**LES API**

Une API est un ensemble de définitions et de protocoles qui facilite la création et l’intégration de logiciels d’applications. Elle est parfois considérée comme un contrat entre un fournisseur d’informations et un utilisateur d’informations, qui permet de définir le contenu demandé au consommateur (l’appel) et le contenu demandé au producteur (la réponse). API est un acronyme anglais qui signifie ‘’Application Programming Interface’’ que l’on traduit par l’interface de programmation d’application…

1. **XMLHttpRequest**

C’était la méthode qui était la plus rependue et totalement utilisée avant qu’on ne créer le **fetch**

Cette méthode n’est pas très puissante car elle demande beaucoup de détails, mais c’est une méthode qui permet de faire des requêtes.

Pour l’utiliser, on commence par faire :

let req = new XMLHttpRequest();

**req()** est un nouvel objet XMLHttpRequest(), vu que c’est un objet donc il possède plusieurs méthodes

req.onload = reqListener;

reqLister est un écouteur de requête

On utiliser ensuite la méthode **open()** qui permet de récupérer les données à distance

NB : Nous pouvons aussi aller récupérer des données d’une URL, ceci va nous renvoyer une réponse d’un serveur distant

*//XMLHttpRequest*

function reqListener()

        {

            console.log(this.responseText);

        }

let req = new XMLHttpRequest();

req.onload = reqListener;

*// req.open('get', 'data.txt', true);*

*// req.open('get', 'data.json', true);*

req.open('get', "https://api.blablagues.net/?rub=blagues", true);

req.send();

Nous pouvons même récupérer des données gardées dans un fichier .json, par exemple nous allons le faire avec le fichier settinsJSON de notre vsCode

1. **FETCH(Vas chercher)**

**Le FETCH est un peu une révolution du XMLHttpRequest, elle est beaucoup plus pratique**

**Voici comment elle se compose :**

fetch("https://api.blablagues.net/?rub=blagues").then((*response*) => {

*//response*

})

.catch((*err*) => {console.log(*err*)})

**Que fait le then ?**

Le **then** est utilisé avec une **Promise** (comme celle que retourne fetch) pour traiter la **réponse une fois qu'elle est prête**.

1. **Le rôle de fetch :**
   * Quand on appelle **fetch**, il envoie une requête à l'URL que tu lui donnes (ici : "https://api.blablagues.net/?rub=blagues").
   * Cela retourne une **Promise**, qui attend que la requête soit terminée.
2. **Le rôle de then :**
   * Quand la requête réussit (par exemple, si l'API répond correctement), **then** reçoit la réponse.
   * Le contenu de **then** *(ici, la fonction (response) => { ... })* permet de **faire quelque chose avec cette réponse**, comme l’analyser ou l’afficher.
3. **catch pour les erreurs :**
   * Si quelque chose tourne mal (par exemple, pas d'internet, problème avec l'API), **catch** attrape l'erreur pour qu’on puisses la gérer (par exemple, l’afficher dans la console avec **console.log(err)).**

Une Promesse (*Promise* en anglais) est un objet en JavaScript qui représente une opération asynchrone, c'est-à-dire une tâche qui prend du temps à s'exécuter (par exemple, appeler une API, lire un fichier, etc.).

Avec une Promesse, tu peux gérer ce qui se passe :

* quand l'opération réussit (résultat disponible),
* ou quand elle échoue (erreur).

**Comment fonctionne une Promesse ?**

Imagine une promesse comme une commande dans un restaurant :

1. Vous passez votre commande (appelle une API avec **fetch**).  
   -> Le serveur vous promet que vous recevrez un plat (succès) ou une excuse (erreur).
2. Le serveur prépare la commande.  
   -> C'est l'attente (*pending*).
3. Le plat arrive.  
   -> Promesse tenue (*resolved*). Avec **then**, vous recevez et utilisez la réponse.
4. Le serveur s'excuse si quelque chose tourne mal.  
   -> Promesse rompue (*rejected*). Avec **catch**, vous gérezl'erreur.

**Les états d'une Promesse**

1. **Pending** *(en attente)* : La promesse est en cours.  
   Exemple : l'API traite la requête.
2. **Fulfilled** *(réussie)* : L'opération a réussi.  
   Exemple : tu reçois une réponse de l'API.
3. **Rejected** *(échouée)* : L'opération a échoué.  
   Exemple : problème réseau, erreur API.

*La fonction* ***fetch()*** *prend en paramètre le lien vers l’API qu’on souhaite exploiter, après on met le* ***then()*** *qui prend en paramètre une fonction fléchée dont le paramètre est la réponse (Du lien passé en paramètre) et dans la fonction fléchée, on peut donc exploiter cette réponse ;*

*Nous pouvons encore ajouter la méthode* ***catch()*** *qui prend en paramètre une fonction fléchée et qui va nous permettre de capturer l’erreur*

NB : Nous pouvons aussi mettre un fichier(Local) à la place du lien :

fetch('data.txt').then((*res*) =>console.log(*res*));

Note\_\_\_ Si on veut pouvoir lire le contenu du fichier text

fetch('data.txt')

.then((*res*) => *res*.text())

.then((*data*) => console.log(*data*));

Note\_\_\_ Pour un fichier JSON:

fetch('data.json')

.then((*res*) => *res*.json())

.then((*data*) => console.log(*data*));

1. **TP1 (Générateur de blagues)**

NB: La première chose à faire lorsqu’on veut travailler avec une API c’est de tester le lien de l’API… Il suffit ici de copier le lien de l’API et de le coller dans la barre de recherche du navigateur !!!

**Note\_\_\_ L’utilisation du plugin JSON VIEW est recommandée**

1. **Méthodes POST, PUT, DELETE**

**On va se simuler un serveur pour voir comment on peut manipuler les données**

1. **JSON (JavaScript Object Notation)**

* **.json() est une méthode qui s’auto-résout en rayant le Body de la requête**
* fetch('data.json').then((*res*)=> *res*.json())
* .then((*data*) => {
* console.log(*data*);
* console.log(JSON.stringify(*data*));*//Pour transformer un objet en chaine de caractère JSON*
* })

fetch('data.json').then((*res*)=> *res*.json())

.then((*data*) => {

    console.log(*data*);

    let settings = JSON.stringify(*data*);*//Pour transformer un objet en chaine de caractère JSON*

*//Parse => transforme JSON en Objet JavaScript*

    console.log(JSON.parse(settings));

});

1. **WEB API (Les API natives du navigateur)**
2. **CLIENT STORAGE**

* **Local Storage**

**Nous pouvons stocker des données sur l’utilisateur en utilisant le local storage, ces données vont rester dans le navigateur de l’utilisateur ce qui fait que la prochaine fois qu’il va revenir sur le site, on aura déjà cette information sur lui**

localStorage.data = "Je stock la data"; *//On stocke cet élément dans le navigateur de l'utilisateur*

console.log(window.localStorage.data);*//On  récupère l'élément stocké dans l'utilisateur*

**NB : Il faut automatiquement passer des chaines de caractères à notre élément.**

**Mais il y’a un moyen de stocker d’autres types d’éléments (Objet, Tableau, etc…)**

localStorage.data = "Je stock la data"; *//On stocke cet élément dans le navigateur de l'utilisateur*

let obj = {

    name: 'Jeovany',

    age: 12

}

localStorage.user = JSON.stringify(obj);*//On transforme l'élément en JSON pour pouvoir la garder*

console.log (JSON.parse (localStorage.user)); *//On la reconverti en JS pour pouvoir la récupérer*

* **SESSION STORAGE**
* sessionStorage.dataSettings = "55px"; *//Permet de stocker l'élément dans le Session Storage du navigateur*

**1. Le problème de base avec les opérations asynchrones :**

Imaginons que tu appelles une API (comme l'API des repas) pour obtenir des données, mais que cela prend un certain temps (quelques millisecondes ou secondes). Pendant que tu attends la réponse, JavaScript continue à exécuter les lignes de code suivantes. Cela peut créer des problèmes, car tu pourrais essayer d'utiliser les données avant même qu'elles ne soient arrivées.

C’est là que **async** et **await** viennent à la rescousse ! Ils te permettent de **"mettre en pause"** le code et d'attendre que la réponse soit prête avant de continuer.

**2. Rôle de async :**

Quand tu mets **async** devant une fonction, tu dis à JavaScript :

"Cette fonction va faire quelque chose qui prend du temps (comme appeler une API), alors prépare-toi à attendre."

Dans ton cas :

async function fetchMeals(params) {

// Ici, tu dis que fetchMeals est une fonction asynchrone

}

**3. Rôle de await :**

**await** est utilisé **à l'intérieur** d'une fonction marquée avec async. Il indique à JavaScript d'**attendre que quelque chose soit terminé avant de continuer**.

Dans ton exemple, voici ce qui se passe :

***await fetch("https://www.themealdb.com/api/json/V1/1/search.php?s=duck")***

Cela signifie que JavaScript va **attendre** que les données soient reçues de l'API avant de continuer. Sans await, JavaScript aurait essayé de continuer l'exécution sans attendre la réponse.

**Explication étape par étape de ton code :**

1. **async function fetchMeals()** :
   * Tu dis que cette fonction est asynchrone, donc elle peut "attendre" certaines opérations comme un appel d'API.
2. **await fetch("https://www.themealdb.com/api/json/V1/1/search.php?s=duck")** :
   * Ici, tu fais un appel à l'API. Grâce à await, JavaScript attend que la réponse arrive avant de continuer.
   * **Sans await**, JavaScript ne ferait pas attention à l'appel d'API et continuerait à exécuter le reste du code, ce qui pourrait poser problème car meals n'aurait pas encore les données.
3. **.then((res) => res.json())** :
   * Une fois que la réponse de l'API arrive, tu transformes les données reçues en un format JSON lisible.
4. **.then((data) => meals = data.meals)** :
   * Une fois les données converties en JSON, tu les stockes dans la variable meals.
5. **console.log(meals)** :
   * Ce console.log s’exécute une fois que toutes les données ont été récupérées et stockées dans meals, donc il affiche les repas reçus de l'API.

**Sans async/await, ça donnerait ceci :**

Si tu n'utilisais pas async et await, tu devrais utiliser seulement les .then() pour t'assurer que tu attends les bonnes étapes :

function fetchMeals() {

**fetch**("https://www.themealdb.com/api/json/V1/1/search.php?s=duck")

.then((res) => res.json())

.then((data) => {

meals = data.meals;

console.log(meals); // Les données sont prêtes ici

});

}

fetchMeals();

Cela fonctionne aussi, mais **async et await** rendent le code plus lisible, comme si tu écrivais du code "synchrone" qui attend naturellement la réponse.

**Conclusion :**

* **async** : Indique que la fonction contient du code asynchrone (qui prend du temps) et qu'elle peut utiliser await.
* **await** : Attends que la promesse (ici, l'appel à l'API fetch) soit résolue avant de continuer le reste du code.