📄 Architecture Logicielle

🔍 Vue d'ensemble du projet

VendreFacile est une plateforme web de petites annonces gratuites permettant aux utilisateurs de publier, consulter et rechercher des annonces, tout en facilitant les échanges entre acheteurs et vendeurs via une messagerie interne.

Acteurs :

- Vendeur
- Client (Bool professionnel)
- Administrateur (modération, support)

🧱 Fonctionnalités retenues

Obligatoires :

- Gestion des annonces (CRUD)
- Comptes utilisateurs (inscription, connexion)
- Messagerie intégrée
- Moteur de recherche (catégorie, lieu, prix)
- Gestion de profil
- Favoris

Souhaitées intégrées :

- Géolocalisation des annonces
- UI responsive mobile

Fonctionnalités non retenues pour MVP :

- Paiement sécurisé (phase ultérieure)
- Comptes professionnels (version 2)
- Alertes (email / notif) (phase ultérieure)

🏛️ Architecture logique

Choix : Architecture microservices + API Gateway

Pourquoi ? L'architecture microservices permet de diviser l'application en services indépendants, facilitant la scalabilité horizontale et le déploiement continu. Chaque service est centré sur un domaine métier (authentification, annonces, messagerie, etc.), ce qui favorise la résilience et l'évolutivité. Cette approche est essentielle pour répondre à la contrainte de scalabilité (>10 000 utilisateurs connectés) et de disponibilité 24/7. Elle facilite aussi l'intégration future de services externes.

Permet une meilleure évolutivité, séparation claire des domaines métier, indépendance des déploiements et gestion fine des responsabilités

Couches :

1. **Frontend** : React
2. **API Gateway** : Flask
3. **Services** :
    a. AuthService -> gestion des connexions, inscription, tokens.
    b. UserService -> gestion du profil utilisateur, favoris.
    c. AnnonceService> CRUD des annonces, images, géolocalisation
    d. SearchService -> moteur de recherche multi-critères.
    e. MessageService -> messagerie interne entre utilisateurs.
    f. NotificationService (optionnel) -> emails, notifications futures.
4. **Base de données** : MariaDB / MySQL (avec partitionnement et réplication)
   Pourquoi ?
    - **Relationnel** : parfait pour les relations structurées (utilisateur ↔ annonces, messages)
    - **Solide** : mature, sécurisé, largement supporté
    - **SQL** : puissant pour des requêtes complexes (ex : recherches combinées)
    - **Partitionnement** (sharding logique) :
    - Exemple : partitionnement des annonces par région
    - **Réplication maître-esclave** possible :
    - Pour séparer les lectures/écritures
    - **Compatible avec ORM comme SQLAlchemy ou Tortoise ORM**

Communications internes : REST

📨 **Messagerie et événements asynchrones : RabbitMQ**

Pourquoi ? Utilisé pour la messagerie interne, gestion des événements (ex : nouvelle annonce, nouveau message). RabbitMQ a de très bons bindings Python (`aio_pika`, `kombu`).

Bounded Contexts (DDD)

- **Utilisateur** : gestion compte, profil
- **Annonce** : création, édition, affichage
- **Recherche** : filtres, catégories, tri
- **Messagerie** : messages internes entre utilisateurs
- **Favoris** : gestion des annonces enregistrées

Principes appliqués

DDD : Le DDD est utilisé pour structurer le code autour du domaine métier. Dans VendreFacile, les contextes liés aux utilisateurs, annonces, favoris et messagerie sont bien délimités. Cette séparation améliore la compréhension du système, la maintenance du code et l'évolution indépendante des fonctionnalités. Chaque service respecte son 'Bounded Context', ce qui réduit le couplage entre les composants

TDD : Le TDD permet d'écrire du code fiable dès le départ. Il est appliqué pour les services critiques comme l'authentification et la gestion des annonces. En écrivant les tests avant le code, on s'assure que les fonctionnalités sont testables, et cela facilite le refactoring tout en maintenant la couverture de tests élevée.

SOLID : Les principes SOLID sont respectés pour assurer la maintenabilité du code :

- S : chaque classe/service a une seule responsabilité (ex. AuthService, MessageService).

- O : les services sont conçus pour être étendus sans être modifiés.

- L : l'héritage est utilisé dans des cas contrôlés (ex : utilisateurs classiques vs pros).

- I : les interfaces sont spécifiques (ex : interface IAuthentification).

- D : les dépendances sont injectées, favorisant le test unitaire.

KISS : L'objectif est de garder chaque composant aussi simple que possible. Chaque microservice a une logique métier bien définie et ne dépend pas de la complexité des autres. Cela améliore la lisibilité, la rapidité de développement et facilite l'onboarding des nouveaux développeurs.

Base distribuée

- **Moteur** : MariaDB
- **Topologie** :

***Réplication maître-esclave (ou maître-multi-maîtres)** :

- Un **nœud principal (maître)** traite les écritures.
- Plusieurs **réplicas (esclaves)** traitent les lectures pour équilibrer la charge.

***Partitionnement logique** :

- Chaque microservice peut avoir sa propre base ou schéma.
- On peut découper les annonces par **région**, ou les messages par **utilisateur.**

- **CAP** : Choix CP (Consistency + Partition Tolerance)
  - **Consistency** : grâce à des transactions SQL, MariaDB garantit une forte cohérence des données (ACID).
  - **Partition Tolerance** : via réplication et basculement automatique possible.

  Ce compromis est crucial pour garantir que la messagerie, les favoris ou les publications ne perdent jamais de données, même en cas de panne de nœud.

- **Partitionnement / Sharding (équivalent logique) :**
  - **Par service** : chaque microservice possède sa propre base isolée.
  - **Par clé logique** (annonce par région, message par user ID).
  - MariaDB propose du **partitionnement horizontal** natif (par plage, clé, hash).