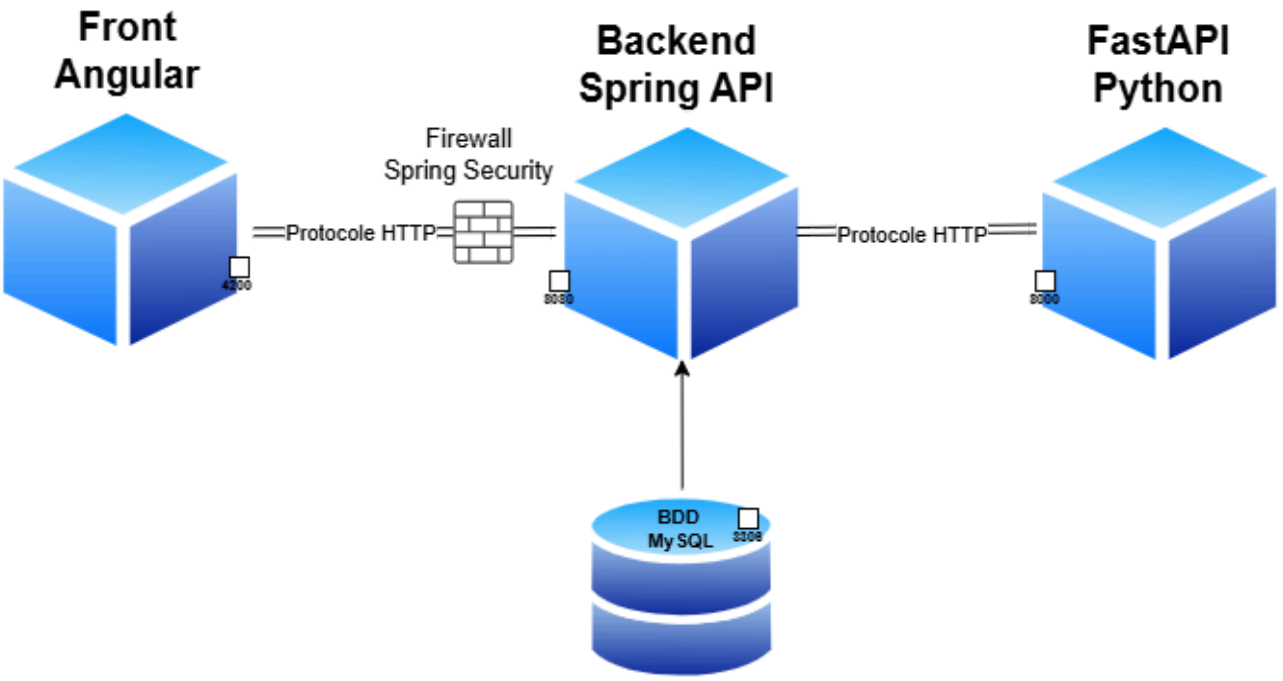
Etude de cas

GAMESUP

Documentation :

I. Diagramme d'architecture :



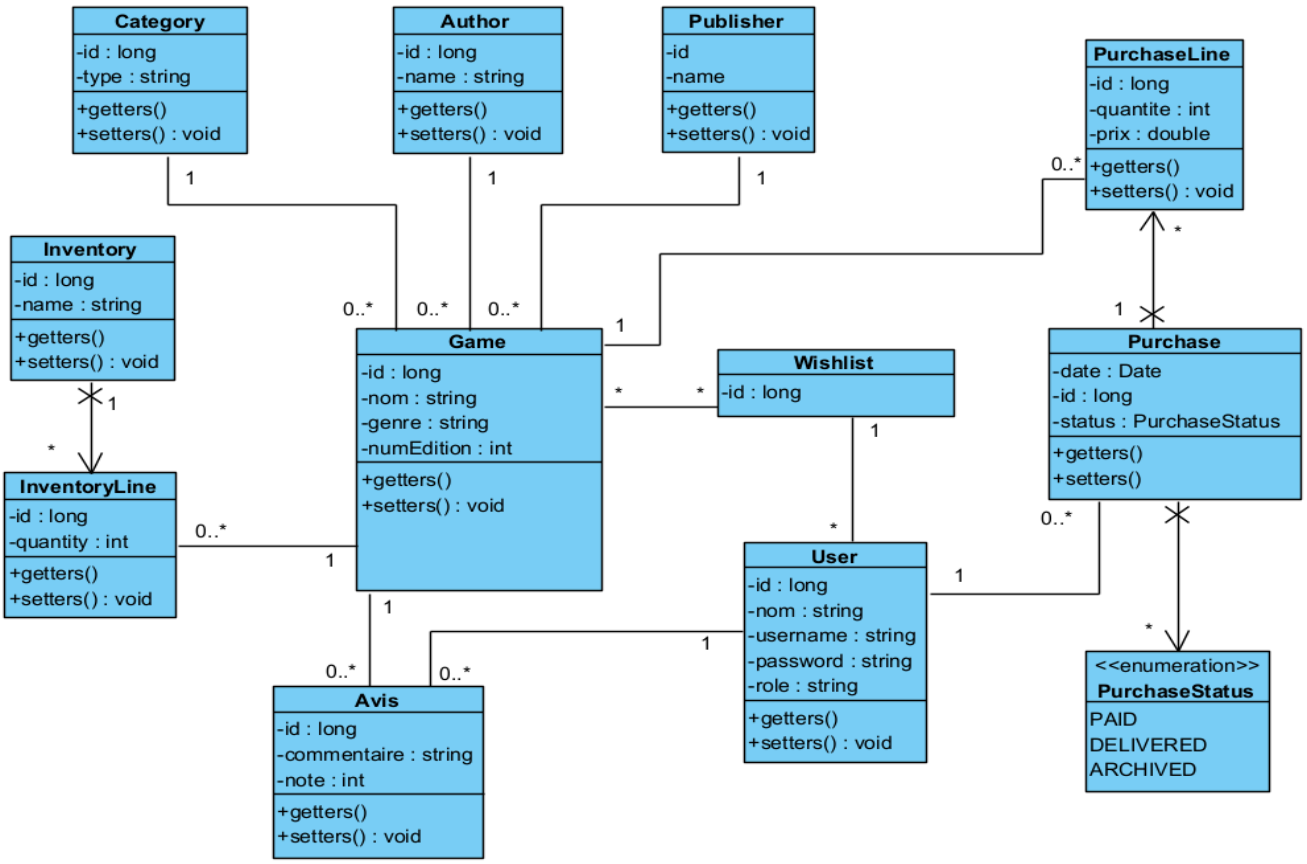
Ce diagramme illustre l'architecture globale conçue pour l'application GamesUP, organisée en plusieurs parties distinctes :

- Client : Interface utilisateur envoyant les requêtes HTTP vers l'API.
- Backend sécurisé (Spring API) : Point d'entrée principal de l'application, protégé par Spring Security via un système de jetons JWT (pare-feu applicatif).
- Service de recommandation (FastAPI) : API Python dédié aux recommandations, accessible depuis l'API via une requête HTTP.

- Base de données MySQL : Contient toutes les entités métier persistées (jeux, utilisateurs, achats...).

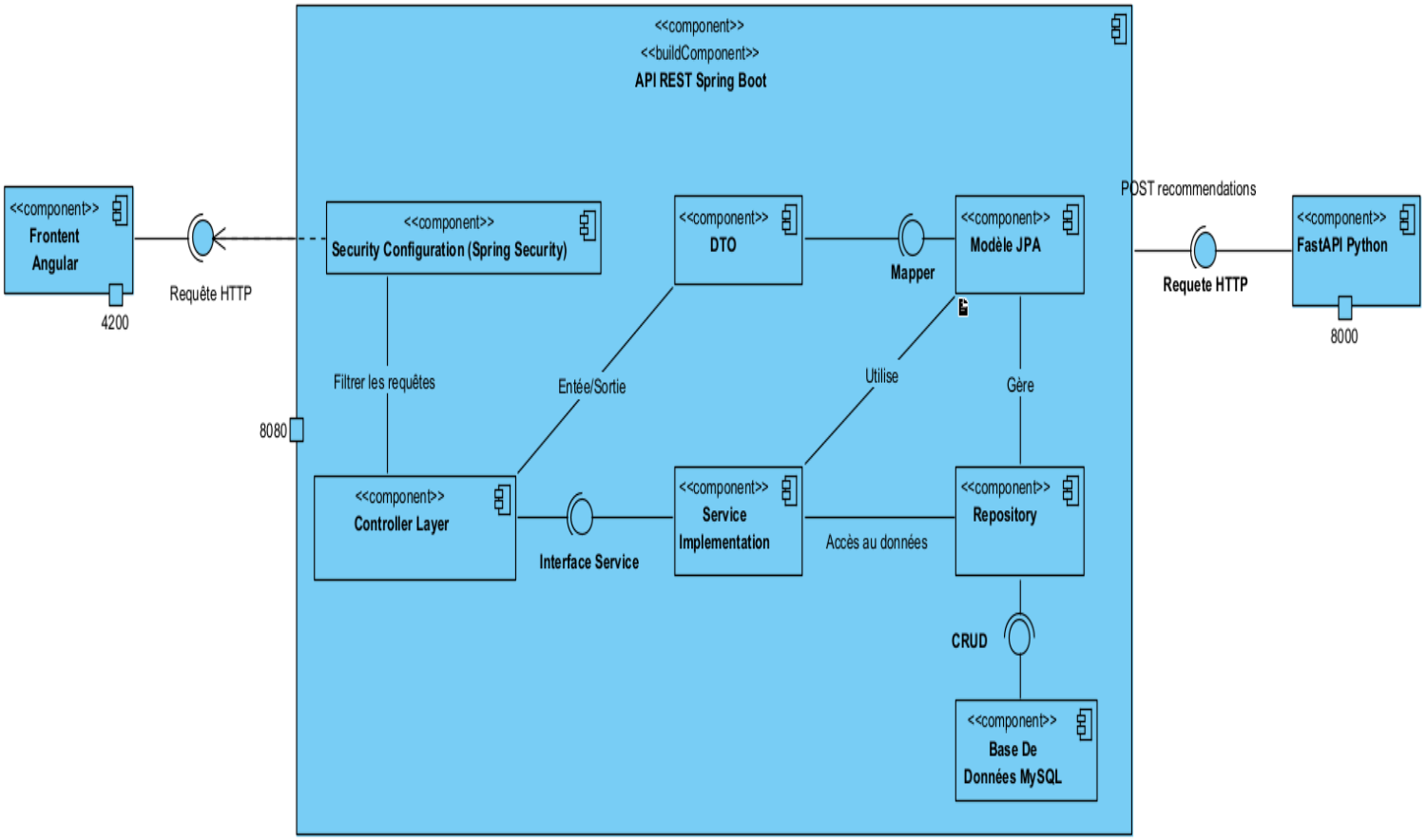
Chaque composant communique via des API REST sur des ports spécifiques, permettant une architecture modulaire, sécurisée et évolutive.

Diagramme de classe :



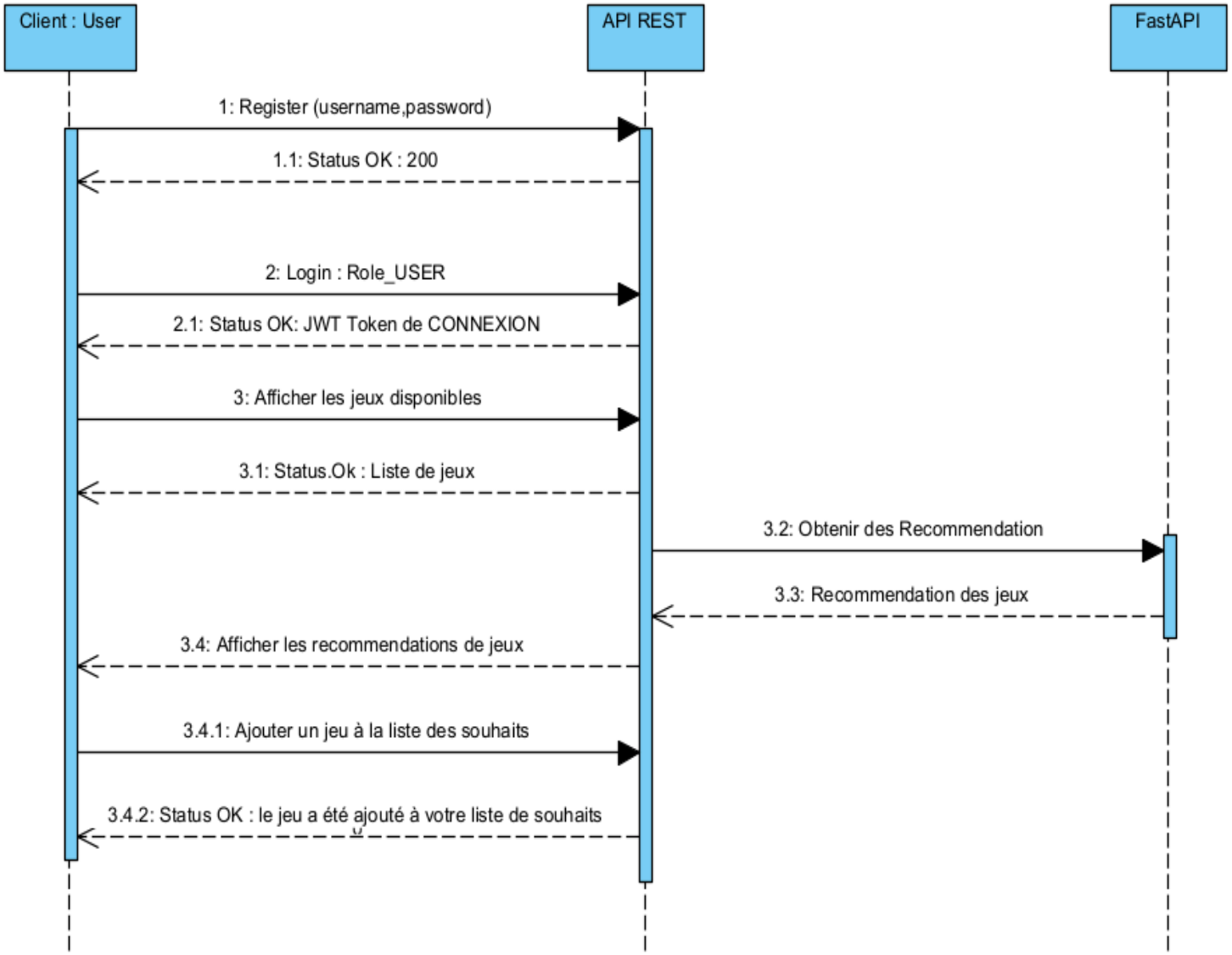
Ce diagramme de classes représente le modèle d'une application de gestion de jeux de société GamesUP, intégrant les notions d'inventaire, d'achats, d'avis, liste de souhaits et de gestion utilisateur.

II. Diagramme de composants :



Le diagramme de composants de l'application GameUP représente l'architecture modulaire de l'API Spring Boot, structurée selon une approche en couches, en intégrant les aspects de sécurité, de persistance et de communication avec le client Frontend et une API externe de recommandation.

III. Diagramme de séquence :



Ce diagramme de séquence représente les principales interactions entre un utilisateur, l'API REST (Spring Boot) et l'API de recommandation (FastAPI).

IV. Explications sur le respect des principes SOLID et les bonnes pratiques :

1. S – Single Responsibility Principle (SRP)

- Chaque Controller (ex. GameController, UserController) est responsable uniquement de la gestion des requêtes HTTP.
- Les Service (ex. GameServiceImpl) sont responsables de la logique métier.
- Les Repository interagissent uniquement avec la base de données.
- Les Class Modèles sont responsables uniquement des objets métiers

2. O – Open/Closed Principle (OCP)

- Utilisation des interfaces (UserService, GameService, etc.) et des implémentations (UserServiceImpl, GameServiceImpl). Cela permet de changer l'implémentation sans modifier les appels à ces services. La même chose pour les Controller et Models, le code est ouvert à l'extension, mais fermé à la modification.

3. L – Liskov Substitution Principle (LSP)

- Aucune hiérarchie d'héritage rigide n'est utilisée, ce qui réduit les risques de violation du LSP. Utilisation des interfaces que leurs implémentations respectent le principe.

4. I – Interface Segregation Principle (ISP)

- J'utilise des interfaces spécifiques à chaque entité (UserService, GameService, etc.), ce qui respecte bien ce principe.

5. D – Dependency Inversion Principle (DIP)

- Le principe d'inversion des dépendances est bien appliqué dans l'architecture de l'API GamesUP.
  Les modules de haut niveau (par exemple, les contrôleurs et les services) ne dépendent pas directement des modules de bas niveau (implémentations concrètes), mais bien d'abstractions représentées par des interfaces (GameService, UserService, etc.).
- L'injection des dépendances est assurée par Spring via l'annotation @Autowired ou, de manière plus propre, par l'utilisation de @RequiredArgsConstructor de Lombok, permettant une injection par constructeur.
  Cela permet de découpler les composants, de faciliter les tests unitaires (grâce aux mocks d'interfaces), et de respecter l'approche port/adapter propre à une architecture modulaire et maintenable.

Bonnes pratiques respectées :

- Utilisation d'une architecture en couches Controller → Service → Repository → Entity et la séparation des responsabilités.
- Utilisation des DTO pour séparer la logique métier du transport de données et protéger les entités contre l'exposition directe (ex : GameDTO, AvisDTO...).
- Validation côté backend en utilisant @Valid, @NotNull, @Size, etc. dans les entités ou DTOs.
- Gestion centralisée des exceptions EntityDontExistException avec @ResponseStatus(HttpStatus.NOT_FOUND) mais aussi avec un @ControllerAdvice global.
- Utilisation de JUnit + Mockito pour tester les models, services, repositories contrôleurs et configurations

V. Les rapports de Couverture de tests :

gamesUP

gamesUP

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.gamesUP.gamesUP.service.impl		81 %		67 %	70	165	30	388	48	131	2	13
com.gamesUP.gamesUP.configuration		75 %		0 %	9	21	38	106	6	18	0	4
com.gamesUP.gamesUP.controller		81 %		0 %	4	67	16	94	3	66	1	13
com.gamesUP.gamesUP.exception		80 %		n/a	2	12	5	30	2	12	0	2
com.gamesUP.gamesUP		37 %		n/a	1	2	2	3	1	2	0	1
com.gamesUP.gamesUP.model		100 %		n/a	0	1	0	4	0	1	0	1
com.gamesUP.gamesUP.config		100 %		n/a	0	2	0	2	0	2	0	1
Total	589 of 2 992	80 %	30 of 76	60 %	86	270	91	627	60	232	3	35

Le projet GameUP bénéficie d'une bonne couverture de tests avec un taux global de 80 % des instructions et 60 % des branches conditionnelles couvertes, selon le rapport généré par Jacoco.

Types de tests réalisés :

- Tests unitaires sur les services et contrôleurs à l'aide de Mockito.
- Tests d'intégration, simulant de vraies requêtes HTTP sur l'API Spring Boot.
- Tests de validation d'exceptions, comme EntityDontExistException.
- Tests de non-régression
- Tests fonctionnels

Résultats notables :

- Controller et service bien testés (81 %).
- Entités (model) entièrement couvertes (100 %).
- Quelques branches (if/else, catch) restent à couvrir.

- La configuration Spring est partiellement testée (75 %).

Recommandations :

- Ajouter des tests pour les cas d'erreurs et exceptions.
- Couvrir les branches (if/else, catch), les cas limites et les composants de configuration.

VI. Le travail effectué pour mettre en place le système de recommandation :

Objectif :

Développer un système de recommandation de jeux de société pour les utilisateurs en fonction de leurs interactions (achats, notes, wishlist, etc). Ce système doit :

- Être intégré à une API FastAPI Python
- Communiquer efficacement avec l'API Spring Boot
- Être prêt à l'entraînement, même en l'absence d'un grand jeu de données réels.

1. Collecte et modélisation des données :

Les données sont basées sur les entités métiers :

- user_id : identifiant utilisateur,
- game_id : identifiant du jeu,
- note : note donnée par l'utilisateur (de 1 à 5),
- quantité : nombre achetés,
- purchase : indicateur d'achat,
- wishlist : présence dans la liste de souhaits.

Ces données sont stockées dans un fichier CSV.

2. Nettoyage et transformation de données :

- Nettoyage : suppression des lignes invalides, conversion des types, suppression des notes hors plage
- Normalisation des champs purchase et wishlist (True/False).

- Construction d'une matrice `user_game_matrix` à partir des notes.

```
# Vérification des colonnes attendues

expected_cols = ['user_id', 'game_id', 'game_name', 'note', 'purchase',
'quantité', 'wishlist']

if not all(col in data.columns for col in expected_cols):

    raise ValueError("Colonnes manquantes dans le fichier CSV")



# Supprimer les lignes avec valeurs critiques manquantes

data.dropna(subset=["user_id", "game_id", "note", "quantité",
"wishlist"], inplace=True)



# Forcer les bons types

data["user_id"] = data["user_id"].astype(int)

data["game_id"] = data["game_id"].astype(int)

data["note"] = data["note"].astype(int)

data["quantité"] = data["quantité"].astype(int)

data["purchase"] = data["purchase"].astype(str).str.lower()

data["wishlist"] = data["wishlist"].astype(str).str.lower()
```

```
# Supprimer les notes hors plage

data = data[(data["note"] >= 0) & (data["note"] <= 5)]



# Supprimer les quantités aberrantes (ex : négatives ou trop grandes)

data = data[(data["quantité"] >= 1) & (data["quantité"] <= 50)]



# Normaliser les champs "purchase"

data["purchase"] = data["purchase"].apply(lambda x: 1 if
str(x).strip().lower() in ["true", "vrai", "yes", "1"] else 0)



#Normaliser les champs "wishlist"

data["wishlist"] = data["wishlist"].apply(lambda x: 1 if
str(x).strip().lower() in ["true", "vrai", "yes", "1"] else 0)




# Nettoyage et transformation

# Création de la matrice utilisateur-jeu (On peut utiliser une note
pondérée qui contient note, quantité, purchase et wishlist)

user_game_matrix = data.pivot_table(index='user_id', columns='game_id',
values='note').fillna(0)
```

3. Analyse exploratoire :

- Afficher une matrice de corrélation entre les jeux.
- Vérification de la cohérence des interactions.

4. Utilisation du modèle KNN :

- Utilisation de scikit-learn et de l'algorithme NearestNeighbors avec distance cosinus.

- Le modèle KNN est entraîné sur la matrice `user_game_matrix`
- Sauvegarde du modèle au format .joblib et de la matrice pour l'utilisation dans FastAPI.

```
#Enregistrement du modele

joblib.dump(model_knn, "sample_data/knn_model.joblib")

user_game_matrix.to_pickle("sample_data/user_game_matrix.pkl")
```

5. Génération des recommandations :

- à l'envoie d'une requête, les données sont récupérés dans la matrice à l'aide de la fonction :

```
def generate_recommendations(user_data: UserData, top_n=3):
```

- Le modèle trouve les utilisateurs similaires via KNN.

- Il agrège les jeux les mieux notés parmi les voisins proches.

- Il exclut les jeux déjà notés.

- Il retourne une liste triée de jeux recommandés.

6. Communication entre Spring Boot et FastAPI

- L’API Spring Boot envoie les données utilisateur (UserData) via RestTemplate à l’endpoint Python POST /recommendations.

- FastAPI traite la demande, génère la recommandation, et renvoie une liste de jeux.
- L’API Spring Boot récupère et affiche la recommandation générée.

Conclusion et perspectives :

- Le système est modulaire, extensible et prêt pour l’intégration de vraies données.

- Il utilise des concepts simples de ML (KNN collaboratif) pour donner des résultats pertinents.

- Il est évolutif : on pourra affiner ou modifier les scores en combinant plusieurs sources (note + achat + wishlist pondérés).

- L'architecture assure une séparation des responsabilités entre traitement ML et gestion métier Java/Spring.

VII. Réflexion sur les bonnes et mauvaises pratiques :

Bonnes pratiques

1. Architecture claire, modulaire et évolutive

- Séparation des couches applicatives :Controller pour exposer les endpoints REST, Service pour la logique métier, Repository pour l'accès aux données.
- Les entités sont bien définies et respectent la normalisation.

2. le respect des principes SOLID

(voir le chapitre dédié)

3. Utilisation de DTOs

- Les DTOs permettent de séparer les objets métiers (entités JPA) de ceux exposés à l'extérieur (API).

- Cela réduit les effets de bord et augmente la sécurité (pas d'exposition involontaire de champs sensibles).

4. Plusieurs types de tests

- De nombreux tests unitaires sont mis en place pour les services et contrôleurs.
- Des tests d'intégration qui testent l'interaction entre plusieurs composants réels (ex. : service + base de données réelle ou simulée)
- Des tests de bout en bout (E2E) assurent la fiabilité globale des API en testant les appels HTTP de bout en bout.
- Tests de couverture qui donnent une analyse de la couverture du code source par les tests.

5. Documentation OpenAPI (Swagger)

- L'API est documentée automatiquement avec Swagger grâce à l'intégration springdoc-openapi, Swagger permet aussi l'utilisation des JWT pour l'authentification.

6. Sécurité avec JWT

- Mise en place d'un système de filtrage via JWT et d'un firewall applicatif pour sécuriser les endpoints sensibles.
- Mettre en place un système de contrôle d'accès basé sur les rôles afin de distinguer les droits entre les utilisateurs simples et les administrateurs.
- Respect de la méthodologie OWASP

7. Utilisation d’un modèle KNN simple et fonctionnel

- Le modèle est simple, interprétable, et bien encapsulé dans un microservice Python.

Mauvaises pratiques ou points d’amélioration

1. Pas de logging centralisé ni traçabilité fine

- Le projet pourrait bénéficier d’un système de journalisation mieux structuré (Log4j2 est intégré mais peu exploité).

2. Séparation des Mappers

- Dans certains services (comme PurchaseServiceImpl), la méthode mapToDTO() contient plusieurs responsabilités (validation, transformation, logique métier). Il serait mieux de séparer cette logique dans un mapper utilitaire (PurchaseMapper par exemple).

3. Ajouter des tests pour augmenter la couverture

- Vérifier qu’une exception est bien levée si un auteur n’est pas trouvé.
- Vérifier qu’un PATCH ne modifie pas certains champs interdits.

4. Modèle KNN non encore entraîné dynamiquement

- Le modèle est statique et ne s’adapte pas en temps réel aux nouvelles données.
- Une évolution future serait d’intégrer une pipeline d’entraînement périodique ou en ligne.