Javascript

Un mot sur node_modules

Voici quelques points clés sur l'utilité de **node_modules** :

- C'est un répertoire qui est créé lorsque vous utilisez npm pour installer des packages.
- Il contient les fichiers des packages que vous avez installés, ainsi que leurs dépendances.

- Les packages sont des morceaux de code réutilisables que vous pouvez inclure dans votre projet pour ajouter des fonctionnalités supplémentaires.
- Les dépendances sont des packages qui sont nécessaires pour que le package principal fonctionne correctement.
- Le répertoire **node_modules** est généralement exclu de la gestion de version de votre projet. ex : Github.
- Cela signifie que vous pouvez facilement partager votre projet avec d'autres développeurs sans inclure les fichiers volumineux **node_modules**.
- Les autres développeurs peuvent simplement exécuter la commande **npm install** pour installer les packages nécessaires à partir de la liste dans le fichier **package.json**.
- En résumé, le répertoire **node_modules** est un répertoire généré automatiquement qui contient les fichiers des packages et leurs dépendances installées à l'aide de npm.

Les différents types de variables

Nous pouvons inspecter le type d'une variable en utilisant l'opérateur **typeof**. Par exemple, pour savoir si une variable nommée **myVariable** est une chaîne de caractères, un nombre, ou un booléen, nous pouvons utiliser la syntaxe suivante :

```
console.log(typeof myVariable)
// renvoie "string", "number", ou "boolean" en fonction du type de la variable
```

Cela peut être utile pour déboguer votre code et vous assurer que les valeurs que vous manipulez sont du type attendu.

Variable	Explication	Exemple
Chaîne de caractères	Une suite de caractères connue sous le nom de chaîne. Pour indiquer que la valeur est une chaîne, il faut la placer entre guillemets.	`let myVariable = 'Bob';`
Nombre	Un nombre. Les nombres ne sont pas entre guillemets.	`let myVariable = 10;`
Booléen	Une valeur qui signifie vrai ou faux. `true` / `false` sont des	`let myVariable = true;`

	mots-clés spéciaux en JS, ils n'ont pas besoin de guillemets.	
Tableau	Une structure qui permet de stocker plusieurs valeurs dans une seule variable.	`let myVariable = [1,'Bob','Étienne',10];` Référez-vous à chaque élément du tableau ainsi `:` `myVariable[0]`, `myVariable[1]`, etc.
Objet	À la base de toute chose. Tout est un objet en JavaScript et peut être stocké dans une variable. Gardez cela en mémoire tout au long de ce cours.	`let myVariable = document.querySelector('h1');` tous les exemples au dessus sont aussi des objets.

Les opérateurs

Opérateur	Explication	Symbole(s)	Exemple
Addition	Utilisé pour ajouter deux nombres ou concaténer (accoler) deux chaînes.	`+`	`6 + 9; "Bonjour " + "monde !";`
Soustraction, multiplication, division	Les opérations mathématiques de base.	`-`, `*`, `/`	`9 - 3; 8 * 2; // pour multiplier, on utilise un astérisque9 / 3;`
Assignation	On a déjà vu cet opérateur : il affecte une valeur à une variable.	`=`	`let myVariable = 'Bob';`
Égalité	Teste si deux valeurs sont égales et renvoie un booléen `true`/`false` comme résultat.	`===`	`let myVariable = 3; myVariable === 4;`
Négation, N'égale pas	Renvoie la valeur logique opposée à ce qu'il précède ; il change `true` en `false`, etc. Utilisé avec l'opérateur d'égalité, l'opérateur de négation teste que deux valeurs *ne sont pas* égales.	`!`, `!==`	L'expression de base est vraie, mais la comparaison renvoie `false` parce que nous la nions : `let myVariable = 3; !(myVariable === 3);` On teste ici que "`myVariable` n'est PAS égale à 3". Cela renvoie `false`, car elle est égale à

			3. `let myVariable = 3; myVariable !== 3;`

Attention ci-dessous, nous essayons d'ajouter une chaîne de caractères ("Hello") à un nombre (5), ce qui est également impossible. La variable `result` sera donc NaN.

```
let x = "Hello";
let y = 5;
let result = x + y;

if (isNaN(result)) {
  console.log("Le résultat n'est pas un nombre.");
} else {
  console.log("Le résultat est un nombre.");
}
```

Les fonctions

```
const paragraph = document.body.querySelector("p");

function showText(domNode, string) {
  return domNode.innerHTML = string
}

showText(paragraph, "Hello World !");
```

Avec les fonctions fléchées :

```
const paragraph = document.body.querySelector("p");

const showText = (domNode, string) => {
  return domNode.innerHTML = string
}

showText(paragraph, "Je suis le résultat d'une fonction fléchée !");
```

Le nom de la fonction est déclaré.
Entre parenthèse viennent les paramètres de la fonction.
Enfin la flèche amène la logique de la fonction.

Vous tomberez certainement sur plusieurs variantes de fonctions fléchées, par exemple :

```
const showText = string => paragraph.innerHTML = string;

showText("Show me some text")
```

Ici pas de parenthèses nécessaires du fait qu'il n'y ai qu'un paramètre.
N'ayant qu'un return dans cette fonction, pas besoin d'accolade non plus, le return est écrit sur la même ligne.

> Fonctions fléchées - JavaScript | MDN (mozilla.org)

Les instructions *let* et *const*

Les instructions *let* et *const* sont des moyens de déclarer des variables en JavaScript. La principale différence entre *let* et *const* est que *const* crée une variable en lecture seule. Cela signifie qu'une fois que vous avez affecté une valeur à une variable constante, vous ne pouvez plus la modifier.

Pour déclarer une variable avec *let*, utilisez simplement le mot-clé *let* suivi du nom de la variable et, éventuellement, d'une valeur initiale. Pour déclarer une variable avec *const*, utilisez le mot-clé *const* suivi du nom de la variable et d'une valeur initiale obligatoire.

Voici un exemple :

```
let x = 10;
const y = 20;
```

Dans cet exemple, *x* est une variable déclarée avec *let* et *y* est une variable déclarée avec *const*.

Les déclarations de variables en JavaScript permettent de créer une zone mémoire pour stocker une valeur. Il existe plusieurs façons de déclarer des variables en JavaScript, notamment :

- `var` : cette instruction permet de déclarer une variable, qui peut être utilisée dans tout le script.
- `let` : similaire à `var`, mais la portée de la variable est limitée à son bloc d'instructions.

- `const` : similaire à `let`, mais la valeur de la variable ne peut pas être modifiée après sa déclaration.

Il est important de noter que lors de la déclaration d'une variable, il est possible de lui affecter une valeur immédiatement, ou de le faire plus tard dans le script.

```
var nom;
nom = "John";

let chien, array;
chien = "dalmatien";
array = [1, 2, 3];
```

```
const constante = 43;
constante = "String" // génèrera une erreur
```

Cela ne signifie pas que la valeur référencée ne peut pas être modifiée ! Ainsi, si le contenu de la constante est un objet, l'objet lui-même pourra toujours être modifié.

> const - JavaScript | MDN (mozilla.org)
> let - JavaScript | MDN (mozilla.org)

Les blocs

En JavaScript, un bloc est une section de code délimitée par une paire d'accolades ({ }). Les blocs sont utilisés pour regrouper des instructions et les exécuter ensemble dans une certaine séquence.

Les blocs peuvent contenir des déclarations de variables, des instructions conditionnelles, des boucles, des fonctions, des expressions, etc. Les blocs peuvent être imbriqués, c'est-à-dire qu'un bloc peut contenir d'autres blocs à l'intérieur.

Voici un exemple simple de bloc en JavaScript :

```
{
  let x = 10;
  console.log(x);
  if (x > 5) {
    console.log("x est supérieur à 5");
  }
}
```

Dans cet exemple, le bloc commence par l'accolade ouvrante ({) et se termine par l'accolade fermante (}). Le bloc contient une déclaration de variable "x", une instruction console.log pour afficher la valeur de "x", et une instruction conditionnelle "if" qui vérifie si "x" est supérieur à 5 et affiche un message si c'est le cas.

Il est important de noter que les variables déclarées à l'intérieur d'un bloc ne sont pas accessibles en dehors de ce bloc, car elles sont considérées comme locales. Cela signifie que si vous déclarez une variable dans un bloc, elle ne sera visible que dans ce bloc et dans les blocs imbriqués à l'intérieur.

```
{
  function maFonction() {
    console.log("Bonjour");
  }

  const abba = () => console.log('money money money')

  maFonction(); // "Bonjour"

  abba(); // money money money
}

abba(); // Erreur: abba n'est pas défini

maFonction(); // "Bonjour"

{
  let maFonction = function() {
    console.log("Bonjour");
  }

  const abba = () => console.log('take it now or leave it');
}

abba(); // Erreur: abba n'est pas défini

maFonction(); // "Bonjour"
```

La portée des variables

let et *const* permettent de déclarer une variable dont la portée est celle du bloc où elle est déclarée.

> ℹ️ La portée d’une variable désigne l’espace dans laquelle cette variable sera disponible.

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // c'est la même variable !
    console.log(x);  // 2
  }
  console.log(x);  // 2
}

function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // c'est une variable différente
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

```
{
  var a = 12;

  let b = 13;

  const c = 14;
}

const abc = () => {
    console.log(a)
    console.log(b) // b is not defined
    console.log(c) // c is not defined
}
abc()
```

Les boucles

Les boucles sont des structures de contrôle qui permettent de répéter une série d'instructions plusieurs fois. En JavaScript, il existe plusieurs types de boucles, notamment :

- La boucle `for` : Cette boucle permet de répéter un bloc d'instructions un nombre déterminé de fois. Elle est souvent utilisée pour parcourir des tableaux ou des listes. La syntaxe de la boucle `for` est la suivante :

```
for (initialisation; condition; incrémentation) {
  // instructions à répéter
}
```

L'initialisation est une instruction qui est exécutée une seule fois au début de la boucle. Elle est généralement utilisée pour déclarer et initialiser des variables.

La condition est une expression booléenne qui est évaluée avant chaque itération de la boucle. Si la condition est vraie, le bloc d'instructions est exécuté ; sinon, la boucle s'arrête.

L'incrémentation est une instruction qui est exécutée à la fin de chaque itération de la boucle. Elle est généralement utilisée pour mettre à jour une variable de comptage.

Voici un exemple de boucle `for` qui affiche les nombres de 1 à 10 :

```
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

- La boucle `while` : Cette boucle permet de répéter un bloc d'instructions tant qu'une condition est vraie. Elle est souvent utilisée lorsque le nombre d'itérations n'est pas connu à l'avance. La syntaxe de la boucle `while` est la suivante :

```
while (condition) {
  // instructions à répéter
}
```

La condition est une expression booléenne qui est évaluée avant chaque itération de la boucle. Si la condition est vraie, le bloc d'instructions est exécuté ; sinon, la boucle s'arrête.

Voici un exemple de boucle `while` qui affiche les nombres de 1 à 10 :

```
let i = 1;
while (i <= 10) {
  console.log(i);
  i++;
}
```

- La boucle `do...while` : Cette boucle est similaire à la boucle `while`, mais elle garantit que le bloc d'instructions est exécuté au moins une fois, même si la condition est fausse dès le début. La syntaxe de la boucle `do...while` est la suivante :

```
do {
  // instructions à répéter
} while (condition);
```

Le bloc d'instructions est exécuté une première fois, puis la condition est évaluée. Si la condition est vraie, le bloc d'instructions est répété ; sinon, la boucle s'arrête.

Voici un exemple de boucle `do...while` qui affiche les nombres de 1 à 10 :

```
let i = 1;
do {
  console.log(i);
  i++;
} while (i <= 10);
```

Il est important de noter que toutes les boucles peuvent entraîner une boucle infinie si la condition n'est jamais fausse. Pour éviter cela, il est recommandé de tester les boucles avec des valeurs limites et de s'assurer que la condition s'arrête à un moment donné.

Exercices

1. Créez une variable `nom` contenant votre nom. Ensuite, créez une fonction `saluer` qui affiche une chaîne de caractères personnalisée pour vous saluer. La chaîne de caractères doit inclure votre nom. Par exemple, si votre nom est "Jean", la fonction doit afficher "Bonjour Jean !". Si la fonction est appelé sans paramètre ou une chaîne de caractère vide, alors une erreur avec un message personnalisé appraraît. Testez votre fonction en l'appelant.
2. Créez une variable `age` contenant votre âge. Ensuite, créez une fonction `doubler` qui double la valeur de l'âge et renvoie la nouvelle valeur. Testez votre fonction en l'appelant.
3. Créez une variable `tableau` contenant un tableau de nombres. Ensuite, créez une fonction `somme` qui calcule la somme de tous les nombres dans le tableau et la renvoie. Testez votre fonction en l'appelant.
4. Créez une variable `temperature` contenant une température en Celsius. Ensuite, créez une fonction `conversion` qui convertit cette température en Fahrenheit et la renvoie. La formule pour la conversion est la suivante : Fahrenheit = Celsius * 1.8 + 32. Testez votre fonction en l'appelant.

5. Créez une variable `mot` contenant une chaîne de caractères. Ensuite, créez une fonction `compter` qui compte le nombre de caractères dans le mot et le renvoie. Testez votre fonction en l'appelant.

Le DOM

Le DOM (Document Object Model) est une représentation en mémoire d'une page web. Il permet de manipuler les éléments d'une page web à travers une interface objet. Chaque élément de la page est représenté par un objet dans le DOM, et ces objets peuvent être manipulés en utilisant JavaScript.

Le DOM est organisé sous forme d'arbre, avec le nœud racine représentant le document HTML. Les nœuds enfants de ce nœud racine représentent les balises HTML, les attributs, les commentaires et les autres éléments de la page.

En utilisant le DOM, vous pouvez ajouter, supprimer et modifier des éléments de la page web, ainsi que réagir aux événements utilisateur (clics de souris, appuis sur des touches, etc.).

Par exemple, pour sélectionner un élément de la page à l'aide de JavaScript, vous pouvez utiliser la méthode `document.querySelector()`. Cette méthode prend en argument un sélecteur CSS et renvoie le premier élément correspondant à ce sélecteur.

```
let element = document.querySelector('#monElement');
element.style.color = 'red';
```

Dans cet exemple, nous sélectionnons l'élément ayant l'ID "monElement" et nous modifions sa couleur en rouge.

En utilisant le DOM, vous pouvez sélectionner un élément par sa classe en utilisant la méthode `document.getElementsByClassName()`. Cette méthode renvoie un tableau d'éléments qui ont la classe spécifiée.

```
let elements = document.getElementsByClassName('maClasse');
for (let i = 0; i < elements.length; i++) {
  elements[i].style.color = 'red';
}
```

Dans cet exemple, nous sélectionnons tous les éléments ayant la classe "maClasse" et nous modifions leur couleur en rouge.

Il y a beaucoup plus à apprendre sur le DOM en JavaScript, mais cela devrait vous donner une idée de ce qu'il est et de ce qu'il peut faire.

En utilisant le DOM, vous pouvez également ajouter des éléments à la page, supprimer des éléments, modifier des éléments existants et plus encore.

Par exemple, vous pouvez utiliser la méthode `document.createElement()` pour créer un nouvel élément HTML, la méthode `element.appendChild()` pour ajouter cet élément comme enfant d'un autre élément, et la méthode `element.removeChild()` pour supprimer un élément.

Exercices

6. Créez un nouveau paragraphe et ajoutez le à la fin du corps de la page.
7. Sélectionnez tous les éléments de la page qui ont la classe "maClasse" et modifiez leur couleur en rouge.

Les objets

En JavaScript, tout est un objet. Un objet est une collection de propriétés, qui sont des paires clé-valeur. Les propriétés peuvent être des fonctions, des tableaux, des chaînes de caractères, des nombres, etc.

Voici un exemple d'objet JavaScript :

```
let person = {
  name: 'John',
  age: 30,
  city: 'New York'
};
```

Dans cet exemple, l'objet `person` définit les propriétés `name`, `age` et `city`.

Les objets peuvent avoir des méthodes, qui sont des fonctions qui sont associées à l'objet. Les méthodes peuvent être appelées sur l'objet pour effectuer une action.

Voici un exemple d'objet JavaScript avec une méthode :

```
let person = {
  name: 'John',
  age: 30,
  city: 'New York',
  sayHello: function() {
    console.log('Hello, my name is ' + this.name);
```

```
  }
};

person.sayHello(); // affiche "Hello, my name is John"
```

Dans cet exemple, l'objet `person` a une méthode `sayHello` qui affiche une chaîne de caractères avec le nom de la personne.

Il est également possible de créer des objets imbriqués, c'est-à-dire des objets qui ont des propriétés qui sont des objets eux-mêmes.

Voici un exemple d'objet JavaScript imbriqué :

```
let person = {
  name: 'John',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'New York',
    state: 'NY'
  }
};
```

Dans cet exemple, l'objet `person` a une propriété `address` qui est elle-même un objet avec les propriétés `street`, `city` et `state`.

Exercices

11. Écrire une fonction qui prend en paramètre un objet et qui affiche toutes ses propriétés et leurs valeurs dans la console.

Les évènements

ℹ️ En JavaScript, une fonction de rappel (ou callback en anglais) est une fonction qui est passée en tant qu'argument à une autre fonction et qui est appelée lorsque cette fonction a terminé son travail. Les fonctions de rappel sont souvent utilisées pour effectuer des tâches asynchrones, telles que l'envoi de requêtes à un serveur ou la manipulation de fichiers.

Par exemple, la méthode `setTimeout()` en JavaScript prend en argument une fonction de rappel à exécuter après un certain délai. Voici un exemple :

```
function maFonction() {
  console.log("La fonction a été exécutée !");
}

setTimeout(maFonction, 3000); // appelle maFonction après 3 secondes
```

Dans cet exemple, la fonction `maFonction()` est passée en tant qu'argument à la méthode `setTimeout()` et est appelée après un délai de 3 secondes.

Les événements en JavaScript sont des actions ou des occurrences qui se produisent dans une page web. Les exemples courants d'événements incluent les clics de souris, les mouvements de souris, les frappes de clavier, les chargements de page, les soumissions de formulaires, et bien plus encore. Les événements peuvent être utilisés pour interagir avec les utilisateurs, pour effectuer des tâches en arrière-plan, ou pour mettre à jour dynamiquement le contenu de la page.

Pour gérer les événements en JavaScript, on utilise des écouteurs d'événements. Un écouteur d'événements est une fonction qui est appelée chaque fois qu'un événement se produit. Pour ajouter un écouteur d'événements à un élément de la page, vous pouvez utiliser la méthode `addEventListener()`. Cette méthode prend en paramètres le type d'événement à écouter (par exemple, "click", "mousemove", "keydown"), ainsi que la fonction à appeler lorsque l'événement se produit.

Voici un exemple d'ajout d'un écouteur d'événements de clic à un bouton :

```
let bouton = document.querySelector('button');
bouton.addEventListener('click', function() {
  alert('Vous avez cliqué sur le bouton !');
});
```

Dans cet exemple, nous sélectionnons le bouton en utilisant la méthode `document.querySelector()` et nous ajoutons un écouteur d'événements de clic en utilisant la méthode `addEventListener()`. Lorsque le bouton est cliqué, la fonction d'alerte est appelée pour afficher un message.

```
let formulaire = document.querySelector('form');
formulaire.addEventListener('submit', function(event) {
  event.preventDefault();
  // effectuez ici le traitement du formulaire
});
```

Dans cet exemple, nous sélectionnons le formulaire en utilisant la méthode `document.querySelector()` et nous ajoutons un écouteur d'événements de soumission en utilisant la méthode `addEventListener()`. Lorsque le formulaire est soumis, la fonction est appelée. Cette fonction appelle ensuite la méthode `preventDefault()` pour empêcher le comportement par défaut de l'événement de soumission, puis effectue le traitement du formulaire.

La méthode `preventDefault()` est une méthode de l'objet `event` en JavaScript qui permet d'annuler le comportement par défaut d'un événement.

Dans l'exemple que j'ai donné précédemment, elle est utilisée pour empêcher le formulaire de se soumettre normalement. En effet, lorsqu'un formulaire est soumis, il provoque généralement un rechargement de la page ou un changement d'URL.

ℹ️ Il est important de noter que `preventDefault()` ne supprime pas l'événement.
Cela signifie que l'événement est toujours présent et peut être utilisé pour effectuer d'autres actions si nécessaire. Par exemple, nous pourrions utiliser l'objet `event` pour valider les données soumises dans le formulaire avant de les envoyer au serveur.
En résumé, `preventDefault()` est une méthode utile pour empêcher le comportement par défaut des événements en JavaScript. Elle peut être utilisée pour personnaliser le comportement des événements en fonction des besoins de l'application. Cependant, il est important de ne pas abuser de cette méthode, car elle peut rendre l'expérience utilisateur confuse si elle est utilisée de manière excessive.

En résumé, les événements en JavaScript ouvrent de nombreuses possibilités pour interagir avec les utilisateurs et pour créer des pages web dynamiques et interactives. La compréhension de leur fonctionnement et de leur utilisation est essentielle pour tout développeur web.

Exercices

9. Sélectionnez tous les liens de la page et ajoutez leur un événement de survol qui change leur couleur en bleu.
10. Créez un formulaire d'inscription qui demande le nom, l'adresse e-mail et le mot de passe de l'utilisateur. Ajoutez un événement de soumission au formulaire qui vérifie que tous les champs sont remplis et que l'adresse e-mail est valide. Si toutes les conditions sont remplies, affichez un message de bienvenue personnalisé avec le nom de l'utilisateur. Si les conditions ne sont pas remplies, affichez un message d'erreur correspondant à la condition manquante.

Le JSON

JSON (JavaScript Object Notation) est un format de données qui permet de stocker et de transmettre des données structurées de manière lisible par l'être humain. Il est basé sur un sous-ensemble de la syntaxe JavaScript, ce qui en fait un choix populaire pour l'échange de données entre des systèmes informatiques.

Les données JSON sont des objets qui peuvent contenir des propriétés (ou des clés) et des valeurs. Les propriétés sont des chaînes de caractères qui décrivent les données, tandis que les valeurs peuvent être des nombres, des chaînes de caractères, des booléens, des tableaux, des objets JSON, ou même des valeurs null.

Voici un exemple de données JSON qui représente une personne :

```
{
  "nom": "John",
  "age": 25,
  "ville": "New York",
  "interets": ["lecture", "sport"]
}
```

Il est possible de convertir des objets JavaScript en JSON et inversement en utilisant les méthodes `JSON.stringify(obj)` pour convertir un objet en chaine de caractère JSON, et `JSON.parse(jsonString)` pour convertir une chaine de caractère JSON en objet javascript.

Exercices

```
let jsonString = `[  { "nom": "John", "age": 30, "ville": "New York" },  { "nom": "Mar
y", "age": 25, "ville": "Los Angeles" },  { "nom": "Bob", "age": 23, "ville": "Chicag
o" }]`;
```

11. À partir de la string JSON ci-dessus, afficher les prénoms des personnes ayant 25 ans et plus.
2. À partir de la même string JSON, retournez un tableau d'objets ne contenant que les personnes ayant la lettre "o" dans leurs prénoms.
3. À partir de la même string JSON, créez un nouvel objet qui contient le nom de chaque personne comme clé et la ville dans laquelle elle vit comme valeur.

Les APIs en javascript

En JavaScript, une API (Application Programming Interface) est un ensemble de fonctions, méthodes et propriétés qui permet à un programme de communiquer avec un autre service ou système. Les API peuvent être intégrées à des bibliothèques ou des frameworks, ou peuvent être fournies par des services externes tels que des bases de données en ligne, des services de stockage de fichiers, des services de géolocalisation, etc.

Il existe deux types d'API en JavaScript :

- Les API natives : Ce sont les fonctionnalités intégrées à JavaScript qui sont prises en charge par les navigateurs web. Ces fonctionnalités incluent les objets de base tels que les objets `String`, `Array`, `Object`, etc., ainsi que les API de navigateur telles que `XMLHttpRequest`, `WebSocket`, `Geolocation`, etc.
- Les API tiers : Ce sont les fonctionnalités qui sont fournies par des services externes et qui peuvent être utilisées dans JavaScript en utilisant des bibliothèques ou des modules. Ces fonctionnalités peuvent inclure des fonctionnalités de mappage, de stockage de données en ligne, de recherche d'images, etc.

Pour utiliser une API, vous devez généralement inclure une bibliothèque ou un module dans votre code et utiliser les fonctions, méthodes et propriétés qu'il fournit. Certaines API nécessitent également une authentification ou une clé API pour accéder aux données.

L’API Fetch

`fetch()` est une API JavaScript standard qui permet de récupérer des données à partir d'une URL. Il est généralement utilisé pour récupérer des données à partir d'un serveur web via une requête HTTP (comme GET ou POST). Il est basé sur la spécification Promesses et retourne une promesse qui est résolue avec les données récupérées ou rejetée avec une erreur.

Voici un exemple d'utilisation de `fetch()` pour récupérer des données à partir d'une URL :

```
fetch('https://api.example.com/data')
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

Dans cet exemple, `fetch()` est utilisé pour envoyer une requête GET à l'URL `https://api.example.com/data`. La méthode `then()` est utilisée pour traiter la réponse obtenue, qui est convertie en un objet JSON en utilisant la méthode `json()`. Les données obtenues sont ensuite affichées dans la console. En cas d'erreur, la méthode `catch()` est utilisée pour gérer l'erreur et l'afficher dans la console.

Blocs then/catch/finally

Les blocs `then`, `catch` et `finally` sont utilisés pour gérer les promesses en JavaScript. Les blocs `then` sont utilisés pour traiter les promesses résolues, les blocs `catch` sont utilisés pour traiter les promesses rejetées, et les blocs `finally` sont utilisés pour exécuter du code indépendamment du résultat de la promesse.

Voici un exemple d'utilisation des blocs `then`, `catch` et `finally` avec une promesse :

```
function effectuerUneTache() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      const resultat = Math.random();
      if (resultat < 0.5) {
        reject('La tâche a échoué');
      } else {
```

```
        resolve('La tâche est terminée avec succès');
      }
    }, 3000);
  });
}

effectuerUneTache()
  .then(resultat => console.log(resultat))
  .catch(erreur => console.error(erreur))
  .finally(() => console.log('La promesse est terminée'));
```

Dans cet exemple, la fonction `effectuerUneTache()` retourne une promesse qui est résolue avec succès ou rejetée avec une erreur en fonction d'un résultat aléatoire. Les blocs `then`, `catch` et `finally` sont utilisés pour gérer le résultat de la promesse. Lorsque la promesse est résolue avec succès, le bloc `then` est exécuté avec le résultat de la promesse comme argument. Lorsque la promesse est rejetée, le bloc `catch` est exécuté avec l'erreur de la promesse comme argument. Le bloc `finally` est exécuté indépendamment du résultat de la promesse.

En résumé, les blocs `then`, `catch` et `finally` sont utilisés pour gérer les promesses en JavaScript. Ils permettent de traiter les promesses résolues et rejetées, ainsi que d'exécuter du code indépendamment du résultat de la promesse.

Exercices

14. Utilisez l'API `fetch()` pour récupérer les données à partir de l'URL `https://jsonplaceholder.typicode.com/posts` et affichez le résultat dans la console.
15. Modifiez votre code pour filtrer les résultats pour n'afficher que les articles dont l'utilisateur avec l'ID 1 est l'auteur.
16. Utilisez l'API `fetch()` pour envoyer une requête POST à l'URL `https://jsonplaceholder.typicode.com/posts` avec les données JSON suivantes :

```
{
  "title": "Mon nouvel article",
  "body": "Contenu de mon nouvel article",
  "userId": 1
}
```

17. Écrire une fonction asynchrone qui utilise l'API `fetch()` pour récupérer les données à partir de l'URL `https://jsonplaceholder.typicode.com/posts` et qui retourne un tableau d'objets qui contient le titre et le corps de chaque article.

18. Écrire une fonction asynchrone qui utilise l'API `fetch()` pour envoyer une requête POST à l'URL `https://jsonplaceholder.typicode.com/posts` avec les données JSON du dessus. La fonction doit retourner l'ID de l'article créé.

Fonctions asynchrones

En JavaScript, les fonctions asynchrones permettent d'effectuer des tâches en arrière-plan sans bloquer l'exécution du code. Pour créer une fonction asynchrone, vous pouvez utiliser le mot-clé `async` avant la définition de la fonction. Lorsqu'une fonction est définie comme asynchrone, elle retourne une promesse qui est résolue avec la valeur de retour de la fonction. Les fonctions asynchrones sont souvent utilisées pour effectuer des tâches qui prennent du temps, comme l'envoi de requêtes à un serveur ou la manipulation de fichiers.

Voici un exemple de fonction asynchrone qui utilise la méthode `setTimeout()` pour simuler une tâche qui prend du temps :

```
async function maFonction() {
  console.log('Début de la fonction');
  await setTimeout(() => console.log('Fin de la fonction'), 3000);
}

maFonction();
console.log('debut')

// résultat dans la console
// instantané :
// "Début de la fonction"
// "debut"
// après 3s
// "Fin de la fonction"
```

Dans cet exemple, la fonction `maFonction()` est définie comme asynchrone et utilise la méthode `setTimeout()` pour simuler une tâche qui prend du temps. La méthode `await` est utilisée pour attendre la fin de la tâche avant de continuer l'exécution de la fonction. Lorsque la tâche est terminée, la fonction affiche un message dans la console.

Les fonctions asynchrones sont souvent utilisées en conjonction avec les promesses pour effectuer des tâches de manière asynchrone. Les promesses sont des objets qui représentent une valeur qui peut ne pas être disponible immédiatement. Les promesses peuvent être en attente, résolues avec une valeur, ou rejetées avec une erreur.

Voici un exemple d'utilisation de promesses avec une fonction asynchrone :

```
function effectuerUneTache() {
  return new Promise((resolve, reject) => {
    setTimeout(() => resolve('La tâche est terminée'), 3000);
  });
}

async function maFonction() {
  console.log('Début de la fonction');
  const resultat = await effectuerUneTache();
  console.log(resultat);
}

maFonction();

// résultat dans la console
// instantané :
// "Début de la fonction"
// après 3s
// "La tâche est terminée"
```

Dans cet exemple, la fonction `effectuerUneTache()` retourne une promesse qui est résolue avec un message après un délai de 3 secondes. La fonction `maFonction()` est définie comme asynchrone et utilise la méthode `await` pour attendre la fin de la tâche avant de continuer l'exécution de la fonction. Lorsque la tâche est terminée, la fonction affiche un message dans la console.

En résumé, les fonctions asynchrones et les promesses sont des outils utiles pour effectuer des tâches de manière asynchrone en JavaScript. Ils permettent d'effectuer des tâches qui prennent du temps sans bloquer l'exécution du code, ce qui améliore l'expérience utilisateur.

Exercices

1. Utilisez la fonction `effectuerUneTache()` de l'exemple précédent pour écrire une fonction qui utilise les blocs `then`, `catch` et `finally` pour afficher un message dans la console lorsque la tâche est terminée avec succès, lorsque la tâche a échoué, ou lorsque la promesse est terminée.
2. Écrire une fonction qui utilise l'API `fetch()` pour récupérer les données à partir de l'URL `https://jsonplaceholder.typicode.com/users`. Utilisez les blocs `then`, `catch` et `finally` pour afficher les données dans la console en cas de succès, afficher une erreur en cas d'échec, et afficher un message indiquant la fin de la promesse dans tous les cas.

Simulez 2 cas, un cas où l'appel s'effectue sans erreurs et retourne la réponse, un autre cas où une erreur survient (à vous de faire en sorte qu'une erreur soit présente) et retourné un message d'erreur.

3. Réparez la fonction suivante qui renvoie un user `undefined`:

```
const fetchUser = (id) => {
  fetch(`https://jsonplaceholder.typicode.com/users/${id}`)
  .then(res => res.json())
  .then(data => console.log(data))
}

const user = fetchUser(4)
console.log("user", user)

// résultat dans la console, dans l'ordre
// "user" undefined
// [résultat du console log de data]
```

Base HTML

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Exercice sur le DOM</title>
  </head>
  <body>
    <h1>Exercice javascript</h1>

    <div class="maClasse">
      <p>Ceci est un paragraphe.</p>
    </div>

    <button>Cliquez ici</button>

    <ul>
      <li><a href="#">Lien 1</a></li>
      <li><a href="#">Lien 2</a></li>
      <li><a href="#">Lien 3</a></li>
    </ul>

    <script src="script.js"></script>
  </body>
</html>
```

Introduction aux composant web natifs

Les composants web natifs sont des éléments HTML qui sont intégrés dans les navigateurs web et qui permettent de créer des interfaces utilisateur interactives et dynamiques. Ces éléments incluent des boutons, des champs de formulaire, des menus déroulants, des listes et bien plus encore. Les développeurs peuvent utiliser ces éléments pour créer des pages web riches en fonctionnalités sans avoir à écrire de code JavaScript personnalisé. Les composants web natifs sont également compatibles avec les technologies d'accessibilité, ce qui permet de créer des sites web accessibles pour tous les utilisateurs.


```
<button onclick="console.log('Cliqué !')">Cliquez ici</button>
```

Dans cet exemple, nous avons créé un bouton HTML et avons ajouté un gestionnaire d'événements onclick qui affiche un message dans la console lorsque le bouton est cliqué

```
<custom-button text="Cliquez ici"></custom-button>
```

```
class CustomButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        button {
          background-color: #4CAF50; /* Green */
          border: none;
          color: white;
          padding: 15px 32px;
          text-align: center;
          text-decoration: none;
          display: inline-block;
          font-size: 16px;
          margin: 4px 2px;
          cursor: pointer;
        }
      </style>
```

```
      <button>${this.getAttribute('text')}</button>
    `;
    this.shadowRoot.querySelector('button').addEventListener('click', () => {
      console.log('Cliqué !');
    });
  }
}

customElements.define('custom-button', CustomButton);
```

Dans cet exemple, nous avons créé un composant web personnalisé appelé `custom-button`. Le composant a un attribut `text` qui spécifie le texte affiché sur le bouton. Le code HTML `<custom-button text="Cliquez ici"></custom-button>` crée une instance de ce composant. Le code JavaScript définit la classe `CustomButton` qui étend la classe `HTMLElement`. Le constructeur de `CustomButton` crée un "shadow DOM" (un DOM interne isolé pour le composant), définit le style et le contenu HTML du bouton, et ajoute un gestionnaire d'événements pour le clic sur le bouton qui affiche un message dans la console.

> La fonction **`super()`** est appelée dans le constructeur d'une classe enfant pour appeler le constructeur de la classe parente. Cela permet à la classe enfant d'hériter des propriétés et des méthodes de la classe parente. Par exemple, si vous avez une classe enfant `Animal` qui étend la classe parente `Mammal`, vous pouvez appeler `super()` dans le constructeur de `Animal` pour appeler le constructeur de `Mammal`. Cela permettra à `Animal` d'hériter des propriétés et des méthodes de `Mammal`.
>
> La fonction **`super()`** peut également être utilisée pour appeler des méthodes de la classe parente à partir de la classe enfant. Par exemple, si vous avez une méthode `doSomething()` dans la classe parente et que vous voulez l'appeler à partir de la classe enfant, vous pouvez utiliser `super.doSomething()`.
>
> Dans le cas des composants web, la classe parente est généralement la classe `HTMLElement`, qui est la classe de base pour tous les éléments HTML. La classe enfant étend la classe

> `HTMLElement` pour créer un nouveau type d'élément HTML avec des fonctionnalités personnalisées. Par exemple, dans l'exemple de composant web donné précédemment, la classe `CustomButton` étend la classe `HTMLElement` pour créer un nouveau type de bouton HTML avec un style personnalisé et un gestionnaire d'événements de clic personnalisé.
>
> La fonction **`super()`** est appelée dans le constructeur de la classe enfant pour appeler le constructeur de la classe parente. Cela permet à la classe enfant d'hériter des propriétés et des méthodes de la classe parente, telles que les propriétés `id`, `className`, `children`, `style`, etc. Ensuite, la classe enfant peut ajouter des fonctionnalités personnalisées en définissant des propriétés et des méthodes supplémentaires.
>
> En résumé, la fonction **`super()`** est utilisée dans les classes enfant pour appeler le constructeur et les méthodes de la classe parente. Cela permet à la classe enfant d'hériter des propriétés et des méthodes de la classe parente.

```
<custom-button text="Cliquez ici"></custom-button>
```

```
class CustomButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.compteur = 0;
    this.shadowRoot.innerHTML = `
      <style>
        button {
          background-color: #4CAF50; /* Green */
          border: none;
          color: white;
          padding: 15px 32px;
          text-align: center;
          text-decoration: none;
```

```
          display: inline-block;
          font-size: 16px;
          margin: 4px 2px;
          cursor: pointer;
        }
      </style>
      <button>${this.getAttribute('text')}</button>
    `;

    this.shadowRoot.querySelector('button').addEventListener('click', () => {
      this.compteur++;
      console.log(`Cliqué ${this.compteur} fois !`);
    });
  }
}

customElements.define('custom-button', CustomButton);
```

Dans cet exemple, nous avons ajouté un compteur qui s'incrémente à chaque clic sur le bouton. Le compteur est initialisé à 0 dans le constructeur de la classe `CustomButton`. Lorsque le bouton est cliqué, le gestionnaire d'événements incrémente le compteur et affiche le nombre de clics dans la console.

```
class CustomButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.compteur = 0;

    this.shadowRoot.innerHTML = `
      <style>
        button {
          background-color: #4CAF50; /* Green */
          border: none;
          color: white;
          padding: 15px 32px;
          text-align: center;
          text-decoration: none;
          display: inline-block;
          font-size: 16px;
          margin: 4px 2px;
          cursor: pointer;
        }
      </style>
      <button>${this.getAttribute('text')}</button>
    `;
    this.shadowRoot.querySelector('button').addEventListener('click', this.handleClic
k.bind(this));
  }

  handleClick() {
    this.compteur++;
    console.log(`Cliqué ${this.compteur} fois !`);
```

```
  }
}

customElements.define('custom-button', CustomButton);
```

Dans cet exemple refactorisé, nous avons créé une méthode `handleClick()` qui incrémente le compteur à chaque clic sur le bouton et affiche le nombre de clics dans la console. Cette méthode est liée à l'instance de la classe `CustomButton` par la méthode `bind()` dans le constructeur. Lorsque le bouton est cliqué, l'événement est transmis à `handleClick()`, qui incrémente le compteur et affiche le nombre de clics dans la console.

> *Pensez à retirer vos events listener lorsque l'élément n'est plus affiché :*

```
class CustomButton extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.compteur = 0;

    this.shadowRoot.innerHTML = `
      <style>
        button {
          background-color: #4CAF50; /* Green */
          border: none;
          color: white;
          padding: 15px 32px;
          text-align: center;
          text-decoration: none;
          display: inline-block;
          font-size: 16px;
          margin: 4px 2px;
          cursor: pointer;
        }
      </style>
      <button>${this.getAttribute('text')}</button>
    `;
    this.handleClick = this.handleClick.bind(this);
    this.shadowRoot.querySelector('button').addEventListener('click', this.handleClic
k);
  }

  handleClick() {
    this.compteur++;
    console.log(`Cliqué ${this.compteur} fois !`);
  }

  disconnectedCallback() {
    this.shadowRoot.querySelector('button').removeEventListener('click', this.handleCl
```

```
ick);
  }
}

customElements.define('custom-button', CustomButton);
```

Dans cet exemple, j'ai ajouté une méthode `disconnectedCallback()` qui retire l'écouteur d'événements lorsqu'un composant personnalisé est déconnecté du DOM. Dans ce cas, l'écouteur d'événements est retiré afin de prévenir les fuites de mémoire et les erreurs potentielles.

Um mot sur le shadowRoot

Le `shadowRoot` est un DOM interne isolé pour les composants web personnalisés. Il permet de créer un arbre DOM qui est séparé du reste de la page, ce qui permet d'éviter les conflits de style et de DOM entre les différents composants. Les éléments créés dans le `shadowRoot` sont appelés "nœuds de la racine de l'ombre" ou "nœuds de l'ombre".

L'utilisation du `shadowRoot` est utile pour créer des composants web personnalisés qui ont leur propre style et leur propre comportement. Les styles appliqués aux éléments du `shadowRoot` ne sont pas appliqués aux éléments en dehors du `shadowRoot`, ce qui permet de créer des designs personnalisés pour les composants web. Les événements déclenchés dans le `shadowRoot` ne sont pas propagés en dehors du `shadowRoot`, ce qui permet d'isoler les comportements des différents composants web.

La propriété `shadowRoot` est disponible sur tous les éléments HTML personnalisés. Pour créer un `shadowRoot`, utilisez la méthode `attachShadow()`. Cette méthode prend un objet `ShadowRootInit` en argument qui spécifie le mode d'ouverture du `shadowRoot`. Il existe deux modes d'ouverture : `open` et `closed`. Dans le mode `open`, le `shadowRoot` est accessible depuis l'extérieur de l'élément. Dans le mode `closed`, le `shadowRoot` est inaccessible depuis l'extérieur de l'élément.

Voici un exemple de création d'un `shadowRoot` :

```
class MaClasse extends HTMLElement {
  constructor() {
    super();
    this.attachShadow({ mode: 'open' });
    this.shadowRoot.innerHTML = `
      <style>
        h1 {
          color: red;
        }
```

```
      </style>
      <h1>Mon titre</h1>
    `;
  }
}

customElements.define('ma-classe', MaClasse);
```

Dans cet exemple, nous avons créé un composant web personnalisé appelé `ma-classe`. La classe `MaClasse` étend la classe `HTMLElement`. Le constructeur de `MaClasse` utilise la méthode `attachShadow()` pour créer un `shadowRoot` dans le mode `open`. Le contenu HTML et les styles de `MaClasse` sont définis dans le `shadowRoot`.

> Si vous attachez un shadow root à un élément personnalisé avec `mode: closed`, vous ne pourrez pas accéder au shadow DOM depuis l'extérieur - `myCustomElem.shadowRoot` renvoie `null`.

Les éléments créés dans le `shadowRoot` peuvent être manipulés à l'aide de la propriété `shadowRoot` de l'objet de l'élément. Par exemple, pour sélectionner l'élément `h1` dans l'exemple précédent, vous pouvez utiliser `this.shadowRoot.querySelector('h1')`.

En résumé, le `shadowRoot` est un DOM interne isolé pour les composants web personnalisés. Il permet de créer des styles et des comportements personnalisés pour les composants web, et d'éviter les conflits de style et de DOM entre les différents composants.

Cycles de vie

Les composants web personnalisés ont des cycles de vie qui leur sont propres. Lorsqu'un composant est créé, il passe par plusieurs phases, comme la phase de construction, la phase de rendu et la phase de destruction. Les développeurs peuvent utiliser des méthodes spéciales pour interagir avec ces phases de vie.

Voici un exemple de cycle de vie de composant :

```
class MonComposant extends HTMLElement {
  constructor() {
    super();
    console.log('Le composant est créé');
  }

  connectedCallback() {
    console.log('Le composant est ajouté au DOM');
```

```
  }

  disconnectedCallback() {
    console.log('Le composant est retiré du DOM');
  }
}

customElements.define('mon-composant', MonComposant);
```

Dans cet exemple, nous avons défini un composant appelé `MonComposant`. Le constructeur est appelé lorsque le composant est créé, la méthode `connectedCallback()` est appelée lorsque le composant est ajouté au DOM, la méthode `disconnectedCallback()` est appelée lorsque le composant est retiré du DOM.

Les développeurs peuvent utiliser ces méthodes pour effectuer des tâches spécifiques à chaque phase du cycle de vie du composant, comme l'initialisation de variables, la mise à jour de l'interface utilisateur ou la libération de ressources.

En résumé, les composants web personnalisés ont des cycles de vie qui leur sont propres, et les développeurs peuvent utiliser des méthodes spéciales pour interagir avec ces phases de vie. Les composants web personnalisés sont un outil puissant pour créer des interfaces utilisateur riches en fonctionnalités et pour améliorer l'expérience utilisateur de leurs sites web.

La méthode `attributeChangedCallback()` est appelée lorsque les attributs du composant sont modifiés. Voici un exemple qui utilise la méthode `getAttribute()` et la méthode `attributeChangedCallback()` :

```
class AppToolbar extends HTMLElement {
  constructor() {
    super();
    console.log('This is from the constructor');
    this.parah = document.createElement('p');
  }
  static get observedAttributes() {
    return ['title'];
  }
  get title() {
    return this.getAttribute('title');
  }
  set title(value) {
    this.setAttribute('title', value);
  }
  connectedCallback() {
    this.parah.textContent = this.title;
    this.appendChild(this.parah);
  }
  disconnectedCallback() {
    console.log('This is from the disconnectedCallback method');
```

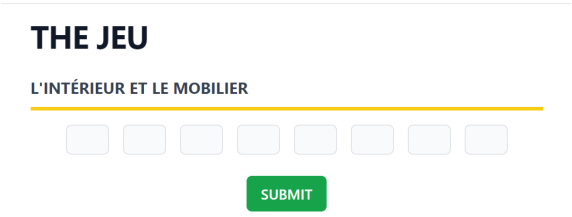
```
  }
  attributeChangedCallback(name, oldValue, newValue) {
    if (name === 'title') {
      this.parah.textContent = newValue;
    }
  }
}

customElements.define('app-toolbar', AppToolbar);
```

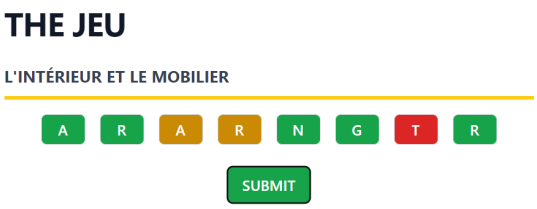
Dans l'exemple du dessus, nous définissons un classe `AppToolbar`, qui crée un élément de paragraphe, et son `connectedCallback`, qui définit le contenu du paragraphe en fonction de l'attribut `title` de l'élément. Il définit également la méthode `observedAttributes`, qui renvoie un tableau de noms d'attributs que l'élément souhaite observer pour les modifications, et la méthode `attributeChangedCallback`, qui est appelée lorsqu'un de ces attributs change.

❓ Bien d'autres choses sont disponibles avec les composants web, ex: https://developer.mozilla.org/en-US/docs/Web/Web_Components/Using_templates_and_slots

Exercice

Création d'un jeu Motus (?)





Le but est de définir un élément HTML personnalisé appelé "word-game".
Il récupère un ensemble de mots aléatoires à partir d'une API https://trouve-mot.fr/ et permet à l'utilisateur de les deviner un par un en entrant des lettres individuelles.

- Le jeu a trois états : START, GAME et END.
- Dans l'état de départ, l'utilisateur saisit le nombre de mots qu'il veut deviner.
- Dans l'état de jeu, le mot actuel est affiché avec des champs de saisie pour chaque lettre. Lorsque l'utilisateur soumet sa réponse, ses entrées sont évaluées et des commentaires sont donnés en fonction de la correction, de l'incorrection ou du placement incorrect des lettres.
- Dans l'état final, l'utilisateur est informé qu'il a gagné et peut redémarrer le jeu.

> ⚠️ Ne vous inquiétez pas du confort de l'utilisateur, ex: quand j'appuie sur la flèche droite de mon clavier, je vais sur l'input d'après ou encore si je remplie un input je passe directement à la suivante. C'est pas le plus important ici (sauf si vous avez fini en avance).

▼ avant d'aller récupérer les données de l'API, vous pouvez utiliser le JSON suivant.

```
const WORDS = [
  {
    "name": "magie",
    "categorie": "SPORTS ET JEUX"
  },
  // {
  //   "name": "signe",
  //   "categorie": "GESTES ET MOUVEMENTS"
  // },
  // {
  //   "name": "durée",
  //   "categorie": "ÉPOQUE – TEMPS - SAISONS"
  // },
  // {
  //   "name": "habit",
  //   "categorie": "VÊTEMENTS – TOILETTE - PARURES"
  // },
  // {
  //   "name": "sucré",
  //   "categorie": "LES ALIMENTS – LES BOISSONS- LES REPAS"
  // },
]
```

▼ je vous conseille de créer des constantes pour le code qui ne change pas et qui est répété, ex : les phases de jeu

```
const STATE = {
  START: "start",
  GAME: "game",
  END: "end",
};
```

- Planifier avant de coder. Je vous conseille de découper votre travail en plusieurs étapes.
    - Dans un premier temps, je construis les étapes du jeu sans appel API et sans mot. Juste en appuyant sur les boutons de soumission, j'arrive à faire une manche entière et à restart.
      Ou se concentrer sur la partie GAME où l'on doit deviner les mots, avec gestions des différends cas de figure (lettre bonne, lettre mal placée, lettre n'est pas dans le mot).
    - Je vous conseille d'attendre que tout fonctionne pour mettre en place l'appel API, ce qui consommera moins de ressources. D'où l'intérêt d'aller récupérer dans la doc généralement ce à quoi ressemblera la réponse (JSON ici).