Documentation Technique Complète du Projet : Gestion de Vélos et Trajets

Table des Matières

---

1. Introduction et Objectifs

Objectifs Fonctionnels

- **Planification des trajets** : Permettre aux cyclistes de planifier leurs trajets et signaler les incidents.
- **Gestion des incidents** : Prioriser et résoudre les incidents signalés.
- **Optimisation des trajets** : Minimiser les distances parcourues et maximiser l'autonomie des vélos.

Objectifs Techniques

- **Modularité** : Une architecture facilement extensible.
- **Sécurité** : Assurer la protection des données utilisateurs.
- **Scalabilité** : Concevoir un système capable de croître en utilisateurs et fonctionnalités.

---

2. Architecture Technique

Modèle-Vue-Contrôleur (MVC)

- **Modèles** : Gestion des données.
- **Contrôleurs** : Prise en charge des requêtes HTTP.
- **Services** : Logique métier (calculs d'itinéraires, gestion des incidents).
- **Utilitaires** : Algorithmes spécifiques (géolocalisation, notifications).

Structure de la Base de Données

- **Utilisateur** : Contient les informations d'identification.
- **Cycliste** : Informations spécifiques (statut, rôle).
- **Vélo** : Autonomie, statut.
- **Trajet** : Points de départ et d'arrivée.
- **Incident** : Signalement des problèmes.

---

3. Installation et Configuration

Prérequis Techniques

- **Node.js**
- **Visual Studio Code (VSCode)**

Étapes d'Installation

1. **Cloner le Projet** : `https://github.com/zkroz1ent/PROJETPOUBELLEVERTE.git`
2. **Frontend** :

Allez dans le répertoire :
`cd frontend`
Installez les dépendances :
`npm install`

3. **Backend** :

Ouvrez une autre fenêtre et placez-vous dans le répertoire backend :
`cd backend`
Installez les dépendances :
`npm install`

Configuration

1. Créez une base de données nommée `poubelle_verte` : `CREATE DATABASE poubelle_verte;`
2. Configurez le fichier de connexion à la base de données.

Lancer les Serveurs

Frontend :
`npm run serve`

Backend :
`npm start`

Comptes Utilisateurs par Défaut :

Rôle	Email
Administrateur	admin1@test.fr
Cycliste	cycliste1@test.fr
Gestionnaire	gestionnaire1@test.fr

---

4. Fonctionnalités Avancées

4.1 Gestion des Itinéraires

La gestion des itinéraires permet aux cyclistes de planifier des trajets optimisés en fonction de plusieurs critères :

1. **Autonomie des vélos** : La distance du trajet est limitée en fonction de l'autonomie disponible.
2. **Points de collecte ou de dépôt** : Les itinéraires peuvent inclure des arrêts intermédiaires pour déposer ou collecter des objets.
3. **Minimisation de la distance** : Utilisation de l'algorithme de Dijkstra pour trouver le chemin le plus court.

Processus :

1. **Entrées** :
    - Point de départ et d'arrivée (coordonnées géographiques ou identifiant des arrêts).
    - Autonomie restante du vélo.
2. **Traitement** :
    - Construction d'un graphe des arrêts disponibles.
    - Calcul du chemin optimal à l'aide de Dijkstra.
    - Validation de la faisabilité en fonction de l'autonomie.
3. **Sortie** :
    - Liste ordonnée des arrêts à traverser.
    - Distance totale et estimation du temps de trajet.

4.2 Gestion des Incidents

La gestion des incidents est une fonctionnalité clé pour assurer la maintenance du parc de vélos et la sécurité des utilisateurs.

Fonctionnalités associées :

1. **Signalement** :
    - Les cyclistes signalent les incidents directement via l'application (panne, accident, etc.).
2. **Classification** :
    - Les incidents sont classés par type et priorisés :
        - **Pannes critiques** : Nécessitent une intervention immédiate.
        - **Pannes mineures** : Planifiées pour une intervention ultérieure.
3. **Résolution** :
    - Les gestionnaires et administrateurs peuvent clôturer les incidents après intervention.
4. **Notifications** :
    - Les incidents critiques déclenchent des notifications en temps réel pour les gestionnaires.

4.3 Gestion des Vélos

La plateforme assure une gestion complète des vélos :

- **Suivi de l'autonomie** : Chaque vélo est équipé d'un capteur rapportant son niveau de batterie.
- **Historique des trajets** : Chaque vélo conserve un historique détaillé des trajets effectués.
- **Maintenance préventive** : Alertes déclenchées en fonction du nombre de kilomètres parcourus.

4.4 Rôles et Permissions

Le système distingue les rôles suivants :

1. **Utilisateur** : Accès à la planification des trajets et à son historique.
2. **Cycliste** : Signalement d'incidents et gestion des trajets.

3. **Gestionnaire** : Résolution des incidents, gestion des utilisateurs et vélos.
4. **Administrateur** : Accès complet à toutes les fonctionnalités.

---

5. API RESTful : Détails Techniques

Exemple : Déclaration d'un Incident

- **Méthode** : POST
- **Endpoint** : `/api/incidents`

Requête :

```
{
  "type": "panne",
  "description": "Problème de chaîne",
  "veloId": 42
}
```

-
- **Réponse** :
  - 201 : Incident créé
  - 400 : Données invalides
  - 500 : Erreur serveur

---

6. Algorithmes Complexes et Explications

6.1 Algorithme de Dijkstra pour les Itinéraires

L'algorithme de Dijkstra est utilisé pour calculer le chemin le plus court entre deux points.

Fonctionnement :

1. **Initialisation** :

- Chaque arrêt est assigné à une distance infinie, sauf le point de départ (distance = 0).

2. **Itération** :
   - À chaque étape, l'arrêt non visité avec la plus courte distance est sélectionné.
   - Les distances des arrêts adjacents sont mises à jour si un chemin plus court est trouvé.
3. **Terminaison** :
   - Lorsque tous les arrêts sont visités ou que le point d'arrivée est atteint, le chemin est reconstruit.

Implémentation (pseudo-code) :

```
function dijkstra(graph, startNode, endNode) {

    const distances = {};

    const previous = {};

    const visited = new Set();


    distances[startNode] = 0;


    while (visited.size < Object.keys(graph).length) {

        let currentNode = findClosestNode(distances,
visited);

        visited.add(currentNode);


        for (let neighbor of graph[currentNode]) {

            let newDistance = distances[currentNode] +
neighbor.weight;
```

```
            if (newDistance < (distances[neighbor.node]
|| Infinity)) {
                distances[neighbor.node] = newDistance;
                previous[neighbor.node] = currentNode;
            }
        }
    }

    return reconstructPath(previous, startNode, endNode);
}
```

6.2 Calcul de Capacité de Vélo

Un algorithme simple calcule si un trajet est faisable avec l'autonomie restante.

Étapes :

1. Distance totale du trajet calculée à partir des arrêts.
2. Comparaison avec l'autonomie restante.
3. Retourne un statut (`"OK"` ou `"Non-faisable"`).

---

7. Gestion de la Sécurité

7.1 Authentification JWT

L'authentification repose sur des JSON Web Tokens (JWT) pour maintenir un système stateless.

Étapes du Processus :

1. **Génération** :
    - Lorsqu'un utilisateur se connecte, un JWT est généré avec ses informations (`id`, `role`).
- Exemple de payload :
  `{`
- `"id": 123,`
- `"role": "Cycliste",`
- `"iat": 1672515600,`
- `"exp": 1672519200`
- `}`
    -
    - Le token est signé avec une clé secrète (stockée dans une variable d'environnement).
2. **Validation** :
    - Les requêtes protégées incluent le token dans l'en-tête `Authorization`.
    - Le middleware décode et vérifie le token avant d'autoriser l'accès.

7.2 Contrôle d'Accès Basé sur les Rôles (RBAC)

Le contrôle d'accès limite les actions disponibles en fonction du rôle utilisateur.

Permissions :

Rôle	Action Autorisée
Utilisateur	Planifier des trajets, consulter l'historique.
Cycliste	Déclarer des incidents.
Gestionnaire	Résoudre des incidents, gérer les vélos.
Administrateur	Accès complet.

7.3 Sécurité des Données

1. **Chiffrement des mots de passe** :
    - Utilisation de `bcrypt` pour hasher les mots de passe.
- Exemple :
  ```
  const hashedPassword =
  bcrypt.hashSync(plainTextPassword, 10);
  ```
    -
2. **Protection contre les attaques courantes** :
    - **CSRF** : Utilisation de jetons anti-CSRF pour sécuriser les formulaires sensibles.
    - **XSS** : Validation stricte des entrées pour empêcher l'exécution de scripts malveillants.
3. **Logs de Sécurité** :
    - Toutes les tentatives de connexion et les actions sensibles sont enregistrées.

---

8. Tests et Validation du Code

8.1 Introduction

Les tests garantissent la qualité du code et permettent d'identifier les bogues avant la mise en production. Voici les principaux objectifs des tests dans ce projet :

- Vérifier le bon fonctionnement des fonctionnalités critiques.
- Garantir la robustesse face aux entrées invalides.
- Assurer la performance des algorithmes.
- Prévenir les failles de sécurité.

8.2 Technologies de Test

- **Mocha** : Framework pour les tests unitaires et d'intégration.
- **Chai** : Assertions pour écrire des tests lisibles.
- **Supertest** : Pour tester les endpoints RESTful.
- **Sinon** : Permet de mocker des fonctions ou modules.
- **nyc** : Génération de rapports de couverture des tests.

8.3 Structure des Tests

Les tests sont divisés par catégories :

```
test/
│
├── unit/                # Tests unitaires
│   ├── trajetService.test.js
│   ├── utilisateurModel.test.js
│   └── notificationService.test.js
│
```

```
├── integration/          # Tests d'intégration
│   ├── apiIncident.test.js
│   ├── apiTrajet.test.js
│   └── authMiddleware.test.js
│
├── performance/          # Tests de performance
│   ├── trajetPerformance.test.js
│   └── notificationPerformance.test.js
│
└── security/             # Tests de sécurité
    └── sqlInjection.test.js
```

8.4 Tests Unitaires

8.4.1 Service Trajet : Calcul Optimal

Valider que le chemin optimal est correctement calculé avec des graphes complexes.

```
describe('Service Trajet - Calcul Optimal', () => {
  it('retourne le chemin le plus court avec plusieurs
alternatives', async () => {
    const graph = {
      1: [{ node: 2, weight: 10 }, { node: 3, weight: 5
}],
      2: [{ node: 4, weight: 10 }],
      3: [{ node: 4, weight: 2 }]
    };
    const path = await calculateOptimalRoute(graph, 1,
4);
    expect(path).to.eql([1, 3, 4]); // Chemin optimal
attendu
  });
```

```
  it('génère une erreur si le chemin est inexistant',
async () => {
    const graph = { 1: [{ node: 2, weight: 10 }] };
    const path = await calculateOptimalRoute(graph, 1,
3);
    expect(path).to.be.null; // Pas de chemin disponible
  });
});
```

8.4.2 Modèle Vélo : Gestion de l'Autonomie

Valider que les vélos ont un statut correct en fonction de leur autonomie.

```
describe('Modèle Vélo - Gestion de l'Autonomie', () => {
  it('identifie un vélo comme nécessitant une recharge si
l'autonomie est faible', async () => {
    const velo = await Velo.create({ autonomie: 5 }); //
Autonomie critique
    expect(velo.needsRecharge()).to.be.true;
  });

  it('retourne un statut "OK" pour une autonomie
suffisante', async () => {
    const velo = await Velo.create({ autonomie: 50 });
    expect(velo.needsRecharge()).to.be.false;
  });
});
```

8.5 Tests d'Intégration

8.5.1 Middleware Auth : Validation des Tokens JWT

Simule une requête avec un token JWT valide et invalide.

```
describe('Middleware Auth - Validation JWT', () => {
```

```
  it('autorise l'accès avec un token valide', async () =>
{
    const token = generateToken({ id: 1, role: 'Cycliste'
}); // Fonction pour générer un token
    const res = await request(app)
      .get('/api/protected-route')
      .set('Authorization', `Bearer ${token}`);
    expect(res.statusCode).to.equal(200);
  });

  it('refuse l'accès avec un token invalide', async () =>
{
    const res = await request(app)
      .get('/api/protected-route')
      .set('Authorization', 'Bearer invalid-token');
    expect(res.statusCode).to.equal(401); // Non autorisé
  });
});
```

8.5.2 Endpoint Trajet : Calcul d'Itinéraire

Simule une requête POST pour calculer un itinéraire.

```
describe('API Trajet - POST /api/trajets/optimiser', ()
=> {
  it('retourne un itinéraire optimisé', async () => {
    const res = await request(app)
      .post('/api/trajets/optimiser')
      .send({ depart: 1, arrivee: 4 });
    expect(res.statusCode).to.equal(200);

expect(res.body).to.have.property('path').that.is.an('arr
ay');
    expect(res.body.distance).to.be.a('number');
  });
```

```
  it('renvoie une erreur si le chemin est inexistant',
async () => {
    const res = await request(app)
      .post('/api/trajets/optimiser')
      .send({ depart: 1, arrivee: 999 }); // Arrêt
inexistant
    expect(res.statusCode).to.equal(400); // Chemin non
valide
  });
});
```

8.6 Tests de Performance

8.6.1 Itinéraire Complexe

Mesure le temps nécessaire pour calculer un trajet dans un graphe de grande taille.

```
describe('Performance - Calcul d'Itinéraire Complexe', ()
=> {
  it('traite un graphe de 1000 nœuds en moins de 500ms',
async () => {
    const largeGraph = generateLargeGraph(1000); //
Fonction générant un graphe complexe
    const start = performance.now();
    await calculateOptimalRoute(largeGraph, 1, 1000);
    const end = performance.now();
    expect(end - start).to.be.lessThan(500);
  });
});
```

8.7 Tests de Résilience

8.7.1 Gestion des Entrées Malformées

Teste les réponses de l'API en cas d'erreurs utilisateur.

```
describe('Résilience - Entrées Invalides', () => {
  it('retourne une erreur 400 pour des champs manquants',
async () => {
    const res = await request(app)
      .post('/api/incidents')
      .send({}); // Données manquantes
    expect(res.statusCode).to.equal(400);
  });

  it('protège contre les dépassements de mémoire avec des
entrées volumineuses', async () => {
    const largeData = 'x'.repeat(1e6); // 1MB de données
    const res = await request(app)
      .post('/api/incidents')
      .send({ description: largeData });
    expect(res.statusCode).to.equal(413); // Payload trop
large
  });
});
```

8.8 Tests de Sécurité

8.8.1 Injection SQL

Vérifie que l'API est protégée contre les attaques SQL.

```
describe('Sécurité - Protection contre les Injections
SQL', () => {
  it('bloque une tentative d'injection SQL', async () =>
{
```

```
    const res = await request(app)
      .post('/api/incidents')
      .send({ description: "'; DROP TABLE incidents; --",
type: 'panne', veloId: 3 });
    expect(res.statusCode).to.equal(400); // Requête
rejetée
  });
});
```

8.8.2 Prévention XSS

Teste la protection contre les scripts injectés dans les champs de texte.

```
describe('Sécurité - Protection contre XSS', () => {
  it('échappe les scripts injectés dans les
descriptions', async () => {
    const res = await request(app)
      .post('/api/incidents')
      .send({ description:
"<script>alert('hack');</script>", type: 'panne', veloId:
3 });
    expect(res.statusCode).to.equal(201); // Incident
créé

expect(res.body.description).to.not.include('<script>');
  });
});
```

8.9 Lancement des Tests

Exécuter tous les tests :

```
npm test
```

Générer un rapport de couverture :

```
npx nyc npm test
```

---

9. Bonnes Pratiques et Performances

- **Optimisation des requêtes** : Utilisation de Sequelize.
- **Mise en cache** : Réduction des temps de calcul avec Redis.
- **Transactions** : Préserver l'intégrité des données.

---

10. Évolutions Futures et Scalabilité

Microservices et Scalabilité Horizontale

Séparation des modules (trajets, incidents) pour un scaling efficace.

Support Multilingue

Faciliter l'adoption par des utilisateurs internationaux.