# Introduction

Aujourd'hui nous avons un défi ! Nous devons travailler sur la partie sécurités de notre application Calorie Traqueur. Pour cela nous allons écrire ici toutes notre stratégie de sécurités et listé également les axes primordiaux de protection

Cette application aura pour but de venir en aide à toutes les personnes souhaitant venir à bout de leurs objectifs physiques.

## LES 3 GRANDS AXES

Les trois grands axes sur lequel nous allons nous concentrer aujourd'hui sont...

- La protection des données personnelles.
- L'identification.
- La protection de la partie client.

# LA MISE EN PLACE

## LA PROTECTION DES DONNÉES PERSONNELLES.

- Utiliser le principe de Moindre Privilège.
- HTTPS .
- CSP ( Permet de sécuriser les échanges avec l'API )
- Les précautions d'usage des bases de données de type IndexedDB
- Ne pas stocker d'information sensible dans les cookies.

## LA SÉCURITÉ LIÉ AU MOTS DE PASSE.

Nous allons dans un premier temps définir notre plan d'actions dans la partie authentification et mots de passe car c'est le premier élément à laquelle notre client utilisateur sera confronter.

- Interdire l'accès après trop de tentative échouer (8) de mots de passe pour contrer les attaques en ligne.
- Hacher le mot de passe ainsi que rajouter un salage pour contrer les attaques hors ligne.
- Utiliser des caractères spéciaux dans les mots de passe pour contrer les attaques par dictionnaire.
- Utiliser un canal sécuriser comme un flux encapsuler pour un protocole TLS pour contrer les attaques par homme du milieu.
- Complexité et longueur minimale requises.
- Notifications d'activités suspect.

## LA SÉCURITÉS LIÉ À L'EXPERIENCE UTILISATEUR.

Dans cette partie nous allons nous intéresser à la partie de la sécurité au moment où l'utilisateur interagie avec l'application.

- Défense en profondeur.
- Réduction de la surface d'attaques.
- HSTS ( Pour limiter les risques d'attaque ' man in the middle ' ).
- SOP
- CORS
- Les solutions face au vulnérabilités XSS.
- Utiliser le X-XSS protection
- Préférer l'utilisation de l'API Fetch à XMLHttpRequest
- Audit.

## LA DESCRIPTION DES ELEMENTS.

### PARTIE 1 :

### Principe de moindre privilège :

Ce principe vise à n'octroyer aux éléments et acteurs du système que les permissions strictement nécessaires pour fonctionner, ceci afin de limiter le risque de vol, d'altération ou de destruction de données dans le cas de compromission d'un ou plusieurs éléments.

- à la conception de l'application, prévoir autant de rôles que de besoins d'accès aux données (lecture seule, écriture, etc.).
- limiter les permissions d'accès aux Application Programming Interfaces (APIs) du navigateur pour une application web.

- limiter les permissions de l'utilisateur applicatif sur le système de fichiers.

## HTTPS :

La mise en place de HTTPS a pour objectif :

- de garantir, autant que possible, l'authenticité du site consulté.
- de garantir également l'intégrité et la confidentialité des données échangées en bloquant les
- attaques de type Man-In-The-Middle (écoute, interception ou modification des échanges à la
- volée par des tiers, à l'insu de l'utilisateur).

## CSP :

Permet de définir une stratégie de contrôle des accès aux ressources atteignables d'un site web donné par l'application de restrictions sous forme de liste d'autorisations. La maîtrise de l'ensemble des ressources récupérées par un site web permet de réduire le risque d'apparition et l'exploitabilité de vulnérabilités XSS
définition de la liste des ressources autorisées peut être effectuée en utilisant.

- l'en-tête HTTP dédié, Content-Security-Policy, dans la réponse http.
- la balise équivalente, <meta http-equiv="Content-Security-Policy">, dans la réponse HTML.

### Exemple

La stratégie CSP suivante demande au navigateur d'accepter uniquement des ressources servies depuis la même `Origin` que la page actuelle via une connexion sécurisée. En particulier, CSP permet la maîtrise des ressources dont l'activation n'est pas contrainte par la SOP (cf. section 5.1.1).

```
default-src 'self' https: ;
```

La stratégie CSP suivante demande au navigateur d'accepter uniquement des ressources servies depuis la même `Origin` que la page actuelle. Elle autorise l'inclusion de code `JavaScript` *inline*, c'est-à-dire directement dans le HTML ainsi que l'utilisation de fonctions d'évaluation de code (`eval()`, `Function()`, etc.), ce qui n'est pas une bonne pratique.

```
default-src 'self' ; script-src 'unsafe-inline' 'unsafe-eval' ;
```

## Les précautions d'usage des bases de données de type IndexedDB

- **Ne pas stocker des informations sensibles dans les bases de données IndexedDB.**

  Les bases de données IndexedDB ne doivent être utilisées que pour le stockage de données non sensibles et pour lesquelles la perte ou la divulgation sera sans conséquence,

pour mettre en cache l'état d'une application web par exemple.

- **Proscrire l'usage de l'API Web SQL Database**

  Interdire l'usage de l'API Web SQL Database, désormais obsolète.

## Ne pas stocker d'informations sensible dans les cookies.

Dans le cadre de la défense en profondeur et à l'exception des jetons de session, il est recommandé de ne pas stocker des informations sensibles dans les cookies. Leur utilisation n'est souhaitable que pour le stockage temporaire d'informations de faible volume, pour lesquelles la perte ou la divulgation sera sans conséquence.

## PARTIE 3 :

## Défense en profondeur :

Le principe général de défense en profondeur consiste à mettre en œuvre plusieurs mesures de protection indépendante en face de chaque menace envisagée. Il est plus facile d'appliquer ce principe si le système à sécuriser est composé d'unités distinctes, aux interactions bien définies et possédant leurs propres mécanismes de sécurité. La défense en profondeur demande que soient mises en œuvre les mesures de protection nécessaires et disponibles au niveau de chaque unité

## Reduction de la surface d'attaque :

La réduction de la surface d'attaque consiste à ne pas exposer des services, accès et autres points
d'entrée s'ils ne sont pas indispensables. Ce principe appliqué au développement logiciel demande
que soit limitée la présence de composants logiciels dont l'usage n'est pas strictement nécessaire

## HSTS :

Il est nécessaire de mettre en oeuvre HSTS afin de limiter les risques d'attaque de type Man-In-The-Middle dus à des accès non sécurisés générés par les utilisateurs ou par un attaquant

Le mise en oeuvre de HSTS se fait par la transmission d'un en-tête HTTP lors de l'accès au site en HTTPS pour assurer son intégrité. Par défaut, la stratégie HSTS d'un site est enregistrée par le navigateur lorsqu'il est visité pour la première fois

**Exemple**

Demander au navigateur d'utiliser exclusivement HTTPS pour se connecter au site visité et à ses sous-domaines, pour une durée d'un an :

```
Strict-Transport-Security: max-age=31536000; includeSubDomains;
```

## SOP :

L'objectif de Same-Origin Policy est de fournir un cadre de contrôle des interactions possiblement effectuées par les éléments embarqués dans une page web. SOP est une contrainte
implémentée par tous les navigateurs du marché. Cette contrainte ne signifie pas que toutes les ressources
doivent provenir d'un même Origin, mais impose des restrictions dans la communication entre composants lorsque ceux-ci ont des Origins différentes.

| Exemple `Origin` | |
|---|---|
| **URL complète** | `Origin` |
| `http://www.exemple.org:8080`<br>`http://www.exemple.org:8080/fichiers/page1.html`<br>`http://www.exemple.org:8080/api/time` | `http://www.exemple.org:8080` |

## CORS :

Il est parfois nécessaire de contourner la SOP (stratégie de sécurité par défaut du navigateur) afin de permettre l'appel de ressources en dehors de l'Origin telles que peuvent en fournir des services web tiers de météo ou d'actualités par exemple. La méthode utilisée dans ce cas est nommée Cross-Origin Resource Sharing. Cette méthode est normalisée et vient en remplacement de plusieurs
autres techniques jusqu'alors utilisées mais considérées comme dangereuses et limitées telles que la proxyfication ou l'utilisation de JSON with Padding. Il s'agit de l'interface structurée de programmation pour les documents.

CORS est un standard qui permet la définition explicite d'un contrat entre le serveur web et le navigateur qui spécifie les conditions d'acceptation d'échanges Cross-Origin. La négociation de ce contrat a lieu par l'intermédiaire d'en-têtes HTTP:

## Les solutions face au vulnérabilités XSS :

Il existe de nombreux scénarios d'attaques mettant en jeu une vulnérabilité XSS. Par exemple, s'il est possible d'injecter du contenu dans une page au travers d'une variable

GET, un attaquant peut inciter (par exemple au moyen d'un courrier électronique trompeur) une victime à cliquer sur un lien fabriqué dans l'objectif d'appeler une page au contenu vulnérable et insérer dans celle-ci un script malveillant. L'attaquant pourra alors contrôler le navigateur de la victime qui pense pourtant visiter un site de confiance. Il est ainsi en mesure, par exemple, de voler la session de la victime et d'usurper son identité sur le site.

- **UTILISER L'API DOM A BON ESCIENT**

  Toute intervention sur le contenu client doit être réalisée via l'API DOM. Il est recommandé de ne pas utiliser, ou à défaut de contrôler l'usage de méthodes et propriétés qui effectuent des substitutions ou modifications de contenu dans un contexte à même d'altérer le comportement de l'application web.

### Exemple

Dans les 2 exemples suivants, on peut observer que l'utilisation côté client de *Template Strings* ES6 présente une vulnérabilité XSS par rapport à l'utilisation de la balise HTML `<template>`.

```
var ligne = `
  <th>${meteo.city.name}</th>
  <td>${meteo.weather.temperature}</td>
`; // Risque XSS
var node = document.createElement("tr");
node.innerHTML = ligne; // Risque XSS
document.getElementById('display-grid').appendChild(node);
```

Listing 5.1 – Utilisation de *Template Strings* ES6

```
<template id="ligne">
  <tr>
    <th id="ville">$ville</th>
    <td id="temperature">$température</td>
  </tr>
</template>

var template = document.querySelector("#ligne");
var ligne = template.cloneNode(true);
ligne.content.querySelector("#ville").textContent = meteo.city.name;
ligne.content.querySelector("#temperature").textContent = meteo.weather.
  temperature;
document.getElementById('display-grid').appendChild(ligne.content);
```

Listing 5.2 – Utilisation de HTML `<template>`

En effet, si l'on considère la valeur frauduleusement modifiée de l'un des champs :

```
meteo.city.name = "<iframe src=\"http://autre.site.web\"></iframe>"
```

Ce contenu sera interprété dans le cas de l'utilisation de *Template Strings* ES6. Tandis qu'il sera affiché, et non interprété, dans le cas de l'utilisation de la balise HTML `<template>` avec `.textContent`.

- **Dissocier clairement la composition des pages web ( HTML, CSS, JAVASCRIPT ).**

Il est recommandé de dissocier clairement les données (JSON), la structure (HTML), le style (CSS) et la logique (JavaScript) d'une page web afin de réduire le risque d'occurrence de vulnérabilités XSS.

**Attention**

De manière générale, la méthode consistant à générer dynamiquement une page sur le serveur par inclusion d'un modèle et fusion avec des données présente plus de risques que d'effectuer le rendu directement sur le navigateur, puisque cela ouvre la porte à des injections côté serveur lors du parsing du *template*. Il n'est pas recommandé de transmettre un contenu tiers ou de faible confiance directement dans la page HTML ou de contourner, côté serveur, les mécanismes de sécurité implémentés par les navigateurs.

## Utiliser de l'API Fetch

L'API Fetch se présente comme une alternative plus flexible à l'utilisation de XHR notamment par l'utilisation de promesses (Promises) JavaScript.

L'option crédenciales permet de contrôler l'émission de cookies, certificats clients, et autres données
d'authentification HTTP. Par défaut, sa valeur est same-origin. Les autres valeurs possibles sont omit et include.

**Exemple**

```
const url = '/api/ville/Paris';
const params = {
  method: 'GET',
  mode: 'same-origin',
  credentials: 'omit',
  cache: 'default',
  referrerPolicy: 'no-referrer',
  redirect: 'error',
  integrity: 'sha256-abcdef1234567890'
};

fetch(url, params)
  .then( res => res.json() )
  .then( data => handleMeteoData(data) );

// avec async/await, équivalent à :

(async () => {
  const meteoDataRaw = await fetch(url, params);
  const meteoDataJson = await meteoDataRaw.json();
  handleMeteoData(meteoDataJson);
})();
```

Listing 5.14 – Exemple API Fetch - Conditions météo sur Paris

## AUDIT :

C'est une bonne pratique de conception qui permet de pouvoir anticiper de futures activités malveillantes du web en mettant une place un processus manuelles ou automatisées permettant l'analyse à la chaine d'intégration continue (ex : détection de dépendances vulnérables, outils d'analyses statique et dynamique ).