TP JavaScript : Smart Dashboard Restaurant

Objectifs du TP

Dans ce projet, tu vas créer une mini-application web de gestion pour restaurant. Elle sera basée sur :

- Une SPA (Single Page Application)
- Le DOM et les événements
- Le stockage local via localStorage
- Un CRUD simple (Create + Read + Delete)
- Des KPIs dynamiques
- Du filtrage et tri
- Une API REST (TheMealDB)
- JSON, try/catch, async/await, fetch…

Tu recevras :

- Le fichier index.html déjà fonctionnel
- Un fichier app.js vide, contenant les noms des fonctions que tu devras compléter
- Ce document pour t'aider à écrire ton JavaScript

I. SPA

Rappel : c'est quoi une SPA ?

SPA = *Single Page Application*

- On ne recharge jamais la page.
- On affiche/masque des sections via CSS + JavaScript.

Méthode JS à connaître

- **document.querySelectorAll("section")**: sélectionner tous les <section> de la page
- **document.getElementById("id")**: sélectionner un élément unique grâce à son id
- **classList.add() / classList.remove()**: ajouter ou retirer une classe CSS d'un élément (ex : afficher/masquer une section)
- **forEach(sec ⇒ sec.classList.add("..."))**: parcourir une liste d'éléments et appliquer une action à chacun (ex : masquer toutes les sections)

À faire : Compléter la fonction showSection(id)

- Masquer toutes les <section>
- Afficher uniquement celle dont l'id est passé en paramètre

II. LocalStorage + JSON

Rappel : C'est quoi localStorage ?

- localStorage est une petite base de données intégrée au navigateur (environ 5 Mo de capacité selon les navigateurs) qui permet de stocker des données de manière permanente sous forme de texte, même après avoir fermé l'onglet ou redémarré l'ordinateur.

Méthode JS à connaître

- Méthodes essentielles :
  - **localStorage.setItem(key, value)**: Enregistrer
  - **localStorage.getItem(key)**: Lire
  - **localStorage.removeItem(key)**: Supprimer
- localStorage **ne stocke que du texte**, donc on utilise :
  - **JSON.stringify(obj)** : pour convertir objet → texte
  - **JSON.parse(txt)** : pour convertir texte → objet

À faire : Puis compléter saveLocal() pour Sauvegarder products dans localStorage

- Créer une variable globale dans app.js :
  - let products = JSON.parse(localStorage.getItem("products")) || [];

III. CRUD (Create + Read + Delete)

Rappel : C'est quoi CRUD ?

- CRUD est un acronyme utilisé en programmation pour désigner les **4 opérations de base** que l'on peut faire sur des données dans une application :
  - **C — Create :** créer une nouvelle donnée (ex : ajouter un plat).
  - **R — Read :** lire ou afficher les données existantes (ex : afficher la liste des plats).
  - **U — Update :** modifier une donnée existante (ex : changer le prix d'un plat).
  - **D — Delete :** supprimer une donnée (ex : retirer un plat du menu).
- Dans ce TP, tu vas implémenter **C**, **R**, et **D**, ce qui te permettra de construire une application complète avec gestion de données côté client. En combinant CRUD avec localStorage, ton application deviendra fonctionnelle, persistante et totalement autonome.

Méthode JS à connaître

- Méthodes essentielles pour Create :
  - **getElementById().value**: lire la valeur d'un champ du formulaire
  - **Array.push()**: ajouter un nouvel élément dans un tableau
  - **Number()**: convertir une chaîne en nombre
  - **alert()**: afficher un message d'information à l'utilisateur

- Méthodes essentielles pour Read:
  - **createElement()**: créer dynamiquement un élément HTML
  - **innerHTML**: insérer du contenu HTML dans un élément
  - **appendChild()**: ajouter un élément HTML dans un parent
  - **forEach()**: parcourir un tab et exécuter une action pour chaque élément
- Méthodes essentielles pour Delete:
  - **confirm()**: afficher une demande de confirmation (oui/non)
  - **Array.splice(i, 1)**: retirer 1 élément d'un tableau à un index i

À faire : Compléter la fonction addProduct(), displayProducts(arr = products) et deleteProduct(i)

- Create:
  - Récupérer name, price, img depuis les inputs
  - Vérifier que les champs ne sont pas vides
  - Ajouter un objet : { name, price, img }
  - Appeler saveLocal(), displayProducts() et updateKPIs()
- Read:
  - Afficher chaque plat dans un
  - Inclure l'image, le nom, le prix et un bouton "Supprimer"
- Delete:
  - Confirmer la suppression
  - Retirer l'élément dans products
  - Sauvegarder → afficher → mettre à jour KPIs

IV. KPIs — Tableaux de bord

Rappel : C'est quoi KPIs?

- Les KPIs (*Key Performance Indicators*) sont des indicateurs chiffrés qui permettent de mesurer rapidement l'état ou la performance d'un système. Dans une application web, un KPI est généralement une valeur calculée automatiquement à partir des données (ex : total du stock, nombre d'éléments, moyenne, pourcentage…). Les KPIs donnent une vue d'ensemble en un coup d'œil et sont souvent utilisés dans les tableaux de bord pour aider à la prise de décision.

Méthode JS à connaître

- Méthodes essentielles :
  - reduce(): sert à accumuler une valeu:
  - products.reduce((somme, p) => somme + p.price, 0);

À faire : Puis compléter updateKPIs()

- Calculer total des prix
- Calculer nombre de plats
- Afficher dans les éléments HTML #kpi1 et #kpi2

V. Filtrage + Tri

Rappel : Comment utiliser sort() et filter() ?

- filter() est une méthode de tableau qui permet de garder uniquement les éléments qui respectent une condition. Elle crée un nouveau tableau filtré, sans modifier l'original.
  - **Syntaxe: array.filter((element) => condition)**
  - **Paramètres**
    - **element**: un élément du tableau (plat, produit, objet…)
    - **condition**: une expression qui renvoie *true* ou *false*
- sort() permet de trier les éléments d'un tableau, en modifiant l'ordre directement dans le tableau. Elle utilise une fonction de comparaison pour définir comment les éléments doivent être triés.
  - **Syntaxe: array.sort((a, b) => comparaison)**
  - **Paramètres:** *a et b, deux éléments du tableau comparés l'un à l'autre*
    - retourne un nombre négatif si a doit venir avant b
    - retourne positif si a doit venir après b
    - retourne 0 si l'ordre ne change pas

Méthode JS à connaître

- Méthodes essentielles Filtrage :
  - filter(): sert à accumuler une valeu:
  - products.filter(p => p.name.toLowerCase().includes(q))
- Méthodes essentielles Tri :
  - sort(): sert à accumuler une valeu:
  - products.sort((a, b) => a.price - b.price)

À faire : Puis compléter filterProducts() et sortAZ()

- Calculer total des prix
- Calculer nombre de plats
- Afficher dans les éléments HTML #kpi1 et #kpi2

VI. API TheMealDB (fetch + async/await)

Rappel : API, fetch, async/await, loader

- Dans ce TP, la partie API sert à récupérer des plats depuis **TheMealDB**, une API publique spécialisée en recettes. Pour contacter une API, on utilise fetch(), qui renvoie une *promesse*. Grâce à async/await, on peut écrire ce code de manière simple et lisible. Le try/catch permet de gérer proprement les erreurs (pas d'internet, API hors service…). Enfin, un **loader** est affiché pendant que l'application attend la réponse du serveur, puis masqué dès que la requête est terminée.

Méthode JS à connaître

- **API**: C'est l'URL qui renvoie une liste de plats dont le nom contient le mot *chicken*:
  - **https://www.themealdb.com/api/json/v1/1/search.php?s=chicken,**
- **fetch**() sert à envoyer une requête HTTP. Il fonctionne en deux étapes :
  - On contacte l'API → reçoit une réponse (response)
    - const response = await fetch(url);
  - On transforme cette réponse en JSON pour obtenir les données
    - const data = await response.json();
- **Async/await**: Tu dois toujours utiliser **await** avec fetch(), sinon tu obtiens une promesse au lieu des données.
- **try/catch: Permet d'éviter que l'application plante en cas d'erreur.**
  - Les erreurs possibles : pas de connexion, API indisponible ou mauvaise URL
- **Loader: Afficher/masquer un indicateur de chargement. Tu utiliseras ces deux lignes au début et à la fin de la fonction API.**
  - loader.classList.remove("hidden");
  - loader.classList.add("hidden");

À faire : Puis compléter loadAPIData()

- Afficher la section API
- Montrer le loader
- Faire un fetch vers TheMealDB
- Récupérer un plat
- Afficher : Nom, Image, Catégorie, Origine
- Gérer les erreurs avec try/catch
- Masquer le loader quand c'est fini

N.B (Initialisation de l'application): À la fin du fichier app.js, ajouter displayProducts() et updateKPIs()

Introduction à Node.js & Express

(Pourquoi dépasser localStorage ?)

Objectifs du TP

- localStorage est pratique pour débuter, mais il a de graves limites :
  - Impossible de partager les données entre plusieurs utilisateurs
  - Impossible d'avoir un vrai accès base de données
  - Pas sécurisé
  - Capacité très limitée (~5 Mo)
  - Pas d'API, pas de backend, pas de logique serveur
  - Il ne fonctionne que dans le navigateur
- Pour créer une vraie application web moderne, on doit avoir un **serveur** qui stocke, enregistre, supprime, filtre ou met à jour des données. C'est là que **Node.js** + **Express** intervient.

Tu recevras :

- Les fichiers index.html, server.js déjà fonctionnelles
- Un fichier app.js, contenant les noms des fonctions que tu devras compléter
- Ce document pour t'aider à écrire ton JavaScript

VII. Installation : Node.js, NPM & Express

Rappel : C'est quoi Node.js ?

- Node.js permet d'exécuter du JavaScript **en dehors du navigateur**, c'est-à-dire côté **serveur**.
- Il permet de lire/écrire des fichiers
- Il permet de créer une API

Rappel : C'est quoi Express.js ?

- Express est un mini-framework qui simplifie la création de routes :
- Dans ce TP on va voir 3 routes:
  - GET: son rôle est de lire et récupérer des données
  - POST: son rôle est d'envoyer et créer des données
  - DELETE: son rôle est de supprimer des données

Méthode Express à connaître

- const express = require("express") ; pour importer le module Express dans ton fichier server.js parceque Express n'est pas installé par défaut.
- const app = express() ; pour créer une application Express. Imagine que "express()" c'est Créer un serveur et que "app" c'est le serveur que tu vas configurer (routes, middlewares…). C'est l'objet principal sur lequel on va ajouter nos routes (GET, POST, DELETE…).
- app.get("/route", (req, res) => {}) ; pour créer une route GET, utilisée pour lire ou récupérer des données. req c'est la requête qui arrive et res c'est la réponse qu'on envoie au client.
- app.post("/route", (req, res) => {}); pour créer une route POST. elle est utilisée pour ajouter/envoyer des données
- app.delete("/route/:id", (req, res) => {}); pour supprimer un élément grâce à un paramètre dynamique. ":id" est le paramètre dans l'URL (e.g /api/products/12). On doit faire "const id = req.params.id;" pour récupérer l'id.
- app.listen(3000) ; pour démarrer le serveur sur le port 3000. Le serveur écoute les requêtes envoyées à l'adresse http://localhost:3000

Manipuler JSON côté serveur

- **fs.readFileSync("db.json")** ; pour lire un fichier texte sur le disque.
- **JSON.parse()** : pour convertir le **t**exte JSON → objet JavaScript.
- **JSON.stringify(data, null, 2)** : pour convertir l'objet JavaScript → texte JSON (Prêt à être sauvegardé dans le fichier db.json) et le null, 2 sert à indenter (format joli)

À faire (Étape installation) :

- Installer Node.js depuis https://nodejs.org/en/download
- Dans le dossier backend/, exécuter
  - npm init -y
  - npm install express cors
- Lancer le serveur
  - node server.js ou npm start

VIII. GET

Rappel : C'est quoi une route GET ?

- GET = Lire
- Elle sert à **récupérer** des données du serveur
- Exemples réels :
  - GET /users → liste des utilisateurs
  - GET /articles → liste des articles
  - GET /products → liste des produits

- Ton code serveur retourne un tableau JSON contenant tous les produits:

```
app.get("/api/products", (req, res) => {
        res.json(db.products);
});
```

Méthode Frontend à utiliser

- **fetch("http://localhost:3000/api/products")** : envoyer une requête HTTP depuis le JavaScript du navigateur vers ton serveur Express. fetch() ne retourne pas directement les données, mais une promesse qui représente la réponse HTTP. Son role :
  - Appeler une URL
  - Contacter une API
  - Récupérer ce que le serveur renvoie (JSON, texte, etc.)

- **response.json()** : Pour convertir la réponse du serveur (texte JSON) en objet JavaScript utilisable. Sans cette méthode Les données reçues sont du texte brut "[{"name":"Pizza","price":80}]". Avec response.json() on obtient [{name:"Pizza", price:80}]. Attention, response.json() est asynchrone donc il doit être précédé de await.

À faire : Implémenter getFromNode()

- Appeler la route GET du serveur
- Récupérer la liste des produits
- L'afficher dans <pre id="nodeResult">

IX. POST

Rappel : C'est quoi une route POST ?

- POST = Créer.
- Elle sert à envoyer des données du frontend vers le backend.
- Lorsque le serveur ajoute un nouveau produit dans db.json, il renvoie le produit ajouté

Méthode JS à connaître

- fetch(url, {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify(data)
  }) : Cette version de fetch() sert à envoyer des données au serveur afin de créer un nouvel élément dans la base (db.json). En résumé, On envoie un objet JavaScript au backend Express, qui va l'ajouter à la liste des produits et renvoyer une réponse JSON. method: "POST" indique que l'on veut ajouter quelque chose. Headers précise que l'on envoie du JSON et body: JSON.stringify(data) convertit l'objet JavaScript → texte JSON pour que le serveur puisse le lire.

À faire : Implémenter addToNode()

- Envoyer un nouvel objet { name, price } au backend
- Attendre la réponse du serveur
- Afficher le résultat dans la balise “pre”

X. DELETE

Rappel : C’est quoi une route DELETE ?

- DELETE = supprimer
- Elle sert à retirer un élément du JSON
- Le serveur supprime un produit dont l’ID est donné dans l’URL (Exempe : DELETE /api/products/1700000123300)

Méthode JS à connaître

- fetch(url, { method: "DELETE" }) : Cette requête fetch() sert à supprimer un élément du serveur, grâce à son identifiant unique. En résumé, Le frontend demande au backend de supprimer un produit précis, et le serveur répond souvent avec un message de confirmation. method: "DELETE" indique au serveur que l’on veut effacer une entrée. L’URL contient /:id, donc on ajoute l’id directement dans l’URL pour dire quel produit supprimer.

À faire : Implémenter deleteFromNode()

- Demander à l’utilisateur un ID via prompt(
- Envoyer une requête DELETE au backend
- Afficher la réponse du serveur

Authentification & Sécurité (niveau débutant)

Objectifs du TP

- Gérer des **rôles utilisateur (admin / user)**
- Mettre en place une **sécurité avec JWT**

1. Gestion des rôles : Admin & User (JWT)

a. Rappel : C'est quoi un rôle utilisateur ?

- Un **rôle** définit ce qu'un utilisateur a le droit de faire dans une application.
- Exemples :
  - **User:** se connecter, voir des données
  - **Admin:** accéder à une page admin, supprimer ou modifier des données, gérer les utilisateurs

Méthode JS à connaître

- document.getElementById("").value.trim()
- errorEl.textContent = "";
- await fetch("http://localhost:3000/api/auth/register", {
  method: "POST",
  headers: {"Content-Type": "application/json"},
  body: JSON.stringify({
  username: username,
  password: password,
  role: role})});
- const data = await response.json();
- Try/Catch
- if (!username || !password) {
  errorEl.textContent = "Veuillez remplir tous les champs";
  return;}
- localStorage.setItem("token", data.token)
- localStorage.removeItem("token");
- localStorage.removeItem("userInfo");

À faire : Compléter handleRegister(), handleLogin() et logout()

- **handleRegister()**:
  - Récupérer : le username, password et le rôle depuis le formulaire
  - Vérifier que tous les champs sont remplis
  - Envoyer une requête POST vers l'API /register
  - Gérer la réponse :
    - afficher un message d'erreur si l'inscription échoue
    - afficher un message de succès si l'inscription réussit
  - Réinitialiser les champs du formulaire après succès

- **handleLogin()**:
    - Récupérer : le username, password
    - Vérifier que les champs ne sont pas vides
    - Envoyer une requête POST vers l'API /login
    - Si la connexion réussit :
        - stocker le token JWT dans localStorage
        - stocker les informations utilisateur (username, role)
        - mettre à jour l'interface utilisateur
    - Afficher un message d'erreur si la connexion échoue
- **logout()**:
    - Supprimer le token JWT du localStorage
    - Supprimer les informations utilisateur
    - Mettre à jour l'interface utilisateur
    - Informer l'utilisateur qu'il est déconnecté

b. Rappel : C'est quoi un JWT (JSON Web Token) ?

- Un **JWT** est une chaîne sécurisée qui contient :
    - l'identité de l'utilisateur
    - son rôle
    - une date d'expiration
- Il remplace le stockage direct de l'utilisateur.
- Format : xxxxx.yyyyy.zzzzz
- Token généré après connexion et envoyé avec chaque requête protégée
- Exemple de payload: {
  email: "admin@test.com",
  role: "admin",
  exp: 1700000000
  }

N.b: cd backend && npm install jsonwebtoken bcryptjs pour installer les 2 libs

Méthode JS à connaître

- **isLoggedIn():**
    - localStorage.getItem("token"): Récupère le token JWT stocké dans localStorage
    - token !== null && token !== ""; Retourne true si un token existe
- **getUserInfo():**
    - localStorage.getItem("userInfo"): Récupère les infos utilisateur stockées
- **updateAuthUI():**
    - if (isLogged && userInfo) {}: Si l'utilisateur est connecté,
    - loginBtn.classList.add("hidden");
    - logoutBtn.classList.remove("hidden");
    - if (header) {header.classList.remove("hidden");}: Affiche le navbar

À faire : Compléter isLoggedIn(), getUserInfo(), updateAuthUI()

- **isLoggedIn()**:
    - Récupérer le token JWT depuis le localStorage

    - Vérifier si :
        - le token existe
        - le token n’est pas une chaîne vide
    - Retourner :
        - true si un token valide est présent
        - false sinon
- **getUserInfo()**:
    - Récupérer les informations utilisateur stockées dans localStorage
    - Vérifier si ces informations existent
    - Si elles existent :
        - convertir le texte JSON en objet JavaScript
        - retourner l’objet utilisateur
    - Sinon :
        - retourner null
- **updateAuthUI()**:
    - Vérifier si l’utilisateur est connecté en utilisant isLoggedIn()
    - Récupérer les informations utilisateur avec getUserInfo()
    - Sélectionner les éléments HTML nécessaires :
        - bouton de connexion
        - bouton de déconnexion
        - zone d’informations utilisateur
        - barre de navigation (header)
    - Si l’utilisateur est connecté :
        - Masquer le bouton de connexion
        - Afficher le bouton de déconnexion
        - Afficher la barre de navigation
        - Afficher le nom et le rôle de l’utilisateur connecté
        - Masquer la section de connexion
        - Afficher la section dashboard par défaut
    - Si l’utilisateur n’est pas connecté :
        - Masquer la barre de navigation
        - Masquer les boutons login et logout
        - Masquer les informations utilisateur
        - Afficher la section login