Projet 5BDDD : Introduction aux structures BDD Data et API

SUJET : Système de gestion de bibliothèque en ligne

Fadi DJELOUAH

Modeste KOUASSI

Hence MULUNDA

Description du projet :
Le projet consiste à développer une API de gestion de bibliothèque en ligne, permettant aux utilisateurs d'emprunter et de rendre des livres, tout en gérant l'inventaire des livres et les utilisateurs inscrits. Ce projet utilisera Oracle comme base de données, Python pour la logique métier, FastAPI pour le développement de l'API, SQLAlchemy et Alembic pour la gestion des interactions avec la base de données et la gestion des migrations, et Pydantic pour la validation des données.

Lien de notre repository github avec le projet final :
https://github.com/modeste15/books-api

Introduction

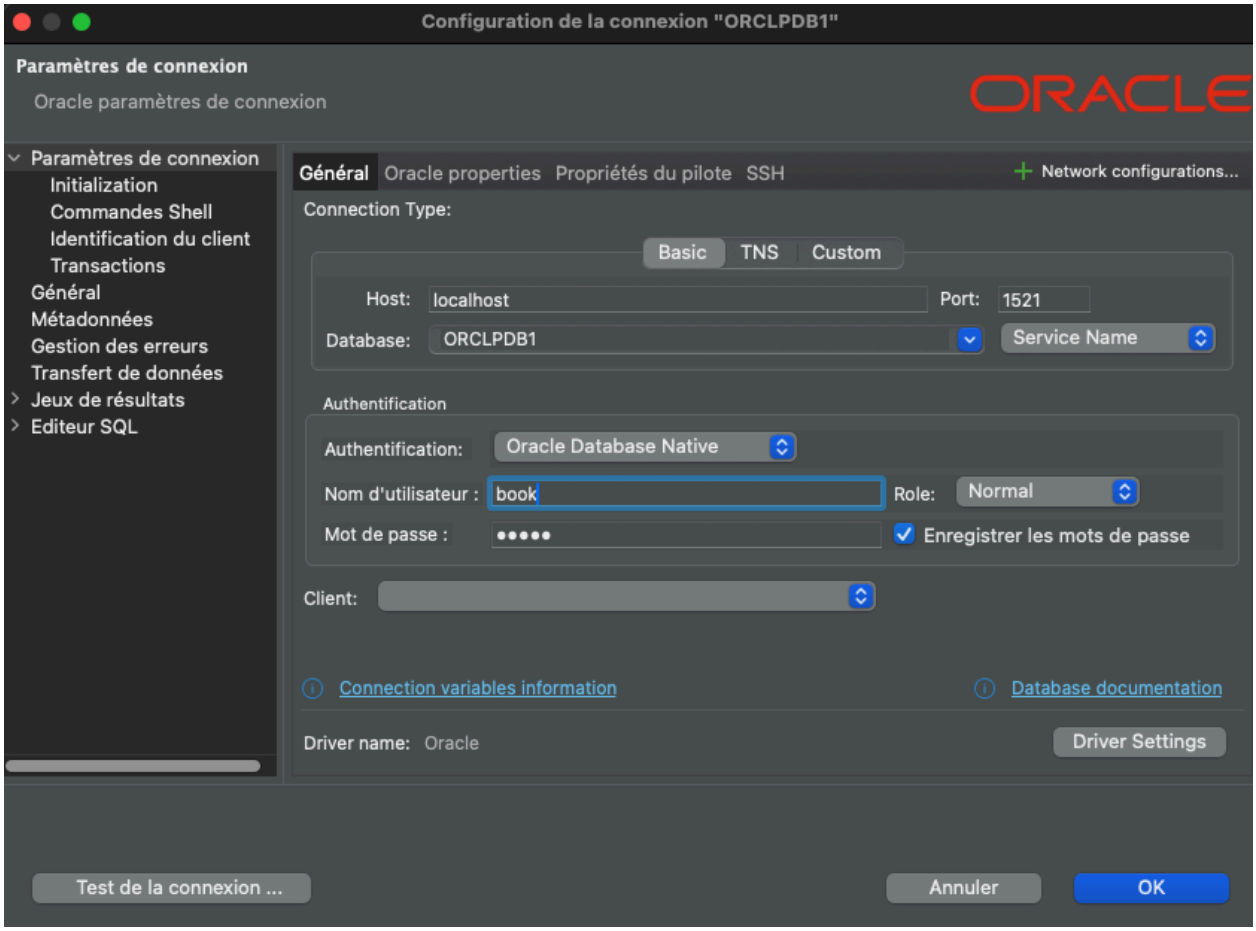
Le projet consiste à développer une API de gestion de bibliothèque en ligne, permettant aux utilisateurs d'emprunter et de rendre des livres, tout en gérant l'inventaire des livres et les utilisateurs inscrits. L'infrastructure repose sur Oracle pour la base de données, Python pour la logique métier, FastAPI pour le développement de l'API, SQLAlchemy et Alembic pour la gestion des interactions et migrations de base de données, et Pydantic pour la validation des données. L'objectif est de fournir une solution robuste, efficace et évolutive pour gérer une bibliothèque en ligne.

1. Installation d'Oracle

La première étape de notre projet a consisté à installer et configurer Oracle comme base de données principale. Oracle a été choisi en raison de sa performance, de ses capacités à gérer des transactions complexes et de sa stabilité pour des applications à grande échelle.
Dans notre projet nous utilisons ORACLE avec un container Docker.

2. Connexion à Oracle via DBeaver

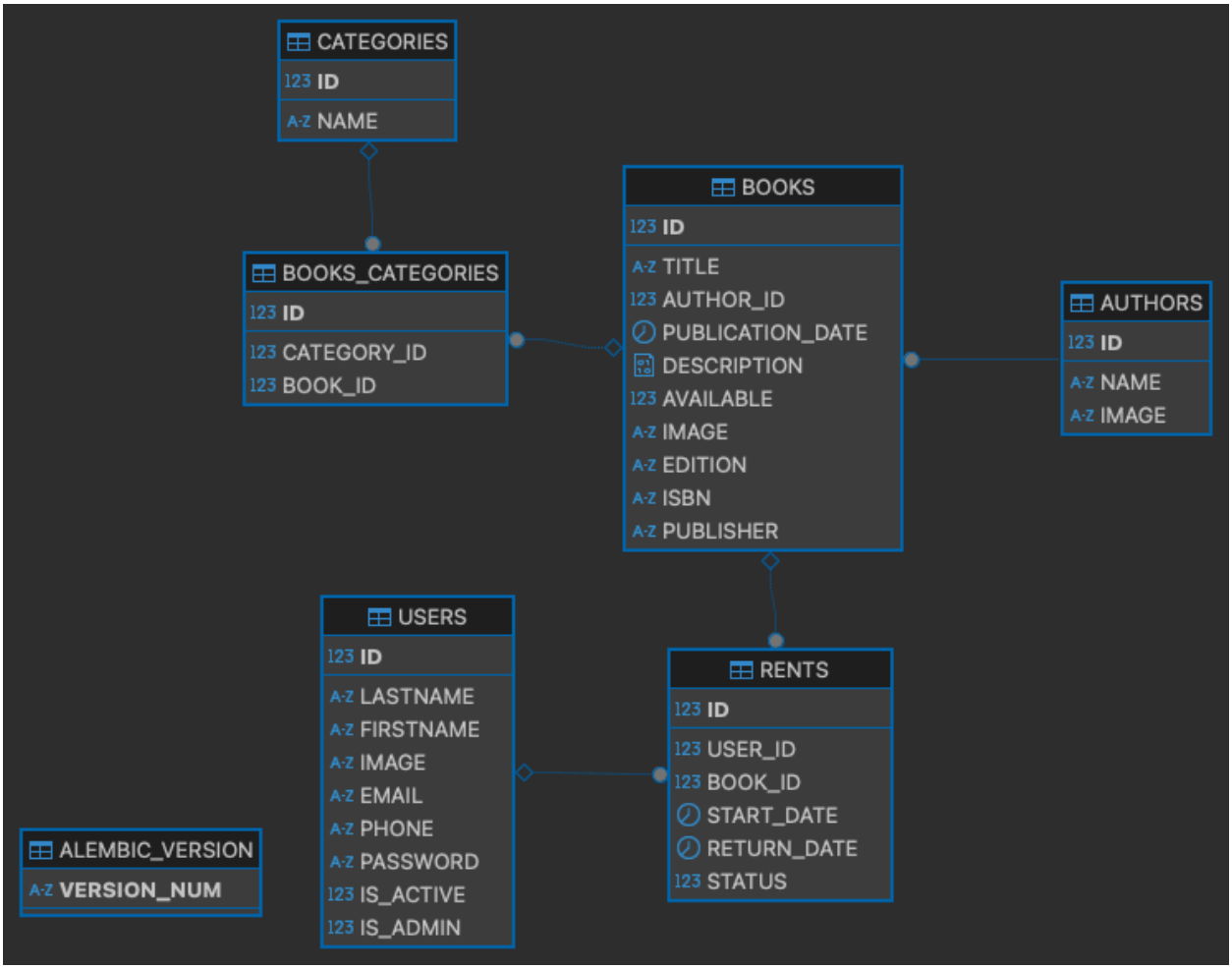
Pour faciliter la gestion et la manipulation de la base de données, nous avons utilisé DBeaver comme interface de gestion. DBeaver permet une connexion intuitive à Oracle, facilitant la gestion des schémas de base de données, l'exécution des requêtes SQL et l'administration des données.



3. Modélisation de la base de données

La modélisation de la base de données a été une étape clé dans la conception de l'API. Le modèle relationnel a été conçu pour inclure les entités suivantes :

- Users : Stocke les informations sur les utilisateurs inscrits (nom, email, téléphone, etc.).
- Books : Gère les informations sur les livres (titre, auteur, date de publication, genre, disponibilité, etc.).
- Rents : Enregistre les emprunts et les retours de livres par les utilisateurs.
- Genres : Définit les genres littéraires associés aux livres.
- Authors : Contient les informations sur les auteurs des livres.



4. Développement avec FastAPI

FastAPI a été utilisé pour développer l'API en raison de sa rapidité et de ses fonctionnalités modernes. FastAPI offre également des outils intégrés comme Swagger pour documenter l'API et OpenAPI pour standardiser les interactions.

4.1 Configuration du .env

```
# connect oracledb
DATABASE_USER=demo
DATABASE_PASSWORD=demo1
DATABASE_DSN=localhost:1521/ORCLPDB1

DATABASE_TABLE=SCHEMA.TABLE_NAME

# POur SQLalchemy:
SQLALCHEMY_DATABASE_URL=oracle+oracledb://demo:demo1@localhost:1521/?service_name=ORCLPDB1


# SECRET  pour l'encryption des jetons d'authentification
SECRET_KEY = "cat"
ALGORITHM = "HS256"
ACCESS_TOKEN_EXPIRE_MINUTES = 60
```

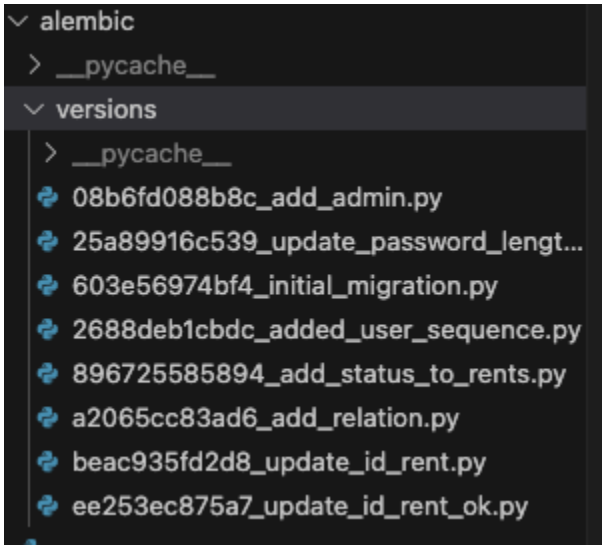
4.2 Transcription de la base de données

La transcription de la base de données a été effectuée en utilisant SQLAlchemy pour créer les modèles de données et Alembic pour la gestion des migrations. Cette étape a permis de traduire la modélisation conceptuelle en objets manipulables dans l'API, tout en assurant la cohérence et l'intégrité des données à travers les migrations successives.

- Exemple du MODEL USER

```python
from sqlalchemy import Column, Text,Integer, String, ForeignKey, DateTime, Boolean, Date,Sequence
from sqlalchemy.orm import relationship,deferred
from .database import Base



metadata = Base.metadata


class User(Base):
    __tablename__ = 'users'
    id = Column(Integer,  Sequence('user_seq')  , primary_key=True,autoincrement=True)
    lastname = Column(String(50))
    firstname = Column(String(50))
    image = Column(String(50))
    email = Column(String(50))
    phone = Column(String(50))
    password = deferred(Column(String(250)))
    is_active = Column(Boolean, default=True)
    is_admin = Column(Boolean, default=True)

    rents = relationship("Rent", back_populates="user")
```

- Versionning Alembic qui gère les migrations



4.2 Insertion des données à partir de Kaggle

Les données de test pour les livres et les utilisateurs ont été insérées via des jeux de données provenant de Kaggle. Ces données ont servi à tester la fonctionnalité de l'API et à simuler des scénarios d'emprunt et de retour.

Lien Kaggle : https://www.kaggle.com/datasets/saurabhbagchi/books-dataset

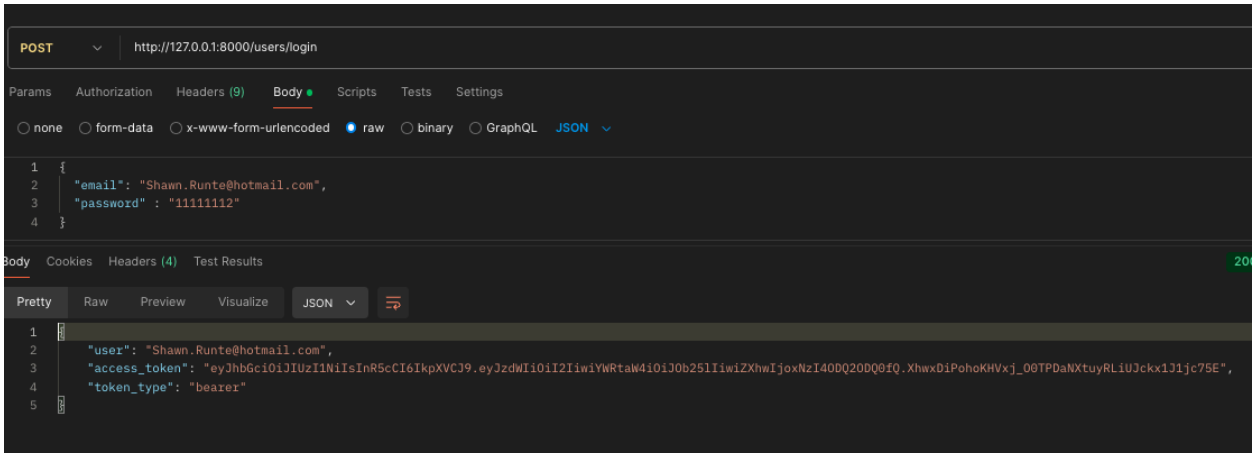
4.3 Structure des fichiers de l'API

La structure du projet suit une organisation modulaire :

- app/ : Contient les principaux modules de l'API.
- models/ : Contient les modèles SQLAlchemy pour les entités (Users, Books, Rents).
- routes/ : Gère les routes de l'API (inscription, gestion des livres, emprunts).
- validators/ : Définit les schémas Pydantic utilisés pour la validation des données.
- tests/ : Inclut les tests unitaires et fonctionnels.
- main.py : Point d'entrée principal de l'API.

4.4 Implémentation de l'authentification

L'authentification des utilisateurs a été implémentée en utilisant le système de jetons JWT (JSON Web Token). Cela permet aux utilisateurs de s'authentifier et de maintenir des sessions sécurisées tout en interagissant avec l'API.



4.5 Validation avec Pydantic

Pydantic a été utilisé pour la validation des données, garantissant que toutes les informations transmises à l'API respectent les contraintes définies dans les modèles. Chaque point d'entrée de l'API utilise des schémas Pydantic pour s'assurer que les données sont bien formatées avant d'être traitées. Voici une exemple de validation sur le model User.

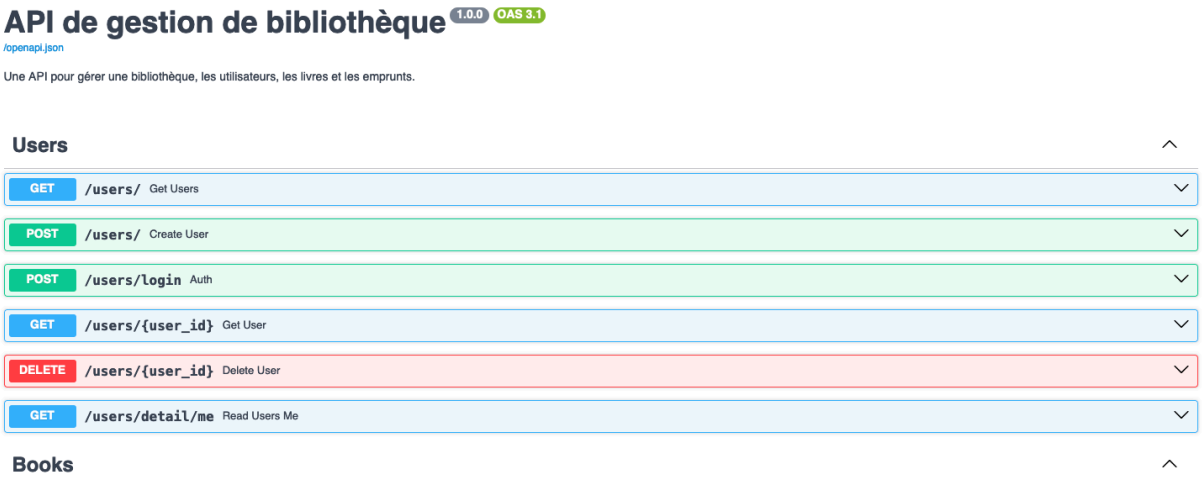
```python
class UserSchema(BaseModel):
    lastname: str
    firstname: str
    email: EmailStr
    phone: str
    password: str
    image : str

    @validator('password')
    def validate_password(cls, value):
        if len(value) < 8:
            raise ValueError('Le mot de passe doit comporter au moins 8 caractères.')
        if not re.search("[a-z]", value):
            raise ValueError('Le mot de passe doit contenir au moins une lettre minuscule.')
        if not re.search("[A-Z]", value):
            raise ValueError('Le mot de passe doit contenir au moins une lettre majuscule.')
        if not re.search("[0-9]", value):
            raise ValueError('Le mot de passe doit contenir au moins un chiffre.')
        if not re.search("[@#$%^&+=]", value):
            raise ValueError('Le mot de passe doit contenir au moins un caractère spécial (@, #, $, %,
etc.).')
        if all(char == value[0] for char in value):
            raise ValueError('Le mot de passe ne doit pas contenir des caractères identiques en
séquence.')
        return value

    @validator('phone')
    def validate_phone(cls, value):
        if not re.match(r'^\+?\d{9,15}$', value):
            raise ValueError('Numéro de téléphone invalide. Il doit contenir entre 9 et 15 chiffres.')
        return value
```

4.6 Tests unitaires

Les tests unitaires ont été écrits pour garantir la stabilité et la robustesse de l'API. Ils couvrent toutes les principales fonctionnalités, notamment l'inscription des utilisateurs, l'emprunt de livres, la gestion des retours et la manipulation de l'inventaire.

4.7 Documentation Swagger

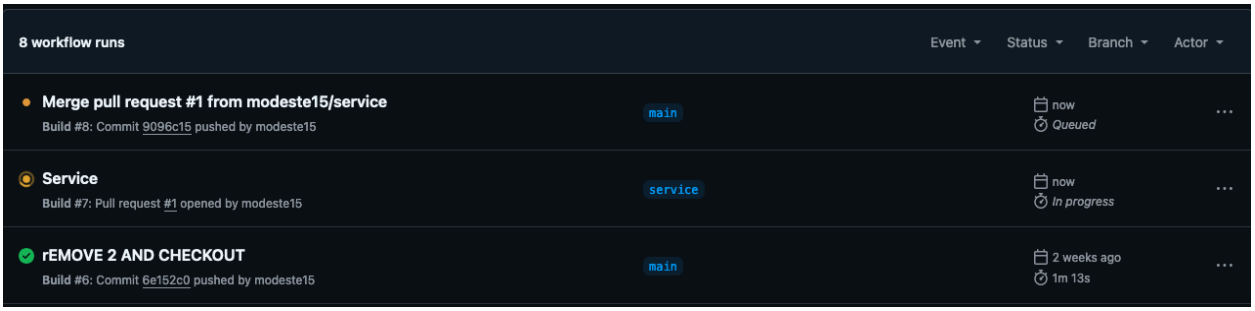
L'API est auto-documentée grâce à Swagger, accessible via l'URL /docs. Cette documentation dynamique permet aux développeurs de visualiser et de tester les points d'entrée de l'API de manière interactive, tout en consultant les détails sur les méthodes, les paramètres, et les réponses attendues.

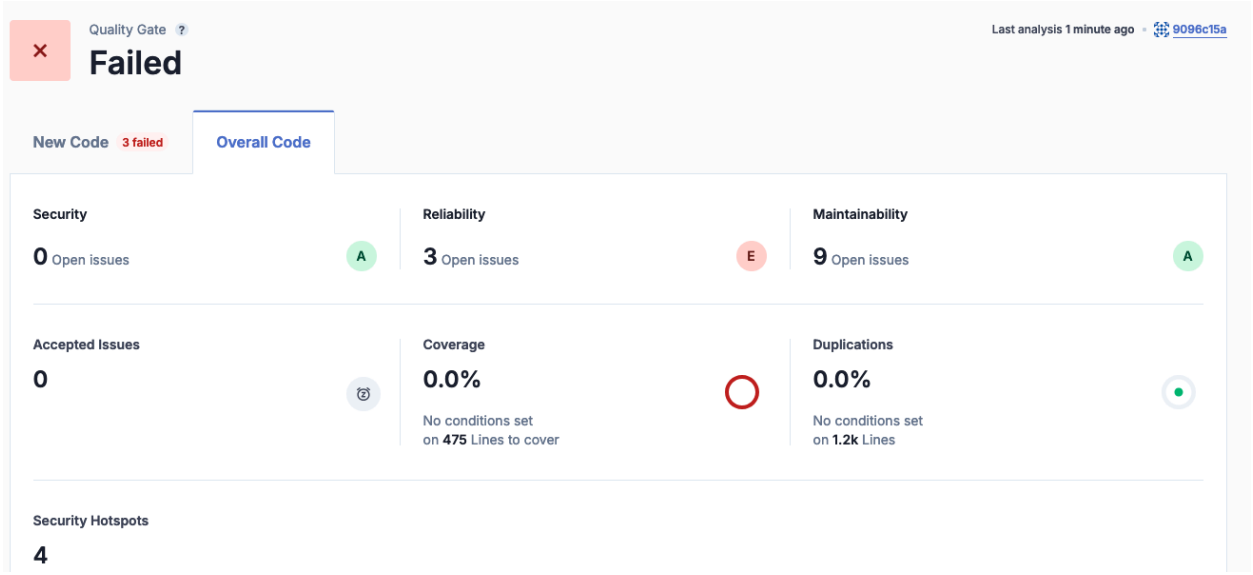


5. Intégration du workflow GitHub et SonarQUBE

Pour assurer la qualité continue du code, l'intégration de GitHub Actions et de SonarQUBE a été réalisée. Le workflow GitHub permet de lancer automatiquement les analyses de qualité du

code à chaque push. SonarQUBE est utilisé pour surveiller la dette technique, les vulnérabilités de sécurité et la couverture des tests.

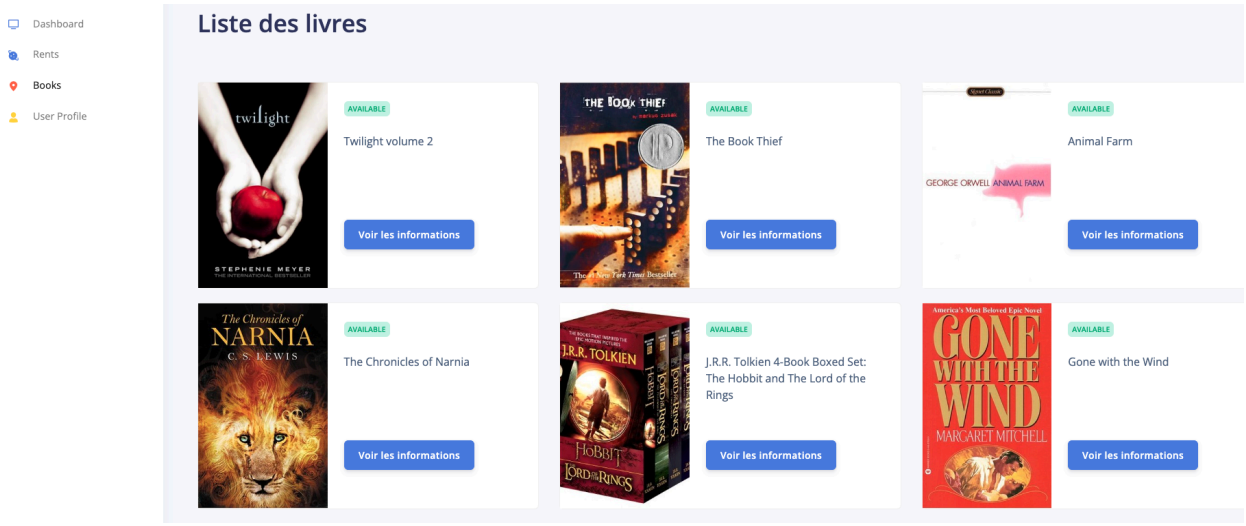




6. Exemple de l'utilisation de l'API

Grâce à l'architecture flexible que nous avons mise en place, nous avons la capacité de développer des microservices indépendants qui interagissent entre eux via des API. Cela permet une meilleure scalabilité et modularité de l'application, en facilitant l'ajout de nouvelles fonctionnalités ou services sans impacter l'ensemble du système. Chaque microservice peut être développé, déployé, et maintenu indépendamment, ce qui améliore la réactivité et la résilience de l'application.

En complément, nous avons intégré des fonctionnalités supplémentaires avec ReactJS pour la partie front-end. Cette approche offre une expérience utilisateur plus fluide et interactive, en particulier pour la gestion des interfaces liées aux interactions utilisateur, comme la visualisation de l'inventaire des livres ou la gestion des emprunts



Conclusion

Ce projet d'API de gestion de bibliothèque en ligne constitue une solution complète pour gérer les utilisateurs, les livres et les emprunts de manière efficace. Grâce à FastAPI, SQLAlchemy, Pydantic et Oracle, nous avons réussi à fournir une architecture robuste et évolutive tout en maintenant des standards élevés de sécurité et de validation des données. L'intégration des tests unitaires et d'outils de qualité de code assure la pérennité et la fiabilité de l'API.