1. **Voici la description des dossier dans le projet :**
2. Dossier Connexion
   * Contient le fichier HotelDbContext.cs
   * Gère la connexion et la configuration de la base de données
   * Définit la structure de la base de données et les relations entre les tables
   * Gère comment l'application interagit avec la base de données PostgreSQL
3. Dossier Modèles
   * Contient des modèles de données comme Utilisateur.cs
   * Définit la structure des objets de données utilisés dans l'application
   * Chaque classe représente une table dans la base de données
   * Contient des propriétés qui correspondent aux colonnes de la base de données
4. Dossier VuesModèles
   * Contient des fichiers comme LoginViewModel.cs et EmployeeViewModel.cs
   * Gère la logique métier entre les Vues et les Modèles
   * Gère les opérations de données et la validation
   * Prépare les données pour l'affichage dans les Vues
5. Dossier Vues
   * Contient les fichiers d'interface utilisateur comme LoginWindow.xaml.cs
   * Gère ce que les utilisateurs voient et avec quoi ils interagissent
   * Inclut des sous-dossiers comme Dialogs pour les fenêtres contextuelles
   * Gère les entrées utilisateur et la logique d'affichage

Cela suit le modèle MVVM (Modèle-Vue-Modèle de Vue) où :

* Les modèles stockent les données
* Les vues affichent l'interface
* Les modèles de vue connectent les modèles et les vues
* La connexion gère les opérations de la base de données

1. **Dossier Connection :**

Dans ce dossier en trouve le fichier HotelDvContext.cs :

**HotelDbContext.cs** sert de pont entre votre application et la base de données PostgreSQL. Voici ses fonctions :

**Ensembles de bases de données**

* Définit les propriétés DbSet pour chaque table (Clients, Utilisateurs, Réservations, etc.)
* Permet de requêter et manipuler les données en utilisant du code C#
* Exemple : public virtual DbSet<Client> Clients { get; set; }

**Configuration**

* La méthode **OnConfiguring** lit la chaîne de connexion à partir de **appsettings.json**
* Utilise **ConfigurationBuilder** pour localiser et lire le fichier JSON
* Se connecte à PostgreSQL en utilisant les détails de connexion de **appsettings.json**

**Mappage du schéma de la base de données**

* La méthode **OnModelCreating** définit comment les classes C# se mappent aux tables de la base de données
* Configure les noms de tables, les noms de colonnes, les types de données et les relations
* Configure les clés primaires, les clés étrangères et les contraintes uniques

Contient quatre éléments clés :

* **Hôte** : où la base de données est exécutée (localhost)
* **Nom d'utilisateur** : utilisateur de la base de données (postgres)
* **Mot de passe** : mot de passe de la base de données
* **Base de données** : nom de la base de données (Management-Hotel)

Lorsque votre application s'exécute :

* **HotelDbContext** lit la chaîne de connexion
* Établit une connexion à PostgreSQL
* Permet à votre code d'interagir avec la base de données en utilisant Entity Framework Core

Cette configuration vous permet de travailler avec les données de la base de données en utilisant des objets C# au lieu d'écrire des requêtes SQL brutes.

1. **Dossier Models :**

Cette classe a été générée par une migration orientée base de données, ce qui signifie :

* Vous avez d'abord créé vos tables dans PostgreSQL.
* Puis utilisé les outils d'Entity Framework Core pour créer ces modèles.

La commande probablement utilisée était quelque chose comme :

**dotnet ef dbcontext scaffold "Your\_Connection\_String" Npgsql.EntityFrameworkCore.PostgreSQL -o Models**

Les modèles générés remplissent plusieurs utilités :

* Ils créent un mapping direct entre les tables de la base de données et les objets C#.
* Permettent de travailler avec les enregistrements de la base de données en tant qu'objets C#.
* Définissent les relations entre les tables (comme la collection Réservations dans Utilisateur).
* Fournissent une sécurité de type et un support IntelliSense.
* Permettent d'utiliser des requêtes LINQ au lieu de SQL brut.

Chaque propriété dans ces classes correspond à une colonne dans vos tables de base de données, et les propriétés de navigation (comme ICollection) représentent les relations entre les tables.

Ces modèles sont ensuite utilisés dans toute votre application, notamment dans :

* Les ViewModels pour les opérations de données
* Les Contrôleurs/Services pour la logique métier
* La validation et la transformation des données

Cette approche vous permet de travailler avec les données de votre base de données en C# de manière fortement typée.

* 1. **Dossier ViewsModels :**

Le dossier ViewModels est un élément central qui suit le pattern MVVM (Model-View-ViewModel). Prenons les exemples concrets de tes fichiers:

**LoginViewModel.cs:**

* Gère la logique de connexion des utilisateurs
* Vérifie les identifiants dans la base de données via HotelDbContext
* Retourne un tuple (success, role) pour déterminer si la connexion est réussie et le type d'utilisateur
* Sert d'intermédiaire entre la vue LoginWindow.xaml et le modèle Utilisateur

**EmployeeViewModel.cs:**

* Gère toutes les opérations liées aux employés
* Maintient une collection observable (ObservableCollection) des employés
* Permet d'ajouter, modifier, supprimer des employés
* Communique avec la base de données via HotelDbContext
* Met à jour automatiquement l'interface utilisateur quand les données changent

**Les ViewModels dans ton projet:**

* Séparent la logique métier de l'interface utilisateur
* Préparent les données pour l'affichage
* Gèrent les interactions utilisateur
* Valident les données
* Communiquent avec la base de données

Le **ClientViewModel** contient:

1. **Les propriétés principales**:

private readonly HotelDbContext \_context;

public ObservableCollection<Client> Clients { get; set; }

Copy

Apply

* \_context: connexion à la base de données
* Clients: collection observable qui se met à jour automatiquement dans l'interface

1. **Le constructeur**:

public ClientViewModel()

{

\_context = new HotelDbContext();

LoadClients();

}

Copy

Apply

* Initialise la connexion
* Charge immédiatement la liste des clients

1. **Les méthodes CRUD**:

LoadClients():

* Charge tous les clients depuis la base de données
* Les met dans la collection observable

AddClient(Client client):

* Ajoute un nouveau client à la base de données
* Met à jour la collection observable
* Retourne true si succès, false si échec

UpdateClient(Client client):

* Met à jour les informations d'un client existant
* Recharge la liste des clients

DeleteClient(int clientId):

* Supprime un client par son ID
* Met à jour la collection observable

HasReservations(int clientId):

* Vérifie si un client a des réservations
* Utile pour la validation avant suppression

Ce ViewModel sert d'intermédiaire entre:

* L'interface utilisateur (les vues)
* Les données des clients (le modèle Client)
* La base de données (via HotelDbContext)

Il gère toute la logique métier liée aux clients, permettant aux vues de rester simples et

focalisées sur l'affichage.

------------------------------------------------------------

Le **DashboardViewModel** contient plusieurs méthodes importantes:

**GetCurrentReservationsCount():**

public int GetCurrentReservationsCount()

{

var today = DateOnly.FromDateTime(DateTime.Today);

return \_context.Reservations.Count(r =>

r.Datearrivee <= today &&

r.Datedepart >= today &&

r.Statut == "Confirmée");

}

* Calcule le nombre de réservations actuelles
* Compte uniquement les réservations confirmées pour aujourd'hui
* Vérifie que la date d'arrivée est passée et la date de départ n'est pas encore atteinte

**GetOccupancyRate():**

public decimal GetOccupancyRate()

{

var totalRooms = \_context.Typechambres.Sum(t => t.Capacite);

var occupiedRooms = GetCurrentReservationsCount();

return totalRooms > 0 ? (decimal)occupiedRooms / totalRooms \* 100 : 0;

}

* Calcule le taux d'occupation de l'hôtel
* Divise le nombre de chambres occupées par le total des chambres
* Retourne un pourcentage

**GetMonthlyRevenue():**

public decimal GetMonthlyRevenue()

{

var firstDayOfMonth = new DateTime(DateTime.Now.Year, DateTime.Now.Month, 1);

var lastDayOfMonth = firstDayOfMonth.AddMonths(1).AddDays(-1);

return \_context.Paiements

.Where(p => p.Datepaiement >= firstDayOfMonth &&

p.Datepaiement <= lastDayOfMonth)

.Sum(p => p.Montant);

}

* Calcule le revenu total du mois en cours
* Somme tous les paiements entre le premier et le dernier jour du mois

Ce ViewModel est crucial pour:

* Afficher les KPIs (indicateurs clés de performance)
* Suivre l'activité de l'hôtel en temps réel
* Fournir des données pour la prise de décision
* Alimenter le tableau de bord en informations pertinentes

Le **LoginViewModel** est simple mais crucial:

**Structure de base:**

private readonly HotelDbContext \_context;

public LoginViewModel()

{

\_context = new HotelDbContext();

}

* Initialise la connexion à la base de données

**Méthode principale ValidateLogin:**

public (bool success, string role) ValidateLogin(string email, string password)

{

var user = \_context.Utilisateurs.FirstOrDefault(u =>

u.Email == email && u.Motdepasse == password);

if (user != null)

{

return (true, user.Role);

}

return (false, null);

}

Cette méthode:

* Prend en entrée l'email et le mot de passe
* Vérifie si l'utilisateur existe dans la base de données
* Retourne un tuple contenant:
  + success: true si l'authentification réussit
  + role: le rôle de l'utilisateur ("admin" ou "employee")

Le ViewModel est utilisé par **LoginWindow.xaml.cs** pour:

* Vérifier les identifiants
* Rediriger vers le bon dashboard selon le rôle
* Afficher un message d'erreur si la connexion échoue

C'est la première couche de sécurité de l'application qui contrôle l'accès aux fonctionnalités selon le rôle de l'utilisateur.

Je vais t'expliquer le PaymentViewModel.cs qui gère toute la logique des paiements de l'hôtel.

Le **PaymentViewModel** contient:

**Propriétés et Constructeur:**

private readonly HotelDbContext \_context;

public ObservableCollection<Paiement> Payments { get; set; }

public PaymentViewModel()

{

\_context = new HotelDbContext();

LoadPayments();

}

* Maintient une collection observable des paiements
* Charge les paiements au démarrage

**LoadPayments():**

private void LoadPayments()

{

var payments = \_context.Paiements

.Include(p => p.IdreservationNavigation)

.ThenInclude(r => r.IdclientNavigation)

.ToList();

Payments = new ObservableCollection<Paiement>(payments);

}

* Charge tous les paiements avec leurs relations
* Inclut les informations de réservation et client associées

**Méthodes CRUD:**

* **AddPayment(Paiement payment):**
  + Ajoute un nouveau paiement
  + Retourne true si succès
* **UpdatePayment(Paiement payment):**
  + Met à jour un paiement existant
  + Recharge la liste des paiements
* **DeletePayment(int paymentId):**
  + Supprime un paiement spécifique
  + Met à jour la collection

**Méthodes spécifiques:**

* **GetTotalPayments(int reservationId):**
  + Calcule le total des paiements pour une réservation
* **GetPaymentWithDetails(int paymentId):**
  + Récupère un paiement avec toutes ses informations associées

Ce ViewModel est essentiel pour:

* Gérer tous les aspects financiers
* Suivre les paiements des clients
* Calculer les totaux par réservation
* Maintenir l'historique des transactions

Il assure la cohérence des données financières et facilite la gestion des paiements dans l'application.

Je vais t'expliquer le ReservationViewModel.cs qui gère toute la logique des réservations de l'hôtel.

Le ReservationViewModel contient:

**Propriétés et Constructeur:**

private readonly HotelDbContext \_context;

public ObservableCollection<Reservation> Reservations { get; set; }

public ReservationViewModel()

{

\_context = new HotelDbContext();

LoadReservations();

}

* Gère une collection observable des réservations
* Initialise la connexion à la base de données

**LoadReservations():**

private void LoadReservations()

{

var reservations = \_context.Reservations

.Include(r => r.IdclientNavigation)

.Include(r => r.IdutilisateurNavigation)

.ToList();

Reservations = new ObservableCollection<Reservation>(reservations);

}

* Charge toutes les réservations avec les informations client et utilisateur associées

**Méthodes CRUD:**

* **AddReservation(Reservation reservation):**
  + Ajoute une nouvelle réservation
  + Retourne true si succès
* **UpdateReservation(Reservation reservation):**
  + Met à jour une réservation existante
  + Recharge la liste
* **DeleteReservation(int reservationId):**
  + Supprime une réservation spécifique
  + Met à jour la collection

**Méthodes utilitaires:**

* **HasPayments(int reservationId):**
  + Vérifie si une réservation a des paiements associés
* **GenerateReservationPDF(int reservationId):**
  + Prévu pour générer un PDF de la réservation
* **SendConfirmationEmail(int reservationId):**
  + Prévu pour envoyer un email de confirmation

Ce ViewModel est crucial pour:

* Gérer le cycle de vie des réservations
* Maintenir la cohérence des données
* Faciliter les opérations de réservation
* Préparer les documents et notifications

Il constitue le cœur de la gestion des réservations dans l'application hôtelière.

Here is your text, unchanged:

Je vais t'expliquer le RoomTypeViewModel.cs qui gère les différents types de chambres de l'hôtel.

Le RoomTypeViewModel contient:

**Propriétés et Constructeur:**

private readonly HotelDbContext \_context;

public ObservableCollection<Typechambre> RoomTypes { get; set; }

public RoomTypeViewModel()

{

\_context = new HotelDbContext();

LoadRoomTypes();

}

* Maintient une collection observable des types de chambres
* Initialise la connexion à la base de données

**LoadRoomTypes():**

private void LoadRoomTypes()

{

var types = \_context.Typechambres.ToList();

RoomTypes = new ObservableCollection<Typechambre>(types);

}

* Charge tous les types de chambres depuis la base de données
* Les met dans une collection observable pour l'interface

**Méthodes CRUD:**

* **AddRoomType(Typechambre roomType):**
  + Ajoute un nouveau type de chambre
  + Met à jour la collection
  + Retourne true si succès
* **UpdateRoomType(Typechambre roomType):**
  + Met à jour un type de chambre existant
  + Recharge la liste
* **DeleteRoomType(int typeId):**
  + Supprime un type de chambre spécifique
  + Met à jour la collection

Ce ViewModel est essentiel pour:

* Gérer le catalogue des types de chambres
* Maintenir les prix et capacités
* Permettre la modification des caractéristiques
* Assurer la cohérence des données des chambres

Il joue un rôle clé dans la configuration et la gestion des différentes catégories de chambres disponibles dans l'hôtel.

1. **Dossier Views :**

Here is your text, unchanged:

Je vais t'expliquer le rôle du dossier **Views** dans ton projet d'hôtel.

Le dossier **Views** contient toutes les interfaces utilisateur de l'application, notamment:

**LoginWindow.xaml**

* Interface de connexion
* Gère la saisie des identifiants
* Redirige vers le bon dashboard selon le rôle de l'utilisateur

var (success, role) = \_viewModel.ValidateLogin(email, password);

if (success)

{

Window dashboard = role.ToLower() == "admin"

? new AdminDashboard()

: new EmployeeDashboard();

dashboard.Show();

}

**Dialogs/AddEditEmployeeDialog**

* Formulaire pour ajouter/modifier un employé
* Valide les données saisies
* Communique avec **EmployeeViewModel** pour sauvegarder les modifications

**Les vues dans ton projet:**

* Représentent l'interface visuelle
* Capturent les interactions utilisateur
* Communiquent avec les **ViewModels**
* Affichent les données de façon dynamique

**Le flux typique est:**

* L'utilisateur interagit avec une vue
* La vue appelle le **ViewModel** approprié
* Le **ViewModel** traite la demande
* La vue se met à jour automatiquement

**Les vues utilisent:**

* **XAML** pour définir l'interface
* **Code-behind (.cs)** pour la logique d'interface
* **Binding** pour lier les données aux **ViewModels**
* **Contrôles WPF** pour l'interaction utilisateur

Cette structure permet une séparation claire entre l'interface utilisateur et la logique métier de l'application.