2 - Récupération des données avec Fetch API

Concepts fondamentaux

L'API Fetch

L'API Fetch est l'outil moderne pour effectuer des requêtes HTTP en JavaScript. Elle retourne des **Promises**, ce qui permet d'utiliser `async/await` pour un code plus lisible.

Async/Await

Ces mots-clés permettent d'écrire du code asynchrone de manière synchrone :

- `async` devant une fonction indique qu'elle retournera une Promise
- `await` met en pause l'exécution jusqu'à la résolution de la Promise

Stratégie de récupération des données

Pour index.html - Liste complète

1. Construire l'URL de votre API (celle déployée sur Render)
2. Effectuer une requête GET vers `/api/cars`

3. Parser la réponse JSON
4. Itérer sur le tableau de voitures
5. Créer dynamiquement les cartes HTML

Pour car.html - Détail d'une voiture

1. Extraire l'ID depuis l'URL (`URLSearchParams`)
2. Construire l'URL `/api/cars/:id`
3. Effectuer la requête GET
4. Parser les données
5. Remplir les différentes sections de la page

Gestion des erreurs

Il est crucial de prévoir :

- **Erreurs réseau** : serveur inaccessible, timeout
- **Erreurs HTTP** : 404 (voiture non trouvée), 500 (erreur serveur)
- **Erreurs de parsing** : JSON mal formé
- **Données manquantes** : champs null ou undefined

Stratégies de gestion :

- Blocs `try/catch` autour des appels async
- Vérification du `response.ok` après fetch
- Messages utilisateur clairs et informatifs
- Valeurs par défaut pour les données manquantes

Principe de base :

```javascript
fetch(url) .then(response =>
response.json()) .then(data =>
console.log(data)) .catch(error =>
console.error(error));
```

Async/Await : une meilleure approche

Au lieu d'enchaîner des `.then()`, on peut écrire du code qui "ressemble" à du code synchrone :

```javascript
async function maFonction() { try {
const response = await fetch(url);
const data = await response.json();
console.log(data); } catch (error) {
console.error(error); } }
```

Avantages :

- Code plus lisible
- Gestion des erreurs plus claire avec try/catch
- Plus facile à déboguer

Les différentes méthodes HTTP

Votre API REST utilise probablement :

- **GET** : récupérer des données (par défaut avec fetch)
- **POST** : créer une nouvelle ressource
- **PUT/PATCH** : modifier une ressource existante
- **DELETE** : supprimer une ressource

✅ Tâches à réaliser

2.1 - Configurer l'URL de votre API

Bonne pratique : Centralisez la configuration

Créez une constante pour l'URL de base de votre API. Cela facilitera les modifications futures.

Questions à vous poser :

- Quelle est l'URL de votre API déployée sur Render ?
- Quel est l'endpoint pour récupérer toutes les voitures ? (ex: `/api/cars` ou `/cars`)
- Faut-il ajouter des headers spécifiques ? (Content-Type, Authorization, etc.)

Conseil : Créez un objet de configuration au début de votre fichier `script.js` :

```JavaScript
const API_CONFIG = { baseURL:
"VOTRE_URL_RENDER_ICI", endpoints: {
cars: "/cars", // ajoutez d'autres
endpoints si nécessaire } };
```

2.2 - Créer une fonction pour récupérer toutes les voitures

Objectif : Écrire une fonction asynchrone qui récupère la liste complète des voitures depuis votre API.

Réflexions importantes :

1. **Gestion des erreurs** : Que se passe-t-il si :
   - L'API ne répond pas ?
   - L'API retourne une erreur (404, 500, etc.) ?
   - Les données reçues ne sont pas au format attendu ?
2. **Vérification de la réponse** :

```JavaScript
if (!response.ok) { throw new
Error(`Erreur HTTP:
${response.status}`); }
```

Structure de la fonction:

- Nom explicite : `fetchAllCars()` ou `getCars()`
- Async/await pour la clarté
- Try/catch pour gérer les erreurs
- Return des données ou null en cas d'erreur

Pseudo-code pour vous guider :

```
FONCTION fetchAllCars(): ESSAYER: (try) faire une requête GET vers l'API SI la réponse n'est pas OK: lancer une erreur convertir la réponse en JSON retourner les données EN CAS D'ERREUR: (catch) afficher l'erreur dans la console (optionnel) afficher un message à l'utilisateur retourner null ou tableau vide
```

2.3 - Créer une fonction pour récupérer une voiture spécifique

Objectif : Écrire une fonction qui récupère les détails d'UNE voiture par son ID.

Cette fonction sera utilisée sur la page `car.html`.

Points à considérer :

- L'endpoint sera probablement : `${baseURL}/cars/${id}`

- Que faire si l'ID n'existe pas dans l'API ?
- Comment gérer une erreur 404 de manière élégante ?

Pseudo-code :

```
Plain Text
FONCTION fetchCarById(id): ESSAYER:
construire l'URL avec l'ID faire une
requête GET SI la réponse est 404:
gérer le cas "voiture non trouvée"
SI la réponse n'est pas OK: lancer
une erreur convertir en JSON
retourner les données EN CAS
D'ERREUR: gérer l'erreur retourner
null
```

2.4 - Tester vos fonctions

Avant de passer à la partie affichage, **testez vos fonctions** !

Méthode de test simple :

```
JavaScript
// En bas de votre script.js
fetchAllCars().then(cars => {
console.log("Voitures récupérées:",
cars); });
```

Vérifications à faire :

- Ouvrez la console du navigateur (F12)

- Les données s'affichent-elles ?
- Le format correspond-il à vos attentes ?
- Les erreurs sont-elles bien gérées ? (testez avec une mauvaise URL)

💡 Points de vigilance

Problèmes CORS potentiels

Si vous voyez une erreur du type :

Access to fetch at '...' from origin '...' has been blocked by CORS policy

Explications :

- CORS (Cross-Origin Resource Sharing) est un mécanisme de sécurité des navigateurs
- Votre API doit autoriser les requêtes depuis d'autres domaines
- Vérifiez la configuration CORS de votre API (côté backend)

Solution temporaire pour le développement :

- Utilisez une extension Chrome comme "CORS Unblock" (SEULEMENT pour le dev !)
- Ou configurez correctement votre API pour accepter les requêtes cross-origin

Gestion des données manquantes

Votre API peut retourner des voitures avec des champs manquants (pas d'image, pas de description, etc.).

Bonnes pratiques :

- Utilisez des valeurs par défaut
- Créez une fonction utilitaire pour valider/nettoyer les données
- Exemple : `const imageUrl = car.imageUrl || 'images/default-car.jpg'`