Javascript module 5

Persistance des données et API

Persistance des données en Javascript

Nous avons appris à manipuler des données en Javascript. Lors de l'exécution de nos scripts et ou des interactions de l'utilisateur, ces données sont amenées à changer. Mais jusqu'ici, à chaque fois que nous rechargeons la page dans le navigateur, nos données sont réinitialisées avec la valeur d'origine que nous avons attribuée à nos variables.

L'idée ici est justement de remédier à cela en demandant au navigateur de garder en mémoire certaines données. Ces données stockées dans le navigateur pourront être appelées ultérieurement. C'est le principe de la persistance des données.

Les navigateurs disposent de plusieurs mémoires dans lesquelles stocker des informations. Il s'agit des *Cookies*, du *Local Storage* et du *Session Storage*. Chacune de ces mémoires a des avantages, des inconvénients et des limites. Dans tous les cas, veuillez à ne jamais stocker d'information sensible des utilisateurs ici (identité, mot de passe, données bancaires,...).

Cookies

Les cookies sont des valeurs stockées dans le navigateur de l'internaute. Elles ont plusieurs spécificités :

- Les cookies sont spécifiques à un navigateur et à un site internet donné.
- Chaque donnée stockée possède une date d'expiration.
- Les données sont envoyées au serveur à chaque demande de page.
- L'espace de stockage est très limité : 4 ko maximum.

En Javascript, l'ensemble des données des cookies stockées sont disponible par la propriété ***cookie*** de ***document*** comme ceci :

```
document.cookie;
```

Ceci a pour valeur une chaîne de caractères contenant toutes les données des cookies stockées pour l'internaute pour le site internet courant (clés, valeurs, date d'expiration...).

Pour stocker des informations qui n'ont pas besoin d'être récupérées côté serveur, nous allons favoriser les autres formes de stockage de données.

Local Storage

Le local storage est de loin le moyen de stockage le plus utile. Il permet de conserver des données pour un même site dans le navigateur qui seront conservées entre 2 sessions. C'est-à-dire que si un internaute quitte votre site internet et ferme son navigateur, les données qui auront été stockées dans le local storage par votre code pourront être récupérées à sa prochaine visite.

Contrairement aux cookies, les données du local storage ne seront pas accessibles côté serveur et resteront dans le navigateur de l'internaute : seul le Javascript peut y accéder.

Manipuler les données

localStorage est une propriété de ***window*** et donc du navigateur. Les données y sont stockées sous la forme de couples clé / valeur. Pour chaque clé vous pouvez stocker une valeur sous la forme d'une chaîne de caractère.

Voici les méthodes courantes permettant de manipuler le local storage :

Action	Méthode
Définir la valeur stockée pour une clé. Si la clé est déjà définie, la valeur sera remplacée.	`localStorage.setItem("myKey", "myValue");`
Récupérer la valeur pour une clé en paramètre.	`localStorage.getItem("myKey");`
Supprimer la clé et la valeur associée du stockage.	`localStorage.removeItem("myKey");`
Supprimer toutes les données stockées pour le site.	`localStorage.clear();`

Vous trouverez ici des documentations plus détaillées sur ce mode de stockage de données :

- https://developer.mozilla.org/fr/docs/Web/API/Window/localStorage
- https://developer.mozilla.org/fr/docs/Web/API/Storage

Stocker des données plus complexe grâce au JSON

Ce stockage dans le local storage est très utile, cependant, le format clé / valeur en chaînes de caractères atteint vite ses limites. En effet, nous pourrions avoir besoin par exemple de stocker les scores d'un jeu. Les données plus complexes pourraient avoir cette forme :

```
Const rounds = [
    {
        date: "2022-06-07",
        level: 2,
        scores: {Samir: 268, Loanne: 325, Tom: 465}
    },
    {
        date: "2022-06-02",
        level: 3,
        scores: {Samir: 488, Loanne: 452, Tom: 665}
    }
];
```

C'est le format JSON qui va être ici la solution.

> Le JSON, ou JavaScript Object Notation, est un format de données texte basé sur les objets Javascript. Il reprend la syntaxe du Javascript avec les accolades pour définir des objets sous forme de clé / valeur, et les crochets pour définir des tableaux.
>
> Vous trouverez plus de détails sur le JSON ici :
>
> - https://fr.wikipedia.org/wiki/JavaScript_Object_Notation
> - https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JSON

Dans notre exemple, nous allons transformer la valeur de la variable ***rounds*** au format JSON afin d'en faire une chaîne de caractère. Ainsi nous allons pouvoir stocker ces données structurées et complexes dans le local storage. A cette fin nous allons utiliser la méthode ***.stringify()*** de ***JSON*** comme ceci :

```
const roundsJson = JSON.stringify(rounds);
localStorage.setItem("rounds", roundsJson);
```

Lorsque nous allons vouloir récupérer ces données dans le local storage, nous allons utiliser la méthode ***.parse()*** de ***JSON*** comme ceci :

```
const roundsJson = localStorage.getItem("rounds");
const rounds = JSON.parse(roundsJson);
```

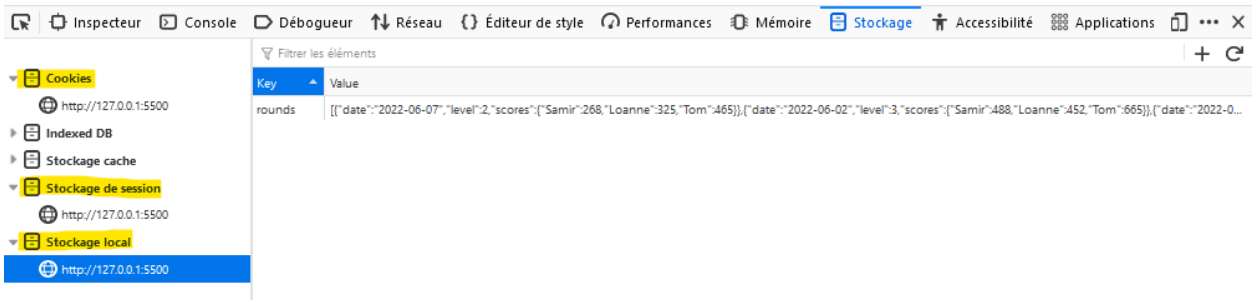
Session Storage

Le session Storage fonctionne exactement de la même manière que le local storage. Il est lui aussi une propriété de ***window*** : ***sessionStorage***. Il possède exactement les mêmes méthodes de manipulation des données.

La seule différence avec le local storage est que les données stockées dans ***sessionStorage*** sont vidées à chaque fois que la session du navigateur prend fin, c'est-à-dire quand l'internaute ferme le navigateur.

Visualiser les données dans le navigateur

Les navigateurs donnent accès aux différents modes de stockage de données que nous venons de voir dans les outils de développement. Ceci permet de visualiser rapidement et de manipuler les données stockées lors du développement. Voici un exemple de l'onglet Stockage de Firefox :



API

Lorsque vous utilisez des applications ou des sites web, ceux-ci utilisent souvent des données qui proviennent eux mêmes d'autres sites. C'est le cas par exemple des sites de comparateurs de prix, de résultats sportifs, de météo, de programmes télé, ...

Cette communication entre 2 sites se nomme API pour Application Programming Interface.

Ajax

Il est de coutume d'appeler la possibilité en Javascript de faire des requêtes HTTP et de changer le contenu de la page sans la recharger Ajax, pour **A**synchronous **JA**vascript **X**ML.

En effet historiquement c'était le format XML qui était utilisé pour charger des données en Javascript. Désormais nous utilisons le JSON.

Utilisation d'une API REST

Les API REST utilisent le protocole HTTP pour échanger des données entre un serveur et un client, tout comme pour la navigation classique sur un site web. Seulement, au lieu d'envoyer des données sous la forme de fichiers au format HTML, le serveur va répondre et envoyer les données structurées selon un certain standard, le plus souvent en JSON.

Nous allons ici interroger une API qui va nous envoyer des données au format JSON.

Cette API est disponible à l'adresse suivante :

- https://jsonplaceholder.typicode.com/todos

En ouvrant cette adresse dans votre navigateur, vous obtiendrez un jeu de données JSON structuré. Ce sont ces données que nous allons chercher à afficher sur notre site.

Pour cela nous allons utiliser la fonction ***fetch()*** (obtenir, ou aller chercher en français) qui va interroger le serveur.

Utilisation d'une promise avec .then()

Voici un exemple de code pour l'interrogation de l'API qui affiche les données dans la console du navigateur.

```
fetch("https://jsonplaceholder.typicode.com/todos")
```

```
    .then(response => response.json())
    .then(json => console.table(json));
```

Ici nous demandons à Javascript d'interroger l'adresse en paramètre de fetch, puis d'interpréter la réponse au format JSON, puis d'afficher ces données dans la console quand elles sont disponibles.

fetch() nous retourne une "promesse" à laquelle nous pouvons appliquer la méthode ***.then()*** qui appellera la fonction de callback en paramètre dès que le serveur aura répondu.

Avec ce code, nous allons donc récupérer dans la variable nommée ***json*** un Array contenant des Objects comme nous avons l'habitude de les manipuler :

```
[
    {
      "userId": 1,
      "id": 1,
      "title": "delectus aut autem",
      "completed": false
    },
    {
      "userId": 1,
      "id": 2,
      "title": "quis ut nam facilis et officia qui",
      "completed": false
    },
    {
      "userId": 1,
      "id": 3,
      "title": "fugiat veniam minus",
      "completed": false
    },
    ...
]
```

Voici une documentation plus poussée du fetch et des promesses:

- https://developer.mozilla.org/fr/docs/Web/API/Fetch_API/Using_Fetch
- https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Using_promises

Utilisation d'une fonction asynchrone

Voici un autre exemple qui fait la même chose, mais en utilisant le concept de fonction asynchrone :

```
async function waitingForResponse() {
    const response = await fetch("https://jsonplaceholder.typicode.com/todos");
    const todoList = await response.json();
    console.table(todoList);
}
waitingForResponse();
```

Pour creuser le sujet des fonctions asynchrone, en voici la documentation :

- https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Statements/async_function

Gérer les exceptions : le try / catch

Le problème quand on interroge un serveur distant c'est que l'on est jamais certain de ce qu'il va se passer : l'API peut ne plus répondre temporairement ou définitivement, elle peut avoir changé d'adresse ou le format des données peut ne plus être celui que l'on attend.

C'est pourquoi nous allons introduire le try / catch : Il va nous permettre de tester notre d'interrogation au serveur. Et si jamais l'interrogation échoue, nous allons attraper l'exception (l'erreur) et définir quoi faire dans ce cas.

Voici la structur classique d'un try / catch

```
try {
    doSomething();
} catch (error) {
    console.error("Something went wrong : " + error);
}
```

Voici ce que celà donne avec l'exemple d'API précédent :

```
async function waitingForResponse() {
    try {
        const response = await fetch("https://jsonplaceholder.typicode.com/todos");
        const todoList = await response.json();
        console.table(todoList);
```

```
    }
    catch(error) {
        console.error("Unable to load todolist datas from the server : " + error);
    }
}
```

Ajax avec XMLHttpRequest

Historiquement ce sont les objets XMLHttpRequest (XHR) qui était utilisés pour effectuer des requêtes Ajax. Ne soyez pas surpris si vous croisez du code avec ces objets, la finalité est la même : demander à Javascript d'exécuter des requêtes HTTP.

Pour notre exemple, nous aurions le code suivant :

```
const xhr = new XMLHttpRequest();
xhr.open("GET", "https://jsonplaceholder.typicode.com/todos");
xhr.responseType = "json";
xhr.send();

xhr.onload = function(){
    if (xhr.status != 200) {
        console.error("Error " + xhr.status + " : " + xhr.statusText);
    } else {
        console.log(xhr.response);
    }
};

xhr.onerror = function(){
    console.error("Request has failed.");
};
```

Une documentation est disponible ici :

- https://developer.mozilla.org/fr/docs/Web/API/XMLHttpRequest/Using_XMLHttpRequest