



Les modules JavaScript

« Précédent

Ce guide aborde l'ensemble des notions vous permettant d'utiliser la syntaxe des modules en JavaScript.

Un peu de contexte

Les programmes JavaScript ont commencé par être assez petits, réalisant des tâches isolées uniquement là où l'interactivité était nécessaire. Après plusieurs années, nous avons maintenant des applications complètes qui sont exécutées dans les navigateurs avec des codes complexes et volumineux. Des programmes JavaScript sont également exécutés dans d'autres contextes ([Node.js](#) par exemple).

Il a donc été question de fournir un mécanisme pour diviser les programmes JavaScript en plusieurs modules qu'on pourrait importer les uns dans les autres. Cette fonctionnalité était présente dans Node.js depuis longtemps et plusieurs bibliothèques et *frameworks* JavaScript ont permis l'utilisation de modules ([CommonJS](#), [AMD](#), [RequireJS](#) ou, plus récemment, [Webpack](#) et [Babel](#)).

Bonne nouvelle, les navigateurs ont également commencé à prendre en charge ces fonctionnalités nativement. C'est le sujet de ce guide.

Cette implémentation permettra aux navigateurs d'optimiser le chargement des modules, rendant le fonctionnement plus efficace qu'une bibliothèque tierce avec un traitement côté client des allers-retours sur le réseau.

Compatibilité des navigateurs

L'utilisation des modules natifs JavaScript repose sur les instructions `import` and `export` dont vous pouvez voir l'état de la compatibilité ici :

`import`

Report problems with this compatibility data on GitHub

import

Chrome	61
--------	----

Edge	16
------	----

Firefox	60
---------	----

Internet Explorer	No
--------------------------	-----------

Opera	48
-------	----

Safari	10.1
--------	------

WebView Android	61
-----------------	----

Chrome Android	61
----------------	----

Firefox Android	60
-----------------	----

Opera Android	45
---------------	----

iOS Safari	10.3
------------	------

Samsung Internet	8.0
------------------	-----

Node.js	13.2.0
---------	--------

Dynamic import

Chrome	63
--------	----

Edge	79
------	----

Firefox	67
---------	----

Internet Explorer	No
--------------------------	-----------

Opera	50
-------	----

Safari	11.1
--------	------

WebView Android	63
-----------------	----

Chrome Android	63
----------------	----

Firefox Android	67
-----------------	----

Opera Android	46
---------------	----

Samsung Internet	8.0
Node.js	13.2.0

Available in workers

Chrome	80
Edge	80
Firefox	No
Internet Explorer	No
Opera	No
Safari	No
WebView Android	80
Chrome Android	80
Firefox Android	No
Opera Android	No
iOS Safari	No
Samsung Internet	No
Node.js	No

Full support

No support

See implementation notes.

User must explicitly enable this feature.

export

Report problems with this compatibility data on GitHub

export

Chrome	61
--------	----

Edge	16
------	----

Firefox	60
---------	----

Internet Explorer	No
--------------------------	-----------

Opera	48
-------	----

Safari	10.1
--------	------

WebView Android	61
-----------------	----

Chrome Android	61
----------------	----

Firefox Android	60
-----------------	----

Opera Android	45
---------------	----

iOS Safari	10.3
------------	------

Samsung Internet	8.0
------------------	-----

Node.js	13.2.0
---------	--------

default keyword with export

Chrome	61
--------	----

Edge	16
------	----

Firefox	60
---------	----

Internet Explorer	No
--------------------------	-----------

Opera	48
-------	----

Safari	10.1
--------	------

WebView Android	No
------------------------	-----------

Chrome Android	61
----------------	----

Firefox Android	60
-----------------	----

Opera Android	45
iOS Safari	10.3
Samsung Internet	8.0
Node.js	13.2.0
<hr/>	
<code>export * as namespace</code>	
Chrome	72
Edge	79
Firefox	80
<hr/>	
Internet Explorer	No
Opera	60
<hr/>	
Safari	No
<hr/>	
WebView Android	No
Chrome Android	72
Firefox Android	80
Opera Android	51
<hr/>	
iOS Safari	No
<hr/>	
Samsung Internet	11.0
Node.js	12.0.0

Full support

No support

See implementation notes.

User must explicitly enable this feature.

Commençons par un exemple

Pour illustrer le fonctionnement des modules, nous avons créé [un ensemble d'exemples disponibles sur GitHub](#). Ces exemples illustrent un ensemble de modules pour créer un

élément `<canvas>` sur une page web puis dessiner (et afficher des informations) sur les différentes formes du canevas.

Ces opérations sont assez simples mais nous les avons choisies pour nous concentrer plutôt sur le fonctionnement des modules.

Note : Si vous souhaitez télécharger les exemples et les exécuter en local, vous devrez utiliser un serveur web local.

Structure de l'exemple

Dans notre premier exemple (cf. [basic-modules](#)), nous avons l'arborescence de fichier suivante :

```
index.html
main.mjs
modules/
  canvas.mjs
  square.mjs
```

Note : Tous les exemples de ce guide suivent la même structure.

Le répertoire dédié aux modules contient deux modules :

- `canvas.mjs` — responsable de fonctions pour gérer le canevas
 - `create()` — crée un canevas avec les dimensions souhaitées (`width / height`) à l'intérieur d'un élément `<div>` doté d'un identifiant et qui est ajouté à l'intérieur d'un élément indiqué. Cette fonction renvoie l'objet contenant le contexte du canevas et l'identifiant du conteneur.
 - `createReportList()` — crée une liste non ordonnée à l'intérieur d'un élément indiqué et dans lequel on affiche des données. Cette fonction renvoie l'identifiant de la liste.
- `square.mjs` ·

- name — une constante qui est une chaîne de caractères : "square".
- draw() — dessine un carré avec une taille/position/couleur données sur le canevas indiqué. Cette fonction renvoie un objet contenant la taille du carré, sa position et sa couleur.

- reportArea() — écrit la surface d'un carré dans une liste donnée en fonction de la longueur de son côté.
- reportPerimeter() — écrit le périmètre d'un carré dans une liste donnée en fonction de la longueur de son côté.

Note: Pour les modules JavaScript natifs, l'extension .mjs a son importance car elle permet d'importer des fichiers avec un type MIME javascript/esm (on pourra utiliser une autre extension qui fournira le type MIME application/javascript) afin d'éviter les erreurs liées à la vérification des types MIME. L'extension .mjs est notamment utile afin de distinguer plus clairement les scripts « classiques » des modules et pourra être exploitée par d'autres outils. Pour plus de détails, voir [cette note de Google](#).

Exporter des fonctionnalités

Pour commencer et afin d'utiliser les fonctionnalités d'un module, on devra les exporter. Pour cela, on utilisera l'instruction `export`.

La méthode la plus simple consiste à placer cette instruction devant chaque valeur qu'on souhaite exporter, par exemple :

```
export const name = 'square';

export function draw(ctx, length, x, y, color) {
  ctx.fillStyle = color;
  ctx.fillRect(x, y, length, length);

  return {
    length: length,
    x: x,
    y: y,
    color: color
  };
}
```

Il est possible d'exporter des fonctions, des variables (qu'elles soient déclarées avec var , let ou const) et aussi des classes (que nous verrons par la suite). Les valeurs exportées doivent être présentes au plus haut niveau du script, il n'est pas possible d'utiliser export dans une fonction.

Une méthode plus concise consiste à exporter l'ensemble des valeurs grâce à une seule instruction située à la fin du fichier : les valeurs sont séparées par des virgules et la liste est délimitée entre accolades :

```
export { name, draw, reportArea, reportPerimeter };
```

Importer des fonctionnalités

Lorsque des fonctionnalités sont exportées par un premier module, on peut les importer dans un script afin de les utiliser. Voici la méthode la plus simple pour ce faire :

```
import { name, draw, reportArea, reportPerimeter } from './modules/square.mjs'
```

On utilise ici l'instruction import , suivi d'une liste d'identifiants séparées par des virgules et délimitée par des accolades, suivie du mot-clé from puis du chemin vers le fichier du module. Le chemin est relatif à la racine du site. Dans notre cas, pour basic-module , on écrira /js-examples/modules/basic-modules .

Ici, nous avons écrit le chemin d'une façon légèrement différente : on utilise le point (.) afin d'indiquer « l'emplacement courant », suivi du chemin vers le fichier. Cela permet d'éviter d'avoir à écrire l'intégralité du chemin à chaque fois, c'est aussi plus court et cela permet de déplacer le script et le modules sans avoir à modifier les scripts.

Ainsi :

```
/js-examples/modules/basic-modules/modules/square.mjs
```

devient :

```
./modules/square.mjs
```

Vous pouvez voir ces lignes dans [main.mjs](#) .

Note : Pour certains systèmes de module, on peut omettre l'extension de fichier et le point (c'est-à-dire qu'on peut écrire '/modules/square'). Cela ne fonctionne pas pour les modules JavaScript !

Une fois les fonctionnalités importées dans le script, vous pouvez utiliser les valeurs dans votre script. Dans `main.mjs`, après les lignes d'import, on trouvera :

```
let myCanvas = create('myCanvas', document.body, 480, 320);
let reportList = createReportList(myCanvas.id);

let square1 = draw(myCanvas.ctx, 50, 50, 100, 'blue');
reportArea(square1.length, reportList);
reportPerimeter(square1.length, reportList);
```

Charger le module via le document HTML

Il faut ensuite pouvoir charger le script `main.mjs` sur la page HTML. Pour cela, nous allons voir qu'il y a quelques différences avec le chargement d'un script « classique ».

Tout d'abord, il est nécessaire d'indiquer `type="module"` dans l'élément `<script>` afin d'indiquer qu'on charge des modules :

```
<script type="module" src="main.mjs"></script>
```

Le script qu'on importe ici agit comme le module de plus haut niveau. Si on oublie ce type, Firefox déclenchera une erreur "*SyntaxError: import declarations may only appear at top level of a module*".

Les instructions `import` et `export` ne peuvent être utilisées qu'à l'intérieur de modules et pas à l'intérieur de scripts « classiques ».

Note : Il est aussi possible d'importer des modules dans des scripts qui sont déclarés en incise si on indique bien `type="module"`. On pourra donc écrire `<script type="module"> //code du script utilisant les modules ici </script>`.

Différences entre les modules et les scripts « classiques »

- ATTENTION AUX TESTS SUR UN ENVIRONNEMENT LOCAL : SI VOUS CHARGEZ LE FICHIER HTML

directement depuis le système de fichier dans le navigateur (en double-cliquant dessus par exemple, ce qui donnera une URL `file://`), vous rencontrerez des erreurs CORS pour des raisons de sécurité. Il faut donc un serveur local afin de pouvoir tester.

- On pourra avoir un comportement différent entre un même script utilisé comme un module et un script utilisé de façon « classique ». En effet, les modules utilisent automatiquement [le mode strict](#).
- Il n'est pas nécessaire d'utiliser l'attribut `defer` (voir [les attributs de <script>](#)) lors du chargement d'un module, ceux-ci sont automatiquement chargés à la demande.
- Enfin, les fonctionnalités importées ne sont disponibles qu'au sein de la portée du script qui les utilise ! Les valeurs importées ne sont manipulables que depuis le script, elles ne sont pas rattachées à la portée globale. On ne pourra par exemple pas y accéder depuis la console JavaScript. Bien que les erreurs soient toujours indiquées dans les outils de développement, certaines techniques de débogage ne seront pas disponibles.

Exports par défaut et exports nommés

Jusqu'à présent, nous avons utilisé des **exports nommés** — chaque valeur est exportée avec un nom et c'est ce nom qui est également utilisé lorsqu'on réalise l'import.

Il existe également un **export par défaut** — conçu pour simplifier l'export d'une fonction par module et pour faciliter l'interopérabilité avec les systèmes de module CommonJS et AMD (pour plus d'informations, voir [ES6 en détails : les modules](#)).

Prenons un exemple pour comprendre le fonctionnement des exports par défaut. Dans `square.mjs`, on a une fonction intitulée `randomSquare()` qui crée un carré avec une taille/couleur/position aléatoires. On souhaite exporter cette fonction par défaut et on écrit donc ceci à la fin du fichier :

```
export default randomSquare;
```

On notera ici l'absence d'accolades.

On aurait également pu ajouter `export default` devant le mot-clé `function` et la définir comme fonction anonyme :

```
export default function(ctx) {  
  ...  
}
```

Dans le fichier `main.mjs`, on importe la fonction par défaut avec cette ligne

```
import randomSquare from './modules/square.mjs';
```

On voit ici aussi l'absence d'accolade car il n'y a qu'un seul export par défaut possible par module (et ici, on sait qu'il s'agit de `randomSquare`). La ligne ci-dessus est en fait une notation raccourcie équivalente à :

```
import {default as randomSquare} from './modules/square.mjs';
```

Note : Pour en savoir plus sur le renommage des objets exportés, voir ci-après [Renommage des imports et des exports](#).

Gestion des conflits de nommage

Jusqu'à présent, notre exemple fonctionne. Mais que se passerait-il si nous ajoutions un module permettant de dessiner une autre forme comme un cercle ou un triangle ? Ces formes disposeraient sans doute également de fonctions telles que `draw()`, `reportArea()`, etc. Si on essaie d'importer ces fonctions avec les mêmes noms dans le module de plus haut niveau, nous allons avoir des conflits et des erreurs.

Heureusement, il existe différentes façons de résoudre ce problème.

Renommage des imports et des exports

Entre les accolades utilisées pour les instructions `import` et `export`, on peut utiliser le mot-clé `as` avec un autre nom afin de modifier le nom par lequel on souhaite identifier la fonctionnalité.

Ainsi, les deux fragments qui suivent permettraient d'obtenir le même résultat de façons différentes :

```
// dans module.mjs
export {
  fonction1 as nouveauNomDeFonction,
  fonction2 as autreNouveauNomDeFonction
};

// dans main.mjs
import { nouveauNomDeFonction, autreNouveauNomDeFonction } from './modules/'
```

```
// dans module.mjs
export { fonction1, fonction2 };

// dans main.mjs

import { fonction1 as nouveauNomDeFonction,
         fonction2 as autreNouveauNomDeFonction } from './modules/module.mjs'
```

Prenons un exemple concret. Dans le répertoire [renaming](#), vous verrez le même système de modules que précédemment auquel nous avons ajouté `circle.mjs` et `triangle.mjs` afin de dessiner et d'écrire des informations sur des cercles et des triangles.

Dans chaque module, on exporte les fonctionnalités avec des noms identiques : l'instruction `export` utilisée est la même à chaque fin de fichier :

```
export { name, draw, reportArea, reportPerimeter };
```

Lorsqu'on importe les valeurs dans `main.mjs`, si on essaie d'utiliser

```
import { name, draw, reportArea, reportPerimeter } from './modules/square.mjs'
import { name, draw, reportArea, reportPerimeter } from './modules/circle.mjs'
import { name, draw, reportArea, reportPerimeter } from './modules/triangle.mjs'
```

Le navigateur déclenchera une erreur telle que "*SyntaxError: redeclaration of import name*" (Firefox).

Pour éviter ce problème, on renomme les imports afin qu'ils soient uniques :

```
import { name as squareName,
         draw as drawSquare,
         reportArea as reportSquareArea,
         reportPerimeter as reportSquarePerimeter } from './modules/square.mjs'

import { name as circleName,
         draw as drawCircle,
         reportArea as reportCircleArea,
         reportPerimeter as reportCirclePerimeter } from './modules/circle.mjs

import { name as triangleName,
         draw as drawTriangle }
```

```
// dans triangle.mjs
reportArea as reportTriangleArea,
reportPerimeter as reportTrianglePerimeter } from './modules/triangl
```

On aurait pu également résoudre le problème dans les fichiers de chaque module.

```
// dans square.mjs
export { name as squareName,
          draw as drawSquare,
          reportArea as reportSquareArea,
          reportPerimeter as reportSquarePerimeter };

// dans main.mjs
import { squareName, drawSquare, reportSquareArea, reportSquarePerimeter } f
```

Les deux approches fonctionnent. C'est à vous de choisir le style. Toutefois, il est souvent plus pratique d'effectuer le renommage à l'import, notamment lorsqu'on importe des fonctionnalités de modules tiers sur lesquels on n'a pas le contrôle.

Créer un objet module

La méthode précédente fonctionne mais reste « brouillonne ». Pour faire mieux, on peut importer l'ensemble des fonctionnalités de chaque module dans un objet, de la façon suivante :

```
import * as Module from './modules/module.mjs';
```

Cela récupère tous les exports disponibles depuis `module.mjs` et les transforme en propriétés et méthodes rattachées à l'objet `Module` qui fournit alors un espace de noms (*namespace*) :

```
Module.function1()
Module.function2()
etc.
```

Là encore, prenons un exemple concret avec le répertoire [module-objects](#). Il s'agit du même exemple que précédemment mais qui a été réécrit afin de tirer parti de cette syntaxe. Dans les modules, les exports sont tous écrits ainsi :

```
export { name, draw, reportArea, reportPerimeter };
```

En revanche, pour les imports, on les récupère ainsi :

```
import * as Canvas from './modules/canvas.mjs';
import * as Square from './modules/square.mjs';
import * as Circle from './modules/circle.mjs';
import * as Triangle from './modules/triangle.mjs';
```

Dans chaque cas, on peut accéder aux imports comme propriétés des objets ainsi créés :

```
let square1 = Square.draw(myCanvas.ctx, 50, 50, 100, 'blue');
Square.reportArea(square1.length, reportList);
Square.reportPerimeter(square1.length, reportList);
```

On obtient alors un code plus lisible.

Classes et modules

Comme mentionné avant, il est possible d'importer et d'exporter des classes. Cette méthode peut aussi être utilisée afin d'éviter les conflits de nommage. Elle s'avère notamment utile lorsque vous utilisez déjà des classes pour construire vos objets (cela permet de garder une certaine cohérence dans le style).

Pour voir le résultat obtenu, vous pouvez consulter le répertoire `classes` du dépôt où l'ensemble a été réécrit pour tirer parti des classes ECMAScript. Ainsi, `square.mjs` contient désormais l'ensemble des fonctionnalités via une classe :

```
class Square {
  constructor(ctx, listId, length, x, y, color) {
    ...
  }

  draw() {
    ...
  }

  ...
}
```

Il suffit d'exporter cette classe :

```
export { Square };
```

Puis de l'importer ainsi dans `main.mjs` :

```
import { Square } from './modules/square.mjs';
```

Ensuite, on peut utiliser cette classe afin de dessiner le carré :

```
let square1 = new Square(myCanvas.ctx, myCanvas.listId, 50, 50, 100, 'blue')
square1.draw();
square1.reportArea();
square1.reportPerimeter();
```

Agréger des modules

Il arrivera qu'on veuille agréger des modules entre eux. On peut avoir plusieurs niveaux de dépendances et vouloir simplifier les choses en combinant différents sous-modules en un seul module parent. Pour cela, on pourra utiliser la notation raccourcie suivante :

```
export * from 'x.mjs'
export { name } from 'x.mjs'
```

Pour voir cela en pratique, vous pouvez consulter le répertoire [module-aggregation](#). Dans cet exemple (construit sur le précédent qui utilise les classes), on a un module supplémentaire intitulé `shapes.mjs` qui agrège les fonctionnalités fournies par `circle.mjs`, `square.mjs` et `triangle.mjs`. Les sous-modules ont également été déplacés dans un répertoire `shapes` situé dans un répertoire `modules`. L'arborescence utilisée est donc :

```
modules/
  canvas.mjs
  shapes.mjs
  shapes/
    circle.mjs
    square.mjs
    triangle.mjs
```

Dans chaque sous-module, l'export aura la même forme :

```
export { Square };
```

Pour l'agrégation au sein de `shapes.mjs`, on écrit les lignes suivantes :

```
export { Square } from './shapes/square.mjs';
export { Triangle } from './shapes/triangle.mjs';
export { Circle } from './shapes/circle.mjs';
```

On récupère ainsi l'ensemble des exports de chaque module et on les rend disponibles via `shapes.mjs`.

Note : Cette notation ne permet que de rediriger les exports via le fichier. Les objets importés/exportés n'existent pas vraiment dans `shapes.mjs` et on ne peut donc pas écrire de code *utile* qui les manipule.

Dans le fichier `main.mjs`, on pourra alors remplacer :

```
import { Square } from './modules/square.mjs';
import { Circle } from './modules/circle.mjs';
import { Triangle } from './modules/triangle.mjs';
```

par :

```
import { Square, Circle, Triangle } from './modules/shapes.mjs';
```

Chargement dynamique de modules

Cette nouvelle fonctionnalité permet aux navigateurs de charger les modules lorsqu'ils sont nécessaires plutôt que de tout précharger en avance de phase. Cette méthode offre de nombreux avantages quant aux performances. Voyons comment cela fonctionne.

Pour utiliser cette fonctionnalité, on pourra utiliser `import()` comme une fonction et lui passer le chemin du module en argument. Cette fonction renverra **une promesse**, qui sera résolue en un module objet donnant accès aux exports.

```
import('./modules/monModule.mjs')
  .then((module) => {
    // Faire qqc avec le module.
  });

```

Dans nos exemples, regardons le répertoire `dynamic-module-imports`, également basé sur les classes. Cette fois, on ne dessine rien au chargement de l'exemple mais on ajoute trois boutons — "Circle", "Square" et "Triangle" — qui, lorsqu'ils seront utilisés, chargeront dynamiquement les modules nécessaires et les utiliseront pour charger la forme associée.

Dans cet exemple, nous avons uniquement modifié `index.html` et `main.js` — les exports restent les mêmes.

Dans `main.js`, on récupère une référence à chaque bouton en utilisant `document.querySelector()`. Par exemple :

```
let squareBtn = document.querySelector('.square');
```

Ensuite, on attache un gestionnaire d'évènement à chaque bouton afin qu'on puisse appuyer dessus. Le module correspondant est alors chargé dynamiquement et utilisé pour dessiner la forme :

```
squareBtn.addEventListener('click', () => {
  import('./modules/square.mjs').then((Module) => {
    let square1 = new Module.Square(myCanvas.ctx, myCanvas.listId, 50, 50, 1
    square1.draw();
    square1.reportArea();
    square1.reportPerimeter();
  })
});
```

On voit ici que, parce que la promesse renvoie un objet module à la résolution, la classe est une propriété de cet objet et qu'il faut ajouter cet espace de nom devant le constructeur exporté pour l'utiliser. Autrement dit, avec cette méthode, on doit ajouter `Module.` devant `Square` (plutôt que d'utiliser uniquement `Square`).

Diagnostiquer les problèmes avec les modules

Voici quelques notes pour aider à comprendre et à diagnostiquer les problèmes parfois rencontrés avec les modules. N'hésitez pas à ajouter vos conseils à cette liste si vous en avez.

- Comme indiqué ci-dessus, les fichiers `.mjs` doivent être chargés avec le type MIME `javascript/esm` (ou avec un autre type MIME compatible JavaScript tel que `application/javascript`), sinon on aura une erreur lors de la vérification du type MIME.
- Si on essaie de charger des fichiers HTML en local à l'aide d'une URL `file://`, on aura des erreurs CORS relatives à la sécurité. Pour tester les modules, on doit donc mettre en place un serveur (ou, par exemple, utiliser les pages GitHub).
- `.mjs` est une extension relativement récente et certains systèmes d'exploitation ne la

reconnaîtront pas et/ou tenteront de la remplacer (ex. macOS pourra silencieusement ajouter un `.js` après le `.mjs`). Dans ce cas, afficher les extensions de tous les fichiers par défaut pourra permettre de vérifier.

Voir aussi

- [Une plongée illustrée dans les modules ECMAScript](#)
- [ES6 en détails : les modules](#)
- [Utiliser les modules JavaScript sur le Web](#), un article par Addy Osmani et Mathias Bynens (en anglais)
- Livre de Axel Rauschmayer (en anglais) : [Exploring JS: Modules](#)

[« Précédent](#)

Last modified: 16 oct. 2020, by [MDN contributors](#)