# Fiche Technique du Projet Frontend Progitek System

## 1. Introduction

Ce document présente la fiche technique détaillée du projet frontend "Progitek System". Ce projet est une application web moderne conçue pour interagir avec l'API backend de Progitek System, offrant une interface utilisateur intuitive et réactive pour la gestion des opérations techniques. L'application est développée en utilisant des technologies frontend de pointe, garantissant une expérience utilisateur fluide et une maintenabilité aisée.

## 2. Technologies Utilisées

Le projet frontend de Progitek System est construit sur une pile technologique moderne et performante, principalement axée sur l'écosystème React et TypeScript. Le choix de ces technologies vise à offrir une interface utilisateur dynamique, une expérience de développement efficace et une maintenabilité à long terme. Voici les composants clés :

### 2.1. Langages et Runtimes

- **TypeScript (v5.5.3)** : Le langage de programmation principal. TypeScript, un sur-ensemble typé de JavaScript, apporte une robustesse accrue au code, une meilleure détection des erreurs en phase de développement et une lisibilité améliorée, ce qui est crucial pour les applications complexes. Il facilite la collaboration et la maintenance du code.
- **JavaScript (ESNext)** : Utilisé en conjonction avec TypeScript, le code est finalement transpilé en JavaScript pour être exécuté par les navigateurs web.

## 2.2. Frameworks et Bibliothèques Principales

- **React (v18.3.1)** : La bibliothèque JavaScript déclarative et basée sur des composants pour la construction d'interfaces utilisateur. React permet de créer des UI complexes à partir de petits morceaux de code isolés et réutilisables, appelés composants. Son approche basée sur le DOM virtuel optimise les mises à jour de l'interface, garantissant une haute performance.
- **Vite (v5.4.2)** : Un outil de build nouvelle génération pour les projets frontend. Vite se distingue par sa rapidité de démarrage du serveur de développement et ses mises à jour instantanées (Hot Module Replacement - HMR) grâce à l'utilisation de modules ES natifs. Il simplifie le processus de développement et de build, offrant une expérience fluide et rapide.
- **React Router DOM (v7.7.0)** : Une bibliothèque de routage déclaratif pour React. Elle permet de gérer la navigation au sein de l'application monopage (SPA), en associant des composants React à différentes URL et en gérant l'historique de navigation, les redirections et les paramètres d'URL.

## 2.3. Gestion de l'État et Requêtes API

- **Axios (v1.10.0)** : Un client HTTP basé sur les promesses pour le navigateur et Node.js. Axios est utilisé pour effectuer des requêtes HTTP vers l'API backend, facilitant l'interaction avec les services RESTful. Il offre des fonctionnalités telles que l'interception des requêtes/réponses, la transformation automatique des données JSON et la gestion des erreurs.
- **Hooks personnalisés (`src/hooks/`)** : Le projet utilise des hooks React personnalisés (e.g., `useAuth`, `useCache`, `useDashboard`, `useSettings`) pour encapsuler la logique d'état et les effets secondaires réutilisables. Cette approche favorise la réutilisabilité du code, la séparation des préoccupations et la simplification des composants fonctionnels.

## 2.4. Styles et Composants UI

- **Tailwind CSS (v3.4.1)** : Un framework CSS utilitaire-first qui permet de construire rapidement des designs personnalisés directement dans le balisage HTML. Tailwind CSS est hautement configurable et favorise une approche de développement rapide et cohérente, en évitant la nécessité d'écrire du CSS personnalisé pour chaque élément.

- **PostCSS (v8.4.35)** : Un outil pour transformer le CSS avec des plugins JavaScript. Il est utilisé en conjonction avec Tailwind CSS pour traiter et optimiser les feuilles de style, permettant des fonctionnalités avancées comme l'imbrication de règles CSS et l'auto-préfixage.
- **Lucide React (v0.344.0)** : Une bibliothèque d'icônes légère et personnalisable pour React. Elle fournit un ensemble d'icônes vectorielles qui peuvent être facilement intégrées et stylisées, améliorant l'esthétique et l'intuitivité de l'interface utilisateur.
- **clsx (v2.1.1)** : Une petite utilité pour construire des chaînes de noms de classes conditionnellement. Elle simplifie la gestion dynamique des classes CSS en fonction de l'état ou des props des composants React.

## 2.5. Outils de Développement et Utilitaires

- **ESLint (v9.9.1)** : Un outil d'analyse statique du code pour identifier les problèmes de programmation, les bugs, les erreurs de style et les constructions suspectes. ESLint aide à maintenir une qualité de code élevée et une cohérence stylistique à travers le projet.
- **Prettier (non listé explicitement mais souvent utilisé avec ESLint)** : Un formateur de code opinionné qui assure une mise en forme cohérente du code, réduisant les conflits de style et améliorant la lisibilité.
- **date-fns (v4.1.0)** : Une bibliothèque moderne et légère pour la manipulation des dates en JavaScript. Elle est utilisée pour formater, analyser et manipuler les dates de manière efficace et cohérente dans l'application.
- **React Hot Toast (v2.5.2)** : Une bibliothèque pour afficher des notifications toast légères et personnalisables. Elle est utilisée pour fournir un feedback non intrusif à l'utilisateur concernant les actions réussies, les erreurs ou les informations importantes.
- **Recharts (v3.1.0)** : Une bibliothèque de graphiques composable construite avec React et D3. Elle est utilisée pour visualiser des données complexes de manière interactive et esthétique, notamment pour les tableaux de bord et les rapports.

## 2.6. Tests

- **Vitest (v2.0.0)** : Un framework de test rapide et moderne, compatible avec Vite. Vitest est utilisé pour les tests unitaires et d'intégration des composants React et

de la logique métier. Il offre une expérience de développement rapide avec HMR pour les tests.

- **@testing-library/react (v16.0.0)** : Une suite de bibliothèques qui fournit des utilitaires pour tester les composants React de manière à se rapprocher le plus possible de la façon dont l'utilisateur interagirait avec l'application. Elle met l'accent sur les tests comportementaux plutôt que sur les détails d'implémentation.
- **@testing-library/jest-dom (v6.4.0)** : Fournit des matchers personnalisés pour Jest qui permettent de faire des assertions sur l'état du DOM, facilitant les tests d'interface utilisateur.
- **jsdom (v25.0.0)** : Une implémentation JavaScript de la spécification WHATWG DOM et HTML Standard, utilisée par Vitest pour simuler un environnement de navigateur sans tête pour les tests.

Cette combinaison de technologies offre une base solide pour un frontend performant, réactif et facile à maintenir, capable de fournir une expérience utilisateur de haute qualité pour le système Progitek.

## 3. Architecture du Projet

L'architecture du projet frontend de Progitek System est conçue pour être modulaire, maintenable et évolutive, en suivant les meilleures pratiques de développement d'applications React. La structure des dossiers reflète une séparation claire des préoccupations, facilitant la navigation, la compréhension du code et la collaboration entre les développeurs. Le répertoire `src/` est le cœur de l'application et contient les sous-répertoires suivants :

```
src/
├── assets/          # Contient les ressources statiques comme les images, les
icônes, les polices, etc.
├── components/      # Regroupe les composants React réutilisables et atomiques
(boutons, cartes, formulaires, etc.).
├── config/          # Fichiers de configuration de l'application, tels que les
constantes ou les variables d'environnement.
├── hooks/           # Contient les hooks React personnalisés pour encapsuler
la logique d'état réutilisable et les effets secondaires.
├── lib/             # Bibliothèques utilitaires ou configurations spécifiques
(e.g., configuration Axios pour l'API).
├── pages/           # Composants de niveau supérieur qui représentent des vues
ou des pages complètes de l'application (e.g., LoginPage, DashboardPage,
ClientsPage).
├── services/        # Contient la logique pour interagir avec l'API backend,
chaque fichier représentant un service pour une ressource spécifique (e.g.,
clientService, missionService).
├── test/            # Fichiers de test pour les composants, hooks, et autres
logiques (e.g., tests unitaires, tests d'intégration).
├── types/           # Définitions de types TypeScript pour les modèles de
données, les interfaces, et les props des composants.
├── utils/           # Fonctions utilitaires génériques et réutilisables qui ne
sont pas spécifiques à un composant ou un service (e.g., fonctions de
validation, de formatage).
├── App.tsx          # Le composant racine de l'application, où le routage et
la disposition globale sont définis.
├── main.tsx         # Le point d'entrée de l'application, où l'application
React est rendue dans le DOM.
├── index.css        # Fichier CSS global pour les styles de base ou les
importations de Tailwind CSS.
└── vite-env.d.ts    # Fichier de déclaration de types pour Vite.
```

Cette architecture favorise une approche basée sur les composants, où chaque partie de l'interface utilisateur est un composant indépendant et réutilisable. La séparation des préoccupations est clairement définie :

- **Composants (`components/`)** : Sont responsables de la présentation et de l'interaction avec l'utilisateur. Ils sont conçus pour être génériques et réutilisables à travers différentes pages.
- **Pages (`pages/`)** : Agissent comme des conteneurs pour les composants, orchestrant leur agencement pour former des vues complètes. Elles gèrent la logique spécifique à la page et l'intégration des données.
- **Services (`services/`)** : Encapsulent la logique de communication avec l'API backend. Chaque service est dédié à une ressource spécifique (par exemple, `clientService.ts` gère toutes les opérations CRUD pour les clients), ce qui rend la gestion des données et les appels API centralisés et faciles à maintenir.
- **Hooks (`hooks/`)** : Permettent de réutiliser la logique d'état et les comportements entre les composants sans avoir à les dupliquer. Ils améliorent la

lisibilité et la modularité du code.

- **Types (`types/`)** : Assurent la cohérence des données et la robustesse du code grâce à la typisation forte de TypeScript, réduisant les erreurs liées aux types de données.

Cette structure modulaire et bien organisée facilite le développement en équipe, la maintenance du code et l'ajout de nouvelles fonctionnalités, tout en garantissant une application performante et scalable.

# 4. Fonctionnalités Clés

L'application frontend de Progitek System est une interface utilisateur complète et interactive, conçue pour permettre aux utilisateurs de gérer efficacement les opérations techniques et administratives. Elle interagit directement avec l'API backend pour récupérer, afficher et manipuler les données. Les principales fonctionnalités, déduites de la structure des pages et des services, incluent :

## 4.1. Authentification et Gestion des Utilisateurs

- **Page de Connexion (`LoginPage.tsx`)** : Permet aux utilisateurs de s'authentifier auprès du système. Elle gère la soumission des identifiants et la réception des tokens d'authentification (access et refresh tokens) du backend. Le hook `useAuth` est probablement utilisé pour gérer l'état d'authentification de l'utilisateur à travers l'application.
- **Gestion des Utilisateurs (`UsersPage.tsx`)** : Offre une interface pour la création, la lecture, la mise à jour et la suppression des comptes utilisateurs. Cela inclut potentiellement la gestion des rôles et des permissions, permettant aux administrateurs de contrôler l'accès aux différentes parties du système.

## 4.2. Tableau de Bord et Rapports

- **Page de Tableau de Bord (`DashboardPage.tsx`)** : Présente une vue d'ensemble des indicateurs clés de performance (KPIs) et des statistiques importantes de l'entreprise. Elle utilise `recharts` pour afficher des graphiques et des visualisations de données, offrant un aperçu rapide de l'état des missions, des

finances, etc. Le hook `useDashboard` et le `dashboardService` sont responsables de la récupération et de la mise en forme de ces données.

- **Page de Rapports (`RapportsPage.tsx`)** : Permet de générer et de consulter divers rapports détaillés sur les activités, les performances des techniciens, les statistiques de missions, et d'autres données opérationnelles. Le `rapportService` gère la communication avec le backend pour obtenir les données nécessaires à ces rapports.

### 4.3. Gestion des Missions et Interventions

- **Page des Missions (`MissionsPage.tsx`)** : Affiche la liste de toutes les missions et permet aux utilisateurs de consulter les détails, de créer de nouvelles missions, de les modifier ou de les supprimer. Le `missionService` est l'interface avec l'API backend pour toutes les opérations liées aux missions.
- **Page des Interventions (`InterventionsPage.tsx`)** : Gère les interventions, qui sont des étapes ou des tâches spécifiques au sein des missions. Cette page permet de suivre l'avancement des interventions, d'assigner des techniciens et de mettre à jour leur statut. Le `interventionService` facilite ces interactions.

### 4.4. Gestion des Clients, Techniciens et Spécialités

- **Page des Clients (`ClientsPage.tsx`)** : Fournit une interface complète pour la gestion des informations clients, y compris la création de nouveaux clients, la modification des données existantes et la consultation des détails. Le `clientService` est dédié à la gestion des données clients.
- **Page des Techniciens (`TechniciensPage.tsx`)** : Permet de gérer les informations relatives aux techniciens, y compris leurs coordonnées, leurs spécialités et leurs disponibilités. Le `technicienService` assure la communication avec le backend pour ces opérations.
- **Page des Spécialités (`SpecialitesPage.tsx`)** : Gère les différentes spécialités techniques, permettant de les ajouter, de les modifier ou de les supprimer. Le `specialiteService` est utilisé pour interagir avec l'API pour cette ressource.

### 4.5. Gestion Financière (Devis, Factures, Paiements)

- **Page des Devis (`DevisPage.tsx`)** : Permet de créer, consulter, modifier et gérer les devis. Cette page intègre probablement des fonctionnalités de calcul et de validation pour les postes de devis. Le `devisService` est responsable de la gestion des données de devis.
- **Page des Factures (`FacturesPage.tsx`)** : Offre une interface pour la gestion des factures, y compris la génération à partir de devis ou d'interventions, le suivi des paiements et la gestion des statuts. Le `factureService` gère les interactions avec l'API pour les factures.
- **Page des Types de Paiement (`TypesPaiementPage.tsx`)** : Permet de configurer et de gérer les différents types de paiement acceptés par le système. Le `typePaiementService` est utilisé pour cette fonctionnalité.

### 4.6. Gestion du Stock

- **Page du Stock (`StockPage.tsx`)** : Fournit une interface pour la gestion de l'inventaire, permettant de suivre les articles en stock, d'enregistrer les entrées et les sorties, et de gérer les niveaux de stock. Le `stockService` est responsable de la communication avec le backend pour les opérations de stock.

### 4.7. Communication et Notifications

- **Page des Messages (`MessagesPage.tsx`)** : Permet aux utilisateurs d'échanger des messages au sein de l'application, facilitant la communication interne. Le `messageService` gère l'envoi et la réception des messages.
- **Page des Notifications (`NotificationsPage.tsx`)** : Affiche les notifications importantes pour l'utilisateur, telles que les nouvelles missions, les mises à jour de statut ou les alertes système. Le `notificationService` est utilisé pour récupérer et afficher ces notifications.

### 4.8. Paramètres et Configuration

- **Page des Paramètres (`SettingsPage.tsx`)** : Permet aux utilisateurs de configurer divers paramètres de l'application, tels que les préférences

d'affichage, les notifications, ou d'autres options personnalisables. Le hook `useSettings` et le `settingsService` gèrent la persistance de ces paramètres.

- **Page des Paramètres Avancés (`AdvancedSettingsPage.tsx`)** : Propose des options de configuration plus complexes, potentiellement réservées aux administrateurs ou aux utilisateurs avancés.

L'ensemble de ces fonctionnalités offre une solution complète pour la gestion des opérations techniques et administratives, avec une interface utilisateur conçue pour l'efficacité et la facilité d'utilisation.

# 5. Dépendances et Intégrations

Le projet frontend de Progitek System s'appuie sur un ensemble de dépendances et d'intégrations pour construire une interface utilisateur riche et interactive, et pour communiquer efficacement avec le backend. Ces éléments sont gérés via `pnpm` et sont définis dans le fichier `package.json`.

## 5.1. Gestion des Dépendances

Les dépendances du projet sont classées en deux catégories principales :

- **Dépendances de Production (`dependencies`)** : Ces bibliothèques sont essentielles au fonctionnement de l'application en production. Elles incluent :
  - **`axios`** : Pour les requêtes HTTP vers l'API backend.
  - **`clsx`** : Pour la construction conditionnelle de chaînes de noms de classes CSS.
  - **`date-fns`** : Pour la manipulation et le formatage des dates.
  - **`lucide-react`** : Pour l'intégration d'icônes vectorielles.
  - **`react`** et **`react-dom`** : Les bibliothèques fondamentales de React pour la construction de l'interface utilisateur.
  - **`react-hot-toast`** : Pour l'affichage de notifications toast.
  - **`react-router-dom`** : Pour la gestion du routage côté client.
  - **`recharts`** : Pour la création de graphiques et de visualisations de données.

- **Dépendances de Développement (`devDependencies`)** : Ces bibliothèques sont utilisées pendant le développement, le test et le build de l'application. Elles comprennent :
    - `@types/*` : Définitions de types TypeScript pour les bibliothèques JavaScript.
    - `@vitejs/plugin-react` : Plugin Vite pour le support de React.
    - `autoprefixer` et `postcss` : Outils pour le traitement CSS, notamment pour Tailwind CSS.
    - `eslint` et `typescript-eslint` : Outils de linting pour maintenir la qualité et la cohérence du code.
    - `jest-dom`, `react-testing-library`, `user-event` : Bibliothèques pour les tests unitaires et d'intégration des composants React.
    - `jsdom` : Un environnement DOM simulé pour les tests.
    - `tailwindcss` : Le framework CSS utilitaire-first.
    - `typescript` : Le compilateur TypeScript.
    - `vite` : L'outil de build et serveur de développement.
    - `vitest` : Le framework de test rapide et moderne.

## 5.2. Intégrations Clés

Le frontend est conçu pour s'intégrer principalement avec :

- **API Backend de Progitek System** : L'intégration la plus critique se fait via `axios` et les services définis dans `src/services/`. Chaque service (e.g., `clientService.ts`, `missionService.ts`) est responsable de la communication avec les points d'accès correspondants de l'API backend, assurant la récupération et la soumission des données.
- **Système de Routage (React Router DOM)** : Le routage est intégré pour gérer la navigation au sein de l'application, permettant une expérience utilisateur fluide sans rechargement complet de la page lors du changement de vue.
- **Gestion des Styles (Tailwind CSS)** : Tailwind CSS est intégré via PostCSS pour compiler et optimiser les styles CSS. Cette approche permet une

personnalisation poussée de l'interface utilisateur tout en maintenant une taille de fichier CSS minimale.

- **Visualisation de Données (Recharts)** : Recharts est intégré pour la création de graphiques interactifs, utilisés notamment dans le tableau de bord pour présenter les données de manière visuelle et compréhensible.
- **Notifications (React Hot Toast)** : L'intégration de `react-hot-toast` permet d'afficher des messages de feedback non bloquants à l'utilisateur, améliorant l'expérience en fournissant des informations en temps réel sur le succès ou l'échec des opérations.

Ces dépendances et intégrations sont choisies pour leur efficacité, leur performance et leur capacité à créer une application frontend moderne, réactive et facile à développer.

# 6. Déploiement et Tests

Le processus de déploiement et la stratégie de test sont des aspects cruciaux pour garantir la qualité, la performance et la disponibilité de l'application frontend de Progitek System. Le projet est configuré pour faciliter ces étapes, en s'appuyant sur des outils modernes et des pratiques de développement établies.

## 6.1. Déploiement

Le déploiement de l'application frontend est géré par Vite, qui est un outil de build optimisé pour les applications web modernes. Le processus de déploiement implique les étapes suivantes :

1. **Build de l'application** : Avant le déploiement en production, le code source de l'application (TypeScript, React, CSS) doit être compilé et optimisé pour la production. Cette étape est réalisée via la commande `pnpm run build`, qui exécute `vite build`. Cette commande génère un ensemble de fichiers statiques (HTML, CSS, JavaScript, assets) dans le répertoire `dist/`.
2. **Servir les fichiers statiques** : Le répertoire `dist/` contient l'application frontend prête à être servie par n'importe quel serveur web statique (Nginx, Apache, ou des services d'hébergement comme Netlify, Vercel, GitHub Pages, etc.). L'application est une Single Page Application (SPA), ce qui signifie que le

fichier `index.html` est le point d'entrée unique et que le routage est géré côté client par `react-router-dom`.

3. **Configuration de l'Environnement** : Les variables d'environnement spécifiques à la production (par exemple, l'URL de l'API backend) sont gérées via des fichiers `.env` ou des configurations spécifiques à l'environnement de déploiement. Vite supporte l'injection de ces variables au moment du build.

Le projet est conçu pour être facilement déployable sur des plateformes d'hébergement statique ou des serveurs web, offrant une grande flexibilité et des coûts d'infrastructure réduits.

## 6.2. Tests

Une stratégie de test complète est mise en place pour assurer la fiabilité et la robustesse de l'application frontend. Le projet utilise Vitest comme framework de test principal, complété par Testing Library pour les tests de composants et d'intégration.

- **Tests Unitaires et d'Intégration (Vitest)** : La commande `pnpm test` exécute la suite de tests. Vitest est un framework de test rapide et compatible avec Vite, offrant une expérience de développement fluide avec des fonctionnalités comme le Hot Module Replacement (HMR) pour les tests. Les tests sont situés dans le répertoire `src/test/` et couvrent des aspects clés de l'application, tels que :
  - **Composants ( `src/test/components/` )** : Des tests sont écrits pour des composants individuels (par exemple, `StatCard.test.tsx`), vérifiant leur rendu, leur comportement et leur interaction avec les utilisateurs.
  - **Hooks ( `src/test/hooks/` )** : Des tests sont dédiés aux hooks personnalisés (par exemple, `useAuth.test.tsx`), s'assurant que la logique d'état et les effets secondaires encapsulés fonctionnent correctement.

- **Testing Library ( `@testing-library/react` , `@testing-library/jest-dom` , `@testing-library/user-event` )** : Ces bibliothèques sont utilisées en conjonction avec Vitest pour écrire des tests qui se concentrent sur la manière dont les utilisateurs interagissent avec l'application. L'objectif est de tester l'application du point de vue de l'utilisateur, en se basant sur le DOM rendu plutôt que sur les détails d'implémentation des composants. Cela garantit que les tests sont plus robustes aux changements internes du code.

- **Couverture de Code (`vitest --coverage`)** : La commande `pnpm test:coverage` permet de générer un rapport de couverture de code, indiquant quelles parties du code sont couvertes par les tests et quelles parties nécessitent une attention supplémentaire. Cela aide à identifier les lacunes dans la suite de tests.
- **Linting (`eslint`)** : La commande `pnpm lint` (et `pnpm lint:fix`) est utilisée pour analyser statiquement le code et identifier les problèmes de style, les erreurs potentielles et les mauvaises pratiques. Cela contribue à maintenir une qualité de code élevée et une cohérence à travers le projet.

Cette approche de test, combinant des tests unitaires, des tests d'intégration et des outils de qualité de code, assure que l'application frontend est fiable, performante et exempte de bugs majeurs avant le déploiement en production.