Ministère de l'Enseignement
Supérieur et de la Recherche
Scientifique
Institut International de Technologie à
Sfax

IIT
INSTITUT
INTERNATIONAL
TECHNOLOGIE
Université Nord Américaine Privée

وزارة التعليم العالي والبحث العلمي
المدرسة العليا الدولية الخاصة للتكنولوجيا
بصفاقس

Rapport de Projet

Élaboré par :
Chedy Chaaben
Taieb Jemal

Développement d'une Application de Commande en Ligne pour Restaurant

Enseignant : **M. Fahmi KALLEL**

Année universitaire : 2023/2024

Table des matières

Introduction générale

Dans un monde de plus en plus connecté et numérique, la gestion efficace des systèmes d'information est essentielle pour garantir la bonne marche des entreprises, y compris celles du secteur de la restauration. L'automatisation des processus, la sécurisation des données et l'amélioration de l'expérience utilisateur deviennent des éléments clés pour répondre aux défis contemporains. Dans ce contexte, la gestion d'un restaurant ne se limite plus à des tâches manuelles, mais s'appuie sur des solutions logicielles intelligentes pour optimiser les opérations quotidiennes, comme la gestion des commandes et du stock des produits.

Ce projet de développement d'une application web pour un restaurant se focalise sur l'utilisation d'une API, offrant ainsi une meilleure flexibilité et sécurité, plutôt que sur un site web traditionnel.

L'objectif est de permettre une gestion fluide des commandesau sein du système tout en garantissant un accès sécurisé aux différentes fonctionnalités de l'application.

Afin d'illustrer et de structurer efficacement cette application, des diagrammes de classe et des cas d'utilisation ont été élaborés pour fournir une vue d'ensemble des entités et de leurs interactions dans l'application. En outre, une série d'endpoints API ont été développés pour permettre la communication entre le serveur et le client, facilitant ainsi les opérations de gestion des commandes.

Chapitre 1

Authentification avec JWT (JSON Web Token)

L'authentification par **JWT (JSON Web Token)** est une méthode largement utilisée pour sécuriser les API web. Un **JWT** est un jeton compact et autonome qui permet de transmettre de manière sécurisée des informations entre un client et un serveur sous forme d'un objet JSON. Ce jeton est signé numériquement, généralement avec un algorithme comme HMAC ou RSA, afin de garantir l'intégrité des données et de vérifier l'authenticité de l'émetteur.

L'authentification par JWT est particulièrement adaptée aux systèmes **stateless**, car elle permet de ne pas avoir besoin de stocker de session côté serveur. En effet, contrairement à une session traditionnelle où des informations sont conservées sur le serveur, le JWT contient toutes les informations nécessaires à l'identification de l'utilisateur et à l'autorisation d'accès aux ressources protégées.

Les principaux avantages de l'utilisation de JWT sont les suivants :

— **Stateless** : Il n'est pas nécessaire de maintenir l'état de la session sur le serveur, ce qui simplifie la gestion de la scalabilité de l'application.

— **Sécurisé** : Le JWT est signé numériquement, ce qui garantit que son contenu n'a pas été modifié et qu'il provient d'une source fiable.

— **Flexible** : Le JWT peut contenir diverses informations utiles, telles que les rôles d'utilisateur, la date d'expiration et d'autres attributs.

1.1 Structure d'un JWT

Un JWT se compose de trois parties distinctes, séparées par des points (`.`) :

— **Header** : Spécifie le type de token (JWT) et l'algorithme de signature utilisé (par exemple, HMAC SHA256 ou RSA).

— **Payload** : Contient les informations ou **claims**, comme l'ID utilisateur, le rôle de l'utilisateur, et la date d'expiration du token.

— **Signature** : Générée en utilisant le Header, le Payload et une clé secrète partagée entre le client et le serveur, afin de garantir l'intégrité et l'authenticité du jeton.

Voici un exemple de structure d'un JWT :

```
<Header>.<Payload>.<Signature>
```

1.2 Pourquoi utiliser JWT ?

L'utilisation de JWT présente plusieurs avantages qui en font un choix idéal pour l'authentification et l'autorisation dans les applications modernes :

— **Authentification sans état** : Le serveur n'a pas besoin de stocker de sessions, ce qui simplifie la gestion des ressources et augmente la scalabilité de l'application.

— **Sécurité** : Le jeton est signé numériquement, ce qui garantit son intégrité. Il peut également être chiffré pour protéger les informations sensibles qu'il contient.

— **Flexibilité** : Le JWT permet d'intégrer divers types de données, telles que les rôles ou les permissions, directement dans le jeton, facilitant ainsi la gestion des accès à différentes ressources.

1.3 Fonctionnement dans l'application

Voici les étapes du fonctionnement de l'authentification avec JWT dans notre application :

1. **Lors de la connexion de l'utilisateur** : Lorsque l'utilisateur saisit son nom d'utilisateur et son mot de passe dans l'interface de connexion, ces informations sont envoyées au backend via une requête HTTP. Si les informations sont valides, le backend génère un JWT contenant des informations telles que l'ID de l'utilisateur, son rôle et une date d'expiration du jeton. Ce jeton est renvoyé au client.
2. **Stockage du JWT côté client** : Une fois le JWT généré et renvoyé, le frontend (le client) stocke ce jeton. Dans notre application, le jeton est généralement stocké dans le `localStorage` du navigateur, bien qu'il soit également possible de l'utiliser dans des cookies pour certaines configurations. Ce stockage permet au client de conserver le jeton entre les sessions et d'envoyer ce jeton pour chaque requête ultérieure.
3. **Envoi du JWT dans les requêtes suivantes** : Pour chaque requête suivante nécessitant une authentification (par exemple, pour consulter le menu ou passer une commande), le client inclut le JWT dans l'en-tête `Authorization` de la requête HTTP. Le format de cet en-tête est :

   ```
   Authorization: Bearer <token>
   ```

4. **Validation du token par le serveur** : À chaque requête, le serveur vérifie la validité du jeton envoyé par le client. Cela se fait à l'aide d'un middleware sur le backend. Ce middleware décode le JWT, vérifie sa signature à l'aide de la clé secrète, et s'assure que le jeton n'a pas expiré. Si le jeton est valide, l'accès à la ressource demandée est accordé. En revanche, si le jeton est invalide ou expiré, le serveur renvoie une erreur d'authentification (généralement avec le code HTTP 401).

Résumé du flux d'authentification :

1. L'utilisateur se connecte avec ses identifiants.
2. Si l'authentification réussit, un JWT est généré et envoyé au client.
3. Le client stocke ce JWT et l'envoie avec chaque requête suivante.
4. Le backend valide le token et permet l'accès aux ressources protégées.

Ce flux d'authentification garantit une gestion sécurisée et scalable des utilisateurs dans l'application, sans la nécessité de maintenir des sessions sur le serveur.

Chapitre 2

Repository Pattern

Le **Repository Pattern** est un design pattern qui permet de créer une couche d'abstraction entre la logique métier d'une application et la persistance des données, généralement une base de données. Ce modèle se situe entre les couches de logique métier (services) et la couche d'accès aux données (base de données ou autre source de données), et permet de centraliser l'accès aux données dans une ou plusieurs classes appelées "repositories".

L'idée principale du **Repository Pattern** est de masquer les détails de l'implémentation de l'accès aux données (requêtes SQL, appels API, etc.) tout en offrant une interface simple et cohérente pour interagir avec ces données. Cela simplifie la gestion des requêtes et la maintenance de l'application en offrant une structure flexible pour effectuer des opérations de lecture et d'écriture, tout en maintenant la séparation des responsabilités.

Le **Repository Pattern** offre plusieurs avantages :

— **Séparation des responsabilités** : La logique métier et l'accès aux données sont séparés, ce qui rend le code plus propre et plus maintenable.

— **Facilité de test unitaire** : Le pattern permet de simuler facilement les interactions avec la base de données grâce à des interfaces et des mocks, ce qui facilite la réalisation de tests unitaires.

— **Abstraction** : Il offre un niveau d'abstraction qui permet de changer la source de données sans affecter le reste de l'application.

En résumé, le **Repository Pattern** aide à maintenir une architecture claire, à réduire la complexité du code, et à rendre l'application plus flexible et testable.

2.1 Implémentation du Repository Pattern dans notre application

Dans notre application, le Repository Pattern a été utilisé pour gérer l'accès aux données, en particulier pour les entités telles que les les commandes, les produits, etc. Pour chaque entité, un **repository** spécifique a été créé, ce qui permet de centraliser toutes les opérations de base (CRUD - Create, Read, Update, Delete) liées à la gestion des données.

2.1.1 Création des interfaces de repository

Nous avons commencé par définir des interfaces de repository qui décrivent les méthodes nécessaires pour interagir avec chaque type de donnée. Par exemple, pour la gestion des produit, nous avons créé une interface `IProductRepository` qui définit les méthodes de base comme `Add()`, `Get()`, `GetByName()`, `Edit()` et `Delete()`. Voici un exemple d'interface pour le repository des utilisateurs :

```
public interface IProductRepository
{
    Task<List<Product>> GetAll();

    Task<Product> Add(Product prod);

    Task<Product> Get(int id);

    Task<Product> GetByName(string nom);
```

```
    Task<Product> Edit(Product prod);

    Task Delete(int id);
}
```

2.1.2 Implémentation des repositories

Ensuite, pour chaque interface de repository, une classe concrète a été créée pour implémenter la logique d'accès aux données. Par exemple, la classe `ProductRepository` implémente l'interface `IProductRepository` et contient les méthodes spécifiques pour accéder à la base de données à l'aide de `Entity Framework Core`, une technologie ORM utilisée dans notre application.

Voici un exemple d'implémentation d'un repository pour l'entité `Product` :

```
public class ProductRepository : IProductRepository
{
    private readonly AppDbContext context;
    public ProductRepository(AppDbContext context)
    {
        this.context = context;
    }

    public async Task<Product> Add(Product prod)
    {
        var result = await context.Products.AddAsync(prod);
        await context.SaveChangesAsync();
        return result.Entity;
    }


    public async Task<List<Product>> GetAll()
    {
        List<Product> Products = await context.Products
                    .Include(c => c.Ingredients)
                    .Include(c => c.Supplements)
                    .Include(c => c.Images)
                    .ToListAsync();
        return Products;
    }

    public async Task<Product> Get(int Id)
    {
        return await context.Products
                    .Include(c => c.Ingredients)
                    .Include(c => c.Supplements)
                    .Include(c => c.Images)
                    .FirstOrDefaultAsync(p => p.Id == Id);
    }

    public async Task<Product> GetByName(string Name)
    {
        return await context.Products
                    .Include(c => c.Ingredients)
                    .Include(c => c.Supplements)
                    .Include(c => c.Images)
                    .FirstOrDefaultAsync(p => p.Name == Name);
    }
```

```
    public async Task<Product> Edit(Product prod)
    {
        context.Products.Update(prod);
        await context.SaveChangesAsync();
        return prod;
    }
    public async Task Delete(int id)
    {
        var prod = await context.Products.FindAsync(id);
        context.Products.Remove(prod);
        await context.SaveChangesAsync();
    }

}
```

Cette classe `ProductRepository` contient toutes les opérations nécessaires pour manipuler les données liées à l'entité `Product` dans la base de données. Nous utilisons ici `Entity Framework Core` pour interagir avec la base de données (Code First).

2.1.3 Injection de dépendances

Afin de permettre à notre application de bénéficier du Repository Pattern, nous avons utilisé l'injection de dépendances pour injecter les repositories dans les services ou contrôleurs où ils sont nécessaires. Cela permet de séparer les responsabilités et de faciliter les tests unitaires.

Voici un exemple de configuration de l'injection de dépendances dans `Program.cs` :

```
builder.Services.AddScoped<IProductRepository, ProductRepository>();
builder.Services.AddScoped<ICategoryRepository, CategoryRepository>();
builder.Services.AddScoped<ICartRepository, CartRepository>();
builder.Services.AddScoped<IOrderRepository, OrderRepository>();
builder.Services.AddScoped<IDrinkRepository, DrinkRepository>();
builder.Services.AddScoped<IIngredientRepository, IngredientRepository>();
builder.Services.AddScoped<ISupplementRepository, SupplementRepository>();
builder.Services.AddScoped<IImageRepository, ImageRepository>();
builder.Services.AddScoped<IProductOfTheDayRepository, ProductOfTheDayRepository>();
builder.Services.AddScoped<IProductCartRepository, ProductCartRepository>();
builder.Services.AddScoped<IProductCartIngredientRepository, ProductCartIngredientRepository>
builder.Services.AddScoped<IProductCartSupplementRepository, ProductCartSupplementRepository>
builder.Services.AddScoped<IDrinkCartRepository, DrinkCartRepository>();
```

Cette configuration permet à `ASP.NET Core` de résoudre automatiquement les dépendances lors de la création d'instances de contrôleurs ou de services. Par exemple, chaque fois qu'un contrôleur a besoin d'un `IProductRepository`, `ASP.NET Core` l'injectera automatiquement.

2.2 Avantages du Repository Pattern

L'implémentation du Repository Pattern dans notre application présente plusieurs avantages notables :

- **Séparation des responsabilités** : Le Repository Pattern aide à maintenir une séparation claire entre la logique métier (services) et la gestion des données (repositories). Chaque classe se concentre sur une seule responsabilité, ce qui améliore la lisibilité et la maintenance du code.
- **Centralisation de l'accès aux données** : Toutes les opérations liées à l'accès aux données sont regroupées dans les repositories, ce qui centralise et simplifie la gestion des requêtes. Si nous devons changer la manière d'interagir avec la base de données (par exemple, changer de technologie ORM), nous n'aurons qu'à modifier les repositories sans affecter la logique métier.

— **Extensibilité** : Le Repository Pattern facilite l'ajout de nouvelles fonctionnalités et entités à l'application. En ajoutant simplement de nouvelles interfaces et classes de repository, nous pouvons gérer de nouvelles entités sans perturber les autres parties de l'application.

En conclusion, le Repository Pattern nous a permis de concevoir une architecture propre et flexible, en séparant clairement la logique métier de la gestion des données dans notre application.

Chapitre 3

Conception

3.1 Diagramme de classe

Le diagramme de classe permet de visualiser les principales entités de l'application et leurs relations.
Dans notre application, les entités principales incluent Utilisateur, Commande, Panier, Produit, Boisson, Ingredient, Supplement, ImageProduit, ProduitDansPanier, BoissonDansPanier, IngredientDansProduitDansPanier, SupplementDansProduitDansPanier, Categorie, Produit du jour.
Ces entités sont liées entre elles par des relations spécifiques, qui seront représentées sous forme de classes et d'associations.

Voici une explication des classes et de leurs relations dans notre modèle :

— **AspNetUsers** : Un utilisateur dans le système `ASP.NET Identity`, représenté dans la table `AspNetUsers`, peut être un client ou un administrateur. Pour faire la distinction, il suffit d'examiner la table `AspNetRoles` et de vérifier si l'utilisateur est associé au rôle admin via une relation dans la table de jonction entre `AspNetUsers` et `AspNetRoles`.

Le diagramme de classe Users peut être représenté de la manière suivante :

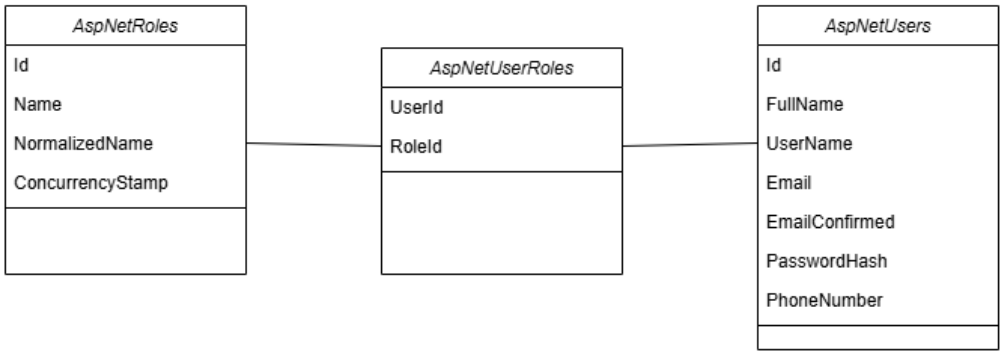

FIGURE 3.1 : Diagramme de classe - Identity User

— **Product** : La table `Product` représente un article destiné à la consommation dans le restaurant.
— **Category** : La table `Category` permet de classer les produits en différentes catégories afin d'organiser efficacement le menu.
— **Ingredient** : La table `Ingredient` centralise les informations relatives aux ingrédients utilisés dans les produits. Chaque ingrédient est identifié par un nom. Cette table permet d'associer des ingrédients spécifiques à chaque produit via une relation avec la table `Product`.
— **Supplement** : La table `Supplement` contient les informations sur les suppléments associés aux produits. Chaque supplément est défini par un nom et un prix. Cette table facilite la gestion centralisée des suppléments et leur association aux produits via une relation avec la table `Product`.

— **Image** : La table `Image` stocke les informations relatives aux images des produits du restaurant. Chaque image peut représenter un plat ou un produit spécifique, afin d'améliorer la présentation visuelle du menu.
— **Drink** : La table `Drink` répertorie les boissons disponibles dans le restaurant. Elle contient des informations telles que le nom, le prix et d'autres caractéristiques pertinentes. Chaque boisson peut être ajoutée à un panier via la table `DrinkCart`, permettant ainsi au client de sélectionner une boisson en complément de sa commande.
— **ProductOfTheDay** : La table `ProductOfTheDay` met en avant un produit spécifique du menu chaque jour, comme une offre spéciale ou un plat du jour. Elle est liée à la table `Product` via un identifiant (`ProductId`).

Le diagramme de classe reliée au gestion du stock peut être représenté de la manière suivante :

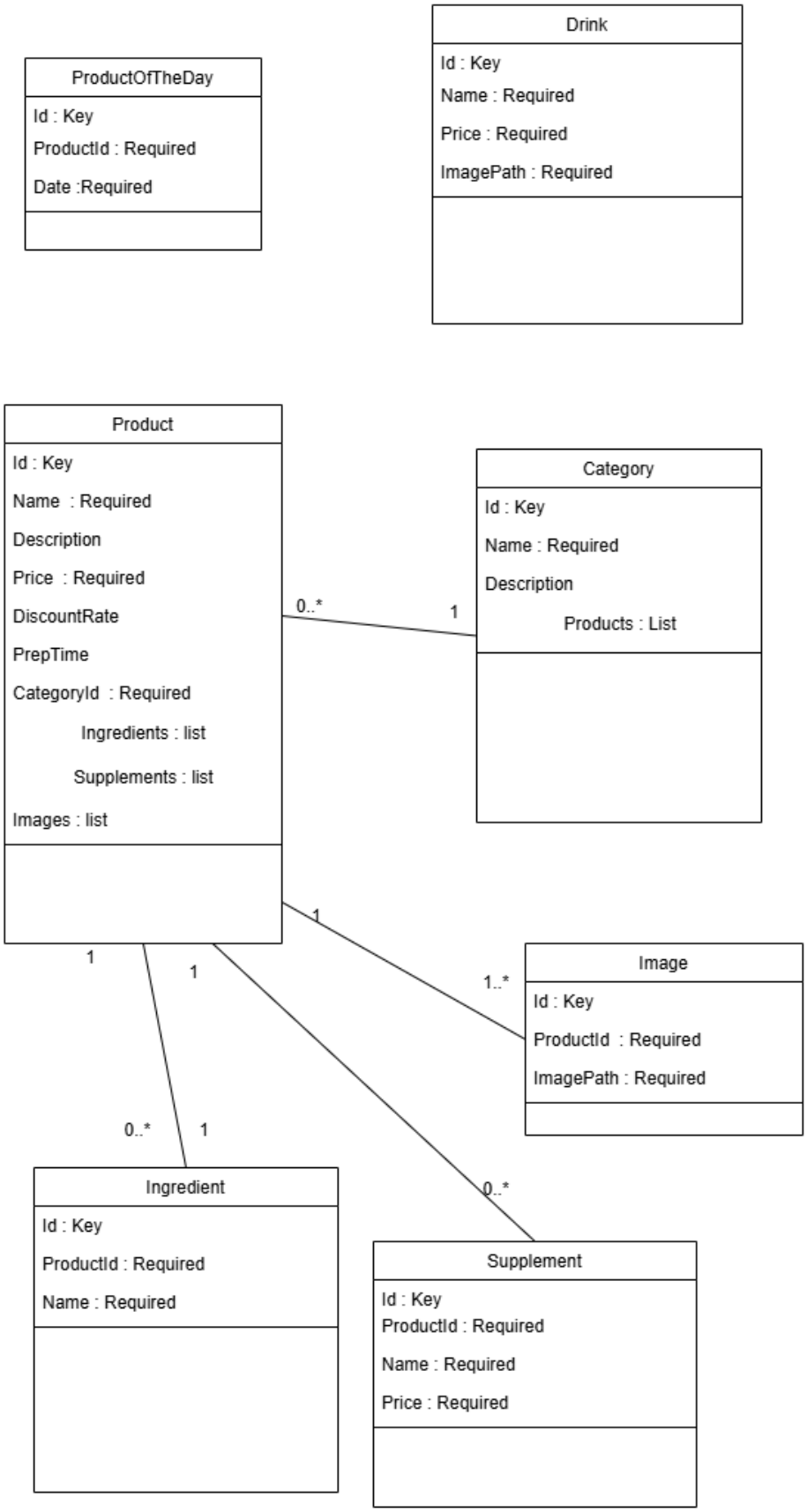

FIGURE 3.2 : Diagramme de classe - Gestion du stock

— **Cart** : La table `Cart` représente le panier, qui regroupe l'ensemble des produits et boissons sélectionnés par un client avant la finalisation de la commande.
— **Order** : La table `Order` représente une commande passée par un client. Elle contient les informations essentielles telles que l'identifiant de la commande, le statut, la date, un message, et l'identifiant du panier associé.
— **ProductCart** : La table `ProductCart` est utilisée pour gérer les produits ajoutés au panier d'un client. Elle enregistre chaque produit, sa quantité ainsi que ses options supplémentaires (telles que `ProductCartIngredient` et `ProductCartSupplement`) lors de la commande. Cette table est liée à la table `Cart`, ce qui permet de suivre précisément les produits associés à chaque panier.
— **ProductCartIngredient** : La table `ProductCartIngredient` stocke les ingrédients ajoutés aux produits présents dans le panier.
— **ProductCartSupplement** : La table `ProductCartSupplement` stocke les suppléments ajoutés aux produits présents dans le panier.
— **DrinkCart** : La table `DrinkCart` gère la relation entre les boissons et les commandes dans le panier.

Le diagramme de classe reliée au commande et panier peut être représenté de la manière suivante :

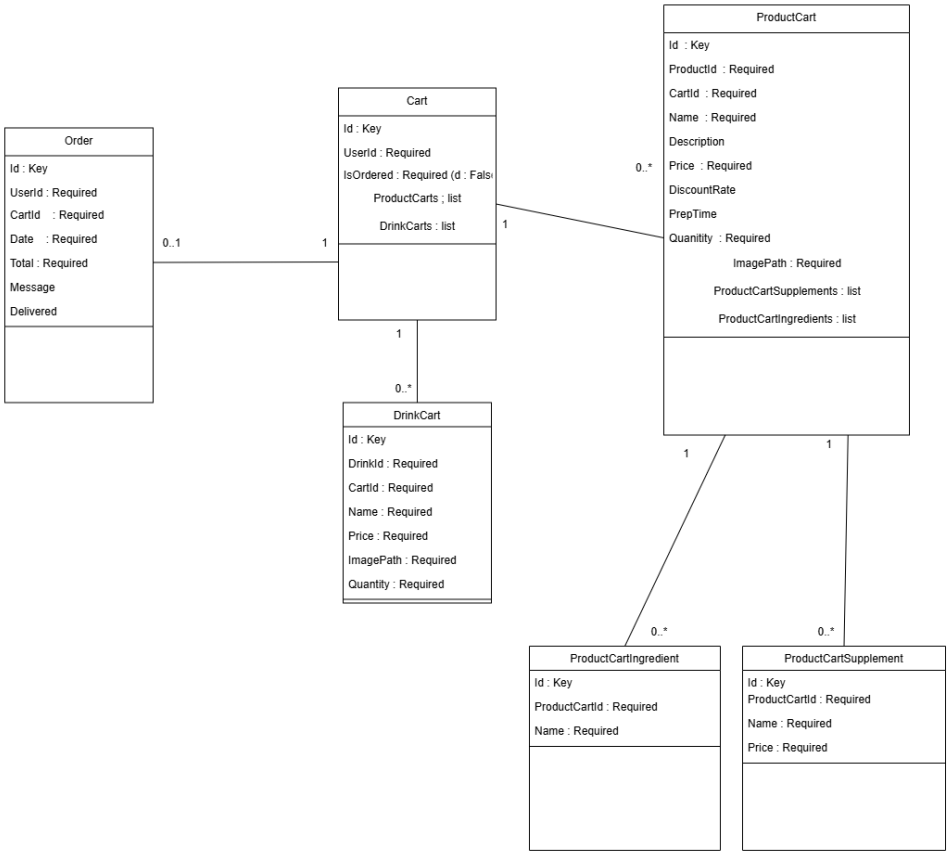

FIGURE 3.3 : Diagramme de classe - commande et panier

3.2 Diagramme des cas d'utilisation

Le diagramme des cas d'utilisation illustre les différentes interactions entre les utilisateurs et l'application, en distinguant les fonctionnalités accessibles à chaque type d'utilisateur. L'application est structurée en deux parties principales : Client et Administrateur.

Client

Le diagramme de cas d'utilisation peut être représenté de la manière suivante :

FIGURE 3.4 : Diagramme de cas d'utilisation - Client

Admin

Le diagramme de cas d'utilisation peut être représenté de la manière suivante :

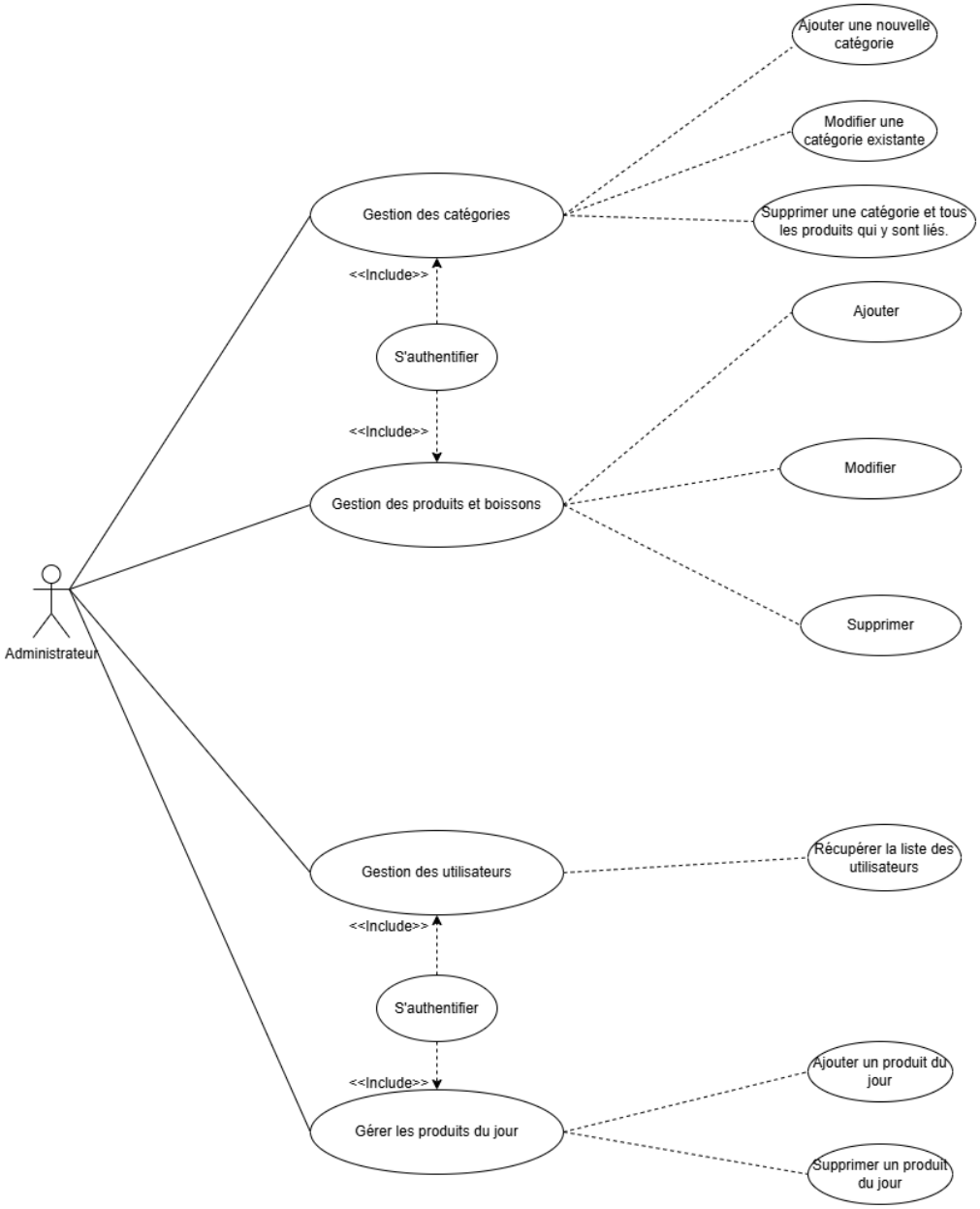

FIGURE 3.5 : Diagramme de cas d'utilisation - Admin

Conclusion

Les diagrammes de classe et de cas d'utilisation sont essentiels pour modéliser la structure et le comportement de l'application. Ils aident à mieux comprendre les relations entre les entités et les actions possibles pour chaque type d'utilisateur, facilitant ainsi le développement et la maintenance de l'application.

3.3 Diagrammes de Séquence

Les diagrammes de séquence permettent de représenter le déroulement chronologique des interactions entre les utilisateurs (Client et Administrateur) et le système. Ils décrivent l'échange de messages entre les différents composants de l'application pour chaque fonctionnalité clé.

Pour notre application, trois diagrammes principaux sont présentés : l'Authentification, la creation d'un compte client et l'Ajout de produit au panier.

Authentification

Le diagramme ci-dessous représente les interactions permettant à un utilisateur (Client ou Administrateur) de s'authentifier dans l'application :

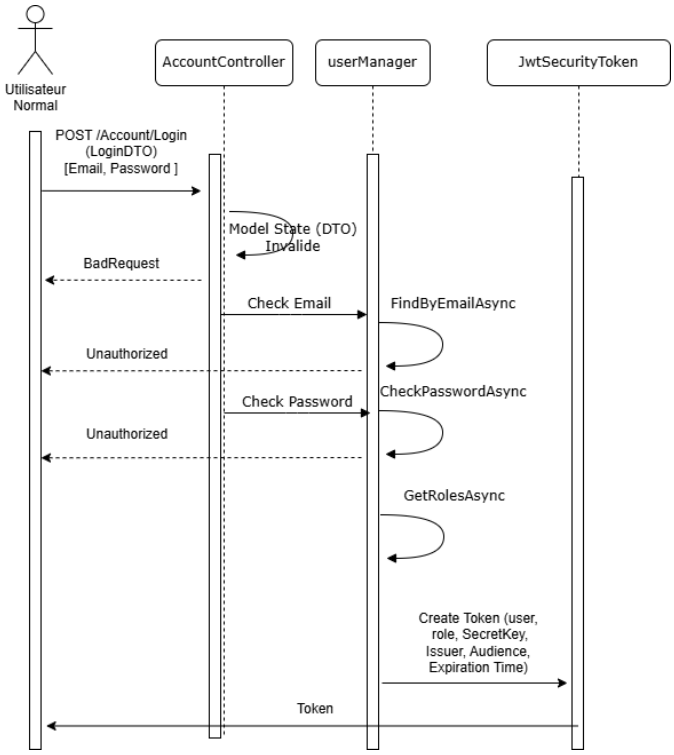

Figure 3.6 : Diagramme de séquence - Authentification

Création du compte client

Le diagramme ci-dessous représente les différentes étapes permettant à un client de créer un compte dans l'application :

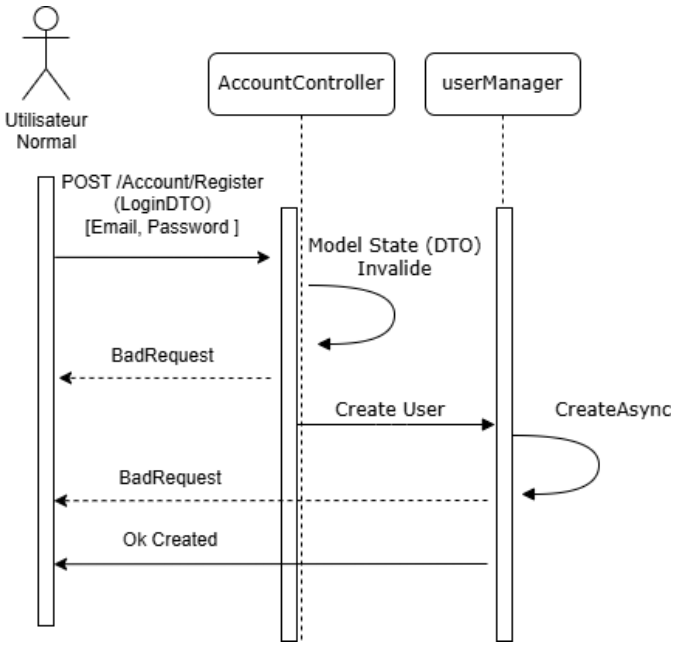

FIGURE 3.7 : Diagramme de séquence - Création de compte client

Ajout d'un produit au panier du client

Le diagramme suivant illustre les interactions liées à l'ajout d'un produit dans le panier du client :

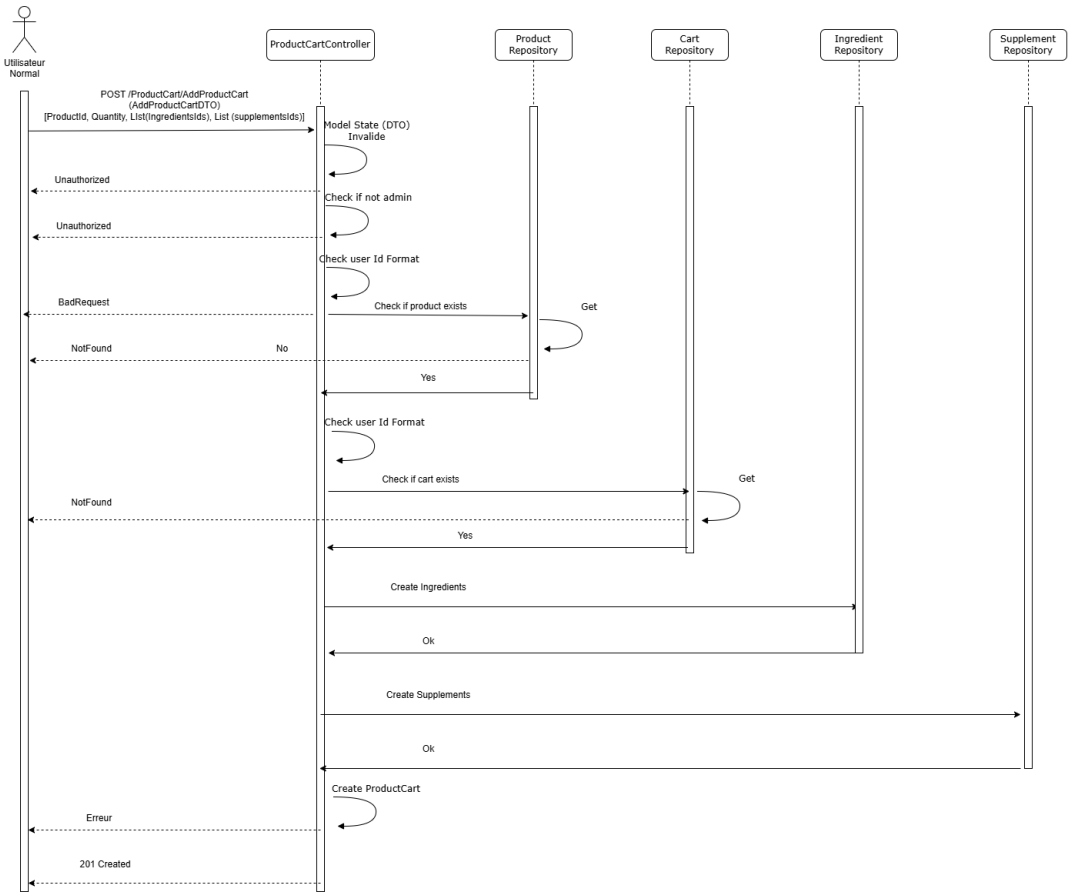

FIGURE 3.8 : Diagramme de séquence - Ajout d'un produit au panier

Conclusion

Les diagrammes de séquence présentés illustrent les interactions clés entre les utilisateurs (Client et Administrateur) et le système. Le diagramme d'authentification montre le processus de connexion des utilisateurs. Celui de la création de compte client décrit l'enregistrement d'un nouvel utilisateur. Enfin, le diagramme d'ajout d'un produit au panier présente l'interaction du client avec le système pour ajouter des produits. Ces diagrammes permettent de visualiser le flux des interactions et la communication entre les différents composants de l'application.

Chapitre 4

Réalisation et Resultats

4.1 Architecture de l'Application

L'application suit une architecture client-serveur pour garantir modularité et évolutivité.

— **Frontend** : Le frontend est développé en **ReactJS**, un framework JavaScript qui permet de concevoir des interfaces interactives et réactives. Il communique avec le backend via des appels API.
— **Backend** : Le backend est implémenté en **ASP.NET API**, qui gère la logique métier, les interactions avec la base de données, et les points d'accès pour le frontend. Pour structurer le backend, les patterns Repository et IRepository ont été adoptés afin d'assurer une séparation claire des responsabilités et de simplifier la maintenance.
— **Base de données** : SQL Server est utilisé pour assurer un stockage structuré et fiable des données.
— **IDE** : Visual Studio et Visual Studio Code offrent un environnement de développement intuitif pour le backend et le frontend.

4.2 Présentation des principaux endpoints

Dans le cadre de l'application de gestion des commandes de restaurant, plusieurs endpoints API ont été développés pour gérer les différentes fonctionnalités de l'application. Ces endpoints permettent d'interagir avec les données du système telles que l'authentification des utilisateurs, la gestion des commandes, et la consultation du menu. Voici une liste des principaux endpoints API développés :

Compte (Account)

— **POST** /api/Account/Register : Inscription d'un nouvel utilisateur. Permet de créer un compte en envoyant les informations nécessaires (nom complet, email, mot de passe).
— **POST** /api/Account/Login : Authentification. Retourne un JWT à utiliser pour sécuriser les requêtes futures.
— **POST** /api/Account/ChangePassword : Permet de modifier le mot de passe de l'utilisateur authentifié.
— **POST** /api/Account/ChangeFullName : Permet de mettre à jour le nom complet de l'utilisateur.
— **GET** /api/Account/GetUsers : Retourne la liste de tous les utilisateurs (accessible avec des permissions spécifiques, par exemple pour un administrateur).
— **GET** /api/Account/GetUser : Retourne les détails de l'utilisateur authentifié.

Panier (Cart)

— **POST** /api/Cart/AddCart : Crée un nouveau panier pour l'utilisateur authentifié.
— **GET** /api/Cart/GetCart : Récupère les détails du panier de l'utilisateur.
— **PUT** /api/Cart/EditCart : Modifie les informations du panier (ajout ou suppression de produits, par exemple).

Catégories (Category)

- **POST** /api/Category/AddCategory : Ajoute une nouvelle catégorie de produits (accessible aux administrateurs).
- **GET** /api/Category/GetCategory/Id : Récupère les détails d'une catégorie spécifique.
- **GET** /api/Category/GetAllCategories : Récupère toutes les catégories disponibles.
- **PUT** /api/Category/EditCategory/Id : Modifie une catégorie existante.
- **DELETE** /api/Category/DeleteCategory/Id : Supprime une catégorie existante.

Produits (Product)

- **POST** /api/Product/AddProduct : Ajoute un nouveau produit (accessibles aux administrateurs).
- **GET** /api/Product/GetProduct/Id : Récupère les détails d'un produit spécifique.
- **GET** /api/Product/GetAllProducts : Récupère tous les produits disponibles.
- **GET** /api/Product/GetProductsByCategory/CategoryId : Récupère les produits liés à une catégorie spécifique.
- **PUT** /api/Product/EditProduct/Id : Modifie les détails d'un produit existant.
- **DELETE** /api/Product/DeleteProduct/Id : Supprime un produit existant.

Commandes (Order)

- **POST** /api/Order/AddOrder : Crée une nouvelle commande en envoyant les détails (produits, quantités, etc.).
- **GET** /api/Order/GetOrder/Id : Récupère les détails d'une commande spécifique.
- **GET** /api/Order/GetUserOrders : Récupère toutes les commandes passées par l'utilisateur authentifié.
- **GET** /api/Order/GetAllOrders : Récupère toutes les commandes (accessible aux administrateurs).
- **PUT** /api/Order/EditOrder/Id : Met à jour les détails d'une commande spécifique.
- **DELETE** /api/Order/DeleteOrder/Id : Supprime une commande existante.
- **PUT** /api/Order/Deliver/Id : Met à jour le statut d'une commande pour la marquer comme livrée.

Boissons (Drink) et Produits du jour (ProductOfTheDay)

Ces endpoints suivent des schémas similaires, avec des options pour ajouter, récupérer, modifier ou supprimer des éléments.

4.3 Fonctionnement des principaux endpoints

POST /api/Account/Login

L'utilisateur envoie ses informations d'identification (email et mot de passe) dans une requête POST. En cas de succès, un JWT est généré et retourné dans la réponse. Ce jeton doit être inclus dans l'en-tête **Authorization** des requêtes nécessitant une authentification.

GET /api/Order/GetUserOrders

Ce point d'accès permet aux utilisateurs authentifiés de récupérer la liste de leurs commandes. Si le JWT est valide, la liste est renvoyée ; sinon, une réponse HTTP 401 Unauthorized est retournée.

POST /api/Order/AddOrder

Les utilisateurs peuvent créer une commande en envoyant les détails (produits, quantités) dans une requête POST. Le JWT est utilisé pour authentifier et associer la commande à l'utilisateur.

PUT /api/Order/EditOrder/Id

Les utilisateurs peuvent modifier leurs commandes en fournissant l'ID de la commande à mettre à jour et les nouvelles informations. Le JWT est vérifié pour garantir que seul l'utilisateur propriétaire ou un administrateur puisse effectuer cette modification.

GET /api/Product/GetAllProducts

Ce point d'accès est ouvert pour récupérer la liste de tous les produits disponibles. Certaines fonctionnalités supplémentaires, comme le filtrage par catégorie, peuvent être implémentées.

4.4 Sécurisation des endpoints

Authentification avec JWT

Le système d'authentification utilise des tokens JWT générés lors de la connexion. Ces jetons permettent de valider l'identité de l'utilisateur pour toutes les requêtes subséquentes.

Middleware de validation

Un middleware assure la validation des JWT. Si le token est absent ou invalide, l'accès à l'endpoint est refusé.

Contrôle d'accès basé sur les rôles

Certains endpoints, comme ceux liés à la gestion des produits ou des catégories, sont restreints aux administrateurs grâce à des rôles inclus dans le JWT.

4.4.1 Création du compte Admin

Lors du lancement de l'application, un compte *Admin* est créé automatiquement, avec l'assignation d'un rôle d'utilisateur *Admin*. Cela permet à l'administrateur d'accéder aux fonctionnalités de gestion, telles que l'ajout, la modification ou la suppression de produits, la gestion des catégories et des utilisateurs, etc.

4.5 Évolutions possibles

Plusieurs améliorations et nouvelles fonctionnalités sont prévues pour enrichir l'application et offrir une expérience utilisateur encore plus fluide et complète :

- **Suivi des commandes en temps réel :** Une fonctionnalité permettant aux utilisateurs de suivre en temps réel l'état de leur commande, depuis la préparation jusqu'à la livraison. Cela ajoutera une couche de transparence et de confort pour l'utilisateur.
- **Gestion des paiements en ligne :** L'intégration d'une solution de paiement en ligne permettra aux utilisateurs de régler leurs commandes directement via l'application, simplifiant ainsi le processus d'achat et améliorant l'expérience utilisateur.
- **Intégration avec des services de livraison :** Cette fonctionnalité permettra une gestion complète des commandes, y compris l'expédition et la livraison des produits. L'intégration avec des services tiers de livraison offrira une solution tout-en-un, améliorant ainsi l'efficacité et la satisfaction client.

Ces évolutions viseront à rendre l'application plus pratique, complète et compétitive sur le marché, tout en répondant mieux aux besoins des utilisateurs.

Conclusion

En conclusion, l'application de gestion des commandes de restaurant que nous avons développée représente une solution complète et moderne pour la gestion des commandes, de l'authentification des utilisateurs et a la consultation du menu. Cette application a été conçue pour répondre à un besoin spécifique dans le domaine de la restauration, en facilitant la prise de commande en ligne et la gestion des données utilisateurs dans un environnement sécurisé.

L'implémentation de l'authentification basée sur JWT (JSON Web Token) assure une gestion fluide et sécurisée des sessions utilisateurs, tout en garantissant que seules les requêtes authentifiées peuvent accéder aux ressources protégées de l'application. Le processus d'authentification est non seulement sécurisé, mais aussi évolutif, permettant à l'application de supporter une base d'utilisateurs croissante sans nécessiter de stockage des sessions côté serveur.

Le Repository Pattern a été appliqué pour maintenir une séparation claire entre la logique métier et la gestion des données. Cela permet une meilleure organisation du code, une facilité de maintenance et une meilleure testabilité des différentes parties de l'application. L'utilisation de ce pattern a permis de centraliser l'accès aux données et de simplifier les opérations CRUD pour les entités principales.

Les différents endpoints de l'API ont été soigneusement développés pour répondre aux besoins des utilisateurs, tout en offrant une interface claire et intuitive pour l'administration et les clients. La sécurisation des endpoints par JWT et la gestion des rôles d'utilisateur (comme l'administrateur et les client) assurent une utilisation sécurisée et flexible de l'application.

De plus, les diagrammes de classe et de cas d'utilisation ont permis de clarifier la structure de l'application, illustrant les relations entre les entités et les principales interactions entre les acteurs du système. Ces diagrammes ont facilité la compréhension du fonctionnement interne de l'application et ont guidé le développement des différentes fonctionnalités.

En somme, cette application démontre l'importance d'une architecture solide et d'une gestion sécurisée des données dans les applications modernes, en particulier celles qui manipulent des informations sensibles. Avec une bonne base en place, il est possible d'ajouter de nouvelles fonctionnalités, comme des systèmes de notifications ou des modules de recommandation pour améliorer l'expérience client. L'utilisation de technologies robustes et la mise en œuvre des meilleures pratiques garantissent la pérennité et la scalabilité de l'application à long terme.