Rapport Détaillé du Développement de l'Application de Gestion des Remboursements

Date de début de la collaboration : 3 juin 2025 **Date de ce rapport :** 5 juin 2025 **Objectif du projet :** Développer une application de bureau pour organiser et suivre les demandes de remboursement pour l'Hôpital Privé Natécia, avec un accès restreint et des fonctionnalités spécifiques pour différents utilisateurs.

Phase 1 : Cadrage Initial du Projet et Exploration des Solutions (3 juin 2025)

1. **Expression du Besoin Initial :**
    - L'utilisateur, stagiaire à l'Hôpital Privé Natécia, a exprimé le besoin d'organiser les demandes de remboursement.
    - Un dossier partagé (`\\192.168.197.43\Commun\REMBOURSEMENT`) était déjà créé et accessible par 7 personnes (6 employés + le stagiaire).
    - Le processus de remboursement débute lorsque Philippe (p.neri@noalys.com) reçoit un email d'un client.
    - Fonctionnalités souhaitées : un "petit logiciel" avec une interface esthétique et pratique pour saisir les informations du client et de la demande, enregistrer la demande, ajouter le client à une liste visuelle des demandes en cours avec leur statut, permettre à plusieurs personnes d'accéder et d'enregistrer des modifications avec mise à jour automatique, et stocker des fichiers (PDF pour factures/RIB, images pour preuves de prélèvement).
    - Demande initiale : ne pas fournir de code, mais une solution et des recommandations technologiques.
2. **Première Proposition de Solution et Technologies :**
    - Solution suggérée : une application web interne.
    - Technologies envisagées :
        - Backend : Python avec Flask ou Node.js avec Express.js.
        - Frontend : HTML, CSS, JavaScript (potentiellement avec un framework comme Bootstrap ou Vue.js).
        - Base de données : PostgreSQL ou MySQL.
        - Stockage des fichiers : dans le dossier partagé existant.
3. **Clarification des Contraintes et Ajustement des Solutions :**
    - **Contrainte majeure de l'utilisateur :** Pas d'accès au réseau de l'hôpital pour héberger un serveur web. L'application doit fonctionner **uniquement depuis et à l'intérieur** du dossier partagé `\\192.168.197.43\Commun\REMBOURSEMENT`. L'accès doit être strictement limité aux 7 personnes autorisées, soulevant des préoccupations de sécurité pour les données sensibles (RIB).
    - **Solution Ajustée 1 (Web Local Portable) :** Proposition d'une application web locale où un des 7 utilisateurs lancerait un mini-serveur (Flask/Node.js) depuis son poste, avec les données (base SQLite) et l'application elle-même

résidant dans le dossier partagé. Les autres y accéderaient via l'IP du poste hôte.
        - Inconvénients : dépendance au poste hôte, configuration potentielle du pare-feu.
    - **Solution Ajustée 2 (Application de Bureau) :** Proposition d'une application de bureau classique.
        - Technologies : Python avec PyQt/Tkinter, ou Electron (HTML/CSS/JS).
        - Stockage des données : Fichiers JSON/CSV individuels ou une base Microsoft Access (`.accdb`) dans le dossier partagé.
        - Gestion de la concurrence : point délicat pour JSON/CSV, mieux géré par Access pour de petits groupes.
    - **Préférence de l'utilisateur :** "Je ne veux pas utiliser Microsoft Access mais bien une application codée par moi." Rejet de la solution web jugée trop complexe pour le contexte. Confirmation que l'application doit être dédiée et lançable depuis le dossier partagé.

Phase 2 : Choix Technologique pour une Application de Bureau et Sécurité (3-4 juin 2025)

1. **Exploration d'une "Interface Web Locale" (limitations) :**
    - L'utilisateur a évoqué l'idée d'un simple site web local (fichiers HTML/CSS/JS) dans le dossier partagé.
    - Explication des limitations de sécurité des navigateurs (`file:///`) empêchant l'écriture de fichiers, la création de dossiers et l'accès direct à une base de données partagée de manière fiable pour plusieurs utilisateurs.
2. **Orientation vers Python et Electron :**
    - Proposition d'Electron comme solution pour une application de bureau avec une interface utilisateur basée sur les technologies web.
    - **Décision finale de l'utilisateur : développer l'application en Python.**
3. **Comparaison Python GUI vs. Electron :**
    - Critères : optimisation, ergonomie (développement/utilisation), esthétique.
    - **Electron :** Potentiel esthétique illimité (HTML/CSS), mais plus lourd.
    - **Python GUI :**
        - CustomTkinter : Léger, moderne, bonne option.
        - PyQt/PySide : Très puissant, courbe d'apprentissage plus élevée.
        - Tkinter : Léger, basique, apparence datée sans effort.
    - **Préférence utilisateur implicite pour CustomTkinter** via l'acceptation des propositions de code ultérieures.
4. **Gestion des Mots de Passe :**
    - Préoccupation de l'utilisateur : confidentialité des mots de passe dans le dossier partagé.
    - Solution proposée et adoptée : **Hachage des mots de passe** à l'aide de la bibliothèque `passlib` avec l'algorithme `bcrypt` (ou `argon2`). Les mots de passe en clair ne sont jamais stockés.
5. **Distribution de l'Application :**

- Clarification que les utilisateurs finaux n'auront pas besoin d'installer Python ou les bibliothèques.
- Solution : Utilisation de **PyInstaller** pour packager l'application Python en un exécutable autonome.

Phase 3 : Développement Initial - Système d'Authentification (4 juin 2025)

1. **Structure Initiale du Projet (simple) :**
   - Développement en local dans `C:\Users\maxen\PycharmProjects\PythonProject`.
   - Création d'un dossier `donnees_partagees_mock` pour simuler le dossier réseau.
   - Fichiers initiaux :
     - `auth_utils.py` : Fonctions pour hacher/vérifier les mots de passe, charger/sauvegarder les données utilisateurs (initialement juste le hachage) dans `utilisateurs.json`.
     - `setup_users.py` : Script pour créer/mettre à jour les comptes utilisateurs.
     - `main_app.py` : Application principale avec l'interface de connexion utilisant CustomTkinter.
2. **Débogage des Problèmes d'Installation et de Compatibilité :**
   - **Erreur 1 : `No module named 'customtkinter'`**
     - Cause : L'interpréteur Python utilisé par PyCharm pour le projet n'avait pas la bibliothèque installée.
     - Solution : Installation de `customtkinter` dans l'environnement virtuel (`.venv`) du projet PyCharm.
   - **Erreur 2 : `(trapped) error reading bcrypt version` / `AttributeError: module 'bcrypt' has no attribute '__about__'`**
     - Cause : Incompatibilité mineure entre `passlib 1.7.4` et la version très récente de `bcrypt (4.3.0)` concernant la lecture des métadonnées de version.
     - Solution : Désinstallation de `bcrypt 4.3.0` et installation d'une version compatible, `bcrypt==4.0.1`.

Phase 4 : Amélioration du Système d'Authentification (4-5 juin 2025)

1. **Nouvelles Fonctionnalités Demandées :**
   - Modification du mot de passe par l'utilisateur (en connaissant l'ancien).
   - Procédure de mot de passe oublié avec envoi d'un code à 5 chiffres par email.
2. **Mise à Jour de la Structure des Données Utilisateur :**
   - Le fichier `utilisateurs.json` a été modifié pour stocker, pour chaque utilisateur, un dictionnaire contenant `"hashed_password"` et `"email"`.

3. **Mises à Jour des Fichiers :**
    - `auth_utils.py` (ancêtre de `user_model.py` et `password_utils.py`) :
        - Ajout de fonctions pour gérer la nouvelle structure de données.
        - Fonctions pour modifier le mot de passe.
        - Fonctions pour générer, stocker (dans `codes_reset.json`), et vérifier les codes de réinitialisation.
        - Simulation initiale de l'envoi d'email (affichage du code en console/messagebox).
    - `setup_users.py` : Modifié pour demander et stocker l'adresse email de chaque utilisateur.
    - `main_app.py` (ancêtre de `login_view.py` et `app.py`) :
        - Ajout de boutons "Modifier mon mot de passe" et "Mot de passe oublié ?" sur l'écran de connexion.
        - Création de fenêtres de dialogue (`CTkToplevel`) pour ces nouvelles fonctionnalités.

Phase 5 : Passage à l'Envoi Réel d'Emails et Restructuration MVC (5 juin 2025)

1. **Demandes de l'Utilisateur :**
    - Implémenter l'envoi réel d'emails (plutôt que la simulation).
    - Adopter une structure de projet plus organisée (type MVC).
    - Ajouter un utilisateur "admin.stagiaire" avec l'email `info.stagiaire.noalys@gmail.com` pour les tests.
2. **Adoption de l'Architecture MVC :**
    - La structure du projet `C:\Users\maxen\PycharmProjects\PythonProject\` a été réorganisée :
        - `config/`: `settings.py` (chemins, chargement config SMTP), `config_email.ini` (fichier de configuration SMTP, avec un `config_email_template.ini` comme modèle).
        - `controllers/`: `auth_controller.py` (logique d'authentification, reset, modification mdp), `app_controller.py` (orchestration des vues), `remboursement_controller.py` (pour la gestion des remboursements).
        - `models/`: `user_model.py` (interaction avec `utilisateurs.json`, `codes_reset.json`), `remboursement_model.py` (gestion des données de remboursement).
        - `views/`: `login_view.py` (interfaces graphiques liées à l'authentification), `main_view.py` (interface principale après connexion).
        - `utils/`: `password_utils.py` (hachage), `email_utils.py` (envoi d'emails via SMTP), `pdf_utils.py` (extraction PDF).
        - `donnees_partagees_mock/`: Dossier local pour les fichiers JSON et les pièces jointes pendant le développement.

        - Racine : `app.py` (point d'entrée), `setup_users.py`.
3. **Implémentation de l'Envoi d'Emails :**
    - `utils/email_utils.py` : Fonction `envoyer_email_reset` utilisant `smtplib` pour se connecter à un serveur SMTP (configuré dans `config_email.ini`) et envoyer un email HTML/texte.
    - `config/settings.py` : Logique pour charger la configuration SMTP depuis `config_email.ini`.
    - L'utilisateur a configuré `config_email.ini` avec ses identifiants Gmail (utilisant un mot de passe d'application).
4. **Débogage de la Structure des Données :**
    - **Erreur 1 : `AttributeError: 'str' object has no attribute 'get'`** lors de la connexion de `p.neri`.
        - Cause : Le fichier `utilisateurs.json` contenait encore des entrées pour `p.neri` (et d'autres) sous l'ancien format (chaîne de hachage directe) au lieu du nouveau format (dictionnaire avec `hashed_password` et `email`).
        - Solution : Suppression et régénération de `utilisateurs.json` en utilisant le script `setup_users.py` mis à jour.
    - **Erreur 2 : `TypeError: string indices must be integers, not 'str'`** dans `setup_users.py` lors de la tentative de mise à jour d'un utilisateur existant.
        - Cause : Similaire à la précédente ; le script tentait de lire `utilisateur_existant['email']` alors que `utilisateur_existant` était une chaîne pour une entrée mal formatée.
        - Solution : Assurer une initialisation propre de `utilisateurs.json`.

Phase 6 : Améliorations de l'Interface Utilisateur et Début de la Gestion des Remboursements (5 juin 2025)

1. **Améliorations de l'Interface Utilisateur (GUI) :**
    - Demande : Fenêtre principale maximisée par défaut, mais réductible. Fenêtres de dialogue pour la gestion des mots de passe agrandies.
    - `app.py` : Modification pour maximiser la fenêtre au démarrage (`self.state('zoomed')` et alternatives, avec un `self.after(100, ...)` pour améliorer la stabilité sur certains systèmes). Suppression des redimensionnements fixes de la fenêtre principale. Taille minimale définie.
    - `controllers/app_controller.py` : Suppression des appels `self.root.geometry()` pour permettre à la fenêtre principale de conserver son état maximisé.
    - `views/login_view.py` : Augmentation des dimensions des fenêtres `CTkToplevel` pour la modification et l'oubli de mot de passe. Le contenu du formulaire de connexion a été placé dans un frame centré pour un meilleur aspect sur un grand écran.

2. **Début de l'Implémentation de la Création des Demandes de Remboursement :**
    - Fonctionnalité cible : Permettre à `p.neri` de créer une demande de remboursement.
    - Champs requis : Nom, Prénom, Référence Facture, Montant demandé.
    - Pièces jointes : Facture (PDF), RIB (PDF).
    - Fonctionnalité d'assistance : Extraction automatique du Nom, Prénom, et Référence depuis le PDF de la facture.
    - Exemple de facture fourni (`Fact_2504868_20250520_162526315.pdf`) avec les informations cibles : DELFOSSE LOUISE, Référence: 25 4868.
3. **Nouvelles Implémentations et Mises à Jour :**
    - **Bibliothèque ajoutée :** `pdfplumber` pour l'extraction de texte des PDF.
    - `utils/pdf_utils.py` : Création de la fonction `extraire_infos_facture` avec des expressions régulières pour tenter d'extraire Nom, Prénom et Référence.
    - `models/remboursement_model.py` :
        - Création des fonctions `_charger_remboursements`, `_sauvegarder_remboursements` (pour `remboursements.json`).
        - Création de la fonction `creer_nouvelle_demande` qui :
            - Génère un ID unique pour la demande.
            - Gère la copie des fichiers PDF (facture, RIB) vers un dossier de stockage structuré.
            - Enregistre les informations de la demande (y compris les chemins relatifs des fichiers) dans `remboursements.json`.
            - Initialise le statut de la demande et un historique.
    - `controllers/remboursement_controller.py` :
        - Création de la classe `RemboursementController`.
        - Méthode pour appeler `pdf_utils.extraire_infos_facture`.
        - Méthode `creer_demande_remboursement` pour valider les entrées du formulaire et appeler le modèle.
        - Méthode `selectionner_fichier_pdf` utilisant `tkinter.filedialog`.
    - `views/main_view.py` :
        - Ajout d'un bouton "Nouvelle Demande de Remboursement" (visible uniquement pour `p.neri`).
        - Création de la méthode `_ouvrir_fenetre_creation_demande` qui affiche un `CTkToplevel` avec un formulaire pour saisir les informations de la demande et sélectionner les fichiers.
        - Logique pour appeler le `remboursement_controller` lors de la soumission du formulaire.
        - Appel à l'extraction PDF lors de la sélection de la facture pour pré-remplir les champs.
    - `controllers/app_controller.py` : Ajout d'une factory `_remboursement_controller_factory` pour injecter le `RemboursementController` (avec l'utilisateur actuel) dans `MainView`.

4. **Affinement de l'Organisation des Dossiers et de l'Extraction PDF :**
    - Demande de l'utilisateur : Toutes les données de remboursement dans un dossier principal `C:\Users\maxen\PycharmProjects\PythonProject\Demande_Remboursement`. À l'intérieur, un sous-dossier par demande, nommé d'après la référence (ex: `Demande_254868`), contenant les pièces jointes.
    - `config/settings.py` : Chemins mis à jour :
        - `DOSSIER_PRINCIPAL_REMBOURSEMENT` pointe vers le nouveau dossier.
        - `DATA_DIR` (pour `utilisateurs.json`, `codes_reset.json`, `remboursements.json`) devient un sous-dossier `_app_data` dans `DOSSIER_PRINCIPAL_REMBOURSEMENT`.
        - `PIECES_JOINTES_BASE_DIR` pointe vers `DOSSIER_PRINCIPAL_REMBOURSEMENT\Demandes_Clients\`.
    - `models/remboursement_model.py` :
        - La fonction `creer_nouvelle_demande` utilise maintenant `PIECES_JOINTES_BASE_DIR` et crée des sous-dossiers nommés `Demande_{reference_nettoyee}`.
        - L'`id_demande` interne reste unique (combinaison de la référence et d'un timestamp/uuid court).
        - Les chemins des fichiers stockés dans `remboursements.json` sont relatifs à `DATA_DIR` (ce point sera à vérifier/corriger si les pièces jointes ne sont plus dans `DATA_DIR` mais dans `PIECES_JOINTES_BASE_DIR`).
    - `utils/pdf_utils.py` : Amélioration des expressions régulières pour une extraction plus précise du Nom, Prénom et Référence basée sur l'analyse détaillée du PDF exemple fourni.

État Actuel du Projet (au 5 juin 2025, fin de session)

- L'application dispose d'un système d'authentification fonctionnel avec :
    - Connexion utilisateur.
    - Modification de mot de passe par l'utilisateur.
    - Procédure de mot de passe oublié avec envoi réel d'un code de réinitialisation par email.
- La structure du projet est organisée selon le modèle MVC.
- L'interface utilisateur est construite avec CustomTkinter et démarre en mode fenêtré maximisé.
- La première étape du processus de remboursement (création d'une demande par `p.neri`) est implémentée :
    - Un formulaire permet de saisir les informations (Nom, Prénom, Référence, Montant).
    - Les fichiers Facture et RIB (PDF) peuvent être sélectionnés.
    - Une tentative d'extraction automatique des informations (Nom, Prénom, Référence) depuis le PDF de la facture est en place.

- Les informations de la demande sont sauvegardées dans `remboursements.json`.
- Les pièces jointes (Facture, RIB) sont copiées dans un dossier structuré (`Demande_Remboursement/Demandes_Clients/Demande_{Référence}/`).

Ce résumé couvre l'ensemble des étapes de développement, des discussions initiales aux implémentations et corrections effectuées jusqu'à présent.

Rapport d'Analyse et de Développement de l'Application "GestionRemboursements"

Introduction et Objectif Initial

Le projet a débuté avec une demande d'analyse complète d'une application existante développée en Python avec la bibliothèque CustomTkinter. L'objectif principal était d'optimiser l'application pour la rendre plus **fluide, rapide et robuste**. Cette analyse devait se faire dans le contexte d'un déploiement futur : l'application serait compilée en un exécutable via PyInstaller et utilisée par six personnes sur un réseau partagé.

Phase 1 : Analyse Initiale et Optimisations Stratégiques

La première phase s'est concentrée sur l'analyse de l'architecture existante et l'identification des points d'amélioration potentiels, notamment en ce qui concerne le déploiement réseau.

1. **Analyse de la Commande PyInstaller :**
    - La commande de compilation a été examinée. Il a été noté que des dossiers de données (`config`, `donnees_partagees_mock`) étaient inclus via l'option `--add-data`.
    - Un point critique a été soulevé : l'inclusion du dossier `donnees_partagees_mock` dans le bundle de l'exécutable entrerait en conflit direct avec le besoin de partager et de modifier des données (comme `remboursements.json`) sur un réseau. Il a été recommandé de modifier le code pour qu'il puisse utiliser des chemins réseau en production, distincts des chemins locaux utilisés pour le développement.
2. **Gestion de la Concurrence des Données :**
    - Le principal risque identifié pour une utilisation multi-utilisateurs était l'utilisation de fichiers JSON comme base de données sur un partage réseau. Sans mécanisme de protection, si deux utilisateurs modifiaient un fichier (ex: `remboursements.json`) simultanément, les modifications de l'un pourraient écraser celles de l'autre, menant à une perte de données.
    - La solution proposée a été la mise en place d'**écritures atomiques**. Cette technique consiste à écrire les nouvelles données dans un fichier temporaire, puis à renommer ce fichier temporaire pour remplacer l'original. Le

renommage étant une opération généralement instantanée (atomique) sur la plupart des systèmes de fichiers, cela réduit drastiquement le risque de corruption en cas d'accès concurrents ou de plantage pendant l'écriture.

3. **Nettoyage et Amélioration de la Structure :**
    - Le fichier `auth_utils.py` a été identifié comme étant redondant et probablement obsolète, ses fonctionnalités étant déjà couvertes de manière plus intégrée par `models/user_model.py`. Sa suppression a été recommandée pour clarifier la base de code.
    - L'utilisation de `print()` pour le débogage a été notée. Il a été suggéré de remplacer ces appels par le module `logging` de Python, qui offre une gestion plus flexible et professionnelle des messages (niveaux de criticité, sortie vers des fichiers, formatage, etc.).

Phase 2 : Implémentation des Premières Corrections et Stabilité

Cette phase a consisté à mettre en œuvre les recommandations de l'analyse initiale pour stabiliser l'application.

1. **Mise en place des Écritures Atomiques :**
    - Une nouvelle fonction `_sauvegarder_fichier_json_atomic` a été développée.
    - Elle a été intégrée dans les fichiers `models/user_model.py` et `models/remboursement_data.py` pour remplacer les appels de sauvegarde standard, protégeant ainsi les fichiers `utilisateurs.json`, `codes_reset.json` et `remboursements.json`.
2. **Correction d'une Erreur d'Attribut (`AttributeError`) :**
    - Une erreur `AttributeError: 'MainView' object has no attribute '_action_voir_historique_docs'` a été signalée lors de la création d'une nouvelle demande.
    - L'analyse a révélé qu'un dictionnaire de callbacks dans `views/main_view.py` faisait référence à une méthode qui n'était pas définie dans la classe `MainView`.
    - La correction a consisté à ajouter la méthode manquante `_action_voir_historique_docs` à la classe `MainView`.

Phase 3 : Tentative d'Implémentation d'une Nouvelle Fonctionnalité (Glisser-Déposer)

Une demande a été faite pour ajouter la fonctionnalité de glisser-déposer (drag and drop) des fichiers dans toutes les fenêtres où une sélection de document est nécessaire. Cette étape s'est avérée être la plus complexe et a révélé des problèmes d'intégration profonds.

1. **Choix de la Technologie :**

    - La bibliothèque `tkinterdnd2` a été choisie, car c'est l'outil standard pour implémenter le glisser-déposer de fichiers avec Tkinter.
2. **Modifications Initiales :**
    - Un nouveau fichier utilitaire, `utils/dnd_utils.py`, a été créé pour héberger une fonction de "nettoyage" des chemins de fichiers reçus via le glisser-déposer.
    - Le fichier `views/main_view.py` a été lourdement modifié pour inclure des zones de dépôt visuelles et les callbacks associés. Une méthode `_creer_zone_selection_fichier_avec_dnd` a été créée pour factoriser ce code.

Phase 4 : Débogage Intensif des Erreurs d'Intégration

L'intégration de `tkinterdnd2` avec CustomTkinter a généré une série d'erreurs complexes.

1. **Problème d'Importation (`ImportError`) et de Reconnaissance :**
    - La première erreur signalée était une `ImportError: cannot import name 'envoyer_email_reinitialisation' from 'utils.email_utils'`. La cause principale identifiée était l'absence d'un fichier `__init__.py` dans le dossier `utils`, empêchant Python de le traiter comme un package. Une incohérence dans le nom de la fonction appelée depuis `auth_controller.py` a également été corrigée.
    - Par la suite, des avertissements persistants `WARNING:root:tkinterdnd2 non trouvé` sont apparus. Cela indiquait que, même si la bibliothèque était installée, l'environnement d'exécution Python ne parvenait pas à l'importer correctement, suggérant un problème d'environnement, d'installation ou de cache.
2. **Erreur Tcl `invalid command name "tkdnd::drop_target"` et Fenêtre "tk" Vide :**
    - C'est l'erreur la plus critique rencontrée. Elle signifie que les commandes Tcl (le langage sous-jacent de Tkinter) spécifiques à `tkinterdnd2` n'étaient pas chargées.
    - Ce problème est un symptôme d'un conflit d'initialisation entre `ctk.CTk()` et `TkinterDnD.Tk()`, qui se battent pour être la fenêtre racine de l'application. La tentative d'utiliser l'héritage multiple a souvent conduit à la création d'une fenêtre "tk" vide et blanche, signe de ce conflit.
3. **Erreur de Layout (`TclError: cannot use geometry manager...`) :**
    - Parallèlement, une erreur de géométrie est apparue, causée par un mélange des gestionnaires `.pack()` et `.grid()` pour les enfants d'un même widget parent dans les fenêtres de dialogue.
    - La solution a consisté à restructurer rigoureusement le layout de ces fenêtres dans `main_view.py`, en utilisant un conteneur principal (`CTkScrollableFrame` ou `CTkFrame`) qui est packé dans le dialogue, et en utilisant ensuite un seul gestionnaire de géométrie (`pack` ou `grid`) pour tous les éléments à l'intérieur de ce conteneur.

Phase 5 : Refonte Architecturale pour la Stabilité

Face aux problèmes d'initialisation persistants, une refonte de l'architecture de démarrage de l'application a été mise en œuvre.

1. **Modification de `app.py` :**
    - La classe `MainApplication` a été transformée en une classe de gestion simple qui ne hérite plus de `ctk.CTk`.
    - À l'initialisation, `MainApplication` crée explicitement une instance de `TkinterDnD.Tk()` comme fenêtre racine (`self.root`). C'est l'approche la plus susceptible de charger correctement les commandes Tcl.
    - Un `CTkFrame` principal (`ctk_root_frame`) est ensuite créé à l'intérieur de `self.root` pour héberger toute l'interface utilisateur CustomTkinter.
2. **Propagation de la Racine (`true_root_window`) :**
    - Cette nouvelle architecture a nécessité de passer la "vraie" fenêtre racine (`self.root`) à `AppController`.
    - `AppController` a été modifié pour accepter `true_root_window` et la passer à son tour à toutes les vues qu'il crée (`LoginView`, `MainView`).
    - Les vues `LoginView` et `MainView` ont été modifiées pour accepter ce paramètre `true_root_window` et l'utiliser comme `parent` pour toutes les fenêtres `CTkToplevel` et les dialogues (`messagebox`, `simpledialog`), assurant ainsi une gestion modale correcte.

Phase 6 : Retour à une Base Stable

Malgré les corrections architecturales, les erreurs complexes liées à l'intégration de `tkinterdnd2` persistaient, rendant l'application inutilisable. Face à cette instabilité, il a été décidé de **revenir à une base de code fonctionnelle en retirant complètement la fonctionnalité de glisser-déposer**.

- Les fichiers `app.py`, `controllers/app_controller.py`, `views/login_view.py`, et `views/main_view.py` ont été restaurés dans leur version stable d'avant l'introduction de `tkinterdnd2`.
- Les fichiers `utils/dnd_utils.py` et les imports associés ont été implicitement abandonnés.
- Cette version finale stable inclut les améliorations importantes de la Phase 2 (écritures atomiques, corrections de bugs initiaux) mais sans les complications du glisser-déposer.

Conclusion

Le projet a évolué d'une simple optimisation à une tentative d'intégration de fonctionnalités complexes, révélant des défis d'intégration significatifs entre les bibliothèques CustomTkinter et `tkinterdnd2`. La base de code finale est revenue à un état stable, fonctionnel et robuste, intégrant les améliorations de gestion des données critiques pour un déploiement multi-utilisateurs.

État Actuel :

- L'application est fonctionnelle et stable.
- Les données sont protégées contre la corruption par des écritures atomiques.
- La structure de code est propre et suit le modèle MVC.
- La fonctionnalité de glisser-déposer a été retirée pour garantir la stabilité.

Recommandations Futures :

Si la fonctionnalité de glisser-déposer reste une priorité, une investigation plus approfondie est nécessaire, potentiellement en explorant des bibliothèques alternatives ou en contactant les développeurs de CustomTkinter et tkinterdnd2 pour des solutions d'intégration spécifiques.

Rapport d'Analyse et de Développement de l'Application "GestionRemboursements"

Introduction et Objectif Initial

Le projet a débuté avec une demande d'analyse complète d'une application existante développée en Python avec la bibliothèque CustomTkinter. L'objectif principal était d'optimiser l'application pour la rendre plus **fluide, rapide et robuste**. Cette analyse devait se faire dans le contexte d'un déploiement futur : l'application serait compilée en un exécutable via PyInstaller et utilisée par six personnes sur un réseau partagé.

Phase 1 : Analyse Initiale et Optimisations Stratégiques

La première phase s'est concentrée sur l'analyse de l'architecture existante et l'identification des points d'amélioration potentiels, notamment en ce qui concerne le déploiement réseau.

1. **Analyse de la Commande PyInstaller :**
    - La commande de compilation a été examinée. Il a été noté que des dossiers de données (`config`, `donnees_partagees_mock`) étaient inclus via l'option `--add-data`.
    - Un point critique a été soulevé : l'inclusion du dossier `donnees_partagees_mock` dans le bundle de l'exécutable entrerait en conflit direct avec le besoin de partager et de modifier des données (comme `remboursements.json`) sur un réseau. Il a été recommandé de modifier le code pour qu'il puisse utiliser des chemins réseau en production, distincts des chemins locaux utilisés pour le développement.
2. **Gestion de la Concurrence des Données :**
    - Le principal risque identifié pour une utilisation multi-utilisateurs était l'utilisation de fichiers JSON comme base de données sur un partage réseau. Sans mécanisme de protection, si deux utilisateurs modifiaient un fichier (ex:

`remboursements.json`) simultanément, les modifications de l'un pourraient écraser celles de l'autre, menant à une perte de données.
    - La solution proposée a été la mise en place d'**écritures atomiques**. Cette technique consiste à écrire les nouvelles données dans un fichier temporaire, puis à renommer ce fichier temporaire pour remplacer l'original. Le renommage étant une opération généralement instantanée (atomique) sur la plupart des systèmes de fichiers, cela réduit drastiquement le risque de corruption en cas d'accès concurrents ou de plantage pendant l'écriture.
3. **Nettoyage et Amélioration de la Structure :**
    - Le fichier `auth_utils.py` a été identifié comme étant redondant et probablement obsolète, ses fonctionnalités étant déjà couvertes de manière plus intégrée par `models/user_model.py`. Sa suppression a été recommandée pour clarifier la base de code.
    - L'utilisation de `print()` pour le débogage a été notée. Il a été suggéré de remplacer ces appels par le module `logging` de Python, qui offre une gestion plus flexible et professionnelle des messages (niveaux de criticité, sortie vers des fichiers, formatage, etc.).

Phase 2 : Implémentation des Premières Corrections et Stabilité

Cette phase a consisté à mettre en œuvre les recommandations de l'analyse initiale pour stabiliser l'application.

1. **Mise en place des Écritures Atomiques :**
    - Une nouvelle fonction `_sauvegarder_fichier_json_atomic` a été développée.
    - Elle a été intégrée dans les fichiers `models/user_model.py` et `models/remboursement_data.py` pour remplacer les appels de sauvegarde standard, protégeant ainsi les fichiers `utilisateurs.json`, `codes_reset.json` et `remboursements.json`.
2. **Correction d'une Erreur d'Attribut (`AttributeError`) :**
    - Une erreur `AttributeError: 'MainView' object has no attribute '_action_voir_historique_docs'` a été signalée lors de la création d'une nouvelle demande.
    - L'analyse a révélé qu'un dictionnaire de callbacks dans `views/main_view.py` faisait référence à une méthode qui n'était pas définie dans la classe `MainView`.
    - La correction a consisté à ajouter la méthode manquante `_action_voir_historique_docs` à la classe `MainView`.

Phase 3 : Tentative d'Implémentation d'une Nouvelle Fonctionnalité (Glisser-Déposer)

Une demande a été faite pour ajouter la fonctionnalité de glisser-déposer (drag and drop) des fichiers dans toutes les fenêtres où une sélection de document est nécessaire. Cette étape s'est avérée être la plus complexe et a révélé des problèmes d'intégration profonds.

1. **Choix de la Technologie :**
    - La bibliothèque `tkinterdnd2` a été choisie, car c'est l'outil standard pour implémenter le glisser-déposer de fichiers avec Tkinter.
2. **Modifications Initiales :**
    - Un nouveau fichier utilitaire, `utils/dnd_utils.py`, a été créé pour héberger une fonction de "nettoyage" des chemins de fichiers reçus via le glisser-déposer.
    - Le fichier `views/main_view.py` a été lourdement modifié pour inclure des zones de dépôt visuelles et les callbacks associés. Une méthode `_creer_zone_selection_fichier_avec_dnd` a été créée pour factoriser ce code.

Phase 4 : Débogage Intensif des Erreurs d'Intégration

L'intégration de `tkinterdnd2` avec CustomTkinter a généré une série d'erreurs complexes.

1. **Problème d'Importation (`ImportError`) et de Reconnaissance :**
    - La première erreur signalée était une `ImportError: cannot import name 'envoyer_email_reinitialisation' from 'utils.email_utils'`. La cause principale identifiée était l'absence d'un fichier `__init__.py` dans le dossier `utils`, empêchant Python de le traiter comme un package. Une incohérence dans le nom de la fonction appelée depuis `auth_controller.py` a également été corrigée.
    - Par la suite, des avertissements persistants `WARNING:root:tkinterdnd2 non trouvé` sont apparus. Cela indiquait que, même si la bibliothèque était installée, l'environnement d'exécution Python ne parvenait pas à l'importer correctement, suggérant un problème d'environnement, d'installation ou de cache.
2. **Erreur Tcl `invalid command name "tkdnd::drop_target"` et Fenêtre "tk" Vide :**
    - C'est l'erreur la plus critique rencontrée. Elle signifie que les commandes Tcl (le langage sous-jacent de Tkinter) spécifiques à `tkinterdnd2` n'étaient pas chargées.
    - Ce problème est un symptôme d'un conflit d'initialisation entre `ctk.CTk()` et `TkinterDnD.Tk()`, qui se battent pour être la fenêtre racine de l'application. La tentative d'utiliser l'héritage multiple a souvent conduit à la création d'une fenêtre "tk" vide et blanche, signe de ce conflit.

3. **Erreur de Layout (`TclError: cannot use geometry manager...`) :**
    - Parallèlement, une erreur de géométrie est apparue, causée par un mélange des gestionnaires `.pack()` et `.grid()` pour les enfants d'un même widget parent dans les fenêtres de dialogue.
    - La solution a consisté à restructurer rigoureusement le layout de ces fenêtres dans `main_view.py`, en utilisant un conteneur principal (`CTkScrollableFrame` ou `CTkFrame`) qui est `pack`é dans le dialogue, et en utilisant ensuite un seul gestionnaire de géométrie (`pack` ou `grid`) pour tous les éléments à l'intérieur de ce conteneur.

Phase 5 : Refonte Architecturale pour la Stabilité

Face aux problèmes d'initialisation persistants, une refonte de l'architecture de démarrage de l'application a été mise en œuvre.

1. **Modification de `app.py` :**
    - La classe `MainApplication` a été transformée en une classe de gestion simple qui ne hérite plus de `ctk.CTk`.
    - À l'initialisation, `MainApplication` crée explicitement une instance de `TkinterDnD.Tk()` comme fenêtre racine (`self.root`). C'est l'approche la plus susceptible de charger correctement les commandes Tcl.
    - Un `CTkFrame` principal (`ctk_root_frame`) est ensuite créé à l'intérieur de `self.root` pour héberger toute l'interface utilisateur CustomTkinter.
2. **Propagation de la Racine (`true_root_window`) :**
    - Cette nouvelle architecture a nécessité de passer la "vraie" fenêtre racine (`self.root`) à `AppController`.
    - `AppController` a été modifié pour accepter `true_root_window` et la passer à son tour à toutes les vues qu'il crée (`LoginView`, `MainView`).
    - Les vues `LoginView` et `MainView` ont été modifiées pour accepter ce paramètre `true_root_window` et l'utiliser comme `parent` pour toutes les fenêtres `CTkToplevel` et les dialogues (`messagebox`, `simpledialog`), assurant ainsi une gestion modale correcte.

Phase 6 : Retour à une Base Stable

Malgré les corrections architecturales, les erreurs complexes liées à l'intégration de `tkinterdnd2` persistaient, rendant l'application inutilisable. Face à cette instabilité, il a été décidé de **revenir à une base de code fonctionnelle en retirant complètement la fonctionnalité de glisser-déposer**.

- Les fichiers `app.py`, `controllers/app_controller.py`, `views/login_view.py`, et `views/main_view.py` ont été restaurés dans leur version stable d'avant l'introduction de `tkinterdnd2`.
- Les fichiers `utils/dnd_utils.py` et les imports associés ont été implicitement abandonnés.
- Cette version finale stable inclut les améliorations importantes de la Phase 2 (écritures atomiques, corrections de bugs initiaux) mais sans les complications du glisser-déposer.

Conclusion

Le projet a évolué d'une simple optimisation à une tentative d'intégration de fonctionnalités complexes, révélant des défis d'intégration significatifs entre les bibliothèques CustomTkinter et `tkinterdnd2`. La base de code finale est revenue à un état stable, fonctionnel et robuste, intégrant les améliorations de gestion des données critiques pour un déploiement multi-utilisateurs.

État Actuel :

- L'application est fonctionnelle et stable.
- Les données sont protégées contre la corruption par des écritures atomiques.
- La structure de code est propre et suit le modèle MVC.
- La fonctionnalité de glisser-déposer a été retirée pour garantir la stabilité.

Rapport de Synthèse du Projet : Application de Gestion des Remboursements

I. Introduction et Objectif du Projet

Le présent document a pour objectif de détailler l'architecture, les fonctionnalités et les évolutions techniques du projet d'application de bureau pour la gestion des remboursements. L'objectif principal de cette application est de numériser, de centraliser et de sécuriser le processus de traitement des demandes de remboursement de trop-perçus clients, depuis leur initiation jusqu'au paiement final.

L'application a été conçue pour répondre aux besoins de plusieurs intervenants aux responsabilités distinctes, en offrant un workflow clair, une traçabilité complète des actions et un accès sécurisé basé sur les rôles de chaque utilisateur. Le projet a évolué d'un concept initial vers une application robuste, prête pour un déploiement en environnement multi-utilisateurs sur un réseau partagé.

II. Architecture Globale de l'Application

Conformément aux bonnes pratiques de développement et à la demande initiale, l'application a été structurée suivant une architecture inspirée du modèle **MVC**

(Modèle-Vue-Contrôleur). Cette séparation des préoccupations garantit une meilleure lisibilité, maintenabilité et évolutivité du code.

La structure du projet se décompose comme suit :

- **`models/`** : Contient toute la logique métier et l'accès aux données. C'est le "cerveau" de l'application.
    - Gestion des utilisateurs (`user_model.py`).
    - Logique de bas niveau pour la manipulation des données des remboursements (`remboursement_data.py`).
    - Définition des étapes du processus (workflow) de remboursement (`remboursement_workflow.py`).
    - Point d'entrée principal pour les opérations sur les remboursements (`remboursement_model.py`).
- **`views/`** : Comprend tous les composants de l'interface graphique (UI) construits avec la bibliothèque `customtkinter`. Chaque fichier représente une fenêtre, une frame ou un composant visuel spécifique.
- **`controllers/`** : Fait le lien entre les modèles (données) et les vues (interface). Les contrôleurs reçoivent les actions de l'utilisateur depuis l'interface et appellent la logique métier appropriée.
- **`utils/`** : Regroupe des modules utilitaires transverses, tels que la gestion des mots de passe, l'envoi d'e-mails, l'extraction de texte depuis des PDF ou la gestion de verrous de fichiers.
- **`config/`** : Centralise la configuration de l'application, incluant les chemins, les statuts, les descriptions des rôles et les paramètres de connexion.
- **`donnees_partagees_mock/` (pour le développement) et `DonneesApplication/` (en production)** : Stocke les fichiers de données JSON qui servent de base de données à l'application.

Le point d'entrée de l'application est `app.py`, qui initialise la fenêtre principale et lance le contrôleur principal (`AppController`).

III. Description Détaillée des Composants

A. Modèles (Logique Métier et Données)

La persistance des données est assurée par des fichiers au format JSON, une solution simple et lisible adaptée au contexte du projet.

1. **Gestion des Utilisateurs (`models/user_model.py`)**
    - **Stockage :** Les informations des utilisateurs (login, mot de passe haché, email, rôles) sont stockées dans `utilisateurs.json`.
    - **Sécurité :** Les mots de passe ne sont jamais stockés en clair. Ils sont systématiquement hachés via l'algorithme `bcrypt` grâce à la bibliothèque

`passlib` (`utils/password_utils.py`), garantissant une sécurité robuste.
    - **Fonctionnalités :** Ce modèle gère l'ajout, la mise à jour, la suppression, et la récupération d'informations sur les utilisateurs. Il inclut également la logique pour la modification de mot de passe par l'utilisateur et la réinitialisation via un code envoyé par e-mail.
2. **Gestion des Remboursements (`models/remboursement_*.py`)**
    - **Stockage :** Toutes les demandes de remboursement sont centralisées dans `remboursements.json`. Chaque demande est un objet JSON contenant toutes les informations pertinentes : détails du patient, montant, statut, historique, pièces jointes, etc.
    - **Fichiers Associés :** Les pièces jointes (factures, RIB, preuves de trop-perçu) sont stockées dans un dossier dédié sur le partage réseau (`DemandesRemboursement/`), avec un sous-dossier par demande pour une organisation claire.
    - **Workflow (`remboursement_workflow.py`) :** Cœur de la logique métier, ce module définit toutes les transitions d'état possibles pour une demande. Chaque action (ex: `accepter_constat_trop_percu_action`, `valider_demande_par_validateur_action`) vérifie le statut actuel, applique les modifications, met à jour le statut et enregistre une entrée dans l'historique de la demande.
    - **Manipulation des données (`remboursement_data.py`) :** Contient les fonctions de bas niveau pour lire et écrire dans le fichier `remboursements.json`. Il gère également la création des dossiers pour les pièces jointes et la suppression en cascade (suppression des fichiers si une demande est supprimée).

B. Vues (Interface Utilisateur)

L'interface a été développée avec `customtkinter` pour offrir une apparence moderne et une bonne expérience utilisateur.

1. **`LoginView` (`views/login_view.py`)** : Premier écran présenté à l'utilisateur. Il fournit les champs pour le nom d'utilisateur et le mot de passe, ainsi que des boutons pour se connecter, modifier son mot de passe ou enclencher la procédure de mot de passe oublié.

2. **`MainView` (`views/main_view.py`)** : Le tableau de bord principal après connexion.
    - Affiche la liste de toutes les demandes de remboursement.
    - Utilise un code couleur distinctif pour identifier rapidement le statut des demandes (Action requise, Terminée, Annulée).
    - Intègre une barre de recherche pour filtrer les demandes par nom, prénom ou référence.

    - Présente des boutons d'action en fonction des rôles de l'utilisateur (ex: "Nouvelle Demande" pour un `demandeur`, "Gérer Utilisateurs" pour un `admin`).
    - Implémente un mécanisme de **polling** intelligent qui vérifie périodiquement la date de modification du fichier de données et ne rafraîchit l'affichage que si des changements ont été détectés, optimisant ainsi les performances.
3. **`RemboursementItemView` (`views/remboursement_item_view.py`)** : C'est le composant qui affiche les détails d'une seule demande dans la liste de `MainView`. Il est particulièrement riche :
    - Affiche les informations clés (patient, montant, statut).
    - Présente une zone de texte avec l'historique complet des actions et commentaires.
    - Fournit des boutons pour voir et télécharger les dernières versions des pièces jointes.
    - Affiche dynamiquement les **boutons d'action du workflow** (ex: "Valider", "Refuser", "Confirmer Paiement") uniquement si l'utilisateur connecté a le bon rôle ET que la demande est au bon statut.
4. **Fenêtres de Dialogue et d'Administration :**
    - **`AdminUserManagementView`** : Une fenêtre d'administration complète permettant aux administrateurs de créer, modifier (login, email, rôles, mot de passe) et supprimer des utilisateurs.
    - **`DocumentViewerWindow`** : Un visualiseur de documents intégré capable d'afficher directement les images (PNG, JPG) et les fichiers PDF (grâce à `PyMuPDF/fitz`), et qui propose d'ouvrir les autres types de fichiers avec l'application par défaut du système.
    - **`HelpView` et `DocumentHistoryViewer`** : Des fenêtres d'aide contextuelle et de visualisation de l'historique des versions des documents.
    - De nombreuses boîtes de dialogue (`simpledialog`, `messagebox`) sont utilisées pour interagir avec l'utilisateur (demander un commentaire, confirmer une action, etc.).

C. Contrôleurs (Le "Chef d'Orchestre")

1. **`AppController` (`controllers/app_controller.py`)** : Le contrôleur principal qui orchestre le flux de l'application. Il gère l'affichage initial de la vue de connexion et, après une authentification réussie, bascule vers la vue principale. Il est également responsable de la déconnexion et de l'affichage de l'avertissement de sécurité pour les administrateurs.

2. **`AuthController` (`controllers/auth_controller.py`)** : Gère toute la logique liée à l'authentification et à la gestion des utilisateurs. Il fait le pont entre les actions de l'utilisateur dans les vues (`LoginView`, `AdminUserManagementView`) et les fonctions du modèle (`user_model.py`).

3. **`RemboursementController` (`controllers/remboursement_controller.py`)** : Gère toutes les interactions liées aux demandes de remboursement. Il prépare les données pour l'affichage, traite les soumissions de formulaires (nouvelle demande, correction), et appelle les fonctions de workflow du modèle en réponse aux clics sur les boutons d'action.

D. Utilitaires et Configuration

- **`utils/pdf_utils.py`** : Fournit une fonction `extraire_infos_facture` qui tente d'extraire automatiquement des informations (nom, référence) d'un PDF de facture, une fonctionnalité à forte valeur ajoutée pour pré-remplir les formulaires.
- **`utils/email_utils.py`** : Gère l'envoi d'e-mails formatés (HTML et texte brut) pour la procédure de réinitialisation de mot de passe, en utilisant les paramètres SMTP définis dans `config_email.ini`.
- **`config/settings.py`** : Agit comme le centre névralgique de la configuration. Il définit de manière centralisée les statuts, les descriptions détaillées des rôles, et surtout, les chemins vers les fichiers de données.

IV. Workflow Détaillé et Rôles Utilisateurs

L'application implémente un workflow métier précis, basé sur une succession de statuts et d'interventions par des utilisateurs aux rôles spécifiques.

1. **Rôles Définis :**
    - **Demandeur** (`p.neri`) : Crée la demande.
    - **Comptable Trésorerie** (`m.lupo`) : Vérifie le trop-perçu.
    - **Validateur Chef** (`j.durousset`, `b.gonnet`) : Valide la demande et les pièces jointes.
    - **Comptable Fournisseur** (`p.diop`) : Effectue et confirme le paiement.
    - **Visualiseur Seul** (`m.fessy`) : Accès en lecture seule.
    - **Admin** (`admin`) : Droits complets sur l'application et ses données.
2. **Cycle de Vie d'une Demande :**
    - **Création (`STATUT_CREEE`) :** Le `demandeur` initie une demande avec les informations du patient et les pièces jointes.
    - **Constat du Trop-Perçu :** La `comptable_tresorerie` analyse la demande.
        - Si conforme, elle ajoute une preuve, un commentaire, et passe le statut à `STATUT_TROP_PERCU_CONSTATE`.
        - Si non conforme, elle la refuse avec un commentaire, passant le statut à `STATUT_REFUSEE_CONSTAT_TP`, ce qui la renvoie au `demandeur`.
    - **Validation :** Le `validateur_chef` examine la demande et les preuves.
        - Si tout est correct, il valide, passant le statut à `STATUT_VALIDEE`.

        - En cas de problème, il refuse avec un commentaire, passant le statut à `STATUT_REFUSEE_VALIDATION_CORRECTION_MLUPO`, la renvoyant à la `comptable_tresorerie`.
    - **Paiement :** Le `comptable_fournisseur` traite la demande validée. Une fois le virement effectué, il confirme le paiement, ce qui clôture la demande avec le statut final `STATUT_PAIEMENT_EFFECTUE`.
    - **Annulation :** Une demande peut être annulée par son créateur (ou un admin) si elle a été refusée, via le statut `STATUT_ANNULEE`.

V. Évolutions Majeures : Robustesse et Préparation au Déploiement

La phase finale du développement s'est concentrée sur la transformation de l'application en un outil prêt pour la production dans un environnement réseau partagé.

1. **Problématique de la Concurrence d'Accès :** Il a été identifié que l'utilisation de fichiers JSON par plusieurs utilisateurs simultanément pouvait mener à des pertes de données (conditions de concurrence de type "lecture-modification-écriture").

2. **Solution - Implémentation d'un Verrouillage Atomique :** Pour résoudre ce problème critique, un mécanisme de verrouillage par fichier (`.lock`) a été mis en place.

    - Un nouveau module `utils/file_lock.py` a été créé pour encapsuler cette logique de manière sécurisée et réutilisable.
    - Ce module utilise un `context manager` (`with FileLock(...)`) pour garantir que le verrou est toujours libéré, même en cas d'erreur.
    - Il gère une attente (timeout) et informe l'utilisateur si les données sont inaccessibles.
    - Les modèles `remboursement_data.py` et `user_model.py` ont été intégralement modifiés pour utiliser ce verrou avant chaque opération de lecture ou d'écriture sur les fichiers JSON partagés, garantissant ainsi l'intégrité des données.
3. **Optimisation pour le Déploiement Réseau :**

    - Pour éviter d'avoir à modifier le code source avant chaque compilation, la gestion des chemins de données a été externalisée.
    - Un nouveau fichier de configuration `config/app_config.ini` a été introduit. Il contient le chemin absolu vers le dossier de données partagé sur le réseau.
    - Le fichier `config/settings.py` a été modifié pour lire ce chemin au démarrage de l'application, la rendant ainsi totalement portable. L'administrateur n'a qu'à éditer ce simple fichier texte pour configurer le déploiement.
    - Cette approche dissocie complètement le code de l'application de son environnement de déploiement.

VI. Conclusion

Le projet a abouti à la création d'une application de bureau complète, fonctionnelle et sécurisée. Elle répond précisément au cahier des charges initial en offrant un workflow structuré, une gestion fine des droits par utilisateur et une traçabilité complète des actions.

Résumé Complet du Développement de l'Application de Gestion des Remboursements

Ce document détaille l'ensemble du processus de développement, des spécifications initiales à la mise en œuvre des fonctionnalités complètes du workflow, en passant par les améliorations de l'interface utilisateur et les optimisations architecturales.

Phase 1 : Initialisation du Projet et Architecture

Le projet a débuté avec l'objectif de créer une application de bureau pour une entreprise, afin de numériser et de suivre un processus de remboursement en plusieurs étapes impliquant différents utilisateurs.

- **Architecture MVC (Modèle-Vue-Contrôleur) :** Dès le départ, une structure de projet organisée a été adoptée et affinée. Les fichiers ont été répartis dans des dossiers logiques :
    - `models/` : Pour la logique métier et la manipulation des données (ex: `user_model.py`, `remboursement_model.py`).
    - `views/` : Pour tous les composants de l'interface graphique (ex: `login_view.py`, `main_view.py`).
    - `controllers/` : Pour orchestrer les interactions entre les modèles et les vues (ex: `app_controller.py`, `auth_controller.py`).
    - `utils/` : Pour les fonctions utilitaires réutilisables (ex: `pdf_utils.py`, `password_utils.py`, `email_utils.py`).
    - `config/` : Pour les fichiers de configuration (`settings.py`, `config_email.ini`).
- **Gestion des Données :** Pour la persistance des données, le choix a été fait d'utiliser des fichiers au format JSON, stockés dans un dossier `donnees_partagees_mock`, simulant un emplacement partagé pour un déploiement futur sur un serveur.
- **Authentification des Utilisateurs :** Un système d'authentification de base a été mis en place, utilisant le fichier `utilisateurs.json` pour stocker les noms d'utilisateur, les emails et les mots de passe hachés à l'aide de la bibliothèque `passlib` (avec l'algorithme bcrypt).

Phase 2 : Implémentation du Workflow - Étape 1 (P. Neri)

La première étape fonctionnelle implémentée a été la création d'une demande de remboursement par l'utilisateur `p.neri`.

- **Création de Demande :**

    - Une fenêtre dédiée a été créée, accessible via un bouton "Nouvelle Demande" dans la vue principale.
    - Le formulaire de création permet de saisir les informations du client (Nom, Prénom), la référence de la facture, le montant et une description de la raison de la demande.
- **Extraction Automatique des Données (OCR via PDF) :**
    - Une fonctionnalité clé a été développée dans `utils/pdf_utils.py` pour automatiser le remplissage du formulaire.
    - En utilisant la bibliothèque `pdfplumber`, l'application lit le contenu textuel d'un fichier PDF de facture téléchargé par l'utilisateur.
    - Des expressions régulières (regex) ont été affinées de manière itérative pour extraire de manière fiable le nom, le prénom et la référence de la facture, en se basant sur la structure constante des documents fournis.
- **Gestion des Pièces Jointes :**
    - La logique de stockage des pièces jointes a été mise en place. Chaque nouvelle demande entraîne la création d'un sous-dossier unique dans le répertoire principal `Demande_Remboursement`, nommé d'après la référence de la facture (après nettoyage pour être un nom de dossier valide).
    - Les fichiers téléchargés (facture et RIB) sont copiés et renommés de manière unique dans ce dossier.
    - Un fichier `informations_demande.txt` récapitulant toutes les informations de la demande est également généré et sauvegardé dans ce même dossier.

Phase 3 : Interface Utilisateur et Affichage des Données

Une grande partie du travail a consisté à rendre l'application fonctionnelle et agréable à utiliser pour tous les utilisateurs.

- **Liste des Demandes :**
    - La vue principale (`MainView`) a été dotée d'une liste scrollable pour afficher toutes les demandes de remboursement existantes.
    - Chaque item de la liste a été conçu pour afficher de manière claire et concise les informations clés : patient, référence, montant, statut, dates, etc.
- **Améliorations Visuelles et de Lisibilité :**
    - L'affichage de chaque demande a été itérativement amélioré, passant d'un simple empilement de textes à une grille structurée multi-colonnes pour une meilleure utilisation de l'espace horizontal.
    - La description et l'historique des commentaires ont été placés dans des zones dédiées pour ne pas surcharger l'affichage principal des informations.
    - **Mise en évidence visuelle :** Un code couleur a été implémenté pour distinguer les demandes en fonction de leur statut :
        - Vert pour une demande nécessitant une action de l'utilisateur connecté.
        - Bleu pour une demande terminée.
        - Rouge/Bordeaux pour une demande annulée.

    - Une **légende des couleurs** a été ajoutée en bas de la fenêtre principale pour expliquer la signification de chaque couleur.
- **Fonctionnalité de Recherche :** Une barre de recherche a été ajoutée au-dessus de la liste, permettant de filtrer dynamiquement les demandes en saisissant une partie du nom, du prénom ou de la référence de la facture.

Phase 4 : Gestion Multi-Utilisateurs et Déploiement

Pour préparer l'application à un environnement multi-utilisateurs sur un réseau partagé, plusieurs mécanismes essentiels ont été mis en place.

- **Universalité des Chemins d'Accès :**
    - Toute la logique de gestion des chemins de fichiers a été centralisée dans `config/settings.py`.
    - Une fonction `get_application_base_path()` a été créée pour que les chemins vers les dossiers de données (`donnees_partagees_mock`, `Demande_Remboursement`) soient relatifs à l'emplacement de l'exécutable final, et non à l'environnement de développement. Cela rend l'application portable.
- **Actualisation des Données en Temps Réel :**
    - Pour que les modifications faites par un utilisateur soient visibles par les autres, un système de **polling** a été implémenté. Toutes les 10 secondes, l'application vérifie la date de dernière modification du fichier `remboursements.json` et rafraîchit automatiquement la liste des demandes si des changements sont détectés.
    - Un bouton de **rafraîchissement manuel** a également été ajouté pour permettre à l'utilisateur de forcer la mise à jour à tout moment.
- **Simplification de la Concurrence :**
    - Une première approche de verrouillage par fichier (`.lock`) a été discutée et implémentée pour éviter les conflits d'édition.
    - Cependant, après analyse, il a été constaté que le workflow séquentiel basé sur les rôles et les statuts rendait ce système complexe et source de problèmes (verrous persistants). Il a donc été **décidé de le retirer** au profit de la logique de workflow qui empêche intrinsèquement deux utilisateurs de rôles différents d'agir sur la même demande au même moment.

Phase 5 : Amélioration de la Visualisation des Documents

L'affichage des pièces jointes a été entièrement revu pour offrir une meilleure expérience utilisateur.

- **Visionneuse Interne :** Une classe `DocumentViewerWindow` dédiée a été créée (dans `views/document_viewer.py`) pour afficher les documents directement dans l'application.
- **Support Multi-Format :**

    - **PDF :** La visionneuse utilise `PyMuPDF` pour afficher **toutes les pages** d'un document PDF verticalement. Le zoom initial est fixé à 130% pour une bonne lisibilité par défaut.
    - **Images :** Les formats d'image courants (PNG, JPG, etc.) sont affichés à leur taille originale (100%) via la bibliothèque `Pillow`.
    - **Autres Formats (`.docx`, `.odt`) :** Après avoir initialement envisagé un affichage en texte brut, il a été décidé de ne pas gérer l'aperçu interne pour ces formats complexes afin d'éviter des dépendances lourdes. À la place, l'application propose à l'utilisateur d'ouvrir ces fichiers avec l'application par défaut de son système d'exploitation.
- **Suppression des Contrôles :** Après plusieurs itérations (avec boutons de zoom, plein écran, etc.), la visionneuse a été simplifiée pour n'offrir qu'un affichage direct, sans boutons de contrôle, le défilement étant géré par les barres de défilement natives.

Phase 6 : Finalisation du Workflow et Gestion des Retours

Le cycle de vie complet d'une demande de remboursement a été implémenté.

- **Étape de `j.durousset` / `b.gonnet` (Validateur Chef) :** Les utilisateurs avec le rôle `validateur_chef` peuvent maintenant agir sur les demandes au statut "Trop-perçu constaté". Ils peuvent :
    - **Valider** la demande, qui passe au statut "Validée (en attente de paiement)".
    - **Refuser** la validation, avec un commentaire obligatoire, ce qui renvoie la demande à l'étape précédente (à `m.lupo`) avec le statut "Refusée - Validation".
- **Étape de `p.diop` (Comptable Fournisseur) :**
    - Les utilisateurs avec le rôle `comptable_fournisseur` voient un bouton d'action sur les demandes au statut "Validée".
    - En cliquant sur "Confirmer Paiement", la demande est finalisée et passe au statut "Paiement effectué (Terminée)". La date et l'heure du paiement sont enregistrées et affichées dans les détails de la demande.
- **Gestion des Retours (Resoumission) :**
    - Une logique complète a été ajoutée pour gérer les cas où une demande est refusée et renvoyée à une étape antérieure.
    - L'utilisateur responsable de l'étape (par exemple, `p.neri` ou `m.lupo`) doit obligatoirement fournir un nouveau commentaire et de nouvelles pièces jointes (RIB, preuve de trop-perçu).
    - Les anciennes versions des documents ne sont pas écrasées mais conservées. Un bouton **"Historique des Documents"** apparaît sur les demandes ayant plus d'une version de pièce jointe, permettant de consulter toutes les itérations.

Phase 7 : Fonctionnalités d'Administration

Les droits et outils spécifiques à l'utilisateur `admin` ont été développés.

- **Popup d'Avertissement :** À chaque connexion, l'administrateur est accueilli par une fenêtre popup qui l'avertit de ses responsabilités et des risques associés à ses droits étendus.
- **Gestion des Utilisateurs :**
    - Un bouton "Gérer Utilisateurs" a été ajouté à la vue principale, visible uniquement par l'admin.
    - Ce bouton ouvre une nouvelle fenêtre (`AdminUserManagementView`) qui liste tous les utilisateurs (sauf le compte "admin" lui-même).
    - Depuis cette fenêtre, l'admin peut :
        - **Supprimer** un utilisateur (avec une confirmation pour éviter les erreurs).
        - **Modifier** un utilisateur existant. Une fenêtre de dialogue permet de changer son **email** et de modifier ses **rôles** via une liste de cases à cocher.
    - Une fonctionnalité **d'information sur les rôles** a été ajoutée, affichant une description détaillée de chaque rôle et la liste des utilisateurs qui y sont actuellement assignés.

Phase 8 : Optimisation et Refactorisation du Code

Tout au long du développement, un effort a été fait pour maintenir une base de code propre et organisée.

- **Séparation des fichiers :** Les fichiers `main_view.py` et `remboursement_model.py`, qui devenaient trop volumineux, ont été scindés en modules plus petits et plus logiques, conformément à l'architecture MVC et au principe de responsabilité unique.
- **Débogage et Correction :** Plusieurs erreurs ont été identifiées et corrigées, notamment des `AttributeError` et `NameError` dues à des incohérences dans les noms de variables ou des appels de méthodes manquantes, ainsi que des problèmes de logique comme la perte d'extensions de fichiers lors du renommage.

Le projet est maintenant fonctionnellement complet, couvrant l'ensemble du workflow de remboursement et incluant des fonctionnalités d'administration robustes.

Rapport Final de Développement : Application de Gestion de Remboursements

Ce document récapitule l'intégralité du processus de développement itératif, d'optimisation et de fiabilisation de l'application de gestion des remboursements. Le projet a évolué d'une application de bureau fonctionnelle à une solution multi-utilisateurs robuste, conçue pour un déploiement en réseau et un usage sur le long terme.

Phase 1 : Analyse Initiale et Audit de la Concurrence

Le projet a débuté avec une base de code complète et fonctionnelle, structurée selon une architecture de type MVC (Modèles-Vues-Contrôleurs). L'objectif initial était de valider la

robustesse de l'application pour un déploiement sur un dossier réseau partagé, accessible par plusieurs utilisateurs simultanément.

- **Architecture Initiale :** Le projet était déjà bien organisé, avec une séparation claire des responsabilités entre les modèles (logique métier et accès aux données), les vues (interface graphique avec `CustomTkinter`), et les contrôleurs (orchestration).
- **Gestion des Données Initiale :** Le stockage reposait sur des fichiers JSON monolithiques (`utilisateurs.json`, `remboursements.json`) centralisés dans un dossier de test (`donnees_partagees_mock`).
- **Problématique Principale :** Assurer que les opérations de lecture et d'écriture sur les fichiers JSON partagés ne mènent pas à de la corruption de données en cas d'accès concurrentiels.

L'audit du système de gestion de fichiers (`utils/data_manager.py` et `utils/file_lock.py`) a révélé un mécanisme déjà très solide. L'utilisation d'un fichier de verrouillage (`.lock`) créé avec des flags exclusifs (`os.O_EXCL`) et la sauvegarde atomique (écriture dans un fichier temporaire puis renommage avec `os.replace`) étaient des pratiques robustes et suffisantes pour garantir l'intégrité des données dans un environnement réseau standard. **Aucune modification majeure de cette logique n'a été jugée nécessaire à ce stade.**

Phase 2 : Optimisation des Performances et Préparation au Déploiement

Le premier axe d'amélioration s'est porté sur la fluidité et la préparation d'un exécutable fonctionnel en réseau.

1. **Mise en Cache des Données :** Pour éviter les lenteurs dues aux lectures répétées du fichier `remboursements.json` sur le réseau, un **cache en mémoire** a été implémenté dans la `MainView`. Au lieu de lire le fichier à chaque recherche ou filtrage, l'application charge désormais l'ensemble des données une seule fois, puis travaille sur cette copie locale, rendant l'interface instantanée. Un mécanisme de *polling* basé sur la date de modification du dossier de données assure le rafraîchissement du cache si un autre utilisateur a effectué une modification.
2. **Externalisation de la Configuration :** Une erreur fondamentale dans la stratégie de déploiement a été corrigée. L'idée initiale d'embarquer les données partagées dans l'exécutable via PyInstaller a été abandonnée, car elle aurait isolé chaque utilisateur. À la place, le chemin vers le dossier réseau partagé a été externalisé dans `config/settings.py`. Une variable `IS_DEPLOYMENT_MODE` a été ajoutée pour permettre de basculer facilement entre un chemin local pour le développement et un chemin UNC pour la production, sans avoir à modifier plusieurs fichiers.
3. **Correction du Déploiement :** La commande `pyinstaller` a été entièrement revue pour :
    - **Exclure** les données partagées de l'exécutable.
    - **Inclure** correctement les dossiers de configuration (`config/`) et d'icônes (`assets/`).

    - **Collecter** de manière exhaustive toutes les dépendances des librairies graphiques (`customtkinter`, `PIL`) et de manipulation de PDF (`pymupdf`).
    - **Ajouter manuellement** les librairies SSL (`libssl-3.dll`) via `--add-binary` pour garantir le fonctionnement des envois d'e-mails dans l'exécutable compilé.

Phase 3 : Refactorisation de l'Architecture pour la Scalabilité

Face à une prévision de 10 nouvelles demandes par semaine (plus de 500 par an), l'architecture basée sur un fichier `remboursements.json` monolithique a été identifiée comme un risque pour la performance et la fiabilité à long terme. Une refactorisation majeure a été entreprise.

- **Passage à une Architecture Granulaire :** Le concept d'un fichier unique a été abandonné. Désormais, **chaque demande de remboursement est stockée dans son propre fichier JSON individuel** (ex: `D20250610....json`).
- **Nouvelle Structure de Dossiers :** La structure du dossier partagé a été réorganisée pour plus de clarté :
    - `remboursements/data/` pour les fichiers JSON des demandes actives.
    - `remboursements/fichiers/` pour les dossiers de pièces jointes.
    - `remboursements/archive/` pour les demandes anciennes.
- **Impact sur la Performance :** Les opérations d'écriture (création, mise à jour de statut) ne modifient plus qu'un petit fichier de quelques kilo-octets au lieu d'un fichier potentiellement lourd de plusieurs mégaoctets, ce qui réduit drastiquement les temps de latence et les risques de conflit de verrouillage.

Cette modification a nécessité une réécriture significative du `modèle de données` (`remboursement_data.py`) mais a posé les bases d'une application capable de gérer des milliers de dossiers sans dégradation de performance.

Phase 4 : Améliorations de la Robustesse et de l'Expérience Utilisateur

Une fois les fondations de performance et de scalabilité assurées, une série de fonctionnalités a été ajoutée pour professionnaliser l'application.

1. **Validation Stricte des Données (`Pydantic`) :** Pour se prémunir contre la corruption de données au-delà des simples erreurs de syntaxe JSON, la librairie `Pydantic` a été intégrée. Un schéma de validation (`models/schemas.py`) a été créé pour définir la structure et le type de chaque champ d'une demande. À chaque lecture, les données sont validées contre ce schéma. Si une incohérence est détectée (ex: un texte à la place d'un montant), le fichier est considéré comme corrompu.
2. **Récupération Automatique des Données :** Le système de validation a été couplé à un mécanisme de sauvegarde et de récupération.
    - À chaque modification d'un fichier de demande, l'ancienne version est automatiquement sauvegardée en `.bak`.
    - Si la validation détecte un fichier corrompu, l'application tente de le restaurer depuis sa sauvegarde `.bak`.

    - Un message d'alerte (`utils/ui_messages.py`) prévient l'utilisateur de l'opération en termes simples.
3. **Archivage Automatique :** Pour maintenir une interface claire et rapide, une routine d'archivage a été mise en place. Au démarrage, l'application déplace automatiquement les demandes terminées ou annulées de plus de 12 mois vers un dossier d'archive, allégeant la charge de travail par défaut.
4. **Personnalisation et Ergonomie de l'Interface :**
    - **Filtres et Tri :** Des menus déroulants ont été ajoutés sur la vue principale pour permettre de filtrer les demandes (par statut, "en attente de mon action", etc.) et de les trier (par date, montant, nom). Une case à cocher permet d'inclure les archives dans les recherches.
    - **Panneau "Mon Profil" :** Une nouvelle fenêtre (`views/profile_view.py`) a été créée, permettant à chaque utilisateur de :
        - Changer son mot de passe et son adresse e-mail.
        - Choisir un thème de couleur (`blue`, `green`, `dark-blue`) et un mode d'apparence (`Light`, `Dark`, `System`).
        - Définir son filtre d'affichage par défaut au démarrage.
        - Mettre une photo de profil (avec création d'un module `utils/image_utils.py` pour le masquage circulaire).
    - **Amélioration Visuelle :** L'affichage du nom et de la photo de profil dans l'en-tête a été agrandi et amélioré pour une meilleure visibilité.
5. **Panneau de Configuration Administrateur :** Pour faciliter la maintenance, un panneau sécurisé (`views/admin_config_view.py`) a été ajouté, accessible uniquement aux administrateurs, leur permettant de modifier les paramètres d'envoi d'e-mail (SMTP) sans avoir à recompiler l'application.

Phase 5 : Débogage et Finalisation

Le processus de développement a été marqué par la résolution de plusieurs problèmes techniques complexes, typiques de la finalisation d'une application de bureau :

- **Conflits de Dépendances :** Résolution des incompatibilités entre les versions de `passlib` et `bcrypt`.
- **Erreurs de Type :** Correction des `TypeError` liés à la conversion des dates par `Pydantic`.
- **Erreurs de Synchronisation d'Interface (`_tkinter.TclError`) :** Après plusieurs tentatives, ce bug tenace a été définitivement résolu en refactorisant en profondeur la gestion des fenêtres dans `app_controller.py`. L'approche finale consiste à pré-charger toutes les vues au démarrage et à les afficher/cacher au lieu de les détruire/recréer, éliminant ainsi toute condition de course.