Plan de Test - Application Web

1. Introduction

Ce document décrit le plan de test pour une application web composée d'une partie frontend (testée avec Playwright) et d'une partie backend (API REST testée avec JUnit). L'objectif est d'assurer la qualité fonctionnelle et technique de l'application à travers des cas de tests bien définis.

2. Objectifs des tests

- Vérifier le bon fonctionnement des fonctionnalités critiques du frontend (authentification, gestion des véhicules, gestion des utilisateurs).
- Garantir la robustesse des endpoints du backend.
- Détecter les erreurs d'entrée et les mauvaises pratiques utilisateur.

3. Approche de test

L'approche de test adoptée repose sur trois niveaux complémentaires :

- **Tests unitaires (Backend)** : réalisés avec **JUnit 5**. Ces tests portent sur des composants isolés (méthodes de contrôleurs et services). Les dépendances sont simulées à l'aide de **doublures de test** (`@MockBean` avec **Mockito**).
  - La suite de tests `UserControllerTest` **par daniloDevJava** est une suite de tests unitaires qui utilise une doublure du composant `UserService`.
  - La suite de tests `VehiculeControllerTest` **par chenjoProsper**est une suite de tests unitaires qui utilise une doublure du composant `VehiculeService`.
- **Tests d'intégration (Backend)** : réalisés avec **Spring Boot Test** et une base de données **H2** en mémoire. Ces tests permettent de vérifier la collaboration entre plusieurs couches de l'application (service, repository).
- **Tests end-to-end (Frontend)** : automatisés avec **Playwright**. Ces tests simulent des scénarios utilisateurs réels dans un navigateur (par exemple : connexion, affichage des véhicules, ajout d'un utilisateur).

Tous les tests sont automatisés et peuvent être intégrés dans le pipeline d'intégration continue (CI).

4. Environnement de test

- **Frontend** : Playwright, navigateur Chrome/Chromium.
- **Backend** : JUnit 5, Mockito, Spring Boot.
- Base de données H2 (en mémoire pour les tests).

5. Données de test

Les jeux de données incluent :

- Utilisateurs avec différents niveaux de validité (email correct/incorrect, mot de passe fort/faible).
- Véhicules avec des prix, immatriculations et modèles variés.

6. Cas de test Frontend (Playwright)

ID	Description du test
F1	Connexion : vérifier qu'un utilisateur peut se connecter avec des identifiants valides.
F2	Véhicules : vérifier que la liste des véhicules s'affiche après authentification.
F3	Utilisateurs : vérifier que l'ajout d'un utilisateur fonctionne et que celui-ci apparaît dans la liste.

7. Cas de test Backend - `UserControllerTests`

ID	Description du test
U1	Succès `/users/add` : statut 201, JSON contenant `id`, `email`, `name (en majuscules)`.
U2	Échec `/users/add` : email mal formé ⇒ 400, JSON erreur.
U3	Échec `/users/add` : mot de passe faible ⇒ 400, JSON erreur.
U4	Succès `/login` : statut 200, JSON avec `accessToken` et `refreshToken`.
U5	Échec `/login` : identifiants invalides ⇒ 404.
U6	Succès `/refresh-access-tokens` : token valide ⇒ 200, réponse $>$ 50 caractères.
U7	Échec token invalide ⇒ 403, message : *Le refresh token est invalide.*
U8	Échec token expiré ⇒ 403, message : *Le refresh token est expiré.*
U9	Succès `/refresh-tokens` ⇒ 200, JSON avec nouveaux tokens.
U10	Échec `/refresh-tokens` avec token expiré ⇒ 403, JSON erreur.
U11	Échec `/refresh-tokens` avec token invalide ⇒ 403, JSON erreur.
U12	Succès `/users/{id}` (PUT) : mise à jour, statut 200, JSON mis à jour.
U13	Échec mise à jour : email invalide ⇒ 400, JSON erreur.

U14	Échec mise à jour : utilisateur inexistant ⇒ 404.
U15	Succès `/users/all` : liste utilisateurs, statut 200, tableau JSON.
U16	Succès `/change-password` ⇒ 200, message : "mot de passe modifié".
U17	Échec `/change-password` : nouveau mot de passe faible ⇒ 400, JSON erreur.
U18	Échec `/change-password` : ancien mot de passe incorrect ⇒ 400, JSON erreur.

8. Cas de test Backend - `VehiculeControllerTests`

ID	**Description du test**
V1	Succès `/vehicules/create` : statut 201, JSON avec données du véhicule.
V2	Échec création : champs vides ou prix négatif ⇒ 400, tableau d'objets JSON d'erreurs.
V3	Échec création doublon registerNumber : premier succès, second rejeté avec erreur structurée.
V4	Succès `/vehicule/{id}` (GET) : statut 200, JSON véhicule.
V5	Succès suppression `/vehicule/{id}` : statut 200, message : "vehicule is deleted successfully".
V6	Échec suppression : id inconnu ⇒ 404.
V7	Échec récupération par ID inconnu ⇒ 404.
V8	Succès `/vehicule/search-by-price` : statut 200, tableau JSON filtré.
V9	Succès `/vehicule/number/{registerNum}` : statut 200, JSON véhicule.
V10	Échec `/vehicule/number/{registerNum}` : registre inconnu ⇒ 404.
V11	Succès update `/vehicule/{id}` : statut 200, JSON véhicule mis à jour.

1 9. Cas de test Backend - `UserServiceTest`

ID	**Description du test**
Us1	Succès `userService.createUser(user)` :

9. Risques et problèmes potentiels

- Mauvaise gestion des tokens expirés.
- Incohérences entre les formats JSON du frontend et du backend.
- Erreurs silencieuses non détectées sans journalisation explicite.

10. Reporting et communication

- Les résultats des tests seront analysés et communiqués après chaque exécution.

- Formats de rapport : HTML et JSON.
- Outils utilisés : CI/CD avec GitHub Actions.

11. Conclusion

Ce plan de test couvre les scénarios principaux de l'application. Il est destiné à être enrichi progressivement, au fur et à mesure de l'ajout de nouvelles fonctionnalités ou de la détection de bugs.