# Documentation de l'Architecture du Projet RestaurantManagement

## Introduction

Le projet **RestaurantManagement** adopte une architecture **Modèle-Vue-Contrôleur (MVC)** combinée à des principes avancés de conception logicielle, notamment le \*\*Pattern Singleton\*\*, la **programmation concurrente** via un **ThreadPool**, et des **factories** pour la création dynamique des objets.

L'objectif principal est de fournir une gestion efficace des ressources (tables, plats, utilisateurs, etc.) dans un restaurant à travers une application robuste et bien structurée.

## Structure de l'Architecture

L'application est divisée en plusieurs modules organisés selon l'architecture MVC :

```

/RestaurantManagement

/Model

- Chef\_de\_rang.h

- Client.h

- Commande.h

- Commis\_de\_salle.h

- Maitre\_d\_hotel.h

- \_personne.h

- Plat.h

- Position.h

- Serveur.h

- Table.h

- PostgreSQLConnection.cpp

- PostgreSQLConnection.h

/Controller

- ControleurService.h

- Factory.h

- ThreadPool.cpp

- ThreadPool.h

/View

- Formulaire.cpp

- MainWindow.cpp

- MenuWindow.cpp

- RestaurantDashboard.cpp

- main.cpp

- CMakeLists.txt

```

## Détails des Composants : Architecture MVC

1. Modèle (Model)

La couche **Modèle** est le cœur de l'application. Elle gère la logique métier, les interactions avec la base de données, et les règles métiers.

**Responsabilités**

- Gestion des données métier (employés, clients, commandes, etc.).

- Interaction avec la base de données PostgreSQL.

- Implémentation des règles spécifiques au domaine (restaurant).

**Classes** **Principales**

**1**. Chef\_de\_rang , Serveur , Maitre\_d\_hotel , Commis\_de\_salle :

- Représentent les rôles des employés avec des responsabilités spécifiques, comme la gestion des commandes ou des clients.

2. Client :

- Contient les informations relatives aux clients (nom, réservations, préférences).

### 3. Commande :

- Représente les commandes effectuées par les clients, incluant les plats et leur statut.

### 4. Plat:

- Gère les informations sur les plats du menu (nom, description, prix, disponibilité).

### 5. Position :

- Représente les informations sur la localisation des tables dans le restaurant.

### 6. Table:

- Modélise les tables du restaurant, leur statut (libre/occupée) et leur capacité.

### 7. personne :

- Classe parente contenant des informations génériques sur une personne (nom, âge), à partir de laquelle les employés et les clients héritent.

### Gestion des Données : PostgreSQLConnection

- **Singleton**: Gère la connexion unique à la base de données PostgreSQL.

- **Thread-safe** : Implémente des mécanismes pour garantir une utilisation concurrente.

- **Rôle** : Assure les opérations CRUD sur les données.

## 2. Contrôleur (Controller)

La couche \*\*Contrôleur\*\* gère la logique applicative et agit comme intermédiaire entre le Modèle et la Vue.

**Responsabilités**

- Traitement des actions utilisateur.

- Mise à jour des données dans le Modèle.

- Transmission des informations mises à jour à la Vue.

**Composants Principaux**

### 1. ControleurService :

- Interface pour interagir avec les services métiers.

- Exemples :

- Gérer les commandes.

- Réserver/libérer des tables.

- Accéder aux données des plats.

### 2. Factory:

- Implémente le pattern Factory pour créer dynamiquement des objets tels que les plats ou les tables.

### 3. ThreadPool:

- Implémente un pool de threads pour exécuter des tâches lourdes (requêtes SQL, calculs complexes) de manière efficace et non-bloquante.

## 3. Vue (View)

La couche **Vue** gère l’interface utilisateur. Elle présente les données et capture les interactions utilisateur.

**Responsabilités**

- Affichage des informations provenant du Modèle.

- Interaction avec l’utilisateur (boutons, formulaires, tableaux).

**Composants Principaux**

1. MainWindow:

- Fenêtre principale de l’application.

- Point d’entrée pour naviguer dans les différentes fonctionnalités.

2. MenuWindow :

- Affiche les plats disponibles avec leurs détails.

3.Formulaire :

- Gère les formulaires pour ajouter/modifier des données.

4. RestaurantDashboard :

- Vue centralisée pour surveiller l’état des tables, commandes, et employés.

**Design Patterns Utilisés**

1. Singleton (PostgreSQLConnection)

- Garantit une connexion unique à la base de données.

- Centralise la gestion des transactions SQL.

2. Factory

- Permet la création dynamique d’objets.

- Exemple : Création de nouveaux plats ou employés.

3. Programmation Concurrente (ThreadPool)

- Optimise l’exécution des opérations lourdes.

- Garantit une interface réactive.

---

**Scénarios Fonctionnels**

1. L'utilisateur ouvre MainWindow :

- La Vue appelle le `ControleurService` pour obtenir les informations des tables via le Modèle.

- Les informations sont affichées dans `RestaurantDashboard`.

2. Une commande est passée :

- L'utilisateur, dans `MenuWindow`, sélectionne les plats souhaités.

- Le `ControleurService` enregistre la commande via le Modèle (classe `Commande` sauvegardée en PostgreSQL).

3. Programmation Concurrente :

- Les requêtes lourdes vers PostgreSQL utilisent `ThreadPool`, assurant des réponses rapides et une interface non-bloquante.

---

**Forces de l'Architecture**

1. Séparation claire des responsabilités :

- Chaque couche a un rôle dédié, ce qui simplifie la maintenance.

2. Réutilisabilité et extensibilité:

- Les composants sont modulaires, facilitant l'ajout de nouvelles fonctionnalités.

3. Performance :

- Gestion efficace des ressources grâce à la programmation concurrente.

4. Centralisation :

- Utilisation de Singleton pour la gestion unique de la connexion.

# Conclusion

Cette architecture, basée sur MVC et enrichie par des patterns avancés comme Singleton, Factory et la programmation concurrente, offre une solution robuste pour gérer efficacement les opérations d'un restaurant. Elle garantit une maintenance aisée, des performances élevées, et une expérience utilisateur fluide.