# CREATION D'UN LOGICIEL DE GESTION DE MÉDIATHÈQUE

Dans ce projet, j'ai développé une application web de gestion de médiathèque en utilisant Django, un framework de Python basé sur la programmation orientée objet (POO). L'application permet de gérer les emprunts de médias (livres, CDs, DVDs, jeux de plateau) par les membres, en assurant une gestion efficace des emprunteurs et des disponibilités des médias. J'ai utilisé les fonctionnalités de Django telles que les vues, les formulaires et les modèles, ainsi que des tests unitaires pour valider le bon fonctionnement de l'application. Ce rapport détaille les différentes étapes de développement, les outils utilisés, et les tests effectués.

## 1. L'ETUDE ET LES CORRECTIFS DU CODE EXISTANT

- Utilisation d'un constructeur (__init__) pour initialiser les attributs des classes.
- Ajout de méthodes pour les comportements spécifiques des classes.
- Utilisez l'héritage pour les classes qui partagent des attributs communs (comme Livre, DVD, CD, et JeuDePlateau)
- Utilisation de modèles pour définir la structure de la base de données, ils sont des sous classes de **models.Model**

## 2. CODE CORRIGE ET ADAPTE

```
class Emprunteur(models.Model):
    nom = models.CharField(max_length=100)
    bloque = models.BooleanField(default=False)
```

```
    def __str__(self):
        return f"{self.nom} - {'Bloqué' if self.bloque else
'Actif'}"
```

```
    def peut_emprunter(self):
        if self.bloque:
            return False
        # Vérifie s'il a des emprunts en retard
        emprunts_en_retard =
Emprunt.objects.filter(emprunteur=self, date_retour__isnull=True,
date_emprunt__lt=timezone.now() - timedelta(days=7))
        if emprunts_en_retard.exists():
            return False
```

```
        return True

    def emprunts_actuels(self):
        return Emprunt.objects.filter(emprunteur=self,
date_retour__isnull=True).count()

class Media(models.Model):
    quantite_disponible = models.PositiveIntegerField(default=1)
# Quantité disponible pour emprunt
    nom = models.CharField(max_length=100)
    disponible = models.BooleanField(default=True)
    date_emprunt = models.DateTimeField(null=True, blank=True)
    date_retour = models.DateTimeField(null=True, blank=True)
    emprunteur = models.ForeignKey(Emprunteur, null=True,
blank=True, on_delete=models.SET_NULL)

    def __str__(self):
        return f"{self.nom} - {'Disponible' if self.disponible
else 'Non disponible'}"


    def est_en_retard(self):
        if self.date_retour:
            return self.date_retour < timezone.now()
        return self.date_emprunt and timezone.now() >
self.date_emprunt + timedelta(days=7)

    def est_disponible_pour_emprunt(self):
        return (self.quantite_disponible > 0 and self.disponible)
and not isinstance(self, JeuDePlateau)
    



    def save(self, *args, **kwargs):
        if self.date_retour:
            self.disponible = True
        else:
            self.disponible = False
        super().save(*args, **kwargs)

class Livre(Media):
    auteur = models.CharField(max_length=100)

    def __str__(self):
        return f"Livre: {self.nom}, Auteur: {self.auteur},
{'Disponible' if self.disponible else 'Non disponible'}"

class DVD(Media):
    realisateur = models.CharField(max_length=100)
```

```
    def __str__(self):
        return f"DVD: {self.nom}, Réalisateur: {self.realisateur},
{'Disponible' if self.disponible else 'Non disponible'}"

class CD(Media):
    artiste = models.CharField(max_length=100)

    def __str__(self):
        return f"CD: {self.nom}, Artiste: {self.artiste},
{'Disponible' if self.disponible else 'Non disponible'}"

class JeuDePlateau(Media):
    createur = models.CharField(max_length=100)

    def __str__(self):
        return f"Jeu de Plateau: {self.nom}, Créateur:
{self.createur}, {'Disponible' if self.disponible else 'Non
disponible'}"

class Emprunt(models.Model):
    media = models.ForeignKey(Media, on_delete=models.CASCADE)
    emprunteur = models.ForeignKey(Emprunteur,
on_delete=models.CASCADE)
    date_emprunt = models.DateTimeField(auto_now_add=True)
    date_retour = models.DateTimeField(null=True, blank=True)

    def est_en_retard(self):
        if self.date_retour:
            return self.date_retour < timezone.now()
        return timezone.now() > self.date_emprunt +
timedelta(days=7)

    def date_retour_estimee(self):
        """Calcul de la date de retour estimée, 7 jours après
l'emprunt"""
        if self.date_emprunt:
            return self.date_emprunt + timedelta(days=7)
        return None
```

## 3. EXPLICATIONS DU NOUVEAU CODE

- **Classes de Modèle Django**

Chaque classe représente un modèle Django et hérite de **models.Model**.
Les attributs des classes (champs de modèle) sont définis à l'aide de champs Django (CharField, DateTimeField, BooleanField, ForeignKey).

• **Heritage**

Les classes **Livre**, **DVD**, **CD** héritent de la classe **Media**, ce qui permet de partager les attributs et les méthodes de base tout en ajoutant des attributs spécifiques à chaque type de média

• **Methodes**

**def méthode (self)** pour définir des méthodes pour des comportements spécifiques des classes.

## 4. LES FONCTIONS

### A - Créer membre emprunter

```
def ajout_membre(request):
    if request.method == 'POST':
        form = EmprunteurForm(request.POST)
        if form.is_valid():
            form.save()
            logger.info(f"Ajout d'un nouveau membre: 
{form.cleaned_data['nom']}")
            return redirect('membres')
        else:
            logger.warning("Formulaire d'ajout de membre 
invalide")
    else:
        form = EmprunteurForm()
        logger.info("Affichage du formulaire d'ajout de membre")
    return render(request, 'bibliothecaire/ajout_membre.html', 
{'form': form})
```

La fonction ajout_membre en Django gère la création d'un nouveau membre via un formulaire par les requêtes POST et GET avec les deux condition if else. Lorsque la méthode est POST, la fonction traite le formulaire soumis : elle crée une instance de EmprunteurForm défini dans le fichier forms.py, avec les données reçues, valide ces données et sauvegarde le nouveau membre dans la base de données, et redirige l'utilisateur vers la page listant les membres. Si la méthode est GET, un formulaire vide est affiché à l'utilisateur pour qu'il puisse ajouter un nouveau membre.

B - Afficher la liste des membres

```
def membres(request):
    logger.info("Affichage de la liste des membres")
    membres = Emprunteur.objects.all()
    return render(request, 'bibliothecaire/membres.html',
{'membres': membres})
```

La fonction membres est une vue Django qui récupère tous les objets de la classe Emprunteur de la base de données définie dans le fichier models.py, en utilisant Emprunteur.objects.all(), puis passe la liste des membres à un template nommé membres.html via le chemin bibliothecaire/membres.html pour être affichée.

C -Afficher la liste des médias.

```
def medias(request):
    livres = Livre.objects.all()
    cds = CD.objects.all()
    dvds = DVD.objects.all()
    jeux = JeuDePlateau.objects.all()

    context = {
        'livres': livres,
        'cds': cds,
        'dvds': dvds,
        'jeux': jeux
    }
    return render(request, 'bibliothecaire/medias.html', context)
```

La fonction medias récupère tous les objets des modèles Livre, CD, DVD, et JeuDePlateau depuis la base de données. Ils stockés dans un dictionnaire de contexte, sous les clés livres, cds, dvds, et jeux.
La fonction utilise la vue pour rendre le template bibliothecaire/medias.html, où la liste de chaque type de média est affichée.

D - Créer un emprunt pour un média disponible.

```
def ajouter_emprunt(request):
    if request.method == 'POST':
        media_id = request.POST.get('media')
        emprunteur_id = request.POST.get('emprunteur')

        media = get_object_or_404(Media, id=media_id)
        emprunteur = get_object_or_404(Emprunteur,
id=emprunteur_id)

        if not emprunteur.peut_emprunter():
```

```
            return render(request, 'bibliothecaire/
error_media_emprunt.html', {'message': "L'emprunteur ne peut pas
emprunter en raison d'emprunts en retard ou de son statut
bloqué.", 'error_type': 'emprunt'})

        if emprunteur.emprunts_actuels() >= 3:
            return render(request, 'bibliothecaire/
error_media_emprunt.html', {'message': "Un emprunteur ne peut pas
avoir plus de 3 emprunts en cours.", 'error_type': 'emprunt'})

        if not media.est_disponible_pour_emprunt():
            return render(request, 'bibliothecaire/
error_media_emprunt.html', {'message': "Le média n'est pas
disponible ou il s'agit d'un jeu de plateau.", 'error_type':
'media'})

        Emprunt.objects.create(media=media, emprunteur=emprunteur)

        media.quantite_disponible -= 1
        media.save()

        return redirect('emprunts')

    membres = Emprunteur.objects.all()
    medias =
Media.objects.exclude(id__in=JeuDePlateau.objects.values_list('id'
, flat=True))
    return render(request, 'bibliothecaire/ajouter_emprunt.html',
{'membres': membres, 'medias': medias})
```

La fonction ajouter_emprunt gère la création d'un emprunt via une requête POST en vérifiant les conditions d'éligibilité de l'emprunteur et la disponibilité du média. En cas d'erreur, elle affiche un message via un template spécifique. Si les conditions sont remplies, elle crée l'emprunt, ajuste la quantité du média et redirige vers la liste des emprunts. Pour une requête GET, elle affiche un formulaire pour sélectionner un emprunteur et un média.

## E - Ajouter un media

```
def ajouter_media(request):
    if request.method == 'POST':
        type_media = request.POST.get('type_media')

        if type_media == 'livre':
            form = LivreForm(request.POST)
        elif type_media == 'dvd':
            form = DVDForm(request.POST)
        elif type_media == 'cd':
            form = CDForm(request.POST)
```

```
        elif type_media == 'jeu_de_plateau':
            form = JeuDePlateauForm(request.POST)
        else:
            form = None

        if form and form.is_valid():
            form.save()
            return redirect('medias')
    else:
        form = LivreForm()  # Affiche le formulaire par défaut, peut-être offrir un choix de type media ici.

    return render(request, 'bibliothecaire/ajouter_media.html', {'form': form})
```

La fonction ajouter_media traite la création de nouveaux médias en fonction du type sélectionné dans le fichier forms.py via une requête POST. Selon le type de média soumis, elle utilise le formulaire approprié pour valider et enregistrer les données. En cas de succès, l'utilisateur est redirigé vers la liste des médias. Pour une requête GET, un formulaire par défaut est affiché.

## 5. STRATEGIE DE TEST

Voici le fichier test.py qui contient les test des principaux fonctionnalités

```
from django.test import TestCase
from django.urls import reverse
from .models import Emprunteur, Media, Emprunt

class MenuViewTests(TestCase):
    def test_menu_view(self):
        response = self.client.get(reverse('bibliothecaire_menu'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Menu Bibliothécaire")

class MembreViewTests(TestCase):
    def test_liste_membres_view(self):
        Emprunteur.objects.create(nom="Test Membre")
        response = self.client.get(reverse('membres'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Test Membre")

    def test_ajout_membre_view(self):
        response = self.client.post(reverse('ajout_membre'), {'nom': 'Nouveau Membre'})
        self.assertEqual(response.status_code, 302)  # Redirection après succès
```

```
        self.assertTrue(Emprunteur.objects.filter(nom='Nouveau
Membre').exists())

class MediaViewTests(TestCase):
    def test_liste_medias_view(self):
        Media.objects.create(nom="Test Media", disponible=True)
        response = self.client.get(reverse('medias'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Test Media")

    def test_ajout_media_view(self):
        response = self.client.post(reverse('ajouter_media'),
{'type_media': 'livre', 'nom': 'Nouveau Livre'})
        self.assertEqual(response.status_code, 200)
        self.assertTrue(Media.objects.filter(nom='Nouveau
Livre').exists())

class EmpruntViewTests(TestCase):
    def test_liste_emprunts_view(self):
        emprunteur = Emprunteur.objects.create(nom="Test Membre")
        media = Media.objects.create(nom="Test Media",
disponible=True)
        Emprunt.objects.create(emprunteur=emprunteur, media=media)
        response = self.client.get(reverse('emprunts'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, "Test Membre")
        self.assertContains(response, "Test Media")

    def test_ajout_emprunt_view(self):
        emprunteur = Emprunteur.objects.create(nom="Test Membre")
        media = Media.objects.create(nom="Test Media",
disponible=True)
        response = self.client.post(reverse('ajouter_emprunt'),
{'media': media.id, 'emprunteur': emprunteur.id})
        self.assertEqual(response.status_code, 302)

self.assertTrue(Emprunt.objects.filter(emprunteur=emprunteur,
media=media).exists())
```

J'ai utilisé la classe TestCase de Django pour vérifier les fonctionnalités principales de l'application.

- **Import de TestCase :** pour faciliter les tests de vues et des modèles en simulant des requêtes HTTP

- **Mise en place des tests :**

- MenuViewTests

- MembreViewTests

- MediaViewTests

- EmpruntViewTests

Pour chaque vue, je  teste d'abord si la page se charge bien (status code 200), et de voir si le contenu attendu est présent.  Les tests de création (ajout) vérifient que les objets sont bien créés dans la base de données après l'envoi d'un formulaire, et qu'il y a une redirection après un succès (status code 302).

Cela permet de s'assurer que les principales fonctionnalités de l'application (affichage et création des membres, médias et emprunts) fonctionnent comme prévu.

## 6. MISE EN PLACE DE LA BDD AVEC DES DONNEES DE TEST

Dans mon projet, j'ai utilisé des **fixtures** pour charger d'une façon automatisée des données de test dans la base de données.

Voici un résumé des fichiers de fixtures utilisés :

- **livres.json** : Contient des informations sur les livres disponibles, comme le titre, l'auteur et la disponibilité.
- **cds.json** : Contient les données des CD, avec le nom de l'artiste, le titre de l'album et leur disponibilité.
- **dvds.json** : Regroupe des détails sur les DVD, incluant le titre, le réalisateur et leur disponibilité.
- **jeux.json** : Décrit les jeux de plateau, avec les informations sur le créateur, le nom du jeu et leur disponibilité.
- **membres.json** : Décrit les membres inscrits, avec leur nom et leur statut (actif ou bloqué).

## 7. INSTRUCTIONS POUR EXECUTER LE PROGRAMME

### 1. Préparer l'Environnement

1. Assurez-vous que Python 3.x est installé.
2. Vous pouvez vérifier cela en exécutant :

   ```
   python --version
   ```

3. Utilisez pip pour installer Django et les autres dépendances. Il est recommandé d'utiliser un environnement virtuel pour isoler les dépendances :
   bash

   ```
   python -m vend env
   ```

4. `source env/bin/activate`
5. `# Sur Windows, utilisez `env\Scripts\activate``
6. `pip install django`

7. Clonez ou téléchargez le projet depuis le dépôt source (par exemple, GitHub). Utilisez git pour cloner le dépôt :

### 2. Configurer le Projet

1. Assurez-vous que toutes les dépendances du projet sont installées. Les dépendances sont listées dans le fichier `requirements.txt`. Installez-les avec :
   `pip install -r requirements.txt`
2. Assurez-vous que le fichier de configuration de la base de données (`settings.py`) est correctement configuré. Ouvrez le et au niveau de INSTALLED_APPS , rajoutez les 3 applications comme suit :

```
INSTALLED_APPS = [

    'mediatheque',
    'bibliothecaire',
    'membre',
]
```

3. Créez les tables nécessaires dans la base de données en appliquant les migrations les models du fichier models.py

4. `python manage.py makemigrations`

5. `python manage.py migrate`

6. **Charger les Données de Test** :

7. `python manage.py loaddata cds.json`
8. `python manage.py loaddata livres.json`

9. `python manage.py loaddata dvds.json`
10. `python manage.py loaddata membres.json`
11. `python manage.py loaddata jeux.json`

## 3. Exécuter le Serveur

1. Lancez le serveur Django pour accéder à l'application via un navigateur web :

   ```
   python manage.py runserver
   ```

2. Ouvrez un navigateur web et allez à l'adresse fournie pour voir l'application en fonctionnement.

En conclusion, ce projet m'a permis de développer un logiciel de gestion de mediathèque avec Django tout en appliquant les principes de la programmation orientée objet (POO). J'ai implémenté la gestion des emprunts, des membres et des médias de manière structurée. Les fixtures ont facilité l'insertion automatique des données de test, simplifiant ainsi le développement et les tests. Les tests unitaires ont assuré la validité de chaque fonctionnalité, garantissant la fiabilité du système. Ce projet m'a également permis d'approfondir la gestion des requêtes HTTP et la validation des données, renforçant mes compétences en développement web orienté objet.