**Projet : Copilot e-commerce**

Colas Rabiller - Juliette Infanti - Flavien Gros - Yasmine Radouani

#### **1. Introduction**

* **1.1. Contexte du projet**
  + Description du projet et des besoins du client.
* **1.2. Contraintes du projet**
  + Contraintes imposées pour la bonne réalisation du projet

#### **2. Design Patterns Mis en Place**

* **2.1. Introduction aux design patterns**
  + Brève explication de ce que sont les design patterns et pourquoi ils sont importants.
* **2.2. MVC**
  + 2.2.1. Description
  + 2.2.2. Implémentation dans le projet
  + 2.2.3. Avantages pour le projet
* **2.3. Injection de dépendances** 
  + 2.3.1. Description
  + 2.3.2. Implémentation dans le projet
  + 2.3.3. Avantages pour le projet
* **2.4.** **Repository**
  + 2.4.1. Description
  + 2.4.2. Implémentation dans le projet
  + 2.4.3. Avantages pour le projet
* **2.5. Builder**
  + 2.4.1. Description
  + 2.4.2. Implémentation dans le projet
  + 2.4.3. Avantages pour le projet

#### **3. Architecture technique**

* **3.1. Architecture globale de l'application**
* **3.2. Architecture des données**
* **3.3. Couche de service**

#### **4. Technologies**

* **4.1. Technologies utilisées**

#### **5. Réponse aux besoins du client**

* **5.1. Solutions apportées**
* **5.2. Problèmes rencontrés**

#### **6. Conclusion**

#### **1. Introduction**

## 1.1. Contexte du projet

La société X souhaite améliorer la productivité de ses équipes marketing lors de la mise en ligne des produits sur son site e-commerce. Pour ce faire, elle envisage de créer un outil, baptisé « Copilot », basé sur de l’IA générative. Cet outil sera capable d’automatiser la génération des descriptions des fiches produits, incluant les titres, descriptions et tags, en utilisant les caractéristiques produits fournies par les équipes marketing.

#### **Objectifs du projet**

* **Automatisation de la génération de contenu** : Utiliser une IA générative pour créer des titres, descriptions et mots-clés pour les fiches produits à partir des caractéristiques fournies.
* **Amélioration de la productivité** : Permettre aux équipes marketing de se concentrer sur des tâches à plus forte valeur ajoutée en automatisant la génération de contenu.
* **Interaction via un Chatbot** : Fournir un chatbot permettant aux équipes marketing d’interagir avec l’IA pour lancer des demandes de génération de fiches produits.
* **Gestion des imports et exports** : Permettre l’importation des caractéristiques produits via un fichier Excel et l’exportation des fiches produits générées, également sous forme de fichier Excel.

## 1.2. Contraintes du projet

Le projet de création du “Copilot” pour la société X doit répondre à plusieurs contraintes spécifiques pour assurer son efficacité et sa pertinence.

#### **1.2.1. Environnement technique**

* **Utilisation d'Azure OpenAI** :
  + Le modèle GPT-3.5 turbo sera utilisé pour la génération de contenu.
  + Chaque appel à l'API nécessite une clé d'accès unique.
  + Chaque région de déploiement disposera de sa propre clé d’authentification pour garantir la sécurité et la conformité régionale.

#### **1.2.2. Gestion des Données d'import**

* **Importation via fichier excel** :
  + Les utilisateurs doivent pouvoir importer les caractéristiques produits via un fichier Excel.
  + Une gestion précise des formats et de la validation des données importées est nécessaire pour éviter les erreurs et assurer la qualité des données.

#### 

#### 

#### **1.2.3. Ordonnancement des demandes**

* **Traitement des demandes en FIFO** :
  + Les demandes de génération de fiches produits doivent être traitées dans l'ordre de leur soumission.

#### **1.2.4. Guidelines marketing**

* **Titres** :
  + Chaque titre doit être en une seule phrase.
  + Les titres ne doivent pas dépasser 50 mots pour un affichage optimal sur mobile.
* **Descriptions** :
  + Chaque description doit être unique et ne pas reprendre le contenu du titre.
  + Les descriptions doivent contenir au maximum 250 mots.
* **Mots-clés** :
  + Chaque fiche produit doit comporter jusqu’à 5 mots-clés.
  + Les mots-clés doivent éviter les doublons et les variantes déjà utilisés pour d'autres produits.

#### **2. Design patterns mis en place**

## 2.1. Introduction aux design patterns

Les design patterns, ou motifs de conception, sont des solutions éprouvées et réutilisables à des problèmes courants rencontrés lors du développement logiciel. Ils offrent des stratégies standardisées pour résoudre des défis architecturaux spécifiques, permettant ainsi aux développeurs de créer des systèmes robustes, maintenables et évolutifs.

Dans le cadre de notre projet de chatbot qui permet aux équipes marketing d'une boutique de plantes d'échanger avec une IA générative pour la génération de fiches produits, voici les design patterns que l’on a choisi et comment on les a utilisé :

## 2.2. MVC

## 2.2.1. Description

Le pattern MVC (Model-View-Controller) est un modèle architectural couramment utilisé dans le développement d'applications web et mobiles. Il sépare l'application en trois composants principaux :

* Le modèle (Model) représente les données de l'application et la logique métier associée.
* La vue (View) est responsable de l'affichage des données à l'utilisateur.
* Le contrôleur (Controller) gère les interactions entre l'utilisateur et l'application en utilisant les données du modèle et en mettant à jour la vue.

Cette séparation permet de réduire la complexité de l'application, de faciliter la maintenance et l'évolution, et de favoriser la réutilisation du code.

**2.2.2. Implémentation dans le projet**

Dans notre projet, nous avons utilisé le pattern MVC pour séparer la logique métier, l'interface utilisateur et les interactions utilisateur. Par exemple, nous avons créé un contrôleur pour gérer la connexion/inscription, un modèle pour stocker les informations d'authentification et une vue pour afficher le formulaire de connexion/inscription à l'utilisateur.

Voici un exemple de code de notre contrôleur de connexion :



**2.2.3. Avantages pour le projet**

L'utilisation du pattern MVC dans notre projet a apporté plusieurs avantages :

* **Séparation claire des préoccupations** : la logique métier, l'interface utilisateur et les interactions utilisateur sont gérées dans des composants distincts, ce qui facilite la maintenance et l'évolution de l'application.
* **Réutilisation du code** : les vues et les contrôleurs peuvent être réutilisés dans d'autres parties de l'application, ce qui réduit la quantité de code à écrire et à maintenir.
* **Testabilité** : la séparation des composants permet de tester plus facilement chaque partie de l'application, en particulier la logique métier.
* **Amélioration de la productivité** : le pattern MVC est largement utilisé et documenté, ce qui permet de trouver facilement des solutions aux problèmes rencontrés et de réduire le temps de développement.

**2.3. Injection de dépendances**

## 2.3.1. Description

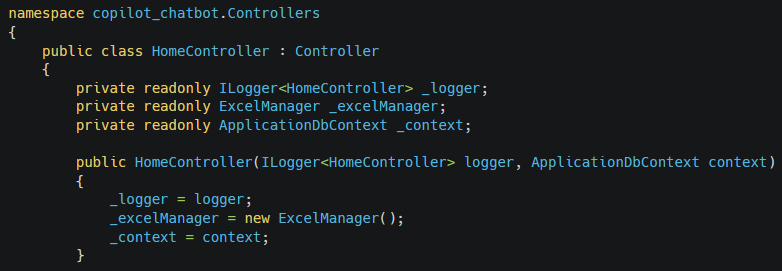
L'Injection de dépendances permet de gérer les dépendances entre les différents composants d'une application de manière à ce qu'ils soient découplés, modulaires et facilement testables.

**2.3.2. Implémentation dans le projet**

Nous avons utilisé l’injection de dépendances dans l’application pour fournir des instances de dépendances aux contrôleurs et aux services. Les dépendances principales que nous avons injectées sont ApplicationDbContext, OpenAIService, et IConfiguration.

Dans notre application, nous avons utilisé le principe d'inversion de contrôle pour implémenter l'Injection de Dépendances. Au lieu d'instancier les dépendances à l'intérieur des composants, nous les avons injectées à travers les constructeurs des composants. Cela permet de centraliser la gestion des dépendances dans un seul endroit, le **conteneur d'injection de dépendances**.

Voici un exemple d’injection de dépendance dans notre application pour les instances ILogger<HomeController> et ApplicationDbContext. Ces dépendances sont injectées dans le constructeur de la classe HomeController :



**2.3.3. Avantages pour le projet**

L'utilisation de l'injection de dépendances dans notre application a apporté plusieurs avantages :

* **Découplage des composants** : l'injection de dépendances permet de réduire les dépendances entre les composants, ce qui facilite la maintenance et l'évolution de l'application.
* **Modularité** : l'injection de dépendances permet de créer des composants modulaires et réutilisables, qui peuvent être facilement remplacés ou étendus.
* **Testabilité** : l'injection de dépendances facilite la testabilité du code en permettant d'injecter des dépendances dans les composants à tester.
* **Centralisation de la gestion des dépendances** : l'injection de dépendances permet de centraliser la gestion des dépendances dans un seul endroit, le conteneur d'injection de dépendances, ce qui facilite la configuration et la maintenance des dépendances.

**2.4.** **Repository**

**2.4.1. Description**

Le Pattern Repository permet d'encapsuler les opérations d'accès aux données dans une couche séparée, afin de faciliter la manipulation des données et d'abstraire la couche d'accès aux données. Le repository agit comme une interface entre la couche d'accès aux données et la couche métier de l'application.

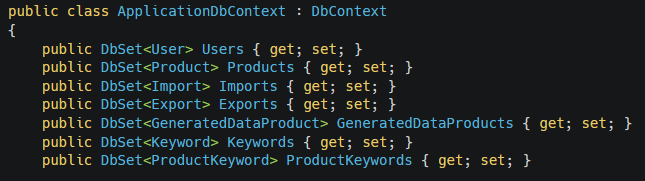
**2.4.2. Implémentation dans le projet**

Dans notre application, nous avons mis en œuvre le pattern Repository en utilisant EntityFrameworkCore pour accéder à la base de données via ApplicationDbContext. Les opérations de base de données sont encapsulées dans des services.

ApplicationDbContext agit comme un repository pour accéder aux entités User, Product, et Import. Il fournit une abstraction de la couche d'accès aux données et permet de manipuler les données de manière simple et efficace.

Nous avons également utilisé des interfaces pour définir les méthodes d'accès aux données, et des classes concrètes pour implémenter ces interfaces. Cela permet de découpler les différentes couches de l'application et de faciliter la maintenance et l'évolution de l'application.

Voici un exemple d'utilisation du repository dans notre application, pour récupérer une liste de produits :



En utilisant l'interface DbSet<T>, on peut encapsuler les opérations de base de données dans des services, ce qui permet de découpler les différentes couches de l'application et de faciliter la maintenance et l'évolution de l'application.

**2.4.3. Avantages pour le projet**

L'utilisation du pattern Repository dans notre projet apporte les avantages suivants :

1. **Abstraction de la couche d'accès aux données** : en utilisant une interface pour définir les méthodes d'accès aux données, on peut manipuler les données de manière simple et efficace, sans avoir à se soucier des détails d'implémentation de la couche d'accès aux données.
2. **Facilité de maintenance et d'évolution** : en encapsulant les opérations de base de données dans des services, on peut réduire la complexité de l'application et faciliter la maintenance et l'évolution de l'application.
3. **Facilité de test** : on peut facilement créer des mocks de l'interface pour tester les services de l'application de manière isolée. Cela permet de réduire la complexité des tests et de faciliter la détection des erreurs.
4. **Flexibilité** : facilite le changement de la couche d'accès aux données de l'application sans avoir à modifier le reste de l'application. Par exemple, si on a besoin de migrer la base de données vers une autre technologie, on peut le faire en modifiant uniquement la couche d'accès aux données, sans avoir à modifier le reste de l'application.

**2.5. Builder**

**2.4.1. Description**

Le Pattern Builder permet de créer des objets complexes de manière simple et flexible. Il permet de séparer la construction de l'objet de sa représentation, ce qui permet de modifier facilement la configuration de l'objet sans affecter directement la logique de l'application.

**2.4.2. Implémentation dans le projet**

Dans notre projet, nous avons utilisé le Pattern Builder pour la création des modèles de données de notre application. Nous avons utilisé la classe ModelBuilder de Entity Framework Core, qui suit plusieurs principes du Pattern Builder.

La classe ModelBuilder permet de configurer un modèle de données étape par étape, en définissant les tables, les relations, et les contraintes.

Voici un exemple d'utilisation de la classe ModelBuilder dans notre projet :



Dans cet exemple, nous utilisons la méthode Entity de la classe ModelBuilder pour configurer les entités Export, Keyword et Product. Nous utilisons ensuite les méthodes HasMany, WithOne, HasForeignKey et OnDelete pour définir les relations entre les entités et les contraintes de la base de données.

**2.4.3. Avantages pour le projet**

L'utilisation du Pattern Builder dans notre projet a apporté plusieurs avantages :

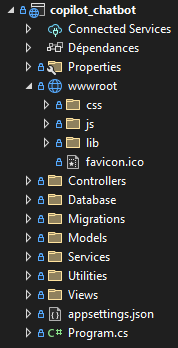
* **Simplicité et flexibilité** : en utilisant une interface fluide pour chaîner les appels de méthode, nous avons pu configurer les modèles de données de manière simple et flexible.
* **Séparation de la construction et de la représentation** : en utilisant le pattern Builder, nous pouvons séparer la construction des modèles de données de leur représentation, ce qui permet de modifier facilement la configuration des modèles sans affecter directement la logique de l'application.
* **Facilité de maintenance et d'évolution** : en utilisant une interface fluide pour configurer les modèles de données, nous avons pu réduire la complexité de l'application et faciliter la maintenance et l'évolution de l'application.

**3. Architecture technique**

**3.1. Architecture globale de l'application**

Notre application web est monolithique. Elle utilise ASP.NET Core pour la logique métier et les contrôleurs, et Razor (cshtml) pour les vues.

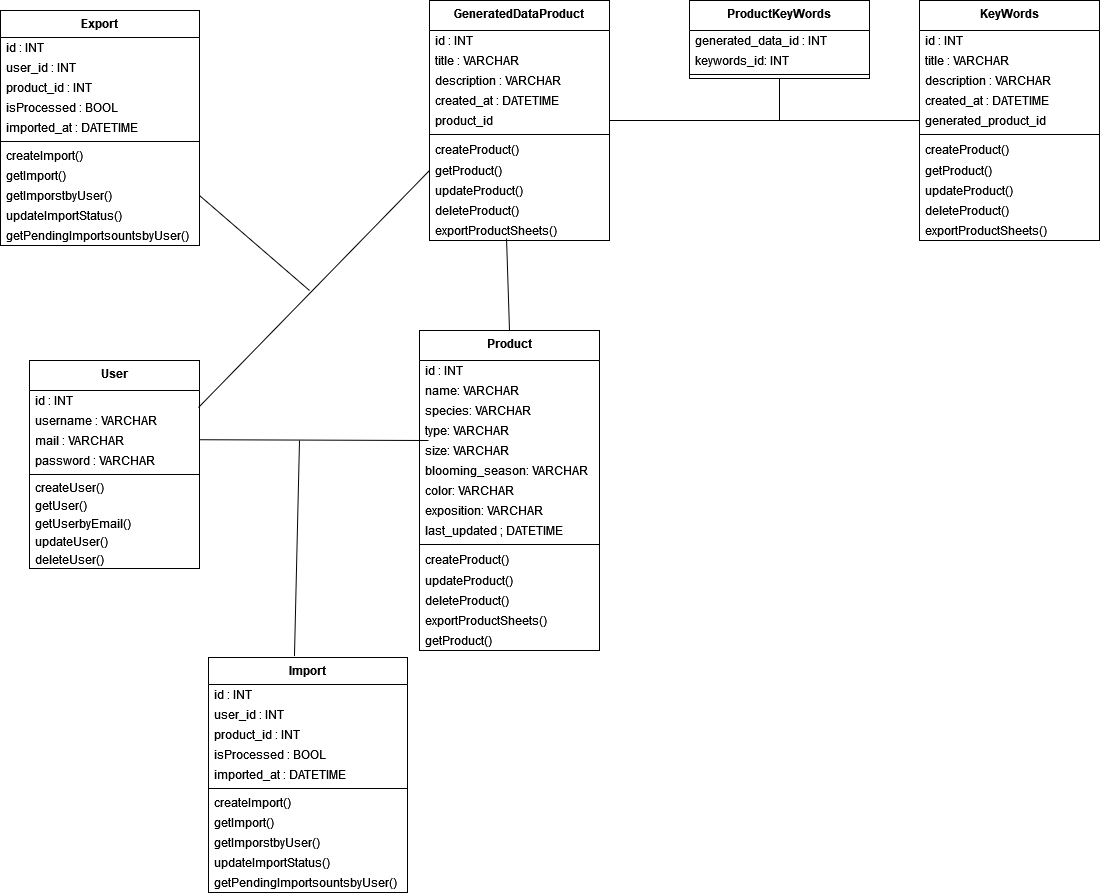
L'application utilise une base de données relationnelle pour stocker les données persistantes, et fait appel à l'API d'OpenAI pour fournir des fonctionnalités de chatbot.



**3.2. Schéma de l'architecture des données**

Afin d’identifier les problèmes potentiels et pour comprendre la structure et le fonctionnement de notre application, nous avons réalisé un modèle conceptuel de données.

Cela nous a également permis d’identifier les entités qui doivent être créées en base de données pour répondre aux spécifications.

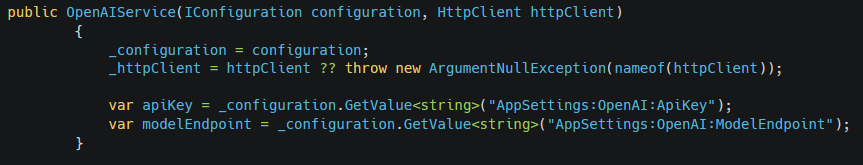
****

**3.3. Couche de service**

Nous avons créé des services pour la connexion à l’API d’OpenAI et la connexion de l’utilisateur car ces méthodes seront utilisées à plusieurs reprises dans l’application.

Ces services permettent d'encapsuler la logique métier et de séparer les préoccupations. Les contrôleurs appellent ces services pour exécuter la logique métier, ce qui favorise la réutilisabilité du code et le rend plus testable.

Voici un exemple de méthode du service OpenAIService pour la configuration des appels à l’API OpenAI :



#### 

#### **4. Configuration des services et technologies**

**4.1. Technologies utilisées**

Dans notre projet, nous avons utilisé les technologies et frameworks suivants :

* **C# / ASP.NET Core** : nous avons choisi C# comme langage de programmation et ASP.NET Core comme framework pour sa flexibilité, sa performance, sa compatibilité multi-plateforme mais surtout car il est étroitement intégré avec le service Azure OpenAI que nous utiliserons.
* **SQLite** : concernant la base de données, nous avons opté pour SqlLite pour sa légèreté et la possibilité qu’elle offre d’être intégrée directement au projet. De plus, nous avions besoin d’une base de données relationnelle car nous avons besoin d’obtenir des informations significatives en reliant les tables.
* **Entity Framework Core** : nous utilisons Entity Framework Core qui est un ORM pour .NET Core, qui permet de manipuler les données de la base de données de manière simple et efficace. Nous avons choisi Entity Framework Core pour sa facilité d'utilisation, sa compatibilité avec ASP.NET Core, et sa prise en charge de SQLite.
* **HTML, CSS, et JavaScript** : pour la création d'interfaces web dynamiques.

#### **5. Réponse aux besoins du client**

**5.1. Solutions apportées**

**Besoin n°1 :**

Échanger avec une IA générative afin de lancer la génération des fiches produits.

**Solution n°1 :**

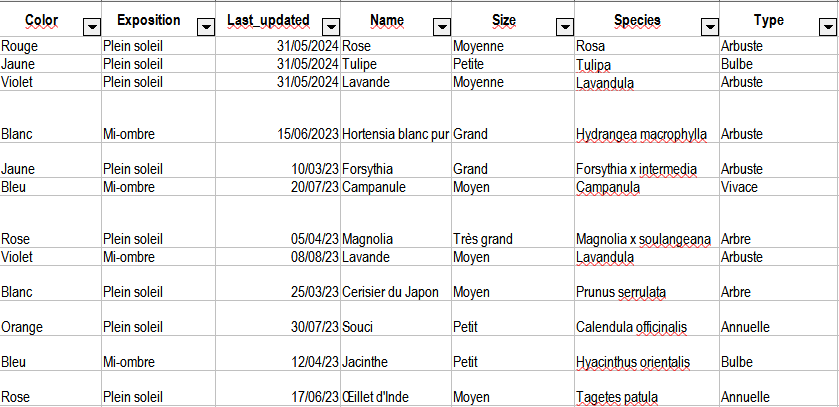
Pour répondre à ce besoin, nous avons créé une interface web dynamique qui permet à l’utilisateur d’écrire du texte. Nous faisons des appels à l’IA générative OpenAI en lui envoyant le message afin qu’elle réponde à l’utilisateur et lui propose d’importer un fichier excel.

**Besoin n°2 :**

Les caractéristiques produits seront importées par chaque utilisateur via un fichier Excel. Chaque demande de génération de contenu par l’IA devra être associée aux caractéristiques produits présentes dans l’Excel au moment de l’import.

**Solution n°2 :**

Nous avons tout d’abord créé un fichier type à importer :



Nous vérifions ensuite que le fichier n’est pas vide et qu’il a la bonne extension .xlsx. Nous utilisons la bibliothèque ClosedXML pour ouvrir le fichier Excel et accéder à la première feuille de calcul.Pour chaque ligne de la feuille de calcul, la méthode crée un nouveau dictionnaire rowData qui associe chaque valeur de cellule à sa lettre de colonne correspondante. Les données de chaque ligne du fichier donc de chaque produit sont stockées en base de données avec les informations sur l’import (table import).

**Besoin n°3 :**

N’importe quel utilisateur peut exporter, sous la forme d’un fichier Excel, l’ensemble des références produits du site et les éléments générés par IA, quel que soit l’utilisateur ayant fait la demande initiale.

**Solution n°3 :**

**Besoin n°4 :**

La génération doit être ordonnée sur une base FIFO

**Solution n°4 :**

Quand l’utilisateur clique sur le bouton importer de l’interface, nous utilisons la classe Queue qui fait partie de la bibliothèque de classe .NET et qui permet d’ajouter le fichier à la file d’attente.

Un booléen isProcessing est passé à true dès qu’un fichier est en train d’être traité. Un fichier ne peut être traité que si isProcessing est à false.

**Besoin n°5 :**

L’utilisateur devra aussi être capable de savoir si des imports en son nom sont toujours en cours de génération

**Solution n°5 :**

Pour connaître les actions des utilisateurs, nous avons ajouté un système d’authentification pour accéder au chat et au système d’import. Cela nous permet de lier en base de données un utilisateur à un import ou un export.

**Besoin n°6 :**

La génération par l’IA doit respecter des guidelines marketing fournies par les équipes communication.

**Solution n°6 :**

**5.2. Problèmes rencontrés**

**5.2.1. L'intégration de l'API d'OpenAI**

Nous avons eu du mal à passer nos premiers appels à l’API d'OpenAI car nous n’avions pas le bon endpoint.

Nous nous somme aidé de la documentation officielle pour trouver la solution :

**https://learn.microsoft.com/en-us/azure/ai-services/openai/**

**5.2.2. Le choix des design pattern**

Au début du développement nous avons eu du mal à trouver les design pattern utiles pour notre application, mis à part le pattern MVC.

C’est au cours de notre développement que certains design pattern se sont imposés de fait, comme le Repository pour la création de la base de données ou le Builder pour les Models.

**5.2.3. La gestion des données**

Il nous a fallu un peu de temps pour comprendre et définir l’architecture adéquate et le système en cascade qu’implique les relations entre les utilisateurs, les imports, les exports, les produits et les mots clé. La partie conception, notamment le diagramme UML nous a permis de clarifier les besoins.

**5.2.4 Les vérifications**

Plusieurs vérification étaient nécessaires pour garantir la cohérence des données, comme la non duplication des produits et fiche produit en base de données ou le format correct xlsx.

#### **6. Conclusion**

Dans le cadre de ce projet, nous avons développé une application web monolithique permettant d'importer des données produits depuis un fichier Excel et de les stocker dans une base de données. Nous avons également intégré un chatbot basé sur l'API d'OpenAI pour répondre aux questions des utilisateurs.

Nous avons utilisé des design patterns tels que MVC, Repository, Injection de dépendances et Builder pour structurer notre code et faciliter sa maintenance.

Nous avons rencontré quelques difficultés lors du choix des design patterns et lors de la mise en place des vérifications, mais nous avons su les surmonter en travaillant ensemble et en nous appuyant sur nos connaissances et notre expérience.

Grâce à ce projet, nous avons pu approfondir nos compétences en développement C# et en intégration d'API, ainsi qu'en conception et en gestion de base de données.