## Rapport de tests personnels(routes.js)

## Auth.route.js :

## ✅ Tests des routes d’authentification

Chaque test vérifie le bon fonctionnement d’une route spécifique de l’API d’authentification. Les routes sont accessibles via /auth/... et suivent les bonnes pratiques REST.

### 1. **POST /auth/register – Enregistrement d’un utilisateur**

* **Description :**  
  Ce test simule l’inscription d’un nouvel utilisateur avec un nom d'utilisateur et un mot de passe. La requête est envoyée avec un corps JSON contenant les informations d'identification.
* **Rôle du test :**  
  Vérifier que la route /auth/register :
  + Accepte une requête POST avec des données valides.
  + Répond avec un **code HTTP 201 (Created)**.
  + Retourne un objet JSON de confirmation (ex : { message: "Registered" }).

### 2. **POST /auth/login – Authentification d’un utilisateur**

* **Description :**  
  Ce test simule la connexion d’un utilisateur existant avec ses identifiants. Il envoie une requête POST contenant le nom d’utilisateur et le mot de passe.
* **Rôle du test :**  
  Vérifier que la route /auth/login :
  + Autorise l’accès avec des identifiants valides.
  + Répond avec un **code HTTP 200 (OK)**.
  + Fournit un **jeton d’accès (token)** dans la réponse, utilisé pour l’authentification future.

### 3. **POST /auth/refresh – Renouvellement du token**

* **Description :**  
  Ce test simule une demande de renouvellement du token JWT, typiquement utilisée lorsque le token initial arrive à expiration. Il envoie un token existant dans le corps de la requête.
* **Rôle du test :**  
  Vérifier que la route /auth/refresh :
  + Accepte une requête contenant un ancien token.
  + Répond avec un **code HTTP 200 (OK)**.
  + Renvoie un **nouveau token valide** dans le corps de la réponse.

### 4. **POST /auth/logout – Déconnexion d’un utilisateur**

* **Description :**  
  Ce test simule la déconnexion d’un utilisateur. Aucune information n’est envoyée dans le corps de la requête, car la simple action de logout suffit dans ce modèle.
* **Rôle du test :**  
  Vérifier que la route /auth/logout :
  + Réagit à une requête POST correcte.
  + Répond avec un **code HTTP 200 (OK)**.
  + Retourne un message de confirmation de déconnexion, par exemple : { message: "Logged out" }.

### 🧠 Remarques générales pour le rapport

* Ces cas de tests sont des **tests d’intégration** : ils vérifient l’interaction entre les routes et leurs contrôleurs.
* Ils permettent de s'assurer que l’API respecte les **conventions REST** (statuts HTTP, structure des réponses).
* Pour une couverture complète, on peut ajouter des **cas d’erreur** (ex : mot de passe manquant, identifiants incorrects, token expiré, etc.).

## User.route.js :

POST /users — Création d’un utilisateur

1. Test : retourne 400 si name ou password est manquant

Objectif : Vérifier que le backend refuse la création d’un utilisateur si l’un des champs obligatoires (name, password) est absent.

Entrée : Une requête POST sans name ou sans password.

Sortie attendue : Code HTTP 400 avec le message { error: "Name and password are required" }.

2. Test : retourne 409 si le nom d’utilisateur est déjà pris

Objectif : S’assurer qu’un nom d’utilisateur existant dans la base provoque une erreur de conflit.

Entrée : Requête POST avec name = "bob" déjà présent dans la base.

Sortie attendue : Code HTTP 409 avec le message { error: "Username already in use" }.

---

🔹 GET /users/:id — Récupération d’un utilisateur

3. Test : retourne 404 si l’utilisateur n’existe pas

Objectif : Tester la réponse quand l’utilisateur demandé n’est pas trouvé en base.

Entrée : ID inexistant, par exemple /users/999.

Sortie attendue : Code HTTP 404 avec { error: "User not found" }.

4. Test : retourne 403 si un utilisateur non-admin tente de voir un autre profil

Objectif : Empêcher un simple utilisateur de consulter les données d’un autre.

Entrée : Requête GET par un utilisateur user pour /users/:id d’un autre utilisateur.

Sortie attendue : Code HTTP 403 avec { error: "Unauthorized access" }.

---

🔹 PUT /users/:id — Mise à jour d’un utilisateur

5. Test : retourne 404 si l’utilisateur n’existe pas

Objectif : Vérifier que le serveur retourne une erreur si l’ID fourni est inconnu.

Entrée : Requête PUT vers /users/999.

Sortie attendue : Code HTTP 404 avec { error: "User not found" }.

6. Test : retourne 403 si un simple utilisateur tente de modifier un autre profil

Objectif : S’assurer que la mise à jour est limitée à l’admin ou à soi-même.

Entrée : Utilisateur user, requête PUT sur un autre utilisateur.

Sortie attendue : Code HTTP 403 avec { error: "Unauthorized access" }.

7. Test : retourne 403 si un admin tente de modifier son propre rôle

Objectif : Empêcher un administrateur de changer son rôle, pour éviter l’auto-destitution.

Entrée : Admin modifiant son propre role.

Sortie attendue : Code HTTP 403 avec { error: "Cannot change your own role" }.

8. Test : retourne 400 si aucun champ valide à modifier n’est fourni

Objectif : Éviter les requêtes de mise à jour vides.

Entrée : Requête PUT sans name, password ni role.

Sortie attendue : Code HTTP 400 avec { error: "No valid fields to update" }.

9. Test : retourne 409 si le nouveau nom est déjà pris

Objectif : Gérer le cas où la mise à jour proposerait un name déjà utilisé.

Entrée : Requête PUT avec name = "bob" (déjà pris).

Sortie attendue : Code HTTP 409 avec { error: "Username already in use" }.

---

🔹 DELETE /users/:id — Suppression d’un utilisateur

10. Test : retourne 404 si l’utilisateur n’existe pas

Objectif : Gérer la suppression d’un ID inexistant.

Entrée : Requête DELETE vers /users/999.

Sortie attendue : Code HTTP 404 avec { error: "User not found" }.

11. Test : retourne 403 si un utilisateur tente de se supprimer lui-même

Objectif : Empêcher tout utilisateur (admin ou non) de supprimer son propre compte.

Entrée : Requête DELETE vers /users/:id correspondant à l’utilisateur connecté.

Sortie attendue : Code HTTP 403 avec { error: "Cannot delete your own account" }.