Résumé de l'audit V1, des problèmes détectés et des solutions adoptées en V2

Lien du dépôt git: https://github.com/MonifD/Exam_EBDD

1. Schéma de la Base de Données

Entités et Relations

La base de données est composée de six tables principales, liées entre elles par des relations clé étrangère. Voici un résumé des entités et de leurs relations :

1. **Categories**
   - Champs :
     - id (INT, PK, Auto-incrément) : Identifiant unique de la catégorie.
     - nom (VARCHAR(100)) : Nom de la catégorie.
   - Relations :
     - Une catégorie peut avoir plusieurs produits (hasMany Produit).
2. **Fournisseurs**
   - Champs :
     - id (INT, PK, Auto-incrément) : Identifiant unique du fournisseur.
     - nom (VARCHAR(100)) : Nom du fournisseur.
     - prenom (VARCHAR(100)) : Prénom du fournisseur.
     - adresse (VARCHAR(100)) : Adresse du fournisseur.
     - tel (VARCHAR(100)) : Numéro de téléphone du fournisseur.
   - Relations :
     - Un fournisseur peut fournir plusieurs produits (hasMany Produit).
3. **Client**
   - Champs :
     - id (INT, PK, Auto-incrément) : Identifiant unique du client.
     - nom (VARCHAR(100)) : Nom du client.
     - prenom (VARCHAR(100)) : Prénom du client.
     - adresse (VARCHAR(100)) : Adresse du client.
     - tel (VARCHAR(100)) : Numéro de téléphone du client.
   - Relations :
     - Un client peut passer plusieurs commandes (hasMany Commandes).

4. **Produit**
    - Champs :
        - id (INT, PK, Auto-incrément) : Identifiant unique du produit.
        - nom (VARCHAR(100)) : Nom du produit.
        - prix_unitaire (FLOAT) : Prix unitaire du produit.
        - quantite_stock (INT) : Quantité en stock du produit.
        - categorie_id (INT, FK) : Référence à la catégorie du produit.
        - fournisseur_id (INT, FK) : Référence au fournisseur du produit.
    - Relations :
        - Un produit appartient à une catégorie (belongsTo Categories).
        - Un produit appartient à un fournisseur (belongsTo Fournisseurs).
        - Un produit peut être associé à plusieurs lignes de commande (hasMany Lignes_Commande).
5. **Commandes**
    - Champs :
        - id (INT, PK, Auto-incrément) : Identifiant unique de la commande.
        - date_commande (DATETIME) : Date et heure de la commande.
        - statut (ENUM('En cours', 'Livrée')) : Statut de la commande.
        - client_id (INT, FK) : Référence au client qui a passé la commande.
    - Relations :
        - Une commande appartient à un client (belongsTo Client).
        - Une commande peut contenir plusieurs lignes de commande (hasMany Lignes_Commande).
6. **Lignes_Commande**
    - Champs :
        - id (INT, PK, Auto-incrément) : Identifiant unique de la ligne de commande.
        - quantite (INT) : Quantité du produit commandé.
        - prix_unitaire_applique (FLOAT) : Prix unitaire appliqué au moment de la commande.
        - commande_id (INT, FK) : Référence à la commande.
        - produit_id (INT, FK) : Référence au produit commandé.
    - Relations :
        - Une ligne de commande appartient à une commande (belongsTo Commandes).
        - Une ligne de commande appartient à un produit (belongsTo Produit)

2. Liste des Endpoints de l'API

Catégories

- GET /categories : Récupère toutes les catégories.
  Retour : Liste des catégories.
  Exemple de requête :

```json
[
  { "id": 1, "nom": "Avions de chasse" },
  { "id": 2, "nom": "Avions civils" }
]
```

- POST /categories : Crée une nouvelle catégorie.
  Paramètres : nom (string).
  Retour : La catégorie créée.
  Exemple de requête :

```json
{ "nom": "Avions militaires" }
```

- PUT /categories/:id : Met à jour une catégorie.
  Paramètres : nom (string).
  Retour : La catégorie mise à jour.
  Exemple de requête :

```json
{ "nom": "Avions historiques" }
```

- DELETE /categories/:id : Supprime une catégorie.
  Retour : Message de confirmation.
  Exemple de retour :

```json
{ "message": "Catégorie supprimée" }
```

Clients

- GET /clients : Récupère tous les clients.
- GET /clients/:id/commandes : Récupère les commandes d'un client.
- POST /clients : Crée un nouveau client.
- PUT /clients/:id : Met à jour un client.
- DELETE /clients/:id : Supprime un client.

Produits

- GET /produits : Récupère tous les produits.
- GET /produits/:id/commandes : Récupère les commandes contenant un produit.
- GET /produits/most-sold : Récupère les produits les plus vendus.
- GET /produits/stock-faible : Récupère les produits en stock faible.
- POST /produits : Crée un nouveau produit.
- PUT /produits/:id : Met à jour un produit.
- DELETE /produits/:id : Supprime un produit.

Fournisseurs

- GET /fournisseurs : Récupérer tous les fournisseurs.
- POST /fournisseurs : Créer un nouveau fournisseur.
- PUT /fournisseurs/:id : Mettre à jour un fournisseur existant.
- DELETE /fournisseurs/:id : Supprimer un fournisseur.

Commandes

- GET /commandes : Récupère toutes les commandes.
- GET /commandes/date : Récupère les commandes par période.
- GET /commandes/search : Recherche multi-critères des commandes.
- POST /commandes : Crée une nouvelle commande.
- PUT /commandes/:id : Met à jour une commande.
- DELETE /commandes/:id : Supprime une commande.

Lignes de Commande

- GET /lignes_commande : Récupère toutes les lignes de commande.
- POST /lignes_commande : Crée une nouvelle ligne de commande.
- PUT /lignes_commande/:id : Met à jour une ligne de commande.
- DELETE /lignes_commande/:id : Supprime une ligne de commande.

3. Résumé de l'Audit V1 et Solutions Adoptées en V2

Problèmes Détectés en V1

1. Requêtes SQL Directes : Les requêtes SQL étaient écrites directement dans le code, ce qui rendait le code vulnérable aux injections SQL et difficile à maintenir.
2. Manque de Validation : Les données entrantes n'étaient pas validées, ce qui pouvait entraîner des erreurs ou des incohérences dans la base de données.
3. Gestion des Relations : Les relations entre les tables étaient gérées manuellement, ce qui rendait les requêtes complexes et peu lisibles.
4. Structure Monolithique : Le code était peu modulaire, avec toutes les routes et les contrôleurs dans un seul fichier.

Solutions Adoptées en V2

1. Utilisation de Sequelize : Sequelize a été introduit pour gérer les requêtes SQL de manière sécurisée et lisible. Les requêtes sont maintenant paramétrées, ce qui réduit les risques d'injection SQL.
2. Validation des Données : Des validations ont été ajoutées dans les contrôleurs pour s'assurer que les données entrantes sont correctes avant de les insérer dans la base de données.
3. Gestion des Relations via Sequelize : Les relations entre les tables sont maintenant gérées via Sequelize, ce qui simplifie les requêtes et les jointures.
4. Modularité : Le code a été restructuré en modules distincts (modèles, contrôleurs, routes), ce qui améliore la lisibilité et la maintenabilité.

Conclusion

La V2 de l'application apporte des améliorations significatives en termes de sécurité, de modularité et de gestion des données. L'utilisation de Sequelize et la restructuration du code ont permis de résoudre les problèmes identifiés en V1, tout en ajoutant des fonctionnalités avancées comme la gestion des stocks et la

recherche multi-critères. Avec ces améliorations, l'application est maintenant plus robuste, sécurisée et facile à maintenir.