Développement Web

GENTON Emilie – TD1 – TP2
ANNEE 2019 - 2020

Table des matières

Table des illustrations

Exercice 1 : ETIQUETTES FORMULAIRES

Dans ce premier exercice, nous récupérons le code fourni appelé etiquettes_formulaire.html. Puis nous créons un fichier CSS qui permettra de mettre en forme ce dernier. Bien sûr, pour lier les deux fichiers, nous n'oublions pas d'ajouter l'instruction ci-dessous dans la balise *head* de l'HTML :

```
<link rel="stylesheet" type="text/css" href="etiquettes_formulaire.css">
```

Figure 1 : ajout d'un lien vers le CSS dans le HTML

Avant d'ajouter le fichier CSS avec les instructions fournies, nous avions un formulaire comme ceci :

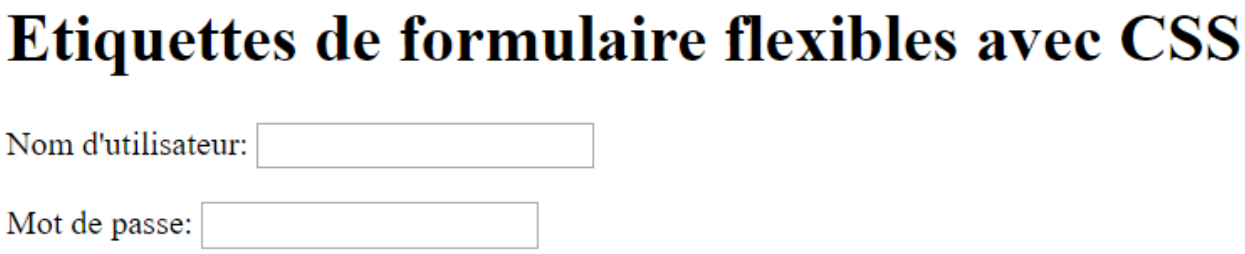

Figure 2 : Formulaire initial

On ajoute donc dans le CSS, les instructions suivantes :

```
h1 {
    font-size: 18pt;
}
label {
    width: 100px;
    text-align: right;
    display: inline-block;
    vertical-align: baseline;
}
```

Figure 3 : code donnée pour le titre H1

Pour expliquer les lignes ci-dessus, on peut dans un premier temps voir que la première ligne s'appliquera seulement pour les éléments dans le HTML de type *h1*. Les autres instructions s'appliqueront sur les éléments *label.* La première ligne permet de modifier la taille de la police. Les labels auront une largeur de 100%, et donc prendront toute la largeur de l'écran. Le texte des labels s'alignera à droite. *Vertical-align : baseline,* permet d'aligner le label verticalement au milieu de son conteneur. Cette dernière fonctionne avec *display : inline-block*. En effet, elle permet de respecter les marges supérieures et inférieures, ici, pour nos *h1*. Ainsi on obtient :

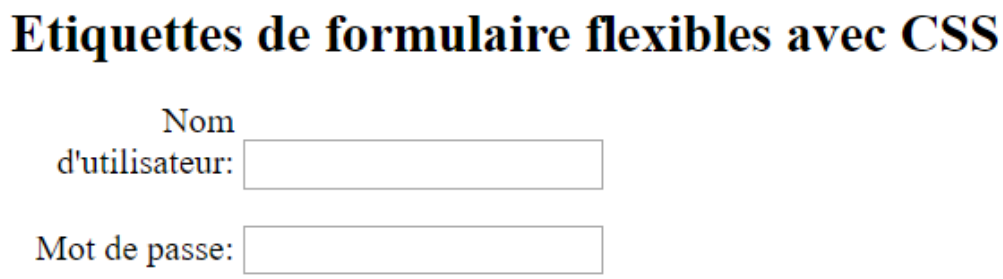

Figure 4 : Formulaire obtenu avec le code de la figure 3

Pour continuer, nous avons écrit un script CSS permettant d'afficher les étiquettes au-dessus des champs texte du formulaire quand la taille de l'écran est inférieure à 480px.
Le code pour réaliser cela, est le suivant :

```css
@media screen and (max-width: 480px){

    label, input {
        text-align: center;
        display: block;
        vertical-align: top;
        width: 100%;
    }
}
```

Figure 5 : ajout d'instruction pour le responsive

Ainsi, on obtient la fenêtre suivante :

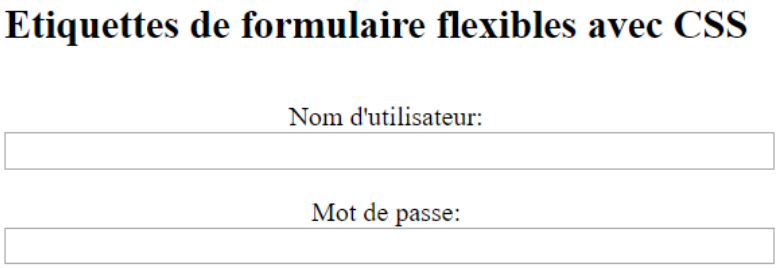

Figure 6 : Formulaire obtenu avec une largeur de 477px

Pour ce faire, nous avons utilisé un *@media*, afin de pouvoir exécuter le code seulement si l'écran est inférieur à 480px. Puis, nous choisissons les éléments que nous voulons modifier : *label*, et *input*. On aligne le texte avec la première instruction. On change l'affichage en utilisant un conteneur comme un block. On précise ensuite que l'on veut que le label soit en haut. Pour un peu d'esthétique, nous ajoutons une largeur de 100%.

Exercice 2 : MEDIA QUERIES

Pour ce deuxième exercice, il nous est demandé d'utiliser le fichier media_queries.html. On peut remarquer qu'en réduisant la taille de la fenêtre, le texte ne se trouve plus sur la partie visible de l'écran. Il faudrait donc repenser la page si l'écran est plus petit qu'une certaine taille, en changeant la mise en page.

Pour cela, nous allons utiliser des *@media*, qui permet de récupérer la largeur de l'écran. Pour obtenir le même affichage que demandé, nous écrivons les instructions suivantes :

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

    #colonneG ul{
        list-style-type: none;
    }

    #colonneG li{
        margin: 10px;
        padding:0;
        display: inline;
    }

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

Figure 7 : si la fenêtre est inférieure à 800px

Pour le titre, nous ajoutant un *text-align* afin d'aligner le texte à droite comme sur le modèle. Nous changeons aussi la couleur du fond en vert. Ensuite, pour mettre notre liste sous forme de ligne, nous ajoutons un *display : inline*, et une *margin : 10px,* afin d'espacer les éléments.
Nous obtenons donc si la page est plus grande que 800px :

Figure 8 : fenêtre avec une largeur de plus de 800px

Et si la page est plus petite que 800px :

Figure 9 : fenêtre responsive avec une largeur de moins de 800px

Exercice 3 : DETECTION DE FONCTIONNALITES

Pour cet exercice, nous utilisons le fichier detection_de_fonctionnalites.html. En l'ouvrant, on observe des titres *h1*, et *h2* ainsi que des paragraphes correspondant aux fonctionnalités du CSS, du HTML ou du JavaScript. Le but de cet exercice consiste à récupérer des informations.
Pour ce faire, il nous est demandé d'inclure la librairie *modernizr.js*. De plus, nous ajoutons à la balise *html* une classe nommée « no-js ». Cette dernière permet de supprimer la classe no-js à l'exécution de la librairie Modernizr et de la remplacer par js. Cela permet donc de vérifier que le javascript est bien activé.

```
<script src="modernizr.js"> </script>
```

Figure 10 : ajout de la librairie

De plus, nous créons un nouveau fichier js que nous incluons de la même manière que la librairie (figure 9). Nous ajoutons le code suivant :

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

Figure 11 : code dans le fichier script.js

Ce dernier permet de vérifier si une fonctionnalité est présente. Par exemple, pour la première ligne, nous cherchons l'élément HTML ayant comme identifiant *geoloc*, et appelons la géolocalisation de la librairie Modernizr. Si elle est présente alors on affiche « pris en charge » sinon « non pris en charge ». Les autres fonctionnalités, à savoir évènement tactile, SVG (Scalable Vector Graphic), canvas (la prise en charge des éléments graphiques) sont testées de la même façon. On obtient alors :

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 12 : exécution de la page HTML

Nous créons ensuite un fichier CSS et nous l'incluons dans le fichier HTML de la même méthode que l'exercice 1, figure 1. Nous écrivons les lignes ci-dessous :

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

Figure 13 : Code dans le CSS

On remarque que ce code permet de sélectionner plusieurs classes. Le premier bloc d'instruction permet de cacher les éléments qui appartiennent à la classe *noanimtest* ou la classe *animtest*. Les deux autres blocs permettent eux aussi de gérer l'affichage. On remarque que nous utilisons une classe *cssanimations* et une classe *no-cssanimations*. Dans le fichier Modernizr.js se trouve cette fonction :

```
tests['cssanimations'] = function() {
    return testPropsAll('animationName');
};
```

Figure 14 : fonction dans modernizr.js

Avec ces blocs, nous faisons savoir que animtest est une animation prise en charge et l'autre ne l'ai pas.
On obtient le résultat suivant :

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 15 : Code obtenu avec l'ajout du CSS

Exercice 4 : Stockage local

Le local storage représente la sauvegarde d'informations sur le navigateur de l'utilisateur.
En ouvrant le fichier stockage_local.html, nous remarquons qu'il s'agit d'un formulaire avec des champs texte ainsi que 2 boutons. Nous pouvons remarquer que si nous appuyons sur n'importe quel bouton, une méthode du javascript est appelée grâce à l'attribut *onclick* présent dans la balise *input*. Pour le bouton « Enregistrer », nous appelons la méthode *storeLocalContent*, où le programme récupère les valeurs des champs ayant les identifiants *firstName*, *lastName* et *postCode*. Pour le bouton « Effacer », une méthode est appelée : *clearLocalContent()*, qui permet comme son nom l'indique, effacer les valeurs remplies dans le formulaire.

Pour la suite, nous ajoutons un fichier javascript que nous relions à notre HTML. On ajoute le code suivant :

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

window.onload = initialize;
```

Figure 16: contenu du JavaScript

« 'LocalStorage' in window » permet selon moi de vérifier si la propriété existe dans l'objet window. La deuxième partie vérifie si le localstorage n'est pas nul.
Pour continuer, dans la fonction *initialize(),* on ajoute le code suivant :

```
if (!bSupportsLocal) {
    document.getElementById('infoform').innerHTML = "<p>Désolé, ce navigateur nes upporte pas l'API Web Storage du W3C.</p>";
    return;
}
```

Figure 17 : ajout dans initialize()

Ici, si bSupportsLocal est égal à nul, on ajoute une balise paragraphe pour informer l'utilisateur que l'API n'est pas prise en charge par le navigateur.

```
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');
}
```

Figure 18 : ajout d'un test

Ce test permet de récupérer les valeurs des champs si le local Stockage est rempli. On récupère donc le prénom, le nom et le code postal de la personne.
De plus, nous créons ensuite une fonction storeLocalContent() qui sera appelée au moment où l'utilisateur cliquera sur le bouton « Enregistrer ».

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

Figure 19 : fonction storeLocalContent

Ici, nous ajoutons donc au tableau local Storage la valeur des champs passés en paramètres et de l'identifiant. Suite à une erreur où le navigateur ne trouvait pas les méthodes javascript, j'ai décidé de changer l'aspect du fichier javascript.

```
window.onload = function() {

   let bSupportsLocal = (('localStorage' in window) && window['localStorage'] !== nul

    if (!bSupportsLocal) {
        document.getElementById('infoform').innerHTML = "<p>Désolé, ce navigateur nes
        return;
    }

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



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

Figure 20: code dans le javascript

Pout la fonction *clearLocalContent()*, nous avons changé la valeur des champs texte, puis nous avons enlevé tous les éléments du local Storage.

```
let clearLocalContent = function() {
    window.localStorage.clear();
    document.getElementById('firstName').value = "";
    document.getElementById('lastName').value = "";
    document.getElementById('postCode').value = "";

}
```

Figure 21 : fonction clearLocalContent

A l'exécution, on entre les données suivantes :

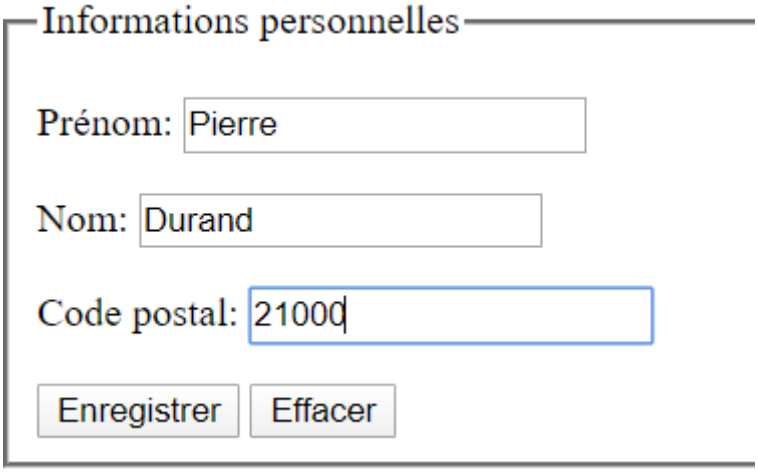

Figure 22 : exemple de données

Le local Storage se remplit donc :

Key	Value
firstName	Pierre
lastName	Durand
postCode	21000

Figure 23 : Le local Storage

Puis en appuyant sur effacer, les valeurs des champs texte redeviennent nulle et le local Storage se vide.

Si on navigue sur une autre page alors que nous avons sauvegarder des informations entrées via le formulaire, elle sont encore là et sauvegarder.

Exercice 5 : Géolocalisation

Dans un premier temps, nous utilisons le fichier geopositionnement.html. Ce fichier est censé affiché la position où nous nous trouvons. Il renseigne la longitude, la latitude, la précision mais aussi la précision, l'altitude, la pression d'altitude, le cap, la vitesse et la distance de l'ESIREM.

Pour renseigner ces informations, nous ajoutons la librairie Modernizr vu à l'exercice 3. Nous ajoutons également une classe à la balise *html*, nommée *no-js*.
Pour continuer, nous créons un fichier javascript, sans oublié de faire un lien dans l'HTML vers ce fichier.

```
function getLocation()
{

}
window.onload = getLocation;
```

Figure 24 : contenu du fichier

La fonction *getLocation()* sera appelée dès le chargement de la page web (5ème ligne). Nous vérifions ensuite que la géolocalisation est possible ; si c'est le cas, on demande au navigateur de récupérer la position courante. Deux méthodes sont utilisées : *geoSuccess* et *geoError*. Si l'utilisateur accepte de partager sa position et qu'il n'y a pas de problème de recherches, alors nous pourrons afficher, sinon un message d'erreur apparaîtra.

Longitude: 3.5454976

Latitude: 47.798681599999995

Précision: 2503

Figure 25 : valeurs affichées

Cependant les autres informations ne s'affichent pas.
Pour calculer la distance entre la position actuelle et la position de l'ESIREM, nous utilisons les méthodes *calculDistance* et *degreesEnRadians* fournies.

```
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 26 : méthodes fournies

Nous créons deux tableaux de coordonnées comme ci-dessous, dans la méthode geoSuccess :

```
let coordonneesESIREM ={
    latitude: 47.3121519,
    longitude: 5.0039326
};
let coordonneesActuelles= {
    latitude: positionInfo.coords.latitude,
    longitude: positionInfo.coords.longitude
};

let distance = calculDistance(coordonneesActuelles, coordonneesESIREM);

document.getElementById("distance").innerHTML = distance;
```

Figure 27 : code pour afficher la distance

Nous appelons la méthode *CalculDistance*, puis nous récupérons la distance et la plaçons dans l'élément HTML ayant l'identifiant distance. Ainsi, à l'écran, nous obtenons :

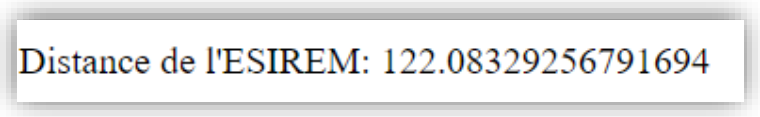

Figure 28: Distance entre l'ESIREM et ma position