Javascript - 03

Romain Jalabert IPSSI

Local Storage

On va pouvoir à l'aide de **localStorage (window.localStorage)** stocker des informations dans notre navigateur. Pratique pour préserver des informations après rafraîchissement.

Ces données sont sauvegardées dans l'objet **Storage.**

Les données stockées dans localStorage **n'ont pas de délai d'expiration.**

On ajoute un item à notre localStorage dans une logique (key, value) :

```
localStorage.setItem('monChat', 'Tom');
```

On récupère dans une variable 'cat' l'item 'monChat'. 'cat' vaudra donc 'Tom'.

```
var cat = localStorage.getItem('monChat');
```

Pour **supprimer un item** : removeItem('myItem')

Session Storage

Un peu comme pour PHP Session Storage va nous permettre de stocker des informations dans le navigateur sous la forme clé / valeur sauf que cette fois-ci la date d'expiration est fixée à la **fermeture du navigateur**.

Le fonctionnement est le même que pour Local Storage :

- sessionStorage.setItem('key', 'value') -> enregistrer des données
- sessionStorage.getItem('key', 'value') -> lire des données

Exercice Todo Local Storage

-> Reprendre la todo list et enregistrer cette fois-ci les informations côté navigateur avec Local Storage. On voudra également les supprimer du storage si nécessaire.

1) Sauvegarder les todos dans un tableau que l'on placera dans le storage

2) Afficher les todos contenues dans le storage

3) Supprimer du tableau todos celles que l'on souhaite enlever

Bonus : à l'aide d'animate.css rendre la todo un peu plus dynamique avec des effets lors de l'ajout d'une todo et la suppression d'une todo.

Asynchronie dans JS

Opérations Asynchrones

On parle d’opérations **synchrones** en JS lorsqu’elles sèxecutent dans l’ordre : c'est-à dire que la 2eme opération attend la fin d'exécution de la première avant de s'exécuter.

L'exécution de la deuxième opération **dépend donc de la première**. Elles s'exécutent à la suite.

Le problème des opérations dites synchrones c’est qu’une opération particulièrement lourde peut bloquer le reste du fil d’opérations (le temps qu’elle ait fini son exécution).

-> C’est là qu’intervient l’utilité des opérations asynchrones.

Ainsi lorsqu’une opération est **asynchrone,** *son exécution ne bloque pas le reste du fil*, elle s'exécute de façon indépendante, hors du ‘main thread’.

Promises (Resolve, Reject)

Une promesse est un objet qui représente l'état d'une opération asynchrone en cours. Il s'agit d'une promesse d'un résultat futur. Il y a 3 états possibles d'une promesse :

- L'opération est en cours
- L'opération est résolue avec succès
- L'opération est rejetée (donc échec)
-

Syntaxe :

```
new Promise((resolve, reject) => {
    // Code opération asynchrone
    // Appel de resolve si la promesse est résolue
    // Appel de reject si la promesse est rejetée
});
```

Exemple de promesse

```
let myPromise = new Promise(function(resolve, reject) {
    // Code qui peut prendre un certain temps

      resolve(); // en cas de succès
      reject();  // en cas d'erreur
    });

    // Code qui attend le résultat du code précédent
    myPromise.then(
      function(value) { /* Code en cas de succès */ },
      function(error) { /* Code en cas d'erreur */ }
    );
```

Utilisation de then() et catch()

Pour exploiter le résultat d'une promesse on va utiliser la méthode **then()**.

La méthode **then()** sera appelée avec une fonction de rappel (ou callback) si la promesse est tenue sinon **catch()** dans le cas où la promesse est rompue.

On préférera utiliser à la fois then() et catch() pour une question d'efficacité du code.

Le chaînage des promesse : la méthode then() retournant à chaque fois une nouvelle promesse il va être possible de les enchaîner. On parle de "chainer" des méthodes (les exécuter à la suite).

On évite ainsi avec cette syntaxe ce que l'on appelle le **callback hell**

Callback Hell example

```
// Callback Hell


a(function (resultsFromA) {
    b(resultsFromA, function (resultsFromB) {
        c(resultsFromB, function (resultsFromC) {
            d(resultsFromC, function (resultsFromD) {
                e(resultsFromD, function (resultsFromE) {
                    f(resultsFromE, function (resultsFromF) {
                        console.log(resultsFromF);
                    })
                })
            })
        })
    })
});

```

Exemple de chaîne de promesses

```
const promesse = new Promise((resolve, reject) => {
    resolve('5')
    // reject('Échec de la promesse')
}

promesse.then((n) => {console.log('Le nombre :', n)})
        .then (() => {console.log('Le nombre 2')})
        .then (() => {console.log('Puis le 3 !')})
        .catch((e) => {console.log('error', e)})
```

Async / Await

async et **await** nous permettent de faciliter l'écriture des promesses.

Le mot clé async placé avant une fonction permet à la fonction de retourner une promesse.

```
async function myFunction() {
    return "Hello";
    }
    myFunction().then(
    function(value) {myDisplayer(value);},
    function(error) {myDisplayer(error);}
    );
```

Le mot clé await peut être utilisé uniquement dans une fonction async. Il permet de mettre en pause l'exécution de la fonction et d'attendre la résolution de la promesse avant de continuer.

Ajax

AJAX = Asynchronous Javascript and XML

Ajax est un concept fondamental en JS. Le terme désigne un ensemble d'outils dont l'objectif est d'effectuer des **requêtes asynchrones vers un serveur** depuis le **navigateur**.

Ainsi avec AJAX on pourra actualiser certaines données sans que le reste des éléments de notre page ne soient impactés (ou qu'il y ait (re)chargement de la page).

AJAX comprend plusieurs librairies dont :

- fetch
- axios
- JQuery AJAX

AJAX (MDN) : *https://developer.mozilla.org/fr/docs/Web/Guide/AJAX*

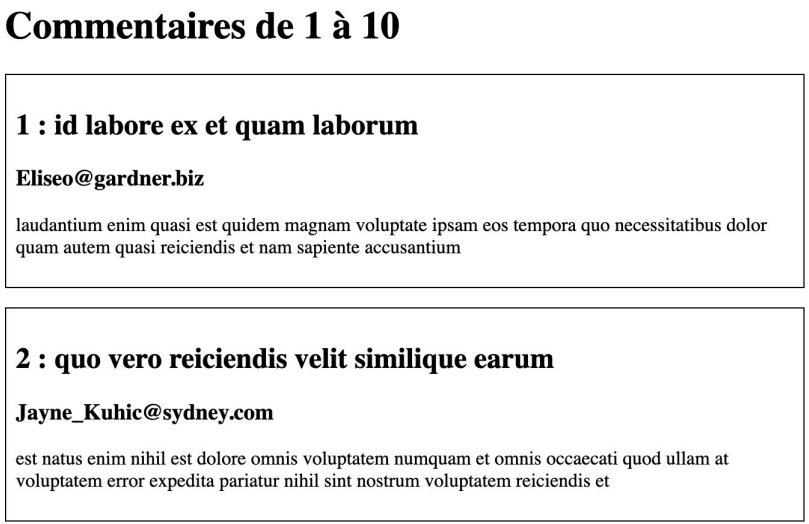
fetch API

fetch API est un des outils qui permet d'effectuer des requêtes asynchrones depuis le navigateur. Il n'est pas supporté par certaines versions des navigateurs (les anciennes) mais le code peut être modifié à cet effet (polyfill)

```
fetch('https://jsonplaceholder.typicode.com/comments', {
    method: 'GET',
    headers: {
        // Format de données que l'on accepte
        "Accept" : "application/json",
        // Format de données qu'on envoit
        "Content-type" : "application/json",
    }
})
    .then(res => res.json())
    .then(body => console.log(body))
```

Exercice API - Commentaires

Avec l'aide de Fetch API, afficher sur une page les 10 premiers commentaires comme suit :



On utilisera comme endpoint : https://jsonplaceholder.typicode.com/comments

Axios

Axios permet de faire des **requêtes HTTP** avec un système de promesse. Axios s'utilise directement depuis le navigateur ou depuis node JS. Il est très utile dans sa gestion des erreurs et compatible avec tous types de navigateur.

Attention il faut dans un premier temps installer axios avec un package manager (npm ou yarn par exemple) ou directement en copiant le lien CDN dans notre <head> (https://axios-http.com/docs/intro)

```
axios.get('https://jsonplaceholder.typicode.com/users')
    .then(response =>
        console.log(response.data))
    .catch(error => console.log(error))
```

Disney API

Encore une fois nous allons faire des requêtes vers une API, cette fois-ci Disney (https://api.disneyapi.dev/characters/). On voudra afficher l'ensemble des personnages qui sont tirés de films Disney comme suit :

Exercice : Random Quote API

À l'aide d'opérations asynchrones et de axios, coder un générateur de citations. Il faut afficher le texte et l'auteur ainsi qu'un bouton de refresh afin de récupérer une nouvelle citation. On utilisera l'API https://type.fit/api/quotes