Fiche Technique du Projet Backend Progitek System

1. Introduction

Ce document présente la fiche technique détaillée du projet backend "Progitek System", une API développée en TypeScript. L'objectif principal de cette API est de fournir une infrastructure robuste et évolutive pour la gestion technique des opérations de Progitek, incluant la gestion des missions, des clients, des techniciens, et d'autres fonctionnalités essentielles à un système de gestion d'entreprise. L'architecture est conçue pour être modulaire, performante et sécurisée, en s'appuyant sur des technologies modernes et des pratiques de développement éprouvées.

2. Technologies Utilisées

Le projet backend de Progitek System est construit sur une pile technologique moderne et robuste, principalement axée sur l'écosystème JavaScript/TypeScript. Le choix de ces technologies garantit une haute performance, une maintenabilité aisée et une grande flexibilité pour les évolutions futures. Voici les composants clés :

2.1. Langages et Runtimes

- **TypeScript (v5.3.2)** : Le langage principal utilisé pour le développement. TypeScript est un sur-ensemble typé de JavaScript qui apporte une meilleure robustesse au code, une détection précoce des erreurs et une meilleure lisibilité, particulièrement utile pour les projets de grande envergure. Il améliore la productivité des développeurs grâce à l'autocomplétion et à la refactorisation.
- **Node.js (v18+)** : Le runtime JavaScript côté serveur. Node.js permet d'exécuter du code JavaScript en dehors d'un navigateur web, offrant une architecture non

bloquante et événementielle, idéale pour les applications backend à forte concurrence et les API RESTful.

2.2. Frameworks et Bibliothèques Principales

- **Express (v4.18.2)** : Un framework web minimaliste et flexible pour Node.js, utilisé pour construire l'API RESTful. Express fournit un ensemble robuste de fonctionnalités pour les applications web et mobiles, y compris le routage, la gestion des requêtes et des réponses, et l'intégration de middleware.
- **Prisma (v5.22.0)** : Un ORM (Object-Relational Mapper) de nouvelle génération pour Node.js et TypeScript. Prisma simplifie l'interaction avec la base de données en fournissant un schéma déclaratif et un client typé. Il gère les migrations de base de données et offre une expérience de développement fluide et sécurisée pour les opérations CRUD (Create, Read, Update, Delete).

2.3. Base de Données

- **PostgreSQL (v14+)** : Une base de données relationnelle open-source, puissante et hautement fiable. PostgreSQL est réputé pour sa conformité aux standards, sa robustesse, ses fonctionnalités avancées (transactions ACID, intégrité référentielle) et sa capacité à gérer de grands volumes de données de manière efficace. Il est le choix idéal pour les applications d'entreprise nécessitant une forte cohérence des données.

2.4. Gestion des Dépendances et Outils de Build

- **pnpm** : Un gestionnaire de paquets rapide et efficace pour Node.js. pnpm se distingue par son approche de lien symbolique pour les dépendances, ce qui permet d'économiser de l'espace disque et d'accélérer les installations par rapport à npm ou Yarn.
- **TypeScript Compiler (tsc)** : Utilisé pour compiler le code TypeScript en JavaScript exécutable. Le processus de build transforme les fichiers `.ts` en fichiers `.js` dans le répertoire `dist/`.
- **tsx** : Un exécuteur TypeScript qui permet d'exécuter des fichiers TypeScript directement sans compilation préalable, facilitant le développement et le débogage avec des fonctionnalités comme le hot reload.

2.5. Sécurité et Performance

- **bcryptjs (v2.4.3)** : Une bibliothèque pour le hachage de mots de passe. Elle est utilisée pour stocker les mots de passe de manière sécurisée en les hachant avec un algorithme robuste, protégeant ainsi les informations d'identification des utilisateurs.
- **jsonwebtoken (v9.0.5)** : Implémente les JSON Web Tokens (JWT) pour l'authentification et l'autorisation. Les JWT sont utilisés pour créer des tokens d'accès et de rafraîchissement sécurisés, permettant une gestion stateless des sessions utilisateur.
- **helmet (v7.1.0)** : Une collection de middleware Express qui aide à sécuriser l'application en définissant divers en-têtes HTTP liés à la sécurité. Il protège contre les vulnérabilités courantes telles que les attaques XSS, le détournement de clics et d'autres menaces web.
- **express-rate-limit (v7.1.5)** : Middleware pour limiter le nombre de requêtes répétées à des points d'accès publics ou à des API, protégeant ainsi l'application contre les attaques par force brute et les abus de service.
- **compression (v1.7.4)** : Middleware Express pour la compression des réponses HTTP. Il réduit la taille des données transférées entre le serveur et le client, améliorant ainsi les performances et la vitesse de chargement de l'application.
- **cors (v2.8.5)** : Middleware Express pour l'activation de Cross-Origin Resource Sharing (CORS). Il permet aux applications web s'exécutant sur un domaine différent d'interagir avec l'API, ce qui est essentiel pour les architectures frontend/backend séparées.

2.6. Outils de Développement et Utilitaires

- **Winston (v3.11.0)** : Une bibliothèque de logging polyvalente pour Node.js. Winston permet de gérer les logs de l'application avec différents niveaux de sévérité et de les acheminer vers diverses destinations (console, fichiers, bases de données, etc.), facilitant le débogage et le monitoring.
- **Swagger-jsdoc (v6.2.8) & Swagger-ui-express (v5.0.0)** : Utilisés pour générer et servir automatiquement la documentation de l'API au format OpenAPI (anciennement Swagger). Cela permet aux développeurs frontend et aux consommateurs de l'API de comprendre facilement les points d'accès, les paramètres et les réponses de l'API.

- **Joi (v17.11.0) & express-validator (v7.0.1)** : Bibliothèques de validation de données. Joi est utilisé pour définir des schémas de validation robustes pour les données entrantes, tandis qu'express-validator intègre cette validation dans le pipeline de requêtes Express, assurant l'intégrité des données.
- **Bull (v4.12.0)** : Une bibliothèque de gestion de files d'attente basée sur Redis. Elle est utilisée pour gérer les tâches asynchrones et les opérations de longue durée en arrière-plan, améliorant la réactivité de l'API et la gestion des ressources.
- **Redis (v4.7.1)** : Un magasin de données en mémoire, utilisé comme cache et pour la gestion des files d'attente avec Bull. Redis offre des performances très élevées pour les opérations de lecture/écriture, ce qui est crucial pour la mise en cache et la gestion des sessions.
- **Nodemailer (v6.9.7)** : Une bibliothèque pour l'envoi d'e-mails. Elle est utilisée pour les fonctionnalités de notification par e-mail, telles que la réinitialisation de mot de passe ou les alertes système.
- **date-fns (v4.1.0)** : Une bibliothèque moderne de manipulation de dates. Elle simplifie les opérations sur les dates et les heures, telles que le formatage, le calcul d'intervalles et la comparaison.
- **Multer (v2.0.0)** : Middleware Express pour la gestion des téléchargements de fichiers. Il est utilisé pour traiter les données de formulaire `multipart/form-data`, principalement pour l'upload de fichiers.

2.7. Tests

- **Jest (v29.7.0)** : Un framework de test JavaScript populaire. Jest est utilisé pour les tests unitaires et d'intégration, garantissant la fiabilité et la robustesse du code. Il offre des fonctionnalités de mock, de couverture de code et une exécution rapide des tests.
- **Supertest (v7.0.0)** : Une bibliothèque qui facilite le test des applications HTTP. Elle est utilisée en conjonction avec Jest pour tester les points d'accès de l'API en simulant des requêtes HTTP et en vérifiant les réponses.

Cette combinaison de technologies offre une base solide pour un backend performant, sécurisé et facile à maintenir, capable de répondre aux exigences d'un système de gestion technique complexe comme Progitek System.

3. Architecture du Projet

L'architecture du projet backend de Progitek System suit une approche modulaire et structurée, favorisant la séparation des préoccupations et la maintenabilité du code. L'organisation des répertoires reflète cette modularité, avec des dossiers dédiés à la configuration, aux contrôleurs, aux middlewares, aux routes, aux types de données, aux utilitaires et au point d'entrée principal du serveur. Cette structure est typique des applications Express bien organisées, facilitant la navigation et la compréhension du flux d'exécution.

```
 src/
├── config/          # Fichiers de configuration pour la base de données, les logs, le cache, et Swagger.
├── controllers/     # Contient la logique métier et les gestionnaires de requêtes pour chaque ressource API (e.g., authController, clientController, devisController).
├── middleware/      # Implémente les fonctions middleware Express pour l'authentification, la mise en cache, le monitoring, la limitation de débit et la validation des données.
├── prisma/          # Contient le schéma Prisma (`schema.prisma`) définissant le modèle de données et les migrations de base de données. Inclut également le fichier `seed.ts` pour le peuplement initial de la base.
├── routes/          # Définit les routes API et associe les chemins d'URL aux contrôleurs correspondants (e.g., auth.ts, clients.ts, devis.ts).
├── types/           # Définit les interfaces et types TypeScript personnalisés utilisés à travers l'application, assurant une forte typisation.
├── utils/           # Fournit des fonctions utilitaires réutilisables, telles que des générateurs, des fonctions de pagination, des gestionnaires de réponse et des calculateurs spécifiques (e.g., devisCalculator).
└── server.ts        # Le point d'entrée principal de l'application, où le serveur Express est initialisé, les middlewares globaux sont configurés et les routes sont chargées.
```

Cette organisation permet une claire distinction entre les différentes couches de l'application :

- **Couche de Configuration (`config/`)** : Centralise tous les paramètres de l'application, rendant la configuration facile à gérer et à adapter à différents environnements (développement, production).
- **Couche de Contrôle (`controllers/`)** : Gère la logique spécifique à chaque ressource. Chaque contrôleur est responsable de la réception des requêtes, de l'interaction avec la logique métier (potentiellement via des services non explicitement séparés dans cette structure, mais implicitement gérés par les contrôleurs eux-mêmes), et de l'envoi des réponses.

- **Couche Middleware (`middleware/`)** : Intercepte les requêtes avant qu'elles n'atteignent les contrôleurs pour effectuer des opérations transversales comme l'authentification, la validation, la journalisation ou la mise en cache. Cela réduit la duplication de code et améliore la modularité.
- **Couche de Routage (`routes/`)** : Définit les chemins d'accès de l'API et dirige les requêtes vers les contrôleurs appropriés. Cette séparation rend l'API facile à comprendre et à étendre.
- **Couche de Données (`prisma/`)** : Gérée par Prisma, cette couche est responsable de l'interaction avec la base de données. Le schéma Prisma sert de source unique de vérité pour la structure de la base de données et les modèles de données de l'application.
- **Couche d'Utilitaires (`utils/`) et de Types (`types/`)** : Ces couches supportent les autres composants en fournissant des fonctions génériques et des définitions de types, garantissant la cohérence et la réutilisabilité du code.

Cette architecture favorise un développement agile, une meilleure collaboration entre les développeurs et une maintenance simplifiée du projet sur le long terme.

4. Fonctionnalités Clés

L'API backend de Progitek System offre un ensemble complet de fonctionnalités pour la gestion des opérations techniques et administratives. Ces fonctionnalités sont exposées via des points d'accès RESTful, permettant une interaction fluide avec les applications clientes (frontend, mobile, etc.). Les principales catégories de fonctionnalités incluent :

4.1. Authentification et Gestion des Utilisateurs

Le système d'authentification est basé sur les JSON Web Tokens (JWT) avec un mécanisme de refresh token pour une sécurité accrue et une gestion des sessions sans état. Les fonctionnalités incluent :

- **Connexion des utilisateurs (`POST /api/auth/login`)** : Permet aux utilisateurs de s'authentifier avec leurs identifiants et d'obtenir un token d'accès et un refresh token.

- **Rafraîchissement des tokens (`POST /api/auth/refresh`)** : Permet de renouveler le token d'accès sans nécessiter une nouvelle connexion, améliorant l'expérience utilisateur et la sécurité.
- **Accès au profil utilisateur (`GET /api/auth/profile`)** : Permet aux utilisateurs authentifiés de récupérer leurs informations de profil.
- **Gestion des utilisateurs (`/api/users`)** : L'API inclut des contrôleurs pour la gestion complète des utilisateurs, permettant la création, la lecture, la mise à jour et la suppression des comptes utilisateurs. Cela inclut probablement la gestion des rôles et des permissions, bien que les détails spécifiques des rôles ne soient pas explicitement mentionnés dans le README.

4.2. Gestion des Missions et Interventions

Au cœur du système, la gestion des missions et des interventions permet de suivre les tâches techniques assignées aux techniciens.

- **Liste des missions (`GET /api/missions`)** : Récupère toutes les missions enregistrées dans le système.
- **Détails d'une mission (`GET /api/missions/{numIntervention}`)** : Affiche les informations détaillées d'une mission spécifique, identifiée par son numéro d'intervention.
- **Création de mission (`POST /api/missions`)** : Permet d'ajouter une nouvelle mission au système.
- **Modification de mission (`PUT /api/missions/{numIntervention}`)** : Met à jour les informations d'une mission existante.
- **Suppression de mission (`DELETE /api/missions/{numIntervention}`)** : Supprime une mission du système.
- **Gestion des interventions (`/api/interventions`)** : Le contrôleur `interventionController` suggère une gestion détaillée des interventions, qui sont probablement des sous-tâches ou des étapes spécifiques au sein d'une mission. Cela pourrait inclure le suivi de l'état, des rapports d'intervention, et l'affectation à des techniciens.

4.3. Gestion des Clients

Le module de gestion des clients permet de maintenir une base de données complète des clients de Progitek.

- **Liste des clients (`GET /api/clients`)** : Récupère la liste de tous les clients.
- **Détails d'un client (`GET /api/clients/{id}`)** : Affiche les informations détaillées d'un client spécifique.
- **Création de client (`POST /api/clients`)** : Permet d'enregistrer un nouveau client.
- **Modification de client (`PUT /api/clients/{id}`)** : Met à jour les informations d'un client existant.
- **Suppression de client (`DELETE /api/clients/{id}`)** : Supprime un client du système.

4.4. Gestion des Techniciens

Ce module est dédié à la gestion du personnel technique, incluant leurs informations et leurs affectations.

- **Liste des techniciens (`GET /api/techniciens`)** : Récupère la liste de tous les techniciens.
- **Détails d'un technicien (`GET /api/techniciens/{id}`)** : Affiche les informations détaillées d'un technicien spécifique.
- **Création de technicien (`POST /api/techniciens`)** : Permet d'ajouter un nouveau technicien.
- **Modification de technicien (`PUT /api/techniciens/{id}`)** : Met à jour les informations d'un technicien existant.
- **Suppression de technicien (`DELETE /api/techniciens/{id}`)** : Supprime un technicien du système.

4.5. Gestion des Devis et Factures

Le système intègre des fonctionnalités pour la création et la gestion des devis et des factures, essentielles pour le suivi financier.

- **Gestion des devis (`/api/devis`)** : Le `devisController` et le `devisCalculator` indiquent des fonctionnalités complètes pour la création, la modification, le calcul et le suivi des devis. Cela inclut probablement la génération de devis, l'ajout de postes, le calcul des totaux et la gestion des statuts (validé, en attente, etc.).
- **Gestion des factures (`/api/factures`)** : Le `factureController` gère les opérations liées aux factures, telles que la génération à partir de devis ou d'interventions, le suivi des paiements, et la gestion des statuts de facture.
- **Types de paiement (`/api/typesPaiement`)** : Le `typePaiementController` permet de gérer les différents types de paiement acceptés par le système, offrant une flexibilité dans les options de facturation.

4.6. Gestion du Stock

Le module de stock permet de suivre les équipements et les pièces utilisées dans les interventions.

- **Gestion du stock (`/api/stock`)** : Le `stockController` gère les entrées, les sorties et l'inventaire des articles en stock, assurant une disponibilité adéquate des ressources pour les techniciens.

4.7. Rapports et Tableau de Bord

Pour le suivi et l'analyse des activités, l'API propose des fonctionnalités de reporting et un tableau de bord.

- **Tableau de bord (`/api/dashboard`)** : Le `dashboardController` fournit des agrégations de données et des indicateurs clés de performance (KPIs) pour offrir une vue d'ensemble des opérations (missions en cours, chiffre d'affaires, etc.).
- **Rapports (`/api/rapports`)** : Le `rapportController` permet de générer divers rapports sur les activités, les performances des techniciens, les statistiques de missions, etc., facilitant la prise de décision.

4.8. Communication et Notifications

Le système intègre des mécanismes de communication pour améliorer l'interaction entre les utilisateurs et le suivi des événements.

- **Messages (`/api/messages`)** : Le `messageController` gère les communications internes ou externes, permettant l'échange de messages entre les utilisateurs du système.
- **Notifications (`/api/notifications`)** : Le `notificationController` gère la création et la distribution de notifications (par exemple, pour de nouvelles missions, des mises à jour de statut, ou des alertes), potentiellement via e-mail (grâce à Nodemailer) ou des notifications in-app.

Ces fonctionnalités couvrent les besoins essentiels d'un système de gestion technique, offrant une solution complète pour la planification, l'exécution et le suivi des opérations.

5. Dépendances et Intégrations

Le projet backend de Progitek System s'appuie sur un ensemble de dépendances et d'intégrations pour étendre ses capacités et interagir avec d'autres services. Ces éléments sont cruciaux pour le bon fonctionnement de l'application et pour la mise en œuvre de fonctionnalités avancées.

5.1. Gestion des Dépendances

Les dépendances du projet sont gérées via `pnpm`, et sont listées dans le fichier `package.json`. Elles se divisent en deux catégories principales :

- **Dépendances de Production (`dependencies`)** : Ce sont les bibliothèques essentielles au fonctionnement de l'application en production. Elles incluent des éléments comme Express pour le serveur web, Prisma pour l'ORM, bcryptjs pour le hachage des mots de passe, jsonwebtoken pour l'authentification JWT, et d'autres modules pour la gestion du cache (Redis, Bull), l'envoi d'e-mails (Nodemailer), la validation (Joi, express-validator), etc. La présence de `compression` et `helmet` souligne l'importance accordée à la performance et à la sécurité.
- **Dépendances de Développement (`devDependencies`)** : Ces bibliothèques sont utilisées uniquement pendant le développement et les tests. Elles comprennent `@types/*` pour les définitions de types TypeScript, Jest et Supertest pour les

tests, tsx pour l'exécution directe de TypeScript, et ESLint pour le linting du code. Ces outils améliorent la qualité du code et l'efficacité du développement.

5.2. Intégrations Clés

Le backend est conçu pour s'intégrer avec plusieurs systèmes et services externes ou internes :

- **Base de Données PostgreSQL** : L'intégration principale se fait avec PostgreSQL via Prisma. Prisma gère la connexion, les requêtes et les migrations de schéma, assurant une interaction fluide et typée avec la base de données relationnelle. Les migrations sont versionnées, ce qui facilite l'évolution du schéma de base de données au fil du temps.
- **Redis** : Utilisé comme magasin de données en mémoire pour la mise en cache et la gestion des files d'attente. L'intégration avec Bull permet de décharger les tâches de longue durée (comme l'envoi d'e-mails ou le traitement de rapports complexes) dans une file d'attente, améliorant ainsi la réactivité de l'API et la gestion des ressources serveur.
- **Système de Fichiers (pour les logs)** : Le projet utilise Winston pour la journalisation, avec la capacité d'écrire les logs dans des fichiers (`logs/progitek.log`). Cela est crucial pour le débogage, le monitoring et l'audit des activités de l'application.
- **Documentation API (Swagger/OpenAPI)** : L'intégration de Swagger-jsdoc et Swagger-ui-express permet de générer et de servir automatiquement une documentation interactive de l'API. Cette documentation est essentielle pour les développeurs frontend et les consommateurs de l'API, car elle décrit les points d'accès, les paramètres, les modèles de données et les réponses, facilitant ainsi l'intégration et la consommation de l'API.
- **Système d'E-mail (via Nodemailer)** : Nodemailer est intégré pour permettre l'envoi d'e-mails. Cela est utilisé pour des fonctionnalités telles que les notifications, les réinitialisations de mot de passe, ou d'autres communications automatisées avec les utilisateurs ou les administrateurs.
- **Postman Collections** : La présence de fichiers `postman-collection-complete.json` et `postman-collection-corrected.json` indique une intégration étroite avec Postman, un outil populaire pour tester et documenter

les API. Ces collections facilitent le test manuel et automatisé des points d'accès de l'API.

Ces dépendances et intégrations sont choisies pour leur robustesse, leur performance et leur capacité à s'adapter aux besoins d'une application d'entreprise, garantissant que le backend de Progitek System est non seulement fonctionnel mais aussi évolutif et maintenable.

6. Sécurité et Monitoring

La sécurité et le monitoring sont des aspects fondamentaux du projet backend de Progitek System, garantissant la protection des données, la disponibilité du service et la détection rapide des anomalies. Plusieurs mécanismes et outils sont mis en œuvre pour adresser ces préoccupations.

6.1. Mesures de Sécurité

Le projet intègre plusieurs couches de sécurité pour protéger l'API et les données des utilisateurs :

- **Authentification JWT avec Refresh Tokens** : Comme mentionné précédemment, l'utilisation de JSON Web Tokens (JWT) pour l'authentification assure que seules les requêtes autorisées peuvent accéder aux ressources protégées. Le mécanisme de refresh token renforce cette sécurité en limitant la durée de vie des tokens d'accès et en permettant leur renouvellement sécurisé sans nécessiter une nouvelle saisie des identifiants.
- **Hachage des Mots de Passe (bcryptjs)** : Les mots de passe des utilisateurs ne sont jamais stockés en clair dans la base de données. Ils sont hachés à l'aide de `bcryptjs`, un algorithme de hachage cryptographique robuste et résistant aux attaques par force brute et aux tables arc-en-ciel. Cela protège les informations d'identification même en cas de compromission de la base de données.
- **Protection contre les Vulnérabilités Web Courantes (Helmet)** : Le middleware `helmet` est utilisé pour définir divers en-têtes HTTP sécurisés qui protègent l'application contre un éventail de vulnérabilités web courantes, notamment :
  - **XSS (Cross-Site Scripting)** : En définissant l'en-tête `X-XSS-Protection`.
  - **Clickjacking** : En utilisant l'en-tête `X-Frame-Options`.

    - **MIME Type Sniffing** : En définissant l'en-tête `X-Content-Type-Options`.
    - **HTTPS Enforcement** : En utilisant l'en-tête `Strict-Transport-Security` (HSTS) pour forcer les connexions sécurisées.
- **Limitation de Débit (express-rate-limit)** : Pour prévenir les attaques par déni de service (DoS) et les tentatives de force brute, `express-rate-limit` est configuré pour limiter le nombre de requêtes qu'un utilisateur peut effectuer sur une période donnée. Cela aide à protéger les points d'accès sensibles comme les pages de connexion.
- **Validation des Données (Joi, express-validator)** : Toutes les données entrantes via les requêtes API sont rigoureusement validées avant d'être traitées ou stockées. Cela empêche les injections de données malveillantes (SQL injection, NoSQL injection) et garantit l'intégrité et la cohérence des données dans le système.
- **CORS Configuration (cors)** : La configuration explicite de CORS permet de contrôler quels domaines sont autorisés à interagir avec l'API, prévenant ainsi les attaques de type Cross-Site Request Forgery (CSRF) et assurant que seules les applications frontend autorisées peuvent accéder aux ressources.

6.2. Monitoring et Observabilité

Le projet intègre des outils et des pratiques pour surveiller la santé et les performances de l'application :

- **Journalisation (Winston)** : Un système de journalisation complet est mis en place avec `Winston`. Les logs sont écrits dans le fichier `logs/progitek.log`, ce qui permet de suivre les événements de l'application, les erreurs, les avertissements et les informations générales. Une journalisation adéquate est essentielle pour le débogage, l'audit de sécurité et l'analyse des performances.
- **Health Check (`GET /api/health`)** : Un point d'accès `/api/health` est exposé pour permettre aux systèmes de monitoring externes de vérifier l'état de santé de l'application. Ce endpoint peut retourner un statut simple (par exemple, 200 OK) si l'application est opérationnelle, ou des informations plus détaillées sur l'état des dépendances (base de données, Redis, etc.).
- **Métriques (Port 9090)** : Le README mentionne la disponibilité de métriques sur le port 9090 (si activé). Cela suggère l'intégration d'un système de collecte de métriques (potentiellement Prometheus ou un outil similaire) qui expose des

données sur les performances de l'application (temps de réponse, nombre de requêtes, utilisation des ressources, etc.). Ces métriques sont cruciales pour l'observabilité et l'optimisation des performances.

- **Monitoring Middleware** : La présence d'un fichier `src/middleware/monitoring.ts` indique qu'un middleware de monitoring personnalisé est utilisé. Ce middleware peut collecter des données sur les requêtes, les temps de réponse et d'autres indicateurs de performance avant de les envoyer à un système de monitoring.

L'ensemble de ces mesures de sécurité et de monitoring contribue à la robustesse et à la fiabilité du backend de Progitek System, permettant une gestion proactive des risques et une maintenance efficace de l'application.

7. Déploiement et Tests

Le processus de déploiement et la stratégie de test sont des composants essentiels pour assurer la qualité et la disponibilité du projet backend de Progitek System. Le projet fournit des scripts et des configurations pour faciliter ces étapes.

7.1. Déploiement

Le déploiement du backend est conçu pour être simple et reproductible, avec des instructions claires fournies dans le `README.md` et un guide de déploiement (`GUIDE_DEPLOIEMENT.md`) mentionné pour des instructions complètes. Les étapes clés pour le déploiement en production sont :

1. **Build de l'application** : Avant le déploiement, le code TypeScript doit être compilé en JavaScript. Cela est réalisé via la commande `pnpm run build` , qui exécute `tsc` (TypeScript Compiler). Le code compilé est ensuite placé dans le répertoire `dist/` .
2. **Démarrage en Production** : Une fois le build effectué, l'application peut être démarrée en mode production via `pnpm start` ou `pnpm run start:prod` . Ces commandes exécutent le fichier `dist/server.js` avec `node` , potentiellement avec des variables d'environnement spécifiques à la production (comme `NODE_ENV=production`).

3. **Configuration de l'Environnement** : Le déploiement nécessite la configuration de variables d'environnement critiques, telles que `DATABASE_URL` (pour la connexion à la base de données PostgreSQL), `JWT_SECRET` et `JWT_REFRESH_SECRET` (pour la sécurité des tokens JWT), et `CORS_ORIGIN` (pour la gestion des origines autorisées). Ces variables sont généralement gérées via des fichiers `.env` ou des systèmes de gestion de secrets dans l'environnement de production.
4. **Prérequis de Base de Données** : Le déploiement implique également la configuration et la migration de la base de données PostgreSQL. Les commandes `pnpm run db:generate` (pour générer le client Prisma) et `pnpm run db:migrate` (pour exécuter les migrations de schéma) sont essentielles pour préparer la base de données à l'utilisation par l'application. L'option `pnpm run db:seed` peut être utilisée pour peupler la base de données avec des données initiales, ce qui est utile pour les environnements de développement ou de test.

Le projet est conçu pour être déployé sur des environnements supportant Node.js et PostgreSQL, ce qui inclut la plupart des plateformes cloud et des serveurs dédiés.

7.2. Tests

Une stratégie de test robuste est mise en place pour garantir la qualité et la fiabilité du code. Le projet utilise Jest comme framework de test principal, complété par Supertest pour les tests d'intégration des API.

- **Tests Unitaires et d'Intégration (Jest)** : La commande `pnpm test` exécute la suite de tests. Jest est configuré pour tester les fichiers TypeScript, comme indiqué par la présence de `jest.config.js` et l'utilisation de `ts-jest`. Les tests sont situés dans le répertoire `tests/` et couvrent des modules clés comme l'authentification (`auth.test.ts`) et les clients (`clients.test.ts`). Ces tests vérifient que les différentes unités de code et les interactions entre elles fonctionnent comme prévu.
- **Tests d'API (Supertest)** : Supertest est utilisé pour simuler des requêtes HTTP vers l'API et vérifier les réponses. Cela permet de tester les points d'accès de l'API de manière isolée et de s'assurer que les contrôleurs et les routes fonctionnent correctement, y compris la validation des données, l'authentification et les réponses attendues.

- **Collections Postman** : La présence de collections Postman (`postman-collection-complete.json` , `postman-collection-corrected.json`) indique que des tests manuels ou automatisés via Postman sont également utilisés. Ces collections définissent des requêtes HTTP préconfigurées pour tester les différents endpoints de l'API, ce qui est utile pour le développement, le débogage et la validation rapide des fonctionnalités.

L'approche de test combinant tests unitaires, tests d'intégration et tests d'API assure une couverture complète et aide à identifier les bugs tôt dans le cycle de développement, contribuant ainsi à la stabilité et à la fiabilité de l'application en production.