Auteur : Eléa BITSCH
Date : Février 2025

RAPPORT DE PROJET - APPLICATION MÉDIATHÈQUE
Gestion d'une médiathèque en Django

TABLE DES MATIÈRES

1. INTRODUCTION

Ce rapport présente le travail réalisé dans le cadre du projet de développement d'une application de gestion de médiathèque. Le point de départ était un code Python basique fourni en début de projet, qu'il fallait analyser, corriger et transformer en une application web complète utilisant le framework Django.

L'objectif principal était de démontrer ma compréhension des concepts de Programmation Orientée Objet (POO), notamment l'héritage, tout en développant une application fonctionnelle répondant aux besoins d'une médiathèque.

2. ANALYSE DU CODE FOURNI ET IDENTIFICATION DES PROBLÈMES

Après analyse, j'ai identifié plusieurs problèmes majeurs :

PROBLÈME 1 : Absence d'héritage (principe POO non respecté) :

Les classes livre, dvd et cd ont des attributs communs (name, dateEmprunt, disponible, emprunteur) mais sont définies séparément sans classe parente.

- Constat : Duplication de code, difficile à maintenir
- Impact : Si on veut ajouter un attribut commun, il faut modifier 3 classes

PROBLÈME 2 : Attributs de classe au lieu d'attributs d'instance :

Les attributs sont définis au niveau de la classe avec des valeurs par défaut :

```
class livre():
    name = ""       # Attribut de CLASSE, pas d'instance !
    auteur = ""
```

- Constat : Tous les objets partageraient les mêmes valeurs
- Impact : Impossible de créer plusieurs livres différents correctement

PROBLÈME 3 : Absence de constructeur (__init__) :

Aucune classe ne possède de méthode __init__ pour initialiser les objets.

- Constat : Pas de moyen propre d'initialiser un objet avec ses données
- Impact : Code peu robuste et non conforme aux bonnes pratiques

PROBLÈME 4 : Types de données incorrects :

Tous les attributs sont initialisés avec des chaînes vides "" :

```
disponible = ""     # Devrait être un booléen (True/False)
bloque = ""         # Devrait être un booléen
```

- Constat : Pas de typage approprié
- Impact : Risques d'erreurs et logique métier impossible à implémenter

PROBLÈME 5 : Nomenclature incohérente :

- Mélange français/anglais : "name" vs "auteur", "disponible" vs "emprunteur"
- Conventions non respectées : "jeuDePlateau" (camelCase) vs "livre" (minuscule)
- Classe "Emprunteur" avec majuscule vs "livre" sans majuscule

PROBLÈME 6 : Fonctions vides sans implémentation :

Les fonctions menu() et menuBibliotheque() n'affichent que du texte.

- Constat : Aucune logique métier implémentée
- Impact : Code non fonctionnel

PROBLÈME 7 : Pas de gestion des relations :

La relation entre un emprunteur et un média est gérée par une simple chaîne :

```
emprunteur = ""
```

- Constat : Pas de lien réel entre les objets

- Impact : Impossible de gérer les emprunts correctement

PROBLÈME 8 : Pas de persistance des données :

Le code ne prévoit aucun mécanisme de sauvegarde des données.
- Constat : Les données sont perdues à chaque exécution
- Impact : Application inutilisable en conditions réelles

3. DÉMARCHE DE TRAVAIL ET MÉTHODOLOGIE

J'ai suivi une approche structurée en plusieurs phases :

Phase 1 : Analyse et compréhension
|
v
Phase 2 : Conception de l'architecture
|
v
Phase 3 : Mise en place du projet Django
|
v
Phase 4 : Implémentation des modèles (POO)
|
v
Phase 5 : Développement des fonctionnalités
|
v
Phase 6 : Tests et validation
|
v
Phase 7 : Documentation

4. CORRECTIONS ET REFONTE DU CODE

4.1 Mise en place de l'héritage

CODE INITIAL (sans héritage) :

```
class livre():
    name = ""
    auteur = ""
    dateEmprunt = ""
    disponible = ""

class dvd():
```

```
    name = ""
    realisateur = ""
    dateEmprunt = ""
    disponible = ""
```

CODE CORRIGÉ (avec héritage) :

```
class Media(models.Model):
    """Classe mère abstraite pour tous les médias"""
    titre = models.CharField(max_length=200)
    auteur = models.CharField(max_length=200)
    nombre_exemplaires = models.PositiveIntegerField(default=1)
    date_ajout = models.DateField(auto_now_add=True)
    disponible = models.BooleanField(default=True)

    class Meta:
        abstract = True  # Classe abstraite, pas de table en BDD

    def est_disponible(self):
        return self.exemplaires_disponibles() > 0

class Livre(Media):
    """Hérite de Media - attributs communs automatiquement inclus"""
    pass

class DVD(Media):
    """Hérite de Media avec attribut spécifique"""
    duree = models.PositiveIntegerField()
```

AVANTAGES DE CETTE CORRECTION :
- Code factorisé : les attributs communs sont définis une seule fois
- Maintenance facilitée : une modification dans Media impacte tous les enfants
- Respect du principe DRY (Don't Repeat Yourself)
- Méthodes partagées : est_disponible() est accessible pour tous les médias

4.2 Correction des attributs de classe vers attributs d'instance

CODE INITIAL :

```
class Emprunteur():
    name = ""          # Attribut de CLASSE
    bloque = ""
```

CODE CORRIGÉ :

```
class Membre(models.Model):
    nom = models.CharField(max_length=100)      # Attribut d'INSTANCE
    prenom = models.CharField(max_length=100)
```

```
    email = models.EmailField(unique=True)
    actif = models.BooleanField(default=True)

    def __str__(self):
        return f"{self.prenom} {self.nom}"
```

EXPLICATION :
Avec Django ORM, chaque champ (models.CharField, etc.) devient automatiquement un attribut d'instance. Chaque objet Membre aura ses propres valeurs.

4.3 Ajout de la logique métier (méthodes)

Le code initial n'avait aucune méthode pour gérer la logique métier.
J'ai ajouté des méthodes dans les classes pour encapsuler le comportement :

```
class Membre(models.Model):
    # ... attributs ...

    def nombre_emprunts_en_cours(self):
        """Compte les emprunts non retournés"""
        return self.emprunt_set.filter(
            date_retour_effective__isnull=True
        ).count()

    def a_emprunt_en_retard(self):
        """Vérifie si le membre a un retard"""
        return self.emprunt_set.filter(
            date_retour_effective__isnull=True,
            date_retour_prevue__lt=timezone.now().date()
        ).exists()

    def peut_emprunter(self):
        """Vérifie les conditions d'emprunt"""
        if not self.actif:
            return False
        if self.nombre_emprunts_en_cours() >= 3:
            return False
        if self.a_emprunt_en_retard():
            return False
        return True
```

AVANTAGES :
- La logique est encapsulée dans la classe (principe d'encapsulation)
- Code réutilisable partout dans l'application
- Tests unitaires possibles sur chaque méthode

4.4 Gestion des relations entre objets

CODE INITIAL :

```
class livre():
    emprunteur = ""     # Simple chaîne, pas de vraie relation
```

CODE CORRIGÉ :

```
class Emprunt(models.Model):
    membre = models.ForeignKey(Membre, on_delete=models.CASCADE)
    livre = models.ForeignKey(Livre, null=True, blank=True, ...)
    dvd = models.ForeignKey(DVD, null=True, blank=True, ...)
    cd = models.ForeignKey(CD, null=True, blank=True, ...)
    date_emprunt = models.DateField(auto_now_add=True)
    date_retour_prevue = models.DateField()
    date_retour_effective = models.DateField(null=True, blank=True)
```

EXPLICATION :
J'ai créé une classe Emprunt qui fait le lien entre un Membre et un média.
Cela permet de :
- Garder l'historique des emprunts
- Gérer les dates de retour
- Savoir qui a emprunté quoi et quand

5. CHOIX TECHNIQUES

CHOIX 1 : Classe Media abstraite
- Pourquoi : Éviter la création d'une table Media en base de données tout en partageant le code commun.
- Alternative rejetée : Héritage concret (aurait créé une table inutile)

CHOIX 2 : JeuPlateau sans héritage de Media
- Pourquoi : Les jeux ne sont pas empruntables, donc pas besoin des attributs et méthodes liés aux emprunts.
- Bénéfice : Modèle plus simple et plus logique

CHOIX 3 : Classe Emprunt séparée (plutôt qu'attribut dans Média)
- Pourquoi : Permet de garder l'historique, gérer plusieurs emprunts simultanés pour un média multi-exemplaires.
- Bénéfice : Flexibilité et traçabilité

CHOIX 4 : Disponibilité calculée dynamiquement
- Pourquoi : Éviter les incohérences entre le champ "disponible" et la réalité des emprunts.
- Méthode : exemplaires_disponibles() = nombre_exemplaires - emprunts_en_cours

6. FONCTIONNALITÉS IMPLÉMENTÉES

6.1 Accès différenciés selon le cahier des charges

ACCES VISITEUR (membre) - Sans authentification :
[x] Consultation de la liste des médias
[x] Visualisation de la disponibilité (X/Y exemplaires)

ACCES BIBLIOTHECAIRE - Avec authentification :
[x] Gestion des membres (CRUD complet)
[x] Gestion des médias (CRUD complet)
[x] Création et suivi des emprunts
[x] Enregistrement des retours

6.2 Règles métier implémentées

RÈGLE 1 : Maximum 3 emprunts par membre
- Implémentation : Méthode peut_emprunter() vérifie nombre_emprunts_en_cours()
- Test associé : test_max_trois_emprunts

RÈGLE 2 : Durée d'emprunt de 7 jours
- Implémentation : date_retour_prevue = date_emprunt + 7 jours (automatique)
- Test associé : test_date_retour_prevue_auto

RÈGLE 3 : Blocage si emprunt en retard
- Implémentation : Méthode a_emprunt_en_retard() + peut_emprunter()
- Test associé : test_membre_avec_retard_bloque

RÈGLE 4 : Jeux de plateau non empruntables
- Implémentation : Pas de ForeignKey vers JeuPlateau dans Emprunt
- Vérification : Pas de bouton "Emprunter" sur les jeux

6.3 Fonctionnalités additionnelles

- Gestion multi-exemplaires : Un même livre peut avoir plusieurs copies
- Emprunt direct depuis la liste : Bouton "Emprunter" sur chaque média
- Historique des emprunts : Emprunts en cours ET terminés visibles
- Logging : Traçabilité des actions (connexions, emprunts, modifications)

7. SÉCURITÉ ET BASE DE DONNÉES

7.1 Configuration sécurisée de la base de données

Pour répondre au critère "Connexion à base de données sécurisée par mot de passe",
j'ai mis en place une configuration utilisant des variables d'environnement :

```
# Dans settings.py
if os.environ.get('DB_NAME'):
    DATABASES = {
        'default': {
            'ENGINE': os.environ.get('DB_ENGINE', 'django.db.backends.postgresql'),
            'NAME': os.environ.get('DB_NAME'),
            'USER': os.environ.get('DB_USER'),
            'PASSWORD': os.environ.get('DB_PASSWORD'),
            'HOST': os.environ.get('DB_HOST', 'localhost'),
            'PORT': os.environ.get('DB_PORT', '5432'),
        }
    }
```

AVANTAGES DE CETTE APPROCHE :

- Les mots de passe ne sont jamais stockés dans le code source
- Configuration différente possible entre développement et production
- Respect des bonnes pratiques de sécurité (12-factor app)

7.2 Variables d'environnement

Un fichier .env.example est fourni pour documenter les variables nécessaires :

```
DB_ENGINE=django.db.backends.postgresql
DB_NAME=mediatheque_db
DB_USER=mediatheque_user
DB_PASSWORD=votre_mot_de_passe_securise
DB_HOST=localhost
DB_PORT=5432
```

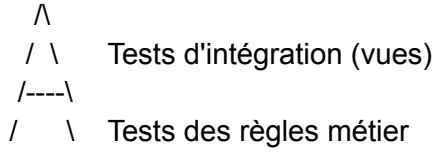
7.3 Autres mesures de sécurité

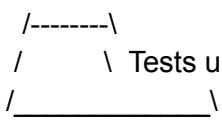
- Protection CSRF activée sur tous les formulaires
- Authentification requise pour les fonctionnalités bibliothécaire
- Décorateurs @login_required et @user_passes_test
- SECRET_KEY externalisable via variable d'environnement

8. STRATÉGIE DE TESTS

8.1 Approche de test adoptée

J'ai adopté une approche de tests en pyramide :





8.2 Tests unitaires des modèles

Objectif : Vérifier que chaque modèle fonctionne correctement isolément.

Exemples de tests :

```
    def test_creation_livre(self):
        """Vérifie qu'un livre se crée correctement"""
        livre = Livre.objects.create(titre="Test", nombre_exemplaires=2)
        self.assertEqual(livre.titre, "Test")
        self.assertEqual(livre.nombre_exemplaires, 2)

    def test_exemplaires_disponibles_avec_emprunt(self):
        """Vérifie le calcul de disponibilité"""
        Emprunt.objects.create(membre=self.membre, livre=self.livre)
        self.assertEqual(self.livre.exemplaires_disponibles(), 1)
```

8.3 Tests des règles métier

Objectif : Vérifier que les contraintes du cahier des charges sont respectées.

```
    def test_max_trois_emprunts(self):
        """Vérifie qu'on ne peut pas dépasser 3 emprunts"""
        Emprunt.objects.create(membre=self.membre, livre=self.livre1)
        Emprunt.objects.create(membre=self.membre, livre=self.livre2)
        Emprunt.objects.create(membre=self.membre, livre=self.livre3)
        self.assertFalse(self.membre.peut_emprunter())

    def test_membre_avec_retard_bloque(self):
        """Vérifie le blocage si retard"""
        emprunt = Emprunt.objects.create(membre=self.membre, livre=self.livre)
        emprunt.date_retour_prevue = timezone.now().date() - timedelta(days=1)
        emprunt.save()
        self.assertFalse(self.membre.peut_emprunter()
```

8.4 Tests des vues

Objectif : Vérifier que les pages fonctionnent et les accès sont contrôlés.

```
    def test_liste_medias_accessible(self):
        """La liste des médias doit être accessible sans connexion"""
        response = self.client.get(reverse('liste_medias'))
        self.assertEqual(response.status_code, 200)

    def test_liste_membres_non_staff(self):
```

```
        """La liste des membres doit être interdite aux non-staff"""
        self.client.login(username='membre', password='test')
        response = self.client.get(reverse('liste_membres'))
        self.assertNotEqual(response.status_code, 200)
```

8.5 Couverture des tests

Total : 45 tests répartis en 10 classes

Classe de test	Nombre de tests
LivreModelTest	6
DVDModelTest	2
CDModelTest	2
JeuPlateauModelTest	2
MembreModelTest	5
EmpruntModelTest	4
ReglesMetierTest	5
VuesPubliquesTest	2
AuthentificationTest	3
VuesBibliothecaireTest	8
AccesNonAutoriseTest	3
GestionEmpruntTest	3
TOTAL	45

9. INSTRUCTIONS D'INSTALLATION ET D'EXÉCUTION

9.1 Prérequis

- Python 3.10 ou supérieur
- pip (gestionnaire de paquets Python)
- Git

9.2 Installation pas à pas

Étape 1 : Récupérer le code source

```
git clone https://github.com/EleaDSB/CEF-mediatheque.git
cd CEF-mediatheque
```

Étape 2 : Créer un environnement virtuel (recommandé)

```
python3 -m venv venv
source venv/bin/activate        # macOS/Linux
# OU
```

```
venv\Scripts\activate          # Windows
```

Étape 3 : Installer Django

```
pip install django
```

Étape 4 : Créer la base de données

```
python3 manage.py migrate
```

Étape 5 : Charger les données de démonstration

```
python3 manage.py loaddata initial_data
```

Étape 6 : Créer un compte bibliothécaire

```
python3 manage.py createsuperuser
# Suivre les instructions (nom, email, mot de passe)
```

Étape 7 : Lancer le serveur

```
python3 manage.py runserver
```

Étape 8 : Accéder à l'application

Ouvrir http://127.0.0.1:8000/ dans un navigateur

9.3 Exécution des tests

Pour vérifier que tout fonctionne :

```
python3 manage.py test mediatheque
```

Résultat attendu : "Ran 45 tests in X.XXXs - OK"

9.4 Contenu des données de démonstration

Les fixtures (initial_data.json) contiennent :
- 4 membres (dont 1 inactif pour tester le blocage)
- 5 livres avec différents nombres d'exemplaires
- 4 DVDs
- 4 CDs
- 4 jeux de plateau
- 4 emprunts (1 retourné, 3 en cours)

10. CONCLUSION ET APPRENTISSAGES

10.1 Compétences développées

Ce projet m'a permis de développer et consolider plusieurs compétences :

PROGRAMMATION ORIENTÉE OBJET :
- Compréhension de l'héritage et son utilité (factorisation du code)
- Différence entre attributs de classe et d'instance
- Encapsulation de la logique métier dans les méthodes

- Classes abstraites et leur intérêt

FRAMEWORK DJANGO :
- Architecture MVT (Modèle-Vue-Template)
- ORM Django pour la gestion de base de données
- Système d'authentification et de permissions
- Écriture de tests unitaires avec TestCase

MÉTHODOLOGIE :
- Analyse d'un code existant et identification des problèmes
- Approche itérative du développement
- Utilisation de Git avec stratégie de branches
- Documentation du travail réalisé

10.2 Difficultés rencontrées et solutions

DIFFICULTÉ 1 : Comprendre pourquoi le code initial était problématique
- Solution : Recherche sur les bonnes pratiques POO, comparaison avec des exemples corrects

DIFFICULTÉ 2 : Gestion dynamique de la disponibilité
- Solution : Création de méthodes calculant en temps réel plutôt que stocker une valeur statique

DIFFICULTÉ 3 : Système d'authentification personnalisé
Solution : Création de vues de login dédiées au lieu d'utiliser l'interface admin Django

10.3 Améliorations possibles

Pour une version future de l'application :
- Système de recherche et filtrage des médias
- Notifications par email pour les retards
- Système de réservation pour les médias indisponibles
- API REST pour une éventuelle application mobile