RAPPORT

L’étude et les correctifs du code fourni

Quand j’ai commencé ce projet pour moderniser le système de gestion de la médiathèque « Notre livre, notre média », j’ai hérité d’un code initial créé par un autre développeur, mais il était limité et de faible qualité. Ce code était une application en mode console avec quelques lignes pour des classes modèles et des fonctions simples, et je me suis rendu compte qu’il fallait tout reprendre pour le transformer en une application web fonctionnelle avec Django.

Ce que j’ai trouvé dans le code initial

Le code incluait des classes comme **livre, dvd, cd, jeuDePlateau, et Emprunteur**, mais elles étaient très basiques. Par exemple, les attributs étaient simplement des variables statiques comme name = "" ou auteur = "", sans aucune logique pour gérer les données dynamiquement. Il y avait aussi des fonctions comme menu(), **menuBibliotheque(), et menuMembre()** qui affichaient uniquement des messages avec print(), mais aucune interaction réelle ou persistance des données. Il n’y avait ni relations entre les classes, ni gestion des emprunts, ni structure adaptée pour une application web. En résumé, c’était une ébauche conceptuelle, mais pas du tout prête

.

Ce que j’ai décidé de garder

Malgré ses défauts, j’ai conservé les concepts principaux, comme l’idée d’avoir des catégories différentes pour les médias (livres, DVDs, CDs, jeux) et un modèle pour les emprunteurs. J’ai également repris les noms des entités, mais je les ai standardisés et améliorés, par exemple en renommant livre en Book, jeuDePlateau en BoardGame, etc.

Ce que j’ai refait et corrigé

J’ai décidé de tout reconstruire avec Django pour en faire une application web, comme demandé. Voici ce que j’ai fait :

- **Passage à Django** : J’ai créé deux applications distinctes, bibliothecaire pour les bibliothécaires et membre pour le public, au lieu de l’application console. J’ai utilisé Django pour gérer une base de données SQLite et offrir une interface web sécurisée.
- **Amélioration des modèles** : J’ai transformé les classes en modèles Django avec models.Model. Pour optimiser la structure, j’ai introduit une classe mère **Media** comme base commune pour les médias empruntables. J’ai fait hériter **Book, DVD, et CD de Media**, ce qui permet de partager des attributs comme title, available, borrow_date, borrower, et empruntable. En revanche, j’ai laissé BoardGame comme une classe séparée, car il n’est pas empruntable, avec un champ empruntable = False.
    - **Pourquoi cette approche est meilleure** : Utiliser une classe mère abstraite Media avec héritage est plus efficace, car cela évite la redondance du code. Par exemple, au lieu de dupliquer les champs title, available, etc., dans

chaque classe Book, DVD, et CD, je les centralise dans Media. Cela rend le code plus maintenable, réduit les erreurs potentielles, et facilite l'ajout de nouvelles fonctionnalités ou contraintes (comme is_overdue()) pour tous les médias empruntables en une seule modification. De plus, cela respecte les principes de programmation orientée objet, notamment l'héritage et la réutilisation du code.

- J'ai aussi ajouté des champs comme first_name et email pour Borrower, ainsi qu'un champ blocked pour gérer les restrictions d'emprunt. J'ai corrigé les noms (par exemple, name → title, auteur → author) et ajouté des relations avec ForeignKey pour lier Borrower aux médias empruntés.
- **Ajout de la logique métier** : J'ai implémenté une méthode is_overdue() dans Media pour vérifier si un média est en retard (après 7 jours), respectant une des contraintes métiers.
- **Sécurisation et fonctionnalités web** : J'ai protégé les vues des bibliothécaires avec @login_required pour qu'elles ne soient accessibles qu'avec un mot de passe, et j'ai créé une application publique pour les membres. J'ai aussi ajouté les fonctionnalités comme créer, mettre à jour, supprimer des membres et médias, gérer les emprunts et retours, tout ça avec des formulaires web et des redirections.
- **Qualité du code** : J'ai veillé à respecter les conventions Python (PEP 8) et Django, comme utiliser des noms en snake_case, documenter les modèles avec __str__, et organiser le code en modules clairs.

En conclusion, j'ai repris les idées de base du code initial, mais j'ai effectué une refonte complète pour en faire une application web robuste, sécurisée, et fonctionnelle avec Django, en corrigeant les failles et en ajoutant les fonctionnalités nécessaires pour la médiathèque.

Stratégie de tests

Pour m'assurer que le système de gestion de la médiathèque fonctionne correctement et respecte toutes les exigences, j'ai mis en place une stratégie de tests rigoureuse en utilisant le framework de tests intégré de Django. Voici comment j'ai abordé les tests pour valider mon application.

Outils utilisés

J'ai utilisé django.test.TestCase comme base pour mes tests, car il permet de créer des cas de test isolés avec une base de données temporaire. J'ai également employé le Client pour simuler des requêtes HTTP, ce qui me permet de tester les vues comme un utilisateur réel interagirait avec l'application. Cela inclut les tests pour les redirections, les codes de statut HTTP, et le contenu des réponses.

Couverture des tests

J'ai conçu mes tests pour couvrir toutes les fonctionnalités demandées par l'énoncé, ainsi que les contraintes métiers spécifiques. Voici une répartition détaillée :

- **Application bibliothécaire (dans** bibliothecaire/tests.py**)** :

    - **Affichage et navigation** : J'ai testé l'accès à la page d'index (test_index) pour vérifier qu'elle est protégée et accessible uniquement après connexion. J'ai aussi vérifié l'accès sans connexion (test_acces_sans_connexion) pour confirmer la redirection vers /login/.
    - **Gestion des membres** : J'ai créé un test pour afficher la liste des membres (test_borrower_list), un test pour créer un nouveau membre (test_borrower_create), un test pour mettre à jour un membre (test_borrower_update), et un test pour supprimer un membre (test_borrower_delete).
    - **Gestion des médias** : J'ai testé l'affichage de la liste des médias (test_media_list) pour tous les types (livres, DVDs, CDs, jeux de plateau). J'ai aussi ajouté un test pour créer un nouveau média, en particulier un livre (test_media_create_book).
    - **Gestion des emprunts** : J'ai testé la création d'un emprunt (test_borrow_media) et le retour d'un média (test_return_media) pour vérifier les changements dans la disponibilité et les associations avec les emprunteurs
    - **Contraintes métiers** : J'ai implémenté des tests spécifiques pour valider les contraintes de l'énoncé :
        - test_max_3_emprunts pour vérifier qu'un membre ne peut pas avoir plus de 3 emprunts simultanés.
        - test_pas_emprunt_si_retard pour s'assurer qu'un membre avec un emprunt en retard (plus de 7 jours) ne peut pas emprunter à nouveau.
        - test_jeux_non_empruntables pour confirmer que les jeux de plateau ne peuvent pas être empruntés.
- **Application membre (dans** membre/tests.py**)** :
    - J'ai testé l'affichage public de la liste des médias (test_media_list) pour vérifier que tous les types de médias (livres, DVDs, CDs, jeux de plateau) sont accessibles sans authentification.

Au total, j'ai écrit 14 tests : 13 pour l'application bibliothécaire et 1 pour l'application membre, assurant une couverture complète des fonctionnalités et des contraintes.

Préparation des tests

Dans la méthode setUp de chaque classe de test, j'ai initialisé un environnement commun pour chaque test. Pour l'application bibliothécaire, j'ai créé un utilisateur bibliothécaire authentifié (biblio avec mot de passe test123), un client HTTP, et des données de test. Pour l'application membre, j'ai créé des instances de chaque type de média sans authentification, simulant un accès public.

Exécution et résultats

Les tests sont lancés avec la commande python manage.py test. Tous les tests passent avec succès, ce qui confirme que les fonctionnalités et les contraintes sont correctement implémentées. Par exemple, les tests sur les contraintes métiers (max 3 emprunts, retards, jeux non empruntables) valident que l'application respecte les règles métier, tandis que les tests sur les vues protégées vérifient la sécurité.

Pourquoi cette stratégie ?
J'ai choisi cette approche pour couvrir chaque aspect fonctionnel de l'application, tout en validant les contraintes métiers spécifiques à la médiathèque. En simulant des interactions utilisateur avec Client, j'ai pu tester le comportement réel de l'application, y compris les redirections et les messages d'erreur. En résumé, ma stratégie de tests m'a permis de valider que l'application répond aux besoins de l'énoncé.

Instructions pour exécuter le programme depuis n'importe quelle machine (sans prérequis)

Cette application de gestion de la médiathèque a été développée avec Django sous PyCharm (version gratuite) et Python, et est hébergée sur un repository GitHub. Pour exécuter le programme, sans prérequis spécifiques autres que Python et Git , les étapes suivantes sont proposées :

1. Cloner le repository GitHub : Ouvrir un terminal ou une invite de commande et exécuter la commande suivante pour cloner le repository **git clone https://github.com/Papinte/mediatheque_project**

2. Ensuite, se déplacer dans le dossier cloné : **cd mediatheque_project/gestion_mediatheque**

3. Disposer de Python et Git : Python (au moins la version 3.8) et Git sont requis. Ces outils, disponibles gratuitement.

4. Créer un environnement virtuel : Exécuter la commande suivante pour créer un environnement virtuel isolé : **python -m venv venv**

5. Activer l'environnement virtuel : **venv\Scripts\activate**

6. L'apparition de (venv) devant le prompt confirme que l'environnement est actif.

7. Installer Django : Installer Django, avec la commande : **pip install django**

8. Appliquer les migrations : Utiliser Django pour configurer la base de données SQLite en exécutant : **python manage.py migrate**

9. Créer un superutilisateur : Générer un compte superutilisateur pour accéder à l'interface administrateur et aux fonctionnalités bibliothécaires avec : **python manage.py createsuperuser**

10. Suivre les invites pour entrer un nom d'utilisateur, un email, et un mot de passe.

11. Lancer le serveur Django : Démarrer le serveur local pour accéder à l'application en exécutant : **python manage.py runserver**

12. Accéder à l’application :
    - Ouvrir un navigateur web et se connecter à http://127.0.0.1:8000/.
    - Pour les bibliothécaires, se connecter via http://127.0.0.1:8000/login/ avec les identifiants du superutilisateur, puis accéder aux fonctionnalités à http://127.0.0.1:8000/bibliothecaire/.
    - Pour les membres, accéder directement à http://127.0.0.1:8000/membre/ pour consulter la liste publique des médias.

Ces instructions permettent d’exécuter le programme sur n’importe quelle machine équipée de Python et Git, sans nécessiter d’outils ou logiciels supplémentaires payants.