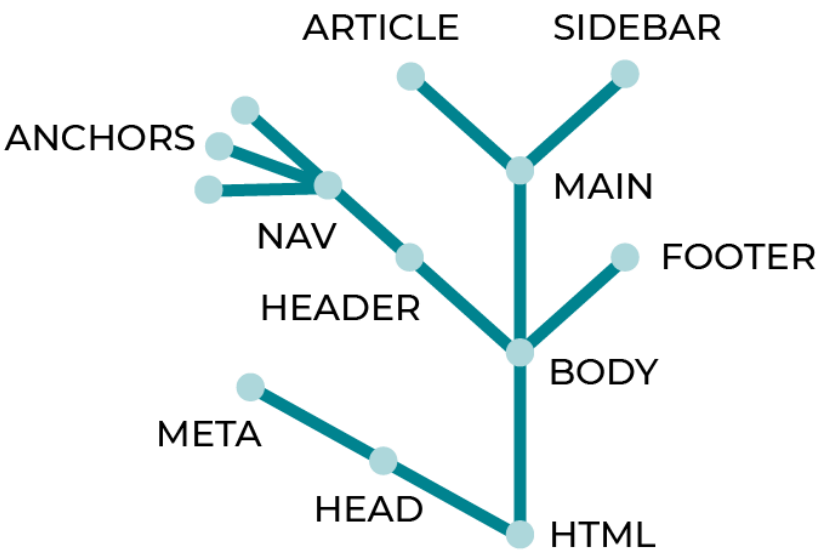
JAVASCRIPT POUR LE WEB

Table des matières

I. Manipuler le DOM

1. Comprenez ce qu'est le DOM

Le DOM signifie Document Object Model. C'est une interface de programmation qui est une représentation du HTML d'une page web. Il permet d'accéder aux éléments de cette page web et de les modifier avec le langage JavaScript.



Il faut voir le DOM comme un arbre où chaque élément peut avoir zéro ou plusieurs enfants, qui peuvent avoir zéro ou plusieurs enfants etc…

Dans le DOM, on commence toujours par un élément racine qui est le point de départ du document : la balise <html>. Celle-ci a pour enfants les balises <head> et <body>. Le contenu de la page se trouve ensuite dans la balise <body>.

On va ensuite pouvoir interagir avec le DOM :

- Modifier le contenu d'un élément bien précis.
- Modifier le style d'un élément.
- Créer ou supprimer des éléments.
- Interagir avec l'utilisateur afin de repérer les clics sur un élément, ou encore récupérer les données d'un formulaire.
- Etc…

https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement

2. Accédez aux éléments du DOM

Chaque élément du DOM est un objet JavaScript avec ses propriétés et ses fonctions pour le manipuler. Tout commence avec le *document*. C'est un objet auquel on a directement accès dans le code JavaScript. Il représente la page entière.

Pour rechercher des éléments du DOM :

- **document.getElementById()**

C'est la méthode la plus utilisée pour retrouver un élément car c'est la seule qui permette de retrouver facilement un élément précis. Elle va rechercher un élément grâce à son *id* et il ne doit y avoir qu'un seul élément pour un *id* donné.

Elle prend en paramètre l'*id* de l'élément que l'on recherche et retournera cet élément s'il a été trouvé.

```html
<p id="my-anchor">My content</p>
```

```js
const myAnchor = document.getElementById('my-anchor');
```

Récupérer le premier ou le dernier enfant d'un élément:

```js
const node = document.getElementById("myList2").firstElementChild;
const node = document.getElementById("myList2").lastElementChild;
```

- **document.getElementsByClassName()**

Fonctionnement similaire à la méthode précédente mais la recherche se fera sur la *class*.

Elle prend en paramètre la *class* des éléments à rechercher et retournera une liste des éléments correspondants. Pour retrouver les éléments on utilisera le code suivant :

```html
<div>
    <div class="content">Contenu 1</div>
    <div class="content">Contenu 2</div>
    <div class="content">Contenu 3</div>
</div>
```

```js
const contents = document.getElementsByClassName('content');
const firstContent = contents[0];
```

- **doucment.getElementsByName()**

Sélectionne les éléments à l'aide de la propriété name et les retourne dans une liste.

- **document.getElementsByTagName()**

Avec cette méthode, on recherche les éléments avec un nom de balise bien précis. Par exemple les liens *<a>*, les boutons *<button> etc…*

De la même manière que précédemment on récupère la liste des éléments correspondants :

```html
<div>
    <article>Contenu 1</article>
    <article>Contenu 2</article>
    <article>Contenu 3</article>
</div>
```

```js
const articles = document.getElementsByTagName('article');
const thirdArticle = articles[2];
```

- **document.querySelector()**

Cette méthode est plus complexe mais aussi beaucoup plus puissante car elle permet de faire une recherche complexe dans le DOM. Elle ne renvoit pas une liste mais le 1er élément trouvé qui correspond à la recherche ou *null* si aucun n'a été trouvé.

```html
<div id="myId">
    <p>
        <span><a href="#">Lien 1</a></span>
        <a href="#">Lien 2</a>
        <span><a href="#">Lien 3</a></span>
    </p>
    <p class="article">
        <span><a href="#">Lien 4</a></span>
        <span><a href="#">Lien 5</a></span>
        <a href="#">Lien 6</a>
    </p>
    <p>
        <a href="#">Lien 7</a>
        <span><a href="#">Lien 8</a></span>
        <span><a href="#">Lien 9</a></span>
    </p>
</div>
```

```js
const elt = document.querySelector("#myId p.article > a");
```

Cela fait une recherche **dans** l'élément ayant pour *id #myId,* les éléments de type *<p>* qui ont pour *class article* afin de récupérer le lien *<a>* qui est un enfant direct (obligatoirement direct).
La recherche retournera ici uniquement *Lien 6*.

Pour retourner une liste de résultats qui correspond à la recherche il faudra utiliser la fonction *document.querySelectorAll()* qui fonctionne de la même manière.

On peut également faire des recherches depuis un élément qui sont aussi des objets JavaScript avec leurs propriétés et fonctions. Et il en existe pour parcourir les enfants et les parents de chaque élément.

- element.children
- element.parentElement
- element.nextElementSibling / element.previousElementSibling (élément suivant ou précédent de même niveau que l'élément en question).

```
<div id="parent">
    <div id="previous">Précédent</div>
    <div id="main">
        <p>Paragraphe 1</p>
        <p>Paragraphe 2</p>
    </div>
    <div id="next">Suivant</div>
</div>
```

```
const elt =
document.getElementById('main');
```

- elt.children retournera les éléments de type *<p>* qui sont les enfants de l'élément *#main*
- elt.parentElement retournera la * * qui a l'*id parent*
- elt.nextElementSibling retournera l'élément qui a l'*id next*
- elt.previousElementSibling retournera l'élément qui a l'*id previous*

3. Modifier le DOM

Une fois que l'on a accéder aux éléments du DOM, on va pouvoir les modifier.

a. **Modifier le contenu d'un élément :**

Les deux principales propriétés sont :

- ***innerHTML***

https://developer.mozilla.org/fr/docs/Web/API/Element/innerHTML

Cette propriété définit ou renvoie la syntaxe HTML d'un élément. S'il y a des balises HTML, elles seront interprétées et non affichées.

Renvoie la propriété :
element.innerHTML

```
<p id="myP">I am a paragraph.</p>
<script>
let html =
document.getElementById("myP").innerHTML;
console.log(html) //"I am a paragraph."
</script>
```

Définit la propriété dans la page HTML :
element.innerHTML = *text*

```
<p id="demo"></p>
<script>
document.getElementById("demo").innerHTML =
"changed"; //ou
document.getElementById("demo").innerHTML =
"<p>Voici un exemple de contenu pour
<strong>innerHTML</strong></p>";
</script>
```

```
let elt = document.getElementById('main');
elt.innerHTML = "<ul><li>Elément 1</li><li>Elément
2</li></ul>";
```

L'élément qui a pour id 'main' aura un nouveau contenu qui deviendra :

```
<div id="main">
    <ul>
        <li>Elément 1</li>
        <li>Elément 2</li>
    </ul>
</div>
```

- ***textContent***

https://developer.mozilla.org/fr/docs/Web/API/Node/textContent

Cette propriété :

- Renvoie le contenu textuel d'un nœud et de tout ses descendants.
- Définit du contenu textuel. Attention : le texte ne sera pas interprété comme du texte HTML. Si on y inclus des balises HTML, elles seront visibles. S'il y a des descendants, ils seront tous supprimés par l'élément texte indiqué.

```html
<p id="mytxt">Exemple de texte</p>
<p id="demo"></p>
```

```html
<script>
let text =
document.getElementById("mytxt").textContent;
console.log(text); //Try it
document.getElementById("demo").innerHTML =
text;
</script>
```

- ***innerText***

Comme textContent sans les éventuels espaces supplémentaire et CSS cachés. Ne prends pas en compte les balises <script> et <style> contrairement à textContent.

b. Modifier le contenu d'un lien :

```html
<p><a id="myAnchor" target="_blank" href="https://www.microsoft.com">Microsoft</a></p>
<button onclick="myFunction()">Change the link</button>

<script>
function myFunction() {
  const element = document.getElementById("myAnchor");
  element.innerHTML = "W3Schools";
  element.href = "https://www.w3schools.com";
}
</script>
```

c. Modifier des classes :

On peut accéder à la liste des classes d'un élément avec la propriété *classList* qui fournit une série de fonction permettant de modifier cette liste de classe dont :

- add(string, [string, …]) : ajoute la ou les classes spécifiées
- remove(string, [string, …]) : supprime la ou les classes spécifiées
- contains(string) : vérifie si la classe spécifiées est contenue dans cet élément
- replace(lod, new) : remplace l'ancienne classe par la nouvelle classe
- toggle(string) : active et désactive le style

https://www.w3schools.com/jsref/prop_element_classlist.asp

```javascript
let elt = document.getElementById("myDiv");
elt.classList.add("nouvelleClasse");    // Ajoute la classe nouvelleClasse à l'élément
elt.classList.remove("nouvelleClasse"); // Supprime la classe nouvelleClasse que l'on
venait d'ajouter
elt.classList.contains("nouvelleClasse");   // Retournera false car on vient de la
supprimer
elt.classList.replace("oldClass", "newClass"): // Remplacera oldClass par newClass si
oldClass était présente sur l'élément
elt.classList.toggle("nouvelleClasse"); // Active et désactive la classe.
```

d. Changer les styles d'un élément :

Avec la propriété *style* on peut récupérer et modifier les différents styles d'un élément. *Style* est un objet qui a une propriété pour chaque style existant. Exemple :

```
elt.style.color = "#fff";      // Change la couleur du texte de l'élément à blanche
elt.style.backgroundColor = "#000"; // Change la couleur de fond de l'élément en noir
elt.style.fontWeight = "bold"; // Met le texte de l'élément en gras
```

e. Modifier les attributs :

On utilise la fonction *setAttribute* pour définir ou remplacer des attributs avec la syntaxe *setAttribute(name, value).* Il existe aussi *getAttribute*, *removeAttribute*

```
elt.setAttribute("type", "password");   // Change le type de l'input en un type password
elt.setAttribute("name", "my-password");    // Change le nom de l'input en my-password
myInput.setAttribute("type", "button");
myAnchor.setAttribute("href", "https://www.w3schools.com");
elt.getAttribute("name");               // Retourne my-password
```

Il est donc théoriquement possible d'ajouter des éléments de style avec cette méthode mais ce n'est pas une bonne pratique. Il faut utiliser la méthode précédente.

```
let answer = myButton.hasAttribute("onclick");
```

Retourne *true* si l'attribut existe *false* sinon.

f. Créer et ajouter de nouveaux éléments :

On utilise la fonction *createElement()* pour créer l'élément, puis la fonction *appendChild()* pour l'ajouter dans la page web. Pour du texte : *createTextNode()*

```
const para = document.createElement("p");
para.innerHTML = "This is a paragraph.";
document.getElementById("myDIV").appendChild(para);
```

Créer un bouton

```
const btn = document.createElement("button");
btn.innerHTML = "Hello Button";
document.body.appendChild(btn);
```

Créer un élément h1 avec du texte :

```
const h1 = document.createElement("H1");
const textNode = document.createTextNode("Hello World");
h1.appendChild(textNode);
document.body.appendChild(h1);
```

Ajouter un élément à une liste :

```
const node = document.createElement("li");
const textnode = document.createTextNode("Water");
node.appendChild(textnode);
document.getElementById("myList").appendChild(node);
```

insertBefore() pour ajouter un élément avant un autre

```
const node = document.getElementById("myList2").lastElementChild;
const list = document.getElementById("myList1");
list.insertBefore(node, list.children[0]);
```

g. Supprimer et remplacer des éléments :

- *replaceChild(new, old)*

```
// Create a new <li> element:
const element = document.createElement("li");
// Append text to the <li> element:
element.innerHTML = "Water";
// Get the <ul> element with id="myList":
const list = document.getElementById("myList");
// Replace the first child node with the new <li> element:
list.replaceChild(element, list.childNodes[0]);
```

- *removeChild(element)* supprimer le 1er élément d'une liste

```
const list = document.getElementById("myList");
list.removeChild(list.firstElementChild);
```

- *remove()*

```
const element = document.getElementById("demo");
element.remove();
```

4. Ecouter les événements

Un événement est une réaction à une action émise par l'utilisateur. Ex : clic sur un bouton, saisi d'un texte sur un formulaire. En JavaScript, un événement est représenté par un nom *click, mousemove* et une fonction que l'on nomme *callback*. C'est à nous de spécifier la fonction *callback*.

Pour cela on utilise la fonction *addEventListener(event, callback)*.

Pour écouter un click de souris sur un élément : *element.addEventListener('click', onClick)*. *onClick* correspond à la fonction que l'on va définir et qui sera appelé lorsque l'utilisateur cliquera sur l'élément. Si c'est un lien, le comportement par défaut se produira quand même : le lien sera ouvert par le navigateur ; si c'est un bouton de validation d'un formulaire, celui-ci sera envoyé.

```
const element = document.getElementById("myBtn");
element.addEventListener("click", myFunction);
function myFunction() {
  document.getElementById("demo").innerHTML = "Hello World";
}
```

Ou alors :

```
const element = document.getElementById("myBtn");
element.addEventListener("click", function() {
  document.getElementById("demo").innerHTML = "Hello World";
});
```

Pour éviter l'action par défaut, il faut rajouter un paramètre dans la fonction callback qui correspond au contenu de l'événement et qui permet d'utiliser quelques fonctions :

```
const element = document.getElementById("mon-lien");
element.addEventListener("click", function(event) {
  console.log(event); // Permet d'accèder aux informations de l'événement
  event.preventDefault();
  document.getElementById("demo").innerHTML = "Hello World";
});
```

Autre écriture :

```
const element = document.querySelector("#myId");
element.addEventListener('click', function(event) {
  console.log(event.target.value);
});
```

Ou

```
function myEventHandler(event) {
    // do something, probably with 'event'
}
const element = document.querySelector("#myId");
element.addEventListener('click', myEventHandler);
```

Il existe également la propriété suivante :

```
event.stopPropagation();
```

Elle permet d'éviter que l'événement (ici le click) ne se transmette aux éléments parents. Par exemple si un élément parent dispose d'une fonction suite à un click, et qu'un de ses enfants aussi. Si on click sur l'élément enfant, par défaut, la fonction de l'élément parent va s'exécuter également.

https://www.w3schools.com/jsref/met_element_addeventlistener.asp
https://www.w3schools.com/jsref/met_document_addeventlistener.asp
https://www.w3schools.com/js/js_htmldom_eventlistener.asp
https://www.w3schools.com/jsref/dom_obj_event.asp

Pour retirer le focus d'un élément :

```
HTMLElement.blur()
```

5. Récupérer des données utilisateurs avec les événements

La fonction callback reçoit un paramètre (précédemment event) qui contient les informations sur cet événement. Chaque événement implémente l'objet Event qui dispose de fonctions et propriétés comme :

- preventDefault() : empêche l'exécution de l'action par défaut de l'élément quand il reçoit l'événement
- stopPropagation() : empêche la propagation de l'événement vers les éléments parents.

https://www.w3schools.com/jsref/obj_event.asp

a. Détection du mouvement de la souris :

Il faut écouter le mouvement *mousemove*, qui fournir un objet de type *MouseEvent*. La fonction callback sera appelée avec un paramètre de type *MouseEvent* qui contient les données sur le mouvement de la souris :

- *clientX/clientY* : postion de la souris dans les coordonnées locales (contenu du DOM)
- *offsetX/offsetY* : position de la souris par rapport à l'élément sur lequel on écoute l'événement
- *pageX/pageY* : positon de la souris par rapport au document entier
- *screenX/screenY* : position de la souris par rapport à la fenêtre du navigateur
- *movementX/movementY* : position de la souris par rapport à la position de la souris lors du dernier événement *mousemove*.

https://www.w3schools.com/jsref/obj_mouseevent.asp

```
elt.addEventListener('mousemove', function(event) {
    const x = event.offsetX; // Coordonnée X de la souris dans l'élément
    const y = event.offsetY; // Coordonnée Y de la souris dans l'élément
});
```

b. Lire le contenu d'un champ texte :

L'événement *change* fonctionne avec *input, select et textarea*. L'événement est déclenché lorsque le champ perd le focus. C'est-à-dire lorsque l'utilisateur passe à autre chose en cliquant ailleurs et qu'il a fini sa saisie pour le champ. Cela fonctionne aussi avec les *checkbox* et les boutons *radio*.

https://www.w3schools.com/jsref/event_onchange.asp

Une fois qu'il a été modifié, on peut accéder à la valeur de l'élément cible par *event.target.value*.

```
//HTML
<input placeholder="Enter some text" name="name"/>
<p id="log"></p>
```

```
//JAVASCRIPT
const input = document.querySelector('input');
const log = document.getElementById('log');
input.addEventListener('change', updateValue);
function updateValue(e) {
  log.textContent = e.target.value;
}
```

```html
<label>Choose an ice cream flavor:
    <select class="ice-cream" name="ice-cream">
      <option value="">Select One …</option>
      <option value="chocolate">Chocolate</option>
      <option value="sardine">Sardine</option>
      <option value="vanilla">Vanilla</option>
    </select>
</label>
  <div class="result"></div>
```

```js
const selectElement = document.querySelector('.ice-cream');
selectElement.addEventListener('change', (event) => {
  const result = document.querySelector('.result');
  result.textContent = `You like ${event.target.value}`;
});
```

Si on veut pouvoir avoir accès à la valeur dès que l'utilisateur ajoute ou supprime une lettre, on utilise l'événement *input* qui fonctionne comme *change* mais qui est déclenché dès que le contenu du champ est modifié.

https://www.w3schools.com/jsref/event_oninput.asp

```js
input.addEventListener('input', function(event) {
    output.innerHTML = event.target.value;
});
```

Autres événements possibles :

https://www.w3schools.com/tags/ref_eventattributes.asp

https://www.w3schools.com/jsref/dom_obj_event.asp

https://www.w3schools.com/js/js_events_examples.asp

https://www.w3schools.com/js/js_htmldom_eventlistener.asp

https://www.w3schools.com/js/js_htmldom_events.asp

II. Communiquer via une API avec un service web

1. Comprenez ce que sont des API et un service web

Un service web est une application sur internet qui répond à un besoin : Google permet de faire des recherches, Facebook permet de garder contact avec ses amis. Le but d'un service web est donc de fournir un service à celui qui le demande. Et pour cela, il met à disposition une API.

Une API est une interface de communication entre une service web et le navigateur. Elle correspond à l'ensemble des demandes que l'on peut faire à un service web : des requêtes. Des protocoles de communication ont donc été mis en place afin de standardiser la communication entre services faisant la même chose.

Un protocole de communication permet de définir comment communiquer avec un service : définition des normes que tout le monde peut utiliser.

Le protocole HTTP est le protocole de communication utilisé par les services web. Il permet de charger des pages HTML, des style CSS, des polices, des images etc… mais aussi d'envoyer des formulaires, de récupérer et d'envoyer toute sorte de données depuis ou vers un serveur grâce à son API. Plusieurs informations se trouvent se trouvent dans une requête http :

- La méthode. L'action que l'on souhaite faire : récupérer une ressource, la supprimer etc...
    1. GET : permet de **récupérer** des ressources. Ex : temps sur un service météo.
    2. POST : permet de **créer** ou **modifier** une ressource. Ex : création d'un nouvel utilisateur sur une application
    3. PUT : permet de **modifier** une ressource. Ex : nom d'utilisateur que l'on vient de créer avec POST.
    4. DELETE : permet de supprimer une ressource. Ex : commentaire dans un fil de discussion
- L'URL. C'est l'adresse sur le service web que l'on souhaite atteindre. Identifiant unique afin que le service web comprenne ce que l'on veut.
- Les données. Si on veut enregistrer des données, il faut donc pouvoir les envoyer au service web.

Ensuite le service web va donc répondre avec entre autres :

- Les données. Les données que l'on a demandé : page HTML etc…
- Un code http
    1. 200 : tout est ok
    2. 400 : requête non conforme
    3. 401 : on doit être identifié pour faire cette requête
    4. 403 : on est bien authentifié mais pas autorisé à faire cette requête.
    5. 404 : ressource demandée n'existe pas
    6. 500 : erreur avec le service web
    7. Etc…

2. Récupérez des données d'un service web

Qu'est ce que Fetch ?

C'est un ensemble d'objets et de fonctions mis à disposition par le langage JavaScript afin d'exécuter des requêtes de manière asynchrone : cela permet d'exécuter du code (une requête ici) sans bloquer l'exécution de la page, en attendant la réponse du service web.

L'API Fetch va nous permettre d'exécuter des requêtes HTTP sans avoir besoin de recharger la page du navigateur. Cela permet :

- D'avoir un site plus réactif : on peut mettre à jour une partie du contenu sans recharger toute la page.
- Améliorer l'expérience utilisateur : nouveau contenu qui se charge au fur et à mesure qu'on le découvre.

Exemple d'une requête :

```
fetch("http://url-service-web.com/api/users");
```

Ce code permet d'envoyer une requête http de type GET au service web se trouvant à l'adresse http://url-service-web.com/api/users

Récupérer les données au format JSON :

Le plus couramment, les service web nous renvoie des données au format JSON (JavaScript Object Notation). C'est un format textuel se rapprochant de celui des objets en JavaScript.

En JavaScript :

```javascript
const obj = {
    name: "Mon contenu",
    id: 1234,
    message: "Voici mon contenu",
    author: {
        name: "John"
    },
    comments: [
        {
            id: 45,
            message: "Commentaire 1"
        },
        {
            id: 46,
            message: "Commentaire 2"
        }
    ]
};
```

Retranscrit en JSON :

```json
{
    "name": "Mon contenu",
    "id": 1234,
    "message": "Voici mon contenu",
    "author": {
        "name": "John"
    },
    "comments": [
        {
            "id": 45,
            "message": "Commentaire 1"
        },
        {
            id: 46,
            "message": "Commentaire 2"
        }
    ]
}
```

L'avantage du JSON est que le navigateur sait directement le lire et le transformer en objet JavaScript. Contrairement au format XML qui est plus verbeux et qu'il faut donc le parser pour en faire ce que l'on en veut. Le format JSON est donc aussi plus léger.

Récupérer le résultat de la requête :

```javascript
fetch("https://mockbin.com/request")
  .then(function(res) {
    if (res.ok) {
      return res.json();
    }
  })
  .then(function(value) {
    console.log(value);
  })
  .catch(function(err) {
    // Une erreur est survenue
  });
```

Pour cela, Fetch va nous envoyer une **Promise**. C'est un objet qui fournit une fonction *then()* qui sera exécutée quand le résultat aura été obtenu, et une fonction *catch()* qui sera appelée s'il y a une erreur qui est survenue.
C'est une requête de type GET (par défaut avec Fetch) à l'URL précisée.
La fonction *then()* récupère le résultat de la requête au format *json* en ayant vérifié au préalable que la requête s'était bien passée avec *res.ok*.
Le résultat *json* étant une *Promise*, on le retourne et on récupère sa vraie valeur dans la fonction *then()* suivante.

3. Validez les données saisies par vos utilisateurs

Il faut toujours contrôler les données saisies par les utilisateurs et ne jamais faire confiance. Ils peuvent être malveillants ou ne pas comprendre ce que l'on souhaite qu'ils fassent. Et cela peut avoir comme conséquences :

- Si l'utilisateur ne comprendre pas entre quelque chose qui ne convient pas, le service web ne fera donc pas ce qu'il faut et l'utilisateur ne comprendre pas pourquoi cela ne fonctionne pas. On risque donc de perdre cet utilisateur.
- Pire, un utilisateur pourrait attaquer notre service web en entrant des données malveillantes dans un champ d'entrée et prendre par exemple le contrôle de notre service, collecter les données des utilisateurs, se faire passer pour un administrateur etc…
- L'utilisateur pourrait faire planter l'application si une entrée incorrecte est saisie.

Valiser les données suite à des événements :

```
myInput.addEventListener('input', function(e) {
    var value = e.target.value;
    if (value.startsWith('Hello ')) {
        isValid = true;
    } else {
        isValid = false;
    }
});
```

Pour vérifier une regex:

```
function isValid(value) {
    return /^e[0-9]{3,}$/.test(value);
}
```

Validation directement dans le code HTML :

Cf doc HTML partie formulaire

1. Sauvegarder des données sur le service web

Envoyer des données, comment ça marche ?

Il va falloir faire une requête http en utilisant des méthodes comme POST et PUT. GET est seulement faite pour récupérer des données. Le fonctionnement de POST et PUT est assez similaire.

Envoyer des données avec une requête POST :

`fetch("http://url-service-web.com/api/users", {` `  method: “POST”,` `  headers: {` `'Accept': 'application/json',` `'Content-Type': 'application/json'` `},` `  body: JSON.stringify(jsonBody)` `});`	Comme on doit envoyer du JSON au service web, on doit d'abord transformer l'objet JavaScript en JSON. Pour cela on utilise la fonction *JSON.stringify(json)*. Il faut aussi prévenir le service web qu'il va recevoir du JSON. Cela se fait grâce aux ***headers***. Ce sont des en-têtes envoyés en même temps que la requête pour donner des informations sur celle-ci.

III. Parallélisez vos longues tâches avec la programmation asynchrone

1. Comprenez comment fonctionne l'asynchrone en JS

JavaScript est synchrone et mono-thread :

Cela signifie qu'il n'y a qu'un seul fil d'exécution et que les lignes de code sont exécutées les unes après les autres, en attendant la fin de l'exécution de la ligne précédente. Le code asynchrone va s'exécuter ligne après ligne mais la ligne suivante n'attendra pas que la ligne précédente ait fini son exécution.

Mais il est possible de faire de l'asynchrone en JavaScript mais l'exécution restera synchrone. Pour cela on va utiliser l'*event loop.*

L'event loop :

Lorsque l'on demande à exécuter une fonction de façon asynchrone, la fonction en question est placée dans une sorte de file d'attente qui va exécuter toutes les fonctions qu'elle contient les unes après les autres. C'est **l'*event loop***.

Jouer avec l'event loop :

La fonction *setTimeout* est la fonction la plus répandue lorsque l'on veut exécuter du code asynchrone sans bloquer le fil d'exécution. Cette fonction prend 2 paramètres :

- La fonction à exécuter de manière asynchrone (ajoutée donc à la file d'attente de l'*event loop*;
- Le délai, en ms, avant d'exécuter la fonction

`` setTimeout(function() { console.log("I'm here!") }, 5000); console.log("Where are you?"); ```	Dans cet exemple, le texte "*Where are you?*" s'affichera avant *I'm here* qui ne sera affiché que 5sec plus tard.

La fonction *setTimeout* retourne une valeur permettant d'identifer le code asynchrone que l'on veut exécuter. Il est possible de passer cet identifiant en paramètre de la fonction *clearTimeout* si on souhaite annuler l'exécution asynchrone de la fonction avant qu'elle ne soit exécutée.

https://www.w3schools.com/jsref/met_win_settimeout.asp

Les autres méthodes moins répandues voir peu utilisées :

- *setInterval* : elle fonctionne exactement comme *setTimeout* mais elle exécute la fonction passée en paramètre en boucle à une fréquence déterminée par le temps en ms passé en second paramètre. Pour arrêter la boucle, il faut passer la valeur de retour de *setInterval* en paramètre de la fonction *clearInterval*.
- *setImmediate* : cette fonction prend un seul paramètre : la fonction à exécuter de façon asynchrone. La fonction sera placée dans la file d'attente de l'event loop mais va passer devant toutes les autres fonctions, sauf certaines spécifiques au JavaScript : les événements, le rendu et l'I/O. Il existe aussi *nextTick* qui permet de court-circuiter tout le monde : attention !

Le cas de l'I/O :

Cela correspond aux événements liés à ***l'input*** et ***l'output*** : lecture/écriture des fichiers, requêtes http.

Lorsqu'on exécutait la fonction *fetch()* lors d'une requête http, elle ne bloque pas l'exécution du code. On n'attend pas que la requête soit envoyée et une réponse reçue avant d'exécuter le reste du code : c'est une ***fonction asynchrone***.

En plus *fetch()* retourne une *Promise* qui est une façon de faire de l'asynchrone car les fonction *then()* et *catch()* sont appelées plus tard lorsque le travail (la requête HTTP par exemple) est terminée.

De la même manière, tout ce que touche à I/O peut être exécuté de manière asynchrone. Heureusement car leur exécution peut prendre du temps.

2. Gérer du code asynchrone

a. La méthode Callbacks :

Une *callback* est une fonction que l'on définit. Il faut la passer en paramètre d'une fonction asynchrone. Une fois que la fonction asynchrone a fini sa tâche, elle va appeler la fonction *callback* en lui passant un résultat. Le code que l'on met dans la fonction *callback* sera donc exécuter de manière asynchrone.

Exemple : les événements :

```
element.addEventListener('click', function(e) {
    // Do something here ...
});
```

Dans cet exemple la fonction envoyée à *addEventListener* est une *callback*. Elle n'est pas appelée de suite, mais plus tard, dès que l'utilisateur clique sur l'élément. Ça ne bloque pas l'exécution du code, c'est donc de l'asynchrone. Les *callback* sont la **base de l'asynchrone** en JavaScript et sont très utilisées. Par exemple, la fonction que l'on passe en paramètre à *setTimeout* est une callback.

```
elt.addEventListener('click', function(e) {
    mysql.connect(function(err) {
        mysql.query(sql, function(err, result) {
            fs.readFile(filePath, function(err, data) {
                mysql.query(sql, function(err, result) {
                    // etc ...
                });
            });
        });
    });
});
```

Le problème peut venir de la lisibilité du code lorsque l'on se retrouve dans des situations où l'on va imbriquer plusieurs couches de *callback*. Le *callback hell* !
Ce code n'est pas facile à lire mais correspond à un cas concret des callbacks : quand l'utilisateur clique sur un élément, on ouvre une connexion MySQL, puis on récupère les données depuis la base, puis on lit un contenu dans un fichier et on fait une nouvelle requête.

Rien ne garantie cependant que tout se soit bien passé. Il faut donc un mécanisme pour savoir si une erreur est survenue.

Pour gérer les erreurs, la méthode la plus utilisée est de prendre 2 paramètres dans le *callback*. Le 2^{e} paramètre est notre donnée et le 1er est l'erreur. Si elle n'est pas *null* ou *undefined*, elle contiendra un message d'erreur indiquant qu'une erreur est survenue.

```
fs.readFile(filePath, function(err, data) {
    if (err) {
        throw err;
    }
    // Do something with data
});
```

Dans l'exemple ci-contre, on voit que la lecture d'un fichier avec le module *fs* peut nous retourner une erreur.

b. Promise :

Les *Promise* sont un peu plus complexes mais bien plus puissantes et faciles à lire que les callbacks.

Lorsque l'on exécute du code asynchrone, celui-ci va immédiatement nous retourner une "promesse" qu'un résultat sera envoyé prochainement. Cette promesse est en fait un objet *Promise* qui peut être *resolve* avec un résultat, ou *reject* avec une erreur. Lorsque l'on récupère une *Promise*, on peut utiliser sa fonction *then()* pour éxécuter du code dès que la promesse est résolue et sa fonction *catch()* pour exécuter du code dès qu'une erreur est survenue.

```
functionThatReturnsAPromise()
    .then(function(data) {
        // Do somthing with data
    })
```

Dans cet exemple, la fonction *functionThatReturnsAPromise* nous renvoie une *Promise*. On peut donc utiliser sa fonction *then()* en lui passant une fonction qui sera exécutée dès qu'un résultat sera

`.catch(function(err) {` `  // Do something with error` `});`	reçu. On peut utiliser sa fonction *catch()* en lui passant une fonction qui sera exécutée si une erreur est survenue.

On peut aussi chaîner les *Promise*. La valeur que l'on retourne dans la fonction que l'on passe à *then()* est transformée en une nouvelle *Promise* résolue, que l'on peut utiliser avec une nouvelle fonction *then()*. Si la fonction retourne une exception, alors une nouvelle *Promise* rejetée est créée que l'on peut intercepter avec la fonction *catch()*. Mais si la fonction que l'on a passée à *catch()* retourne un nouvelle valeur, alors on a à nouveau une *Promise* résolue que l'on peut utiliser avec une fonction *then()*.

```
returnAPromiseWithNumber2()
    .then(function(data) { // Data is 2
        return data + 1;
    })
    .then(function(data) { // Data is 3
        throw new Error('error');
    })
    .then(function(data) {
        // Not executed
    })
    .catch(function(err) {
        return 5;
    })
    .then(function(data) { // Data is 5
        // Do something
    });
```

Exemple qui montre comment on utilise des *Promise* pour chaîner du code asynchrone :
La fonction *returnAPromiseWithNumber2* renvoie une *Promise* qui va être résolue avec le nombre 2 :

- La première fonction *then()* va récupérer cette valeur ;
- Cette fonction retourne "2+1" ce qui crée une nouvelle *Promise* qui est immédiatement résolue avec "3".
- Puis dans le *then()* suivant on retourne une erreur.

Donc, le *then()* qui suit ne sera pas appelé et c'est le *catch()* suivant qui va être appelé avec l'erreur en question. Lui-même retourne une nouvelle valeur qui est transformée en *Promise* qui est immédiatement résolue avec la valeur "5". Le dernier *then()* va être exécuté avec cette valeur.

L'API *fetch()* utilise les *Promise* pour gérer les réponses aux requêtes HTTP comme vu précédemment.

Pour gérer les erreurs, on les intercepte avec la fonction *catch()* de la *Promise.*

c. Async/await :

Les mots clés *async* e *await* permettent de gérer le code asynchrone de manière plus intuitive, en bloquant l'exécution du code asynchrone jusqu'à ce qu'il retourne un resultat.

```
async function fonctionAsynchrone1() {/* code asynchrone */}
async function fonctionAsynchrone2() {/* code asynchrone */}

async function fonctionAsynchrone3() {
 const value1 = await fonctionAsynchrone1();
 const value2 = await fonctionAsynchrone2();
 return value1 + value2;
}
```

Ici il y a 3 fonctions asynchrones. Quand on utilise *async* et *await*, une fonction asynchrone doit avoir le mot clé *async* avant la fonction. Ensuite dans le code, on peut faire appel à des fonctions asynchrones et attendre leur résultat grâce au mot clé *await* que l'on met devant l'appel de la fonction.

Async et *await* utilise les *Promise*, la levée d'erreur se fait aussi par une exception. Pour intercepter cette erreur, par contre, il suffit d'exécuter le code asynchrone dans un bloc "*try {} catch (e) {}*", l'erreur étant envoyée dans le *catch.*

3. Parallélisez plusieurs requêtes http

But : Exécuter 2 requêtes GET en même temps (en parallèle), puis 1 requête POST une fois que les deux requêtes précédentes sont terminées.

a. Enchaîner les requêtes avec les callbacks :

```
var GETRequestCount = 0;
var GETRequestResults = [];

function onGETRequestDone(err, result) {
    if (err) throw err;

    GETRequestCount++;
    GETRequestResults.push(result);

    if (GETRequestCount == 2) {
        post(url3, function(err, result)
{
            if (err) throw err;

            // We are done here !
        });
    }
}

get(url1, onGETRequestDone);
get(url2, onGETRequestDone);
```

Nous avons accès à deux fonctions *get()* et *post()*. Elles font respectivement une requête GET et une requête POST et elles prennent en paramètre :

- L'URL de la requête
- Une callback à exécuter quand on a le résultat avec une variable d'erreur en 1er paramètre.

Pour exécuter 2 requêtes GET, on appelle deux fois la fonction *get()*. Cette fonction étant **asynchrone**, elle ne bloquera pas l'exécution du code. Donc l'autre fonction *get()* sera aussi appelée alors que la 1ère ne sera pas encore terminée.
Pour exécuter la requête POST une fois les deux requêtes terminées, la variable *GETRequestCount* est créée et incrémentée dans la fonction *callback* envoyée à *get(),* et si on atteint 2 (nombre de requêtes GET), on va exécuter la requête POST.

GETRequestResults sert à conserver les réponses des requêtes GET car on ne les a pas toutes en même temps.

b. Enchaîner les requêtes avec les Promises :

Grâce à la fonction *Promise.all*.

Nous avons accès à deux fonctions *get()* et *post()*. Elles font respectivement une requête GET et une requête POST quand leur passe l'URL de la requête. Ces fonctions retourneront une *Promise* avec le résultat de la requête.

```
Promise.all([get(url1), get(url2)])
    .then(function(results) {
        return Promise.all([results,
post(url3)]];
    })
    .then(function(allResults) {
        // We are done here !
    });
```

On utilise la fonction *Promise.all* qui prend en paramètre une liste de *Promise* (ou de simples valeurs qui sont transformées en *Promise* résolues), et qui permet de les exécuter en parallèle et de retourner une nouvelle *Promise* qui sera résolue quand toutes les promises seront résolues.
Ainsi la fonction *then()* recevra les résultats de toutes les *Promise* sous forme d'un tableau.
Afin d'exécuter la requête POST une fois que les requêtes GET sont terminées, on l'exécute avec la fonction *then().*

Dans la fonction *then()*, on fait encore une fois appel à la fonction *Promise.all* en lui passant les résultats des requêtes GET et de notre requête POST.

Comme *Promise.all* considère les simples valeurs comme des *Promise* résolues, cela permet dans le *then()* suivant de récupérer une liste qui contient les résultats des requêtes GET et POST :

allResults = [[getResult1, getResult2], postResult]

c. Enchaîner les requêtes avec les async/await :

Nous avons accès à deux fonctions *get()* et *post()*. Elles font respectivement une requête GET et une requête POST quand leur passe l'URL de la requête. Ces fonctions sont asynchrones avec le mot clé *async*.

`` async function requests() { var getResults = await Promise.all([get(url1), get(url2)]); var postResult = await post(url3); return [getResults, postResult]; } requests().then(function(allResults) { // We are done here ! }); ```	On utilise aussi la fonction *Promise.all* car c'est comme cela que l'on peut exécuter des fonctions asynchrones en parallèle : *async* correspond en arrière-plan à une *Promise.* On utilise *await* devant *Promise.all* afin d'attendre l'exécution des 2 requêtes GET. Puis on utilise *await* devant la requête POST afin d'attendre son résultat. Puis on renvoie un tableau avec tous les résultats.

Lorsque l'on appelle la fonction *requests()*, on utilise *then()* pour récupérer tous les résultats. On aurait pu utiliser aussi *await* au sein d'une autre fonction avec le mot clé *async*.

IV. Mettez en place les bons outils pour travailler

1. Optimisez votre code

a. Linter :

C'est un programme qui va analyser le code et détexter les erreurs de syntaxe, les varaibles non utilisées, les variables qui n'existent pas, la mauvaise organisation du code, le non respect des bonnes pratiques etc…

Comme le JavaScript est un language non compilé, on ne voit les erreurs qu'à l'exécution du code, c'est-à-dire que lorsqu'une fonction qui utilise une variable non définie est exécutée. La probabilité qu'une erreur "d'inatention" passe en production est donc inversement proportionnelle à son utilisation.

Le linter pourra donc aider à régler ces erreurs avant que les utilisateurs n'en subissent les conséquences. Les tests sont aussi un très bon moyen complémentaire au linter pour éviter les erreurs. Il pourra donc nous aider à optimiser le code et d'une grande aide pour définir les bonnes pratiques d'écriture du code lorsque l'on travaillera en équipe.

https://www.jslint.com/

https://eslint.org/

b. Minifier :

JavaScript est chargé par notre navigateur, c'est donc lui qui va demander au serveur notre code JavaScript pour l'exécuter. Moins le code sera lourd, plus il pourra donc être chargé rapidement.

Un minifier est donc un programme respnsable de la minification de notre code. Il va essayer de rendre le code le plus léger possible en retirant les espaces et les retours à la ligne inutiles, en renommant les variables avec des nombres plus courts, en supprimant le code non utilisé, en supprimant les commentaires, en optimisant certains bouts de code pour les réécrire avec une syntaxe plus légère.

https://github.com/srod/node-minify

https://github.com/mishoo/UglifyJS#readme

c. Bundler :

Il est aussi important de réduire au maximum le nombre de fichiers qui composent le code. C'est le rôle du *bundler*. Il va se charger de **packager** notre code pour qu'il tienne dans un seul fichier. On peut donc coder dans plusieurs fichiers pour plus de clarté dans notre code, mais le navigateur n'aura besoin de charger qu'un seul fichier lorsqu'un visiteur ira sur notre site.

https://webpack.js.org/

d. Transpiler :

Le langage JavaScript évolue. Le *transpiler* permet de coder dans la dernière version de JavaScript tout en étant compatible avec tous les navigateurs.

2. Gérez vos dépendances

3. Compilez et exécutez votre code

https://github.com/OpenClassrooms-Student-Center/5543061-ecrivez-du-javascript-pour-le-web-activity