MEMORY CARD

Fonctionnalités demandées :

- Au commencement du jeu, des cartes sont disposées face cachée à l'écran.

- Le joueur doit cliquer sur deux cartes. Si celles-ci sont identiques, la paire est validée. Sinon, les cartes sont retournées face cachée, et le joueur doit sélectionner une nouvelle paire de cartes.

- Une compteur de temps, avec une barre de progression, s'affiche en dessous du plateau.

- Le joueur gagne s'il arrive à découvrir toutes les paires avant la fin du temps imparti.

- Chaque temps de partie effectuée doit être sauvegardée en base de données.
Avant le début du jeu, les meilleurs temps s'affichent à l'écran.

Côté Front :

L'application est développé en respectant le design pattern **MVC,** et en utilisant également un design pattern **singleton.**

Pour le **MVC** (Modèle, Vue, Contrôleur) le Contrôleur va récupérer tous les événements du navigateur. Ainsi, en JavaScript, on peut récupérer les clics, les entrées dans un « input », les scrolls…

La classe Modèle sert à modéliser les données, c'est-à-dire effectuer un traitement sur les données avant de les enregistrer ou lorsqu'elles remontent de la base de données. On fera les échanges entre le côté client et le côté serveur via des requêtes XMLHttpRequest également appelé Ajax en jQuery.

La classe Vue, va s'occuper de l'ensemble des affichages et des traitements liés à ceux-ci.

Côté Back:

Le traitement des données se fera en PHP et MySQL.

La base de données contient une table score avec une clé primaire (« id ») qui s'auto-incremente à chaque nouvelle entrée de ligne et une colonne « résultat » qui enregistre les résultats. Bien sûr on aurait pu ajouter d'autres colonnes, comme «joueur» pour enregistrer le nom du joueur...

Au niveau du PHP, on enregistre les informations grâce à une classe « BDDManager » qui est appelé pour exécuter des requêtes personnalisées dans notre main.php,
La connexion est créé et fermé à chaque requête

Les informations de connexion à la base de données sont enregistrées dans des variables globales afin d'être accessible partout dans le PHP.

Le jeu de Mémoire

INDEX.HTML

Notre fichier HTML est relativement simple :

- il comporte dans ces balises :

Pour la balise **<head>** :

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Memory Card</title>
    <link rel="icon" href="/assets/favicon.ico" type="image/x-icon">
    <link rel="stylesheet" href="/css/style.css">
    <script src="https://code.jquery.com/jquery-3.6.1.js" integrity="sha
    <script type="module" src="/js/index.js"></script>
  </head>
  <body>
```

1/ une balise <link> qui référence à **style.css** dans lequel on a défini le style de notre jeu.

2/ une balise <script> qui nous lie à la librairie **jQuery**

3/ enfin une balise <script> de type module (c'est important car sans cette précision nous ne pouvons importer des objets dans notre JS principal) et nous pointons vers le fichier **index.js** qui est notre point d'entrée vers le code JavaScript de l'application

Pour la balise **<body>**:

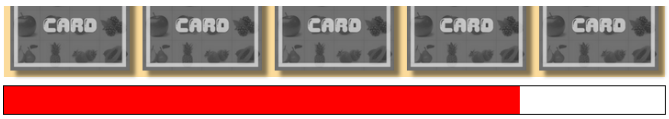

```
<div class="score-display-container">
  <div class="score-display">
    <div class="hall-of-fame">
      <span id="hofTitle">Meilleurs Scores</span>
      <ul id="betterScores">
        <p style="font-size:0.7em">Pas encore de re
        <p style="font-size:0.7em">A toi de jouer !
      </ul>
    </div>
    <button type="button" id="start">
      Nouvelle Partie
    </button>
  </div>
  <div class="opac-screen"></div>
</div>
```

La « **score-display-container** » sert à afficher les meilleurs scores en début de jeu et retourne un petit message de félicitations en cas de victoire ou de défaite. Cette se comportera comme une fenêtre derrière laquelle se trouve un écran opaque qui empêche de cliquer sur la page tant que la fenêtre est affiché, Elle comporte également un bouton de démarrage du jeu.


```
<div class="game-board-container">
  <div class="game-board">
    <div class="row">
      <img id="card0" class="mem-card"/>
      <img id="card1" class="mem-card"/>
      <img id="card2" class="mem-card"/>
      <img id="card3" class="mem-card"/>
      <img id="card4" class="mem-card"/>
    </div>
    <div class="row">
      <img id="card5" class="mem-card"/>
      <img id="card6" class="mem-card"/>
```

La principal de notre application est «**game-board-container** », elle contient le plateau de jeu composé de 4 rangées contenant chacun 5 images identifié par le terme **card0 ...** jusqu'à **card19** afin de pouvoir les retrouver ensuite dans le code.

Grâce aux propriétés css de « Flex » nous allons pouvoir afficher ce tableau en 4 colonnes de 5 rangées afin d'avoir un joli plateau de jeu.




```
    </div>
    <div class="remaining-time-container">
      <div class="remaining-time-bar"></div>
    </div>
</div>
```

Enfin la dernière et qui est une part importante du jeu la «**remaining-time-container** » qui sert à afficher le temps restant ou plus précisément la progression du temps.

Lorsque la barre est pleine, le temps est écoulé et si le joueur n'a pas fini de trouver toutes les cartes alors il a perdu.

INDEX.JS

Comme nous l'avons vu précédemment **index.js** est le point d'entrer dans notre code JavaScript.

```
1  // import = type/module dans l'HTML
2  import { Model } from './model.js';
3  import { View } from './view.js';
4  import { Controller } from './controller.js';
5
```

En en-tête de fichier on voit 3 imports qui nous permettent de créer nos objets **Model, View** et **Controller.**

```
12  $(document).ready(function () {
13    model = new Model()
14    view = new View(model)
15    controller = new Controller(model, view)
16    init()
17  });
```

Au démarrage de la page (ce qui correspond à l'événement **$(document).ready**), on instancie nos classes, ce qui revient à dire que l'on créé nos objets, et qu'on les stocke dans les variables *model* *view* et *controller*.

Ce sont des « **singleton »,** c'est-à-dire que l'objet *model* que l'on a créé est unique.
Il est passé par paramètre à la construction de l'objet *view* et l'objet *controller* reçoit également en paramètree les objets *model* et *view* créés précédemment.

Ainsi *controller* pourra accéder aux méthodes de *model* et *view* (qu'il a reçu à sa construction) sans que l'on recréer perpétuellement de nouveaux objets.

Ce design patterns du « **singleton »** permet d'éviter la multiplication des créations d'objets.
Ce qui est très important surtout pour certains types d'applications par exemple les applications de jeux vidéo dans lesquelles certains objets d'affichage et de rendu graphique peuvent être très lourd.

```
3  function init(){
4      model.loadHallOfFame()
5      controller.init()
6    }
```

Une fois que nos objets sont créés on peut alors lancer la fonction **init()** qui va appeler directement le chargement des affichage des meilleurs scores avec la fonction la méthode **loadHallOfFame()** la classe *model* et **init()** de la classe *controller.*

MODEL.JS

La classe *model* contient dans ses membres le tableau de fruits, le plateau de jeu encore vide, les scores à venir.

Cette classe se charge de récupérer les scores. Elle a une méthode, ***serverRequest*()** qui lui permet d'envoyer des requêtes et de récupérer des réponses depuis le serveur.

Elle affiche également les scores et elle prépare le plateau de jeu en affichant les fruits de façon aléatoire. Elle a aussi quelques méthodes utilitaires qui permettent de trouver un chiffre au hasard, qui renvoie le plateau de jeu ou qui transforme une durée en milliseconde en temps compréhensible par l'être humain.

VIEW.JS

La classe *view* possède dans ses membres différentes variables qui vont lui servir à vérifier le score et à faire s'étendre la barre de progression qui représente l'écoulement du temps.

Les méthodes ***animateUpdate*()** et ***updateProgress*()** permettent de réaliser cette animation de la barre du temps qui progresse.

Les méthodes ***scoreChecker()*** et ***victoryOrDefeat()*** servent à vérifier si on a atteint les 20 points nécessaires à la victoire et afficher le petit message de félicitations ou d'avertissement de la défaite.

CONTROLLER.JS

s'occupe de récupérer les événements sur le DOM et exécute le code attendu derrière chacun de ces événements.

Ainsi la méthode ***startOnClick()*** appelle des méthodes de la classe *model* et de la classe *view* qui permettent de créer le plateau de jeu, de lancer le chronomètre et l'animation de la barre de progression du temps.

La méthode ***returnCardOnClick()*** va, elle, effectuer les opérations attendues lorsque l'on clique sur les cartes : ainsi elle va montrer le côté recto de chaque carte et appeler une autre méthode ***getCardsToCompare()*** qui sert à récupérer les cartes retournées afin de déterminer grâce à la méthode ***compareReturnedCards()*** si on a trouvé une paire.

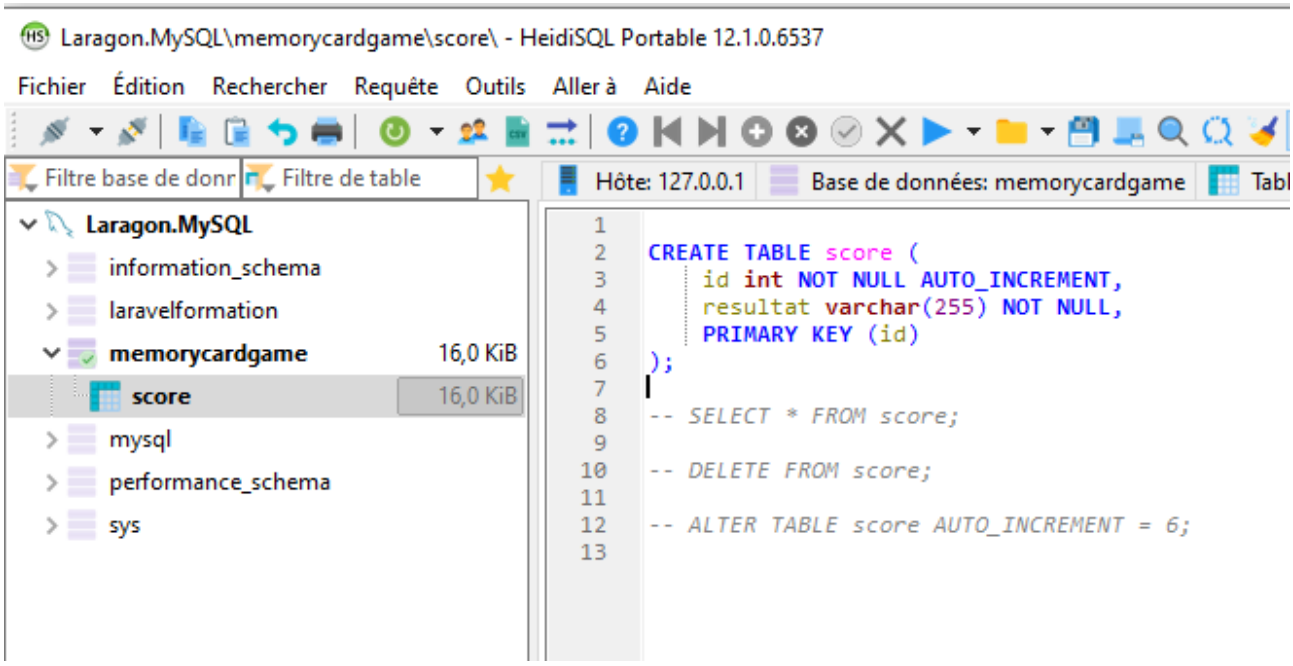
~~ASTUCE~~ → Le fonctionnement de chaque méthode de ces classes est expliquée en détail dans les commentaires de code.

Persistance des données

Afin de conserver nos données on va tout d'abord avoir besoin de créer une base de donnée.

En partant du principe que vous avez un serveur local sur lequel MySQL est installé. Je vous conseille très fortement d'utiliser Laragon plutôt que Wamp afin de disposer d'un serveur local (https://laragon.org/) .

Après avoir créer votre base de données, nommée par exemple : '**memorycardgame**', vous pouvez créer votre table :



La requête ci-dessus vous permet de créer une table dont l'**id** s'incrémente automatiquement à chaque nouvelle entrée de ligne, on créer également une colonne **resultat** qui permettra l'enregistrement des scores du joueur.

MAIN.PHP

Lorsqu'on reçoit un JSON depuis notre fonction **serverRequest()** de la classe *model* celui contient un ordre et une information utilisée dans la requête SQL.

```php
$data = json_decode($_POST['data']);
$dataBaseOperation = $data[0];
$dataScore = $data[1];
```

Ici, on récupère le JSON "posté" lors de la requête faite par l'ajax.
Le JSON est décodé (devient un tableau), on peut alors rentrer chacune de ses données dans une variable.

```php
$db = new BDDManager();
```

Grâce à notre dépendance définie plus haut, on accède à la classe **BDDManager**, on peut donc créer un objet (une instance de classe) : **$db**. **$db** est donc un objet de "type" : **BDDManager.**

```php
$connection = new mysqli($servername, $username, $password, $dbname);
// connect_error est une fonction de la classe mysqli qui vérifie la connection
if ($connection->connect_error) {
  die("Connection failed: " . $conn->connect_error);
}
```

Le language Php contient nativement des outils (classes) de connections aux bases de données (**pdo**, **mysqli…**)

On peut créé un objet **$connection** de "type" : mysqli, sans préciser de dépendance

Paramètres : les paramètres demandées par mysqli sont enregistrés dans un fichier à part (***globals.php***) afin d'être accessible partout dans l'application.

```php
if($dataBaseOperation == 'insert'){
  $req = "INSERT INTO score (resultat) VALUES ($dataScore)";
  $res = $db->insert( $connection, $req);
  $connection->close();
}
```

On reçoit un ordre stocké dans la variable $dataBaseOperation.
Celui-ci nous permet de re-diriger le fil d'exécution du code.

Dans le cas d'un '**insert**', on créé une requête permettant d'insérer dans la table "**score**", la valeur contenue dans **$dataScore**.

On appelle ensuite la méthode 'insert' de l'objet **$db**, en lui passant par référence l'objet **$connection** et la requête que l'on vient de créer.

Lorsque l'opération est terminée, on ferme la connexion pour éviter de conserver une multitude de connections ouvertes.

```php
if($dataBaseOperation == 'read'){
  $req = "SELECT $dataScore FROM score";
  $res = $db->select( $connection, $req);
  $connection->close();
  while ($row = mysqli_fetch_assoc($res)) {
    $scores[] = $row;
  }
  echo json_encode($scores);
}
```

Dans le cas d'un '**read**', on créé une requête permettant de récupérer les résultats contenus dans la table "**score**".
On appelle la méthode 'select' de l'objet **$db**, en lui passant par référence l'objet **$connection** et la requête créée au-dessus.
Lorsque l'opération est terminée, on ferme la connection.
La boucle *while* itère (passe sur chaque élément) sur le résultat de la requête SQL et extrait chaque ligne de résultat en une ligne de tableau associatif (grâce à la fonction **mysqli_fetch_assoc()**) qui est ajoutée au tableau **$scores**.

Ce tableau est encodé en JSON puis renvoyé côté client (navigateur).

On retrouve ce paquet de données (JSON) dans le paramètre '**success**' de notre fonction **ajax()** qui a émis la requête.

BDDMANAGER.PHP

Dans cette classe, on exécute les requêtes transmises par paramètres en utlisant l'objet **$connection** transmis également par paramètre.

On exécute ici les opérations du CRUD (Create, Read, Update, Delete).

```php
public function select( $connection, $req ){
    try{
      $res = $connection->query($req);
      return $res;
    }catch(Exception $e){
        throw New Exception( $e->getMessage() );
    }
    return false;
}
```

La méthode **select()** exécute des requêtes SQL de type "SELECT". Elle retourne un résultat.

```php
public function insert( $connection, $req ){
    try{
      $connection->query($req);
    }catch(Exception $e){
        throw New Exception( $e->getMessage() );
    }
    return false;
}
```

La méthode **insert()** exécute des requêtes SQL de type "INSERT". Elle ne retourne pas un résultat.

GLOBALS.PHP

```php
$GLOBALS["servername"] = "localhost";
$GLOBALS["username"] = "root";
$GLOBALS["password"] = "";
$GLOBALS["dbname"] = "memorycardgame";
```

Les variables $GLOBALS sont accessibles depuis n'importe quelle classe fonction ou fichier php. Ce qui nous permet de les utiliser directement dans la création de connexion dans le **main.php**

```php
$connection = new mysqli($servername, $username, $password, $dbname);
```