Projet architecture applicatives

“Copilot e-commerce”

Saad KADIRI - Erwan NICOLE - Yohan MARTIN - Marco PYRE

Introduction

Pourquoi avoir choisi Angular ?

Angular est un framework front complet, ce qui signifie qu'il fournit tout ce dont nous avons besoin pour développer une application, de la gestion des dépendances à la liaison de données du back au front en passant par le routage. Cela peut rendre le développement plus rapide, guidé et donc plus facile, car nous n'avons pas à assembler nous-mêmes une pile de technologies.

Angular fournit une liaison de données bidirectionnelle, ce qui signifie que les modifications apportées au modèle de données de notre application sont automatiquement reflétées dans la vue, et vice versa. Cela peut réduire considérablement la quantité de code que nous devons écrire et faciliter la gestion de l'état de notre application.

Angular bénéficie d'une grande communauté de développeurs, ce qui signifie que nous pouvons trouver de nombreuses ressources pour nous aider à apprendre et à résoudre les problèmes, comme des tutoriels, des bibliothèques tierces et des forums de discussion.

Avec Angular, nous pouvons créer des composants réutilisables, ce qui peut nous faire gagner du temps et améliorer la cohérence de notre application. De plus, Angular fournit des outils en ligne de commande pour la génération de code, ce qui peut rendre la création de nouveaux composants encore plus rapide.

Pourquoi avoir choisi NestJS ?

NestJS utilise une architecture modulaire, ce qui signifie que nous pouvons organiser notre code en modules réutilisables et faciles à tester. Cela peut nous aider à maintenir notre application à mesure qu'elle grandit et à améliorer la collaboration entre les développeurs.

Comme Angular, NestJS est écrit en TypeScript, ce qui signifie que nous pouvons utiliser les mêmes fonctionnalités de vérification de type statique et de classe pour notre code backend. Cela peut rendre notre code plus robuste et plus facile à comprendre et à maintenir.

NestJS fournit une intégration facile avec Angular, ce qui signifie que nous pouvons facilement connecter le backend au frontend en utilisant des services Angular. De plus,

NestJS fournit des outils pour la génération de code, ce qui peut nous aider à créer rapidement des contrôleurs et des services pour notre backend.

NestJS fournit une interface de ligne de commande (CLI) pour la génération de code et la gestion de notre application. Cela peut nous aider à gagner du temps et à réduire les erreurs en automatisant certaines tâches de développement.

NestJS bénéficie d'une communauté active de développeurs et de nombreuses ressources en ligne, telles que des tutoriels, des bibliothèques tierces et des forums de discussion. Cela peut nous aider à apprendre et à résoudre les problèmes plus rapidement.

Design Pattern implémentés dans le projet

Dans le cadre de ce projet, nous avons implémenté les design pattern suivants :

- MVVM (Model-View-ViewModel)
- BFF (Back-end For Front-end)
- Layered Architecture

Voici comment ces designs pattern ont été utilisés et en quoi ils répondent aux problématiques du projet :

Modèle-Vue-VueModèle (MVVM)

Le design pattern MVVM a été choisi pour sa capacité à séparer clairement les préoccupations concernant l'interface utilisateur, la logique métier et les données. Cela permet une meilleure maintenabilité, extensibilité et testabilité de l'application.

Model

Le Modèle représente les données et la logique métier de l'application.
Dans notre cas, le Modèle est responsable de la gestion des données relatives aux caractéristiques des produits ainsi que des données générées par l'IA.
Les caractéristiques produits importées par les utilisateurs sont gérées par le Modèle, et les données générées par l'IA sont également stockées dans une base de données Postgresql et manipulées par cette composante.
Dans la version 1 de l'application, le Modèle est étendu pour inclure les fonctionnalités suivantes :

- Calcul du nombre de références produits dans la base.
- Suivi du nombre d'importations effectuées par l'utilisateur.
- Indication de l'état des importations en cours de génération.

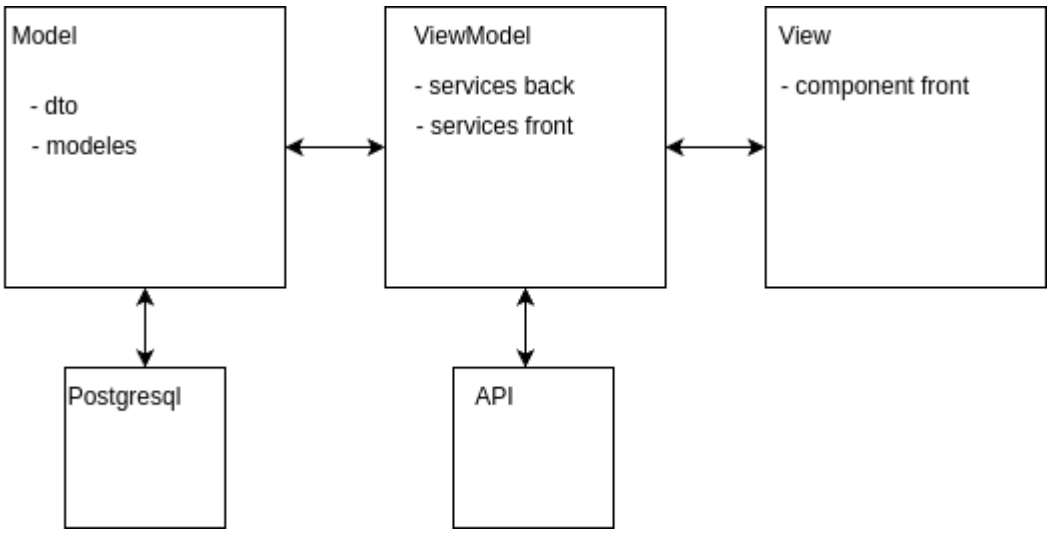
View

La View est responsable de l'affichage des données à l'utilisateur et de la réception des interactions utilisateur. Dans notre application, la View correspond à l'interface utilisateur du chatbot.
Elle présente les données au format approprié et récupère les entrées utilisateur pour les transmettre au ViewModel.

ViewModel

Le ViewModel agit comme un intermédiaire entre la Vue et le Modèle. Il récupère les données du Modèle et les prépare pour l'affichage dans la vue. Il traite également les interactions utilisateur de la Vue et les traduit en actions sur le Modèle.
Dans notre application, le ViewModel est responsable de la gestion des interactions utilisateur avec le chatbot, de l'envoi des demandes de génération de contenu à l'IA, et de la mise à jour de l'interface utilisateur avec les données générées.
Dans la version 1 de l'application, le ViewModel est étendu pour inclure les fonctionnalités de suivi des imports précédents et de l'état des imports en cours de génération.

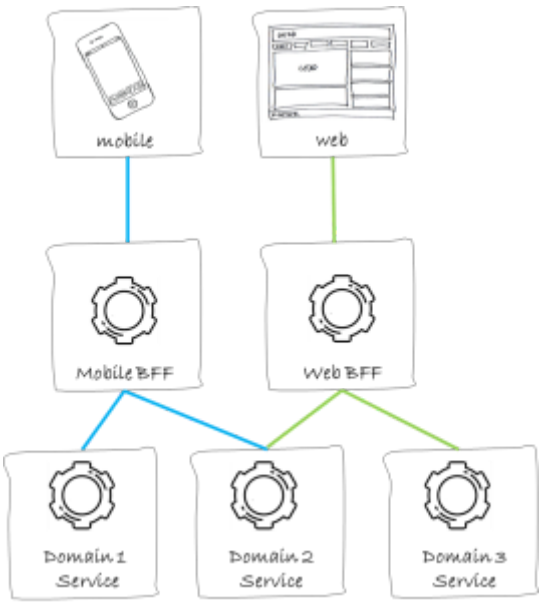
Les générations sont faites de manières asynchrones afin de pouvoir continuer la génération en parallèle.



BFF

Le design pattern BFF (Back-end For Front-end) a également été utilisé, nous avons mis en place une séparation du back-end et du front-end afin de maximiser la performance de l'application et d'organiser correctement les différents services.

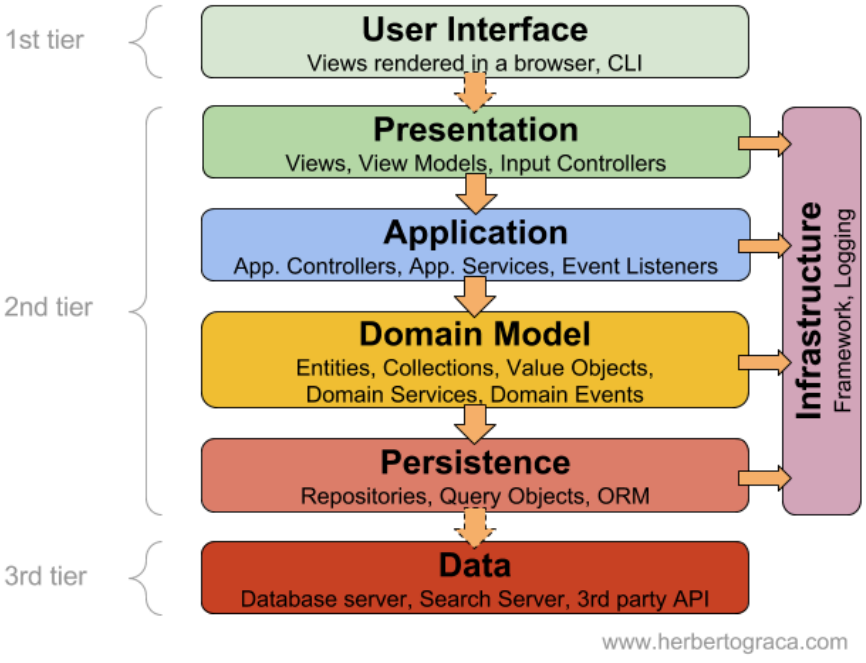
Le back end sert de "tampon" pour traiter la donnée que le front-end lui envoi, puis il la renvoi à l'api et à la base de donnée, la même chose est effectué pour la donnée venant de l'api, elle est traité, stocké, puis renvoyé au front-end



Layered Architecture

Le design pattern de Layered Architecture a également été appliqué au sein de l'application front et back, cette architecture permet de séparer les différents services en couches, des couches qui vont interagir entre elles sans passer à travers l'une de l'autre.

Ce design pattern permet de maximiser la rentabilité du code, car elle va scinder l'application en services qui pourrait être utilisé dans différents composants de plus bas niveaux dans ces mêmes couches.



L’application nécessite également de suivre une template afin de générer simplement les données:

Titre	Mot clé
<<titre du produit à générer>>	<<mot clé 1, mot clé 2, mot clé 3 ...>>

Notre application fait également l’usage de tailwind afin de rendre le styling plus facile et rapide.

Difficultés rencontrées

La majeure difficulté rencontrée a été de faire comprendre au bot via l'api distant le résultat voulu pour l'application. En effet, certains paramètres donnés au bot ont été complexes à mettre en place, comme par exemple le fait qu'il soit conscient qu'une génération est en cours. Pour cela nous avons utilisé une entrée en base de donnée, qui est stockée temporairement le temps de la génération, cette entrée est associée à la conversation sur laquelle la génération ce produit. Elle permet donc de donner au bot l'information qu'une génération est en cours dans le contexte dans lequel il complète la discussion.

Ce paramètre n'était au début pas pris en compte par le chatbot, nous avons donc essayé de corriger ce souci en réduisant la température du bot. En réduisant sa température, le bot devient moins créatif, ce qui dans notre cas, nous permettrait de résoudre ce problème. Hors le problème était partiellement résolu car la réponse à une question relative à la génération en cours n'était pas 100% fiable.

Nous avons donc par la suite ajusté le message permettant de générer la donnée. En ajoutant un message demandant au chatbot d'attendre qu'une validation de l'application soit faite pour valider qu'une génération soit finie, nous avons également ajusté le message validant la fin de la génération en ajoutant une sorte de pondération. En répétant plusieurs fois la même chose au chatbot de manières différentes afin qu'il comprenne correctement la validation de la génération.

Afin de fiabiliser le bot, une température conditionnelle a été également ajoutée. Sur une génération la température serait de 1 afin que le chatbot soit plus créatif, alors qu'une simple réponse serait à une température de 0,7 afin de maximiser la fiabilité des réponses.

Lancement de la solution

Voici les étapes pour compiler la solution :

- Clonez le dépôt Git du projet à partir de l'URL suivante : https://github.com/marcopyre/archiApplicative.git

- Assurez-vous d'avoir Node.js et docker installés sur votre système.

- Dans le répertoire back du projet, démarrez le backend:
    - **npm install**
    - **docker compose up**
    - **npm run**

- Dans le répertoire front du projet, démarrez le frontend:
    - **npm install**
    - **npm run start**

- L'application sera disponible à l'adresse :
    - **http://localhost:4200/**

Conclusion

Ce projet nous a permis de développer nos compétences sur plusieurs sujets. Premièrement le principe des designs pattern, leur fonctionnement et l'intérêt de les utiliser pour mettre en place le chatbot.
Ensuite, nous avons pu voir et paramétrer le chatbot en récupérant les informations d'une API et veiller à la pertinence des réponses de ce chatbot. Enfin, nous avons pris en compte la problématique du contexte et sa gestion pour avoir un chatbot qui maintient son niveau de performance et de logique en fonction de chaque questions successives de l'utilisateur.