

Jour 3

JS



Exploiter les API HTML 5 en JavaScript





Validation des formulaires

La validation de formulaires consiste à vérifier les données entrées directement sur le navigateur. Si elles ne respectent pas les conditions que nous imposons alors elles ne seront pas envoyées au serveur.

La validation est incluse dans HTML5, grâce à des attributs qui vont nous permettre de poser ces conditions.

Le javascript lui va nous permettre de contrôler nos balises et gérer l'affichage des messages d'erreur notamment.



Les attributs de validation

- 📌 `min` et `max` vérifie que la donnée entrée est bien supérieure à un minimum ou inférieure à un maximum
- 📌 `required` impose de ne pas laisser le champ vide
- 📌 `step` vérifie que le nombre entré est bien un int.
- 📌 `minlength` et `maxlength` impose une longueur minimum ou maximum à un champ demandant une chaîne de caractères.
- 📌 `pattern` impose un format à l'entrée, exprimé grâce à une expression régulière (regex)



Voici un exemple de code HTML avec des attributs de validation:

```
<form id='form'>  
  <label for="email">Entrez votre adresse mail</label>  
  <input id="email" name="email" required pattern="^[^@]+@[^\.\.]+\.\.+">  
  <button type='submit'>Valider</button>  
</form>
```



Dans ce segment de code, nous avons donc une balise `input` avec l'attribut `required` qui impose donc que le champ soit rempli.

Avec `pattern` Nous lui donnons comme valeur une expression régulière correspondant à la validation d'email (Regard si il y a des caractères avant et après un @ et si il est bien suivi d'un point lui même suivi de caractères.)



En CSS, nous pouvons modifier le style d'un champ grâce aux pseudoclasses `:valid` et `:invalid` :

```
input:invalid {  
  border: 1px solid red // le bord est rouge si non valide  
}  
input:valid {  
  border: 1px solid black // le bord est noir si valide  
}
```



En Javascript nous appelons la méthode `preventDefault` pour empêcher le formulaire d'être envoyé au serveur si il est invalide:

```
const form = document.querySelector('#form');  
form.onsubmit = (e) => {  
  e.preventDefault();  
}
```




Voyons un exemple un peu plus riche avec ce code HTML:

```
<form id='form'>
  <label for="name">What's your name?</label>
  <input id="name" name="name" required minlength='5' maxlength='20'>
  <br>
  <span id='name-too-short' hidden>Name is too short</span>
  <span id='name-too-long' hidden>Name is too long</span>
  <br>
  <button type='submit'>Submit</button>
</form>
```



Pour ce code html, nous avons ce code Javascript:

```
const form = document.querySelector('#form');
const name = document.querySelector('#name');
const nameTooShort = document.querySelector('#name-too-short');
const nameTooLong = document.querySelector('#name-too-long');

form.onsubmit = (e) => {
  e.preventDefault();
}
name.oninput = (e) => {
  nameTooShort.hidden = true;
  nameTooLong.hidden = true;
  if (e.srcElement.validity.tooShort) {
    nameTooShort.hidden = false;
  }
  if (e.srcElement.validity.tooLong) {
    nameTooLong.hidden = false;
  }
}
```



Nous utilisons la méthode `preventDefault` pour empêcher l'envoi du formulaire en cas d'échec.

Dans la fonction appelée `oninput`, d'abord nous cachons les messages `nameTooShort` et `nameTooLong` (récupérés avec le `querySelector` au sommet du fichier).

Dans cette même fonction, grâce à `srcElement.validity.tooShort` si l'entrée est trop courte, alors nous affichons le message `nameTooShort`.

De même pour `nameTooLong` en vérifiant

`e.srcElement.validity.tooLong`.



TP : Validation des formulaires en JavaScript

- 📌 Créer un formulaire d'inscription avec les champs suivants :
nom, prénom, email, mot de passe, confirmation du mot de passe, date de naissance, sexe, pays, telephone.
- 📌 Ce formulaire servira a créer un compte utilisateur pour notre jeu.



Solutions de stockage

Il arrive que l'on ait besoin de stocker des données sur le navigateur de l'utilisateur. Pour cela, il existe plusieurs solutions :

 **Le LocalStorage**

 **Les IndexedDB**



JSON

- 📌 **JSON veut dire JavaScript Object Notation. C'est un format de données très utilisé pour échanger des données entre un serveur et un client.**
- 📌 **Un objet JSON est un objet JavaScript, mais il est écrit sous forme de texte.**



- 📌 JSON est un format de données très utilisé en JavaScript. Il permet de stocker des données sous forme de clé/valeur.
- 📌 Pour accéder aux données stockées dans un fichier JSON, il faut utiliser les méthodes suivantes :
- 📌 `JSON.stringify(object)` transforme un objet JavaScript en chaîne de caractères JSON
- 📌 `JSON.parse(string)` transforme une chaîne de caractères JSON en objet JavaScript

LocalStorage




- 📌 Le LocalStorage permet de stocker des données sur le navigateur de l'utilisateur. Ces données sont stockées sous forme de clé/valeur.
- 📌 Pour accéder aux données stockées dans le LocalStorage, il faut utiliser les méthodes suivantes :
- 📌 `localStorage.getItem(key)` retourne une valeur stockée dans le LocalStorage
- 📌 `localStorage.setItem(key, value)` stocke une valeur dans le LocalStorage
- 📌 `localStorage.removeItem(key)` supprime une valeur stockée dans le LocalStorage




IndexedDB


- 📌 Les IndexedDB permettent de stocker des données sur le navigateur de l'utilisateur. Ces données sont stockées sous forme de clé/valeur.
- 📌 Pour accéder aux données stockées dans les IndexedDB, il faut utiliser les méthodes suivantes :
- 📌 `indexedDB.open(name, version)` ouvre une base de données
- 📌 `indexedDB.deleteDatabase(name)` supprime une base de données




 `db.createObjectStore(name, options)` **crée un object store**

 `db.deleteObjectStore(name)` **supprime un object store**



 `db.transaction(storeNames, mode)` **crée une transaction**

 `tx.objectStore(name)` **retourne un object store**



- 📌 `store.get(id)` **retourne une valeur stockée dans le store**
- 📌 `store.put(value)` **stocke une valeur dans le store**
- 📌 `store.delete(id)` **supprime une valeur stockée dans le store**
- 📌 `store.clear()` **supprime toutes les valeurs stockées dans le store**



```
let openRequest = indexedDB.open("store", 1);

openRequest.onupgradeneeded = function() {
  // Initialisation de la base de données
  // db.createObjectStore(name[, keyOptions]);
  db.createObjectStore('books', {keyPath: 'id', autoIncrement: true});
};

openRequest.onerror = function() {
  // Gestion des erreurs
  console.error("Error", openRequest.error);
};

openRequest.onsuccess = function() {
  // Récupération de la base de données
  let db = openRequest.result;
  // On peut maintenant utiliser la base de données
  let transaction = db.transaction("books", "readwrite"); // (1)

  // get an object store to operate on it
  let books = transaction.objectStore("books"); // (2)

  let book = {
    id: 'js',
    price: 10,
    created: new Date()
  };

  let request = books.add(book); // (3)
  request.onsuccess = function() {S // (4)
    console.log("Book added to the store", request.result);
  };

  request.onerror = function() {
    console.log("Error", request.error);
  };
};
```



TP : Stockage des données en local

- 📌 Stocker l'historique des parties dans le indexDB
- 📌 Stocker les preferences de l'utilisateur dans le localStorage (dark theme)



WebWorkers

- 📌 Les WebWorkers permettent d'exécuter du code JavaScript en arrière-plan.
- 📌 Pour créer un WebWorker, il faut utiliser la méthode suivante :
- 📌 `new Worker(url)` crée un WebWorker
- 📌 Pour envoyer des données à un WebWorker, il faut utiliser la méthode suivante :
- 📌 `worker.postMessage(message)` envoie un message à un WebWorker



📌 Pour recevoir des données d'un WebWorker, il faut utiliser la méthode suivante :

📌 `worker.onmessage = function(event) {}` exécute une fonction lorsque le WebWorker envoie un message

📌 Pour arrêter un WebWorker, il faut utiliser la méthode suivante :

📌 `worker.terminate()` arrête un WebWorker

WebSockets



- 📌 Les WebSockets permettent d'établir une connexion bidirectionnelle entre un client et un serveur.

Pour créer une connexion WebSocket, il faut utiliser la librairie socket io.

- 📌 Ouvrir une connexion WebSocket `socket = io.connect('http://localhost:3000');`

- 📌 Envoyer un message `socket.emit('message', 'Hello World');`

- 📌 Recevoir un message `socket.on('message', function(data) { console.log(data); });`

JS





Différences dans les langages

- 📌 Le Javascript navigateur et le Javascript serveur sont très proches.
- 📌 Le Javascript navigateur est plus limité que le Javascript serveur.
- 📌 Le Javascript navigateur est plus lent que le Javascript serveur.



Asynchronisme

- 📌 **JavaScript est un langage asynchrone. Cela signifie que le code est exécuté de manière non linéaire. Par exemple, si on exécute une fonction qui prend du temps à s'exécuter, le code suivant sera exécuté avant que la fonction ne soit terminée.**
- 📌 **Pour éviter ce problème, il faut utiliser des fonctions de rappel. Une fonction de rappel est une fonction qui est exécutée lorsque la fonction qui l'a appelée est terminée.**
- 📌 **Les fonctions de rappel sont souvent utilisées avec des fonctions asynchrones.**



📌 Nous avons deux mots clefs a retenir : `async` et `await`

📌 `async` permet de déclarer une fonction asynchrone

📌 `await` permet d'attendre la fin de l'exécution d'une fonction asynchrone (elle ne peut être utilisée que dans une fonction asynchrone)



```
async function myFunction() {  
  let movies = await fetch('https://example.com/movies.json');  
  let moviesJson = await movies.json();  
  
  console.log(moviesJson);  
}  
  
myFunction().then(() => {  
  console.log('Finished!');  
});
```

REST serveur en Node.js et REST client en JavaScript

