Introduction

Ce document constitue le plan de test pour l'application **vehicle_propelize**, une API REST développée avec Node.js et Express.js. L'objectif de ce plan est de définir la portée, les objectifs, les types de tests, les outils, ainsi que l'environnement nécessaires pour assurer la qualité et la fiabilité de l'application.

L'application **vehicle_propelize** est une solution backend conçue pour gérer un système de location de véhicules. Elle expose plusieurs endpoints permettant d'enregistrer, consulter, modifier, filtrer et supprimer des véhicules, tout en intégrant une gestion d'authentification sécurisée des utilisateurs. Elle repose sur une architecture **MVC modulaire** et utilise **Sequelize** pour l'interaction avec une base de données **MySQL**, le tout orchestré via **Docker**.

Les fonctionnalités principales de l'application incluent :

- Enregistrement et authentification sécurisée des utilisateurs (avec JWT et cookies).
- Gestion complète du catalogue de véhicules : création, modification, suppression, recherche.
- Filtrage par immatriculation et par tranche de prix.
- Application sécurisée par des tokens d'accès et de rafraîchissement.
- Mise à jour en temps réel du serveur via **Nodemon** et exécution de tests unitaires automatisés avec **Jest**.

Ce plan de test couvre les tests unitaires et d'intégration liés aux fonctionnalités critiques de l'API, en s'appuyant sur le framework **Jest** et le module **supertest** pour simuler les requêtes HTTP.

L'objectif global est de garantir que l'API fonctionne conformément aux spécifications techniques fournies, qu'elle réagit correctement face aux cas d'usage valides comme aux cas d'erreurs, et qu'elle respecte les principes de sécurité attendus.

Testing Objectives

1. Validation de la logique métier

- **Création d'utilisateurs et de véhicules** : Vérifier que les entités utilisateurs et véhicules peuvent être créées correctement avec des données valides.
- **Mise à jour et suppression** : Tester la capacité de modification et de suppression pour assurer une manipulation CRUD complète des ressources.
- **Gestion des duplications** : Garantir que l'application empêche les enregistrements en double (utilisateur déjà existant, immatriculation identique).

2. Sécurité et authentification

- **Authentification JWT** : Vérifier que l'authentification par jetons JWT fonctionne, que le token contient bien l'id et le name, et qu'il est correctement généré lors du login.
- **Refresh et Logout** : S'assurer que le mécanisme de renouvellement des tokens (refreshToken) fonctionne et que la déconnexion invalide bien la session.
- **Accès protégé** : Vérifier que les routes sensibles comme /vehicules (POST, PUT, DELETE) ne sont accessibles qu'avec un token JWT valide via le header Authorization.

3. Gestion des erreurs et robustesse

- **Codes HTTP appropriés** : Assurer le renvoi des bons codes HTTP selon les situations :
  - 201 pour les créations réussies,
  - 200 pour les requêtes GET/PUT valides,
  - 204 pour les suppressions sans retour de contenu,
  - 404 pour les entités inexistantes,
  - 401 pour les erreurs d'authentification,
  - 409 pour les conflits de duplication.
- **Messages d'erreurs explicites** : Valider que chaque réponse d'erreur contient un message compréhensible, facilitant le diagnostic côté client ou développeur.

4. Couverture des cas d'utilisation

Les tests couvrent l'ensemble des **routes REST exposées par l'API**, notamment :

- /users/register, /users/login, /users/logout, /users/refresh
- /vehicules, /vehicules/:registration, /vehicules/prix

Chaque route est testée à la fois pour des **scénarios de succès** et des **cas limites ou erreurs** (invalidité, non-existence, absence de jeton, duplication, etc.).

5. Alignement avec les spécifications fonctionnelles

Ces tests permettent de garantir que :

- Les contraintes de validation (ex : champs requis, format, unicité) sont bien appliquées.
- Les données manipulées via l'API sont persistées de façon cohérente dans la base de données MySQL via Sequelize.
- Le comportement de l'API est conforme à la documentation métier, notamment en termes de gestion de flotte (CRUD véhicules) et de sécurité utilisateur (login, sessions).

6. Mesurabilité et atteignabilité

Les objectifs sont **mesurables** à travers :

- Le taux de réussite des tests Jest.
- La couverture des branches de logique dans les contrôleurs.
- L'assertion des résultats retournés (statuts, payloads, erreurs).

Ils sont **atteignables** dans le cadre du projet Propelize puisque :

- Le système est conteneurisé (Docker) donc facilement déployable.
- Les dépendances sont bien définies dans package.json.

Approche de Test

1. Méthodologie Générale

L'approche combine **tests unitaires, tests d'intégration, tests de sécurité**, et **tests de bout en bout (end-to-end)** :

Type de test	Outils utilisés	Portée
Test unitaire	Jest	Contrôleurs, logique métier isolée
Test d'intégration	Jest + Supertest	Routes, logique HTTP, réponse des endpoints
Test de sécurité	Jest + Supertest	Authentification, validation des tokens
Test end-to-end	Jest + Supertest	Scénarios utilisateur complets, cycle de vie

2. Détail des Types de Tests et Scénarios Couverts

♣ **Tests unitaires (logique métier isolée)**

Bien que centrés sur les tests d'intégration, quelques tests valident indirectement la logique des contrôleurs :

- **Validation des entrées :** empêche l'enregistrement d'utilisateurs ou de véhicules en doublon (tests 409).
- **Traitement d'erreurs :** réponse 404 ou 401 renvoyée si une ressource est absente ou des identifiants sont invalides.
- **Encapsulation des données sensibles :** vérifie que le mot de passe n'est jamais retourné dans les réponses (expect(res.body.password).toBeUndefined()).

♣ **Tests d'intégration (interactions entre routes, contrôleurs, et BD)**

Ces tests couvrent les routes principales, avec création/suppression dynamique des données :

user.test.js

- POST /users/register : création d'un utilisateur, puis rejet si doublon.
- GET /users : liste tous les utilisateurs sans mot de passe.
- PUT /users/:id : met à jour un utilisateur existant, échec si ID inconnu.
- POST /users/login : connexion avec nom + mot de passe valides ou invalides.
- POST /users/refresh : génère un nouveau JWT via cookie.
- POST /users/logout : invalide le cookie (refreshToken vidé).

vehicle.test.js

- GET /vehicules : retour d'une liste vide ou remplie.
- POST /vehicules : création de véhicule, gestion de doublons.
- GET /vehicules/:registration : lecture d'un véhicule par immatriculation.
- PUT /vehicules/:registration : mise à jour d'un véhicule existant ou 404 si inexistant.
- DELETE /vehicules/:registration : suppression avec 204 ou 404 si non trouvé.
- GET /vehicules/prix?min=&max= : filtrage dynamique par prix.

Tous ces tests vérifient :

- **Les codes de réponse HTTP** (200, 201, 204, 404, 409, 401).
- **Le format de la réponse JSON**.
- **L'impact réel sur la base de données** (insertions, modifications, suppressions).

♣ **Tests de sécurité**

Ces tests assurent la robustesse de l'authentification et la confidentialité des données :

- **JWT sécurisé :**
    - Vérification que le accessToken contient bien les bonnes infos utilisateur (id, name), est bien signé et a une durée de validité.
    - Décodage avec jwt.verify() et contrôle de l'expiration (exp > iat).
- **Cookies HTTPOnly :**
    - Contrôle de la présence d'un cookie refreshToken après connexion.
    - Révocation du cookie via logout (refreshToken=;).
- **Protection des routes :**
    - Les routes sensibles (POST, PUT, DELETE sur /vehicules) nécessitent un token valide dans l'en-tête Authorization (Bearer <token>).

♣ **Tests end-to-end (scénarios complets)**

Ces tests reproduisent des parcours réels :

- **Cycle de vie complet d'un utilisateur :**
    1. Enregistrement.
    2. Connexion.
    3. Récupération de token.
    4. Actualisation de token.
    5. Déconnexion.
- **Gestion complète des véhicules :**
    1. Création avec authentification.
    2. Consultation par immatriculation ou tranche de prix.
    3. Modification du tarif.
    4. Suppression.

Chaque cycle est réinitialisé à l'aide de sequelize.sync({ force: true }) en début de test, garantissant un environnement **propre et isolé**.

3. Outils et Environnement de Test

- **Jest** pour l'exécution et les assertions (expect()).
- **Supertest** pour l'envoi de requêtes HTTP simulées.
- **Base de données MySQL sous Docker** : conforme à la production.

- **Nodemon** utilisé pour relancer le serveur en développement, **non utilisé pendant les tests** (mode CLI uniquement).
- Fichier .env chargé pour fournir les variables de configuration du test.

Commandes utiles :

 # Lancer les tests

npm run test

 # Lancer la base de données Docker (si pas déjà fait)

sudo docker-compose up -d

4. Rôles et Responsabilités

Rôle	Tâches spécifiques
Développeur	Écriture des tests, exécution locale, correction des erreurs.
QA Engineer	Revue des cas de test, validation des cas limites, couverture maximale des erreurs.
Chef de projet	S'assurer que tous les endpoints critiques sont couverts par des tests.
DevOps (optionnel)	Intégration des tests dans le pipeline CI/CD (ex : GitHub Actions).

5. Objectifs de l'approche

- **Fiabilité** : garantir que chaque endpoint retourne des résultats corrects.
- **Robustesse** : valider la résistance aux entrées malformées ou non autorisées.
- **Sécurité** : tester tous les aspects liés à l'authentification et aux tokens JWT.
- **Maintenance** : faciliter la détection rapide de régressions fonctionnelles.

Testing Schedule

Cette section décrit le calendrier détaillé des tests automatisés mis en œuvre avec **Jest** pour les modules user et vehicle de l'API **Propelize**. Les phases de test sont alignées avec les étapes standards d'un cycle de qualité logicielle : **planification, conception, exécution et reporting**.

1. Phase de planification (Jour 1)

- **Objectif** : Identifier les entités critiques à tester (users, vehicules), définir les cas de test prioritaires (création, lecture, mise à jour, suppression, authentification).

- **Durée estimée** : 1 jour
- **Livrables** :
    - Arborescence de test (tests/user.test.js, tests/vehicle.test.js)
    - Définition des critères d’acceptation

2. Phase de conception (Jours 2–3)

- **Objectif** : Écrire les cas de test dans Jest selon les spécifications de l’API REST.
- **Contenu couvert** :
    - user.test.js :
        - Inscription / login / logout / token
        - Mise à jour utilisateur
        - Erreurs : doublons, credentials invalides
    - vehicle.test.js :
        - CRUD complet véhicule
        - Filtrage par prix
        - Authentification obligatoire pour les routes sensibles
- **Durée estimée** : 2 jours
- **Livrables** :
    - Scripts de test Jest complets
    - Mocks éventuels
    - Configuration de base (Docker, Sequelize reset, JWT, etc.)

3. Phase d'exécution (Jour 4)

- **Objectif** : Lancer tous les tests dans un environnement de développement (Docker MySQL actif) pour vérifier la stabilité et la cohérence des fonctionnalités.
- **Commandes exécutées** :

sudo docker-compose up -d

npm run test

- **Durée estimée** : 1 jour
- **Résultats attendus** :
    - 100 % des tests passent

- Gestion correcte des erreurs (401, 404, 409, etc.)
- Respect des statuts HTTP

4. Phase de reporting (Jour 5)

- **Objectif** : Documenter les résultats, corriger les anomalies, valider la couverture.
- **Livrables** :
  - Rapport de couverture Jest (--coverage)
  - Listing des tests critiques validés
  - Mise à jour du README pour les instructions de test

Calendrier récapitulatif

Phase	Durée prévue	Résultat principal
Planification	Jour 1	Définition des cas de test
Conception	Jours 2–3	Scripts de test opérationnels
Exécution	Jour 4	Tests automatisés passés avec succès
Reporting	Jour 5	Couverture validée + documentations mises à jour

Test Environment

L'environnement de test mis en place pour l'API Véhicules **Propelize** repose sur une stack cohérente et réaliste, reproduisant les conditions de production afin d'assurer la fiabilité des tests unitaires et d'intégration. Ci-dessous les composants nécessaires pour exécuter correctement les tests définis dans user.test.js et vehicle.test.js.

1. Configuration matérielle minimale

- Processeur : Dual-core 2.0 GHz
- RAM : 4 Go minimum
- Stockage : 500 Mo d'espace libre pour les logs et les containers
- Système d'exploitation : Linux, macOS ou Windows avec WSL2

2. Environnement logiciel requis

Composant	Version minimale recommandée	Rôle dans les tests
Node.js	18.x	Environnement JavaScript pour

		exécuter le code
npm	9.x	Gestionnaire de dépendances
Docker	20.x	Conteneurisation de la base de données MySQL
Docker Compose	1.29.x	Orchestration de services (MySQL container)
MySQL (via Docker)	8.0.41	SGBD utilisé dans les tests
Jest	Dernière version stable	Framework de test utilisé pour les scripts
Postman *(optionnel)*	N/A	Pour tests manuels et vérification d'API

3. Dépendances du projet (via npm)

Assurez-vous d'installer les dépendances suivantes via npm install (voir package.json) :

- **Serveur et ORM :** express, sequelize, mysql2
- **Sécurité et Auth :** jsonwebtoken, bcrypt, cookie-parser
- **Tests :** jest, supertest
- **Utilitaires :** dotenv, cors, nodemon

4. Configuration réseau

L'environnement utilise un conteneur **MySQL** exposé sur le port 3307 (externe) redirigé vers 3306 (interne) via Docker. Il est impératif de s'assurer que :

- Aucun autre service n'occupe le port 3307
- Le fichier .env contient les bonnes variables de connexion :

PORT=3000

DB_HOST=localhost

DB_USER=root

DB_NAME=propelize

DB_PORT=3307

JWT_SECRET=un_secret_exemple

5. Initialisation et exécution des tests

Étapes à suivre avant d'exécuter les tests :

- ♣ Démarrer la base de données :

sudo docker-compose up -d

- ♣ Vérifier que la base tourne bien :

sudo docker ps

- ♣ Lancer les tests :

npm run test

Les tests utilisent une base recréée à chaque session via :

await sequelize.sync({ force: true });

Cela garantit un état propre et isolé à chaque test (important pour les assertions comme les conflits ou les authentifications).

Données de Test (Test Data)

Cette section décrit les jeux de données nécessaires pour exécuter les différents cas de test des routes utilisateurs (/users) et véhicules (/vehicules). Ces données sont cruciales pour simuler les scénarios attendus (création, duplication, mise à jour, suppression, etc.) dans l'environnement de test.

Source des Données de Test

Les données sont directement définies dans les fichiers de test user.test.js et vehicle.test.js, en utilisant la librairie supertest pour simuler les appels HTTP. La base de données est synchronisée à chaque exécution (sequelize.sync({ force: true })), assurant un environnement propre.

Données pour les tests utilisateurs (user.test.js)

Les jeux de données suivants sont utilisés pour tester l'ensemble des fonctionnalités liées aux utilisateurs :

- **Utilisateur initial :**

```
{
  id: 1,
  name: "Alias",
  password: "root"
}
```

- Données pour mise à jour :

```
{
```

name: "AliasUpdated",

password: "newpassword"

}

- **Utilisateurs invalides ou inexistants :**
    - Nom incorrect : "Inexistant"
    - Mot de passe incorrect : "mauvaismotdepasse"
- **Connexion / Authentification :**
  Les tests vérifient également la génération de tokens (accessToken, refreshToken) et leur contenu via décodage (jwt.verify).
- **Test de renouvellement et suppression de session :**
  Des cookies (refreshToken) sont utilisés pour simuler une session utilisateur valide.

Données pour les tests véhicules (vehicle.test.js)

Les données sont conçues pour tester toutes les routes REST de gestion de véhicules :

- **Véhicule de base :**

{

registration: 'CAR123',

make: 'Peugeot',

model: '208',

year: 2020,

rentalPrice: 50.0

}

- **Véhicule supplémentaire pour tests de prix :**

{

registration: 'CAR999',

make: 'Renault',

model: 'Clio',

year: 2021,

rentalPrice: 40.0

}

Véhicules inexistants (test d’erreur 404) :

- registration: 'INEXISTANT'

- **Mise à jour de prix :**
    - rentalPrice: 60.0 ou 100.0 selon le test

Contraintes et Limitations

- Les identifiants (id, registration) doivent être uniques pour tester les conflits (code 409).
- Le champ year est validé dans le modèle avec une plage entre 1600 et 2025.
- Le champ rentalPrice est borné entre 0 et 9 999 999 (conforme au modèle).
- Les mots de passe ne doivent jamais être exposés dans les réponses (password masqué).
- Les appels protégés (ex: POST, PUT, DELETE sur /vehicules) nécessitent un accessToken JWT valide via l'en-tête Authorization.

Initialisation et Nettoyage

- beforeAll: synchronisation de la base avec suppression ({ force: true }) pour garantir un état initial propre.
- afterAll: fermeture propre de la connexion Sequelize après tous les tests.

Test Cases

Unit Test Plan – Module Users

Section	Détails
Module #:	U01
Application #:	Propelize-API
Tester:	Alias
Test Manager:	Alias
Module Overview:	Le module gère les utilisateurs : inscription, authentification (JWT), actualisation de token, déconnexion, CRUD.
Module Inputs:	name, password, id dans le corps de la requête HTTP
Module Outputs:	JSON avec statut, données utilisateur (sans mot de passe), accessToken, cookies (refreshToken)
Logic Flow:	
Test Data:	Voir sections suivantes.

Positive Test Cases

#	Test	Résultat attendu
TC-U01	Création utilisateur	201 Created, JSON avec id, name, pas de mot de passe visible
TC-U02	Connexion valide	200 OK, token JWT retourné, cookie avec refreshToken
TC-U03	Actualisation token	200 OK, nouveau token différent
TC-U04	Récupération utilisateur	200 OK, tableau d’utilisateurs, sans mot de passe
TC-U05	Mise à jour utilisateur	200 OK, données modifiées visibles
TC-U06	Déconnexion utilisateur	204 No Content, cookie refreshToken vidé
TC-U07	Token JWT valide	Contient id, name, expiration future

Negative Test Cases

Test	Données invalides	Résultat attendu
Inscription doublon	Nom déjà utilisé	409 Conflict
Connexion nom inexistant	name: 'fakeUser'	401 Unauthorized
Connexion mot de passe incorrect	password: 'wrong'	401 Unauthorized
Mise à jour utilisateur inexistant	id: 999	404 Not Found

Interface Modules

Module lié	Type d’interface
vehicleController.js	Internal Program Interface
authMiddleware.js	Internal Program Interface
jwt, cookie-parser, .env	External Program Interface (librairies, config)

Test Tools

- **Jest** : exécution des tests unitaires (npm run test)

- **Supertest** : simulation des requêtes HTTP
- **Location fichiers tests :** /Test/user.test.js

Archive Plan

- **Archivage test logs :** Terminal / CI (GitHub Actions ou GitLab)
- **Données sensibles masquées** (ex: password)
- **Procédure :** accessible via commit Git, ou CI/CD artifacts

Updates

- **Ajout d'un champ à l'utilisateur** ⇒ mise à jour des tests d'inscription / affichage
- **Nouveaux endpoints** ⇒ ajout d'une nouvelle section de tests (ex: /users/profile)

Unit Test Plan – Module Vehicles

Section	**Détails**
Module #:	V01
Application #:	Propelize-API
Tester:	Alias
Test Manager:	Alias
Module Overview:	Ce module gère les opérations CRUD pour les véhicules, ainsi que la recherche par prix ou immatriculation.
Module Inputs:	registration, make, model, year, rentalPrice dans le corps de la requête ou query params
Module Outputs:	JSON avec les informations du véhicule ou erreur
Logic Flow:	
Test Data:	Voir ci-dessous

Positive Test Cases

#	**Test**	**Résultat attendu**
TC-V01	Création véhicule	201 Created, JSON avec données envoyées
TC-V02	Lecture tous véhicules	200 OK, tableau (vide ou rempli)

TC-V03	Lecture par immatriculation	200 OK, véhicule spécifique
TC-V04	Filtrage prix min/max	200 OK, véhicules dans la plage
TC-V05	Mise à jour véhicule existant	200 OK, données mises à jour
TC-V06	Suppression véhicule existant	204 No Content

Negative Test Cases

Test	**Données invalides**	**Résultat attendu**
Création doublon	registration déjà existant	409 Conflict
Requête sur véhicule inexistant	/vehicules/FAUX123	404 Not Found
Suppression véhicule inexistant	DELETE /vehicules/XYZ	404 Not Found
PUT sur véhicule inexistant	PUT /vehicules/XYZ	404 Not Found

Interface Modules

Module lié	**Type d'interface**
userController.js	Internal Program Interface (auth)
Middleware JWT	External Interface (authentification requise)
Sequelize	Internal ORM interface
.env, MySQL Docker	External Program Interface (config/db)

Test Tools

- **Jest**
- **Supertest**
- **Localisation :** /Test/vehicle.test.js

Archive Plan

- Données test non sensibles
- **Logs dans CI/CD** ou sortie console
- **Backup éventuel :** /logs/test-results.log si configuré

Updates

- Nouveau champ véhicule ⇒ mise à jour des tests POST, PUT, GET
- Ajout endpoint (ex: /vehicules/marque/:make) ⇒ nouvelle section de tests

Test Automation Plan

Élément	Détails
Test Automation Tools	- **Jest** : framework principal pour l'exécution des tests- **Supertest** : pour simuler les requêtes HTTP Express
Test Scripts	- user.test.js : couvre toutes les routes utilisateurs- vehicle.test.js : couvre les routes véhicules avec auth
Automated Test Types	- Tests unitaires sur les contrôleurs / services- Tests d'intégration sur les routes API REST
Test Data	- Données statiques définies dans chaque script : utilisateur Alias, véhicule CAR123
Test Scenarios (user)	- Création utilisateur- Doublon utilisateur- Authentification JWT- Token Refresh / Logout- Accès conditionnel
Test Scenarios (vehicules)	- Insertion véhicule- Doublon- Récupération (GET, GET par prix, immat.)- Update / DELETE- Authentification par Bearer token
Automation Trigger	- npm run test (exécute Jest automatiquement)
CI/CD Compatible	Oui – peut être intégré à un pipeline GitHub Actions / GitLab CI pour déclenchement à chaque push / pull request
Regression Testing	Tous les tests sont versionnés et exécutables à tout moment sur base de données Docker stable
Test Reports	- Utilise la sortie CLI de Jest- Peut être configuré avec jest-html-reporter ou jest-junit pour rapports formels
Script Update Policy	Les tests sont mis à jour manuellement à chaque modification de code (ex: ajout de nouvelle route, champ, logique métier)
Archive & Access	- Les fichiers de test sont situés dans /test/- L'historique Git permet d'accéder à toutes les versions antérieures
Future Automation	- Prévu : couverture des erreurs 500 / timeout- Prévu : mocks plus fins avec Jest pour tests unitaires indépendants

Risks and Issues

Identifiant	Description du Risque	Impact sur les Tests	Stratégie de Mitigation
R1	**Dépendance à Docker pour MySQL** : Si le conteneur Docker ne démarre pas correctement.	Les tests nécessitant une base de données (ex : user.test.js, vehicle.test.js) échouent dès l'amorçage (beforeAll).	Fournir des instructions de vérification (docker ps, docker logs), et documenter les étapes de redémarrage.
R2	**Mauvaise configuration du fichier .env**	Échec de connexion à la base de données, entraînant l'échec de tous les tests.	Ajouter une validation de .env au démarrage et inclure un exemple .env.example dans le repo.
R3	**Effets de bord entre tests (données persistantes)**	Un test pourrait échouer car un autre test a modifié ou supprimé une ressource partagée.	Utiliser sequelize.sync({ force: true }) en beforeAll, et structurer les tests pour être indépendants.
R4	**Tests sensibles à l'ordre d'exécution**	L'échec ou la suppression d'un test préalable invalide un test suivant (ex : création de l'utilisateur dans beforeAll).	Initialiser l'état requis au sein de chaque describe ou beforeEach pertinent.
R5	**JWT invalide ou expiré pendant les tests**	Certains endpoints (POST /vehicules, PUT, DELETE) nécessitent une authentification avec JWT.	Générer un token valide dans un beforeEach, ou mocker les middlewares d'auth dans des tests unitaires séparés.
R6	**Fausse alerte sur les tests à cause du cache Jest / erreurs de timeout**	Certains tests peuvent échouer sporadiquement si le cache Jest interfère ou si Sequelize met trop de temps à répondre.	Utiliser --clearCache pour Jest régulièrement, et augmenter les timeouts si besoin (jest.setTimeout(30000)) .
R7	**Doublons non gérés**	Le test POST /users ou	Nettoyer ou isoler la

	dans les tests (ex: POST déjà exécuté)	POST /vehicules peut échouer si les données ont déjà été insérées.	base à chaque test (via sequelize.sync({ force: true })) ou utiliser des identifiants dynamiques.
R8	**Non gestion des erreurs réseau (API ou DB indisponible)**	Les tests peuvent échouer brutalement sans message clair si le serveur ou la DB est injoignable.	Implémenter des try/catch et logger clairement les erreurs. Ajouter une vérification de disponibilité en pré-test.
R9	**Erreurs de routes dynamiques mal encodées** (/users/${id}, /vehicules/${registration} avec syntaxe erronée)	Les routes mal construites (.put(/vehicules/${...})) peuvent ne pas être testées du tout.	Corriger la syntaxe en utilisant des backticks (`) au lieu de /.../ pour les templates littéraux dans les routes.
R10	**Tests trop couplés à des données statiques (nom d'utilisateur, plaques)**	Si ces données changent, les tests échouent ou perdent en pertinence.	Externaliser les jeux de données dans des fixtures ou générer dynamiquement les entrées.

Remarques supplémentaires

- **Code fragile** : certaines routes utilisent des chaînes non interpolées (ex: .put(/users/${userData.id})) — cela doit être corrigé pour assurer l'exécution du test.
- **Docker** : Documenter les étapes pour forcer la recréation de conteneur si docker-compose up -d échoue.
- **Logs et traçabilité** : Ajouter des logs personnalisés dans les tests permet d'identifier rapidement le bloc défaillant en cas de panne.

Reporting and Communication

1. Fréquence des rapports

Les rapports de test seront générés :

- **À chaque exécution des tests Jest**, localement ou via un pipeline CI/CD (comme GitHub Actions ou GitLab CI).

- **À chaque mise à jour importante du code source** (merge sur la branche principale, pull request, etc.).
- **Périodiquement**, avant chaque version majeure ou livraison, pour valider la non-régression.

2. Format des rapports

Par défaut, Jest génère un **rapport en ligne de commande** clair et lisible, avec :

- Le **nombre total de tests exécutés**, réussis et échoués.
- Les **messages d'erreur** ou d'échec détaillés.
- Les **fichiers de test concernés** (user.test.js, vehicle.test.js, etc.).

Extrait typique d'un rapport :

PASS test/user.test.js

✓ crée un utilisateur (201)

✓ échoue en cas de doublon (409)

...

Test Suites: 1 passed, 1 total

Tests: 12 passed, 12 total

En cas de besoin, un format **plus structuré** (ex. : JSON, JUnit XML) peut être activé via l'ajout de --json ou d'un reporter personnalisé dans jest.config.js, pour une intégration dans un tableau de bord ou outil CI/CD.

3. Destinataires des rapports

Les résultats des tests sont destinés à différents intervenants :

Partie prenante	**Mode de réception du rapport**	**Objectif**
Développeurs	Terminal local / GitHub / GitLab CI	Corriger les erreurs rapidement
Chef de projet	Résumé (copié-collé ou PDF)	Suivi de la qualité logicielle
Testeurs QA	Rapport complet ou filtré	Analyser les causes d'échecs
Clients / Donneurs d'ordre	Résumé (facultatif, sur demande)	Transparence et preuve de tests

4. Outils complémentaires de communication

- **Postman** peut être utilisé pour reproduire manuellement les erreurs détectées.

- En cas de CI/CD, les tests peuvent être automatiquement lancés à chaque *push*, avec notification par email ou via Slack.
- En local, un résumé peut être copié dans un document partagé (Notion, Confluence, etc.).

5. Suivi des anomalies

Les erreurs détectées durant les tests automatisés sont :

- **Documentées dans un fichier log ou un outil de ticketing** (GitHub Issues, Jira).
- **Assignées à un développeur**, avec reproduction possible grâce aux fichiers *.test.js.

Conclusion

- **Robustesse des routes utilisateurs :**
  Les tests vérifient la création, la mise à jour, la connexion (avec vérification JWT), le rafraîchissement et la déconnexion. Toutes les erreurs courantes (utilisateur existant, non trouvé, mot de passe incorrect) sont aussi testées.
- **Fiabilité de la gestion des véhicules :**
  Chaque opération CRUD est testée en conditions normales et erronées (conflits, objets inexistants). La recherche par tranche de prix est également validée.
- **Authentification et sécurité :**
  L'authentification par token JWT est correctement testée. Le test valide également le contenu du token et le bon fonctionnement du rafraîchissement et de la suppression du cookie refreshToken.
- **Base de données Dockerisée :**
  L'environnement MySQL est simulé en conteneur Docker, assurant l'indépendance de l'environnement local et la cohérence des résultats.

Recommandations :

- Ajouter des tests pour les cas limites (par ex. formats invalides, injection SQL, données trop longues).
- Compléter la couverture pour les erreurs de validation côté modèle (ex. années invalides dans les véhicules).
- Intégrer les tests à un pipeline CI/CD pour garantir la stabilité continue du code.
- Envisager d'utiliser des **fixtures ou factories** pour simplifier la création de jeux de données.

Les tests réalisés démontrent que l’API est globalement stable et bien structurée selon les principes REST. La séparation claire des responsabilités (routes, contrôleurs, modèles) facilite la maintenabilité. Le projet est prêt pour évoluer vers une interface React, avec un backend robuste et testé.