

Module 307

Réaliser des pages Web interactives

Rapport personnel

Malori Burgy

Module du 15.05.2023 au 13.06.2023

Version 1

Du 13.06.2023

Table des matières

1	Introduction	4
2	Exercices	4
2.1	Exercice 1	4
2.2	Exercice 2	5
2.3	Exercice 3	7
2.4	Exercice 4	9
2.5	Exercice 5	10
2.6	Exercice 6	13
2.7	Exercice 7	15
2.8	Exercice 8	18
2.9	Exercice 9	20
2.10	Exercice 10	22
2.11	Exercice 11	27
2.12	Exercice 12	30
2.13	Exercice 13	31
2.14	Exercice 14	33
2.15	Exercice 15	35
2.16	Exercice 16	37
2.17	Exercice 17	38
2.17.1	REST architecture générale	38
2.17.2	SOAP	39
2.17.3	Comment faire un GET, avec quel outil et détaillez un exemple de GET sur Postman	40
2.17.4	en REST, comment faire un POST, avec quel outil et détaillez un exemple de POST sur Postman	41
2.17.5	en REST, comment faire un PUT, avec quel outil et détaillez un exemple de PUT sur Postman	42

2.17.6	<i>en REST, comment faire un DELETE, avec quel outil et détaillez un exemple sur Postman</i>	42
2.18	Exercice 18	44
2.19	Exercice 20	46
3	Projet	48
3.1	Introduction du projet	48
3.1.1	<i>Définition: but, objectifs et fonctionnement désiré de votre projet</i>	48
3.1.2	<i>Explications API</i>	48
3.1.3	<i>Site de référence</i>	48
3.1.4	<i>Fonctionnement: requête, réponse, format</i>	48
3.1.5	<i>Exemple d'utilisation</i>	49
3.2	Analyse	50
3.2.1	<i>Diagramme de use cases / Explications des cas</i>	50
3.2.2	<i>Maquette</i>	50
3.3	Conception	52
3.3.1	<i>Diagramme de navigation</i>	52
3.4	Implémentation	53
3.4.1	<i>HTML / CSS (buts des fichiers, éventuellement des extraits de code avec explications)</i>	53
3.4.2	<i>Javascript général (buts des fichiers, extraits de code avec explications)</i>	54
3.4.3	<i>Javascript spécifiques (buts des fichiers, extraits de code avec explications)</i>	54
3.4.4	<i>Descente de code complète pour une action qui fait appel à un service Web</i>	56
3.5	Test	57
4	Conclusion	58

1 Introduction

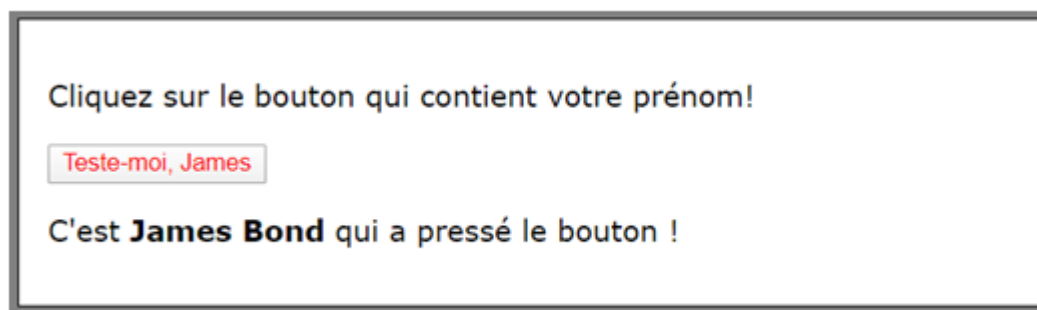
Dans ce module, nous allons apprendre les bases de javascript via des exercices. Nous allons tester et documenter ces résultats pour pouvoir ensuite arriver à la partie la plus intéressante du module, un projet d'application Web qui fonctionne selon les modèles que nous aurons vu pendant les exercices et qui utilisera des Webservices.

2 Exercices

2.1 Exercice 1

Dans cet exercice, nous avons un affichage qui contient du texte et un bouton avec notre prénom. Quand on clique sur ce bouton, il va dire que c'est nous qu'il l'avons pressé.

Voici un aperçu de la maquette avant que je change les informations avec mon nom et prénom :



- Comment ajouter un écouteur "click" via Javascript.

Dans le modèle, l'écouteur du bouton est ajouté depuis le fichier javascript. Pour rajouter un événement, on utilise simplement la fonction « `addEventListener` » avec l'ID qui fait référence à notre bouton, ici il s'appelle « `testez` ». Voici à quoi ressemble le code :

```
// Ecouteur du bouton "Testez-moi..."
document.getElementById("testez").addEventListener("click", testez);
```

On peut voir aussi dans l'html que l'id de mon bouton est bien « `testez` ».

```
<body onload="initCtrl()">
  <div id="container">
    <p>Cliquez sur le bouton qui contient votre prénom (js) !</p>
    <button id="testez">Teste-moi, James</button>
    <p id="info">&nbsp;</p>
  </div>
</body>
```

Dans l'exercice, il est demandé de changer cette manière de faire et de directement ajouter l'écouteur sur le bouton dans le fichier html. Il est préférable d'utiliser l'autre méthode à l'avenir mais pour l'exercice, c'est bien de pouvoir expérimenter les deux.

J'ai donc directement rajouté l'attribut « `onclick` » à mon bouton et je lui ai donné ma fonction javascript « `testez ()` ».

```
<body onload="initCtrl()">
  <div id="container">
```

```

<p>Cliquez sur le bouton qui contient votre prénom (html) !</p>
<button onclick="testez()">Teste-moi, Malori</button>
<p id="info">&nbsp;</p>
</div>
</body >

```

Voici ce que cela donne une fois que j'ai tout changé :

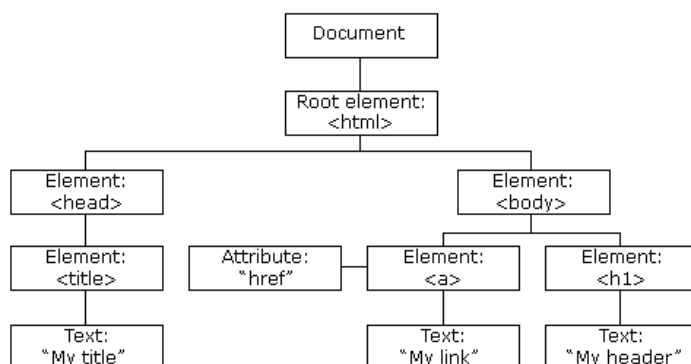
Cliquez sur le bouton qui contient votre prénom (html) !

Teste-moi, Malori

C'est **Malori Burgy** qui a pressé le bouton !

Il est aussi possible d'exécuter le code JS une fois que tout a été chargé en changeant simplement e place le « onload » on peu le mettre sur la balise de fermeture du Body ce n'est pas un problème.

En web, on va parler de « DOM » Document Object model, c'est un peut l'arborescence de la page et de ce qui la compose.

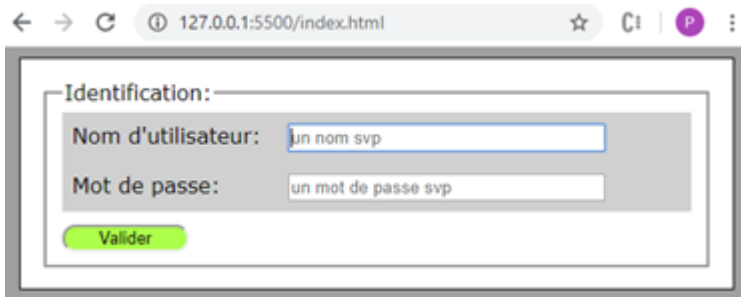


Le script javascript, ne peut être lancé uniquement une fois que toute cette arborescence a été créé. On peut la lancer en début ou en fin de chargement comme indiqué au-dessus et on peut aussi déplacer el lancement du script au besoin.

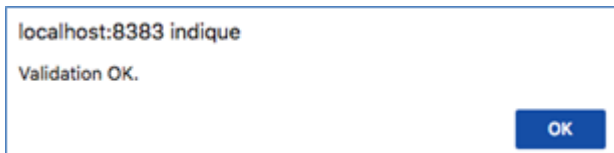
2.2 Exercice 2

Dans cet exercice, on nous demande de simuler une page de login qui va afficher un formulaire de connexion. Pour le moment le test de login va se faire localement avec javascript. Les informations de connexions étant les suivantes : « admin » quelle que soit la casse pour l'utilisateur et « emf123 » pour le mot de passe).

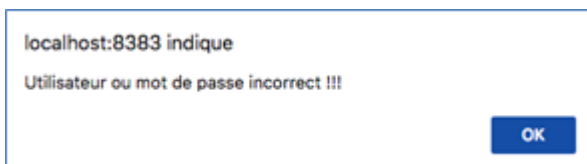
Voici la maquette :



Si on introduit les bonnes informations de connexion :



Et si on n'introduit pas les bonnes informations :



On va commencer par analyser le code HTML de la page :

```
<body onload="initCtrl()">
  <form class="user-form">
    <fieldset>
      <legend>Identification:</legend>
      <div class="field">
        <label for="username">Nom d'utilisateur:</label>
        <input type="text" size="30" id="username" placeholder="un nom svp" autofocus />
      </div>
      <div class="field">
        <label for="password">Mot de passe:</label>
        <input type="password" size="30" id="password" placeholder="un mot de passe svp" />
      </div>
      <input type="button" value="Valider" id="valider" onclick="validerUtilisateur();" />
    </fieldset>
  </form>
</body>
```

L'attribut « **placeholder** » va nous permettre de mettre du texte dans le champ que l'utilisateur devra remplir, cela lui permet aussi de savoir ce qu'il faudra qu'il y rentre.

L'attribut « **autofocus** » est Boolean qui indique que le « focus » de la page doit se faire sur cet élément.

On peut voir qu'ici on utilise un « **input** » et pas un « **button** », la seule différence qu'il y a entre les deux, c'est que le « button » est un peu plus personnalisable et polyvalent que « input ». On peut lui spécifier une valeur qui sera son texte mais c'est tout.

On peut aussi voir déjà la fonction js « validerUtilisateur » qui est appelée lorsque l'on clique sur le bouton valider.

On peut après se pencher sur le fichier javascript. J'ai réalisé une fonction « `validerUtilisateur` » qu'on définit avec le préfixe « **function** » qui récupère les informations de connexion entrées et teste si elles sont valides. Elle va tester si le nom d'utilisateur en minuscule est valable. Pour cela, on utilise la méthode « **toLowerCase** » comme en java.

```
function validerUtilisateur() {  
  
    // récupère les objets InputText du formulaire HTML et leur valeur  
  
    let username = document.getElementById("username").value;  
    let password = document.getElementById("password").value;  
  
    // validation du user et password  
    if (username.toLowerCase() === "admin" && password === "emf123") {  
        window.alert("Validation OK.");  
    } else {  
        window.alert("Utilisateur ou mot de passe incorrect !!!");  
    }  
}
```

J'ai fait deux variables grâce au mot clé « **let** » suivi d'un nom qui vont rechercher grâce à leur id les éléments « `username` » et « `password` ». Ils sont ensuite testés dans une condition en dessous.

En javascript, il n'est pas nécessaire de déclarer le type des variables.

La fonction « **alert()** » sert à afficher un pop-up pour dire si la validation est ok ou non.

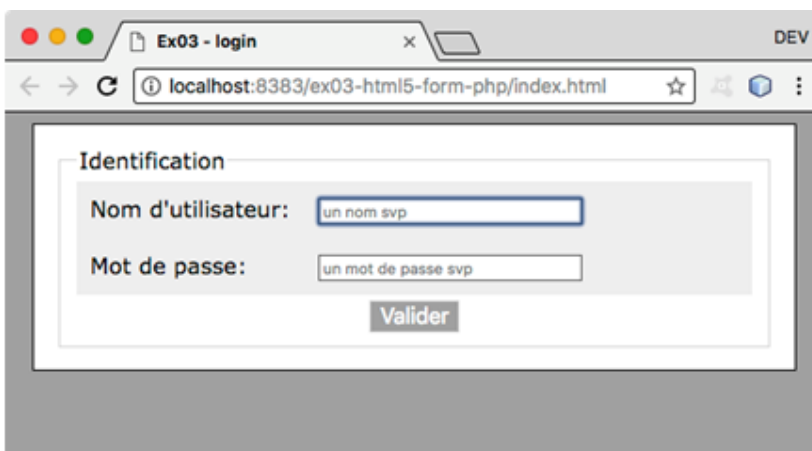
Si on doit déboguer et afficher par exemple les valeurs des deux champs, on peut utiliser la ligne suivante dans notre fonction JavaScript :

```
console.log("utilisateur: " + username + " mot de passe: " + password);
```

On peut afficher cette console en appuyant sur **F12**.

2.3 Exercice 3

Pour cet exercice, on va partir sur la base de ce que l'on a fait à l'exercice précédent. Il va falloir changer un peu l'affichage de notre page pour qu'elle ressemble à ça :



La première question à répondre est Comment allons-nous centrer et styliser ce bouton de validation ?

On va commencer par encapsuler le bouton dans une balise « **div** » à qui on va rajouter une classe que je nomme « **button** ».

```
</div>
    <div class="button">
        <input type="button" value="Valider" id="valider" onclick="validerUtilisateur();">
    </div>
```

Il va suffire ensuite d'ajouter la classe « **button** » dans mon fichier css et dans le bouton lui-même dans le fichier html changer son type en « **submit** » Nous n'aurons donc plus besoin du « onclick » présent.

Le nouveau code est donc :

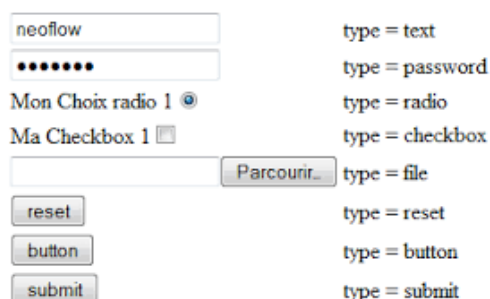
```
<div class="button">
    <input type="submit" value="Valider" id="valider" >
</div>
```

Voici à quoi ressemble mon code css :

```
.user-form input[type=submit] {
    width: auto;
    background-color: grey;
    border: none;
    padding: 5px 20px;
    font-size: 16px;
    color: white;
    margin-top: 0.5cm;
    font-weight: 400;
}
.user-form .button {
    display: flex;
    justify-content: center;
    align-items: center;
```

Pour la deuxième partie, la question est la suivante : Comment envoyer el formulaire à un script PHP qui fera la validation ?

Pour rappel, il existe une multitude de types de boutons en html. Voici une image récapitulative de ces différents types :



Dans notre cas, on a changé notre bouton en type « **submit** » parce qu'il envoie un formulaire vers un serveur quand on le clique et redirigera l'utilisateur vers une page qui a été défini dans l'attribut « **action** », le « **button** » est simplement un bouton cliquable sans grandes fonctionnalités particulières.

On a donc effectivement besoin d'un « **submit** » et comme indiqué dans l'explication ci-dessus, on va devoir spécifier la page dans l'attribut « **action** » de la balise « **form** »

Ma balise « **Form** » ressemble donc à ça :

```
<form class="user-form" action="http://burgym.emf-informatique.ch/307/Exercices/exercice_3/login.php" method="POST">
```

L'attribut « **action** » envoie le contenu du formulaire vers une page PHP.

Et l'attribut « **method** » qui est soit **GET** soit **POST**, ici **POST** à l'occurrence indique que l'on fait une modification /un ajout. Un **GET** aurait simplement permis de ressortir une information et non de la modifier.

J'ai ensuite rajouté le script PHP fourni dans l'exercice, maintenant le fichier javascript ne va plus nous servir la validation étant faite avec le script PHP. Pour PHP, il faut des attributs « **name** » qui fonctionnent exactement de la même manière que les id que nous avons déjà.

Voici le code PHP qui était fourni, on peut voir qu'il commence et se termine par une balise « **< ?PHP>** ». Pour convertir mon nom d'utilisateur en minuscule, ici la fonction utilisée est « **strtolower** », j'affiche les informations grâce à un « **echo** »

```
<?PHP
//Burgy Malori
// le 15.05.2023
//script PHP de l'exercice 3

// test si on a reçu une donnée de formulaire nommée "username"
if (isset($_POST['username'])) {

    // récupération des données transmises dans des variables locales
    $username = strtolower($_POST['username']);
    $password = $_POST['password'];

    // affichage des infos reçues
    echo "username: ".$username."<br>";
    echo "password: ".$password."<br>";

    // test username et mot de passe
    if (($username == "admin") && ($password == "emf123")) {
        echo "<script>alert('Validation OK');</script>";
    } else {
        echo "<script>alert('Utilisateur ou mot de passe incorrect !!!');</script>";
    }
}
?>
```

2.4 Exercice 4

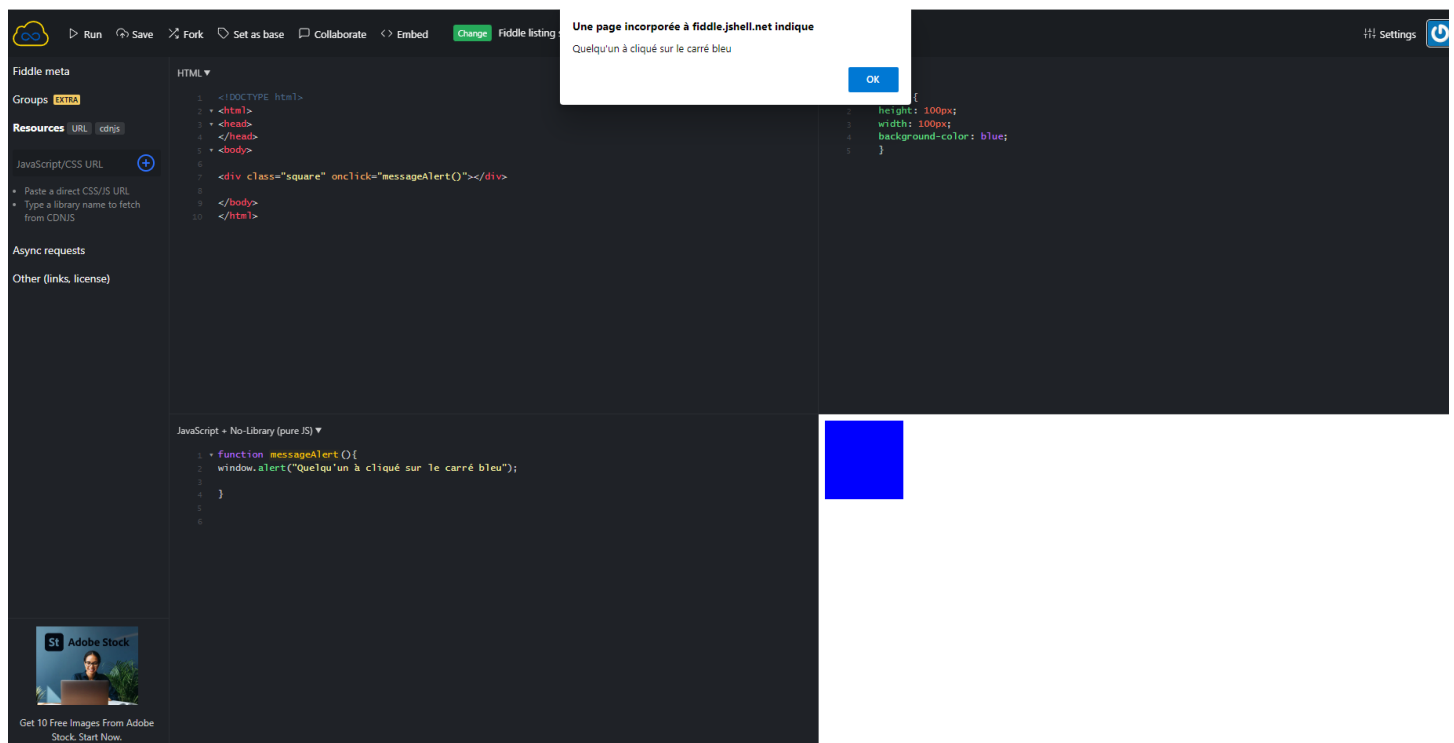
Cet exercice nous sert d'introduction à un outil qui sert à tester du JavaScript à partir de code HTML et CSS. On appelle cela un « **fiddle** » qui se traduirait comme une astuce.

J'ai donc créé un compte sur le site : <http://jsfiddle.net/>

Pour pouvoir tester ces « **fiddle** ». Grâce à ce site, il est possible de tester du code sans écrire toutes les références aux fichiers, il nous suffit juste de taper le code dont nous avons besoin.

Voici un exemple de petite application, j'ai un carré bleu sur une page et quand je clique dessus, un pop-up m'indique que ce carré a été cliqué.

Voici à quoi ressemble la page de l'outil, pour montrer que je n'ai vraiment pas écrit grand-chose et que tout fonctionne :



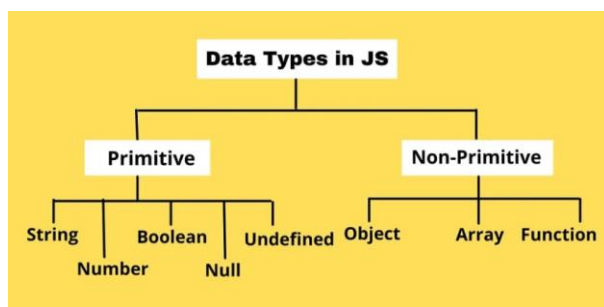
Cela illustre bien l'utilité d'un tel outil. On peut facilement et rapidement tester des bouts de code selon nos besoins.

2.5 Exercice 5

Dans cet exercice, nous allons nous pencher sur les variables en JavaScript. A noter qu'il y a différentes manières de déclarer une variable :

1. `myVar;`
2. `var myVar;`
3. `let myVar;`
4. `const myVar=1;`

Une variable peut aussi avoir un type même s'il n'est pas obligatoire de le préciser, voici une image qui récapitule les types que l'on peut rencontrer en JavaScript :



Les deux premières solutions sont équivalentes mais on ne les utilise pas vraiment car elles laissent beaucoup trop de liberté, on peut par exemple déclarer deux fois la même variable avec le même nom ce qui est plutôt problématique...

On devrait plutôt prioriser le mot-clé « **let** ».

Le mot-clé « **const** » est surtout utilisé pour déclarer des constantes et les objets car on peut toujours changer ses propriétés.

La grande différence qu'il y a entre les deux mots est qu'une variable déclarée avec un « **let** » peut être réaffectée, on peut changer sa valeur, cela n'est pas possible avec une variable déclarée avec « **const** ».



Comme en Java, la convention de mettre les constantes en majuscules est utilisée.

Pour tester tout ça, l'exercice proposait une série d'opérations à effectuer dans une console du navigateur ou dans un jsfiddle, j'ai choisi la deuxième option.

- Effacer le contenu de la console avec « `console.clear()` »

```
console.clear
```

- Créer une variable nommée « `a` » ;

```
let a;
```

- Afficher le contenu de « `a` » avec `console.log`;

```
console.log(a);
```

- Stocker la valeur 15 dans cette variable ;

```
a = 15;
```

- Afficher le contenu de cette variable dans la console sous la forme « Ma variable `a` = ? »

```
console.log("Ma variable a = " + a);
```

- Créer une variable nommée « `b` » et lui assigner directement la valeur 9 ;

```
let b = 9;
```

- Afficher le contenu de cette variable dans la console sous la forme « Ma variable `b` = ? »

```
console.log("Ma variable b = " + b);
```

- Faire l'addition de ces 2 variables en affichant directement le résultat dans la console sous cette forme : « `15 + 9 = ?` » ; (essayer d'utiliser un littéral avec ``...${...}...``)

```
console.log(" 15+ 9 = "+ (a+b));  
$c = a+b;  
console.log("Avec un littéral: 15+ 9 = "+ $c);
```

- Compléter en faisant de même pour une soustraction, une multiplication et une division des deux variables ;

```
console.log(" 15/9 = "+ (a/b));  
console.log(" 15*9 = "+ (a*b));  
console.log(" 15- 9 = "+ (a-b));
```

- Stocker « Bonjour » dans la variable `a` ;

```
a="bonjour";
```

- Stocker « les amis » dans la variable `b` ;

```
b= "les amis"
```

- Afficher « bonjour les amis » dans la console en concaténant les variables ;

```
console.log(a+" "+ b)
```

- Faites la même chose en utilisant un littéral avec ``...${...}...`` ;

```
$c=a+b;
console.log($c);
```

- Stocker « true » dans la variable a ;

```
a=true ;
```

- Stocker « false » dans la variable b ;

```
b=false ;
```

- Effectuer une opération AND entre les 2 variables et afficher le résultat sous cette forme « true AND false = ? » ;

```
console.log("true AND False = " + a&b);
```

- Effectuer une opération OR entre les 2 variables et afficher le résultat sous cette forme « true OR false = ? » ;

```
console.log("true OR False = " + a|b);
```

- Stocker la date du jour dans la variable a avec `new Date()`;

```
a = new Date(Date.now());
```

- Calculer une nouvelle date dans la variable b qui est 61 jours avant la date courante (utilisation `getDate`, `setDate`)

```
let e = (24*60*60*1000) * 61;
b= new Date(a.setTime(a.getTime()-e));
```

- Afficher les dates contenues dans les variables a et b en vous aidant de « `toLocaleString()`, `toLocaleDateString()` et `toLocaleTimeString()` ». Afficher la date et l'heure, la date uniquement et l'heure uniquement.

```
console.log(b.toLocaleString());
console.log(new Intl.DateTimeFormat('FR-CH').format(b));
console.log(b.toLocaleTimeString());
```

- Le mot réservé « `typeof` » permet de connaître le type utilisé momentanément pour une variable. Stocker la valeur de `Math.PI` dans a, « bonjour » dans b, créer et assigner `true` dans c, créer, assigner la date courante dans d et déclarez la variable e sans rien lui affectez, puis afficher le type pour les 5 variables :

```
a= Math.PI;
b= "Bonjour";
let c = true;
let d = new Date(Date.now());
let e;
console.log(typeof(a));
console.log(typeof(b));
console.log(typeof(c));
console.log(typeof(d));
console.log(typeof(e));
```

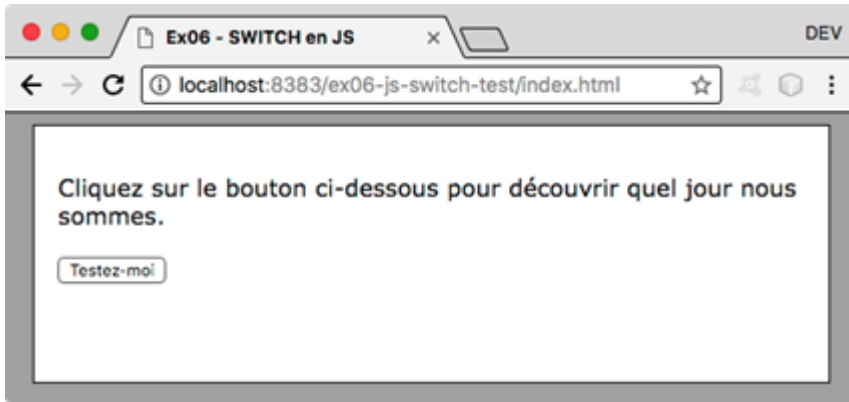
```
"number"
"string"
"boolean"
"object"
"undefined"
```

Voici le résultat que ces dernières lignes m'ont donné :

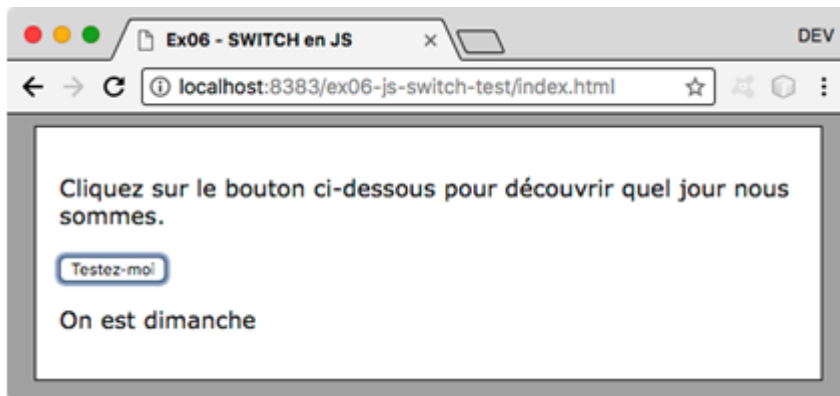
2.6 Exercice 6

Ici, nous allons reprendre l'exercice un et cette fois allons le réaliser avec un switch. On va changer quelques données pour que l'application affiche le jour de la semaine directement en remplacement du contenu qu'une balise html.

Voici la maquette au démarrage :



Après le clic, on affiche le jour de la semaine :



On reprend le code de l'exercice 1, on change l'HTML selon la consigne, le texte change et la fonction appelée va changer. Voici à quoi ressemble le nouveau « **body** » de l'exercice basé sur ce qui est dit dans la consigne et le code existant :

```
<body>
  <div id="container">
    <p>
      Cliquez sur le bouton ci-dessous pour découvrir quel jour nous sommes.
    </p>
    <button onclick="afficherJourSemaine();">Testez-moi</button>
    <p id="info">&nbsp;</p>
  </div>
</body>
```

Dans le fichier javascript, on va donc créer notre fonction « **afficherJourSemaine()** ». Pour ce faire, il va falloir utiliser un switch avec le numéro du jour et on va utiliser une méthode de la classe « **Date** » pour retourner le numéro du jour de la semaine.

Voici à quoi ressemble ma fonction :

```
function afficherJourSemaine() {
```

```

let info = document.getElementById("info");

//ATTENTION avec la méthode getDay(), le 0 corresponds à DIMANCHE et non lundi

let jSem = new Date().getDay();

// je fait une variable avec le début de ma phrase
//et je lui rajouterai le jour correspondant dans mon switch selon la valeur de getDay()

let onEst = "On est ";
switch (jSem) {
  case 0:
    onEst += "dimanche";
    break;
  case 1:
    onEst += "lundi";
    break;
  case 2:
    onEst += "mardi";
    break;
  case 3:
    onEst += "mercredi";
    break;
  case 4:
    onEst += "jeudi";
    break;
  case 5:
    onEst += "vendredi";
    break;
  case 6:
    onEst += "samedi";
    break;
}
// l'instruction inner html nous permet de modifier le contenu d'une balise ou d'intégrer un objet dans
une page
//ici on rajoute donc la phrase "on est[..] suivit du jour correspondant"
info.innerHTML = onEst;
}

```

L'autre possibilité, est d'utiliser un tableau. On aura moins de code avec un tableau et il fonctionne de la même manière qu'un tableau en java. La différence est qu'on peut agrandir et rétrécir un tableau javascript.

Pour faire un tableau avec mes jours de la semaine par exemple, je vais déclarer une constante, on va l'appeler « tab » et je vais lui donner entre croches tous les jours de la semaine.

Voici un exemple de déclaration de tableau avec tous les jours de la semaine :

```
const jourSemaine = ["dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"];
```

Pour faire une fonction javascript, on va regarder si un élément « getDay » corresponds à un des index de notre tableau, l'affichage est le même que pour la fonction d'avant. Il faut faire attention ici, car le nombre 1 corresponds en fait au dimanche. Il faudra juste commencer notre tableau par dimanche du coup.

Voici à quoi ressemble la fonction JavaScript, elle utilise le tableau que j'ai déclaré en dessus :

```
const jourSemaine = [
  "dimanche",
  "lundi",
  "mardi",
  "mercredi",
  "jeudi",

```

```

    "vendredi",
    "samedi",
];

function afficherJourSemaineTab() {
    //on va aller rechercher l'élément avec l'ID "info"
    //Ensuite on crée une nouvelle date qui nous donne le numéro du jour de la semaine
    //Puis on crée un élément div pour que l'on puisse mettre en ligne nos deux résultat
    //on passe le jour de la semaine à notre tableau qui aura le jour correspondant à l'index qu'on lui
    donne vu que tout est dans l'ordre
    //on rajoute au paragraphe avec l'id "info" la nouvelle ligne qui nous dit quel jour on est avec un
    tableau

    let info = document.getElementById("info");
    let jSem = new Date().getDay();
    let div = document.createElement("div");
    div.innerHTML = "On est " + jourSemaine[jSem] + " via [tableau]";
    info.appendChild(div);
}

```

A noter que ce n'est pas tout ce qu'on peut faire avec un tableau en JavaScript évidemment.

Pour accéder à un élément dans un tableau c'est comme en Java, on indique le numéro de l'index de la cellule dont on veut la valeur :

```
jourSemaine[0] ;
```

Me redonnerait par exemple « **dimanche** ».

Pour modifier un élément on part de la ligne d'au-dessus et on ajoute la nouvelle valeur :

```
jourSemaine[0] = « Premier jour de la semaine »;
```

Change donc Dimanche en « **premier jour de la semaine** ».

Pour ajouter et supprimer des éléments dans un tableau on utilise les méthodes « **push** » et « **pop** ».

« **Push** » va nous permettre de rajouter quelque chose dans notre tableau par exemple :

```
jourSemaine.push(« je suis un nouveau jour ») ; ;
```

« **Pop** » va supprimer le dernier élément de la liste, si on reprend la ligne du dessus il va supprimer « **je suis un nouveau jour** » :

```
jourSemaine.pop();
```

Pour afficher le contenu d'un tableau, on fait comme en Java et on peut utiliser la méthode « **toString()** » :

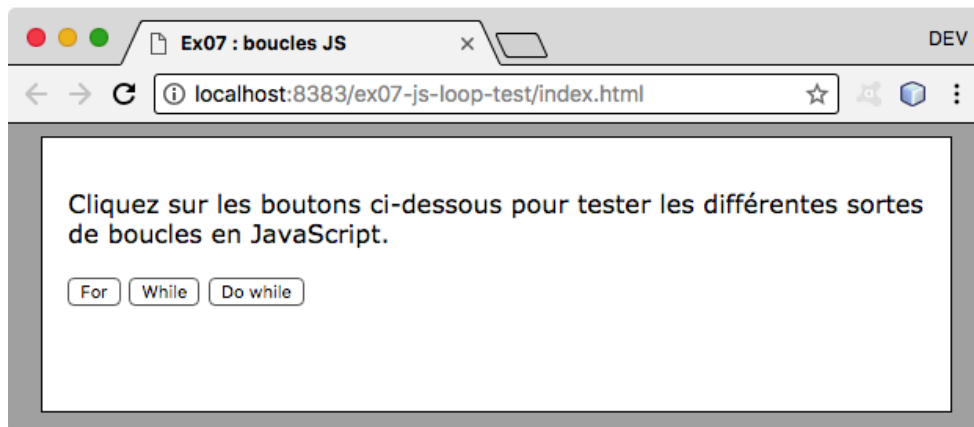
```
jourSemaine.toString() ;
```

Me redonnerais donc : « dimanche, lundi, mardi, mercredi, jeudi, vendredi, samedi ».

2.7 Exercice 7

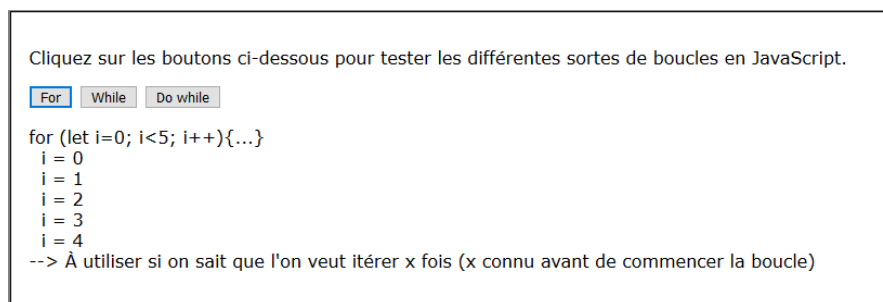
En se basant sur l'exercice précédent, nous avons dû ici tester plusieurs sortes de boucle en JavaScript.

La maquette ressemble à ça :

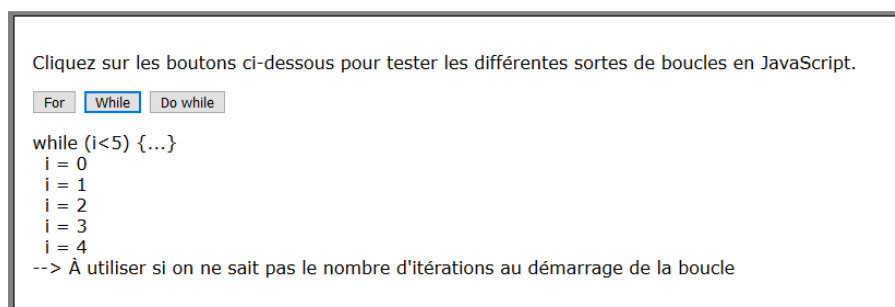


On peut voir que chaque bouton correspond à une boucle et qu'en cliquant dessus, on va évidemment les utiliser dans notre code.

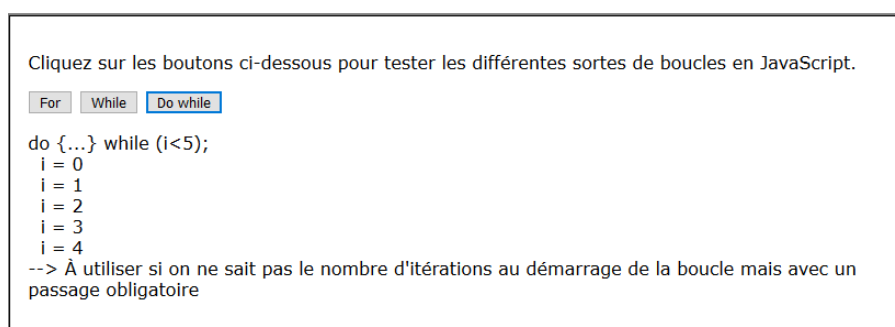
Voici à quoi cela doit ressembler si on clique sur le bouton « **for** »



Le bouton « **while** »



Et enfin le bouton « **do while** »



Je commence par changer un peu le fichier html pour que l'affichage ressemble à celui de la maquette. Voici le nouveau « **body** » du fichier :

```
<body onload="initCtrl()">
  <div id="container">
    <p>
      Cliquez sur les boutons ci-dessous pour tester les différentes sortes de
      boucles en JavaScript.
    </p>
    <button id="for">For</button>
    <button id="while">While</button>
    <button id="do">Do while</button>
    <p id="info">&nbsp;</p>
  </div>
</body>
```

On peut voir que chaque bouton aura donc sa propre fonction qui fonctionnera avec une boucle différente à chaque fois.

Pour commencer, je vais utiliser des écouteurs pour chacun de mes boutons dans cet exercice. Mais comme le script est chargé avant le DOM, il pourrait me donner des erreurs car certains éléments pourraient ne pas être créés.

Je vais donc mettre tous les écouteurs dans la fonction « **onload** » de mon body pour les initialiser

```
function initCtrl() {
  document.getElementById("for").addEventListener("click", testerFor); // appel de la fonction testerFor
  avec un événement "click"

  let butWhile = document.getElementById("while");
  butWhile.addEventListener("click", testerWhile); // appel de la fonction testerWhile avec un événement
  "click"

  let butDo = document.getElementById("do");
  butDo.addEventListener("click", testerDoWhile); // appel de la fonction testerDoWhile avec un
  événement "click"
}
```

On va commencer par la boucle « **For** ». Elle va nous permettre d'itérer des éléments x fois. Avec cette boucle on sait combien de fois on veut et on va itérer notre élément.

Voici à quoi ressemble mon code :

```
function testerFor() {
  let info = document.getElementById("info");
  let div = document.createElement("div");

  div.innerHTML += "for (let i = 0; i < 5; i++) {...} <br>";

  for (let i = 0; i < 5; i++) {
    div.innerHTML += " i= " + i + "<br>";
  }
  div.innerHTML +=
    "--> A utiliser si on sait que l'on veut itérer x fois (x connu avant de commencer la boucle)";

  info.appendChild(div);
}
```

On peut aussi voir que j'ai utilisé une balise « **
** » qui sert à faire un retour à la ligne à l'affichage. J'ai testé directement la boucle avec l'affichage et la condition est que jusqu'à ce que i soit plus

grand que 5 j'affiche sa valeur et je l'augmente de 1. Le chiffre **5** ne s'affichera donc jamais, si j'avais voulu l'afficher j'aurais dû mettre un « **<=** »

Pour la boucle « **while** », on va l'utiliser si on n'a pas défini ce nombre d'itération ou si on ne sait pas combien d'itérations vont se faire.

Dans ma boucle ci-dessous, je vais donc afficher la valeur de **i** tant qu'elle est plus petite que **5** et l'augmenter d'un à chaque fois.

```
function testerWhile() {
  let info = document.getElementById("info");
  let div = document.createElement("div");

  let i = 0;
  div.innerHTML += "while(i<5){...} <br>";
  while (i < 5) {
    div.innerHTML += "i= " + i + "<br>";
    i++;
  }
  div.innerHTML +=
    "--> A utiliser si on ne sait pas le nombre d'itérations au démarrage de la boucle";
  info.appendChild(div);
}
```

Et finalement, pour la boucle « **do...while** », On va aussi l'utiliser quand le nombre d'itérations nous est inconnu mais contrairement à la boucle « **while** » ici, on va obligatoirement passer au moins une fois dans la boucle.

Pour la boucle ci-dessous, je vais afficher la valeur de **i** tant qu'elle est plus petite que **5**. Si la valeur avait été plus grande, elle serait quand même affichée car comme dit au-dessus, un passage est obligatoire au minimum dans cette boucle.

```
function testerDoWhile() {
  let info = document.getElementById("info");
  let div = document.createElement("div");
  let i = 0;
  div.innerHTML += "do{...} while(i<5); <br>";

  do {
    div.innerHTML += "i= " + i + "<br>";
    i++;
  } while (i < 5);

  div.innerHTML +=
    "--> A utiliser si on ne sait pas le nombre d'itérations au démarrage de la boucle mais avec un passage obligatoire";
  info.appendChild(div);
}
```

2.8 Exercice 8

On va partir sur la base de l'exercice précédent pour tester les tableaux en Json. On commence par supprimer tout le code javascript de l'exercice précédent pour repartir de 0.

On va tester une imbrication de deux boucles sur les données d'un tableau de personne défini par un objet de type JSON.

Sa syntaxe ressemble fortement à du javascript. Ce qui lui donne un avantage car il sera donc plus compréhensible de ce point.

Voici un exemple pour illustrer tout ça

```
// JSON exemple
const personne = {
  prenom: "Jules",
  nom: "Tartampion",
  age: 25
};
```

Ci-dessus, on peut voir que par exemple, je donne les informations d'une personne.

On utilise l'opérateur « **in** » dans un « **for** » pour balayer tous les champs « **f** » (field) de l'objet.
Exemple :

```
for (let f in personne) {
  console.log("field: " + f + ", valeur: " + personne[f]);
}
```

On peut aussi faire des tableaux d'objets, voici un exemple avec des personnes :

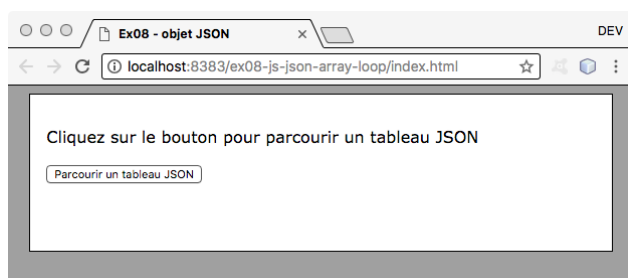
```
const json = {
  personnes: [
    {prenom: "John", nom: "Doe", age: 44},
    {prenom: "Anna", nom: "Smith", age: 32},
    {prenom: "Peter", nom: "Jones", age: 29}
  ]
};
```

On parcourt un tableau avec une boucle « **for** » comme en java.

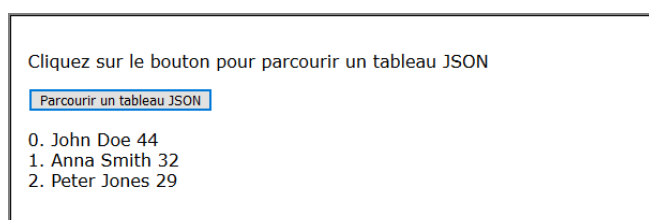
```
for (let i = 0; i < json.personnes.length; i++) {
  let personne = json.personnes[i];
  // ...
}
```

Donc pour en revenir à l'exercice, on va tester la théorie ci-dessus, on va faire une application qui va nous permettre de parcourir un tableau JSON.

Voici la maquette de l'application :



Et une fois que le bouton a été pressé on devrait avoir ceci :



On commence par changer un peu le code html pour qu'il corresponde à la maquette. Voici le nouveau body de l'application :

```
<body>
  <div id="container">
    <p>Cliquez sur le bouton pour parcourir un tableau JSON</p>
    <button onclick="parcourirUnTableauJSON();">
      Parcourir un tableau JSON
    </button>
    <p id="info">&nbsp;</p>
  </div>
</body>
```

On peut voir que l'on va donc devoir créer une fonction « **parcourirUnTableauJSON()** » dans notre fichier javascript.

J'ai donc créé un tableau de personne que j'ai appelé « **lesPersonnes** », je l'ai rempli avec les trois personnes de la consigne et ensuite j'ai fait une boucle for sur les personnes de mon tableau en faisant « **lesPersonnes.personnes** ». J'ai fait une variable « **personne** » à qui je donne la valeur de « **lesPersonnes.personnes[i]** ». Je vais prendre l'index de ces personnes représenté par « **i** » et je vais rajouter un point à côté pour que l'affichage soit le même que dans la consigne. Je vais ensuite faire une boucle dans ces nouvelles personnes et je vais simplement les afficher avec un espace. Pour finir, je rajoute un retour à la ligne avec la balise « **br** » et je rajoute mes lignes dans mon fichier html avec « **innerHTML** ».

Voici à quoi ressemble le code de ma fonction :

```
function parcourirUnTableauJSON() {
  //on déclare un tableau de personne et on le remplit
  const lesPersonnes = {
    personnes: [
      { prenom: "John", nom: "Doe", age: 44 },
      { prenom: "Anna", nom: "Smith", age: 32 },
      { prenom: "Peter", nom: "Jones", age: 29 },
    ],
  };
  let res = ""; // je fais une variable résultat pour pouvoir y stocker l'affichage de mes personnes
  for (i = 0; i < lesPersonnes.personnes.length; i++) {
    let personne = lesPersonnes.personnes[i];
    res += i + ". "; //je rajoute un "." après el numéro d'index pour l'affichage
    for (let f in personne) {
      res += personne[f] + " ";
    }
    res += "<br>"; //je fais un retour à la ligne
  }
  document.getElementById("info").innerHTML = res; // on rajoute les 3 lignes de mes personnes dans
  l'élément html avec l'id "info"
}
```

2.9 Exercice 9

Dans l'exercice précédent, nous avons vu comment créer un objet avec la notation JSON. Il y a une 2e façon de faire en créant une instance à partir de la fonction « **Object** ». Ensuite, on peut définir des classes pour créer des objets, on a même 3 manières de faire.

Cet exercice est une démonstration de ces différentes manières de faire.

Pour commencer sans utiliser de classe, pour créer un objet on le déclare simplement comme suis :

```
const p1 = {  
  prenom: "Jules",  
  nom: "Tartampion",  
  age: 25  
};
```

Et si on veut rajouter par exemple un attribut a cet objet « **p1** » on fait :

```
p1.ville = "Fribourg" ;
```

Une autre manière de créer un objet, est d'utiliser la fonction « **object** ». On va donc faire des « **new()** ».

```
const p2 = new Object();  
p2["prenom"] = "Juliette";  
p2["nom"] = "Tartampion";  
p2["age"] = 23;
```

Javascript gère les attributs d'un objet un peu comme des éléments d'un hashmap (clé-valeur). Grace à ça on peut ajouter, supprimer ou même modifier ceux-ci. Autre chose d'intéressant est que comme on l'a déjà mentionné, les variables sont non typées en javascript. Ça nous permet de stocker un peu tout ce que l'on veut dans un objet.

Avec ça, on va donc pouvoir surcharger des méthodes et/ou en ajouter. On peut par exemple surcharger la méthode « **toString** » en remplaçant l'attribut nommé « **toString** » de l'objet.

Voici une première manière de le faire qui est la plus courante :

```
p1.toString = function() {  
  return this.prenom + " " + this.nom;  
};
```

Et une deuxième qui est plutôt rare à rencontrer :

```
p2["toString"] = function() { // très rare  
  return this.prenom + " " + this.nom;  
};
```

On peut aussi créer un objet avec une « **Function Objects** », ça ressemble presque à une fonction java, on peut aussi utiliser la méthode to string et on se rapproche de ce que l'on a vu plus haut.

```
function Personne(prenom, nom, age) {  
  this.prenom = prenom;  
  this.nom = nom;  
  this.age = age;  
  this.toString = function(){  
    return this.nom + " " + this.prenom + " (" + this.age + ")";  
  }  
}
```

On peut donc maintenant instancier un objet avec un « **new** » comme en java, mais le problème revient le même qu'avant, la fonction « **toString** » fait partie de l'objet et elle est donc copiée pour chaque nouvel objet. Malgré cela, il est possible de rajouter des fonctions par après.

Pour éviter la duplication des fonctions, il faut utiliser la programmation par prototypage grâce à l'attribut « **prototype** » qui se trouve dans tout objet JS. Voici un exemple et cette fois la fonction a disparu de l'objet, on s'approche de ce qui est fait en Java :

```
class Personne {
  constructor(prenom, nom, age) {
    this.prenom = prenom;
    this.nom = nom;
    this.age = age;
  }

  toString() {
    return this.nom + " " + this.prenom + " (" + this.age + ")";
  }
}
```

2.10 Exercice 10

Voici la consigne de l'exercice :

Depuis votre outil de développement, veuillez copier l'exercice « ex03... » vers « ex10-js-poo ». Supprimez tout le code JavaScript, pour le reconstituer plus tard avec les informations qui vont suivre. Cet exercice est l'un des plus importants de l'apprentissage de JavaScript, car il concerne la « **programmation orientée objets** » (POO) et la structuration de l'application en « **modules-fichiers** » de type MVC.

Le but est ici de construire une application qui permette de gérer simplement les informations d'une liste de personnes (prénom, nom, âge) avec des opérations « **métier** » comme « **ajouter une personne** » ou « **supprimer une personne** » que nous mettrons dans un fichier « **worker.js** ». Pour gérer des personnes, il faut également disposer d'un bean « **Personne** » (à la Java) que nous stockerons dans un fichier « **personne.js** ».

Vous pouvez dès à présent créer ces 2 fichiers JavaScript, mais vides pour le moment.

Nous ferons toute la partie « contrôle » de l'affichage et les interactions utilisateur dans le contrôleur nommé « **indexCtrl.js** »

Voici à quoi ressemble la maquette :

Maquette d'interface utilisateur pour la gestion de personnes. Le formulaire est intitulé « Veuillez introduire une nouvelle personne ou sélectionner une existante : ». Il contient trois champs de saisie : « Prénom: » avec la valeur « prénom », « Nom: » avec la valeur « nom », et « Âge: » avec la valeur « âge ». En dessous des champs, il y a trois boutons : « Ajouter », « Supprimer » et « Nettoyer ». En bas du formulaire, il y a une liste à puces de personnes existantes : « Doe John (44) », « Jones Peter (29) » et « Smith Anna (32) ».

Je commence donc par créer mes deux fichiers, « **personne.js** » qui représentera mon bean personne et « **worker.js** » qui sera le worker de mon application.

Il faut rajouter dans la partie « **head** » du fichier html les informations qui vont me permettre de charger les deux fichiers JavaScript que j'ai créé.

J'ai donc juste rajouté les deux lignes suivantes dans mon fichier :

```
<script src="js/worker.js" ></script>
<script src="js/indexCtrl.js" ></script>
```

Il va ensuite falloir transformer notre formulaire. Donc, la balise « **form** » n'a plus besoin de propriété « **action** » car on a plus de fichier PHP à gérer.

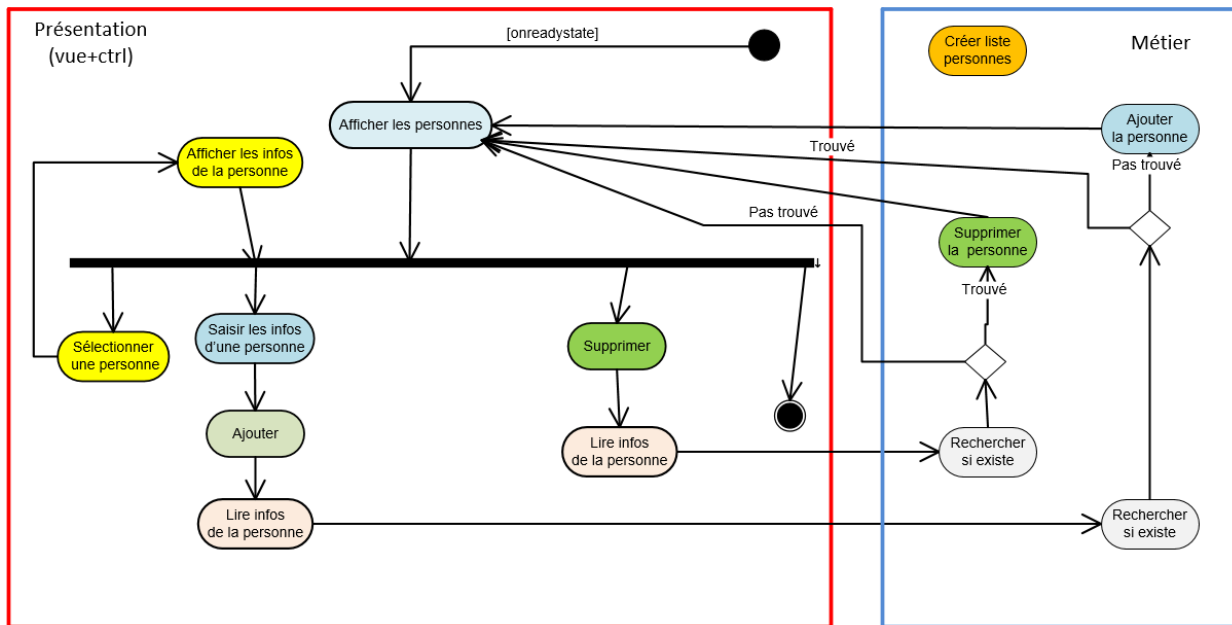
Les propriétés « **name** » des balises « **input** » peuvent être modifiées en « **id** », car on a plus à envoyer une requête HTTP de type GET ou POST, mais on devra récupérer ces informations dans le contrôleur avec des « **document.getElementById(id).value** ».

Il n'y aura plus de bouton de type « **submit** » non plus, car encore une fois, on a plus de formulaire à envoyer par HTTP(S).

Voici à quoi ressemble notre nouveau « **body** »:

```
<body>
  <div id="container">
    <form class="user-form">
      <fieldset>
        <legend>Veuillez introduire une nouvelle personne
          ou sélectionner une existante&nbsp;  </legend>
        <div class="field">
          <label for="prenom">Prénom:</label>
          <input type="text" size="20" id="prenom"
            placeholder="prénom" required="required" autofocus>
        </div>
        <div class="field">
          <label for="nom">Nom:</label>
          <input type="text" size="20" id="nom"
            placeholder="nom" required="required">
        </div>
        <div class="field">
          <label for="nom">Âge:</label>
          <input type="text" size="20" id="age"
            placeholder="âge" required="required">
        </div>
        <div class="button">
          <input type="button" value="Ajouter" onclick="ajouter();">
          <input type="button" value="Supprimer" onclick="supprimer();">
          <input type="reset" value="Nettoyer">
        </div>
      </fieldset>
    </form>
    <div id="info"></div>
  </div>
</body>
```

Il est maintenant temps de regarder le diagramme d'activités, voici à quoi il ressemble :



On va commencer par faire le bean « **personne** », comme en java les beans ont des propriétés et des méthodes. On va lui donner 3 attributs, un nom, un prénom et un âge qu'il va falloir écrire avec, en plus, une fonction (prototype) `toString()`. Chaque fois qu'on utilise un attribut dans le bean, on va devoir le précéder par le mot-clé **this**.

Voici à quoi ressemble mon bean :

```
function Personne(prenom, nom, age) {
  this.prenom = prenom;
  this.nom = nom;
  this.age = age;
}

Personne.prototype.toString = function () {
  return this.prenom + " " + this.nom + " " + this.age;
};
```

Ensuite, dans la couche métier on va devoir faire une fonction privée, qui va nous permettre d'ajouter et de supprimer une personne. La fonction prendra la forme de « **_trouverPersonne(p)** », On teste donc que si une personne doit être ajoutée à la liste, on regarde si elle n'y est pas déjà et même chose si on doit en supprimer une, on teste si elle existe dans notre liste.

Cette fonction « **_trouverPersonne** » peut retourner l'index (idx=0, 1, 2 ...) dans la liste de la personne recherchée ou éventuellement -1 si elle n'a pas été retrouvée. Cela permettra de la supprimer facilement avec la fonction « **.splice(idx, 1)** » disponible sur les tableaux. La fonction « **.push(p)** » permettra d'ajouter une personne à la liste.

On ne commence par créer notre liste de personnes :

```
// définition du modèle de données, créé au chargement du js dans le index.html
const personnes = [
  new Personne("John", "Doe", 44),
  new Personne("Anna", "Smith", 32),
  new Personne("Peter", "Jones", 29)
];
```

On utilise la méthode « **sort** » pour trier nos personnes :


```
// premier tri de la liste de personnes
personnes.sort();
```

On fait une fonction privée pour retrouver l'index d'une personne dans le tableau. Je parcours mes personnes et je teste si l'index de la personne est égal à l'index que je recherche.

Voici à quoi ressemble cette première fonction privée :

```
// fonction privée pour retrouver l'index d'une personne dans le tableau, -1 autrement
// il faut comparer avec toString()
function _trouverPersonne(p) {
    let idx = -1;
    for (let i = 0; i < personnes.length; i++) {
        if (p.toString() === personnes[i].toString()) {
            idx = i;
            break;
        }
    }
    return idx;
}
```

Je fais ensuite une deuxième fonction privée qui va me permettre d'ajouter une personne dans mon tableau. Je rajoute cette personne grâce à son index et je retri mon tableau. Pour tester si elle existe, je me sers de la fonction que j'ai créée précédemment, si elle me retourne « -1 » cela veut dire que la personne n'existe pas dans mon tableau et que donc je peux l'ajouter.

Voici le résultat :

```
// ajouter une personne dans la liste des personnes si pas trouvée
function ajouterPersonne(p) {
    let idx = _trouverPersonne(p);
    if (idx < 0) {
        personnes.push(p);
        personnes.sort();
    }
}
```

Finalement, pour supprimer une personne, je teste via la méthode « **trouverPersonne** » que la personne que je veux supprimer existe bien dans mon tableau, pour cela, la méthode doit me retourner l'index de la personne que je veux supprimer, si cette personne existe bien, je la supprime de mon tableau via « **splice()** »

```
/ supprimer une personne dans la liste des personnes si trouvée
function supprimerPersonne(p) {
    let idx = _trouverPersonne(p);
    if (idx >= 0) {
        personnes.splice(idx, 1);
    }
}
```

Pour le contrôleur, on peut faire comme en java FX et el séparer en 3 parties. La première sera le démarrage de l'application avec ma fonction « **document.onreadystateChange** », L'attribut **onreadystatechange** est en fait un événement qui se déclenche à chaque fois que l'attribut **readyState** change de valeur.

```
/*
 * 1. DOM PRET : DEMARRAGE DE L'APPLICATION
 */
document.onreadystatechange = function () {
    if (document.readyState === "complete") {
```

```

    _afficherPersonnes();
  }
};

```

En deuxième partie, les méthodes privées de lecture et d'écriture de la vue. On va donc retrouver une fonction qui va afficher mes personnes, une qui va afficher les infos d'une personne dans le formulaire et une qui va lire le contenu de ce qui a été entré dans le formulaire pour créer une nouvelle personne.

Je commence par la méthode qui va afficher la liste de données au bas de la vue, je fais une liste pour ça et je vais rajouter des éléments dans cette liste. Je parcours mon tableau via une boucle « **for** », j'utilise la méthode « **selectionnerPersonne()** » quand je clique sur el bouton, je lui spécifie la personne courante de la boucle et ensuite j'affiche cette personne.

```

// affiche la liste des données au bas de la vue (avec du HTML généré)
function _afficherPersonnes() {
  let txt = "<ul>";
  for (let i = 0; i < personnes.length; i++) {
    txt += '<li><a href="#" onclick="selectionnerPersonne(' + i + ');">' + personnes[i] + '</a></li>';
  }
  txt += "</ul>";
  document.getElementById("info").innerHTML = txt;
}

```

Pour afficher une personne, je me sers simplement de son « id » et je dis que c'est égal à l'attribut correspondant, le champ nom est égal à l'attribut nom etc...

```

// affiche les infos d'une personne dans le formulaire
function _afficherInfosPersonne(p) {
  document.getElementById("prenom").value = p.prenom;
  document.getElementById("nom").value = p.nom;
  document.getElementById("age").value = p.age;
}

```

Pour lire les informations, je lis les informations entrées dans les champs de mon formulaire et avec ce qui est écrit je vais créer une nouvelle personne et l'afficher dans mon tableau.

```

/ lit le contenu des masques de saisie pour en faire une personne
function _lireInfosPersonne() {
  let p = null;
  let prenom = document.getElementById("prenom").value;
  let nom = document.getElementById("nom").value;
  let age = parseInt(document.getElementById("age").value);
  if (prenom.length > 0 && nom.length > 0 && age > 0) {
    p = new Personne(prenom, nom, age);
  }
  return p;
}

```

La troisième partie, sont les méthodes publiques qui sont nécessaires à la vue., la première méthodes « **selectionnerPersonne()** », va me servir à récupérer les informations d'une personne, via une autre méthode.

```

// appelée depuis la vue pour afficher les données de la personne sélectionné
function selectionnerPersonne(i) {
  _afficherInfosPersonne(personnes[i]);
}

```

La deuxième fonction me sert à ajouter une personne. Je lis les infos de cette personne, je l'ajoute et finalement je l'affiche.

```
// appelée depuis la vue pour ajouter une personne
function ajouter() {
  let p = _lireInfosPersonne();
  ajouterPersonne(p);
  _afficherPersonnes();
}
```

Finalement, la dernière me sert à supprimer une personne sur le même principe que la fonction précédente.

```
// appelée depuis la vue pour supprimer une personne
function supprimer() {
  let p = _lireInfosPersonne();
  supprimerPersonne(p);
  _afficherInfosPersonne();
}
```

2.11 Exercice 11

En partant de l'exercice 10, le défi d'ici est de transformer les modules de code de cet exercice en créant des classes pour « **Personne** », « **worker** » et « **Ctrl** ».

Ensuite, la seule grande difficulté est de savoir comment créer un objet contrôleur pour qu'il soit disponible pour la vue ? La solution est donnée dans la consigne, l'objet doit être stocké dans l'objet « **Window** » du navigateur par ce que les méthodes de « **indexCtrl** » sont appelés depuis « index.html ». L'objet « **window** » doit être ajouté à la variable ou on ne met rien et la variable devient globale. On ne fait surtout pas de let.

```
/*
 * DOM PRET : DEMARRAGE DE L'APPLICATION dans le fichier indexCtrl.js
 */
document.onreadystatechange = function () {
  if (document.readyState === "complete") {
    window.ctrl = new Ctrl(); // ou ctrl = new Ctrl();
    ctrl.afficherPersonnes();
  }
};
```

Important: il faut faire référence aux attributs et méthodes de la classe en utilisant le mot-clé **this**. Comme c'est le Ctrl qui crée le Worker, on peut accéder aux attributs en utilisant **this.wrk.personnes** ou **this.wrk.ajouterPersonne(p)** pour une méthode.

```
/*
 * Première ligne de la classe Ctrl
 */
class Ctrl {
  constructor() {
    this.wrk = new Worker();
    this.afficherPersonnes();
  }
}
```

Je vais ensuite simplement mettre toutes les fonctions qu'il y avait en dessous dans ma classe Ctrl, le mot clé « **function** » va donc disparaître. Il va falloir faire attention à où je dois appeler les choses depuis mon worker.

Voici à quoi cela ressemble :

```
/*
 * Première ligne de la classe Ctrl
 */
class Ctrl {
  constructor() {
    this.wrk = new Worker();
    this.afficherPersonnes();
  }

  /*
   * 2. METHODES PRIVEES DE LECTURE/ECRITURE DANS LA VUE
   */

  // affiche la liste des données au bas de la vue (avec du HTML généré)
  afficherPersonnes() {
    let txt = "<ul>";
    for (let i = 0; i < this.wrk.personnes.length; i++) {
      txt +=
        '<li><a href="#" onclick="ctrl.selectionnerPersonne(' +
        i +
        ');">' +
        this.wrk.personnes[i] +
        "</a></li>";
    }
    txt += "</ul>";
    document.getElementById("info").innerHTML = txt;
  }

  // affiche les infos d'une personne dans le formulaire
  afficherInfosPersonne(p) {
    document.getElementById("prenom").value = p.prenom;
    document.getElementById("nom").value = p.nom;
    document.getElementById("age").value = p.age;
  }

  // lit le contenu des masques de saisie pour en faire une personne
  lireInfosPersonne() {
    let p = null;
    let prenom = document.getElementById("prenom").value;
    let nom = document.getElementById("nom").value;
    let age = parseInt(document.getElementById("age").value);
    if (prenom.length > 0 && nom.length > 0 && age > 0) {
      p = new Personne(prenom, nom, age);
    }
    return p;
  }

  // appelée depuis la vue pour afficher les données de la personne sélectionnée
  selectionnerPersonne(i) {
    this.afficherInfosPersonne(this.wrk.personnes[i]);
  }

  // appelée depuis la vue pour ajouter une personne
  ajouter() {
    let p = this.lireInfosPersonne();
    this.wrk.ajouterPersonne(p);
    this.afficherPersonnes();
  }

  // appelée depuis la vue pour supprimer une personne
  supprimer() {
    let p = this.lireInfosPersonne();
    this.wrk.supprimerPersonne(p);
    this.afficherPersonnes();
  }
}
```

Attention à ne pas oublier de faire référence au « **ctrl** » dans l'html quand on va appeler les fonctions.

```
<input type="button" value="Ajouter" onclick="ctrl.ajouter();">
<input type="button" value="Supprimer" onclick="ctrl.supprimer();">
```

On va ensuite faire la classe « **personne** » sur le même principe que pour « **ctrl** ». On a toujours les mêmes attributs et notre « **toString** ».

```
class Personne {
  constructor(prenom, nom, age) {
    this.prenom = prenom;
    this.nom = nom;
    this.age = age;
  }

  toString() {
    return this.nom + " " + this.prenom + " (" + this.age + ")";
  }
}
```

Et finalement, on va s'occuper de la classe « **worker** », on met notre tableau de personne dans son constructeur et on adapte les fonctions déjà existantes avec le mot clé « **this** ».

```
class Worker {

  constructor() {
    this.personnes = [
      new Personne("John", "Doe", 44),
      new Personne("Anna", "Smith", 32),
      new Personne("Peter", "Jones", 29),
    ];
    this.personnes.sort();
  }

  // fonction privée pour retrouver l'index d'une personne dans le tableau, -1 autrement
  _trouverPersonne(p) {
    let idx = -1;
    for (let i = 0; i < this.personnes.length; i++) {
      if (p.toString() === this.personnes[i].toString()) {
        idx = i;
        break;
      }
    }
    return idx;
  }

  // ajouter une personne dans la liste des personnes
  ajouterPersonne(p) {
    let idx = this._trouverPersonne(p);
    if (idx < 0) {
      this.personnes.push(p);
      this.personnes.sort();
    }
  }

  // supprimer une personne dans la liste des personnes
  supprimerPersonne(p) {
    let idx = this._trouverPersonne(p);
    if (idx >= 0) {
      this.personnes.splice(idx, 1);
    }
  }
}
```

2.12 Exercice 12

Dans cet exercice, nous allons voir les différentes manières d'écrire une fonction.

La première méthode ci-dessous, est exécutée et affiche 1 dans la console avec un simple « **console.log** », rien de très compliqué.

```
function a() {  
  let val = 1;  
  console.log(val) ;  
}  
a();
```

A noter que si on écrit la dernière ligne avant la fonction, cela fonctionne aussi !

La deuxième méthode nous permet de déclarer une fonction dite « **anonyme** » on se sert d'une variable « **b** » et pas directement du mot clé « **function** ».

```
function a() {  
  let val = 1;  
  console.log(val) ;  
}  
a();
```

La troisième méthode, nous permet de déclarer une fonction « **flèche** » ou en anglais « **arrow function** » pour faire ça on se sert du caractère « **=>** » pour faire une flèche et indiquer comment on arrive au résultat d'une variable par exemple.

```
// Déclaration d'une fonction flèche (arrow function) ; 3 cas de figure  
let c = (a,b) => a + b ;  
console.log(c(2,5)) ; // = 7  
  
let d = val => val*val ; // avec un paramètre  
console.log(d(5)); // = 25  
  
let e = (a,b) => { // avec plusieurs instructions  
  let somme = a+b;  
  return somme;  
};  
console.log(e(5,7)); // = 12
```

Il est, quelquefois, nécessaire de créer une fonction et l'exécuter directement : IIFE « **Immediately-Invoked Function Expression** ». Voici deux manières d'écrire une fonction JS IIFE:

```
(function() {  
  let val = 3;  
  console.log(val) ;  
})(); // => exécution et affichage de 3 dans la console  
  
(() => {  
  let val = 4;  
  console.log(val) ;  
})(); // => exécution et affichage de 4 dans la console via une fonction flèche
```

Pour l'exercice, j'ai réalisé un « **additionneur** », j'ai trois boutons sur mon interface et quand j'appuie dessus, leur valeur s'incrémente de 1.

Pour réaliser cette petite application j'ai fait un fichier html simple avec les 3 boutons :

```

<html>
  <!--
  But : page index de l'ex 12
  Auteur : Malori Burgy
  Date : 22.05.23 / V1.0
  -->

  <!-- entête de la page -->
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="icon" href="#" />
    <link rel="stylesheet" href="styles/main.css" >
    <script type="text/javascript" src="js/indexCtrl.js" async></script>
    <title>Ex12</title>
  </head>

  <!-- corps de la page -->
  <body>
    <div id="container">
      <p>Cliquez pour incrémenter les compteurs</p>
      <button id="button1" onclick="actionCompteur(1);">Compteur No 1</button>
      <button id="button2" onclick="actionCompteur(2);">Compteur No 2</button>
      <button id="button3" onclick="actionCompteur(3);">Compteur No 3</button>
    </div>
  </body>
</html>

```

En ce qui concerne le fichier javascript, j'ai réalisé une fonction « **additionne** » qui va incrémenter la valeur de mes boutons de 1 à chaque fois que je clique dessus. Pour que je puisse faire une fonction pour 3 boutons, je me suis servi des index, selon l'index qui est cliqué j'incrémente le bouton correspondant.

```

function additionne(){
  let count=0;
  return function(){
    return count++;
  };
}

let mesBoutons = [additionne(), additionne(), additionne()];

function actionCompteur(idx){
  let val = mesBoutons[idx-1].call();
  document.getElementById("button"+idx).innerHTML="Compteur No "+idx+": <b>"+val+"</b>";
}

```

2.13 Exercice 13

Dans cet exercice, nous allons introduire le concept de cookies. Par l'intermédiaire de cookies, il est possible d'écrire sur le poste de l'internaute des informations de manière permanente. Les cookies qui se présentent comme de petits fichiers texte peuvent entreposer un nombre limité de données mais cette technique peut se révéler précieuse pour conserver des contenus de variables mémoires réutilisables au fil de la navigation au travers des nombreuses pages de votre site web (conservation d'identifiants, de mots de passe même si cela est dangereux, de préférences utilisateur...).

Il est évidemment possible sur maintenant presque tous les sites d'interdire que les cookies soient utilisables pour cibler nos préférences sur un site de vêtements par exemple.

En Javascript, on peut créer un cookie avec « **document.cookie** » il nous suffit de l'affecter à une chaîne de caractères. Il existe une multitude de paramètres pour définir les cookies.

Dans le cadre de cet exercice, on va retenir les 3 principaux :

Le nom : un même site peut enregistrer plusieurs cookies différents il faut donc savoir les identifier

Le contenu : c'est l'élément qu'on veut stocker dans ce cookie et qu'on récupérera plus tard

La date d'expiration : c'est la date à partir de laquelle le cookie cessera d'exister.

Voici un exemple de création d'un cookie :

```
document.cookie = "cookie_exercice_13=" + contenu + "; expires=" + dateExpiration.toUTCString() + "; path=/";
```

Pour récupérer ce qui est stocké dans le cookie, il faut lire le contenu de l'objet « **document.cookie** » il nous sera ensuite possible d'extraire les informations dont on a besoin.

Pour l'exercice, il est question de récupérer le nom et le prénom que l'utilisateur saisira dans un champ texte via un cookie. L'enregistrement du cookie est déclenché quand l'utilisateur appuie sur le bouton « **enregistrer** ». La date d'expiration du cookie doit être de +3 minutes par rapport à la date et l'heure actuelles. Le nom du cookie est "cookie_exercice_13".

Au démarrage de la page, il faut vérifier si un cookie existe. Si c'est le cas, il faut lire le contenu du cookie afin de récupérer le prénom et le nom qui y sont stockés. Une fois ces 2 informations récupérées, il faut les réafficher dans le formulaire.

Voici la maquette de la vue :



Prénom : Nom :

Je remplis donc le fichier javascript fournie en me servant des commentaires présents dans le code et de la théorie ci-dessus.

Je récupère les valeurs des champs « noms » et « **prénoms** », je crée un objet json avec les valeurs que je viens de récupérer. Je transforme cet objet json en chaîne de caractères et j'utilise la méthode « **setCookie** » pour créer mon cookie.

```
function saveData() {  
    // Récupérer les valeurs contenues dans les champs "prenom" et "nom".  
    let nom = document.getElementById("nom").value;  
    let prenom = document.getElementById("prenom").value;  
    // Créer un objet JSON "personneJson" contenant le prénom et le "nom".  
    let personneJson = { nom: nom, prenom: prenom };  
    // Convertir l'objet JSON "personneJson" en chaîne de caractères.  
    let stringPersonne = JSON.stringify(personneJson);  
    // nom du cookie  
    // Appeler la méthode "setCookie" en passant en paramètre la chaîne de caractères "stringPersonne"  
    // convertie et en précisant le délai de 3 minutes.  
    setCookie(stringPersonne, 3);  
}
```


Je génère dans la fonction ci-dessous, je génère une date d'expiration, je crée d'abord une date pour ça et ensuite je lui rajoute les minutes que j'ai en paramètres. Ces deux choses ensemble me feront ma date d'expiration. Je crée le cookie avec el contenu que j'ai en paramètre et la date d'expiration que je viens de créer.

```
function setCookie(contenu, minutes) {
    // Générer une date d'expiration. Il doit s'agir de la date et de l'heure actuelles + le nombre de
    minutes passé en paramètre.
    const d = new Date();
    d.setTime(d.getTime() + minutes * 60000);
    let expire = "expires=" + d.toUTCString();
    // Créer le cookie.
    document.cookie = "monCookie=" + contenu + "; " + expire + ";path=/";
    console.log("Données enregistrées dans le cookie");
}
```

Pour ma fonction « **getCookie** » on va d'abord tester si le cookie existe bien regardant si sa taille est supérieure à 0. Ensuite on va séparer noter cookie où il y a un égal et on va transformer ce qu'il se trouve à droite de cela en objet Json. J'affiche mon cookie pour pouvoir m'aider dans le débbuga mais cela n'est pas obligatoire. Je fini donc par juste remplir mes champs avec les attributs de mon objet json et si on refresh la page les champs auront garder en mémoire ce qu'il y a été tapé dedans.

```
function getCookie() {
    if (document.cookie.length > 0) {
        let cookieSplit = document.cookie.split("=");
        let cookieJson= JSON.parse(cookieSplit[1]);
        console.log("ContenuTxt : " + cookieSplit);

        document.getElementById("prenom").value = cookieJson.prenom;
        document.getElementById("nom").value = cookieJson.nom;
    }

    console.log("Data retrieved from cookie");
}
```

2.14 Exercice 14

Pour cet exercice, nous allons aborder les bases de JQuery, ce que c'est et à quoi ça sert.

Donc, JQuery est une librairie qui va simplifier l'interaction entre Html et JavaScript. Elle à été créer pour simplifier les manipulations du DOM et pour simplifier la vie des développeurs en général. JQuery est très bien documenté, il est facile de trouver des informations en ligne, elle comporte une grande communauté de développeurs aussi ce qui la rends intéressante. Elle est aussi compatible avec de nombreux navigateurs et une multitude de plugins est disponible.

Un objet JQuery est précéder du signe « **\$** », en réalité, c'est une fonction JavaScript. On peut rechercher des éléments dans une page à l'aide d'un sélecteur css par exemple un « **div** ». Don si on note « **\$ (« div »)** » cela va nous retourner tous les éléments « **div** » on va pouvoir par après par exemple lui rajouter une classe avec un « **addClass(..)** ».

Voici ci-dessous un exemple qui va rechercher tous les « **div** » et qui va leur rajouter une classe qui s'appelle ici « **demo** »

```
$("#div").addClass("demo");
```

Pour manipuler une page html on doit attendre qu'elle soit prête et qu'elle ait été chargé entièrement. On va utiliser en JQuery l'évènement « **ready** »

```
$(document).ready(function(){
    // le code jQuery doit se trouver ici
});
```

Parfois une sélection par sélecteurs CSS n'est pas suffisant. JQuery propose une suite de méthodes pour parcourir le DOM.

Il en existe plusieurs : `.parent()`, `.next()`, `.prev()`, `.children()`, `.siblings()` ...

Voici un exemple :

```
$("#button").parent().css("border", "3px solid red");
```

On peut aussi enchaîner les actions sur un set d'éléments. Chaque méthode jquery retourne un set d'élément jquery pour de futures actions (à moins qu'une valeur soit retournée).

Exemple :

```
$("#div").hide();
$("#div").hide().css("color", "blue");
$("#div").hide().css("color", "blue").remove();
```

On peut faire pleins d'autres choses avec JQuery, on peut rajouter des événements pour faire des animations ou changer des couleurs etc...

Voici quelques exemples :

Ici on peut rajouter quelque chose après un éléments quand mon bouton est cliqué :

```
$("#a[target=_blank]")
    .after("<img src='images/open.png' />");
```

On peut aussi appondre du texte :

```
$("#a[target=_blank]")
    .append("(mon texte)");
```

On peut rajouter du css à un certain élément :

```
$("#li a").css({
    color: "red",
    fontWeight: "bold"
});
```

On peut sélectionner certains éléments HTML pour pouvoir les modifier :

```
$("#<li><a></a></li>")
    .find("a")
    .attr("href", "http://www.emf.ch/")
    .html("EMF")
    .end()
    .appendTo("ul");
```

On peut même faire des formulaires avec l'événement « **submit** » :

```
$("#form").submit(function(){
    if ( $("#name").val() === "" ) {
        $("#span.help").show();
    }
});
```

```
    return false;
  }
});
```

Et comme dit plus haut on peut animer des choses :

```
$("#div.block").animate({
  fontSize: "2em",
  width: "+=20%",
  backgroundColor: "green"
});
```

On peut cacher ou montrer des choses sur notre interface :

```
$("#div.block").hide("slow", function(){
  $(this).show("slow");
});
```

Avec un plugin nommer « Ajax », il nous est possible de lire du contenu XML, sur le même principe que les méthodes précédentes :

```
$("#div.block").hide("slow", function(){
  $(this).show("slow");
});
```

On peut aussi lire du JSON :

```
$.getJSON("file.json", function( obj ) {
  for ( var prop in obj ) {
    $("#ul").append(
      "<li>" + prop + ": " + obj[prop] + "</li>");
  }
});
```

On peut charger des fichiers avec la méthode « **.load()** » :

```
$("#div.load").load("file.html");
```

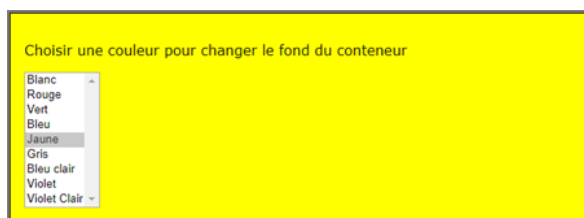
Pour rajouter un plugin, il nous suffit de mettre son lien dans une balise « **script** » comme pour un fichier css :

```
<script src='http://code.jquery.com/jquery.js'></script>
```

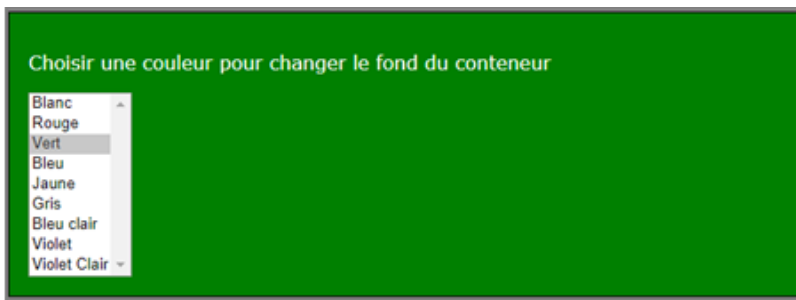
2.15 Exercice 15

On va partir de l'exercice 1 et changer un peu la vue via JQuery.

Au démarrage de l'application la couleur sélectionnée par défaut est jaune :



Et à chaque fois qu'on va sélectionner une nouvelle couleur le fond va s'adapter.



Le body de l'exercice était fourni et il ressemblait à ça :

```
<body>
  <div id="container">
    <form>
      <p id="titre">Choisir une couleur pour changer le fond du conteneur</p>
      <select
        size="9"
        id="couleurs"
        onChange="ctrl.changerCouleur(couleurs.value);"
      >
        <option value="white">Blanc</option>
        <option value="red">Rouge</option>
        <option value="green">Vert</option>
        <option value="blue">Bleu</option>
        <option selected value="yellow">Jaune</option>
        <option value="#F0F0F0">Gris</option>
        <option value="#0FABCD">Bleu clair</option>
        <option value="#FB33BB">Violet</option>
        <option value="#CDB0DC">Violet Clair</option>
      </select>
    </form>
  </div>
</body>
```

On peut voir ma liste déroulante et les noms des couleurs de chaque élément de cette liste.

Dans mon code JavaScript, je suis partie sur un modèle avec classe pour mon contrôleur. Je me suis ensuite occupé de la fonction « **changerCouleur** », je lui passe en paramètre la couleur qui est sélectionné dans la liste et je change la couleur de fond selon cette couleur. Je fais attention si les couleurs sont du rouge, du bleu ou du vert, le texte sera écrit en blanc, pour ça j'ai juste fait une condition et je change la couleur de ma police :

```
$(document).ready(function () {
  ctrl = new Ctrl();
  let couleur = $("#couleurs").val();
  console.log(couleur);
  ctrl.changerCouleur(couleur);
});

class Ctrl {
  constructor() {}

  changerCouleur(couleur) {
    $("#container").fadeOut(1000, function () {
      $("#container").css("background-color", couleur);
      if (couleur === "red" || couleur === "blue" || couleur === "green") {
        $("#container").css("color", "white");
      } else {
        $("#container").css("color", "black");
      }
    });
    $("#container").slideToggle(1000);
  }
}
```

```

    });
  }
}

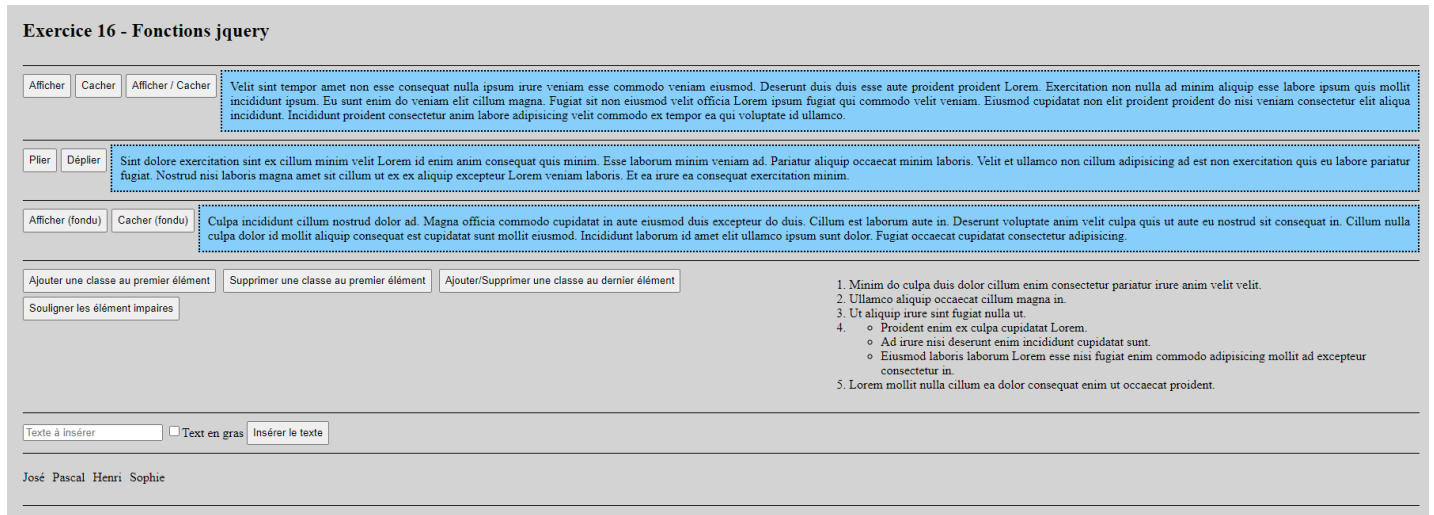
```

2.16 Exercice 16

[EXERCICE NON FINI]

Dans cet exercice, nous avons une interface graphique qui effectuait plusieurs actions via JQuery.

Voici à quoi ressemble l'interface :



Pour la première partie, il fallait simplement cacher ou afficher du texte. Pour réaliser cela, j'ai utilisé mes boutons et quand je clique dessus, cela va cacher ou afficher le texte. Pour le dernier bouton j'ai utilisé l'option « **toggle** » qui me permet d'afficher et cacher le texte quand j'appuie deux fois dessus par exemple.

```

$("#btnShow").click(function () {
    $("#div_1").show();
});
$("#btnHide").click(function () {
    $("#div_1").hide();
});

$("#btnToggle").click(function () {
    $("#div_1").toggle();
});

```

J'ai utilisé la même méthode pour « **plier** » et « **déplier** » mon texte en utilisant « **slideUp** » et « **slideDown** »

```

//plier et déplier
$("#btnPlier").click(function () {
    $("#div_2").slideUp();
});
$("#btnDéplier").click(function () {
    $("#div_2").slideDown();
});

```

Et finalement, avec le fondu, j'ai utilisé « **fadeIn** » et « **fadeOut** »

```
//fendu
$("#btnFonduShow").click(function () {
    $("#div_3").fadeIn();
});

$("#btnFonduHide").click(function () {
    $("#div_3").fadeOut();
});
```

Pour la deuxième partie, j'ai dû rajouter des classes à des éléments et en enlever. Pour ajouter une classe CSS j'ai utilisé « **addClass** ». Pour ne prendre que le premier élément d'une liste, je me suis servi de son index « **li:eq(0)** » et pour le dernier élément, j'ai utilisé « **last()** ».

```
//Ajouter une classe au premier élément: ajoute la classe CSS boldBlueText au premier élément de la
liste.
$("#btnSetClass").click(function () {
    $("li:eq(0)").addClass("boldBlueText");
});

//Supprimer une classe au premier élément: supprimer la classe CSS boldBlueText du premier élément
de la liste.
$("#btnRemoveClass").click(function () {
    $("li:eq(0)").removeClass("boldBlueText");
});

//Ajouter/Supprimer une classe au dernier élément: ajoute la classe CSS boldBlueText au dernier
élément de la liste
//si elle n'y est pas et supprime cette même classe CSS si elle est présente. Vous ne devez pas
utiliser l'instruction if.

$("#btnToggleClass").click(function () {
    $("li").last().toggleClass("boldBlueText");
});
```

Pour le dernier bouton, cela c'est compliquer un peu, j'ai dû sélectionner tous les éléments qui commençaient par un chiffre et pas un point et en plus seulement ceux qui étaient impair.

J'ai d'abord défini que ma balise « **li** » devait être dans un élément « **ol** » car les sous éléments de ma liste sont dans un « **ul** » et en seront donc pas pris en compte et il faut préciser que c'est tous les éléments impair grâce au mot « **even** » qui veut dire impair en anglais.\$

```
//Souligner les éléments impaires: ajoute ou supprime la classe CSS underlineClass au éléments
impaires de la liste.
//Seul les éléments de la liste numérotée doivent être pris en considération, pas ceux de la liste
avec un point devant.
$("#btnPairesClass").click(function () {
    $("ol> li").even().toggleClass("underlineClass"); //ol>li = tout les li dans le ol
});
```

2.17 Exercice 17

Dans cet exercice, nous avons tous été départagé par groupe pour réaliser des présentations sur différents sujets. Les différents sous-chapitres suivants abordent les thèmes vus durant ces présentations.

2.17.1 REST architecture générale

REST est une façon de concevoir des sites web pour qu'ils fonctionnent de manière efficace et puissent communiquer entre eux.

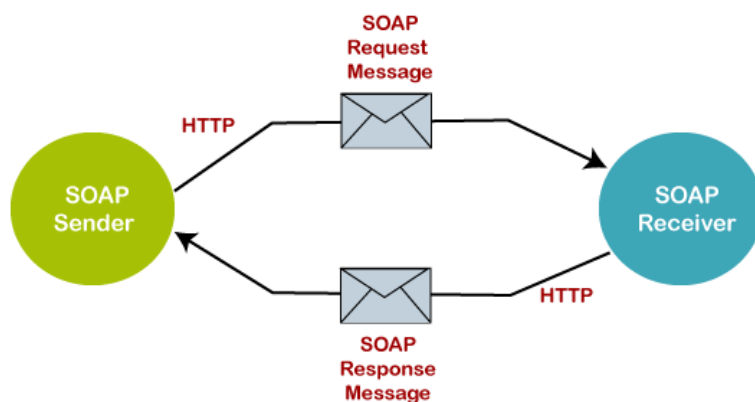
Pour qu'ils puissent communiquer entre eux ils utilisent des requêtes http, voici les différentes actions possibles que l'on peut effectuer avec celles-ci :

- **GET** : Utilisée pour récupérer des données à partir d'une ressource spécifique. Par exemple, obtenir une liste de tâches.
- **POST** : Utilisée pour envoyer des données et créer une nouvelle ressource. Par exemple, ajouter une nouvelle tâche.
- **PUT** : Utilisée pour mettre à jour une ressource existante avec de nouvelles données. Par exemple, modifier les détails d'une tâche.
- **PATCH** : Utilisée pour effectuer une mise à jour partielle d'une ressource. Elle permet de modifier uniquement certaines parties de la ressource.
- **DELETE** : Utilisée pour supprimer une ressource spécifique. Par exemple, supprimer une tâche.

L'architecture REST est conçue pour être "sans état" ou « stateless ». Cela signifie que chaque requête du client doit contenir toutes les informations nécessaires pour que le serveur comprenne et traite la requête. Le serveur ne conserve pas l'état de la session entre les requêtes.

2.17.2 SOAP

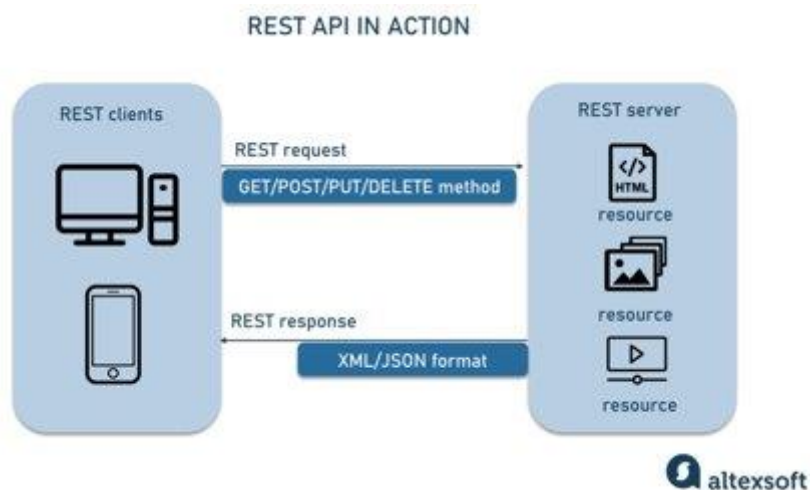
SOAP (Simple Object Access Protocol) est un protocole utilisé pour échanger des données structurées entre des applications via des services web. Contrairement à REST, SOAP est basé sur XML et suit un modèle de communication plus complexe.



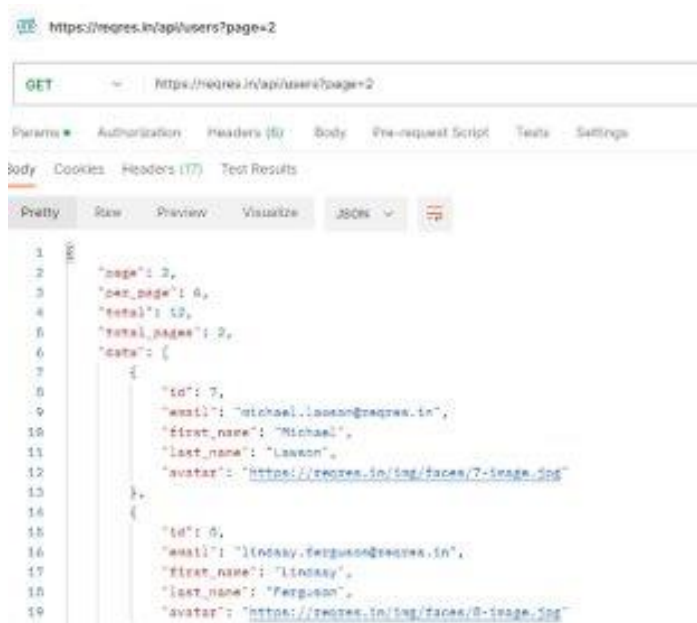
Dans SOAP, les données sont encapsulées dans des messages XML. Chaque message SOAP est composé d'un en-tête (header) et d'un corps (body). L'en-tête peut contenir des informations supplémentaires, telles que des données d'authentification ou de routage, tandis que le corps contient les données principales à échanger.

2.17.3 Comment faire un GET, avec quel outil et détaillez un exemple de GET sur Postman

Un GET ça sert à récupérer des informations dans une base de données, on retourne des informations au format json, xml ou encore html.

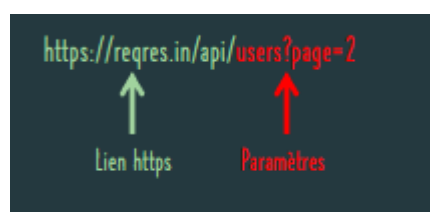


Voici un exemple de GET sur Postman :



On peut voir qu'ici on cherche tous les utilisateurs de la page 2. On tape le lien https suivi des paramètres dans l'url.

La syntaxe ressemble donc à :



Il est aussi important de savoir que les codes d'erreurs que l'on peut recevoir on chacun une signification précise et représente un statut de la requête http, voici un tableau qui regroupe les codes les plus fréquents :

HTTP STATUS CODES	
2xx Success	
200	Success / OK
3xx Redirection	
301	Permanent Redirect
302	Temporary Redirect
304	Not Modified
4xx Client Error	
401	Unauthorized Error
403	Forbidden
404	Not Found
405	Method Not Allowed
5xx Server Error	
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout

2.17.4 en REST, comment faire un POST, avec quel outil et détaillez un exemple de POST sur Postman

Un POST permet d'ajouter un objet dans une API, il n'y a pas de paramètres dans l'url. On pourrait par exemple l'utiliser pour demander un token pour avoir certains droits.

En résumé, une requête POST est utilisée pour envoyer des données au serveur. Elle est couramment utilisée pour créer de nouvelles ressources, comme un nouveau compte utilisateur, en envoyant les informations nécessaires au serveur.

Voici un exemple dans postman :



Et le résultat obtenu est la suivante :



2.17.5 en REST, comment faire un PUT, avec quel outil et détaillez un exemple de PUT sur Postman

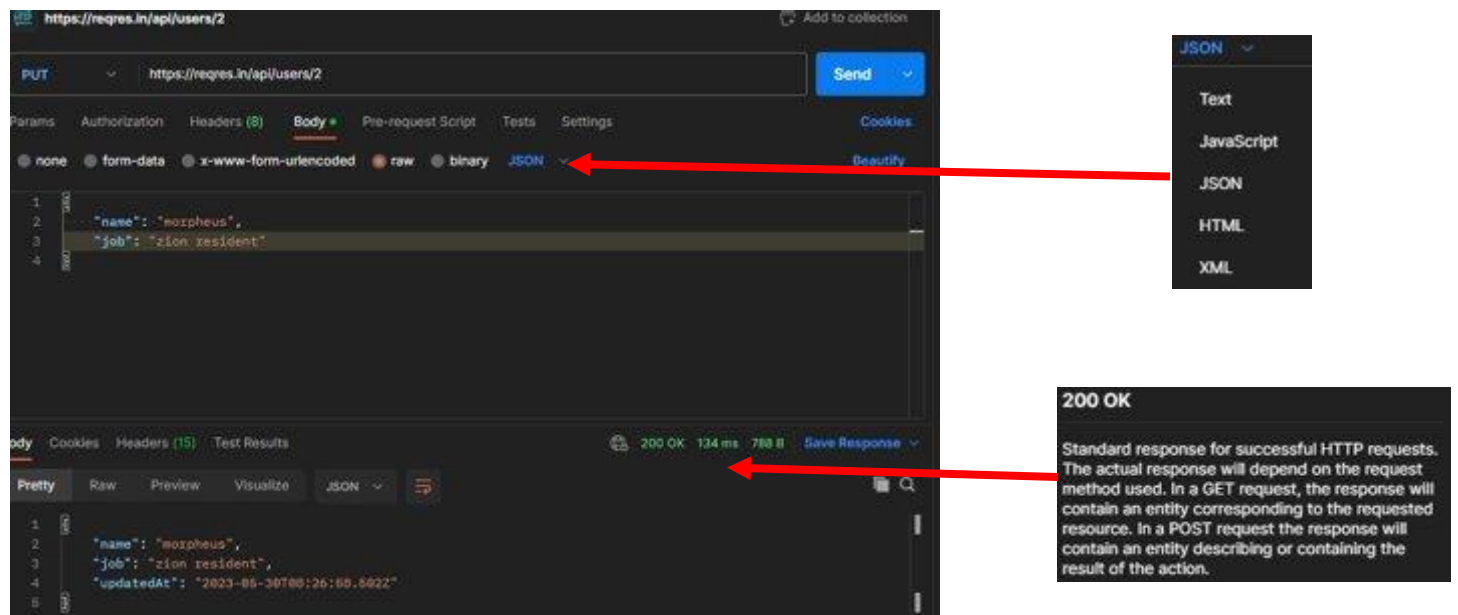
Lorsqu'on utilise la requête PUT, on met à jour une information existante sur un serveur. C'est comme si on remplaçait complètement une fiche par une nouvelle fiche.

Par exemple, imaginons qu'on ait une liste de tâches et qu'on veuille modifier une tâche déjà enregistrée. On envoie une requête PUT au serveur avec les nouvelles informations, comme le titre de la tâche et sa date d'échéance.

Le serveur reçoit cette requête PUT et remplace toutes les informations de la tâche existante par les nouvelles données que l'on a fournies. Ainsi, la tâche est mise à jour avec les informations modifiées.

En résumé, avec une requête PUT, on met à jour une information existante en la remplaçant entièrement par de nouvelles données. Cela permet de modifier complètement l'information enregistrée sur le serveur.

Voici comment faire sur postman :



2.17.6 en REST, comment faire un DELETE, avec quel outil et détaillez un exemple sur Postman

Lorsqu'on utilise la requête DELETE, on supprime une ressource existante sur le serveur. C'est comme si on enlevait complètement une fiche de notre collection.

Par exemple, imaginons qu'on ait une liste de tâches et qu'on souhaite supprimer une tâche spécifique. On envoie une requête DELETE au serveur en précisant l'identifiant de la tâche que l'on veut supprimer.

Le serveur reçoit cette requête DELETE et vérifie l'identifiant fourni. Si l'identifiant correspond à une tâche existante, le serveur la supprime de la liste des tâches.

En résumé, avec une requête DELETE, on supprime une ressource existante sur le serveur. Cela permet de retirer complètement une information de la collection.

Voici un exemple sur postman :

Nous devons sélectionner le mot-clé « DELETE » suivi de l'URL de l'élément qu'on veut supprimer. Nous pouvons en bas à droite, le code de retour : ici nous avons reçu le code « 204 » qui signifie que notre requête a abouti.



On peut ajouter des paramètres grâce au caractère « ? »



Et on peut afficher des données dans le « body », dans la requête de l'exemple il n'y a pas de body parce que c'est une requête delete :



2.18 Exercice 18

Dans cet exercice, nous allons voir qu'on peut aussi créer des Webservices en php. En partant de la base de l'exercice 1, on va tester tout ça.

Je charge les deux fichiers php que je vais utiliser en ligne. Le premier est un webservice de type POST retournant du html.

```
<?PHP
// convert_temp_p_xml.php
// Webservice Conversion de température POST et réponse XML
// Naël Telfser
//
header('Content-Type: application/xml');
$temp = intval($_POST['Temperature'], 10);
$from = $_POST['FromUnit'];
$to = $_POST['ToUnit'];
if ($from === 'C') {
    $tempf = $temp * 9 / 5 + 32;
} else if ($from === 'F') {
    $tempf = ($temp - 32) * 5 / 9;
}
$result = "<temperature>" . $tempf . "</temperature>";
echo $result;
?>
```

Et le deuxième est de type GET et retourne du JSON, normalement on privilégierait cette deuxième méthode. Premièrement parce que comme l'appel du web service ne modifie pas la base de données et que l'on peut l'appeler aussi souvent que l'on veut, on doit utiliser GET et la réponse JSON (JavaScript Object Notation) est plus simple à traiter en JavaScript.

```
<?php
// convert_temp_g_json.php
// Webservice Conversion de température GET et réponse JSON
//
header('Content-Type: application/json');
$temp = intval($_GET['Temperature'], 10);
$from = $_GET['FromUnit'];
$to = $_GET['ToUnit'];
if($from === 'C'){
    $tempf = $temp*9/5+32;
} else if($from === 'F') {
    $tempf = ($temp-32)*5/9;
}
$result = array('temperature' => $tempf);
echo json_encode($result); // fonction php pour transformer des données au format JSON
?>
```

On va ensuite changer la base de l'exercice 1 pour que l'interface corresponde à la maquette ci-dessous :

Convertisseur de température
°C: °F:

Voici à quoi ressemble le code du nouveau body :

```
<body onload="initCtrl()">
    <div id="container">
```

```

<h3>Convertisseur de température</h3>
<div class="form-group">
  <label for="celsius1">°C:</label>
  <input
    type="text"
    size="20"
    id="celsius1"
    name="celsius1"
    placeholder="une température °C"
    autofocus
    onkeyup="ctrl.celsius2Fahrenheit();"
  />
  <label for="fahrenheit1">°F:</label>
  <input
    type="text"
    size="20"
    id="fahrenheit1"
    name="fahrenheit1"
    disabled="disabled"
  />
</div>
</div>
</body>

```

On va pouvoir rencontrer une erreur qui va nous parler de « **cors** » qui veut dire Cross-origin Resource Sharing. Cette erreur nous indique que certaines requêtes qui proviennent de sources « non sûres » peuvent être bloquées.

Dans notre cas ça va un peut être problématique pour le fonctionnement de notre application. Pour remédier à cela, il y a deux solutions.

Premièrement dans l'entête de notre fichier PHP on peut rajouter la ligne :

```
header('Access-Control-Allow-Origin: *');
```

Qui va donc contraindre cette norme. Il existe aussi une extension chrome du nom de « **CORS** » mais on va plutôt privilégier l'entête dans le fichier.

J'ai réalisé deux fichiers JavaScript qui vont me permettre d'utiliser mes deux fichiers php. Un qui va s'appeler httpServ qui va se charger d'effectuer les requêtes « ajax »

```

/*
  But : js de l'exercice 18
  Auteur : Burgy Malori
  Date : 05.06.23 / V1.0
*/

class HttpServ {
  constructor() {}

  celsius2Fahrenheit(degrees, successCallback) {
    let url =
      "https://307.burgym.emf-informatique.ch/Exercices/Exercice_18/php/POST.php";
    let param = "Temperature=" + degrees + "&FromUnit=C&ToUnit=F";

    // envoi de la requête
    $.ajax(url, {
      type: "POST",
      contentType: "application/x-www-form-urlencoded; charset=UTF-8",
      data: param,
      success: successCallback,
      error: this.centraliserErreurHttp,
    });
  }
}

```

```

}
// Ajouter ceci, ôter le errorCallback dans le ajax et son paramètre.
// Dans l'indexCtrl, appelez cette méthode avec une fonction callback
centraliserErreurHttp(httpErrorCallbackFn) {
  $.ajaxSetup({
    error: function (xhr, exception) {
      let msg;
      if (xhr.status === 0) {
        msg = "Pas d'accès à la ressource serveur demandée !";
      } else if (xhr.status === 404) {
        msg = "Page demandée non trouvée [404] !";
      } else if (xhr.status === 500) {
        msg = "Erreur interne sur le serveur [500] !";
      } else if (exception === "parsererror") {
        msg = "Erreur de parcours dans le JSON !";
      } else if (exception === "timeout") {
        msg = "Erreur de délai dépassé [Time out] !";
      } else if (exception === "abort") {
        msg = "Requête Ajax stoppée !";
      } else {
        msg = "Erreur inconnue : \n" + xhr.responseText;
      }
      httpErrorCallbackFn(msg);
    },
  });
}
}

```

Et un autre qui sert juste à faire les conversions degrés en fahrenheit.

```

$(document).ready(function () {
  ctrl = new Ctrl();
  httpServ = new HttpServ();
});

class Ctrl {
  constructor() {
  }

  celsius2Fahrenheit() {
    // Lire les degrés du formulaire
    let degres = parseFloat($("#celsius1").val());
    httpServ.celsius2Fahrenheit(degres, this.OKCelsius2Fahrenheit, this.KOCelsius2Fahrenheit);
  }

  KOCelsius2Fahrenheit(xhr) {
    let erreur = xhr.status + ': ' + xhr.statusText;
    alert('Erreur - ' + erreur);
  }

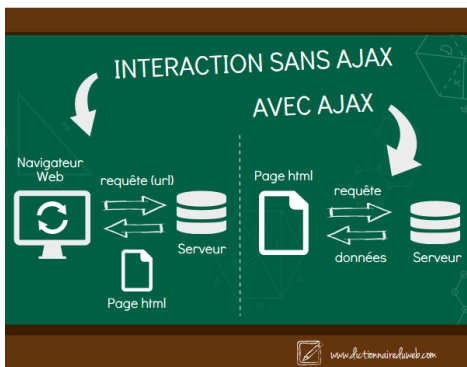
  OKCelsius2Fahrenheit(data) {
    let temp = $(data).find("temperature").text();
    let affiche = temp !== "NaN" ? temp : "";
    $("#fahrenheit1").val(affiche);
  }
}

```

2.19 Exercice 20

Dans cet exercice, nous devons réparer une application qui était déjà presque fonctionnelle.

On a utilisé ici « **ajax** » qui nous permet de directement interagir entre une page web et un serveur, sans nécessairement passer par un navigateur.



Voici la syntaxe d'une fonction ajax :

```
$(document).ready(function(){
    $.ajax({
        //L'URL de la requête
        url: "une/url/au/choix",

        //La méthode d'envoi (type de requête)
        method: "GET",

        //Le format de réponse attendu
        dataType : "json",
    })
})
```

On peut très bien aussi faire des requêtes de type « **POST** » et c'est ce qui est fait dans l'exercice.

Elle prend une multitude de paramètres, mais les plus importants sont :

- **url** : URL de la requête. Seule option strictement obligatoire ;
- **method** (valeur par défaut : GET) : Permet de préciser la méthode d'envoi de la requête (GET, POST ou plus rarement PUT, DELETE, etc.) .
- **data** : Contient les données à envoyer au serveur. Si ces données ne sont pas au format chaîne de caractères, elles seront converties en chaîne .

Voici les changements que j'ai fait dans le code :

Dans `httpServicejs` j'ai changé le lien vers mon fichier php :

```
// Uploade votre propre fichier PHP et adaptez l'URL ci-dessous.
let url = "https://307.burgym.emf-informatique.ch/307/Exercices/Exercice_20/php/login20.php";
```

Dans `indexctrl` j'ai rempli la fonction « **loadCompte** » pour pouvoir changer de vue , ici on peut voir que j'appelle la fonction « **chargerVue** » qui si on va avoir, va logiquement charger du contenu sur ma page de base:

```
loadCompte() {
    this.vue.chargerVue("compte", function(){
        new CompteCtrl();
    })
}
```

Dans `httpService`, il manquait l'information du domaine. On doit préciser « **identifiant.domaine** » pour qu'on puisse recevoir l'information de celui-ci sous forme lisible. Si on ne met pas cela, on risque de se retrouver avec quelque chose comme « **[object]** »

```
let param =  
  "username=" +  
  identifiant.username +  
  "&password=" +  
  identifiant.password +  
  "&domaine=" +  
  identifiant.domaine +  
  "&mail=" +  
  identifiant.mail +  
  "&langue=" +  
  identifiant.langue;
```

3 Projet

3.1 Introduction du projet

Voici maintenant ci-dessous, toute la documentation de mon projet.

3.1.1 Définition: but, objectifs et fonctionnement désiré de votre projet

Dans ce projet, j'aimerais pouvoir créer une application web qui propose des menus de restaurants, il sera possible de choisir des plats selon des catégories et d'ensuite trouver via une map où se trouve ce restaurant.

3.1.2 Explications API

L'api ci-dessous, fournit toutes informations des menus et des boissons proposées dans ce restaurant fictif :

[igdev116/free-food-menus-api: Free API](#)  ([github.com](#))

LA deuxième API me permet d'intégrer une carte a mon application et de placer des points sur celle-ci :

<https://www.openstreetmap.org/>

3.1.3 Site de référence

[igdev116/free-food-menus-api: Free API](#)  ([github.com](#))

<https://www.openstreetmap.org/>

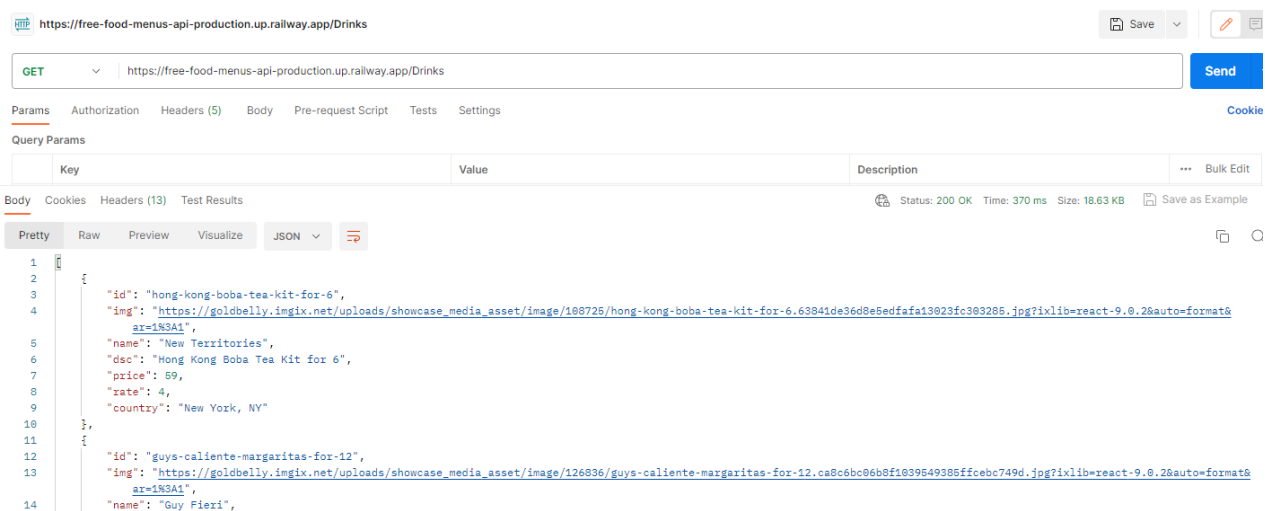
3.1.4 Fonctionnement: requête, réponse, format

Pour faire une requête avec cette api je dois utiliser la syntaxe suivante :

```
https://free-food-menus-api-production.up.railway.app/<params>
```

Les paramètres sont a remplacé avec ce que je veux rechercher par exemple « drinks » pour la liste de toutes les boissons.

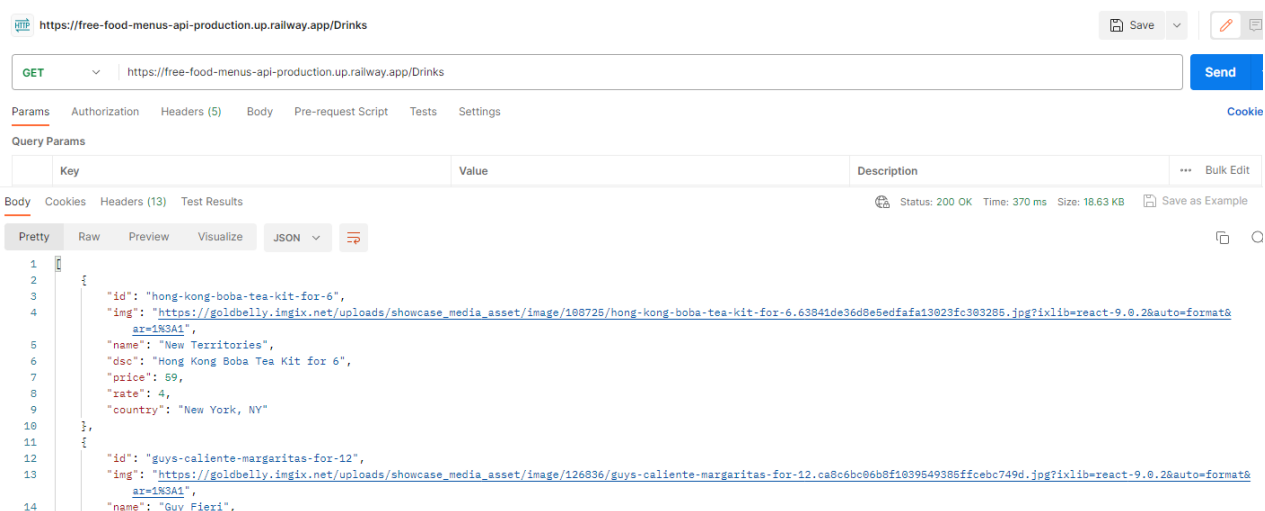
On peut voir ci-dessous que le format des réponses est en json :



3.1.5 Exemple d'utilisation

Ici je cherche dans mon api la liste de toutes les boissons disponibles :

`https://free-food-menus-api-production.up.railway.app/Drinks`



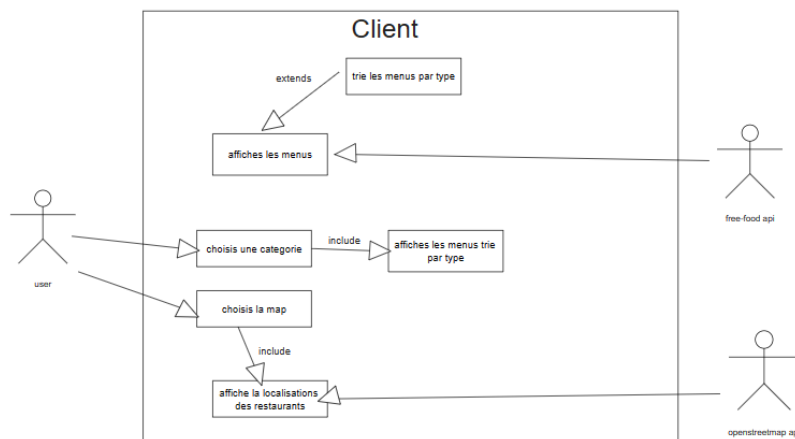
On peut donc voir que cette requête me retourne plusieurs choses :

- **L'id** de la boisson pour l'identifier
- **L'image** de cette boisson
- **Le nom**
- **Le « dsc »** qui est la description de cette boisson
- **Le prix** de la boisson
- **Le « rate »** qui correspond à la note de ce plat
- **Le « country »** le pays où se trouve le restaurant

3.2 Analyse

3.2.1 Diagramme de use cases / Explications des cas

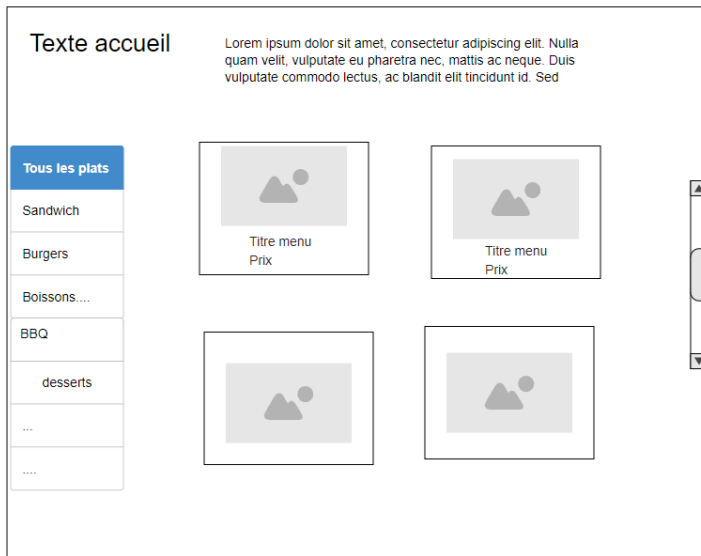
Voici mon diagramme « use cases », donc l'utilisateur arrive sur le site, les données donc les plats sont triés par type, boissons, burgers, desserts etc...L'utilisateur va pouvoir choisir aussi dans le menu d'afficher la map et cela affichera une map avec la localisation de tous les restaurants.



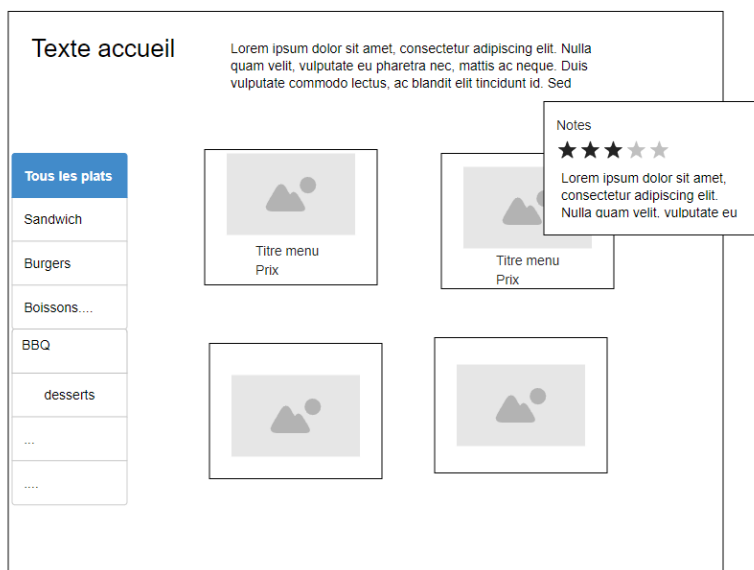
3.2.2 Maquette

Pour ma maquette, je veux rester dans quelque chose d'assez sobre, Une interface claire, les plats apparaîtront dans des sortes de petites bulles, alignées, quand on passe la souris dessus on peut aussi voir le nombre d'étoiles que ce plat a. Pour la page de la map, quelque chose de simple aussi, on affiche la map avec le restaurant qui correspond l'adresse et on affiche évidemment son adresse. En dessous de la map sera aussi afficher les autres plats contenus dans la base de données de ce restaurant.

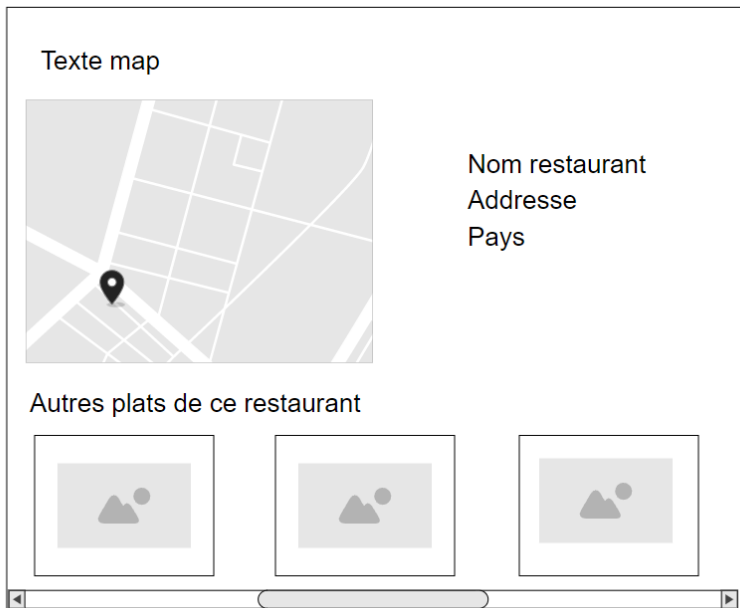
Voici la page d'accueil avec les « filtres » selon les types de plats :



Quand on passe la souris sur une photo pendant un certain temps on va pouvoir voir les notes de ce plat :



Et la page de la map qui va afficher sur la carte l'emplacement du restaurant ainsi que les informations sur l'adresse le nom et le pays. En dessous on va pouvoir retrouver les autres plats de ce même restaurant qui sont dans la base de données

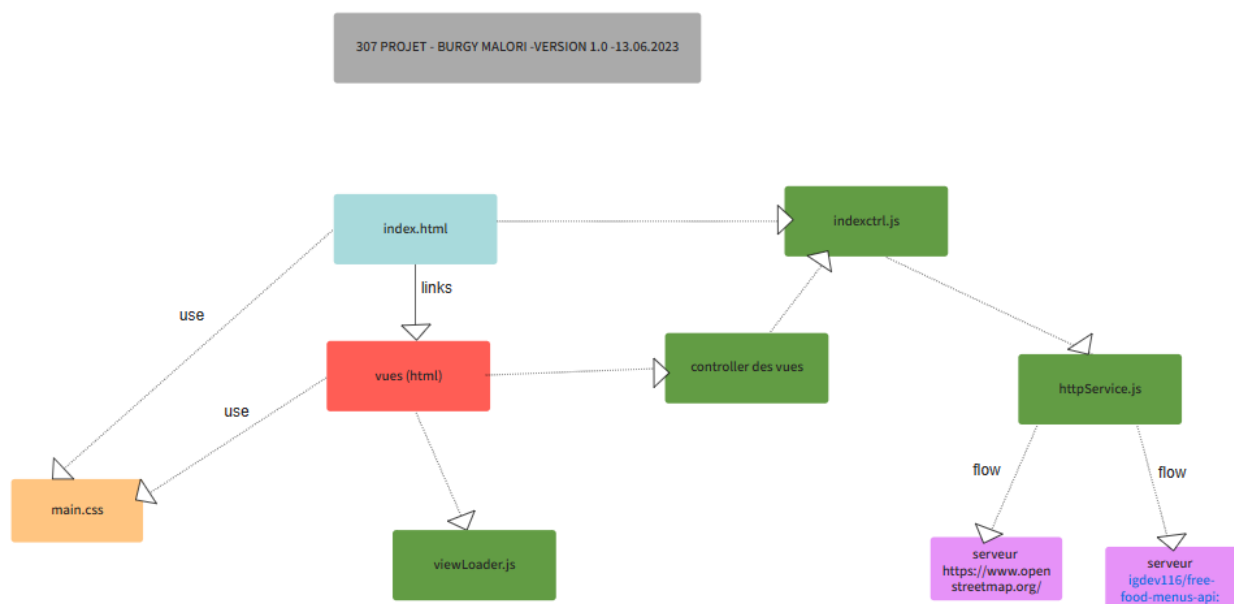


3.3 Conception

3.3.1 Diagramme de navigation

Pour le diagramme de navigation, on peut voir mes fichiers et comment ils sont reliés entre-eux. Le « vue » en rouge, représente toutes les vues de mon application donc boissons.html, home.html etc. Et pareil pour « contrôleur des vue » qui représente donc les contrôleurs de chacune des vues.

On peut aussi voir que « httpService » va faire les requêtes vers mes Api. Et que j'ai un contrôleur pour mes vues qui les charges.



3.4 Implémentation

3.4.1 HTML / CSS (buts des fichiers, éventuellement des extraits de code avec explications)

Dans le fichier html, on va retrouver mon index.html qui est un peu la page de base où je vais charger toutes mes vues.

J'ai également comme cité, des vues pour chacun des boutons présents sur mon interface.

Pour mettre en forme tout ça, j'ai un seul gros fichier css qui est séparé en plusieurs parties par des commentaires pour que je m'y retrouve plus facilement.

Voici mon fichier index.html qui charge toutes mes vues, ou plutôt qui va les contenir :

```
!DOCTYPE html>
<html>
  <!--
    But : page index
    Auteur : Malori Burgy
    Date : 13.06.23 / V1.0
  -->
  <!-- entête de la page -->
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="https://unpkg.com/leaflet@1.6.0/dist/leaflet.css"
      integrity="sha512-
xwE/Az9zrjBIPhAcBb3F6JVqxf46+CDLwfLMHloNu06KEQCAWi6HcDUbeOfBIptF7tcCzusKFjFw2yuvEpDL9wQ=="
      crossorigin="" />
    <script src="https://unpkg.com/leaflet@1.6.0/dist/leaflet.js"
      integrity="sha512-
gZwIG9x3wUXg2hdXF6+rVKLF/0Vi9U8D2Ntg4Ga5I5BZpVkvx1JWbSQtXPSiUTtC0TjtG0mxa1AJPuV0CPthw=="
      crossorigin=""></script>
    <link rel="icon" href="data:;base64,iVBORw0KGgo=">
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.7.0/jquery.min.js"></script>
    <script type="text/javascript" src="js/indexCtrl.js"></script>
    <script type="text/javascript" src="js/burgerCtrl.js"></script>
    <script type="text/javascript" src="js/pizzaCtrl.js"></script>
    <script type="text/javascript" src="js/dessertCtrl.js"></script>
    <script type="text/javascript" src="js/drinkCtrl.js"></script>
    <script type="text/javascript" src="js/mapController.js"></script>
    <script type="text/javascript" src="js/viewLoader.js"></script>
    <script type="text/javascript" src="js/homeCtrl.js"></script>
    <script type="text/javascript" src="js/httpService.js"></script>
    <script type="text/json" src="json/restaurants.json"></script>

    <link rel="stylesheet" href="css/main.css" />
    <title>mon projet</title>
  </head>
  <!-- corps de la page -->
  <body>

    <div id="view"></div>
  </div>
</body>
</html>
```

On peut voir qu'ici, je vais charger tous les autres fichiers que je vais utiliser pour le fonctionnement de mon application. On peut voir aussi une balise « div » qui va servir à afficher toutes mes vues.

Du côté du css, voici un exemple de style que j'ai utilisé pour donner une bordure et de l'ombre à ma map, on peut voir le commentaire qui précède ce bout de code qui me permet de me retrouver dans ma feuille de style :

```
/* -----pour la map-----*/  
  
#mapid {  
    margin-left: 1cm;  
    margin-bottom: 3cm;  
    width: 96%;  
    height: 600px;  
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.2);  
    border: 1px solid #9e9999;  
}
```

3.4.2 Javascript général (buts des fichiers, extraits de code avec explications)

Dans les fichiers javascript « généraux », j'ai mon « indexCtrl ». C'est lui qui va charger toutes mes vues grâce au « viewLoader ».

Voici un exemple de chargement d'une vue :

```
loadBurger() {  
    this.view.viewLoader("burger", function () {  
        new BurgersCtrl();  
    });  
}
```

On peut voir que je fais appelle à mon controller de vue pour charger la vue correspondante et que je crée un nouveau controller spécifique pour cette vue.

Voici un aperçu du fichier « viewLoader » :

```
/*  
 * @author BUrgy Malori  
 * @version 1.0 / 12.06.2023  
 */  
  
class viewLoader {  
    constructor() {}  
  
    viewLoader(view, callback) {  
        $("#view").load("views/" + view + ".html", function () {  
            if (typeof callback !== "undefined") {  
                callback();  
            }  
        });  
    }  
}
```

C'est ici que je vais charger toutes les vues de mon application. A chaque fois qu'on clique sur un bouton, c'est une nouvelle page html qui est chargée.

3.4.3 Javascript spécifiques (buts des fichiers, extraits de code avec explications)

Chacune de mes vues comme cité plus haut, possède son propre controller. Il me permet de dire quelles actions vont être possible sur la page.

Mes controller se ressemblent un peut tous, je gère la navigation en redirigeant sur la bonne page lorsque qu'on clique sur un des « item » de mon menu et j'appelle le service « http » pour faire l'appel à mon API.

Voici un exemple de controller :

```
/*
 * @author Burgy Malori
 * @version 1.0 / 13.06.2023
 */

class DrinkCtrl {
  constructor() {
    http.loadDrink();
    $("#all").click(() => {
      indexCtrl.loadHome();
    });
    $("#burger").click(() => {
      indexCtrl.loadBurger();
    });
    $("#pizza").click(() => {
      indexCtrl.loadPizza();
    });
    $("#drink").click(() => {
      indexCtrl.loadDrinks();
    });
    $("#dessert").click(() => {
      indexCtrl.loadDessert();
    });
    $("#map").click(() => {
      indexCtrl.loadMap();
    });
  }

  afficherErreurHttp(msg) {
    alert(msg);
  }
}
```

Un de mes controller qui est un peu différent, est celui de la map. Je gère aussi évidemment ma navigation mais je gère ici aussi l'affichage de ma map.

Pour afficher les coordonnées sur celle-ci, j'ai un fichier json qui contient la liste des restaurants de mon api ainsi que leurs coordonnées. Je vais parcourir ce fichier json pour pouvoir ressortir toutes ces coordonnées et les afficher sur map.

Voici à quoi ressemble ma fonction dans mon « mapCtrl » :

```
afficherCarte() {
  const RedMarkerIcon = L.icon({
    iconUrl: "https://307.burgym.emf-informatique.ch/Projet/img/redmarkericon.png",
    iconSize: [18, 30],
    iconAnchor: [9, 30],
    popupAnchor: [0, -20],
  });
  const mapid = L.map("mapid").setView(
    [40.732179019666106, -73.98438249553737],
    3
  );
  L.tileLayer("https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png", {
    attribution:
      '&copy; <a href="https://www.openstreetmap.org/copyright">OpenStreetMap</a> contributors',
```

```

    }).addTo(mapid);

    //Ceci va me permettre d'aller chercher mon fichier json et d'ajouter des points sur la map.
    indexCtrl.readTextFile("https://307.burgym.emf-informatique.ch/Projet/json/restaurants.json",
    function (text) {
        var data = JSON.parse(text);

        data.forEach(function (element) {

            console.log(element.coordonnees);

            L.marker(element.coordonnees, { icon: RedMarkerIcon })

                .addTo(mapid)

                .bindPopup(element.nom);
        });
    });
}

```

3.4.4 Descente de code complète pour une action qui fait appel à un service Web

Pour faire les appels à mes apis, j'ai créé un « service », un fichier javascript que j'ai nommé « httpService » qui va me servir à faire toutes les requêtes nécessaires vers mes apis.

Ci-dessous un exemple de requête :

```

loadAll() {
    const self = this;

    $.ajax({
        url: "https://free-food-menus-api-production.up.railway.app/best-foods",
        type: "GET",
        success: function (data) {
            var container = document.getElementById("data");
            container.innerHTML = "";

            var row;

            for (var i = 0; i < data.length; i++) {
                var food = data[i];

                if (i % 4 === 0) {
                    row = document.createElement("div");
                    row.className = "row";
                    container.appendChild(row);
                }
                var square = document.createElement("div");
                square.className = "food-square";

                var name = document.createElement("h3");
                name.textContent = food.name;
                square.appendChild(name);
                var image = document.createElement("img");
                image.src = food.img;
                image.onerror = function () {
                    this.classList.add("image-not-available");
                };
                square.appendChild(image);
                var price = document.createElement("p");
                price.textContent = "Price: " + food.price + " $";
                square.appendChild(price);
            }
        }
    });
}

```



```

    var popup = document.createElement("div");
    popup.className = "description-popup";
    popup.innerHTML = "<p>Notes: ";
    for (var j = 0; j < food.rate; j++) {
        popup.innerHTML += "☆";
    }
    popup.innerHTML += "</p><p>" + food.dsc + "</p>";
    square.appendChild(popup);

    square.addEventListener("mouseover", function () {
        popup.style.display = "block";
    });

    square.addEventListener("mouseout", function () {
        popup.style.display = "none";
    });

    row.appendChild(square);
}
},
error: function (error) {
    console.log(
        "Erreur lors de la récupération des données de l'API :",
        error
    );
},
});
}
}

```

Dans cet exemple, on peut voir que je recherche <https://free-food-menus-api-production.up.railway.app/best-foods> C'est le lien vers mon api. Je vais ensuite remplir ma page avec les données que je suis allé rechercher. Pour cela, je recherche dans ma page l'élément avec l'id « data » et je vais rajouter dans cet élément qui dans mon code est un « div » toute la liste des « best-foods » sous forme de carré. Dans cette fonction je gère aussi le petit pop-up qui s'affiche quand on passe la souris sur le carré du menu.

3.5 Test

Voici les quelques tests que j'ai réalisé pour vérifier le bon fonctionnement de mon projet :

Pour les fonctions internes :

Test	Resultat
Chargement du json	OK : le chargement du fichier n'affiche pas d'erreur et tous les points sont présents sur la map
Requête vers API	OK : Toutes les données s'affiche correctement et sont présente les requêtes sont correcte
Lien avec les librairies externe (jQuery et Leaflet	PARTIEL : le texte des points maps, la légende qui s'affiche quand on clique sur un point ne s'affiche pas toujours, en rafraichissant la page le problème disparaît.

Pour les fonctions externes :

Test	Resultat
Communication avec best-food-API	OK: les données s'affichent correctement
Communication avec openstreetmap	OK : la map s'affiche correctement.

4 Conclusion

Pour conclure, je dirais que le module c'est bien passé. J'ai trouvé que les sujets abordés étaient intéressants et variés.

L'idée de d'abord effectuer les exercices puis le projet est une bonne idée, cela nous prépare un peu à ce que l'on va devoir faire et cela nous mets un peu sur la piste. Cependant, j'ai trouvé qu'ils étaient plutôt long, ce que je veux dire c'est que nous avons passé la plupart du module sur les exercices et donc nous n'avons pas eu beaucoup de temps pour le projet.

En parlant du projet, j'aime beaucoup le web pour le côté créatif, on peut faire du design et « s'amuser » un peu avec notre créativité. Mais comme dit plus haut, le temps était très limité. J'ai dû avancer mon projet à la maison pour être sûr de ne pas être en retard et d'avoir le temps de tout faire en cours.

Malgré ces quelques points négatif, l'ambiance générale du module était très agréable et j'ai apprécié travailler en autonomie.