École Nationale des Sciences Appliquées d'Oujda
Troisième année Génie Informatique

Rapport – Application de Suivi de Traitements Médicaux

MOUSTAGHIT Souhail
RHIATE Ayoub
JALOULI Abdelilah
LEMRANI Moad

Année universitaire : 2024–2025

Table des matières

Liste des figures

Liste des tableaux

Concéption et analyse du projet 1

1.1 Introduction

1.1.1 Présentation générale du projet

Ce projet a pour objectif de concevoir et développer une application de bureau simple, intuitive et fonctionnelle dédiée au suivi des traitements médicaux des patients. Cette application s'adresse principalement aux professionnels de santé (comme un médecin ou des assistants médicaux), afin de faciliter la gestion quotidienne des dossiers médicaux.
Dans un contexte médical où la précision et l'organisation sont essentielles, cette solution permettra de centraliser les informations des patients, de suivre leurs traitements, de gérer les rendez-vous, et d'accéder rapidement à des statistiques utiles. L'outil a pour vocation de remplacer les documents papier ou les suivis manuels, souvent source d'erreurs ou de perte d'informations.

1.1.2 Objectif principal de l'application

L'objectif principal est de développer un outil informatique fiable et ergonomique destiné à assister les professionnels de santé dans la gestion et le suivi des traitements médicaux. Cette application vise à :

- Gérer les fiches des patients.
- Associer et suivre les traitements médicaux.
- Consulter l'historique médical de chaque patient.
- Rechercher rapidement un patient ou un traitement, avec des options de filtrage.
- Analyser des statistiques médicales utiles.

Ce projet repose sur une interface utilisateur conviviale réalisée avec JavaFX, une gestion de données via une base de données SQLite, et l'intégration de plusieurs composants graphiques modernes (tableaux dynamiques, alertes, listes, filtres, etc.).
Il s'agit d'un outil destiné à améliorer le suivi personnalisé des patients et l'organisation du travail médical tout en facilitant l'accès rapide à des données critiques.

1.2 Analyse des besoins

1.2.1 Identification des utilisateurs

L'application est destinée à être utilisée par différents types d'utilisateurs, chacun avec des rôles et des niveaux d'accès spécifiques :

Table 1.1: Description des rôles utilisateurs dans l'application.

Rôle	Description
Médecin	Utilisateur principal qui gère les dossiers des patients, ajoute/modifie les traitements, consulte les statistiques.
Secrétaire	Gère les informations administratives des patients, planifie les rendez-vous, aide à la saisie des données.

1.2.2 Fonctionnalités principales

Les fonctionnalités sont décrites sous forme de fiches fonctionnelles, avec les objectifs et les règles de gestion à respecter.

Ajouter un patient :
Objectif : Permettre à un utilisateur habilité d'ajouter un patient à l'aide d'un formulaire.
Règles de gestion :

- Le champ Nom est obligatoire.
- Le numéro de sécurité sociale doit être unique et conforme au format attendu.
- L'âge est automatiquement calculé à partir de la date de naissance.
- Le numéro de téléphone doit respecter un format standard (ex. : 10 chiffres).
- Le sexe doit être choisi parmi une liste prédéfinie.
- Un même patient ne peut être enregistré qu'une seule fois (contrôle d'unicité via le numéro de sécurité sociale).

Modifier un patient :
Objectif : Permettre à un utilisateur autorisé de modifier les informations d'un patient existant.
Règles de gestion :

- La modification n'est autorisée qu'aux utilisateurs habilités.
- Le numéro de sécurité sociale ne peut pas être modifié.
- L'âge se recalcule automatiquement en cas de modification de la date de naissance.
- Les données modifiées doivent être validées avant d'être enregistrées.

Supprimer un patient :
Objectif : Permettre à un utilisateur autorisé de supprimer un patient de la base de données.
Règles de gestion :

- Une confirmation est exigée avant la suppression.
- La suppression entraîne également la suppression de tous les traitements associés.

Gérer les traitements d'un patient :
Objectif : Ajouter, modifier ou supprimer un traitement médical lié à un patient.
Règles de gestion :

- Chaque traitement est associé à un patient via son identifiant unique.
- Les champs type et date de début sont obligatoire.
- La date de fin doit être postérieure ou égale à la date de début.
- Le statut du traitement est automatiquement défini selon les dates (ex. : actif, terminé).
- La suppression d'un traitement nécessite une confirmation.

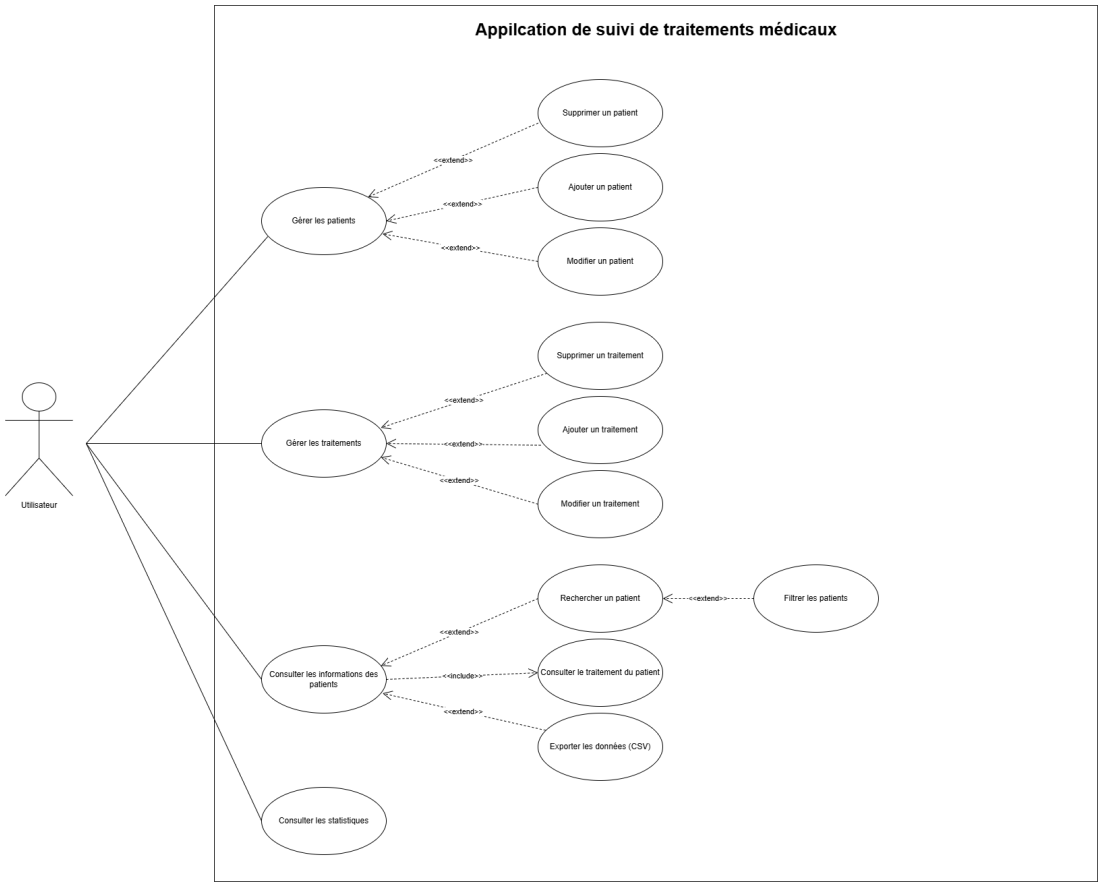
Visualiser les informations patients et traitements :
Objectif : Permettre la consultation détaillée des informations administratives et médicales d'un patient.
Règles de gestion :

- Les traitements sont affichés en liste triée par date.
- Une fonction de recherche permet de filtrer les patients (nom, numéro de sécurité sociale, etc.).

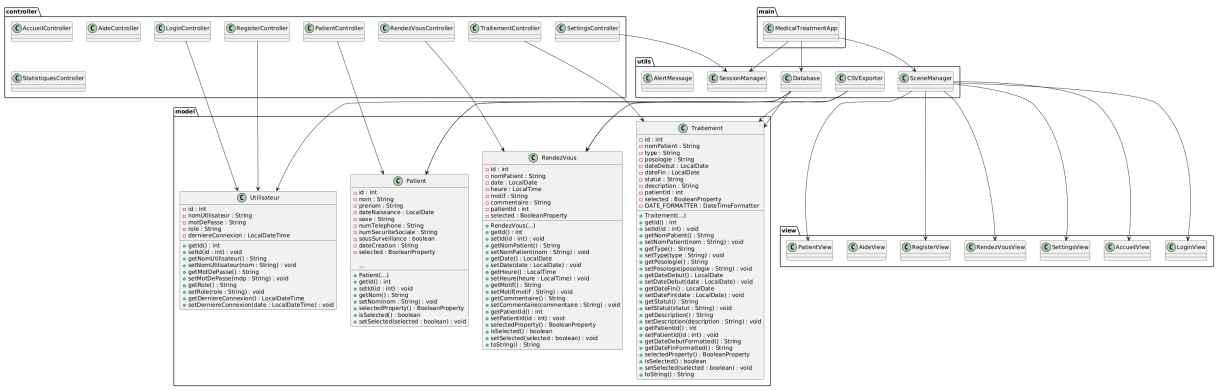
1.3 Modélisation

1.3.1 Diagrammes UML

▶ Diagramme de cas d'utilisation :



▶ Diagramme de classes :



1.3.2 Modéle de données

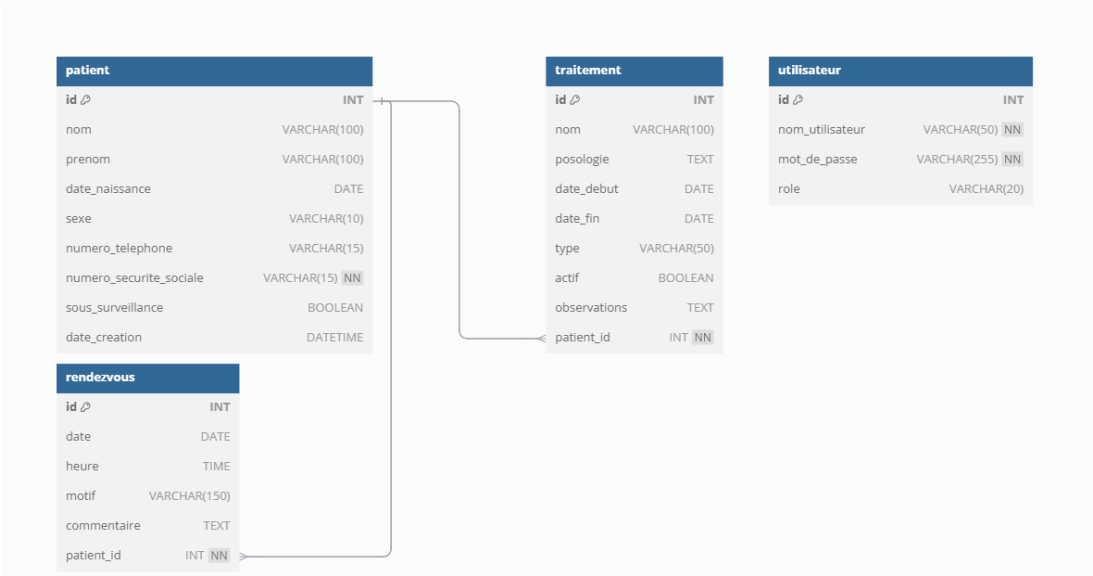


Figure 1.1: Structure de la base de données relationnelle

1.4 Conception de l'interface utilisateur

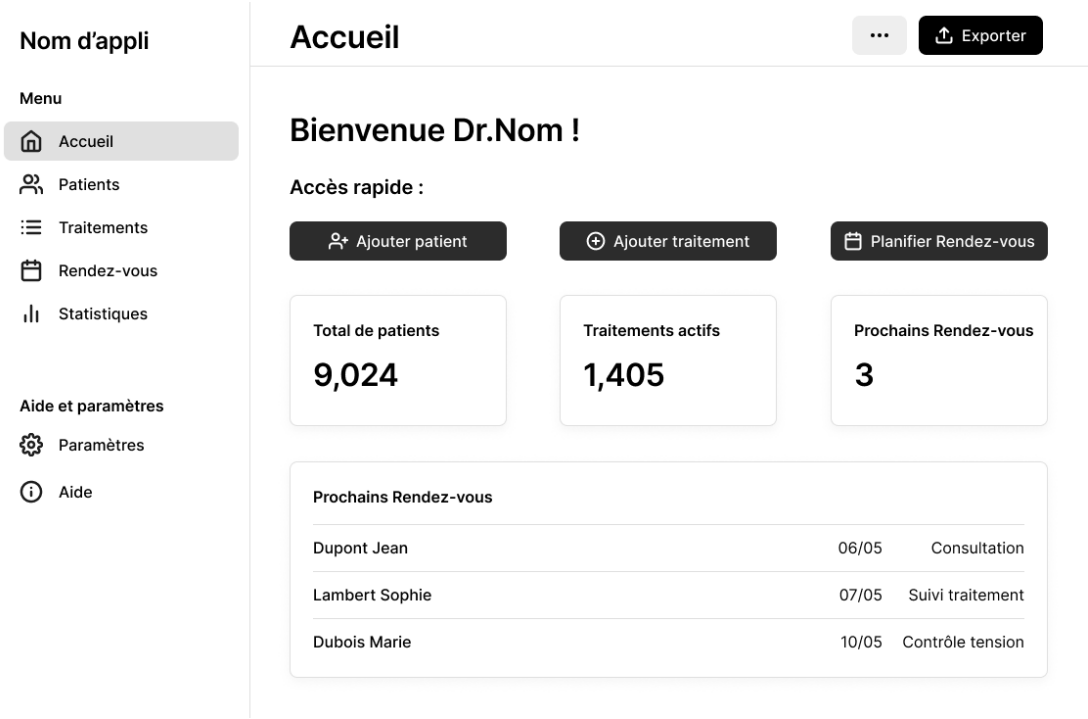


Figure 1.2: Interface d'accueil de l'application

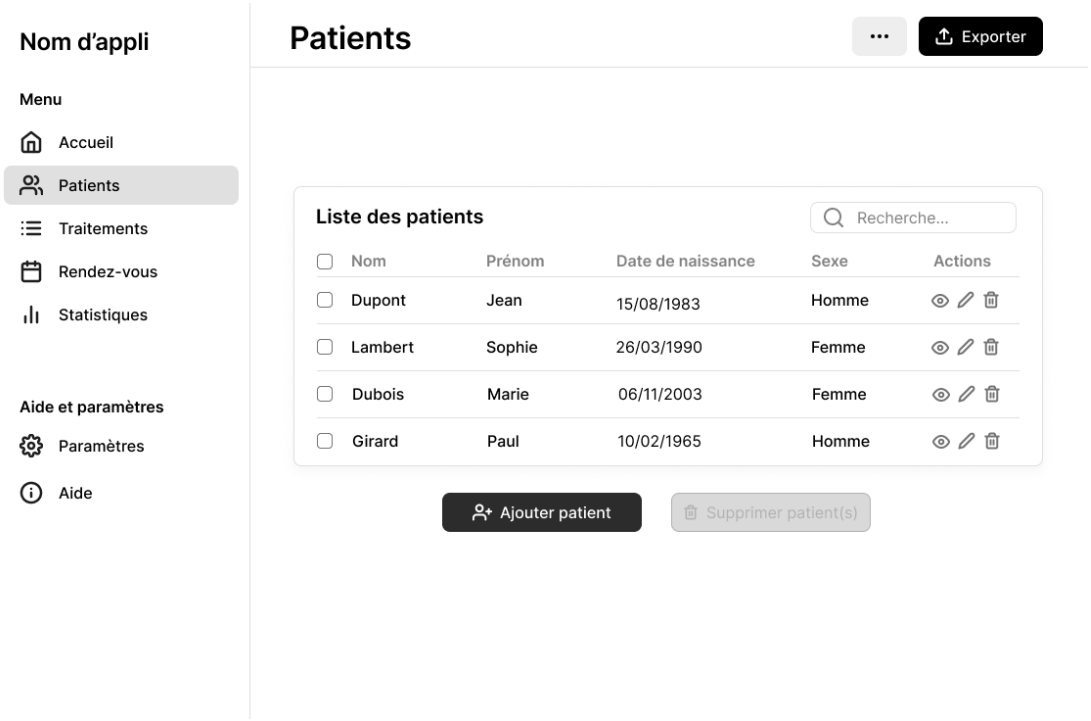


Figure 1.3: Interface de gestion des patients

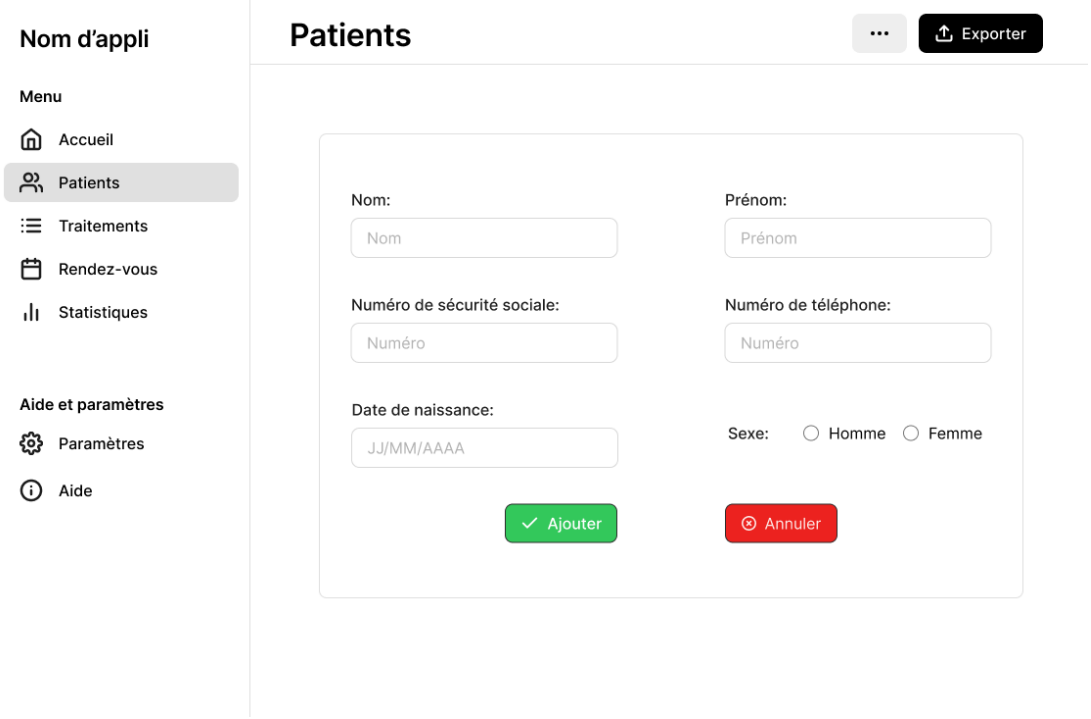


Figure 1.4: Formulaire d'ajout d'un patient

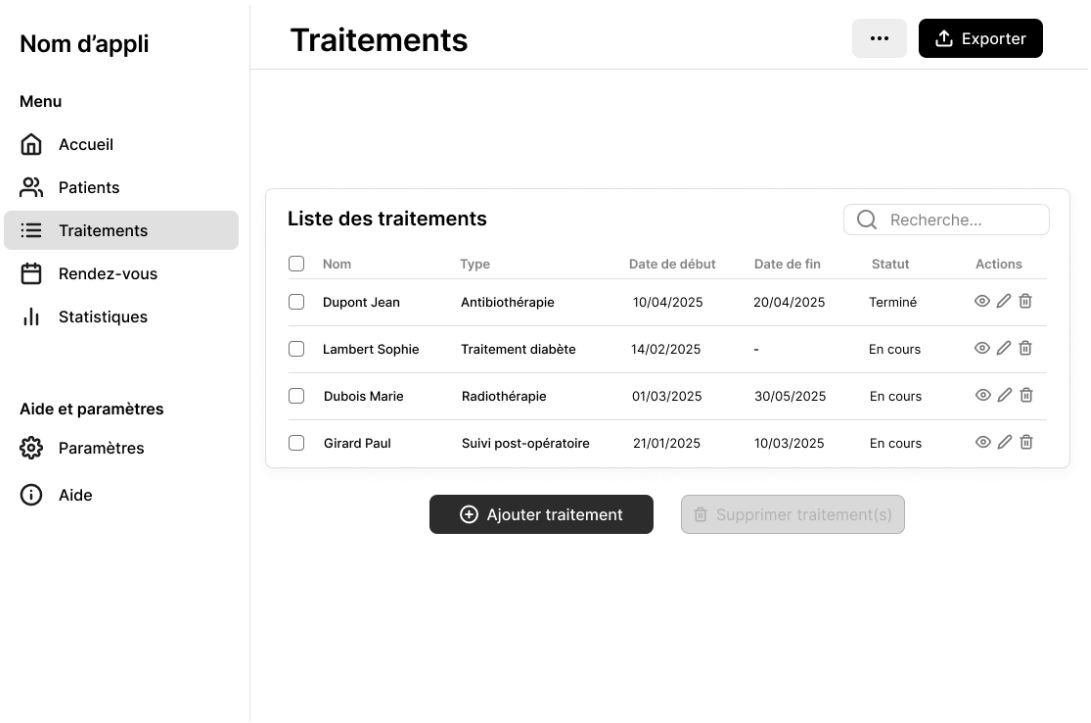


Figure 1.5: Interface de gestion des traitements

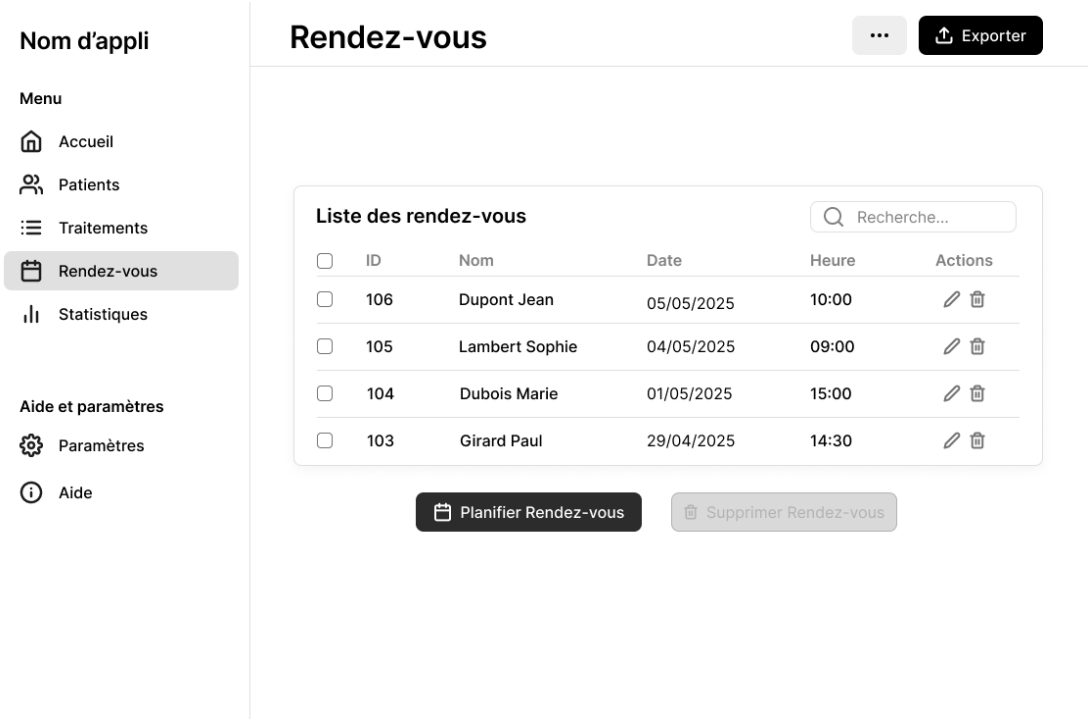


Figure 1.6: Interface de gestion des rendez-vous

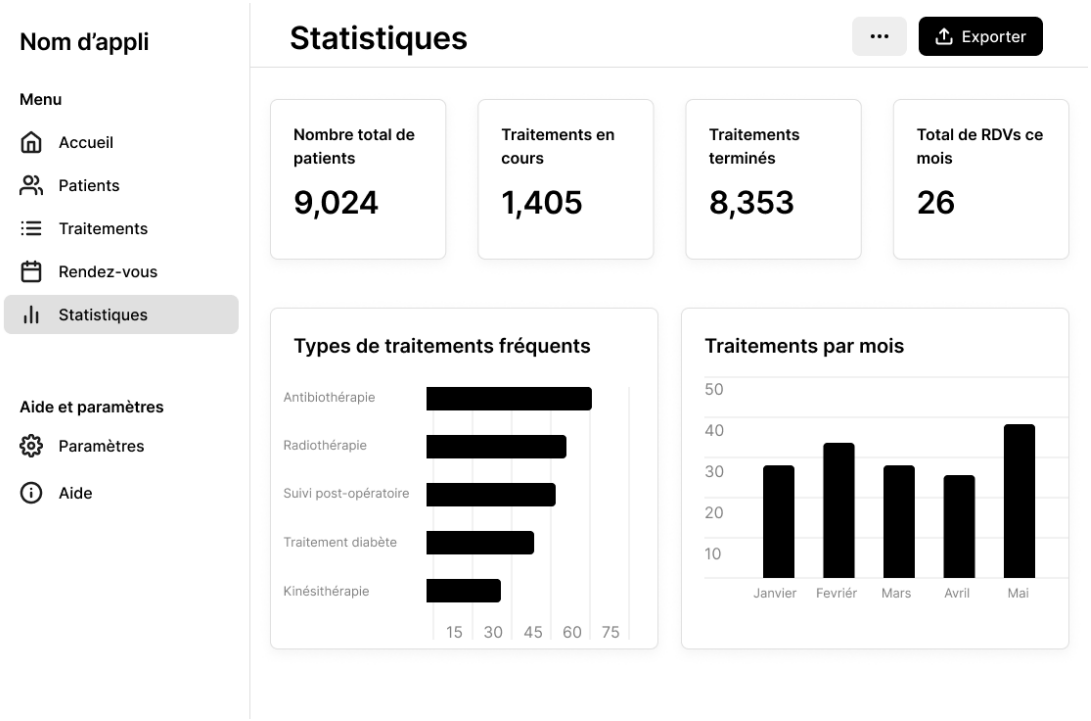


Figure 1.7: Tableau de bord des statistiques médicales

1.5 Structure du projet

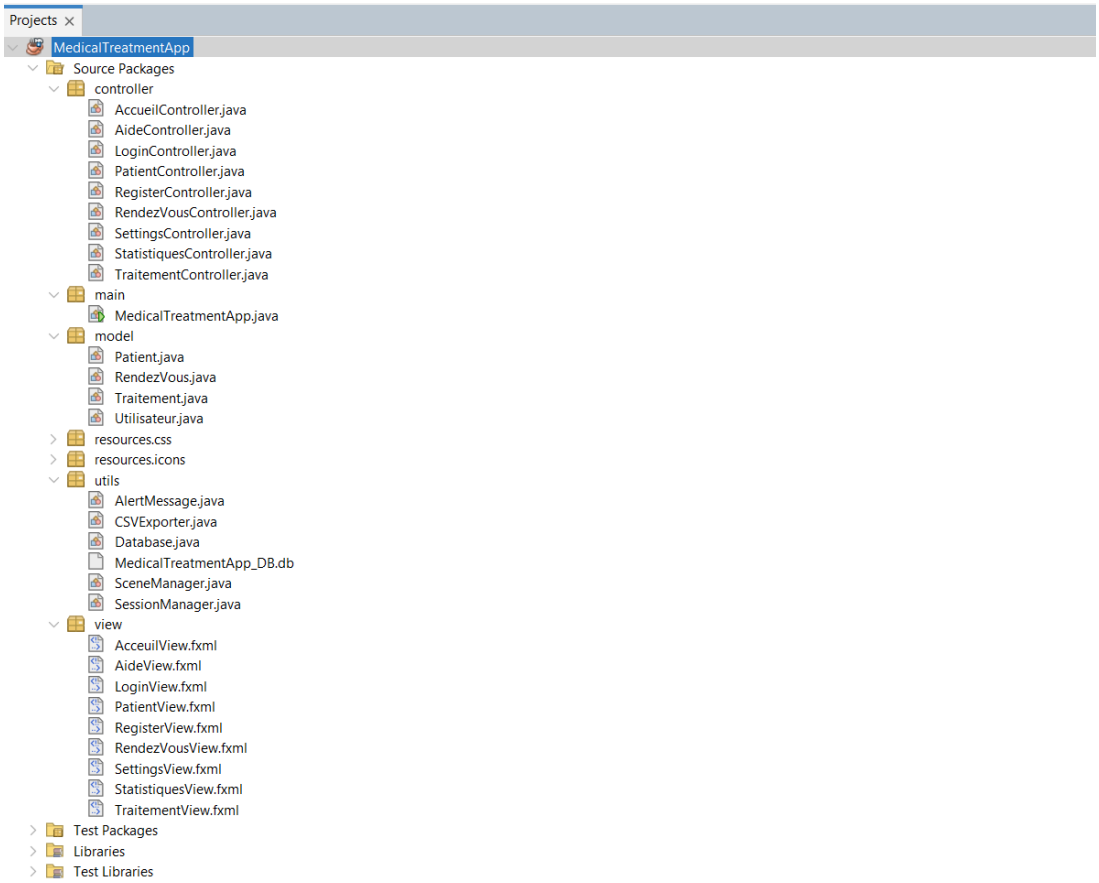


Figure 1.8: Organisation du projet en MVC

Développement et mise en œuvre de l'application 2

2.1 Introduction

Après avoir défini les aspects conceptuels du projet dans le premier chapitre, ce second chapitre se concentre sur la mise en œuvre technique de l'application. Il détaille les choix réalisés lors du développement, les fonctionnalités implémentées, ainsi que l'organisation du code et de la base de données.
Nous allons présenter la structure relationnelle de la base de données et son interaction avec l'application via JDBC. Ensuite, nous décrirons de manière précise les différentes fonctionnalités développées, comme la gestion des patients, des traitements et des rendez-vous. Chaque fonctionnalité sera accompagnée de son interface graphique et des éléments de contrôle associés.
Enfin, ce chapitre abordera également la conception de l'interface utilisateur, les aspects liés à la sécurité et la validation des données, ainsi qu'un bilan global sur l'état du projet et les améliorations possibles.

2.2 Utilisation de Git et GitHub

Le développement de l'application s'est appuyé sur l'utilisation de Git comme système de gestion de versions, et de GitHub comme plateforme d'hébergement et de collaboration. Cet environnement a permis de suivre l'évolution du projet, de gérer les différentes branches de développement, et de conserver un historique clair des modifications apportées au code source.

L'objectif principal de l'utilisation de Git était d'assurer un suivi rigoureux des changements effectués sur les fichiers du projet, garantissant ainsi une traçabilité complète. De plus, Git a facilité la collaboration entre les membres de l'équipe, même à distance, en limitant les risques de conflits lors des fusions. Enfin, la gestion des branches a offert la possibilité d'expérimenter de nouvelles fonctionnalités sans compromettre la stabilité de la version principale de l'application, permettant ainsi un développement agile et maîtrisé.

2.2.1 Principales commandes utilisées :

- `git init` : initialisation du dépôt Git local au démarrage du projet.
- `git add .` : ajout de tous les fichiers modifiés ou nouveaux à l'index en vue d'un commit.
- `git commit -m "Message"` : enregistrement d'un état du projet avec un message décrivant les modifications.
- `git branch nom-branche` : création de branches dédiées pour isoler certaines fonctionnalités (ex. `gestion-rdv`, `interface-patient`).
- `git checkout` / `git switch` : navigation entre les différentes branches.
- `git merge` : fusion des branches après validation des développements.
- `git pull` / `git push` : synchronisation avec le dépôt distant GitHub.

2.2.2 Organisation des commits :

Chaque modification majeure du projet a été enregistrée par un commit clair et précis. Voici quelques exemples représentatifs :

- `Initialisation du projet JavaFX avec configuration JDBC`
- `Ajout et amélioration des fonctionnalités CRUD pour Patient, Traitement et RendezVous`
- `Mise en place de l'export CSV pour les patients, traitements et rendez-vous`
- `Correction des bugs de validation, notamment SSN`
- `Ajout du composant DatePicker dans les formulaires`
- `Développement de l'interface graphique pour la prise de rendez-vous`

- `Fusion des nouvelles fonctionnalités validées dans la branche principale`
- `Amélioration du style UI et correction du bug d'export dans la table Traitement`

2.2.3 Gestion des branches :

Le développement s'est organisé autour de deux branches principales :

- `main` : branche stable contenant la version validée et fonctionnelle de l'application.
- `features` : branche dédiée à l'intégration des nouvelles fonctionnalités avant validation.

Cette organisation a permis une meilleure coordination du travail, une traçabilité claire des évolutions, ainsi qu'une réduction des conflits lors des fusions.

2.3 Gestion de la base de données

La gestion des données médicales est un pilier fondamental de l'application **MediConnect**, impliquant une structuration rigoureuse des entités, une interaction fluide avec la base de données, ainsi qu'un encadrement des opérations CRUD (Create, Read, Update, Delete) à travers des classes Java dédiées. Cette section présente l'architecture de la base de données, les technologies utilisées pour l'accès aux données, et les principales opérations réalisées dans l'application.

2.3.1 Structure relationnelle de la base de données

L'application repose sur une base de données relationnelle **SQLite**, choisie pour sa légèreté, sa portabilité et son intégration native avec les applications Java de type desktop. Le modèle relationnel a été soigneusement conçu pour refléter les entités clés du domaine médical, tout en respectant les règles de *normalisation* afin de réduire la redondance et de garantir l'*intégrité des données*.

Les entités principales sont modélisées sous forme de tables :

- **patient** : stocke les données personnelles (nom, prénom, date de naissance, numéro de sécurité sociale).
- **traitement** : regroupe les traitements médicaux prescrits aux patients, incluant les posologies, durées, observations, et statut actif.
- **rendezvous** : enregistre les rendez-vous programmés, associés à un patient donné, avec motif, date, heure, et commentaire.
- **utilisateur** : contient les informations de connexion des utilisateurs de l'application (nom d'utilisateur, mot de passe, rôle).

Les relations entre les tables sont assurées par des *clés étrangères*, activées par la commande `PRAGMA foreign_keys = ON;`. Par exemple, `rendezvous.patient_id` référence `patient.id`.

Le modèle de données a initialement été décrit en **DBML (Database Markup Language)** pour permettre une visualisation rapide sur des outils tels que `dbdiagram.io`, puis traduit en **SQL** et exécuté automatiquement lors de l'initialisation de l'application.

2.3.2 Interaction via JDBC

L'interaction entre l'application JavaFX et la base de données SQLite est assurée par l'API **JDBC (Java Database Connectivity)**. Cette API permet de :

- Se connecter à la base de manière fiable.
- Exécuter des requêtes SQL, simples ou paramétrées.
- Manipuler les résultats via des objets `ResultSet`.

- Gérer les transactions, exceptions et fermetures de connexions.

Un fichier central, **Database.java**, situé dans le package `utils`, encapsule toute la logique de connexion à la base. La méthode statique `connectDB()` établit une connexion SQLite sécurisée :

```
    public static Connection connectDB() {
        Connection conn = null;
        try {
            String url = "jdbc:sqlite:src/utils/MedicalTreatmentApp_DB.db";
            conn = DriverManager.getConnection(url);
            try (Statement stmt = conn.createStatement()) {
                stmt.execute("PRAGMA foreign_keys = ON;");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return conn;
    }
```

Ce design permet une *réutilisation centralisée* de la connexion à travers toute l'application, renforçant la *modularité* et la *maintenabilité* du code.

2.3.3 Gestion des opérations CRUD

L'ensemble des opérations **CRUD** (Create, Read, Update, Delete) sur les entités de la base de données est réalisé à travers des **classes de contrôle Java** (telles que `RendezVousController`, `PatientController`, etc.). Ces classes utilisent des requêtes SQL paramétrées, mises en œuvre via des objets `PreparedStatement`.

Parmi les opérations principales gérées dans l'application :

- **Création** : insertion d'un nouveau rendez-vous ou d'un nouveau patient dans les tables correspondantes.
- **Lecture** : récupération de la liste des rendez-vous, traitements ou patients, avec des jointures sur d'autres tables (ex. nom du patient via `JOIN`).
- **Mise à jour** : modification des données d'un rendez-vous existant, telles que la date, l'heure ou le motif.
- **Suppression** : suppression d'un rendez-vous, d'un patient ou d'un traitement à partir de son identifiant.

Les résultats obtenus à partir de ces requêtes sont traités grâce aux objets `ResultSet`, qui permettent d'itérer sur les lignes retournées par la base et de les injecter dans des objets Java (`Patient`, `RendezVous`, `Traitement`, etc.). Ce procédé facilite l'intégration des données dans les interfaces graphiques de l'application.

2.3.4 Exemples de requêtes SQL utilisées

```
    -- Insérer un nouveau patient
    INSERT INTO patient (nom, prenom, date_naissance, sexe, numero_telephone,
    numero_securite_sociale, sous_surveillance, date_creation)
    VALUES (?, ?, ?, ?, ?, ?, ?, CURRENT_TIMESTAMP);

    -- Récupérer tous les traitements actifs d'un patient
    SELECT * FROM traitement WHERE patient_id = ? AND actif = 1;

    -- Supprimer un rendez-vous
    DELETE FROM rendezvous WHERE id = ?;

    -- Mise à jour d'un rendez-vous
    UPDATE rendezvous
    SET date = ?, heure = ?, motif = ?, commentaire = ?,
    patient_id = (SELECT id FROM patient WHERE CONCAT(prenom, ' ', nom) = ?)
    WHERE id = ?;

```

```
18    -- Authentification d'un utilisateur
19    SELECT * FROM utilisateur WHERE nom_utilisateur = ? AND mot_de_passe = ?;
```

2.4 Fonctionnalités et interfaces de l'application

Cette section détaille les principales fonctionnalités proposées par l'application ainsi que les interfaces associées, conçues pour faciliter la gestion médicale et administrative.

2.4.1 Authentification

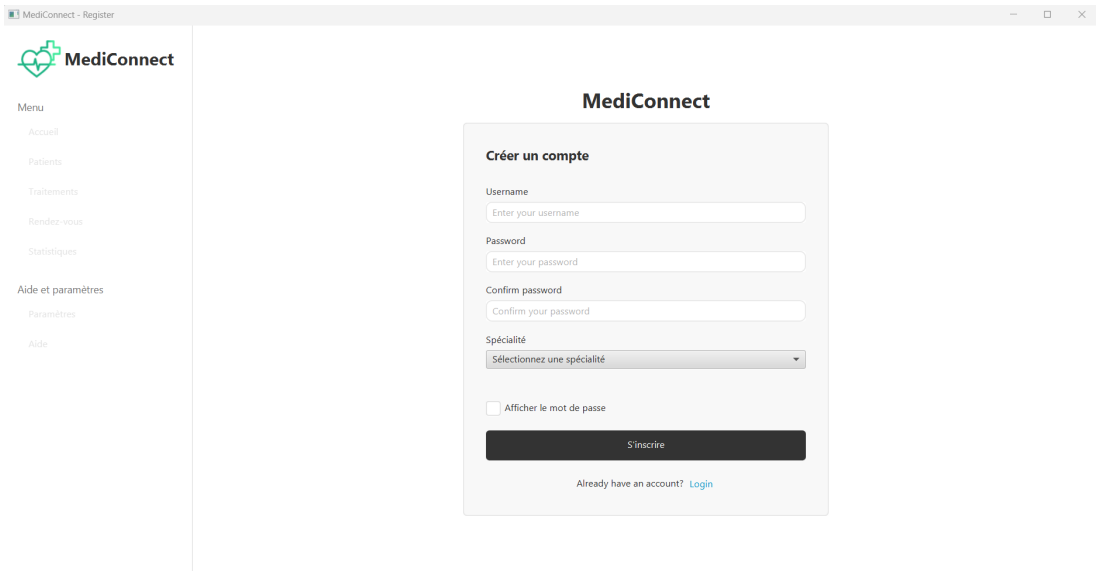


Figure 2.1: Interface d'inscription

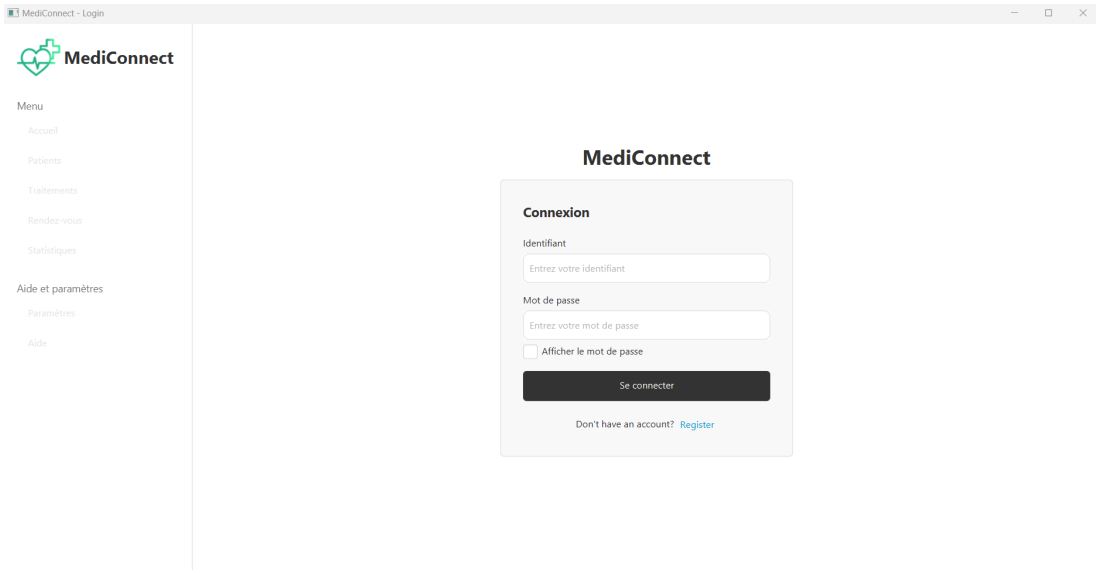


Figure 2.2: Interface de connexion

Le système d'authentification permet aux utilisateurs de s'inscrire en choisissant leur rôle, soit *secrétaire*, soit *médecin*. La connexion est sécurisée et intègre une gestion des rôles, offrant une interface commune à tous les utilisateurs, avec la possibilité d'adapter certaines fonctionnalités selon le profil, si nécessaire.

Après la connexion, un message de bienvenue dynamique s'affiche :

- Bienvenue Dr. [Nom] pour les médecins.
- Bienvenue [Nom d'utilisateur] pour les secrétaires.

Des alertes claires informent l'utilisateur en cas d'erreurs, d'avertissements ou de confirmations, par exemple pour une mauvaise saisie ou la réussite d'un enregistrement.

2.4.2 Page d'accueil

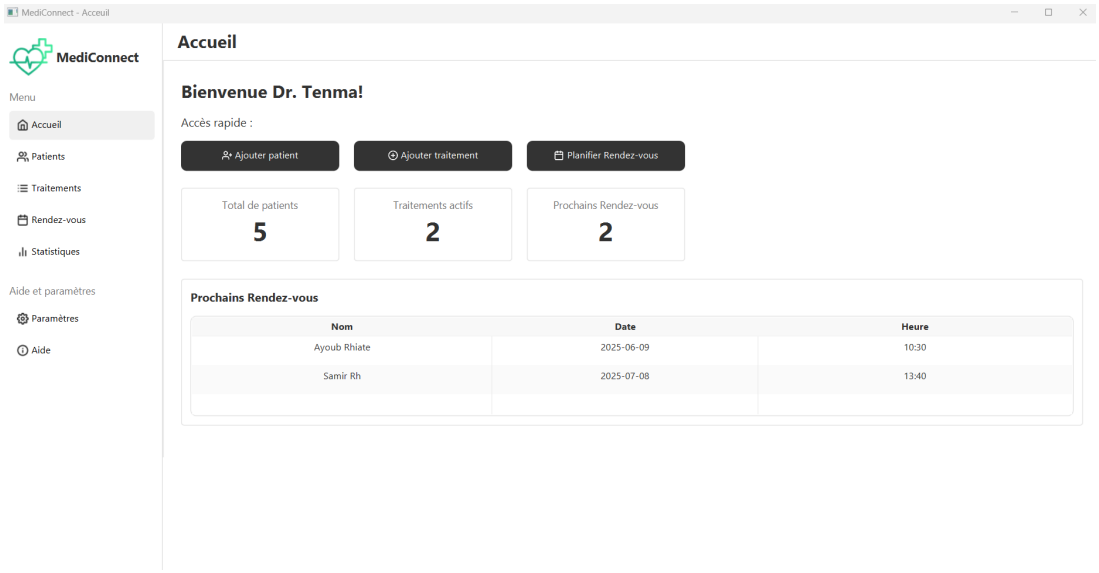


Figure 2.3: Interface d'accueil

Le tableau de bord présente un résumé synthétique des éléments clés :

- Les prochains rendez-vous à venir.
- Le nombre total de patients enregistrés.
- La liste des patients avec rendez-vous programmés.

Des boutons d'accès rapide sont disponibles en permanence, facilitant la navigation vers les différentes sections de l'application.

2.4.3 Gestion des patients

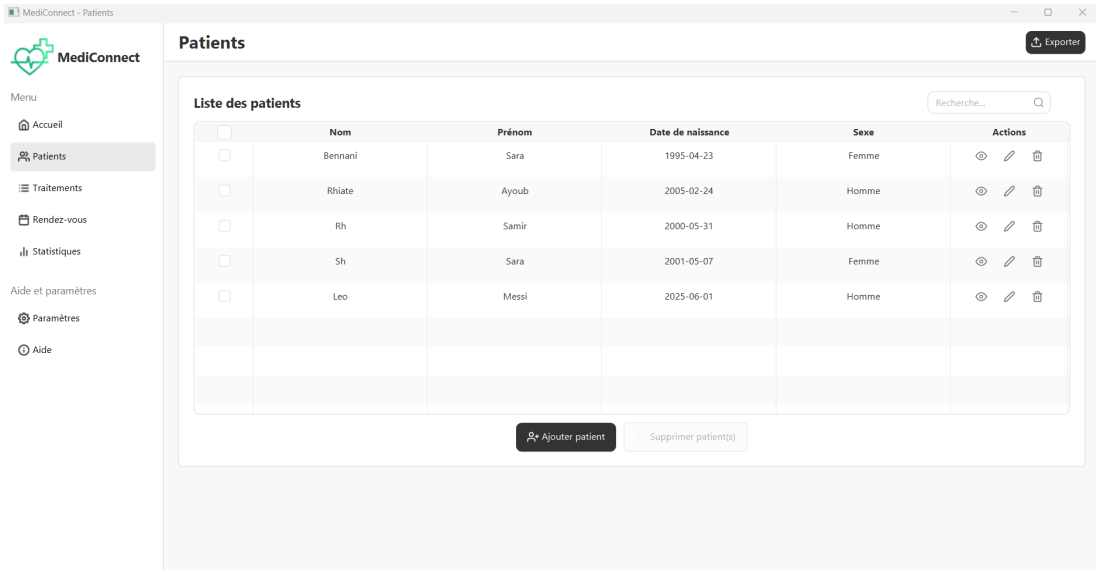


Figure 2.4: Interface Patients

La page dédiée aux patients offre une gestion complète, incluant l'ajout, la modification, la suppression et la consultation des dossiers patients. Les informations médicales sont affichées de manière claire et lisible. Un moteur de recherche intégré permet de retrouver rapidement un patient spécifique.

2.4.4 Gestion des traitements

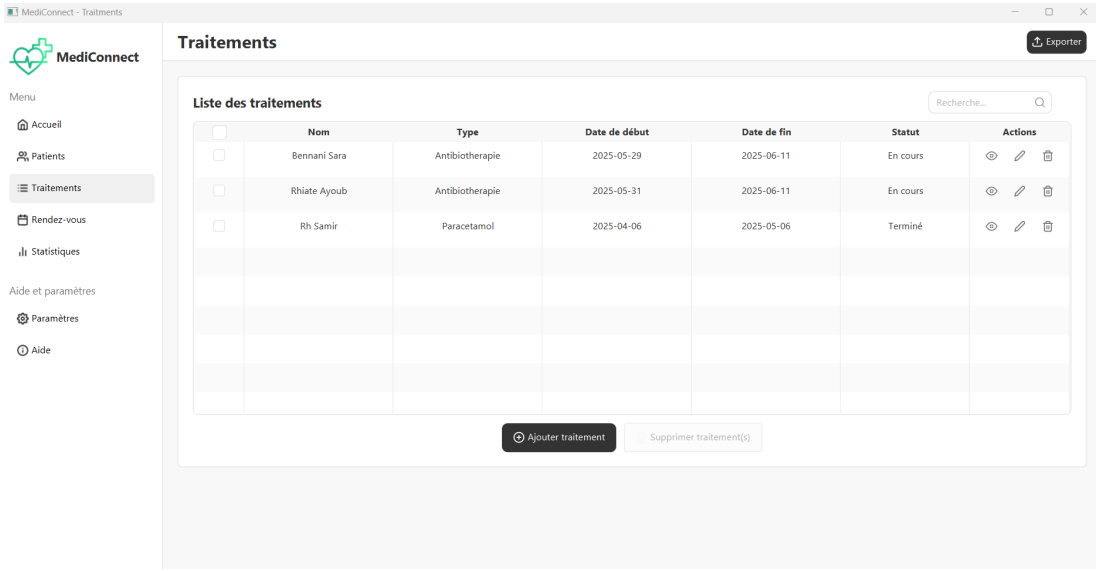


Figure 2.5: Interface Traitements

Cette interface permet de créer, consulter, mettre à jour et supprimer les traitements médicaux. Chaque traitement est lié directement au patient concerné. L'interface est intuitive, facilitant la gestion quotidienne des traitements.

2.4.5 Prise de rendez-vous

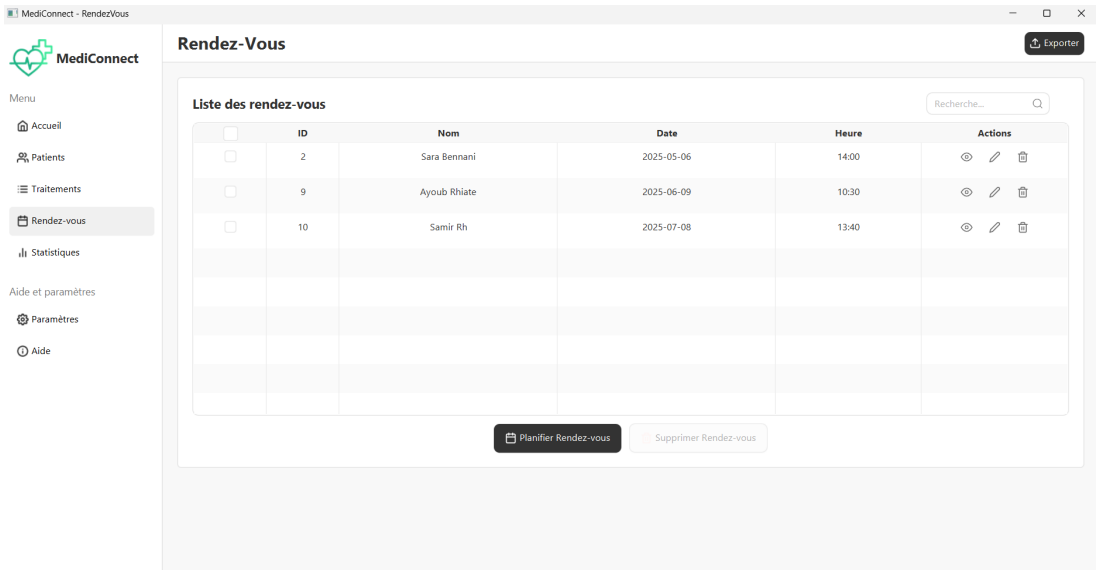


Figure 2.6: Interface RendezVous

La gestion des rendez-vous inclut toutes les opérations classiques : ajout, modification, suppression et consultation. Chaque rendez-vous est associé à un patient et à un médecin. Le calendrier des consultations est affiché clairement pour un suivi optimal. La navigation est simplifiée grâce aux boutons d'accès rapide présents en permanence.

2.4.6 Recherche et filtres dynamiques

L'application propose des fonctionnalités de recherche avancée et de filtres dynamiques sur les listes de patients, traitements et rendez-vous, permettant aux utilisateurs de trouver rapidement les informations pertinentes selon différents critères.

2.4.7 Statistiques médicales

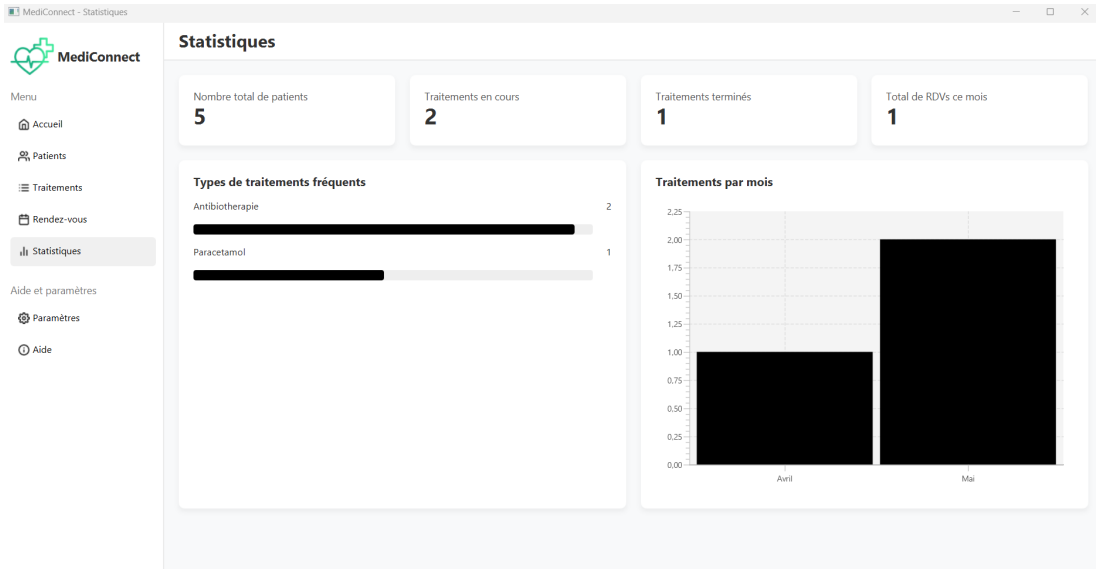


Figure 2.7: Interface Statistiques

La page des statistiques affiche des graphiques et des indicateurs clés concernant les patients, les rendez-vous et les traitements. Ces éléments visuels aident à suivre l'activité médicale de manière efficace. L'interface est conçue pour une lecture rapide et intuitive, tout en conservant les boutons d'accès rapide pour la navigation.

2.4.8 Paramètres

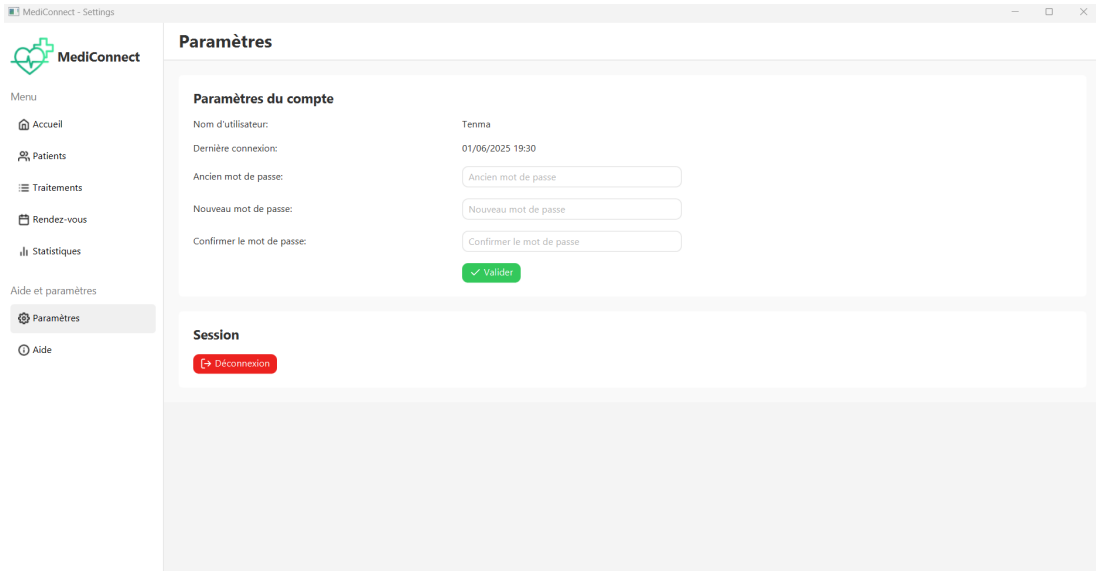


Figure 2.8: Interface des paramétres

La section paramètres offre la possibilité de modifier le mot de passe utilisateur, affiche la dernière date de connexion, et propose un bouton de déconnexion sécurisée.

2.4.9 Exportation des données

L'application permet d'exporter les données au format CSV, incluant les patients, traitements, rendez-vous, etc. Cette fonctionnalité facilite les sauvegardes externes ou les traitements hors application. L'export est accessible via un bouton sur chaque tableau.

2.5 Sécurité et validation

La sécurité applicative et la validation des données sont des aspects essentiels de l'application MediConnect. Des mécanismes ont été mis en place pour garantir la fiabilité des informations saisies par l'utilisateur, prévenir les erreurs de saisie, et limiter les risques d'exploitation malveillante.

La validation côté client (dans le contrôleur JavaFX) permet de garantir que seuls des formats attendus sont transmis à la base de données, tandis que des requêtes SQL paramétrées (PreparedStatement) protègent contre les attaques par injection SQL.

2.5.1 Système d'authentification

Le module d'authentification de MediConnect repose sur un mécanisme simple, mais sécurisé, basé sur une vérification des identifiants dans la base de données. Ce système est implémenté au sein du contrôleur `LoginController` et du contrôleur `RegisterController`.

Connexion :

Lorsqu'un utilisateur tente de se connecter, les informations saisies (nom d'utilisateur et mot de passe) sont d'abord validées localement via la methode `isValidLogin()`. Si elles sont valides, une requête SQL est exécutée :

```
SELECT * FROM utilisateur
WHERE nom_utilisateur = ? AND mot_de_passe = ?;
```

Cette requête est sécurisée grâce à l'utilisation de `PreparedStatement`. Si un enregistrement correspondant est trouvé, l'utilisateur est authentifié et redirigé vers l'interface d'accueil. Ses informations (nom d'utilisateur, rôle, date de connexion) sont encapsulées dans un objet `Utilisateur` , puis stockées dans une classe de session (`SessionManager`) pour être accessibles durant toute la navigation.

Inscription :

Lors de l'inscription, plusieurs validations sont effectuées :

- Le nom d'utilisateur doit suivre une expression régulière stricte : commencer par une lettre, et contenir 3 à 20 caractères.
- Le mot de passe doit comporter au minimum 8 caractères.
- Le mot de passe et sa confirmation doivent être identiques.
- Une spécialité (rôle) doit être sélectionnée.

Une fois ces vérifications réussies, l'unicité du nom d'utilisateur est contrôlée par une requête :

```
SELECT * FROM utilisateur WHERE nom_utilisateur = ?;
```

Si le nom d'utilisateur est disponible, l'insertion est réalisée avec :

```
INSERT INTO utilisateur (nom_utilisateur, mot_de_passe, role)
VALUES (?, ?, ?);
```

Une fois inscrit, l'utilisateur est informé via une alerte graphique, et redirigé automatiquement vers la vue de connexion.

Gestion des erreurs :

L'application MediConnect intègre un mécanisme de gestion des erreurs centralisé, basé sur la classe utilitaire AlertMessage. Chaque fois qu'une anomalie est détectée — comme un champ vide, une donnée invalide, un identifiant incorrect ou une erreur de base de données — un message d'alerte clair est affiché à l'utilisateur. Ce système permet de guider l'utilisateur en temps réel, tout en évitant les blocages ou comportements imprévus dans l'application.

2.5.2 Validation des champs

Lors de la saisie de données via les différents formulaires de MediConnect (patients, rendez-vous, traitements), un ensemble de **règles de validation robustes** est appliqué pour garantir la **cohérence fonctionnelle** et **l'intégrité des informations enregistrées**.

Principales règles de validation :

Les validations couvrent plusieurs aspects :

- **Champs obligatoires** : certains champs doivent impérativement être renseignés, comme :
  - Nom, prénom du patient ;
  - Numéro de sécurité sociale ;
  - Spécialité médicale, date du rendez-vous, etc.

  La méthode de vérification consiste à tester si chaque champ est vide (`.isEmpty()` après un `.trim()`) avant de poursuivre l'enregistrement.

▶ **Contrôle de formats spécifiques** :

- Le **numéro de téléphone** doit contenir uniquement des chiffres (vérification via une expression régulière comme `[0-9]{10}`).
- La **date de naissance** ou toute date (ex. traitement) doit respecter un format logique (`LocalDate`, `DatePicker` contrôlé).
- L'**heure du rendez-vous** est validée avec `LocalTime.parse()` accompagné d'un `DateTimeFormatter`, qui génère une alerte en cas de format incorrect (ex. `12:00`).

▶ **Restrictions logiques** :

- La **date de fin d'un traitement** ne peut être antérieure à sa **date de début**.
- Lors de la planification, la date du rendez-vous ne peut être dans le passé (comparaison avec `LocalDate.now()`).
- Les champs comme `mot de passe` et `confirmation` doivent être strictement identiques.

Exemple d'implémentation :

Le code suivant présente une méthode centralisée pour valider les informations saisies dans un formulaire patient. Elle contrôle plusieurs champs essentiels : *nom*, *prénom*, *date de naissance*, *numéro de sécurité sociale* et *numéro de téléphone*. Chaque champ est vérifié individuellement, elle s'assure que le nom, le prénom et la date de naissance ne sont pas vides. Elle vérifie aussi que le numéro de sécurité sociale contient bien 15 chiffres et que le numéro de téléphone en contient 10.

```
private boolean validatePatientInputs(TextField nom, TextField prenom, DatePicker
    dateNaissance, TextField numeroSecuriteSocialeField, TextField numeroTelephoneField) {
    StringBuilder errorMessage = new StringBuilder();

    // Vérification des champs obligatoires
    if (nom.getText().trim().isEmpty()) {
        errorMessage.append("- Le nom est obligatoire\n");
    }

    if (prenom.getText().trim().isEmpty()) {
        errorMessage.append("- Le prénom est obligatoire\n");
    }

    if (dateNaissance.getValue() == null) {
        errorMessage.append("- La date de naissance est obligatoire\n");
    }

    // Vérification du numéro de sécurité sociale (15 chiffres)
    String nss = numeroSecuriteSocialeField.getText().trim();
    if (nss.isEmpty()) {
        errorMessage.append("- Le numéro de sécurité sociale est obligatoire\n");
    } else if (!nss.matches("[0-9]{15}")) {
        errorMessage.append("- Numéro de sécurité sociale invalide (15 chiffres requis)\n");
    }

    // Vérification du numéro de téléphone (10 chiffres)
    String tel = numeroTelephoneField.getText().trim();
    if (tel.isEmpty()) {
        errorMessage.append("- Le numéro de téléphone est obligatoire\n");
    } else if (!tel.matches("[0-9]{10}")) {
        errorMessage.append("- Numéro de téléphone invalide (10 chiffres requis)\n");
    }

    // Affichage d'une alerte si des erreurs sont détectées
    if (errorMessage.length() > 0) {
        AlertMessage.showErrorAlert(
        "Erreur",
        "Validation échouée",
        "Veuillez corriger les erreurs suivantes :\n" + errorMessage.toString()
```

```
        );
        return false;
    }

    return true;
}
```

Les messages d'erreur sont affichés via la classe `AlertMessage`, offrant un retour immédiat à l'utilisateur en cas de saisie incorrecte. Cette méthode permet de bloquer toute opération tant que les données ne respectent pas les contraintes définies.

2.5.3 Prévention des doublons

Pour éviter les enregistrements redondants ou incohérents, **MediConnect** intègre une stratégie rigoureuse de contrôle d'unicité, particulièrement pour les entités sensibles comme les patients.

Données sensibles vérifiées :

Le numéro de sécurité sociale (`NSS`) joue le rôle d'identifiant unique pour chaque patient. Ce dernier est :

- Déclaré comme `UNIQUE` dans la base de données SQLite via la contrainte au niveau de la table `patient`.
- Vérifié manuellement au moment de l'ajout d'un nouveau patient via une requête de sélection, afin de bloquer toute tentative de duplication.

Vérification avant insertion :

Avant toute tentative d'enregistrement, une requête SQL est exécutée pour rechercher l'existence d'un patient avec le même `NSS` :

```
    SELECT * FROM patient WHERE numero_securite_sociale = ?;
```

Si un résultat est retourné, cela signifie que le patient existe déjà. Dans ce cas, une alerte explicite est déclenchée à l'utilisateur à l'aide de la classe `AlertMessage`.

Exemple d'implémentation :

L'extrait suivant illustre l'utilisation de la requête de vérification avant insertion dans le contrôleur :

```
private boolean ajouterPatientDansDB(Patient patient) {
    String checkSecuSQL = "SELECT * FROM patient WHERE numero_securite_sociale = ?";
    String insertSQL = "INSERT INTO patient (nom, prenom, date_naissance, sexe,
    numero_telephone, numero_securite_sociale, date_creation) VALUES (?, ?, ?, ?, ?, ?,
    datetime('now'))";

    try (Connection connect = Database.connectDB();
    PreparedStatement checkStmt = connect.prepareStatement(checkSecuSQL);
    PreparedStatement insertStmt = connect.prepareStatement(insertSQL, Statement.
    RETURN_GENERATED_KEYS)) {

        checkStmt.setString(1, patient.getNumSecuriteSociale());
        ResultSet result = checkStmt.executeQuery();
        if (result.next()) {
            AlertMessage.showErrorAlert(
            "Erreur",
            "Patient déjà existant",
            "Un patient avec NSS: " + patient.getNumSecuriteSociale() + " existe déjà  !");
            return false;
        }

```

```
19          // Code d'insertion ici (si NSS est unique)
20
21      } catch (SQLException e) {
22          e.printStackTrace();
23      }
24  }
```

2.6 Conclusion

Ce second chapitre a permis de présenter en détail la réalisation fonctionnelle de l'application, depuis la gestion de la base de données jusqu'à l'implémentation des différentes fonctionnalités métier. L'architecture technique repose sur une communication fluide entre l'interface utilisateur, les contrôleurs JavaFX et le moteur de base de données SQLite via JDBC.
Les principales opérations (ajout, modification, suppression, recherche) ont été intégrées de manière ergonomique et sécurisée. Le respect des bonnes pratiques de développement, notamment la séparation des couches (modèle, vue, contrôleur), a permis de garantir un code structuré, lisible et maintenable.
Enfin, des mécanismes de validation, de sécurité et de test ont été mis en œuvre pour assurer la robustesse de l'application. Ces éléments constituent une base solide pour d'éventuelles évolutions futures, comme l'ajout d'un module d'authentification avancée, une synchronisation en ligne, ou une gestion multi-utilisateur plus poussée.
Ce chapitre témoigne donc de la mise en pratique concrète des concepts étudiés en programmation orientée objet, en développement d'interface graphique, et en gestion de base de données.