FORMATION DEV WEB WEB MOBILE
CCP 2 - DÉVELOPPER LA PARTIE BACK-END D'UNE APPLICATION WEB

JAVASCRIPT

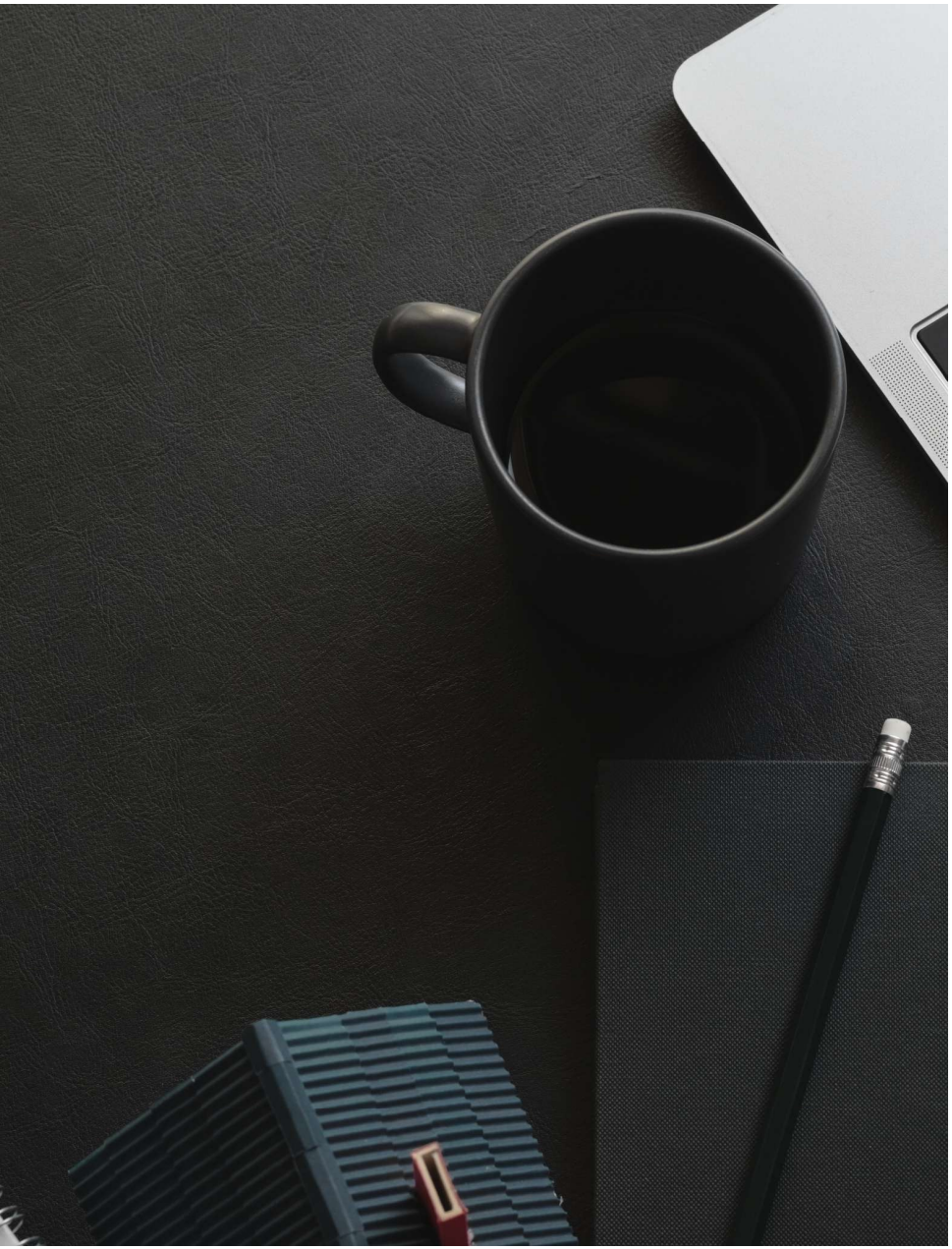
JOUR 2 - ASYNCHRONE ET JS AVANCE

PRESENTE PAR NATACHA DESSE

AFEC

DEROULE MODULE

- Les types et les fonctions
- **Asynchrone et JS avancé**



ASYNCHRONE ET JS AVANCE

LES OBJECTIFS

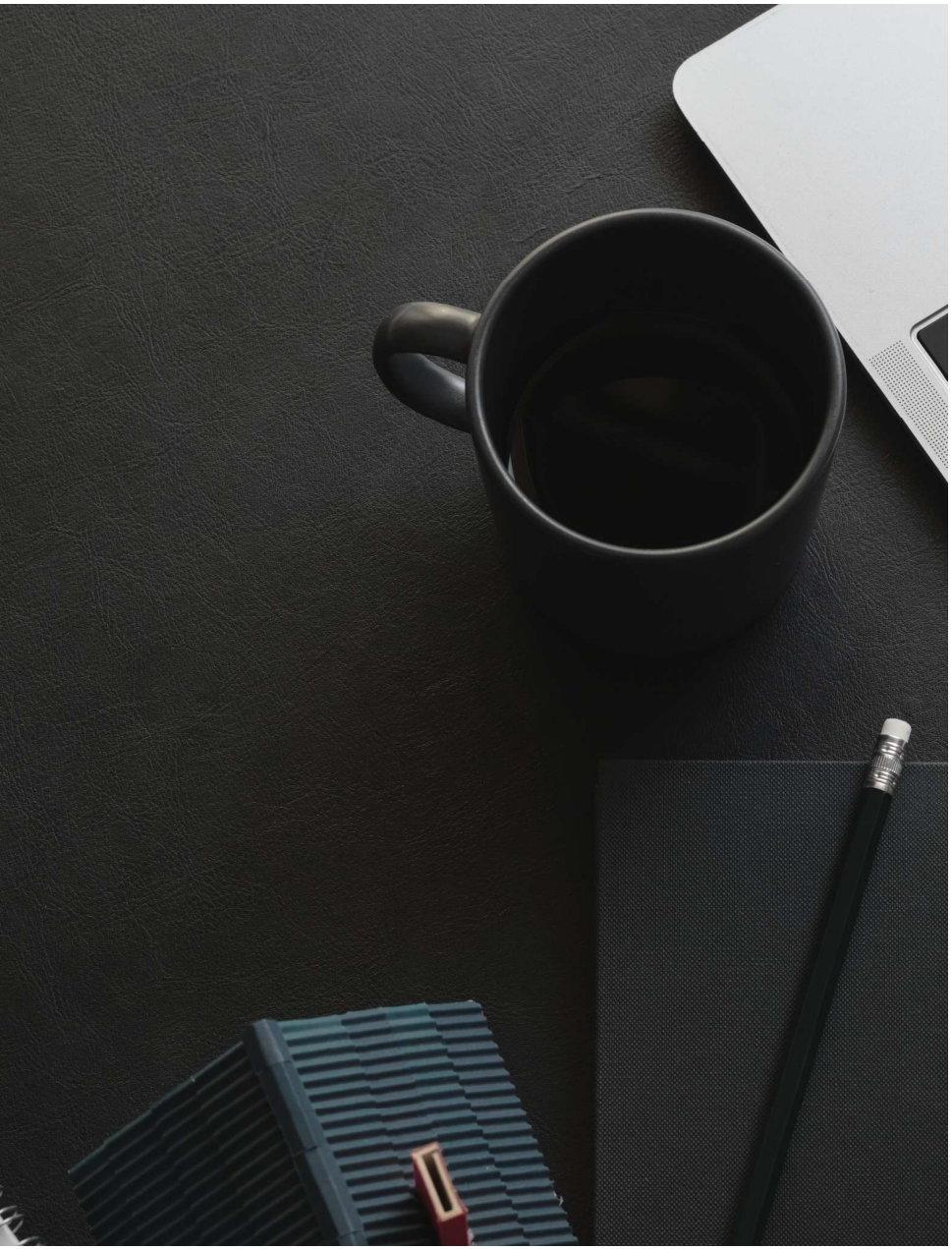
Approfondir la maîtrise des objets, des fonctions avancées.

Maîriser l’asynchrone en JS

S’approprier les outils modernes d'écriture (ES6+) : destructuration, modules, classes.

NUAGE DE MOTS

Quelles fonctions Javascript avez vous retenu ?



COPIE D’OBJETS

COPIE D'OBJETS

En JavaScript, les objets sont copiés par référence. Ça signifie que :

```
1 const a = { nom: "Alice" };
2 const b = a;
3
4 b.nom = "Bob";
5 console.log(a.nom); // "Bob" 😳
```

a et b pointent vers le même objet en mémoire.

SHALLOW COPIE

Spread operator

Elle duplique uniquement les propriétés du premier niveau.
Mais si l'objet contient un objet ou tableau imbriqué, la référence interne est partagée !

```
const original = {
  nom: "Alice",
  adresse: {
    ville: "Paris"
  }
};

const copie = { ...original };
copie.nom = "Bob";
copie.adresse.ville = "Lyon";

console.log(original.nom);         // "Alice"
console.log(original.adresse.ville); // "Lyon" 😬
```

Ici, copie.adresse est une référence vers le même objet adresse.

Object.assign()

Autre syntaxe mais même comportement que le spread.

```
const copie2 = Object.assign({}, original);
```

DEEP COPY

Elle duplique tout l'objet, y compris les objets imbriqués.

JSON.parse(JSON.stringify(...))

```
const original = {
  nom: "Alice",
  adresse: {
    ville: "Paris"
  }
};

const copieProfonde =
JSON.parse(JSON.stringify(original));
copieProfonde.adresse.ville = "Lyon";

console.log(original.adresse.ville); // "Paris"
```

Attention : cette méthode ne fonctionne pas avec :

- les méthodes (function),
- les dates (Date),
- les undefined,
- les types complexes (Map, Set, etc.).

structuredClone()

Fonction native moderne dans les navigateurs récents et Node.js :

```
const copie = structuredClone(original);
```

- Fonctionne mieux que JSON.parse, même avec des dates, tableaux, etc.
- Pas encore dispo dans tous les environnements (vérifie la compatibilité).

EXERCICES

Objets avancés

SHALLOW VS DEEP COPY

Voici une personne avec ses coordonnées

```
const person = {
  name: "Alice",
  age: 30,
  address: {
    city: "Paris",
    zip: "75001"
  }
};
```


```
const person = {
  name: "Alice",
  age: 30,
  address: {
    city: "Paris",
    zip: "75001"
  }
};
```


Enoncé :

1. Fais une **copie superficielle** (shallow copy) de person en utilisant Object.assign() ou le spread operator ({ ...person }).
2. Modifie le champ city de l'adresse dans la copie.
3. Observe si l'adresse de l'objet original a aussi été modifiée.
4. Ensuite, fais une **copie profonde** (deep copy) de person avec JSON.parse(JSON.stringify(...)).
5. Recommence la modification de l'adresse dans la copie profonde et observe le résultat

CORRECTION

```
const person = {
  name: "Alice",
  age: 30,
  address: {
    city: "Paris",
    zip: "75001"
  }
};

// Shallow copy
const shallowCopy = { ...person };
shallowCopy.address.city = "Lyon";

console.log("Original:", person.address.city);   // "Lyon" → modifié !
console.log("Shallow Copy:", shallowCopy.address.city);

// Deep copy
const deepCopy = JSON.parse(JSON.stringify(person));
deepCopy.address.city = "Marseille";

console.log("Original after deep copy:", person.address.city);   // Lyon
console.log("Deep Copy:", deepCopy.address.city);                // Marseille

```

COMPTAGE UTILISATEURS PAR ROLE

Tu disposes d'un tableau d'utilisateurs avec un champ role, par exemple :

```
const utilisateurs = [
  { id: 1, nom: "Alice", role: "admin" },
  { id: 2, nom: "Bob", role: "user" },
  { id: 3, nom: "Charlie", role: "user" },
  { id: 4, nom: "Dave", role: "moderator" },
  { id: 5, nom: "Eve", role: "admin" }
];
```

Crée une fonction compterParRole() qui retourne un objet indiquant le nombre d'utilisateurs par rôle.

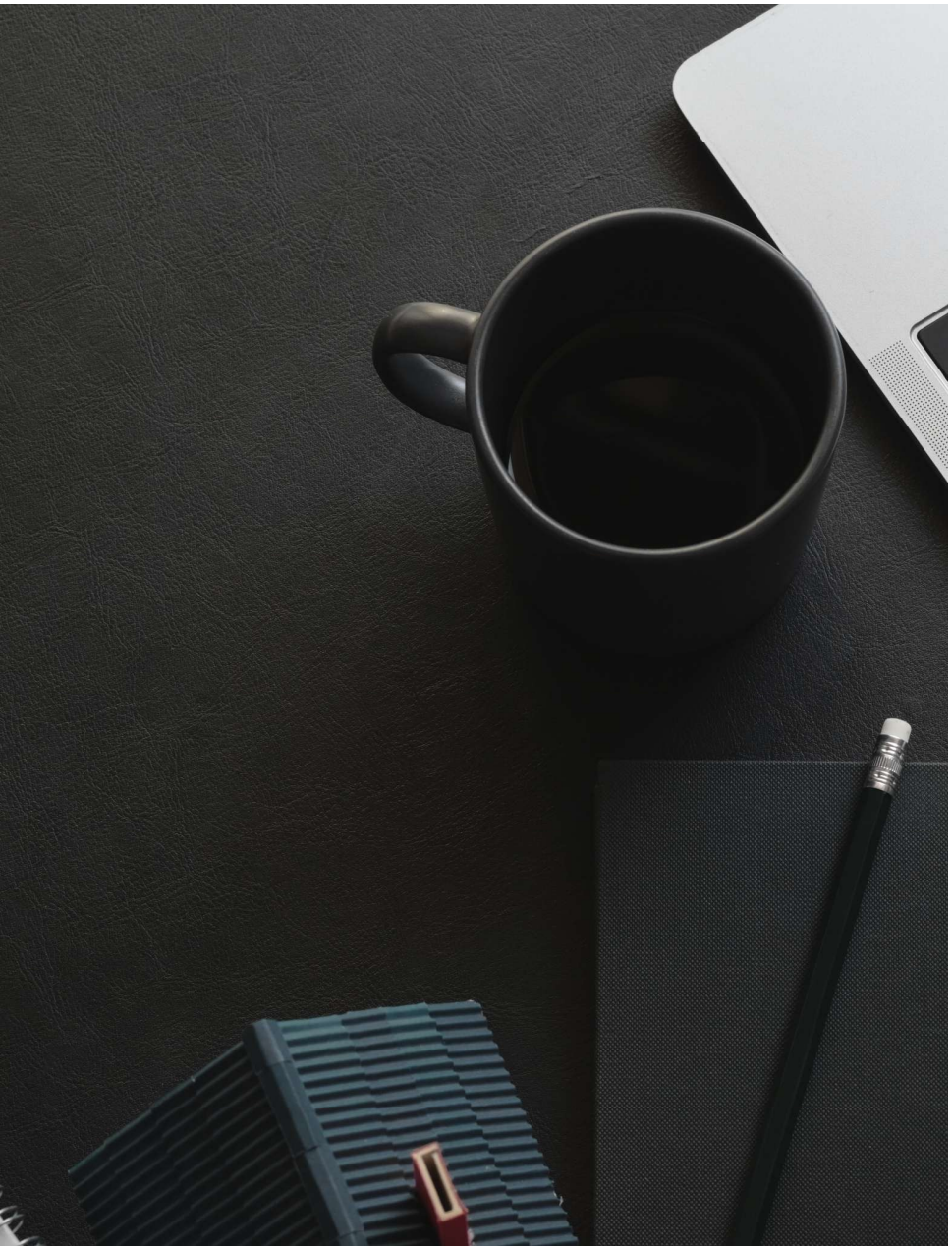
```
{
  admin: 2,
  user: 2,
  moderator: 1
}
```

Etapes :

- Utilise une boucle pour parcourir les utilisateurs.
- À chaque itération :
- Vérifie si le rôle existe déjà dans ton objet de comptage.
- Si oui, incrémente.
- Sinon, initialise à 1.

CORRECTION

```js
const utilisateurs = [
  { id: 1, nom: "Alice", role: "admin" },
  { id: 2, nom: "Bob", role: "user" },
  { id: 3, nom: "Charlie", role: "user" },
  { id: 4, nom: "Dave", role: "moderator" },
  { id: 5, nom: "Eve", role: "admin" }
];

function compterParRole(liste) {
  const compteur = {};

  for (const utilisateur of liste) {
    const role = utilisateur.role;

    // Si le rôle existe déjà, on incrémente
    if (compteur[role]) {
      compteur[role]++;
    } else {
      // Sinon, on initialise à 1
      compteur[role] = 1;
    }
  }

  return compteur;
}

// Test de la fonction
const resultats = compterParRole(utilisateurs);
console.log(resultats);

// Sortie attendue :
// { admin: 2, user: 2, moderator: 1 }

```



TABLEAUX ET MODULES

DESTRUCTURATION DE TABLEAUX

Déstructuration simple

```
const fruits = ["pomme", "banane", "kiwi"];

const [f1, f2] = fruits;
console.log(f1); // "pomme"
console.log(f2); // "banane"

```

Ramasser le reste

```
const [premier, ...restants] = ["a", "b", "c"];
console.log(restants); // ["b", "c"]
```

Ignorer des éléments

```
const [, , f3] = fruits;
console.log(f3); // "kiwi"
```

Marche aussi dans les fonctions

```
function somme(...nombres) {
  return nombres.reduce((acc, n) => acc + n, 0);
}
console.log(somme(1, 2, 3)); // 6
```

LES MODULES

Un « module » en programmation correspond à un bloc cohérent de code, c'est-à-dire à un bloc de code qui contient ses propres fonctionnalités fonctionnant ensemble et qui est séparé du reste du code. Généralement, un module possède son propre fichier. L'avantage principal des modules est une meilleure séparation qui résulte dans une meilleure maintenabilité et lisibilité du code.

En Js on distingue deux types de modules :

- Modules ES (modernes)
- CommonJS (Node.js)

MODULES ES - LES EXPORTS

les exports permettent de rendre des variables/focntions disponibles à l'extérieur

export nommé

```
1 // utils.js
2 export const PI = 3.14;
3
4 export function addition(a, b) {
5   return a + b;
6 }
7
```

export default

```
1 // calcul.js
2 export default function multiplication(a, b) {
3   return a * b;
4 }
5
```

On peut tout à fait combiner les export nommé et les export par défaut

MODULES ES - LES IMPORTS

les exports permettent de récupérer ce qui a été exporté

Import nommé

```
1 // main.js
2 import { PI, addition } from './utils.js';
3
4 console.log(addition(2, 3));
5
```

Import par défaut

```
1 // main.js
2 import multiplication from './calcul.js';
3
4 console.log(multiplication(4, 5));
```

Importer tout

```
1 import * as utils from './utils.js';
2
3 console.log(utils.addition(2, 3));
```

HTML ET LES MODULES

Dans un fichier html

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

L’ordre a une importance dans l’importation des fichiers

COMMONJS

Avant les modules ES, Node utilisait CommonJS :

```
1 // utils.js
2 module.exports = {
3   direBonjour: function(nom) {
4     return `Salut ${nom}`;
5   }
6 };
7
8 // main.js
9 const { direBonjour } = require('./utils');
10 console.log(direBonjour("Charlie"));
11
```

Aujourd'hui on a tendance à préférer les modules ES, mais on peut encore voir cette syntaxe écrite car elle fonctionne partout

EXERCICES

Tableaux et modules

CATALOGUE DE LIVRES

Objectif :

Créer une petite application en ligne de commande permettant de :

- Gérer un catalogue de livres (tableau d'objets)
- Utiliser des fonctions dans un module externe (catalogue.js)
- Manipuler les tableaux avec les méthodes modernes (filter, find, map, etc.)

Structure du projet

/exercice-catalogue/
├──── catalogue.js ← contient les fonctions
└──── main.js ← utilise les fonctions

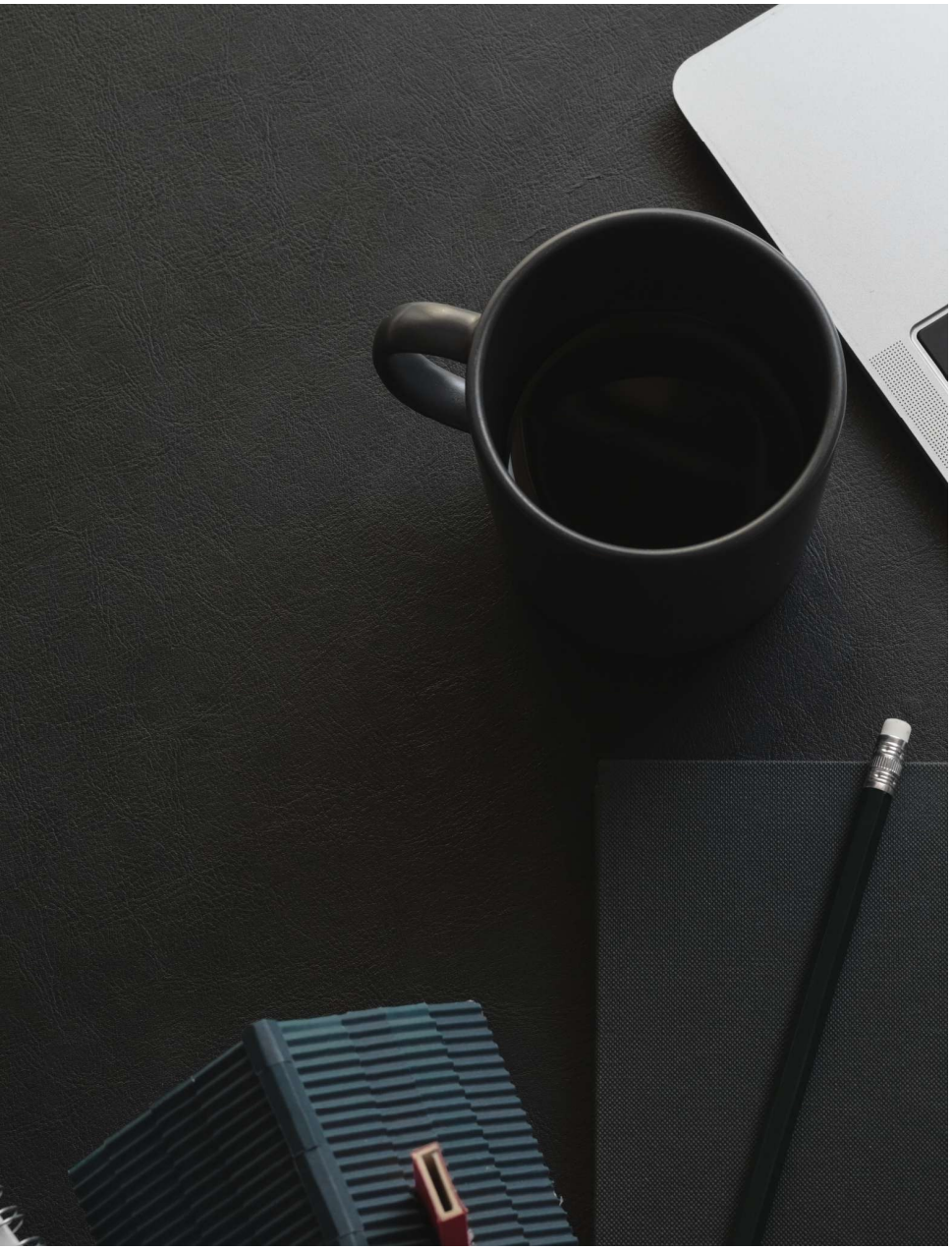
Etapes :

Implémenter les fonctions suivantes et les exporter :

1. listerLivres()
   Affiche tous les titres des livres
2. rechercherParAuteur(auteur)
   Retourne les livres écrits par un auteur donné
3; ajouterLivre(livre)
   Ajoute un livre au tableau catalogue
4. livresParAnnee(annee)
   Retourne les livres publiés à une année donnée
5; supprimerLivre(titre)
   Supprime un livre par son titre

Structure d'un livre :

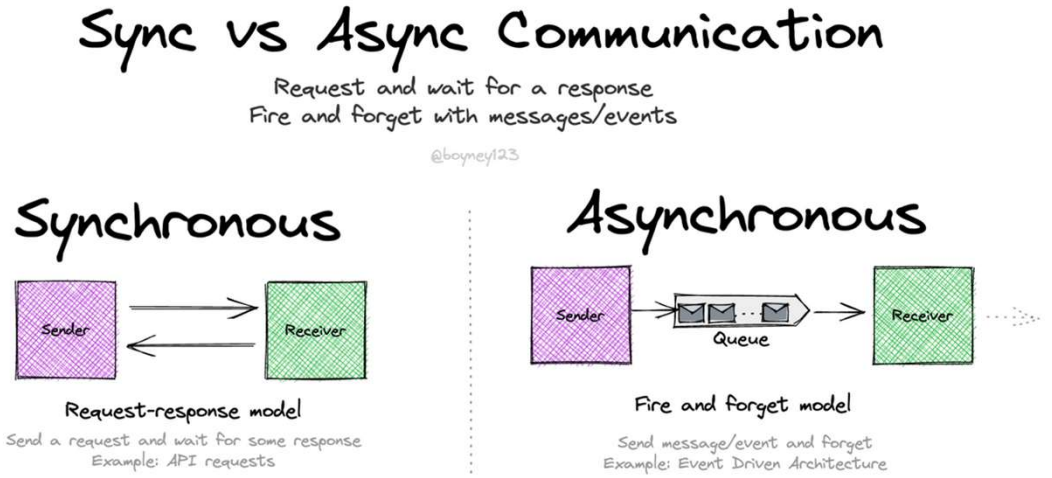
```
{
  titre: "Le Nom du vent",
  auteur: "Patrick Rothfuss",
  annee: 2007
}
```



ASYNCHRONISME ET PROMESSES

SYNCHRONE ET ASYNCHRONE

Dans la vie de tous les jours, on dit que deux actions sont synchrones lorsqu'elles se déroulent en même temps ou de manière synchronisée. Au contraire, deux opérations sont asynchrones si elles ne se déroulent pas en même temps ou ne sont pas synchronisées.



En informatique, on dit que deux opérations sont synchrones lorsque la seconde attend que la première ait fini son travail pour démarrer. Ce qu'il faut retenir de cette définition est le concept de dépendance (la notion de « synchronisation » dans la première définition donnée de synchrone au-dessus) : le début de l'opération suivante dépend de la complétude de l'opération précédente.

Au contraire, deux opérations sont qualifiées **d'asynchrones en informatique lorsqu'elles sont indépendantes** c'est-à-dire lorsque la **deuxième opération n'a pas besoin d'attendre que la première se termine** pour démarrer.

LES CALLBACK

En JavaScript, les opérations asynchrones sont placées dans des files d'attentes qui vont s'exécuter après que le fil d'exécution principal ou la tâche principale (le « main thread » en anglais) ait terminé ses opérations. Elles ne bloquent donc pas l'exécution du reste du code JavaScript.

Le premier outil utilisé en JavaScript pour générer du code asynchrone a été les fonctions de rappel. En effet, une fonction de rappel ou « callback » en anglais est une fonction qui va pouvoir être rappelée (« called back ») à un certain moment et / ou si certaines conditions sont réunies.

```
1 console.log("Début");
2
3 setTimeout(() => {
4   console.log("Traitement asynchrone terminé !");
5 }, 2000);
6
7 console.log("Fin");
8
```

CALLBACK HELL

Utiliser des callback pour générer du code asynchrone fonctionne mais possède certains défauts. Le principal défaut est qu'on ne peut pas prédire quand notre fonction de rappel asynchrone aura terminé son exécution, ce qui fait qu'on ne peut pas prévoir dans quel ordre les différentes fonctions vont s'exécuter.

En revanche, cela va être un vrai souci si la réalisation d'une **opération asynchrone dépend de la réalisation d'une autre opération asynchrone**.

Imaginons par exemple un code JavaScript qui se charge de télécharger une autre ressource relativement lourde. On va vouloir charger cette ressource de manière asynchrone pour ne pas bloquer le reste du script et pour ne pas que le navigateur « freeze ».

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

C'est le début du callback hell

SORTIR DU CALLBACK HELL : PROMISE

L'utilisation de fonctions de rappel pour effectuer des opérations asynchrones a pendant longtemps été la seule option en JavaScript.

En 2015, cependant, le JavaScript a intégré un nouvel outil dont l'unique but est la génération et la gestion du code asynchrone : les promesses avec l'objet constructeur Promise. C'est à ce jour l'outil le plus récent et le plus puissant fourni par le JavaScript nous permettant d'utiliser l'asynchrone dans nos scripts (avec la syntaxe async et await basée sur les promesses).

```
1 const promesse = new Promise((resolve, reject) =>
  {
2   setTimeout(() => {
3     resolve("Succès !");
4     // ou reject("Erreur !");
5   }, 1000);
6 });
```

**Une Promesse est un objet qui représente une opération asynchrone en attente,
et qui pourra être réussie (resolve) ou échouée (reject).**

En pratique, la majorité des opérations asynchrones qu'on va vouloir réaliser en JavaScript vont déjà être pré-codées et fournies par des API. Ainsi, nous allons rarement créer nos propres promesses mais plutôt utiliser les promesses renvoyées par les fonctions de ces API.

SORTIR DU CALLBACK HELL : PROMISE

Chaînage avec .then() et .catch()

```
promesse
  .then(resultat => {
    console.log("Réussi :", resultat);
  })
  .catch(erreur => {
    console.error("Échoué :", erreur);
  });

```

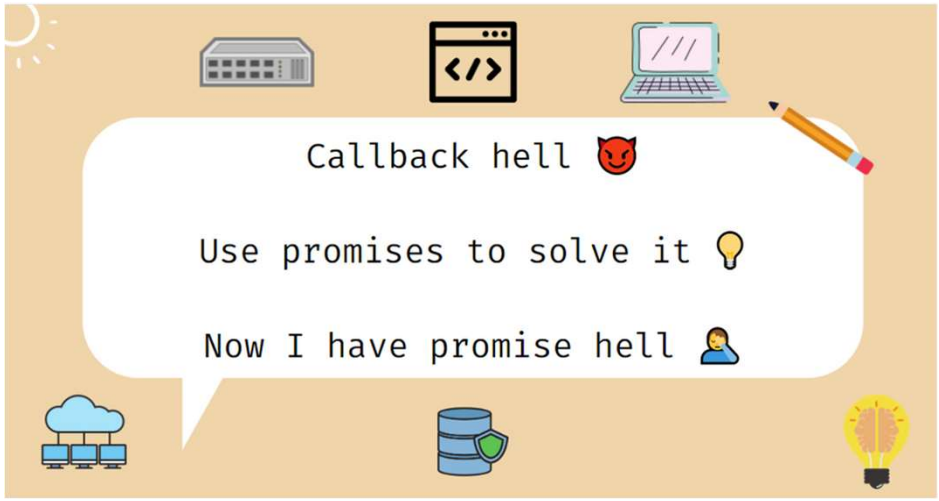
Lorsque notre promesse est créée, celle-ci possède deux propriétés internes : une première propriété state (état) dont la valeur va initialement être « pending » (en attente) et qui va pouvoir évoluer « fulfilled » (promesse tenue ou résolue) ou « rejected » (promesse rompue ou rejetée) et une deuxième propriété result qui va contenir la valeur de notre choix.

Si la promesse est tenue, la fonction resolve() sera appelée tandis que si la promesse est rompue la fonction reject() va être appelée. Ces deux fonctions sont des fonctions prédéfinies en JavaScript et nous n'avons donc pas besoin de les déclarer. Nous allons pouvoir passer un résultat en argument pour chacune d'entre elles. Cette valeur servira de valeur pour la propriété result de notre promesse.

POUR RERENTRER DANS LE PROMISE HELL

Le chainage peut se faire à l'infini

```
function operationAsync(nombre) {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve(nombre * 2);
    }, 1000);
  });
}

operationAsync(5)
  .then(res => {
    console.log("Résultat :", res);
    return operationAsync(res);
  })
  .then(final => {
    console.log("Résultat final :", final);
  });

```



STRUCTURE ASYNC/AWAIT

- async permet d'écrire des fonctions asynchrones de manière synchronisée visuellement.
- await attend la résolution d'une promesse.

La gestion des erreurs se fait désormais en try / catch

```
async function executer() {
  try {
    const resultat1 = await operationAsync(3);
    const resultat2 = await operationAsync(resultat1);
    console.log("Résultat final :", resultat2);
  } catch (erreur) {
    console.error("Erreur :", erreur);
  }
}

executer();
```

ATELIER

Simulation d’une API

CONSIGNES

Nous allons créer un gestionnaire de tâches mais cette fois ci les appels sont simulés comme des appels API asynchrones.

Suivez le guide

TP - Gestionnaire de Taches

📁 Structure du projet

```
/atelier-async/
├── api.js        # Fonctions asynchrones simulées (déjà fournies)
└── main.js       # À compléter par l'élève avec des appels à api.js
```

🚀 Lancer l'atelier

1. Ouvre le fichier `index.html` dans ton navigateur
2. Ouvre la console développeur (`F12` ou clic droit > Inspecter > Console)
3. Observe les résultats au fur et à mesure de ton développement

📦 api.js (fourni)

Ce fichier contient des **fonctions asynchrones simulant des appels API** :

```
export async function fetchTaches() { ... }

export async function ajouterTache(titre) { ... }

export async function marquerCommeFaite(id) { ... }
```

👉 Etapes à réaliser

Étape 1

QUIZZ