TP : Modules & Asynchronisme

Découper en Modules (Import/Export)

Fini le fichier unique. Nous allons organiser notre code en plusieurs fichiers qui communiquent entre eux.

A. Le Contrat (src/types.ts)

Créez le fichier et définissez la structure de vos données.

```
// src/types.ts
export interface Utilisateur {
   // TODO: Ajoutez les propriétés : id (number), nom (string), email (string)
}
```

B. Les Données (src/service.ts)

Ce fichier va simuler notre base de données. Il doit connaître le type Utilisateur.

```
// src/service.ts
import { Utilisateur } from "./types";

// TODO: Créez un tableau de 3 utilisateurs
export const mockUsers: Utilisateur[] = [
   { id: 1, nom: "Alice", email: "alice@test.com" },
   // Ajoutez-en d'autres...
];
```

C. Le Point d'Entrée (src/index.ts)

Ce fichier orchestre tout.

```
// src/index.ts
import { mockUsers } from "./service";

console.log(`Nombre d'utilisateurs : ${mockUsers.length}`);
```

Lancer le code : npx ts-node src/index.ts

Les Promesses (Promise<T>)

Dans la vraie vie, les données n'arrivent pas instantanément. Elles arrivent "plus tard". En JS/TS, cela s'appelle une **Promesse**.

Modifiez src/service.ts pour ajouter cette fonction :

```
// src/service.ts

// Notez le type de retour : on promet de renvoyer un tableau d'utilisateurs plus tard
export function fetchUtilisateurs(): Promise<Utilisateur[]> {
    return new Promise((resolve) => {

        // setTimeout simule la lenteur du réseau (2 secondes)
        setTimeout(() => {
            console.log("... Données récupérées !");
            resolve(mockUsers); // La promesse est tenue, on envoie les données
        }, 2000);

    });
}
```

Async / Await

Maintenant, nous devons consommer cette promesse dans notre fichier principal. Si on n'attend pas (await), le programme finira avant d'avoir reçu les données.

Modifiez src/index.ts :

```
// src/index.ts
import { fetchUtilisateurs } from "./service";

// Une fonction qui utilise 'await' DOIT être marquée 'async'
async function main() {
    console.log("Chargement des données...");

    // TODO: Appelez fetchUtilisateurs() avec le mot-clé 'await' devant
    // const users = ...

    // TODO: Affichez les users reçus dans la console
```

```
}

main();
```

Gestion des Erreurs (Try/Catch)

Parfois, l'API plante (Serveur éteint, pas de wifi...). Il faut gérer ce cas.

A. Simuler la panne (src/service.ts)

Mettez à jour la promesse pour qu'elle échoue aléatoirement.

```
// src/service.ts
export function fetchUtilisateurs(): Promise<Utilisateur[]> {
    return new Promise((resolve, reject) => { // Notez l'ajout de 'reject'
        setTimeout(() => {
            const success = Math.random() > 0.5; // 1 chance sur 2

            if (success) {
                resolve(mockUsers);
            } else {
                // TODO: Appelez reject() avec un message d'erreur (string)
            }
        }, 1000);
    });
}
```

B. Attraper l'erreur (src/index.ts)

Protégez votre appel avec un bloc de sécurité.

```
// src/index.ts
async function main() {
    try {
        console.log("Tentative de connexion...");
        const users = await fetchUtilisateurs();
        console.log(users);

    } catch (error) {
```

```
        console.log("error :", error);
    }
}
main();
```

✅ Résultat attendu

Quand vous lancez le script (npx ts-node src/index.ts), vous devez voir :

1. "Tentative de connexion..."
2. (Une pause de 1 à 2 secondes)
3. Soit la liste des objets utilisateurs.
4. Soit le message d’erreur