3IN017 - TECHNOLOGIES DU WEB

Introduction à JavaScript

28 janvier 2025

Gilles Chagnon

Plan

SCIENCES SORBONNE UNIVERSITÉ

Plantons le décor

Généralités

SCIENCES SORBONNE UNIVERSITÉ

1992 : HTML c'est bien mais...

- Traditionnel : client/serveur et pages statiques
- Lenteur de l'interaction avec l'utilisateur (une requête → une nouvelle page à charger du serveur !)
- Sous-utilisation du client (simple restitution du code fourni par le serveur)

Deux pistes de solutions :

- Modules externes : Flash, applets Java
  - \+ code compilé (machine virtuelle)
  - – pas d'interaction avec le HTML
  - – en perte de vitesse : Flash a disparu, Java presque sur le navigateur...
- JavaScript
  - – code non compilé (interprété)
  - \+ interaction avec le HTML
  - \+ standard vivace

Bref historique de JavaScript

SCIENCES SORBONNE UNIVERSITÉ

1996 JavaScript (standard ECMA-262)

2006 jQuery : pour faciliter l'interaction avec le HTML

2010 Node.js : JavaScript sort du navigateur

2010 Angular, 1er *framework* moderne (cadriciel)

2012 TypeScript : typage pour JS

2013 Electron : applications natives (UI) en JS

2013 React (Facebook)

2014 Vue.js

2019 WebAssembly (à suivre ?)

Pour suivre l'actualité et la mode des frameworks : `https://stateofjs.com/`

Une ressource incontournable, Mozilla Developper Network :
`https://developer.mozilla.org/fr/docs/Web/JavaScript`

Principe : côté client

Trois manières d'intégrer du code JavaScript dans un document HTML :

- dans un attribut HTML. Par exemple

```
<button onclick="console.log('Bouton cliqué')">Cliquez-moi!</button>
```

  En pratique, complique lisibilité et maintenance.

- dans un élément `script`, dans le `head` du fichier HTML ou juste avant sa fin, après le `body` :

```
<script>
// code JavaScript
</script>
```

  moins difficile à maintenir, mais encore problématique si le code doit être partagé entre plusieurs pages.

- dans un fichier externe

```
<head>
        <script src="cheminVersFichier/fichier.js">
        (...)
</head>
```

Principe : côté serveur

- Utilisation du moteur V8 (si, si...), moteur JavaScript de Chromium
- Création de **Node.js** en 2009
- Utilisation de JavaScript pour écrire des programmes interprétés fonctionnant *via* la ligne de commande
- Accès au système de fichiers
- Par l'ouverture de ports, implémentation de serveur

Nous y reviendrons...

Syntaxe : instructions de base

Instructions de base

Comme pour la plupart des langages :

- sensible à la casse (majuscules/minuscules)
- on retrouve les blocs
- on retrouve les mots-clés if, for, while, break, continue, switch
- pour les commentaires, `//` pour une seule ligne `/*(...)*/` pour plusieurs lignes

Par exemple...

```
// Boucle for
for (let i = 0 ; i<5 ; i++) {...}
// Boucle while
while (true) {
        if (keyboardPressed()) break;
        i++
}
```

Exceptions

Mécanismes d'exception comme en C++/Java/Python

... mais un seul `catch`

```
try {
        // génère une exception (n'importe quelle valeur fait l'affaire)
        throw "monException";
} catch(e){
// Traitement de l'erreur (propriétés e.name ou e.message)
}
```

Messages

Pas de notion de « sortie standard » ; JS permet d'afficher des messages *via* sa console :

```
console.log("Voici un message");
console.debug("Message de debug");
console.warn("Avertissement");
console.error("Message d'erreur");
```

Ces messages sont affichés :

- dans la console du navigateur (outils de développement, raccourci F12) quand JS est utilisé côté client
- dans la ligne de commande du serveur quand JS est utilisé côté serveur

Outils de développement des navigateurs

Ces outils de développement sont incontournables :

- la console affiche messages d'erreur et avertissements JavaScript.
- l'inspecteur de code permet de vérifier le code HTML interprété par le navigateur – mais mis en page ce qui peut induire parfois en erreur
- les outils réseaux donnent accès aux requêtes et réponses HTTP...

Structure : variables, objets, fonctions et classes

Types de données

Pas de typage fort des données en JS :

- `Boolean`
- `Number` : pas d'entier, flottant. Juste nombre...
- `String`
- `BigInt`
- `Symbol`
- `undefined`
- `null`

Et c'est tout !

Pour connaître le type d'une variable, *opérateur* `typeof` :
`console.log(typeof "Bonjour!")` renvoie String.

Le typage fort est ajouté par la bibliothèque Typescript.

Déclaration de variables

- variables constantes avec const (portée=bloc)
- variables avec let (portée=bloc)
- variables avec var (portée=fonction)
- sans mot-clé de déclaration : variable globale (portée=tout le script) mais à ne pas utiliser (surcharge de l'objet `window` dans un navigateur)

Fonctions – Déclaration

En JavaScript, les fonctions sont des objets comme les autres, de type Function. On peut donc les manipuler comme des variables.

- Fonctions simples :

```
function f (a, b) {
        return a+b
}
// Une fonction étant une variable comme une autre,
// elle peut être passée comme paramètre :
// on appelle cela un callback
function g (f, a, b) {
        return ( f(a,b) *3)
}
```

- Fonctions « fléchées » :

```
/* La fonction h réalise la même opération que f
quand on lui fournit deux nombres ou deux chaînes de caractères */
let h = (c,d) => c+d

// Avec décomposition
let h2 = ({c, d}) => c+d
```

Fonctions – Paramètres

- Valeur des paramètres par défaut :

```
function f (a, b=0){
// Si b n'est pas spécifié, ce paramètre vaudra 0
}
```

- Nombre variable de paramètres

```
function f (a, ...reste) {
(...)
}

f(1, 2, 3) // reste = [2,3]
f(4, 5) // reste = [5]
```

POO ? POP ?

SCIENCES SORBONNE UNIVERSITÉ

JavaScript est un langage « orienté prototype ». Pas de *vraie* classe mais on crée des objets (des prototypes) qui sont ensuite copiés :

```
1  objet1 = new Object;
2  objet1.propriete = "valeur";
3  objet1.methode   = function(){return this.propriete}
4
5  objet2 = Object.create(objet1);
6  console.log(objet2.propriete); // renvoie "valeur"
```

Ultérieurement, un « sucre syntaxique » a été défini pour faire ressembler la syntaxe JS aux classiques classes et instanciations.

Objets

Association d'identifiants et de valeurs :

```
let exempleObjet = {a: 1, b: 2, c: 3}
console.log(exempleObjet.a);

// Boucle sur les clefs:
for (let key in exempleObjet) {
        console.log(key, exempleObjet[key])
}
/* renvoie
 a 1
b 2
c 3 */
```

Objets et JSON

JSON très proche de la notation des objets en JS

```
// Renvoie { result: true, count: 42}
let o = JSON.parse('{"result": true, "count": 42}');

// Renvoie la chaîne '{"result": true, "count": 42}'
console.log(JSON.stringify(o));
```

Classes

La syntaxe de déclaration des classes est un sucre syntaxique, disponible depuis ECMAScript 2015.

```
class X {
      constructor(x) {
        this.x = x
      }
      static f() { /* membre statique */ }
      g() {
          console.log("X =", this.x)
      }
    }

X.f() // appel statique
let x = new X(1) // nouvelle instance
x.g() // Affiche X = 1
```

Classes et héritages

SCIENCES SORBONNE UNIVERSITÉ

On peut créer des sous-classes avec `extends` :

```
class Y extends X {
      g() {
        super.g()
        console.log("Y =", this.x)
      }
    }

let y = new Y(1) // nouvelle instance
y.g() // Affiche X = 1 \n Y = 1
```

Le Document Object Model (DOM)

DOM - Principe

Une manière structurée de concevoir un document balisé. Ce peut être un document XML comme du SVG, ou du HTML :

- un accès hiérarchique à l'arbre des éléments et attributs de la page (mais aussi commentaires, instructions de traitement XSLT, sections CDATA, *etc.*)
- un moyen organisé d'accéder aux événements déclenchés par l'utilisateur (mouvement de souris, click, touche du clavier, mouvement du smartphone...), et de gérer leurs conséquences

DOM : Accéder à l'arbre

SCIENCES SORBONNE UNIVERSITÉ

Attention

Ces méthodes ne sont utilisables que si le DOM a été entièrement chargé.

Par exemple, pour ce code...

```
<p id="id1">...</p>
<p class="classe2">...</p>
<p class="classe2"><span>...</span>...</p>
<p class="classe3">...</p>
```

Accès direct :

- par l'identifiant de l'élément (attribut id, unique dans le document) : `document.getElementById("id1")`
- par leur classe : `document.getElementsByClassName("classe2")`
- par leur nom d'élément : `document.getElementsByTagName("p")`, renvoie un tableau
- par une expression CSS :
  - `document.querySelectorAll("p[class]")` (pour tous les éléments ciblés), renvoie un tableau
  - `document.querySelector("p[class]")` (seulement le premier)

DOM : Accéder à l'arbre

SCIENCES SORBONNE UNIVERSITÉ

Accès indirect, à partir d'un nœud déjà identifié (par exemple, on a déjà exécuté `elt=document.getElementById("idpara1")` :

- elt.childNodes donne la liste de tous les nœuds-enfants de l'élément elt sous la forme d'un tableau.
- elt.children donne la liste de tous les éléments enfants de l'élément elt sous la forme d'un tableau.
- elt.firstChild est équivalent à elt.childNodes[0], et renvoie le premier nœud-enfant de l'élément elt.
- elt.firstElementChild est équivalent à elt.children[0], et renvoie le premier élément enfant de l'élément elt.
- elt.lastChild renvoie le dernier nœud-enfant de l'élément elt.
- elt.lastElementChild renvoie le dernier élément-enfant de l'élément elt.
- elt.parentNode renvoie le nœud parent de l'élément elt.

Remarque : on écrit ici « tableau » par abus de langage : il s'agit d'autres types d'objet qui en sont assez proches.

Nœuds enfants et éléments enfants

Attention à ce code HTML :

```
<ul id="liste">
        <li>...</li>
        <li>...</li>
</ul>
```

À cause des renvois à la ligne et de la mise en page avec des tabulations :

```
console.log(document.getElementById("liste").childNodes.length) // renvoie 5
console.log(document.getElementById("liste").children.length) // renvoie 2
```

Attention : les outils de développement restituent ce genre de code avec cette mise en page (renvois à la ligne et tabulations) quel que soit le code réel du document.

DOM : Création d'éléments

SCIENCES SORBONNE UNIVERSITÉ

```
newP = document.createElement("p") // crée un élément p
newText = document.createTextNode("du texte") // crée un nœud de type texte
```

Ces éléments et nœuds de type texte doivent ensuite être affectés à des éléments déjà existants :

```
elt = document.getElementsByTagName("header")[0];

newP.appendChild(newText);
elt.appendChild(newP);
```

Il existe d'autres méthodes (insertAdjacentHTML, insertBefore...).

Vous trouverez parfois dans des exemples de code la propriété `innerHTML`.
Cette propriété peut entraîner des problèmes d'accessibilité de la page et doit être évitée pour les modifications et créations de contenus.

DOM : suppression d'éléments

Deux méthodes :

- avec `elt.remove()` pour supprimer l'élément elt du DOM
- avec `eltParent.removeChild(eltEnfant)` pour supprimer eltEnfant des enfants d'eltParent. À la différence de `eltEnfant.remove()`, cette manière de faire déclenche une erreur si eltEnfant n'est pas un enfant d'eltParent.

DOM / Gestion des événements avec addEventListener

SCIENCES SORBONNE UNIVERSITÉ

On utilise des *callbacks*, c'est-à-dire des fonctions qui sont simplement appelées par leur nom, ou des fonctions anonymes, ou des fonctions fléchées :

```
let btn1 = document.getElementById("bouton1");
let btn2 = document.getElementById("bouton2");
let btn3 = document.getElementById("bouton3");

function fonctionGestionClick (evt) {
// le paramètre evt contient des informations sur l'événement:
// position du curseur, touche appuyée en même temps...
}

btn1.addEventListener("click", fonctionGestionClick, false)
// le dernier paramètre false est facultatif, c'est une longue histoire...
btn2.addEventListener("click", function(evt){
// gestion de l'événement
}, false)
btn3.addEventListener("click", (evt) => ..., false)
```

Attention

Il faut passer comme premier paramètre le nom de l'événement, *pas l'attribut HTML* associé. On a par exemple écrit ici « click » et pas « onclick ».

Comment attendre le chargement de la page

SCIENCES SORBONNE UNIVERSITÉ

On a déjà dit qu'avant d'utiliser des méthodes comme **getElementById** il fallait attendre que le DOM soit chargé. Il y a principalement trois manières de faire :

- placer les éléments `script` à la toute fin du code de la page, juste avant la balise fermante `</html>`
- ajouter l'attribut `defer` lors de l'appel au script :
  ```
  <script src="moncode.js" defer>
  ```
- utiliser l'événement `DOMContentLoaded` de l'objet window :
  ```
  window.addEventListener("DOMContentLoaded", fonctionALancer)
  ```

Programmation asynchrone

Introduction

SCIENCES SORBONNE UNIVERSITÉ

Problème

Dans le web, beaucoup d'événements asynchrones

- Actions des utilisateurs
- Accès des API Web

Solution

- Processus unique (monothread)
- Programmation asynchrone pour les opérations longues (I/O)

Programmation asynchrone : Boucle d'événements

JavaScript est mono-processus (monothread) :

- Chaque message (tâche = ensemble d'instructions) est exécuté totalement
- pseudo-multitâche I/O asynchrone

Pseudo-algorithme :

```
while (queue.attendreMessage()){
        // En vrai, il y a plusieurs files d'attentes
        // (I/O, micro et macro-tâches)
        queue.traiterProchainMessage();
}
```

Boucle d'événements : exemples

Appel d'une API : recherche d'un utilisateur
$\rightarrowtail$On attend la réponse

Affichage d'un choix pour l'utilisateur pendant un temps limité
$\rightarrowtail$On attend que l'utilisateur clique

Appel d'une API : Affichage de la carte une fois le choix effectué
$\rightarrowtail$On attend la réponse

Pendant l'attente I/O, d'autres parties du code peuvent s'exécuter

Programmation asynchrone : Callbacks

SCIENCES SORBONNE UNIVERSITÉ

```
// Quand est-ce que je m'affiche ?
let now = Date.now()
console.log("[start]", Date.now() - now, "ms")
setTimeout(() => { console.log(Date.now() - now, "ms")}, 1000)
setTimeout(() => { console.log(Date.now() - now, "ms")}, 1100)
setTimeout(() => { console.log(Date.now() - now, "ms")}, 1200)
console.log("[end]", Date.now() - now, "ms")
```

setTimeout est une fonction asynchrone : la fonction de callback sera placée dans la pile d'exécution dès l'expiration du délai, et exécutée dès que possible. Il existe une autre fonction liée au temps, setInterval (appel à intervalles réguliers)

Programmation asynchrone : Promises

Pour travailler de manière plus confortable en asynchrone, on utilise les *Promises* (promesses)

```
let promise = maFonctionAsynchrone()

promise.then((value) => {
        console.log("Got U", value)
}).catch((e) => {
        console.error("Caramba, encore raté : "+e.message)
}).finally(() => {
        console.log("Time to cleanup")
})
```

Nous reviendrons plus tard sur les promesses. C'en est une.

JS côté serveur et modularisation

Le problème...

Les bibliothèques, c'est pratique... mais pas prévu au départ dans JavaScript.

Pas de mécanisme d'inclusion

Répartition du même code à exécuter en plusieurs fichiers

- Conflit sur les variables
- Inclusions au niveau global (balise <script> en HTML)

Résultat...

```
<script src="library1.js"></script>
<script src="library2.js"></script>
<script src="moncode.js"></script>
```

- très difficile à utiliser (surtout quand on doit maintenir un site avec plusieurs documents HTML !)
- De moins en moins possible quand on augmente le nombre de bibliothèques utilisées

Les solutions

- Common JS (CJS) : inclusion dynamique `require("...")`, porté initialement par Node.js
- Asynchronous module definition (AMD) / RequireJS : adaptation pour les navigateurs... mais abandonné
- ES Modules (ESM) : inclusion *statique* ou *em dynamique* (standard de plus en plus répandu, maintenant adopté par Node.js)

Solutions très différentes

Incompatibilité (mais on peut utiliser un module CJS avec ESM)

On se focalisera ici sur ESM, le standard le plus moderne. On rencontre encore des appels à `require` dans certains codes Node.js donc il faut savoir que cela existe...

Pour utiliser un module ESM avec Node.js, il faut lui donner l'extension `.mjs`.

ESM : Exporter / Importer des symboles

La solution la plus simple (inspiré de la documentation de Node.js) :

```
// addTwo.mjs
const addTwo = (num) => num+2;

export { addTwo };
export CONSTANTE = 42;
```

et pour l'import...

```
// app.mjs
import { addTwo, CONSTANTE } from './addTwo.mjs';

// Prints: 48
console.log(addTwo(4)+CONSTANTE);
```

ESM : Exporter par défaut

SCIENCES SORBONNE UNIVERSITÉ

```
// @filename: util.mjs
export default (x, y) => x + y;
```

On peut alors le nommer comme on le souhaite lors de l'import...

```
// @filename: util.mjs
import bidule from 'util'
console.log(bidule(2, 3))
```

ESM : Import dynamique

- Fonctionne dans la plupart des navigateurs (`https://caniuse.com/es6-module-dynamic-import`)
- Fonctionne avec le mécanisme des promesses :

```
import("util").then(module => {
        console.log(module.sum(2,3))
})
```

Quelques mots sur npm

- un gestionnaire de paquets (Node Package Manager)
- une surcouche, yarn, existe mais est moins répandu

Pour créer un nouveau projet :

1. créer un nouveau répertoire "projet_node"
2. s'y placer (cd projet_node)
3. lancer la commande npm init, répondre aux questions
4. le fichier package.json est créé

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

npm : ajout de dépendances

SCIENCES SORBONNE UNIVERSITÉ

On peut maintenant ajouter des modules (ajoutés dans le répertoire node_modules) :

```
 # Ajoute une dépendance
npm add NOM_DU_MODULE
# ou pour un module de développement (modules)
npm add -d NOM_DU_MODULE
```

Un répertoire node_modules est créé avec un dossier par module ajouté ; on peut alors importer ces modules. L'installation d'un module peut aussi s'accompagner de l'installation de dépendances supplémentaires.

npm : modules standards

Node.js vient avec une bibliothèque standard (`https://nodejs.org/api/`). En particulier, on peut citer :

- un certain nombre de variables globales sont prédéfinies :
  - console
  - setTimeOut, setInterval...
- `fs` qui permet d'accéder au système de fichiers
- `path` qui permet de manipuler les chemins (fichiers)
- `http` et `url` pour pouvoir créer plus facilement un serveur et un routeur
- ...