ES6 : les modules

Lorsqu'une application JavaScript commence à prendre de l'ampleur il devient de plus en plus difficile de l'organiser et de la gérer. Il est alors judicieux de la découper en petits morceaux fonctionnels plus faciles à manipuler et dont l'intendance pose moins de problèmes.

Au-delà du découpage en fonctions et classes on peut aussi découper le code en modules cohérents ayant un certain niveau d'abstraction. D'autre part un module embarque son intendance et devient facile à utiliser et réutiliser. JavaScript ne connaît malheureusement pas les espaces de noms qui existent dans de multiples langages.

Avec ES5 on peut s'en sortir avec les objets et les fermetures et il existe une multitude d'implémentations. ES6 nous offre enfin une modularisation native !

Les modules ES6

Les modules ES6 :

- ont une syntaxe simple et sont basés sur le découpage en fichiers (un module = un fichier),
- sont automatiquement en mode « strict »,
- offrent un support pour un chargement asynchrone.

Les modules doivent exposer leurs variables et méthodes de façon explicite. On dispose donc des deux mots clés :

- **export** : pour exporter tout ce qui doit être accessible en dehors du module,
- **import** : pour importer tout ce qui doit être utilisé dans le module (et qui est donc exporté par un autre module).

Exporter et importer

Exporter

Puisque tout ce qui est écrit dans un module est interne à celui-ci il faut exporter ce qu'on veut rendre utilisable par les autres modules.

Pour exporter on utilise le mot-clé **export** :

```
export function rename(nom) {
  var gestionnaire = new Gestionnaire();
  return gestionnaire.changeNom(nom);
}

export class Identite {
  // code
}

function verifyIdentity() {
  // code
}
```

Ici on exporte la méthode **rename** et la classe **Identite**, par contre la méthode **verifyIdentity** reste interne au module.

Une autre façon de procéder est de regrouper ce qui doit être exporté :

```
function rename(nom) {
  var gestionnaire = new Gestionnaire();
  return gestionnaire.changeNom(nom);
}

class Identite {
  // code
}

function verifyIdentity() {
  // code
}
```

```
export { rename, Identite };
```

Vous pouvez placer cette liste n'importe où dans le code et même la scinder en plusieurs morceaux.

Importer

Pour importer dans un module quelque chose qui est exporté par un autre module on utilise le mot clé **import** :

```
import { rename, Identite } from "./identite.js";

import { rename as renameIdentite, Identite } from "./identite.js";
```

On peut aussi renommer ce qu'on exporte avec la même syntaxe.

Si on veut tout importer d'un module on peut utiliser cette syntaxe :

```
import * from "./identite.js";
```

Pour la résolution de l'emplacement du fichier voici un petit résumé :

- avec « / » la résolution se fait à la racine,
- avec « ./ » la résolution se fait dans le dossier actuel,
- avec « ../ » la résolution se fait dans le dossier parent.

Le chargement des modules

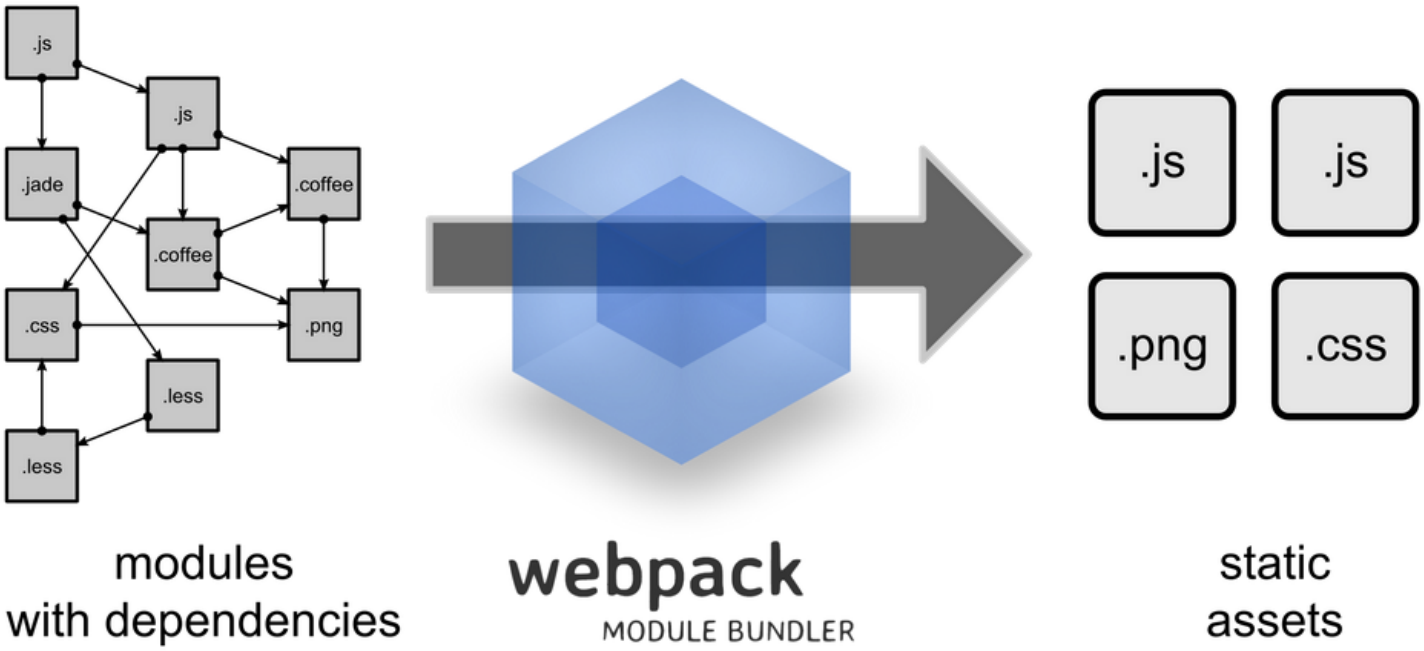
On a vu ci-dessus que la syntaxe est simple. Ce qui l'est moins c'est qu'actuellement elle n'est pas reconnue par nos navigateurs et il faut donc se débrouiller pour faire fonctionner tout ça.

D'autre part ES6 ne précise pas comment on doit charger les modules, autrement dit l'implémentation de cet aspect est laissé à la libre appréciation des développeurs, sans doute parce qu'il était difficile de définir une spécification universelle.

En plus pour le développement de l'aspect client d'une application (le frontend) on n'a pas seulement le JavaScript, il y a forcément

aussi du code CSS avec utilisation probable d'un préprocesseur genre Sass. Il serait donc bien de disposer d'un outil qui nous permette de gérer tout ça. La bonne nouvelle c'est que ça existe !

Il y a même plusieurs solutions mais celle qui semble avoir le plus de succès est Webpack. On lui donne des modules avec des dépendances (js, css, png…) et il transforme tout ça en assets statiques utilisables :



Exactement ce qu'il nous faut !

Webpack

Installation

Pour utiliser Webpack il faut commencer par l'installer. Mais il faut déjà disposer de:

- node.js, donc installez-le si vous ne l'avez pas, il vous servira pour bien d'autres choses !
- npm, c'est le gestionnaire de dépendances de **node.js**, lui aussi il faut l'installer si vous ne l'avez pas (la bonne nouvelle c'est qu'il s'installe avec **node.js**).

Vous pouvez alors installer Webpack :

```
npm install webpack -g
```

Avec l'option **-g** il s'installe globalement, vous en disposez donc partout, ce qui est plus simple. Mais dans la suite de cet article on va l'installer dans le projet.

Il faut ensuite initialiser **npm** :

```
npm init
```

Acceptez toutes les valeurs par défaut (elles ne présentent d'intérêt que si vous voulez publier un package), vous créez ainsi un fichier **package.json** de ce genre :

```
{
  "name": "test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Vous êtes maintenant prêt à installer des éléments dans votre projet. On va ajouter Babel qui va nous permettre de transformer le code ES6 en code ES5 (j'installe aussi Webpack dans le projet) :

```
npm install babel-core babel-loader babel-preset-es2015 webpack --save-dev
```

Il faut un petit moment pour que le dossier **node_modules** se crée et se remplisse de toutes les dépendances.

Si vous regardez votre fichier **package.json** vous allez trouver ces dépendances ajoutées :

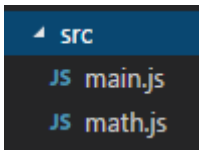
```
"devDependencies": {
  "babel-core": "^6.26.0",
  "babel-loader": "^7.1.2",
  "babel-preset-es2015": "^6.24.1",
```

```
  "webpack": "^3.10.0"
}
```

Il nous faut à présent une petite application pour essayer ça…

Une application d’exemple

On ne va pas se compliquer la vie en créant juste deux fichiers :



Un module (**math.js**) avec deux fonctions mathématiques exportées :

```
export function double(x) {
  return 2 * x;
}
export function multiplie(x, y) {
  return x * y;
}
```

Bon, c’est pour l’exemple !

Et un autre module (**main.js**) qui va utiliser ces fonctions en les important :

```
import * as math from "./math.js";

console.log("Le double de 3 est " + math.double(3));
console.log("La multiplication de 15 par 3 donne " + math.multiplie(15, 3));
```

Il nous faut enfin une page **index.html** pour charger le script généré :

```
<!DOCTYPE html>
<html lang="fr">
  <body>
  <script src="/dist/app.js"></script>
  </body>
</html>
```

Il nous faut donc le fichier JavaScript résultant (en ES5 après transformation par Babel) dans le dossier **dist** et qu'il s'appelle **app.js**.

Configurer et lancer Webpack

On va donc expliquer à **Webpack** ce qu'on veut faire. Il a besoin d'un fichier **webpack.config.js** avec ces renseignements :

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
    entry: './src/main.js',
    output: {
        path: path.resolve(__dirname, 'dist'),
        filename: 'app.js',
    },
    module: {
        loaders: [{
            test: /\.js$/,
            loader: 'babel-loader',
            query: {
                presets: ['es2015']
            }
        }]
    }
}
```

Voyons ça de plus près :

- **entry** : ici on donne le module d'entrée de l'application,
- **output** : ici on indique où doit se trouver le résultat,
- **module** : **loaders** : ici on ajoute des fonctionnalités à Webpack, on demande à Babel de charger les modules et de transformer le code en ES5 (**presets**).

Le **test** définit avec une expression régulière les fichiers à prendre en compte.

Il suffit maintenant de lancer Webpack :

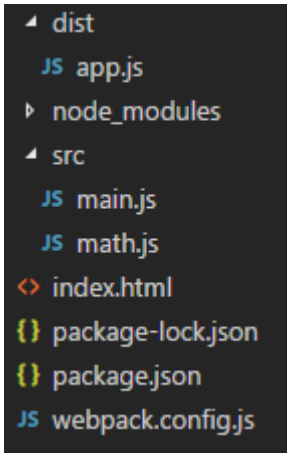
```
webpack
```

Et le fichier **dist/app.js** est créé ! Allez voir le code pour constater la transformation…

Pour avoir du code minifié pour la production il faut ajouter l'option **-p** :

```
webpack -p
```

Voilà un petit bilan des dossiers et fichiers :



Si vous ouvrez la page dans un navigateur vous allez normalement voir ceci dans la console :

```
Le double de 3 est 6
La multiplication de 15 par 3 donne 45
```

Si ce n'est pas le cas revoyez le processus décrit dans ce chapitre.

Pour avoir une compilation à chaque modification il faut ajouter **watch: true** dans le fichier **webpack.config.json** :

```
module.exports = {
    ...
    watch: true
}
```

Ce n'est qu'une introduction à Webpack et Babel, si vous voulez en savoir plus allez consulter leurs documentations.

En résumé

- ES6 offre la possibilité de diviser le code en modules distincts.
- Les modules communiquent avec des exportations et des importations.
- On peut utiliser Webpack et Babel pour charger les modules et convertir le code en ES5.