Mazurek Pierre Esirem 3A IE

DM Developpement Web

Table des matières

Table des figures

Exercice n°1

On importe tout d'abord le fichier « etiquettes_formulaire.html ». Ce fichier, ouvert dans un navigateur, donne le résultat suivant :

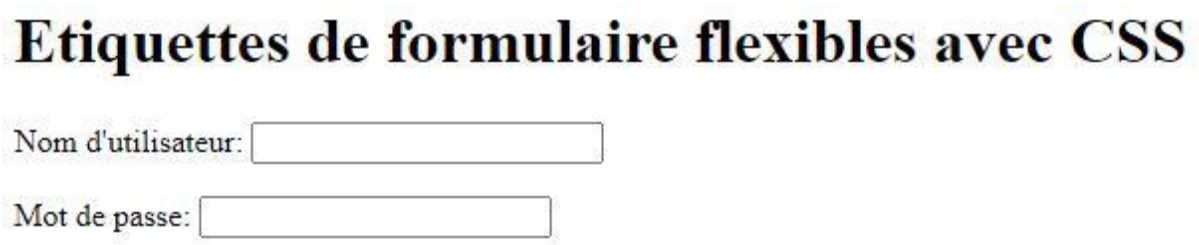


Figure 1 : etiquettes_formulaire.html sous navigateur

On va à présent créer un fichier CSS et le lier à ce fichier HTML. Pour ce faire, on va ajouter la ligne suivante dans le fichier HTML :

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <link rel="stylesheet" type="text/css" href="etiquettes_formulaire.css">
5      <title>Example etiquettes de formulaire</title>
6      <meta name="viewport" content="width=device-width" />
7  </head>
8  <body>
9  <h1>Etiquettes de formulaire flexibles avec CSS</h1>
10 <form action="" method="get">
11 <p><label for="username">Nom d'utilisateur:</label> <input type="text" /></p>
12 <p><label for="password">Mot de passe:</label> <input type="password" /></p>
13 </form>
14 </body>
15 </html>
16
```

Figure 2 : ajout de lien du CSS au fichier *etiquettes_formulaire.html*

On lui indique de se lier au fichier « etiquettes_formulaire.css ». Ce fichier CSS contient le code suivant :

```
1  h1 {font-size: 18pt;}
2  label
3  {
4      width: 100px;
5      text-align: right;
6      display: inline-block;
7      vertical-align: baseline
8  }
```

Figure 3 : code du fichier *etiquettes_formulaire.css*

Voici la signification de ces différentes lignes :

On indique que chaque titre de type *h1* a une taille de 18 points.

Pour chaque éléments de type *Label*, ce dernier prend une largeur de 100 pixels, le texte est aligné à droite, il est affiché sur la même ligne que le bloc auquel il est lié. Sa ligne de base est alignée avec celle de son parent, c'est-à-dire la box.

Après avoir éditer le fichier CSS, on rouvre le fichier HTML avec un navigateur :

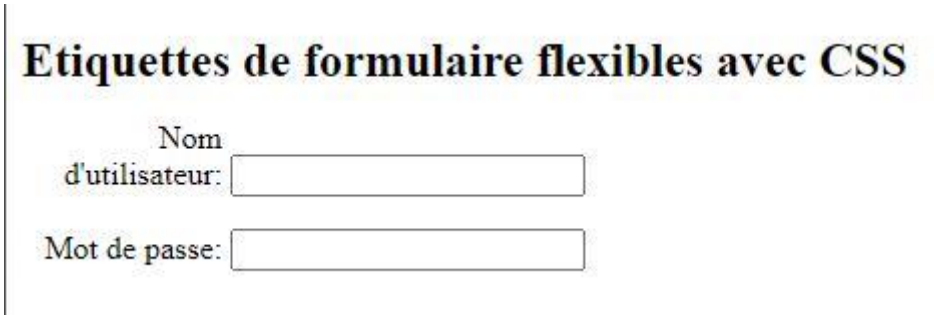


Figure 4 : réouverture de *etiquettes_formulaire.html* après modification du fichier CSS

On constate que le fichier CSS a pour effet de rendre la police du titre moins grosse. De plus, les deux box sont alignées. Les caractères présents à droite de ces box sont eux alignés sur la droite sans empiéter sur ces box.

On va ensuite éditer le code CSS pour que les étiquettes soient affichées au-dessus des champs du formulaire pour une taille de fenêtre inférieure à 480 pixels.

Voici le nouveau code du fichier CSS :

```
1   h1 {font-size: 18pt;}
2
3   label
4       {
5       width: 100px;
6       text-align: right;
7       display: inline-block;
8       vertical-align: baseline
9       }
10
11
12
13  @media only screen and (max-width: 480px) {
14      label
15      {
16      width: 100px;
17      text-align: right;
18      display: block;
19      vertical-align: baseline
20      }
21  }
22
```

Figure 5 : ajout des requêtes média dans *etiquettes_formulaire.html*

On conserve le contenu de la classe *label.* Cependant, on ajoute une requête média qui stipule que pour une fenêtre de taille inférieure à 480 pixels, la ligne *display : inline-block* est modifiée en *display : block ;* ce qui aura simplement pour effet d'afficher l'élément *label* au-dessus de son parent en l'occurrence le champ de formulaire.

On actualise le fichier HTML. Au-dessus d'une taille de fenêtre de 480 pixels, on constate que l'affichage est le même que précédemment.

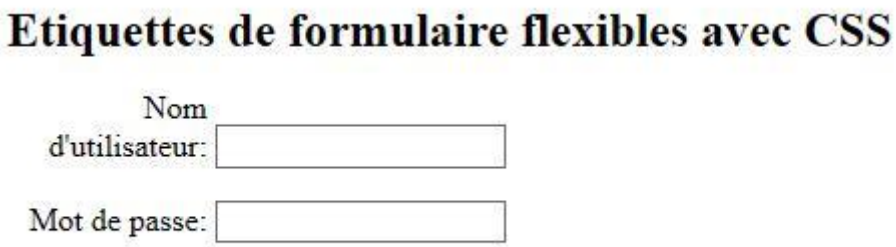


Figure 6 : affichage de *etiquette_formulaire.html* dans une fenêtre de largeur > à 480 pixels

Cependant, en réduisant la taille de fenêtre en dessous de 480 pixels, on obtient l'affichage suivant :

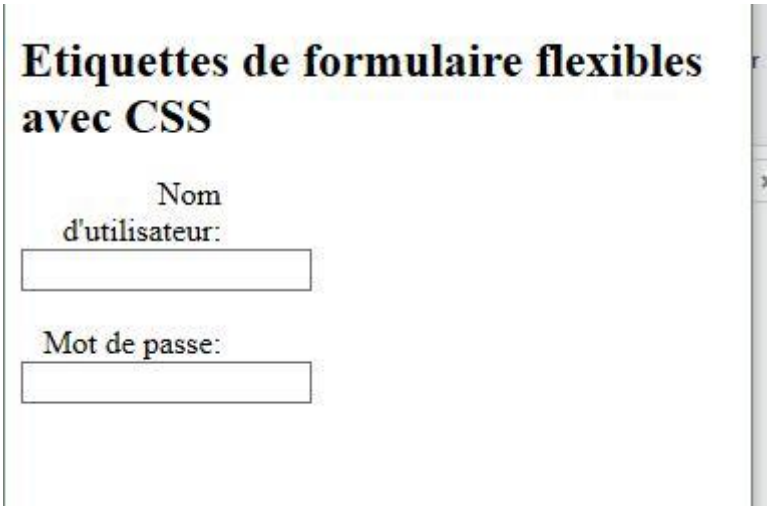


Figure 7 : affichage de *etiquette_formulaire.html* dans une fenêtre de largeur inférieure à 480 pixels

Les éléments *label* s'affichent bien au-dessus de leur parent.

Exercice n°2

On affiche le fichier « Media_queries.html » dans le navigateur :

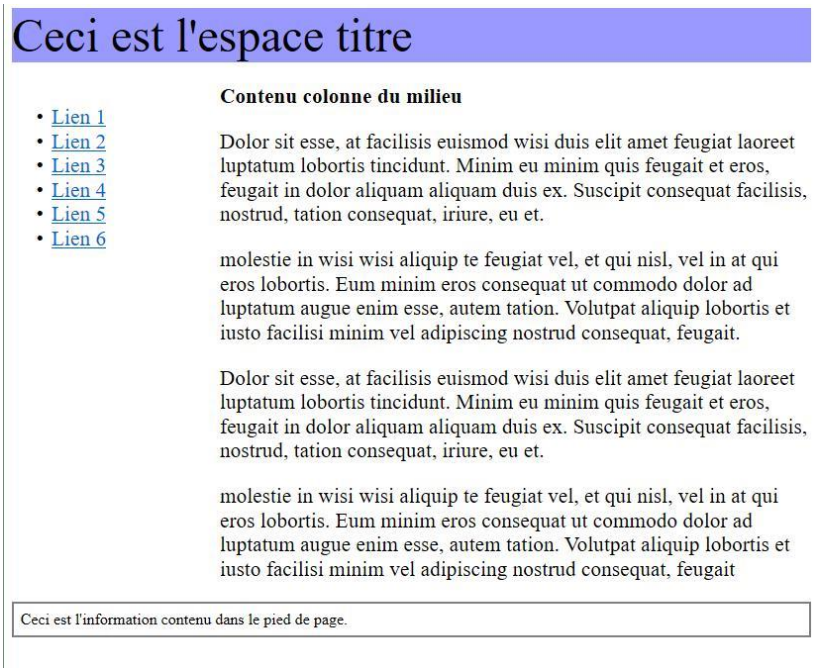


Figure 8 : affichage de *media_queries.html* sous navigateur

On constate que, lorsqu'on réduit la taille de la fenêtre, les champs s'adaptent à la taille de cette dernière (sauf la liste des liens). Arrivé à un certain point, la taille de la fenêtre est trop petite pour contenir tous les éléments et ces derniers sont hors-affichage. La fenêtre n'a pas de taille minimale.

Le fichier CSS présente le code suivant :

```css
#titrePage {
    font-size:36pt;
    font-family:"Times New Roman", Times, serif;
    background-color:#9999FF
}

#container {
    font-size: 16pt;
    position: relative;
    width: 100%;
}

#colonneG {
    width: 200px;
    height: 100%;
    float:left;
}

#contenuPage {
    margin-left: 210px;
}

#footer {
    border: 2px gray solid;
    padding: 5pt;
    margin-top: 5pt;
}
```

Figure 9 : code de *media_queries.css*

On va ensuite modifier le fichier CSS pour provoquer l'affichage demandé. Au-dessus d'une largeur de fenêtre de 800 pixels, l'affichage ne change pas. On effectue donc une requête média avec en taille minimum 801 pixels avec le même code :

```css
@media screen and (min-width: 801px){
    #titrePage {
        font-size:36pt;
        font-family:"Times New Roman", Times, serif;
        background-color:#9999FF
    }

    #container {
        font-size: 16pt;
        position: relative;
        width: 100%;
    }

    #colonneG {
        width: 200px;
        height: 100%;
        float:left;
    }

    #contenuPage {
        margin-left: 210px;
    }

    #footer {
        border: 2px gray solid;
        padding: 5pt;
        margin-top: 5pt;
    }
}
```

Figure 10 : ajout d'une requête média pour une largeur supérieure à 800 pixels dans le fichier CSS

Pour parmétrer l'affichage de la fenêtre en-dessous de 800 pixels, on effectue une nouvelle requête média dans le même fichier CSS pour une largeur maximale de la fenêtre de 800 pixels avec le code suivant :

```css
@media screen and (max-width: 800px){
    #titrePage {
        color: #FFFFFF;
        text-align: right;
        font-size:36pt;
        font-family:"Times New Roman", Times, serif;
        background-color:#808101
    }

    #container {
        font-size: 16pt;
        width: 100%;
    }

    #colonneG ul {
        margin-left: 0;
        padding-left: 0;
    }


    #colonneG li {
        display:inline;
        list-style: none;
    }

    #colonneG {
        float: none;
        display: inline;
        text-align: left;
        width: 200px;
        height: 100%;
    }

    #contenuPage {
    }

    #footer {
        border: 2px gray solid;
        padding: 5pt;
        margin-top: 5pt;
    }
}
```

Figure 11 : ajout d'une requête média pour une largeur inférieure à 800 pixels dans le fichier CSS

Détaillons les modifications apportées au code original :

Dans la div *#titrepage*, on a changé la couleur de l'arrière-plan, on a mis la couleur de police en blanc et on a précisé un alignement du texte à droite.

On a ajouté du code pour la classe *ul* fille de la div *#colonneG*. On précise que le *padding* est à 0 ce qui a pour effet de supprimer la tabulation habituelle appliquée à la liste.

On ajoute également du code pour la classe *il* fille de la div *#colonneG*. On indique que l'affichage doit être en *inline*, ce qui signifie que la liste va être affichée de manière horizontale. Le paramètre *list-style : none* supprime les puces normalement affichées avant les éléments de la liste.

Venons-en à la classe *#colonneG*. On passe le paramètre *float* à *none* ce qui a pour effet d'afficher la liste au-dessus du texte. Avec *text-align : left*, on précise que les éléments de la liste doivent être alignés sur la gauche. Avec *display : inline*, on permet aux éléments de la ligne d'être affichés sur une seule et même ligne.

Dans la classe *#contenuPage*, on va simplement enlever la marge pour qu'il n'y ait aucun décalage du texte.

On réaffiche ensuite le fichier HTML à l'aide d'un navigateur :

Au-dessus d'une largeur de 800 pixels, rien ne change. Mais en-dessous de cette largeur on obtient l'affichage suivant :

Figure 12 : affichage de *media_queries.html* pour une largeur de fenêtre inférieure à 800 pixels

Cela correspond bien à l'affichage attendu.

Exercice n°3

Pour cet exercice, on utilise le fichier HTML « detection_de_fonctionnalites.html » que l'on va lier à deux fichier JS. On ajoute les balises suivantes :

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="modernizr.js"></script>
        <html class="no-js">
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width" />
</head>
<script type="text/javascript" src="function.js"></script>
<p>Cet exemple illustre comment utiliser Modernizr pour détecter la prise en charge de fonctionnalités dans un navigateur donné.
    L'ensemble du code est exécuté côté client, et utilise à la fois du JavaScript et du CSS pour détecter la prise en charge
    de fonctionnalités individuelles, au lieu d'employer du code côté serveur.</p>
<h3>Fonctionnalités JavaScript</h3>
<p>Geolocalisation: <span id="geoloc"></span></p>
<p>Evenements tactiles: <span id="touch"></span></p>
<h3>Fonctionnalités HTML5</h3>
<p>SVG: <span id="svg"></span></p>
<p>Canvas: <span id="canvas"></span></p>
<h3>Fonctionnalités CSS3</h3>
<p class="animtest">animations CSS prises en charge</p>
<p class="noanimtest">animations CSS non prises en charge</p>
</body>
</html>
```

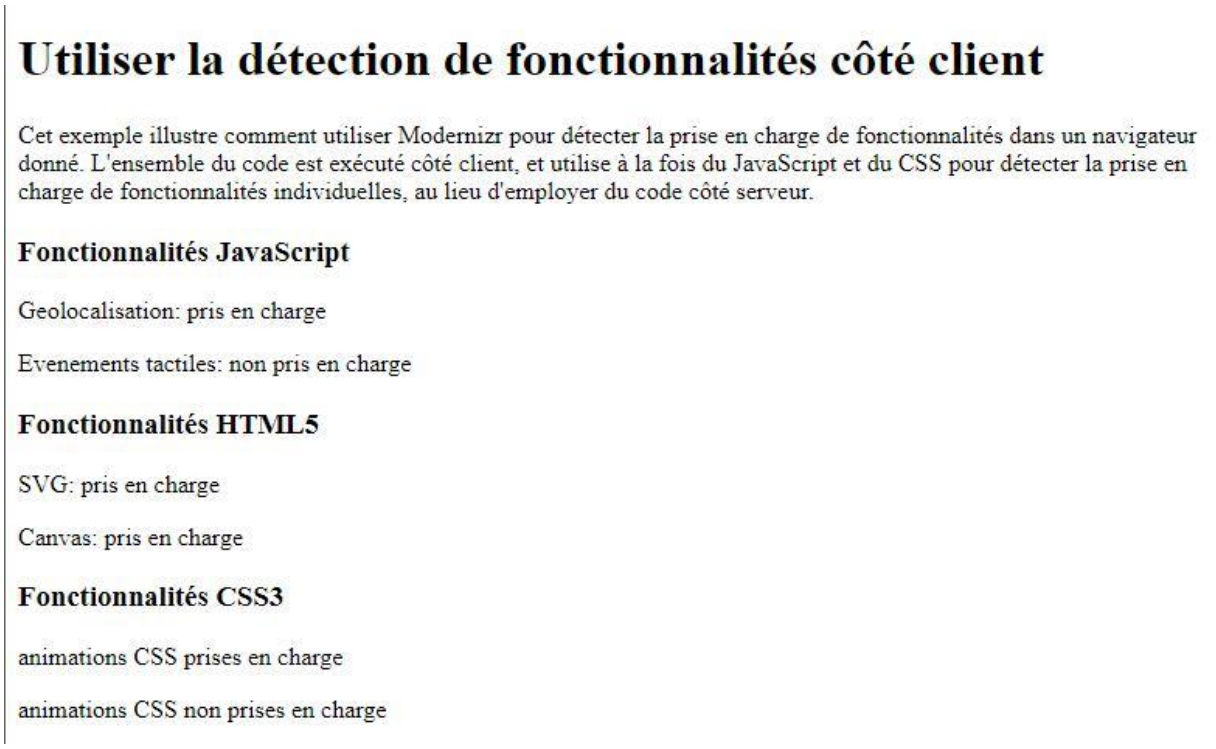
Figure 13 : liaison de *detection_de_fonctionnalites.html* aux deux fichiers JS

Ici, on a ajouté le chemin vers le fichier JS *Mordernizr* recupéré sur le site https://modernizr.com/ ainsi que la balise *<html class= « no-js »>* dans l'en-tête du fichier. Dans le corps du fichier HTML, on lie ce dernier au fichier « function.js » qui présente le code suivant :

```
function testFonctionnalites()
{
    document.querySelector("#geoloc").innerHTML = Modernizr.geolocation ? "pris en charge" : "non pris e
    document.querySelector("#touch").innerHTML = Modernizr.touch ? "pris en charge" : "non pris en charg
    document.querySelector("#svg").innerHTML = Modernizr.svg ? "pris en charge" : "non pris en charge";
    document.querySelector("#canvas").innerHTML = Modernizr.canvas ? "pris en charge" : "non pris en cha
window.onload = testFonctionnalites;
```

Figure 14 : code de *function.js*

Lorsqu'on ouvre le fichier HTML sur un navigateur, on obtient la fenêtre suivante :



Utiliser la détection de fonctionnalités côté client

Cet exemple illustre comment utiliser Modernizr pour détecter la prise en charge de fonctionnalités dans un navigateur donné. L'ensemble du code est exécuté côté client, et utilise à la fois du JavaScript et du CSS pour détecter la prise en charge de fonctionnalités individuelles, au lieu d'employer du code côté serveur.

Fonctionnalités JavaScript

Geolocalisation: pris en charge

Evenements tactiles: non pris en charge

Fonctionnalités HTML5

SVG: pris en charge

Canvas: pris en charge

Fonctionnalités CSS3

animations CSS prises en charge

animations CSS non prises en charge

Figure 15 : ouverture du fichier *detection_de_fonctionnalites.html* après modification du fichier JS

Le code du fichier « function.js » va en fait appeler les fonctions instanciées dans le fichier « mordernizr.js ». Ces fonctions consistent en fait à rechercher si des fonctionnalités précises sont prises en charge par le PC sur lequel est situé le fichier « modernizr.js ». « Function.js » va ensuite questionner ces fonctions pour voir leur résultats. Il va ensuite fournir une réponse « pris en charge » ou « non pris en charge » en fonction du résultat récupéré. Il va ensuite envoyer cette réponse au fichier HTML qui va l'afficher.

Ici, 2 fonctionnalités de type JavaScript sont testées : *géolocalisation* et *évènements tactiles*. Deux fonctionnalités HTML5 sont également testées : *SVG* et *Canvas*.

On crée ensuite le fichier « detection.css » avec le code suivant :

```
1   .animtest, .noanimtest { display: none;}
2   .cssanimations .animtest { display:block;}
3   .no-cssanimations .noanimtest { display:block;}
```

Figure 16 : code du fichier *detection.css*

On lie ce fichier CSS avec le fichier HTML de la manière suivante :

```
<!DOCTYPE html>
<html>
<head>
    <script type="text/javascript" src="modernizr.js"></script>
        <html class="no-js">
    <link rel="stylesheet" type="text/css" href="detection.css">
    <meta charset="UTF-8"/>
    <meta name="viewport" content="width=device-width" />
</head>
<body>
<script type="text/javascript" src="function.js"></script>
<h1>Utiliser la détection de fonctionnalités côté client</h1>
<p>Cet exemple illustre comment utiliser Modernizr pour détecter la prise en charge de fonctionnalités
dans un navigateur donné.
    L'ensemble du code est exécuté côté client, et utilise à la fois du JavaScript et du CSS pour
    détecter la prise en charge
    de fonctionnalités individuelles, au lieu d'employer du code côté serveur.</p>
<h3>Fonctionnalités JavaScript</h3>
<p>Geolocalisation: <span id="geoloc"></span></p>
<p>Evenements tactiles: <span id="touch"></span></p>
<h3>Fonctionnalités HTML5</h3>
<p>SVG: <span id="svg"></span></p>
<p>Canvas: <span id="canvas"></span></p>
<h3>Fonctionnalités CSS3</h3>
<p class="animtest">animations CSS prises en charge</p>
<p class="noanimtest">animations CSS non prises en charge</p>
</body>
</html>
```

Figure 17 : liaison de *detection_de_fonctionnalites.hmtl* avec le fichier CSS

On ouvre ensuite le fichier HTML avec un navigateur et voici ce qu'on obtient :

Utiliser la détection de fonctionnalités côté client

Cet exemple illustre comment utiliser Modernizr pour détecter la prise en charge de fonctionnalités dans un navigateur donné. L'ensemble du code est exécuté côté client, et utilise à la fois du JavaScript et du CSS pour détecter la prise en charge de fonctionnalités individuelles, au lieu d'employer du code côté serveur.

Fonctionnalités JavaScript

Geolocalisation: pris en charge

Evenements tactiles: non pris en charge

Fonctionnalités HTML5

SVG: pris en charge

Canvas: pris en charge

Fonctionnalités CSS3

animations CSS prises en charge

Figure 18 : réouverture de *detection_de_fonctionnalites.html* après modification du fichier CSS

Détaillons le code du fichier CSS. Celui-ci concerne 4 classes : *.animtest*, *.noanimtest*, et 2 classes du fichier « modernizr.js » qui sont *.cssanimations* et *.no-cssanimations*. La classe *.cssanimations* correspond à un résultat positif de test de la prise en charge des animations CSS3 et la classe *.no-cssanimations* correspond à un résultat négatif. Dans le cas où la classe *.cssanimations* est effective (que les animations CSS3 sont prises en charge), alors on affiche le block contenu dans la classe .animtest qui est : « animations CSS prises en charge ». Dans le cas où la classe *.no-cssanimations* est effective (que les animations CSS3 ne sont pas prises en charge), alors on affiche le block contenu dans la classe *.noanimtest* qui est : « animations CSS non prises en charge ». Dans le cas une classe n'est pas effective, la ligne *display : none* indique que l'on n'affiche pas le bloc. Ce code permet simplement de vérifier si les animations CSS sont prises en charge ou non par le PC.

Exercice n°4

On récupère le fichier « stockage_local.html » qu'on ouvre ensuite avec un navigateur :

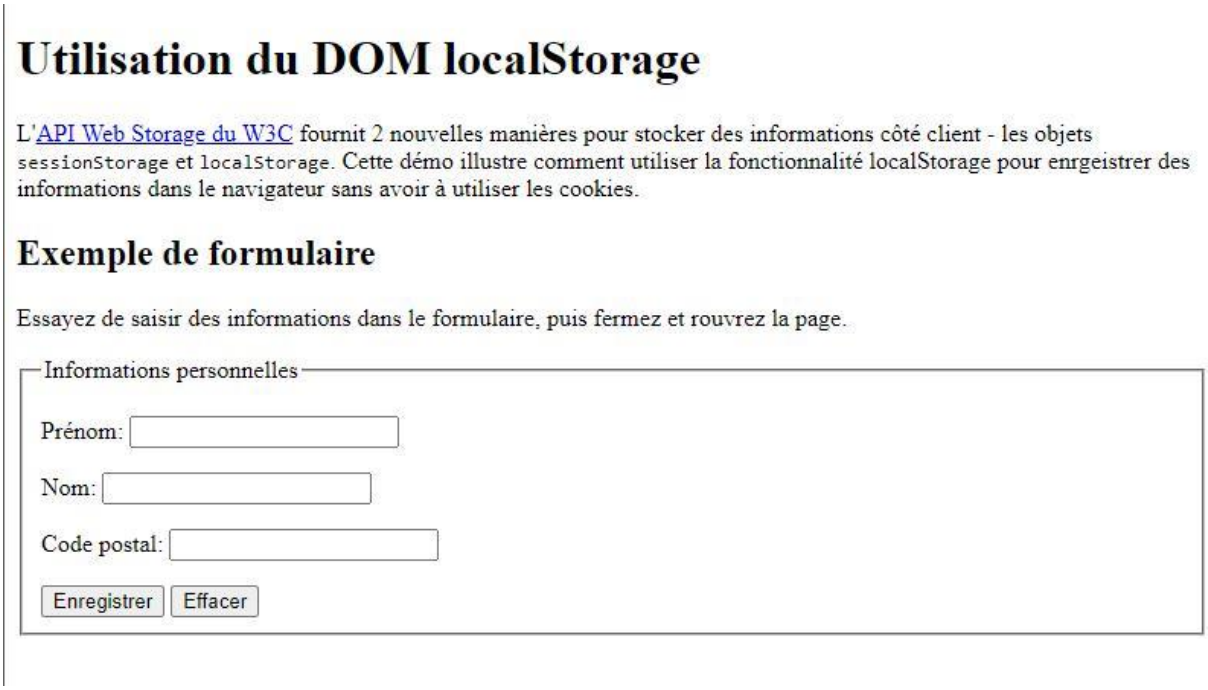


Figure 19 : affichage de *stockage_local.html* sous navigateur

On constate qu'il s'agit d'un formulaire que l'on peut remplir et qui présente deux options, enregistrer et annuler. Pour l'instant, l'option enregistrer ne fonctionne pas car en actualisant le fichier, on perd les données rentrées précédemment. De même pour l'option Effacer qui n'a aucun effet. Il faut donc définir les deux fonctions JavaScript correspondant au stockage des données entrées dans le formulaire et à leur effacement.

On va pour cela créer le fichier « stockage.js » que l'on appelle du fichier HTML de cette manière :

```html
<!DOCTYPE html>
<html>
    <head>
        <title>API Web Storage du W3C</title>
        <meta charset="UTF-8"/>
        <meta name="Viewport" content="width=device-width" />
    </head>
    <body>
        <script type="text/javascript" src="stockage.js"></script>
        <div>
            L'<a href="http://www.w3.org/TR/webstorage/">API Web Storage du W3C</a>
            fournit 2 nouvelles manières pour stocker des informations côté client -
            les objets <code>sessionStorage</code> et <code>localStorage</code>. Cette démo illustre
            comment utiliser la fonctionnalité localStorage pour enrgeistrer des informations dans le
            navigateur
            sans avoir à utiliser les cookies.</div>
        <h2>Exemple de formulaire</h2>
        <p>Essayez de saisir des informations dans le formulaire, puis fermez et rouvrez la page.</p>
        <div>
            <form action="" method="post" id="infoform">
                <fieldset name="LocalStorage" id="ls">
                <legend>Informations personnelles</legend>
                <p><label id="fnLabel" for="firstName">Prénom: </label>
                <input id="firstName" name="firstName" /></p>
                <p><label id="lnLabel" for="lastName">Nom: </label>
                <input id="lastName" name="lastName" /></p>
                <p><label id="pclabel" for="postCode">Code postal: </label>
                <input id="postCode" name="postCode" /></p>
                <input type="button" value="Enregistrer" onclick="storeLocalContent(
                    document.querySelector('#firstName').value,
                    document.querySelector('#lastName').value,
                    document.querySelector('#postCode').value
                )">
                <input type="button" value="Effacer" onclick="clearLocalContent()">
                </fieldset>
            </form>
        </div>
    </body>
</html>
```

Figure 20 : liaison de *stockage_local.html* avec le fichier JS

On commence le code du fichier JS comme ceci :

```js
function initialize() { /*code à ajouter par la suite */ }
window.onload = initialize;
```

Figure 21 : code de *stockage.js*

Ces lignes de code permettent d'instancier la fonction *initialize* et de l'appliquer sur la fenêtre du fichier HTML auquel le fichier JS est lié.

On ajoute ensuite la ligne suivante :

```js
function initialize()
{
    var bSupportsLocal = (('localStorage' in window) && window['localStorage'] !== null);
}
window.onload = initialize;
```

Figure 22 : contenu de la fonction *initialize* du fichier CSS

Cette ligne permet d'initialiser une variable *bSupportsLocal*, qui vérifie s'il y a du stockage disponible pour le navigateur.

On ajoute ensuite les lignes suivantes :

```
function initialize()
{
    var bSupportsLocal = (('localStorage' in window) && window['localStorage'] !== null);
    if (!bSupportsLocal) {
        document.getElementById('infoform').innerHTML = "<p>Désolé, ce navigateur ne supporte pas l'API Web Storage du W3C.</p>";
        return;
    }


}
window.onload = initialize;
```

Figure 23 : contenu de la fonction *initialize* du fichier CSS (partie 2)

Ici on indique que s'il n'y a pas de stockage disponible, on affiche le message « Désolé, ce navigateur ne supporte pas l'API Web Storage du W3C ».

A présent, on écrit le code suivant :

```
function initialize()
{
    var bSupportsLocal = (('localStorage' in window) && window['localStorage'] !== null);
    if (!bSupportsLocal) {
        document.getElementById('infoform').innerHTML = "<p>Désolé, ce navigateur ne supporte pas l'API
        return;
    }

    if (window.localStorage.length != 0) {
    document.getElementById('firstName').value = window.localStorage.getItem('firstName');
    document.getElementById('lastName').value = window.localStorage.getItem('lastName');
    document.getElementById('postCode').value = window.localStorage.getItem('postCode');
    }


}
window.onload = initialize;
```

Figure 24 : contenu de la fonction *initialize* du fichier CSS (partie 3)

Cette dernière stipule que s'il y a du stockage disponible, alors on stocke les trois informations, à savoir les champs *firstName*, *lastName* et *postCode* qui sont des variables du fichier HTML contenant les éléments entrés dans les champs du formulaire.

On termine l'édition de ce code par l'ajout des lignes suivantes :

```
function initialize()
{
    var bSupportsLocal = (('localStorage' in window) && window['localStorage'] !== null);
    if (!bSupportsLocal) {
        document.getElementById('infoform').innerHTML = "<p>Désolé, ce navigateur ne supporte pas l'API
        return;
    }

    if (window.localStorage.length != 0) {
    document.getElementById('firstName').value = window.localStorage.getItem('firstName');
    document.getElementById('lastName').value = window.localStorage.getItem('lastName');
    document.getElementById('postCode').value = window.localStorage.getItem('postCode');
    }


}
window.onload = initialize;

function storeLocalContent(fName, lName, pCode)
{
    window.localStorage.setItem('firstName', fName);
    window.localStorage.setItem('lastName', lName);
    window.localStorage.setItem('postCode', pCode);
}
```

Figure 25 : ajout de la fonction *storeLocalContent* au fichier CSS

Ici on crée une autre fonction nommée *storeLocalContent* qui va s'activer lorsqu'on appuie sur le bouton « enregistrer » comme précisé dans le fichier HTML (cf figure 20, ligne 29). Cette fonction a pour effet d'affecter aux champs du formulaire les valeurs stockées au préalable. Lorsqu'on actualise la page, les champs du formulaire restent complétés.

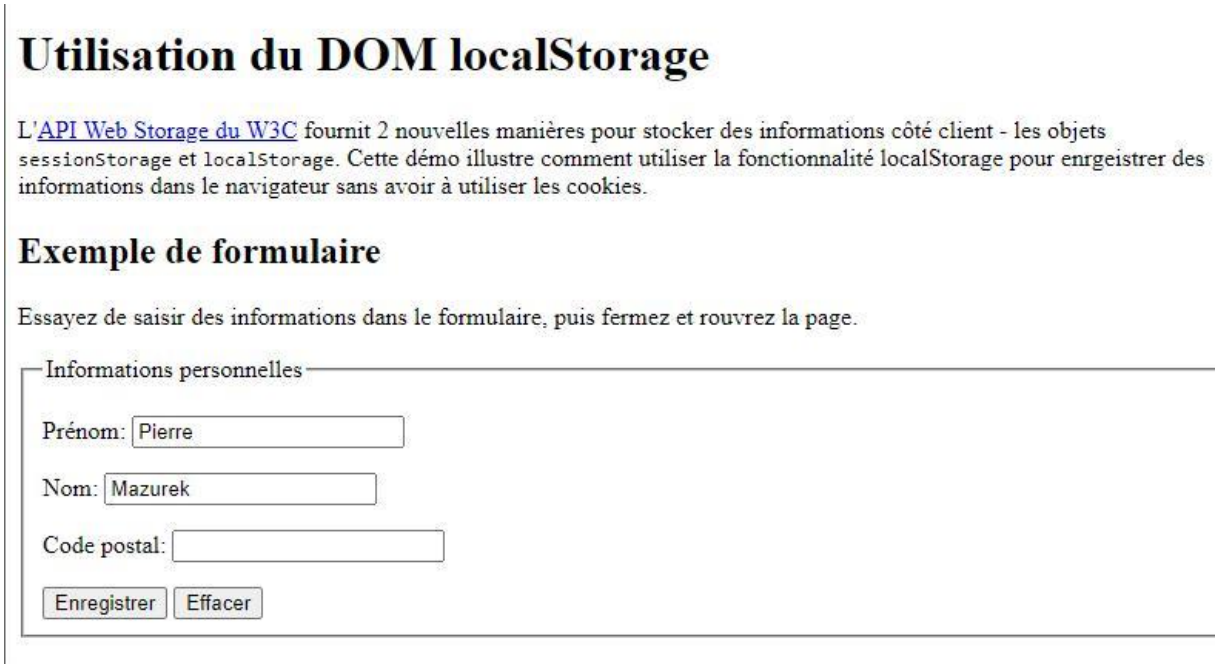


Figure 26 : réaffichage de *stockage_local.html* après l'ajout de la méthode *StoreLocalContent*

On va maintenant créer la fonction *clearLocalContent* qui va permettre de vider le stockage et qui sera affecté au bouton « Effacer ».

```
22  function clearLocalContent (){
23      window.localStorage.removeItem('firstName');
24      window.localStorage.removeItem('lastName');
25      window.localStorage.removeItem('postCode');
26  }
```

Figure 27 : code de la fonction *clearLocalContent* du fichier CSS

On va simplement utiliser l'attribut *removeItem* sur chaque champ du formulaire. Ils n'auront plus de valeur et seront vides.

Sur le fichier HTML, la fonction *clearLocalContent* est déjà appelée par le bouton « Effacer » à la ligne 34.

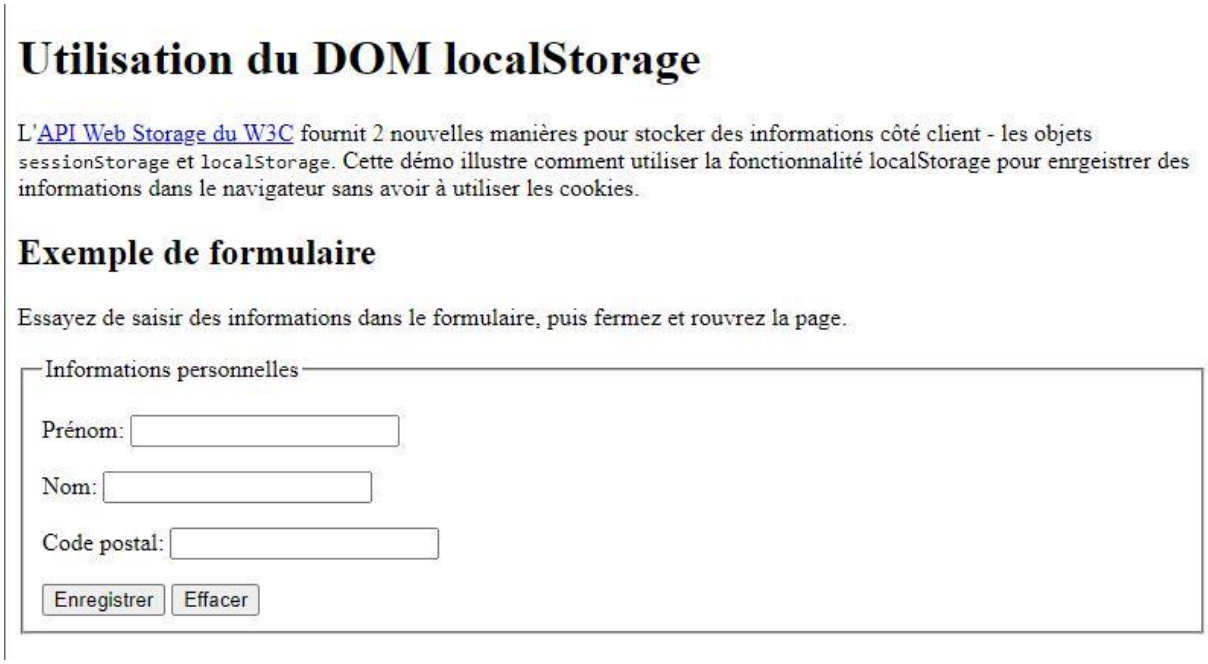


Figure 28 : réaffichage de *stockage_local.html* après l'ajout de la méthode *clearLocalContent*

Si on sauvegarde les données, qu'on navigue vers une autre page puis qu'on revient vers cette même page juste après, les données sont conservées.

Exercice n°5

Pour cet exercice, on aura besoin de lier deux fichiers JS au fichier HTML, « Modernizr.js » que l'on a déjà appelé dans un précédent exercice et le fichier « geo.js » avec lequel on va utiliser les fonctions de l'autre fichier JS. On lie donc le fichier HTML de cette manière :

```
<!DOCTYPE html>
<head>
    <script type="text/javascript" src="modernizr.js"></script>
    <html class="no-js">
    <meta name="viewport" content="width=device-width" />
    <meta charset="UTF-8"/>
</head>
<body>
    <script type="text/javascript" src="geo.js"></script>
<p>Cet exemple illustre comment utiliser la fonction de géopositionnement du terminal mobile.</p>
<h2>Données de position:</h2>
<p>Longitude: <span id="longitude"></span></p>
<p>Latitude: <span id="latitude"></span></p>
<p>Précision: <span id="precision"></span></p>
<p>Altitude: <span id="altitude"></span></p>
<p>Précision altitude: <span id="precisionAltitude"></span></p>
<p>Cap: <span id="cap"></span></p>
<p>Vitesse: <span id="vitesse"></span></p>
<p>Distance de l'ESIREM: <span id="distance"></span></p>
</body>
</html>
```

Figure 29 : liaison de *geopositionnement.html* avec les deux fichiers JS

En affichant le fichier HTML avec un navigateur, voici ce que l'on obtient :

Figure 30 : affichage de *geopositionnement.html* sous navigateur

Il semble que ce fichier est censé donner tous les paramètres indiqués sur celui-ci, se rapportant à la position GPS de l'appareil dans lequel il est ouvert. On entre le code suivant pour le fichier « geo.js ».

```
1
2   function getLocation() { /*code à ajouter par la suite */ }
3
4   window.onload = getLocation;
```

Figure 31 : code de *geo.js*

La fonction *getLocation* sera appelée à l'ouverture de la fenêtre du fichier HTML sous un navigateur.

On ajoute ensuite les ligne suivantes :

```
1
2 ▼ function getLocation() {
3       if (Modernizr.geolocation) {
4           navigator.geolocation.getCurrentPosition(geoSuccess, geoError);
5       }
6   }
7
8   window.onload = getLocation;
```

Figure 32 : contenu de la fonction *getLocation* du fichier *geo.js*

La condition *if* que nous posons est en fait une vérification. On veut tout d'abord savoir s'il existe la fonction *geolocation* dans le fichier « Mordernizr.js ». On utilise ensuite la fonction *getCurrentPosition*, attribut de cette fonction. On demande une réponse à cela, *geoSuccess* ou *geoError*.

On définit ensuite les fonctions correspondant à ces deux réponses :

```
function getLocation() {
    if (Modernizr.geolocation) {
        navigator.geolocation.getCurrentPosition(geoSuccess, geoError);
    }
}

window.onload = getLocation;

function geoSuccess(positionInfo) {
    document.getElementById("longitude").innerHTML = positionInfo.coords.longitude;
    document.getElementById("latitude").innerHTML = positionInfo.coords.latitude;
    document.getElementById("precision").innerHTML = positionInfo.coords.accuracy;
    document.getElementById("altitude").innerHTML = positionInfo.coords.altitude;
    document.getElementById("precisionAltitude").innerHTML = positionInfo.coords.altitudeAccuracy;
    document.getElementById("cap").innerHTML = positionInfo.coords.heading;
    document.getElementById("vitesse").innerHTML = positionInfo.coords.speed;
}

function geoError(positionError) {
    if (errorInfo.code == 1) alert("L'utilisateur ne souhaite pas partager sa position");
    else if (errorInfo.code == 2) alert("Impossible de déterminer une position");
    else if (errorInfo.code == 3) alert("Délai de recherche de position trop long");
}
```

Figure 33 : ajout des fonctions *geoSuccess* et *geoError* dans le fichier *geo.js*

Dans la fonction *geoSuccess*, on va simplement faire prendre aux variables du fichier HTML, les éléments correspondant, résultats renvoyés par la fonction *getCurrentPosition* du fichier « Modernizr.js ». Ces variables seront ensuite affichées par le fichier HTML.

Dans la fonction *geoError*, on va, en fonction du numéro de code d'erreur reçu, afficher le message d'erreur correspondant.

On ouvre ensuite le fichier HTML à l'aide d'un navigateur. Tout d'abord, on a un message nous demandant si on accepte que le fichier utilise notre géolocalisation :

Figure 34 : demande d'accès à la position

On autorise bien évidemment l'accès. Voici ce qu'on obtient ensuite :

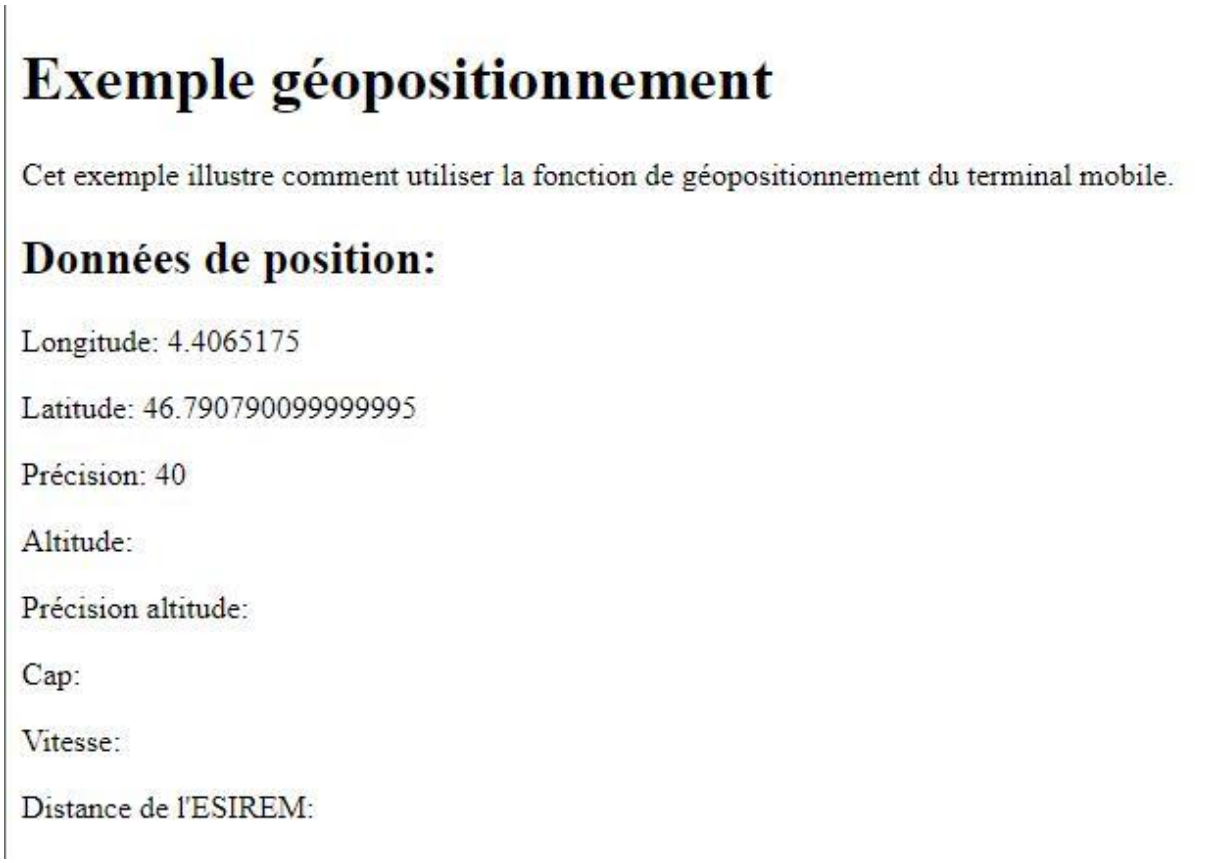


Figure 35 : réaffichage de *geopositionnement.html* après édition du fichier *geo.js*

On obtient une valeur pour la longitude, latitude et précision. Etant sur un ordinateur assez ancien, il est certainement normal de ne pas avoir de valeur pour les autres éléments.

Cherchons à connaître notre position par rapport à ESIREM. On va pour cela éditer la méthode *CalculDistance.* Cette dernière prend en paramètre deux objets et calcul la distance qui les sépare à partir de leur latitude et longitude. Toutefois, pour faire le calcul d'une distance, il faut que les latitudes et longitudes soient en radians. Or ici, elles sont en degrées. On crée donc la fonction degreesEnRadians pour effectuer la bonne conversion. Voici le code de ces deux méthodes :

```
function calculDistance(startCoords, destCoords) {
    var startLatRads = degreesEnRadians(startCoords.latitude);
    var startLongRads = degreesEnRadians(startCoords.longitude);
    var destLatRads = degreesEnRadians(destCoords.latitude);
    var destLongRads = degreesEnRadians(destCoords.longitude);

    var Radius = 6371; // rayon de la Terre en km
    var distance = Math.acos(Math.sin(startLatRads) * Math.sin(destLatRads) + Math.cos(startLatRads) * Math.cos(destLatRads) * Math.cos(startLongRads - destLongRads)) * Radius; return distance;
}

function degreesEnRadians(degrees) {
    radians = (degrees * Math.PI)/180;
    return radians;
}
```

Figure 36 : code des méthodes *CalculDistance* et *degreesEnRadians*

Pour pouvoir effectuer le calcul et afficher le résultat au bon endroit, on écrit les lignes suivantes :

```
27 ▼ function geoSuccess(positionInfo) {
28       let positionEsirem = {'latitude': 47.3121519, 'longitude': 5.0039356};
29       let distanceEsirem = calculDistance(positionInfo.coords, positionEsirem);
30       document.getElementById("longitude").innerHTML = positionInfo.coords.longitude;
31       document.getElementById("latitude").innerHTML = positionInfo.coords.latitude;
32       document.getElementById("precision").innerHTML = positionInfo.coords.accuracy;
33       document.getElementById("altitude").innerHTML = positionInfo.coords.altitude;
34       document.getElementById("precisionAltitude").innerHTML = positionInfo.coords.altitudeAccuracy;
35       document.getElementById("cap").innerHTML = positionInfo.coords.heading;
36       document.getElementById("vitesse").innerHTML = positionInfo.coords.speed;
37       document.getElementById("distance").innerHTML = distanceEsirem;
38   }
39
```

On commence d'abord par initialiser l'objet *Esirem* avec sa latitude et longitude. On crée ensuite un autre objet qui n'est autre que le résultat de la fonction *CalculDistance* entre la position d'Esirem et notre position actuelle. On affiche le tout à la bonne place grâce à la dernière ligne.

Une fois ouvert dans le navigateur, voici ce que nous affiche le fichier HTML :

Exemple géopositionnement

Cet exemple illustre comment utiliser la fonction de géopositionnement du terminal mobile.

Données de position:

Longitude: 4.4066358

Latitude: 46.7906525

Précision: 53

Altitude:

Précision altitude:

Cap:

Vitesse:

Distance de l'ESIREM: 73.55496648825115

Figure 377 : réaffichage du fichier HTML après le calcul de la distance avec Esirem

Rappelons que la distance est ici exprimée en km. Etant actuellement au Creusot, la distance paraît tout à fait plausible avec l'école. La fonction marche !!