run**. The start script is usually created to launch a web server so we can test our application. Sometimes it's also used to launch a TypeScript or SASS compiler, for example.

In our example, the start script will just launch a chrome browser window with an URL where our web server is running, so we can test our application:

```
"scripts": {
    "start": "google-chrome http://localhost/project"
}
```

We can try this script executing **npm start**.

Test scripts

In a serious application, you'll want to do some real testing of your code before publishing anything. You can create a script (or more) that will run your tests inside the **package.json** file.

We are going to make an example testing a JavaScript file with **ESLint**. This tool does code testing (code errors, good practices,…) but no Unit testing or other types or testing, for example (better use other tools like mocha, jasmine, karma,…).

First we'll install ESLint globally executing **npm i eslint -g**. Then execute the command: **eslint --init** inside your project directory. It will ask you some questions if you don't choose to create a default configuration file:

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ eslint --init
? How would you like to configure ESLint? Answer questions about your style
? Are you using ECMAScript 6 features? No
? Where will your code run? Browser
? Do you use CommonJS? No
? Do you use JSX? No
? What style of indentation do you use? Tabs
? What quotes do you use for strings? Double
? What line endings do you use? Unix
? Do you require semicolons? Yes
? What format do you want your config file to be in? JSON
Successfully created .eslintrc.json file in /home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project
```

This is the configuration file created (**.eslintrc.json** in my case, in Linux). You can change some rules that give an error by default to just give a warning (change "error" to "warn"):

```
{
    "env": {
        "browser": true
    },
    "extends": "eslint:recommended",
    "rules": {
        "indent": ["error",  "tab"],
        "linebreak-style": ["error",  "unix" ],
        "quotes": ["warn", "double"],
        "semi": [error", "always"]
    }
}
```

Now, we can run tests against any JavaScript file:

```
'strict mode';

function printNum() {
        num = 34; // Indented with tab
    return num * 2; // Spaces instead of tab
}

alert('Hello World!');
printNum() // I forgot the semicolon ';'
```

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ eslint main.js

/home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project/main.js
  1:1   warning  Strings must use doublequote                        quotes
  4:2   error    'num' is not defined                                no-undef
  5:5   error    Expected indentation of 1 tab character but found 0 indent
  5:12  error    'num' is not defined                                no-undef
  8:7   warning  Strings must use doublequote                        quotes
  9:11  error    Missing semicolon                                   semi

✖ 6 problems (4 errors, 2 warnings)
```

Now, we can just put that command in our **package.json**, so when we execute **npm test** (or npm tst), it will run eslint against that file:

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "eslint main.js"
}
```

Pre and post-script hooks

Some tasks may take a few steps, like for example minifying our JavaScript and CSS files before running the application (start script). To execute more than one task or command we can link them with && (and). Those commands will be executed in order, until one of them fails.

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "echo 'We are going to make some tests' && eslint main.js && echo
'Test successful!'",
}
```

However, it becomes cleaner if we use NPM's predetermined hooks (prefixes) **pre** and **post**. Just create a script with one of these prefixes and it will run before (pre) or after (post) the script that's being executed. In this example we'll create a **pretest** and **posttest** script.

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "eslint main.js",
    "pretest": "echo 'We are going to make some tests'",
    "posttest": "echo 'Test successful!'"
}
```

No, if we execute **npm test**, it will run pretest, test and posttest in this order. If one script fails, the next one won't be executed.

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm test -s
We are going to make some tests

/home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project/main.js
  1:1  warning  Strings must use doublequote  quotes

✖ 1 problem (0 errors, 1 warning)

Test successful!
```

Example: Minifying with uglify

Uglify is a tool that compresses JavaScript files, making them lighter and hard (obfuscated) to read by others. This makes our web pages load faster. In our project, uglify is included as a development dependency (make sure it's installed).

Now just add this line to the scripts section (run first **npm i uglify-js -D**):

```
"build": "uglifyjs  -mc -o bundle.js js/*.js"
```

We're going to compress **main.js** into a new file called **bundle.js** (we have to **include this one** in our HTML). If we want to compress more than one file into a single file, just add them to the command separated by space (**order does matter!**). If we execute **npm run build**, it will generate bundle.js with this JavaScript code inside:

```
"strict mode";function printNum(){var r=34;return 2*r}alert("Hello
World!"),printNum();
```

Example: Executing a script when some files change

There's a tool that can be used to execute a script when some files inside a

directory change. This tool is called **watch**, and we can add it to our development dependencies executing **npm i watch --save-dev**.

To use this tool, create a script that executes: **watch 'command' directory**. When a change in a JavaScript file inside the directory is detected, a command will be run automatically.

```
    "build": "uglifyjs  -mc -o bundle.js js/*.js",
    "build:watch": "watch 'npm run build' ./js" →If a change is detected in ./js
```

Now, if we execute **npm run build:watch**, this tool will be continuously running and when a file inside the **js** directory changes, it will run the **build** script. To stop this tool, just press **Ctrl+c** on the console.

Example: Launching a simple web server and watch JS changes to run uglify concurrently

If we run a web server using NPM (we'll use lite-server), we won't be able to do another task until it's closed (finished). There's a package in NPM that allow us to run more than one task concurrently (its name is also concurrently).

First we'll install those two packages for developing purposes (it won't be shipped in our final application (**npm i lite-server -D**) (**npm i concurrently -D**). Then we create a task that executes two task using "*concurrently*" (it's parameters should be strings and node tasks).

```
    "start": "concurrently 'npm run build:watch' 'npm run serve'",
    "serve": "lite-server",
    "build": "uglifyjs  -mc -o bundle.js js/*.js",
    "build:watch": "watch 'npm run build' ./js"

  "devDependencies": {
    "concurrently": "^2.2.0",
    "lite-server": "^2.2.2",
    "uglify-js": "^2.7.10",
    "watch": "^0.19.2"
  }
```

Some NPM scripts are so common that we don't need to use **npm run start** for example, just **npm start**. Also**, lite-server** searches for an **index.html** file to load.

```
$: npm start

> project@1.0.0 start /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
> concurrently 'npm run build:watch' 'npm run serve'

[0]
[0] > project@1.0.0 build:watch /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
[0] > watch 'npm run build' ./js
[0]
[1]
[1] > project@1.0.0 serve /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
[1] > lite-server
```

Webpack

Webpack is a module bundler, it allows to generate just one file (bundle), or more, from the many JavaScript, CSS or HTML files that compose our project. It's also capable to understand and automatically compile ES2015, TypeScript, Sass, etc. using plungins. There are other tools similar like browserify, uglify, requirejs, … but Webpack is by far the most powerful and configurable of them, and also becoming more and more integrated in many projects (including Angular, for example).

Another advantage of Webpack is that it bundles only those classes or functions that are going to be needed in a web page (tree shaking) reducing size and optimizing speed of our application.

Webpack installation

Is very simple to install Webpack in our project, just install **webpack** package as a development dependency:

```
npm i D webpack webpack-cli
```

In order to develop our application with interactive bundling and a local web server, we should install **webpack-dev-server**. This way, apart from having a web server running locally to test our app, if a file changes, Webpack will automatically generate a new bundle and refresh our app in the browser.

```
npm i -D webpack-dev-server
```

Now, lets create some NPM scripts to run webpack (using the **-p** option will generate a **production** build with minified JavaScript using UglifyJS):

```
"scripts": {
   "build": "webpack",
   "build:watch": "webpack --watch",
   "server": "webpack-dev-server"
}
```

webpack --watch runs the webpack bundler in watch mode (when a file changes, it runs webpack automatically). But we'll usually want to execute webpack-dev-server.

However, there's an important step left, creating a **Webpack configuration file**. This file will be in the project's main directory and it will normally be called **webpack.config.js**. We can choose other name, but we'd have to run webpack with the option --config "config_filename".

Configuring Webpack's dev server

To configure webpack-dev-server we include an entry named "devServer" in our Webpack configuration file (**webpack.config.js**) . This is a configuration example:

```
module.exports = {
    devServer: {
```

```
        contentBase: __dirname, // Default (project's root directory)
        publicPath: '/dist/', // Path where bundles are
        compress: true, // Enable gzip compresion when serving content
        port: 8080 // Default
    }
}
```

Entry points

Imagine that we have an **index.html** file and the JavaScript file associated with its functionality is called **index.js**. The way to tell Webpack to generate a bundle from that file (single entry point) would be:

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

module.exports = {
    context: path.join(__dirname, './src'),
    entry: './index.js',
    ...
}
```

Using **context**, are telling Webpack that all our source files will be located in the **src**/ directory. What happens when we have multiple pages in our application, and each page has an associated JavaScript file?. We can set multiple entry points in our application. Each one will generate a different bundle:

```
module.exports = {
    …
    entry: {
        page1: './page1.js',
        page2: './page2.js',
        page3: './page3.js'
    }
};
```

Output

Setting only entry points won't make Webpack work. You also need to set the output file(s) that will be generated. If we have only one entry point, it's easy, just specify a name for the bundle file and an **absolute** path.

```
module.exports = {
    …
    entry: './index.js',
    output: {
        filename: 'bundle.js',
        path: path.join(__dirname + '/dist')
    } // Generates dist/bundle.js (include this in you HTML file)
};
```

Note that you **must** include the generated **output** file in your HTML and not the original one, or it wouldn't make sense to use Webpack.

Even if we have multiple multiple entry points, we can only specify one output configuration. This is the way we can generate a different bundle for each entry point using only one output:

```
module.exports = {
```

```
    …
    entry: {
        page1: './page1.js',
        page2: './page2.js',
        page3: './page3.js'
    },
    output: {
        filename: '[name].bundle.js',
        path: path.join(__dirname + '/dist')
    } // dist/page1.bundle.js dist/page2.bundle.js and dist/page3.bundle.js
};
```

ES2015 module export/import

Webpack supports loading modules using different methods. The two most used are CommonJS **require** (same syntax as in NodeJS), and ES2015 import.We'll use the EcmaScript **import** syntax.

Adding manually all the JavaScript files in a HTML page has several disadvantages. It causes confusion when we load many files (libraries, own files,…), it makes our HTML dirty, and many files add global variables that can be used in other files, but also overridden by another variable with the same name.

With module loading, we decide what a file (module) should make available to the rest of modules. One way to do that is exporting just one function, class, variable, etc. using the **export default** syntax (we don't need to give it a name). Then, from other files, we import giving it the name we want.

File: src/person.class.js

```
export default class {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

File: src/main.js

```
import Person from './person.class'; // Import it with a name

let p = new Person("Peter", 42);
console.log(p.name); // Prints "Peter"
```

If we want to export anything with the same name (and also being able to export more than one thing), we don't use the default keyword. But this time, we must give a name to the item we're exporting. Note that the import has to be done inside curly brackets {}, and we must use the same name.

File: src/person.class.js

```
export class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

File: src/main.js

```
import {Person} from './person.class'; // Import it with a name

let p = new Person("Peter", 42);
console.log(p.name); // Prints "Peter"
```

Another way to export is to declare at the end of the file, what are we exporting. We can import whatever we want from other files. We don't have to import everything that's being exported in a file, only what we need.

File: src/person.class.js

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}

const ROLES = ["admin", "guest", "user"];
const GUEST_NAME = "Anonymous";

export {Person, ROLES, GUEST_NAME};
```

File: src/main.js

```
import {Person, GUEST_NAME} from './person.class'; // Import it with a name

let p = new Person(GUEST_NAME, 30);
console.log(p.name); // Prints "Anonymous"
```

Also, we can import everything from a file using an alias (without {}). Then, we can access all that's exported from the alias (it's like an object containing all exports).

```
import persons from './person.class'; // Import it with a name

let p = new persons.Person(persons.GUEST_NAME, 30);
console.log(p.name); // Prints "Anonymous"
```

Finally, if a file doesn't export anything but we want to load its contents (execute it and create variables, functions, etc.), we import it just using the name of the file.

```
import './functions.js'; // We can call those functions now
```

Loaders

Loaders are transformations applied to source before including it in the final bundle. It can be applied not also to JavaScript code but also to CSS, HTML, images, etc. For example, it allows you to import CSS files from your JavaScript code.

CSS loader

We can load CSS from our JavaScript files (which will be applied to the HTML that's loading that CSS) using the CSS loader. It's important to note that since we're loading CSS from a JavaScript file, we don't have to do it from an HTML file.

If we have our CSS divided in more than 1 file, this CSS loader supports the **@import** instruction to include more CSS files from a main file.

First, we install this loader (in combination with style loader) in our project:

```
npm i -D css-loader style-loader
```

After installing this loader, we can create a simple Webpack configuration for loading CSS files from JavaScript:

```
module.exports = {
  ...
  module: {
    rules: [
      {
        test: /\.css$/,
        exclude: /node_modules/,
        use: [ 'style-loader', 'css-loader' ]
      }
    ]
  }
}
```

Finally, we can import a CSS file in our JavaScript code (using this format):

```
import css from 'file.css';
```

ES2015 (Babel) loader

In order to translate ES2015 (or newer) into ES5 (compatible with older browsers), we can use the Babel compiler. To integrate it with Webpack we'll install babel and the babel loader.

```
npm install -D babel-loader @babel/core @babel/preset-env
npm install -S @babel/polyfill
```

Then, we have to tell Webpack to apply this loader to JavaScript files before bundling them:

```
        rules: [
            {
                test: /\.js$/,
                exclude: [/node_modules/],
                use: [{
                    loader: 'babel-loader',
                    options: { presets: ['@babel/env'] },
                }],
            },
            ...
        ]
```

Also import @babel/polyfill at the **top** (the first import) of each entry point file (the ones listed on the **entry** section of the Webpack configuration file:

```
import "@babel/polyfill";
import ...;
...
```

Handlebars loader

Handlebars is a useful HTML templating library (see Annex). Using the

Handlebars loader it can be integrated easily with Webpack. Handlebars templates will be precompiled and we won't need to include this library in our HTML (this loader will do it for us).

```
npm i -D handlebars handlebars-loader
```

Now, we configure this loader telling to compile files with **.handlebars** extension:

```
module.exports = {
    ...
    module: {
        rules: [
            ...
            { test: /\.handlebars$/, loader: "handlebars-loader" },
        ]
    }
};
```

Now, in the JavaScript file we want to use a template we must import it like this:

```
import productsTemplate from '../templates/products.handlebars';
```

And generate the corresponding HTML like this:

```
let htmlProds = productsTemplate(prodsJSON);
```

Plugins

Plugins are meant to extend Webpack capabilities, at least those we can't extend using loaders. A plugin (if you ever want to make one) is a JavaScript object that has at least a method called **apply()**, which webpack will call during compilation process.

SplitChunksPlugin

This plugin creates a separate file (chunk) with common modules (files) shared between more than 1 file (multiple entry points). By doing this, the common file will be loaded once and stored in browser's cache (when other page needs it, won't need to download it.

This is how we use this plugin in the configuration file:

```
let webpack = require('webpack');

module.exports = {
    ...
    optimization: {
        splitChunks: {
            cacheGroups: {
                commons: {
                    chunks: "initial", // Optimize chunks generation
                    name: "commons", // chunk name
                    minChunks: 2, // How many files import this chunk
                    minSize: 0 // Minimum size of the separated chunk
                }
            }
        }
    }
};
```

This means that we are going to create a file called **commons.js** (which you have to include in your HTML files that need it) with common modules (functions, classes, variables, etc…) imported from different entry points (pages). This is useful because once this file has been downloaded in the browser it will be in cache and won't be downloaded again when you load another page. If we didn't do this, this common functionality would be included in every page bundle.

```
### functions.js
export function sayHello() {
    console.log("Hello world!");
}

export function sayGoodbye() {
    console.log("Bye Bye!");
}

### index.js
import {sayHello, sayGoodbye} from './functions.js';
import css from './css/styles.css';

sayHello();
sayGoodbye();

### page2.js
import {sayHello} from './functions.js';
import css from '../styles.css';

sayHello();
```

```
Version: webpack 4.20.2
Time: 846ms
Built at: 2018-10-01 18:16:11
            Asset      Size  Chunks             Chunk Names
commons.bundle.js  6.05 KiB       0  [emitted]  commons
  index.bundle.js  1.54 KiB       1  [emitted]  index
  page2.bundle.js  1.52 KiB       2  [emitted]  page2
Entrypoint index = commons.bundle.js index.bundle.js
Entrypoint page2 = commons.bundle.js page2.bundle.js
[0] ./functions.js 251 bytes {0} [built]
[1] ../css/styles.css 1.05 KiB {0} [built]
[2] ./index.js 308 bytes {1} [built]
[3] ../node_modules/css-loader!../css/styles.css 183 bytes {0} [built]
[7] ./page2.js 278 bytes {2} [built]
    + 3 hidden modules
```

The generated **commons.js** file will include the contents of **functions.js** and **styles.css** (shared between 2 pages).

However, when we're using external libraries (usually installed with NPM and imported from **node_modules**), they will usually be included in the commons chunk if they're imported from more than 1 page. The problem comes when we change anything which affects the commons bundle, it will be regenerated entirely (and with a lot of libraries this can be slow).

As we know that external libraries are not going to change (maybe we'll include a new one or stop including another but not often), so we'll generate another bundle for these libraries. So now, when we change anything in our code which is used in many places, the commons chunk to be generated will be much smaller.

```
optimization: {
    splitChunks: {
        cacheGroups: {
            commons: {
                ...
            },
            vendors: {
                test: /[\\/]node_modules[\\/]/,
                name: 'vendors',
                chunks: 'all'
            }
        }
    }
}
```

Don't forget to include this new file in your HTML:

```
<script src="dist/vendors.bundle.js"></script>
<script src="dist/commons.bundle.js"></script>
<script src="dist/page1.bundle.js"></script>
```

Debugging code

When an error is shown on the console, you won't see the original code (everything is minified and bundled into one file. But you can generate **source map** files to help the browser's debugger know what the original code looks like.

The browser will still execute the bundled code, but it will look in the corresponding source map file when there's an error, and you will be able to see where it has occurred in your code.

Just add this option to the Webpack configuration file:

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

module.exports = {
    ...
    devtool: 'source-map',
    ...
}
```