ASSOCIATION DE BOTANISTE

AROSAJE

Designed by nosithouss

CONTEXT

L’entreprise Arosaje aide les amoureux de la nature à prendre soin de leur plante.
Elle propose deux services principaux;

- En allant garder leurs plantes lorsque les propriétaires sont absents
- En prodiguant des conseils d entretien afin que les propriétaires s’occupent de mieux en mieux de leurs plantes.

Cependant, depsui la pandemie, arosaje subit une forte hausse de demande, qu’elle ne peut pas traiter, elle a besoin d’une aplication communautaire et automatique pour aider les mains vertes.

L’application doit permettre:

- De faire garder ces plantes pas un autre.
- De partager des photo de plantes entre utilisateur et permettre au botanniste de donnée quelques conseil.
- Aux utilisateurs et les gardien doivent pouvoir se contacter via l’application.

C’est la que Nosithouss intervient...

LA SOLUTION

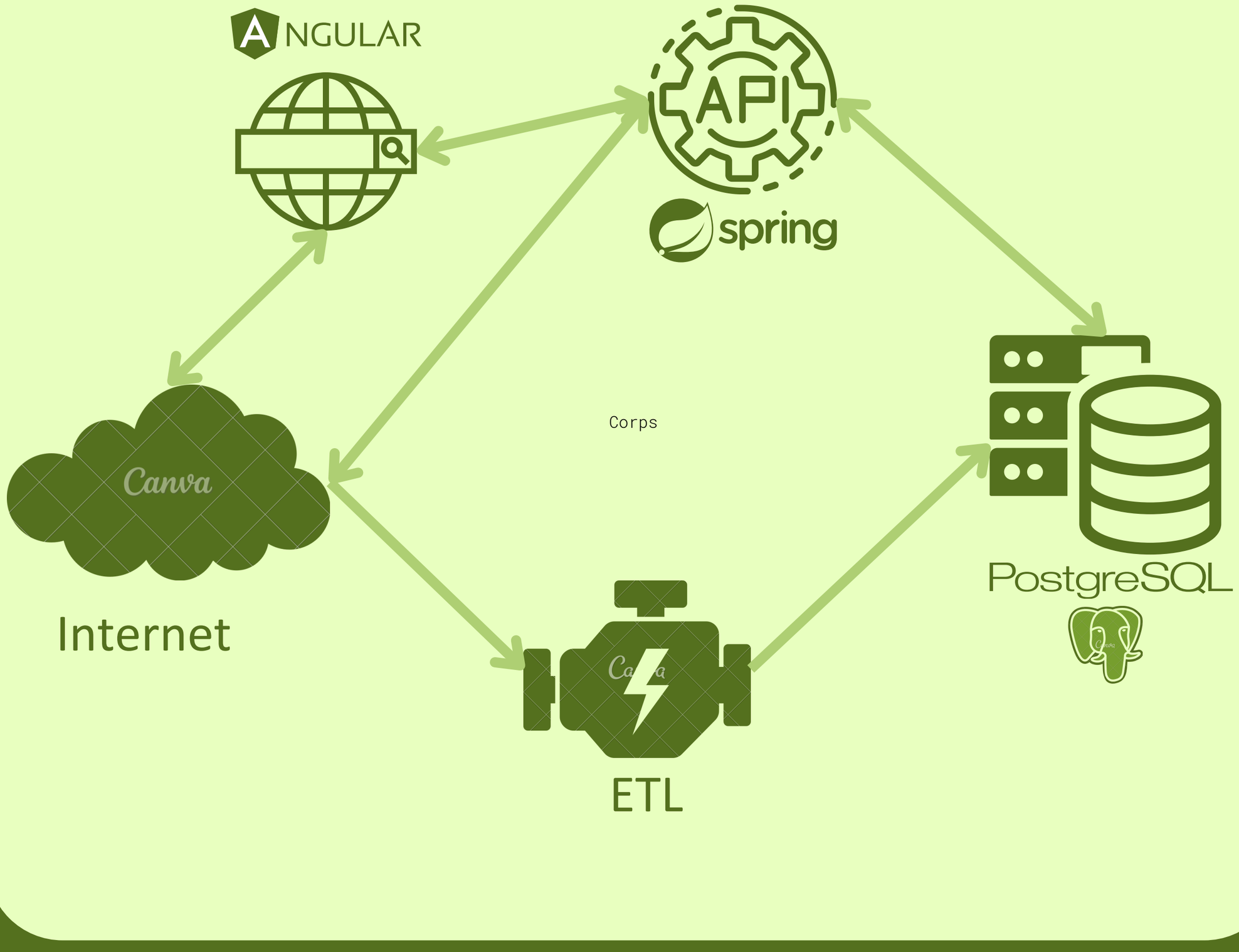
Une application web conçue en mobile first; “Arosaje” et une api REST permettant de brancher plusieur front dessus.

L’idée est de commencer par une application web accessible via un navigateur en mobile first, cet à dire que l’application a été conçue pour être utilisé depuis un téléphone mobile (un peu comme le site d’Instagram...).

Le fait de commencer par une application web offre plusieurs avantages : l’accessibilité, la rapidité de développement, facilité de partage, etc...

Il est bien sûr possible de passer à une application mobile notamment grâce à “Electron”

L’INFRASTRUCTURE



Dans ce schéma d’infrastructure, on retrouve 4 principaux éléments :

- API:

Construite en Java avec le framework Spring Boot, elle expose des endpoints (sécurisé ou non). Chacun de ses endpoint correspond a une action, un traitement de donnée qui vient (le plus souvent) interagir avec la base de donnée (Ajout de plante, d’utilisateur, etc...)

- BDD:

La base de donnée PostgreSQL, c’est à la fois notre espace de stockage pour nos plants, mais aussi notre ancre entre les différents services...

- ETL:

Il est chargé d’extraire toutes les données de l’api spécial botaniste “trefle.io” pour enrichir notre catalogue stocké en base de donnée.

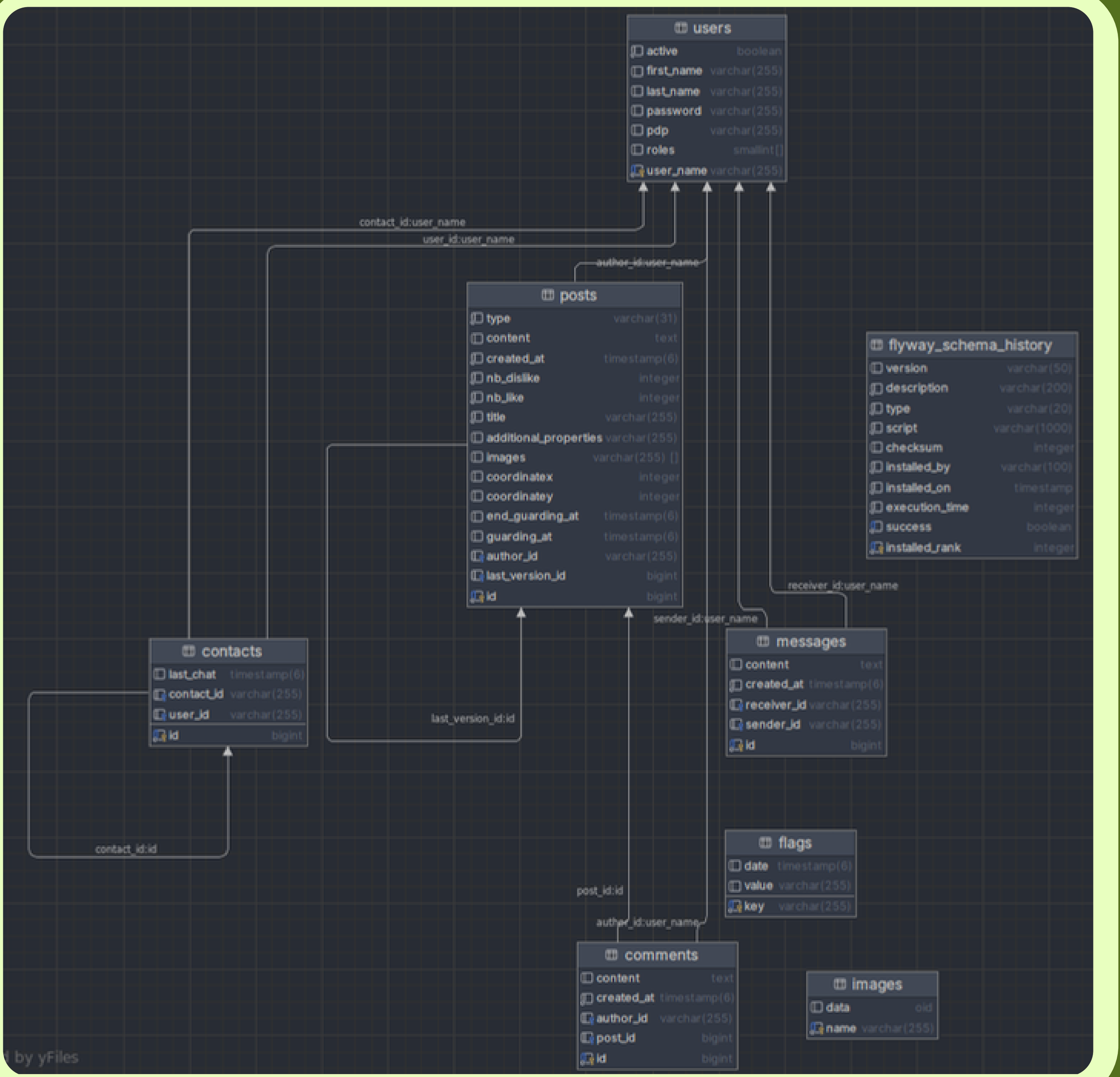
- FRONT:

Le front en Angular offre une interface sympas et pratique pour nos premiers utilisateurs, il communique uniquement avec notre api Spring Boot.

Il est important de noter que chacune de ces briques est conteneurisée via docker, ce qui facilite leurs déploiements...

Entrons un peu plus dans le détail...

POSTGRESQL - BDD



On remarque dans notre schémas une table “Images”, sans relation, avec deux champs...

Comme déjà évoque précédemment, nous avons deux instances de notre application qui peuvent tourner en parallèle, l’ETL et l’application web. L’ETL stocke des images et l’application web a besoin de les afficher. La nécessiter d’externalisé la gestion d’image se fait sentir, malheureusement nous n’avons pas réussi à créer de serveur sftp alors le stockage d’image en base de donnée paraissait être le meilleur moyen.

Bien que ce ne soit pas très pertinent, car les image ne sont pas des objets “requetable”, ce qui semble entée en contradiction avec le sql, nous avons malgré tout choisi de leurs dédiés une petite table, sans relation parce qu'elle ne pourrait être que provisoire...

Notre deuxième table un peu spéciale est “Flags”, table à trois colonnes (clef, valeur, date) elle fait office d’ancre entre les instances d’application, de log ou simplement de configuration...

SPRING BOOT - API

A noté que l’API et l’ETL sont en réalité le même projet Spring Boot mais deux configurations différentes...

Controllers

Dans notre dossier controllers, on retrouve nos classes gérant les endpoints, de manière générale, on retrouve une classe de controller par entité de notre base de donnée... Elle communique directement avec notre couche service...

Services

C’est notre couche métier, elle effectue la première transformation entre les objets envoyée par l’utilisateur et le format attendue par la base de donnée... Elle peut communiquer avec plusieurs branches de notre projet, mais son objectif reste de liaison entre nos contrôleurs et nos repositories...

Repositories

Cette couche communique avec la base de donnée, elle va effectuer des requêtes pour extraire, modifier, supprimer ou insérer de la donnée en base. La plupart sont des interfaces héritées de “JpaRepository” qui favorise grandement l’utilisation du SQL. Néanmoins, une classe a besoin de requête SQL plus complexe, on utilise alors la logique de Persisence et d’EntityManager

Dtos

On retrouve ici nos objets, nos formats de réponse de requête, c’est par le billet de ces objets que l’on communique avec les utilisateurs. Séparer en deux dossiers (responses, request). Ce fonctionnement offre plus de cadrage et sert a notre swagger.

Clients

On a ici nos classes de communication avec d’autre API (trefle.io pour le moment). On utilise des WebClient.

SWAGGER-UI

La dépendance spring doc nous propose un moyen simple de documenter notre API, spécialement nos endpoint, dans notre cas, on utilise swagger-ui, disponible à cette adresse : “http://localhost:8080/swagger-ui/index.html#/” (L’api doit être lancé au préalable).

L’architecture de notre application avec des dtos et la doc sur nos endpoint, indiquent a quiconque de la manière de discuter avec notre api.

SPRING SECURITY

Une La dépendance Spring-security permet d’avoir une couche de sécurité via token baerer JWT qui englobe notre application.

Lorsqu’un utilisateur veut accéder à notre application, il se crée un compte avec un mot de passer et un nom d’utilisateur, il se connecte, un token de sécurité contenant son nom, son rôle, une date d’expiration, etc... lui est transmis, à partir de ce token, il peut accéder à toute l’application.

Si l’utilisateur veut créer une nouvelle annonce, pas besoin de renseigner son pseudo ou son nom, tout est dans le token.

Toutes les routes sont protégées par ce système mis à part la création dun compte, la connexion à un compte et les ressource /assets/, nous y reviendront.

MIGRATIONS

Que ce soit pour ajouter de la configuration, de la donnée de test ou modifier notre base de donnée, il est important d’avoir un jeu de migration organisée qui se joue au démarrage de notre application...

Dans notre cas nous utilisons flyway. Ces migrations nous permette principalement a avoir de la donnée de test ds utilisateur, des post, etc...

Pour garder la main sur nos données, la première migration à être jouée est “V1__Init.sql”. Bien que Spring Boot et Hibernate nous permettent de générer automatiquement nos relations et nos tables au démarrage de notre application, il est important de centraliser, d’historiser et de garder la main sur la modification en base, surtout quand plusieurs instances d’application peuvent se connecter dessus, et c’est notre cas avec l’ETL...

Même si ce script vient couper l’herbe sous le pied d’Hibernate, c’est en réalité un DDL généré une fois que spring ait correctement créer notre table. Nos entité sont donc correctement configurées au sein de notre application.

ETL

Une des fonctionnalités de notre application étant le catalogue, les utilisateurs doivent pouvoir scroller sur leur page feed et voir défiler des plantes, des fleurs avec leur nom et quelques infos, mais ces infos, il faut bien aller les chercher quelque part...

trefle.io est une api pour les passionnés de botanique, qui réfèrent plus de 400 plantes différentes. Le rôle de notre ETL est d’extraire la totalité de ces plantes et de la stocker en base... (en réalité, on peut lui spécifier une limite...)

L’ETL e lance selon une configuration spring particulière :

```
spring:
    main:
        web-application-type: none
        etl: true
```

Avec la conteneurisation, toute cette configuration est gérée par le fichier docker-compose-etl.yml, on utilise un profil différent application-etl.yml...

Il est possible de fixer une limite au nombre de plantes à enregistrer, en argument de commande “-limit=50”, par défaut cette limite est fixé à 200.

Que se passe-t-il en cas d’erreur, en cas de coupure, etc...

Notre base de donnée possède une table flags avec trois colonnes, valeur, clef, date. À chaque “extraction”, on vient sauvegarder en base la dernière plante enregistrée, le nombre de plantes enregistré depuis que l’etl est en marche.

On peut donc savoir s'il y a eu un problème, si la limite est fixée à 200 et que l’etl a enregistré 100 plantes, on peut le relancer, il sait alors la dernière plante qu’il a traitée, il reprend depuis là...

PS: L’ETL se lance uniquement si l’application n’est pas une application web, autrement dit, il sera impossible d’accéder au endpoint de notre ETL...

DOCKER

Nous avons essayé de conteneuriser au maximum notre application, c’est pourquoi on retrouve (pour l’instant) 3 fichiers docker-compose, un pour la base de donnée, un pour notre application en mode etl et le dernier pour notre application en mode serveur web.

Nous espérons pouvoir mettre notre front en angular sur un conteneur nginx ou apache, classique...

Deux dossiers important, “data-nosithous” qui contient le fichier jar de notre application, les deux docker-compose vont venir monter un volume dessus, l’idée est que dès qu’une fonctionnalité est jugée prêtre, on génère un nouveau jar qu’on glisse dans ce répertoire.

“assets”, ce dossier est monté comme volume ne par notre bdd, il contient different images utilisé par les migration pour ajouter de la donnée de teste.