TP : Le Fetch

Objectif : Ne plus jamais faire planter son application à cause d'un réseau lent ou d'une erreur serveur.

Contexte : Vous allez créer un composant TaskList qui récupère les tâches de votre serveur Node, mais en gérant tous les scénarios possibles (Chargement, Erreur, Succès).

1. La "Sainte Trinité" du State

Pour un appel réseau propre, on a toujours besoin de 3 variables d'état, pas juste une.

Fichier : client/src/components/TaskList.tsx

```
import { useState, useEffect } from 'react';

interface Task {
  id: number;
  label: string;
  isDone: boolean;
}

export default function TaskList() {
  // 1. Les Données (Succès)
  const [tasks, setTasks] = useState<Task[]>([]);

  // 2. L'état de chargement (Pendant l'attente) -> true par défaut
  const [isLoading, setIsLoading] = useState(true);

  // 3. L'erreur (En cas de pépin) -> null par défaut
  const [error, setError] = useState<string | null>(null);

  useEffect(() => {
    // On lance la récupération
    fetchTasks();
  }, []);

  const fetchTasks = async () => {
     // TODO : (voir étape 2)
  };

  // --- RENDU CONDITIONNEL ---
```

```
  // Cas 1 : Ça charge
  if (isLoading) {
    return <div className="loading-spinner">Chargement des tâches...</div>;
  }

  // Cas 2 : Il y a une erreur
  if (error) {
    return <div className="error-message">Error : {error}</div>;
  }

  // Cas 3 : Tout va bien, on affiche la liste
  return (
    <ul>
      {tasks.map(t => (
        <li key={t.id}>{t.label}</li>
      ))}
    </ul>
  );
}
```

2. La Logique Robuste (Try / Catch / Finally)

C'est ici qu'on écrit le code "pro". Complétez la fonction fetchTasks avec la logique suivante :

1. **Reset de l'erreur** au début (au cas où on réessaie).
2. **Try** : On tente l'appel.
3. **Vérification HTTP** : fetch ne plante pas sur une erreur 404 ou 500. Il faut vérifier response.ok.
4. **Catch** : On capture l'erreur réseau ou l'erreur qu'on a levée manuellement.
5. **Finally** : Quoi qu'il arrive (succès ou échec), on arrête le chargement.

```
  const fetchTasks = async () => {
    try {
      // On s'assure que l'erreur est vide avant de commencer
      setError(null);

      const response = await fetch('/api/tasks');

      // ÉTAPE CRUCIALE : Vérifier le status HTTP
      if (!response.ok) {
        throw new Error(`Erreur HTTP: ${response.status}`);
      }
```

```
      const data = await response.json();
      setTasks(data);

    } catch (err: any) {
      // Gestion de l'erreur
      console.error("Erreur fetch:", err);
      setError(err.message || "Impossible de contacter le serveur");

    } finally {
      // C'est fini, on enlève le loader (Succès OU Échec)
      setIsLoading(false);
    }
  };
```

3. Challenge : Tester les cas limites

Intégrez ce composant <TaskList /> dans votre App.tsx et testez les scénarios :

1. **Le Cas Heureux :** Lancez votre serveur (npm run dev côté server). La liste doit s'afficher après un bref "Chargement...".
2. **Le Cas "Serveur Éteint" :** Coupez votre serveur Node (Ctrl+C). Rafraîchissez la page React.
   - *Attendu :* Vous devez voir le message d'erreur au lieu d'une page blanche ou d'un crash.
3. **Le Cas "Réseau Lent" (Simulation) :**
   - Dans votre navigateur -> Inspecteur -> Onglet Réseau (Network).
   - Cherchez le menu "No throttling" (Pas de limitation) et changez-le en "Slow 3G".
   - Rafraîchissez. Vous devriez voir le message "Chargement..." rester affiché plusieurs secondes.

4. Pourquoi c'est important ? (Mémo)

Problème	**Solution**
L'écran reste blanc pendant le chargement	Variable isLoading + Rendu conditionnel
Le serveur renvoie une erreur 500 mais React essaie d'afficher les données quand même	Vérification if (!response.ok)
Le serveur est éteint et l'app plante	Bloc try / catch
Le loader tourne à l'infini en cas d'erreur	Bloc finally { setIsLoading(false) }