Javascript : les modules

1. Introduction

Une application d’envergure a plusieurs fichiers.

Chaque fichier correspond à un module en javascript de nos jours.

Jusqu’à ES6 il n’existait pas de système de module.

L’utilisation des modules (type = module) est toujours différé (defer), il ne bloque pas le traitement HTML qui se charge en parallèle.

L’ordre relatif des scripts est maintenu : les scripts qui entrent en premier dans le document est toujours en premier.

Dans notre cours nous utiliserons le système préconisé.

Avec nodeJs nous verrons les modules avec require.

Il fallait utilisait :

* AMD
* CommonJS
* Ou UMD

Avant ES6 on pouvait aussi utiliser les IIFE pour isoler chaque fichier et éviter les conflits.

1. Exporter

Pour exporter des fichiers nous utilisons le mot clef « export »

export const tableau = [1, 2, 3];

export function Compter() {}

export class User {}

1. Déclarer un objet d’export

On utilise les {} pour exporter l’ensemble des objets à exporter.

const tableau = [1, 2, 3];

function Compter() {}

class User {}

export { tableau, Compter, User }

1. Modifier le nom d’un identifiant pour l’export

le mot clef as permet de donner un alias à notre export :

const tableau = [1, 2, 3];

function Compter() {}

class User {}

export { tableau as tableauNombres, Compter as CompterUsers, User }

1. Export par défaut

L’export par defaut permet de déclarer un unique export qui sera utilisé lorsque nous ne précisons pas ce que nous souhaitons importer dans les modules de définitions.

export default function Compter() {}

1. Importer
2. Importer des identifiants exportés explicitement

import { export1, export2, ... } from "chemin-vers-module";

* un exemple complet d’import / export :
* export :

export const tableau = [1, 2, 3];

export function Compter() {}

export class User {}

import { tableau , Compter, User } from "chemin-vers-module";

* import

import { tableau , Compter, User } from "chemin-vers-module";

1. importer l’intégralité d’un module

on utilise le \* et un alias pour effectuer cette tache

import \* as monModule from 'chemin-vers-module';

// Accès aux exports

monModule.tableau;

monModule.Compter;

monModule.User;-module";

A utiliser avec parcimonie.

1. Importer en utilisant un alias

On a la possibilité de rajouter un alias à un objet qu’on importe.

import { tableau as tableauNombre, Compter, User } from "chemin-vers-module";

1. Importation par défaut

L’importation par défaut n’utilise pas les {}

import unIdentifiant from 'chemin-vers-module';

1. Réexporter des modules

Il est possible d’importer et d’immédiatement réexporter des identifiants.

Pour cela on utilise la paire export … from

export { monExport } from 'chemin-du-module';

C’est utile quand l’application commence à devenir importante et qu’il existe un grand nombre de modules.

Il est courant de créer un fichier un fichier index.js dans un dossier qui ne contiendra que les exports.

Ex dans un dossier authentification :

// 📁 authentification/index.js

export { signup } from 'chemin-module-inscription';

export { login, logout } from 'chemin-module-connexion';

export { delete } from 'chemin-module-suppression';

et l’import ne récupérera que ce dont il a besoin :

// 📁 authentification/connexion/formulaire.js

import { login } from '/projet/app/src/authentification';

1. Les imports dynamiques :

Actuellement nous ne faisons que des imports statiques.

Il est possible d’effectuer des imports dynamiques qui permet d’effectuer un import que sous certaines conditions.

On utilise pour cela le système d’import dynamique de JS qui utilise cette syntaxe :

import('chemin-module')

.then(obj => console.log(obj))

.catch(err => console.error(err))

Ce fonctionnement nécessite de bien maitriser l’asynchrone avec Javascript et particulièrement les promesses.