420-3U3-SO

Les modules en ES6

Jordan Rioux-Leclair

Présentation

- Introduction
- C’est quoi un module?
- Le mode strict
- Les fonctionnalités principales
- Les fonctionnalités spécifiques au navigateur

Introduction

Au fur et à mesure que notre application s'agrandit, nous voulons la diviser en plusieurs fichiers appelés **modules**. Un module peut contenir une classe ou une bibliothèque de fonctions dans un but précis.

Pendant longtemps, JavaScript a existé sans syntaxe de module au niveau du langage. Au départ, ce n'était pas un problème puisque les scripts étaient petits et simple. Cependant, les scripts sont devenus de plus en plus complexes, de sorte que la communauté a inventé une variété de façon d'organiser le code en modules en utilisant des bibliothèques spéciales pour charger des modules à la demande.

Le standard des modules au niveau du langage est apparu en 2015 et est maintenant supporté par tous les navigateurs principaux et par Node.js. Il n'est donc plus nécessaire d'utiliser des bibliothèques pour obtenir la fonctionnalité des modules, mais il se peut que vous voyez encore des vieux scripts utilisant ses bibliothèques.

Introduction

Anciennes bibliothèques

- **AMD** - https://requirejs.org/docs/whyamd.html#amd
- **CommonJS** - http://wiki.commonjs.org/wiki/Modules/1.1
- **UMD** - https://github.com/umdjs/umd

C’est quoi un module?

Un module n’est qu’un fichier. Un script est un module. C’est aussi simple que cela.

Les modules peuvent se charger les uns les autres et utiliser des directives spéciales d’exportation et d’importation pour échanger des fonctionnalités, appeler des fonctions d’un module à partir d’un autre.

- **export** indique les variables et les fonctions qui doivent être accessibles de l’extérieur du module courant.
- **import** permet l’importation de fonctionnalités à partir d’autres modules.

C’est quoi un module?

Prenons comme exemple, le module **say.js** qui fournit une fonction **say** pour dire un message. Pour ce module, on utilisera le **export** pour exporter la fonction qui nous intéresse.

Ensuite, nous aurions un autre fichier **main.js** qui pourra importer la fonction et l’utiliser grâce à **import**.

```
// say.js
export function say(message) {
  console.log(message);
}
```

```
// main.js
import { say } from './say.js';

say('Hello World!');
```

C’est quoi un module?

La directive **import** charge le module par le chemin relatif **‘./say.js’** par rapport au fichier courant et affecte la fonction exportée **say** à la variable correspondante.

Pour rappel, il s’agit du **Destructuring assignment** (ou plus spécifiquement du **Object Destructuring**). Le **import** du module va retourner un objet littéral avec toutes les variables et les fonctions exportées comme propriété de l’objet. On récupère donc seulement les propriétés (e.g. fonctions, etc.) qui nous intéressent en indiquant le nom des fonctions ce qui va créer automatiquement les variables faisant référence aux fonctions.

```
// say.js
export function say(message) {
  console.log(message);
}
```

```
// main.js
import { say } from './say.js';

say('Hello World!');
```

C’est quoi un module?

Object Destructuring

Avant de poursuivre sur les fonctionnalités principales des modules, voici un petit exemple pour vous remémorer le **Object Destructuring**.

```
const options = {
    width: 200,
    height: 120,
    cssClasses: 'modal-dialog-centered'
};

// On récupère seulement les propriétés désirées de l'objet
// en indiquant leur nom. Ceci va automatiquement créé des
// variables locales ayant le même nom que les propriétés
const { width, height } = options;

console.log(width);  // 200
console.log(height); // 120

// Sans Object Destructuring, il faut mentionner l'objet à chaque fois
console.log(options.width);   // 200
console.log(options.height);  // 120
```

C’est quoi un module?

Object Destructuring avec un module

Voyons maintenant un exemple avec un module qui exporte plusieurs fonctions.

```
// say.js
export function say(message) {
    console.log(message);
}

export function sayLoudly(message) {
    alert(message);
}

export function sayQuietly(message) {
    console.log(message); // shhhh! same code as say
}
```

```
// main.js
// NOTE: On importe pas la fonction sayQuietly ici, on ne l'utilisera pas
import { say, sayLoudly } from './say.js';

say('Hello World!');
sayLoudly('Hello everyone!!!');
```

C’est quoi un module?

Syntaxe du import

Il est important de noter qu’il y a plusieurs syntaxes différentes pour importer un module:

- https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/import

```
// Object Destructuring pour chaque propriété désiré
import { export1 } from "module-name";
import { export1, export2 } from "module-name";


// Utiliser un alias pour renommer une propriété (fonction, etc.)
import { export1 as alias1 } from "module-name";


// Il est possible de spécifier un export par défaut avec le mot-clé default.
// Un module ne peut qu'avoir une seule propriété en tant que export par défaut.
export default function say(message) {
    console.log(message)
}

import sayAndYouCanRenameTheDefaultNameIfYouWant from "say.js";


// Importer tout le module dans un objet "name"
import * as name from "module-name";
```

C’est quoi un module?

Syntaxe du export

Ceci est également vrai pour l’exportation:

- https://developer.mozilla.org/en-US/docs/web/javascript/reference/statements/export

```
// export devant chaque propriété que l'on veut rendre accessible de l'extérieur
export function say(message) {
    console.log(message)
}


// export default permet de spécifier la propriété exportée par défaut (un seul par module)
export default function cube(x) {
  return x * x * x;
}


function say(message) {
    console.log(message);
}

function sayLoudly(message) {
    alert(message);
}

// On peut combiner le export pour plusieurs propriétés
export { say, sayLoudly };
```

Le mode strict

Pendant longtemps, JavaScript a évolué sans problèmes de compatibilité. De nouvelles fonctionnalités ont été ajoutées au langage alors que les anciennes fonctionnalités n'ont pas changé.

Cela avait l'avantage de ne jamais cassé le code existant, mais l'inconvénient était que toute erreur ou décision imparfaite prise par les créateurs de JavaScript restait à jamais dans le langage.

Ce fut le cas jusqu'en 2009, lorsque ECMAScript 5 (ES5) est apparu. Il a ajouté de nouvelles fonctionnalités au langage et modifié certaines des fonctionnalités existantes. Pour que l'ancien code fonctionne, la plupart des modifications sont désactivées par défaut. Vous devez les activer explicitement avec une directive spéciale: “use strict”;

Le mode strict

Pour utiliser le nouveau mode strict, il suffit de mettre “use strict” comme première ligne d’un fichier JavaScript pour que tout le fichier soit interprété en mode strict (si le “use strict” n’est pas à la première ligne du fichier, alors il sera tout simplement ignoré). Il est également possible d’activer le mode strict pour une fonction en mettant le “use strict” au début de la fonction.

À noter qu’une fois le mode strict activé, il est impossible de le désactiver au moment de l’exécution.

Devrions-nous utiliser “use strict”?

Le JavaScript moderne (modules, classes, etc.) active automatiquement le mode strict. Nous n’avons donc pas besoin de le déclarer nous-même.

“use strict” peut être utiliser pour les anciens scripts, mais puisque notre code devrait être écrit avec des modules, il n’est donc pas nécessaire le mettre vu que le code sera automatiquement en mode strict.

Le mode strict

Vous pouvez voir les avantages du mode strict sur la documentation MDN:

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode

Voici comment activer le mode strict pour un fichier ou une fonction:

```
"use strict"; // le code sera en mode strict si “use strict” est la première ligne du fichier
...
```

```
// On peut l’activer pour une fonction aussi
function foo() {
    "use strict"; // le code de la fonction sera en mode strict
    ...
}
```

Les fonctionnalités principales

Toujours en mode strict

Les modules utilisent toujours le mode “use strict” par défaut.

Il est aussi important de noter que les modules fonctionnent seulement pour le protocole HTTP(S). Ils ne fonctionnent pas sur les fichiers locaux chargés avec *file://*. Il est donc obligatoire d’avoir un serveur Web pour utiliser les modules.

Pour débuter, il est recommandé d’utiliser l’extension **Live Server** disponible sur Visual Studio Code pour se créer facilement un serveur Web local. Par la suite, vous allez utiliser une machine virtuelle comme environnement de développement pour avoir accès à un serveur Web.

Les fonctionnalités principales

Portée au niveau du module

Chaque module a sa propre portée. En d'autres termes, les variables et fonctions d'un module ne sont pas vues dans les autres scripts.

Prenons comme exemple deux scripts qui sont importés individuellement en tant que module. Les fonctions du module **say.js** ne seront pas disponibles pour le module **app.js**. Il ne sera donc pas possible pour **app.js** d'utiliser les fonctions *say*, *sayLoudly* et *sayQuietly*.

```
<script type="module" src="say.js"></script>
<script type="module" src="app.js"></script>
```

Les fonctionnalités principales

Portée au niveau du module

On s'attend que les modules vont exporter ce qu'ils souhaitent rendre accessible de l'extérieur et vont importer ce dont ils ont besoin.

Nous devrions donc importer **say.js** dans **app.js** et récupérer les fonctionnalités requises au lieu de nous fier à des variables globales (comme dans l'exemple précédent).

```
// app.js
import { say, sayLoudly, sayQuietly } from './say.js';

say('Hello World!’);
sayQuietly('Hello World!');
sayLoudly('Hello everyone!!!');
```

Les fonctionnalités principales

Portée au niveau du module

Dans le navigateur, il existe aussi une portée indépendante de module au niveau de la balise script qui inclut le code directement (au lieu d'être un fichier externe).

```
<script type="module">
  // La variable est seulement visible pour ce module (e.g. pour l'intérieur de la balise script)
  let name = "Jordan";
</script>

<script type="module">
  alert(name); // Error: name is not defined
</script>
```

Si nous avons vraiment besoin de créer une variable globale, nous pouvons l'affecter explicitement à l'objet *window* et y accéder en tant que *window.name*. Cependant, ceci est un **cas exceptionnel et nécessite une bonne raison.**

Les fonctionnalités principales

Un code de module n’est évalué que la première fois lors de l’importation

Si le même module est importé à plusieurs autres endroits, son code n’est exécuté que la première fois, puis les exportations sont données à tous les importateurs.

Cela a des conséquences importantes.

Si l’exécution d’un code module entraîne des effets secondaires, comme l’affichage d’un message, alors l’importer plusieurs fois ne le déclenchera qu’une seule fois - la première fois.

```
// alert.js
alert("Module est évalué!");
```

```
// Importation du même module dans des fichiers différents

// module1.js
import `./alert.js`; // Le module est évalué!

// module2.js
import `./alert.js`; // Affiche rien!
```

Les fonctionnalités principales

Un code de module n’est évalué que la première fois lors de l’importation

En pratique, le code de module est principalement utilisé pour l’initialisation, la création de structures de données internes et, si nous voulons que quelque chose soit réutilisable, exportez-le.

Principalement, vous devez faire attention si votre module exécute du code pour faire une action (e.g. affichage d’une alerte) ou si votre module expose des variables qui peuvent être modifier de l’extérieur par un autre module.

Les fonctionnalités principales

import.meta

L’objet *import.meta* contient les informations sur le module actuel.

Son contenu dépend de l’environnement. Dans le navigateur, il contient l’URL du script ou l’URL de la page Web s’il s’agit d’un script *inline*.

```
<script type="module">
  alert(import.meta.url); // Affichera l'URL de la page Web vu que le script est inline
</script>
```

Les fonctionnalités principales

“this” n’est pas défini

Dans un module, *this* est non-défini (e.g. *undefined*).

En comparaison, dans un script non-module, *this* est l’objet global. Dans le cas d’un navigateur, *this* fait référence à *window*.

```
<script type="module">
  alert(this); // undefined
</script>
```

Les fonctionnalités spécifiques au navigateur

L’utilisation des modules dans le navigateur requiert d’ajouter l’attribute *type=“module”* à la balise *script* comme nous avons pu voir dans certains des exemples précédents.

Toutefois, il y a plusieurs différences entre une balise *script* avec le *type=“module”* et les balises *script* régulières que nous allons voir.

Les fonctionnalités spécifiques au navigateur

Les scripts de modules sont différés

Les scripts de module sont toujours *différés*, même effet que l'attribut *defer*, pour les scripts externes et inline.

En d'autres mots:

- Le téléchargement de scripts de module externes ne bloquent pas le traitement HTML. Ils se chargent en parallèle avec d'autres ressources.
- Les scripts de module attendent que le document HTML soit complètement prêt (même s'ils sont plus petits et se chargent plus rapidement que le HTML), puis s'exécutent.
- L'ordre relatif des scripts de module est conservé: les scripts qui vont en premier dans le document, s'exécutent en premier.

Comme effet secondaire, les scripts de module voient toujours la page HTML complètement téléchargée incluant les éléments HTML en dessous des scripts.

Les fonctionnalités spécifiques au navigateur

Les scripts de modules sont différés

```
<script type="module">
  alert(typeof button); // object: le script peut voir le bouton dans le HTML
  // les modules sont différés, le script va donc s'exécuter après que la page soit complètement chargée
</script>

<script>
  alert(typeof button); // Error: button is undefined, le script ne peut pas voir les éléments en dessous
  // les scripts non-modules s'exécutent immédiatement, avant que le reste de la page soit chargée
</script>

<button id="button">Button</button>
```

NOTE: Le deuxième script va s'exécuter en premier puisque les scripts de modules sont différés et doivent donc attendre que le document soit processé.

Lors de l'utilisation de modules, nous devons être conscient que la page HTML apparaît lors du chargement et que les modules s'exécutent après cela, de sorte que l'utilisateur peut voir la page avant que l'application ne soit prête. Certaines fonctionnalités peuvent ne pas encore fonctionner. Nous devrions mettre des indicateurs de chargement ou nous assurer que le visiteur ne sera pas confus par cela.

Les fonctionnalités spécifiques au navigateur

Async fonctionne sur les scripts de module inline

Pour les scripts non-module, l'attribut *async* fonctionne seulement sur les scripts externes. Un script *async* s'exécute immédiatement lorsque prêt, indépendamment des autres stricts dans le document HTML.

Pour les scripts de module, l'attribut *async* fonctionne sur les scripts *inline*.

Par exemple, le script *inline* ci-dessous a l'attribut *async* donc il n'attend rien. Il effectue l'importation (récupère **./analytics.js**) et s'exécute lorsqu'il est prêt, même si le document HTML n'est pas encore terminé ou si d'autres scripts sont encore en attente.

L'attribut async est bon pour les fonctionnalités qui ne dépendent de rien, comme les compteurs, les annonces, les écouteurs d'événements au niveau du document, etc.

```
<script async type="module">
  import { counter } from './analytics.js';

  counter.count();
</script>
```

Les fonctionnalités spécifiques au navigateur

Les scripts externes

Les scripts externes de module sont différents pour deux aspects:

1. Les scripts externes avec la même **src** s'exécute une seule fois.

2. Les scripts externes qui sont récupérés à partir d'une autre origine (e.g. un autre site) nécessitent des en-têtes CORS. En d'autres termes, si un script de module est extrait d'une autre origine, le serveur distant doit fournir un en-tête *Access-Control-Allow-Origin* permettant la récupération.

Les en-têtes CORS seront vues plus en détails dans le côté *Sécurité Applicative* en *Web IV*.

Les fonctionnalités spécifiques au navigateur

Aucun module “nu” autorisé (*bare* modules)

Dans le navigateur, **import** doit avoir une URL relative ou absolue. Les modules sans chemin sont appelés modules “nus”. Ces modules ne sont pas autorisés à l’importation.

Il faut également préciser l’extension du fichier (e.g. .js).

```
import { say } from 'say'; // Error, "bare" module
// Le module doit avoir un chemin comme './say.js'
```

Les fonctionnalités spécifiques au navigateur

Compatibilité “nomodule”

Les anciens navigateurs ne comprennent pas *type=“module”*. Les script d’un type inconnu sont simplement ignorés. Pour eux, il est possible de fournir une solution de secours en utilisant l’attribut *nomodule*.

```
<script type="module">
  alert("Navigateur moderne!");
</script>

<script nomodule>
  alert("Navigateur moderne va ignorer ce script vu qu'ils connaissent le type=module et nomodule")
  alert("Les anciens navigateurs vont ignorer le script avec type=module (inconnu pour eux), mais exécuté ce script.");
</script>
```

Les fonctionnalités spécifiques au navigateur

Outils de *build*

Dans la vie réelle, les modules de navigateurs sont rarement utilisés sous leur forme “brute”. Habituellement, nous les regroupons avec un outil spécial tel que Webpack et les déployons sur le serveur de production.

L’un des avantages de l’utilisation des *bundlers* - ils donnent plus de contrôle sur la façon dont les modules sont résolus, autorisant des modules “nus” et bien plus encore, comme les modules CSS / HTML.

Si nous utilisons des outils de bundle, alors que les scripts sont regroupés dans un seul fichier (ou quelques fichiers), les instructions **import** / **export** à l’intérieur de ces scripts sont remplacées par des fonctions de *bundler* spéciales. Ainsi, le script “*bundle*” résultant ne contient aucun **import** / **export**, il ne nécessite donc pas de type=“module” et nous pouvons le mettre dans un script normal.

Les fonctionnalités spécifiques au navigateur

Outils de *build*

Les outils de *build* font ce qui suit:

1. Prenne un module “main”, celui destiné à être mis dans <script type=“module”> en HTML.
2. Analyse ces dépendances: **import** et **import** des **import**, etc.
3. Crée un (ou plusieurs) fichier(s) avec tous les modules en remplaçant les appels d’importations natifs par des fonctions du *bundler*. Les types de module “spéciaux” tels que les modules HTML / CSS sont également pris en charge par les *bundler*.
4. Dans le processus, d’autres transformations et optimisations peuvent être appliqués:
   - Code inaccessible supprimé
   - Les exportations inutilisées sont supprimées
   - Les déclarations spécifiques au développement telles que la console et debugger sont supprimées
   - La syntaxe moderne peut être transformée en une syntaxe plus ancienne à l’aide de Babel
   - Le fichier résultant est vinifié (espaces supprimés, variables remplacées par noms plus courts, etc.)