MÉDIATHÈQUE AVEC DJANGO

SOMMAIRE

LIEN DU PROJET

GitHub Repository
https://github.com/AlexD004/CEF-Mediatheque

Pré-requis

Il faut que python soit installé.

Création d'un environnement virtuel

Pour commencer, il faut créer un environnement virtuel afin que le projet puisse fonctionner. Pour cela, rendez-vous dans le dossier de votre projet, normalement vide, puis entrez les lignes de commande suivantes dans votre interface CLI.

```
py -m venv [nom de l'environnement]
```

Par exemple :

```
py -m venv CEF-Mediatheque
```

Cloner le repository

Ensuite, dans le dossier créé lors de la création de l'environnement virtuel, on clone le repository fourni en page 3 de ce document.

```
git clone https://github.com/AlexD004/CEF-Mediatheque
```

Vous avez maintenant tous les fichiers nécessaires pour le lancement de la médiathèque.

Activer l'environnement virtuel

Tout d'abord, positionnez-vous dans le dossier racine de l'environnement virtuel. À la suite du clonage du repository, vous devriez remonter d'un niveau dans la hiérarchie.

```
cd ../
```

Ensuite, positionnez-vous dans le dossier 'Scripts' et activez l'environnement virtuel.

```
cd Scripts
```

```
.\activate
```

Installer Django

Toujours en ligne de commande, remontez à la racine de l'environnement virtuel :

```
cd ../
```

Puis isntallez django avec pip :

```
pip install Django
```

Si pip n'est pas installé :

```
py -m ensurepip —upgrade
```

Se rendre dans l'application

Maintenant, il faut aller dans le dossier de l'application, c'est-à-dire celui que vous avez cloné depuis le repository.

```
cd CEF-Mediatheque
```

Enfin, lancez le serveur :

```
py manage.py runserver
```


La médiathèque proposée ici possède deux interfaces :

- Une interface '**par défaut**' pour les **utilisateurs sans accès**.
  Elle ne permet que de visualiser les médias et les jeux disponibles dans la médiathèque.

- Une interface '**bibliothécaire**' qui permet de visualiser les médias et les jeux mais aussi les membres inscrits.
  Elle permet également d'ajouter, modifier et supprimer médias, jeux et membres. Enfin, elle permet de lier des membres et des médias pour créer des emprunts et de supprimer ceux-ci pour signaler un retour de l'emprunt.

Pour vous connecter en tant que 'Bibliothécaire', cliquez sur 'connexion' en haut à gauche de la page d'accueil et entrez les identifiants suivants :

Nom d'utilisateur :

CEFMediatheque

Mot de passe :

CEFMediatheque49?

```python
def menu():
    print("menu")

if __name__ == '__main__':
    menu()


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

class dvd():
    name = ""
        realisateur = ""
        dateEmprunt = ""
        disponible = ""
    emprunteur = ""


class cd():
    name = ""
    artiste = ""
    dateEmprunt = ""
    disponible = ""
    emprunteur = ""

class jeuDePlateau :
    name = ""
    createur = ""

class Emprunteur():
    name = ""
        bloque = ""


def menuBibliotheque() :
    print("c'est le menu de l'application des bibliothéquaire")



def menuMembre():
    print("c'est le menu de l'application des membres")
    print("affiche tout")
```

Mauvaise indentation

Pour commencer, corrigeons les mauvaises indentations du code que nous pouvons constater dans les class ***dvd*** (realisateur, dateEmprunt, disponible) et ***Emprunteur*** (bloque).

Cohérence et capitales

Les noms de class doivent commencer par une capitale.

Respect du brief

Les class ***livre***, ***dvd*** et ***cd*** doivent être enfants d'une class mère ***Medias***. Ce n'est pas le cas ici. De plus, avec l'ajout de cette class mère, il sera possible de placer certaines données dans celle-ci afin d'éviter les répétitions (name, disponible, emprunteur). Les class enfants pourront ainsi hériter de ces paramètres, il n'est pas utile de les remettre à chaque fois.

Modèles incomplets

Afin de pouvoir enregistrer les données dans la base de données, il faut informer sur les types de données attendues afin de pouvoir créer les tables via la commande 'migrate'. Nous devrions avoir des déclarations ressemblant à ceci :

```
from django.db import models
name = models.CharField(max_length = 100, default = «Inconnu», unique = True)
```

Noms parfois flous

Essayer d'avoir des noms de données les plus clairs possibles afin de faciliter la relecture. Le terme 'bloque' peut paraître clair sur le moment mais manque un peu de contexte. Qu'est-ce qui est bloqué ? La capacité d'emprunt à priori, mais cela pourrait aussi être la possibilité de se connecter ?

Données inutiles

Avec les informations présentes, l'information sur la disponibilité des médias me semble inutile. Il est possible de considérer un média comme étant disponible ou non avec seulement la date d'emprunt. S'il y a une date d'emprunt, le média est emprunté, il n'est donc pas disponible. Au contraire, s'il n'y a pas de date, alors le média est disponible.

Les méthodes de visualisation

Disponibles pour tous (emprunteurs et bibliothécaires), elles permettent de visualiser les objets contenus dans la base de données.
Elles correspondent au mot clé 'READ'. Seule exception, les données de la class 'Membres' ne sont disponibles que par les utilisateurs connectés, c'est-à-dire les 'bibliothécaires'.

Il y a plusieurs contextes de visualisation dans ce projet :

Les listes 'classiques' :
Ici on affichera dans le template 'listMedias.html' tous les objets étant instances de la class Medias. Fonctionnement identique pour les jeux et les membres.

```
def listMedias(request):
  return render(request, «logisticMediatheque/lists/listMedias.html», {«items»: Medias.objects.all()})
```

Les listes 'par type' :
Petite particularité des médias, ils sont subdivisés en livres, dvds et cds. Il existe donc une fonctionnalité pour filtrer les médias par type.
On ajoute simplement un filtre aux objets de la class Medias pour ne sélectionner que ceux dont le 'mediaType'
en base de données correspondant à celui demandé dans la réquête, stocké dans la variable 'item_type'.

```
def listMediasByType(request, item_type):
  return render(request, «logisticMediatheque/lists/listMedias.html», {
    «items»: Medias.objects.all().filter( mediaType = item_type),
    «type»: item_type
  })
```

Les listes 'détaillées' :
Ici l'objectif est d'afficher toutes les informations disponibles pour un médias (ou un membre, ou un jeu...) précis.
On va donc récupérer l'id de l'item qui a été cliqué afin de retrouver l'objet correspondant en base de données et pouvoir afficher les autres informations dans le template.

```
def mediaDetail(request, item_type, item_id ):
  if item_type == 'livre':
    context = { «item»: get_object_or_404( Livres, pk = item_id )}
  elif item_type == 'cd':
    context = { «item»: get_object_or_404( CDs, pk = item_id )}
  elif item_type == 'dvd':
    context = { «item»: get_object_or_404( DVDs, pk = item_id )}
  return render(request, «logisticMediatheque/itemDetails/mediaDetails.html», context)
```

Les méthodes de suppression

**Disponibles seulement pour les bibliothécaires, elles permettent de supprimer les objets contenus dans la base de données.
Elles correspondent au mot clé 'DELETE'.**

Concrêtement, on récupère l'id unique de l'item cliqué afin de pouvoir le cibler précisément dans la base de données. On peut ainsi le supprimer, c'est-à-dire enlever sa ligne dans la table. L'objet n'existera plus.

Ensuite, on déclare une variable 'medias' (dans l'exemple, mais fonctionne pareillement pour jeux et membres) qui contient tous les objets de la class Medias pour s'en servir lors du rendu.

On utilise la méthode 'logger' afin d'enregistrer l'action dans les logs.

Et enfin, on redirige l'utilisateur sur la page (ici la page sur laquelle il était déjà mais permet une actualisation des informations affichées). C'est ici qu'on utilise la variable 'medias' pour afficher la liste de tous les médias (sauf celui qui vient d'être supprimé).

```
def removeMedia(request, item_id):
    media = Medias.objects.get( pk = item_id )
    media.delete()
    medias = Medias.objects.all()

    logger.info(«Suppression Livre par « + request.user.username + « | Titre du livre : « + media.title )

    return render(request, 'logisticMediatheque/lists/listMedias.html', {'items': medias})
```

Les méthodes d'ajout

Disponibles seulement pour les bibliothécaires, elles permettent d'ajouter de nouveaux objets dans la base de données.
Elles correspondent au mot clé 'CREATE'.

Concrêtement, avec l'ajout d'un membre (l'ajout de média étant particulier car prend en compte le type de média à ajouter [livre, cd, dvd] selon le bouton cliqué).

On déclare une variable 'action_type' qui permettra seulement d'afficher un texte au dessus du formulaire.
Le formulaire étant le même pour l'ajout ou la mise à jour, cette variable permet d'informer l'utilisateur sur l'action en cours.

On pose une condition :
si on accède à cette méthode à la suite d'une soumission de formulaire via la méthode 'POST'
alors on récupère les données de ce formulaire, et SI il est valide
alors on enregistre ces données dans la base de données
puis on déclare une variable avec les listes des items, ici les membres, pour l'affichage du rendu
comme vu précédemment
on enregistre l'action dans les logs
et on renvoit l'utilisateur sur la liste de ce qu'il vient de créer, ici les membres
sinon
on renvoit l'utilisateur sur le formulaire permettant l'ajout, qui lui-même renverra vers cette même méthode avec un formulaire soumis avec POST et donc validera la condition

```
def addMembre(request):
  action_type = 'Ajout'

  if request.method == 'POST':
    form = addMembreForm(request.POST)

    if form.is_valid():
      form.save()
      membres = Membres.objects.all()

      logger.info(«Ajout Membre par « + request.user.username + « | Nom du membre : « + request.POST.get('lastname') + « « + request.POST.get('firstname'))

      return render(request, 'logisticMediatheque/lists/listMembres.html',{'membres': membres})
  else:
    form = addMembreForm()
    return render(request, 'logisticMediatheque/forms/addMembreForm.html',{'form': form, 'actionType': action_type})
```

Les méthodes de mise à jour

**Disponibles seulement pour les bibliothécaires, elles permettent de mettre à jour, d'actualiser, les objets contenus dans la base de données.
Elles correspondent au mot clé 'UPDATE'.**

Concrêtement, avec la mise à jour d'un membre (la mise à jour de média étant particulière car prend en compte le type de média à mettre à jour [livre, cd, dvd] selon l'objet cliqué. Le principe reste le même mais certains mots affichés informent l'utilisateur. Il y a donc quelques conditions et variables en plus).

On déclare une variable 'action_type' qui permettra seulement d'afficher un texte au dessus du formulaire. Le formulaire étant le même pour l'ajout ou la mise à jour, cette variable permet d'informer l'utilisateur sur l'action en cours.

Le principe est sensiblement le même que pour l'ajout, à ceci prêt que l'on récupère d'abord l'id de l'item (membre, jeu, media) cliqué en premier lieu.

On pose ensuite une condition :
- **si on accède à cette méthode à la suite d'une soumission de formulaire via la méthode 'POST'**
  - **alors on récupère les données de ce formulaire et la ligne correspondant à l'item dans la base de donnée, et SI le formulaire est valide**
  - **alors on enregistre ces données dans la base de données, à la place de celles existantes dans CETTE ligne, on n'en crée pas une nouvelle**
  - **puis on déclare une variable avec les listes des items, ici les membres, pour l'affichage du rendu comme vu précédemment**
  - **on enregistre l'action dans les logs**
  - **et on renvoit l'utilisateur sur la liste de ce qu'il vient de créer, ici les membres**
- **sinon**
  - **on renvoit l'utilisateur sur le formulaire permettant l'ajout mais avec l'instance de class Membre correspondant à l'id récupéré. Ainsi le formulaire est pré-rempli avec les informations déjà existantes qu'il n'y a plus qu'à modifier. Le formulaire renverra vers la méthode ci-dessous avec un formulaire sousmis avec POST et donc validera la condition**

```
def updateMembre(request, item_id):
  action_type = 'Modification'
  membre = Membres.objects.get( pk = item_id )

  if request.method == 'POST':
    form = addMembreForm(request.POST, instance=membre)

    if form.is_valid():
      form.save()
      membres = Membres.objects.all()

      logger.info(«MaJ Membre par « + request.user.username + « | Nom du membre : « + request.POST.get('lastname') + « « + request.POST.get('firstname'))

      return render(request, 'logisticMediatheque/lists/listMembres.html',{'membres': membres})
  else:
    form = addMembreForm(instance=membre)
    return render(request, 'logisticMediatheque/forms/addMembreForm.html',{'form': form, 'actionType': action_type})
```

Les méthodes d'emprunt

```
membre = form.cleaned_data['membre']
media = form.cleaned_data['media']

membreLoaning = Membres.objects.get( pk = membre.id )
mediaLoaned = Medias.objects.get( pk = media.id )

mediaLoans = Medias.objects.all().filter(borrower = membre.id)

for media in mediaLoans:
        now = datetime.now().date()
        dateLoan = media.dateLoan
        timeLoan = now - dateLoan
        media.timeLoan = timeLoan.days

        if media.timeLoan > 7 :
                membre.canLoan = False
                membre.save()

if membre.canLoan == True:
        membreLoaning.numLoan += 1
        mediaLoaned.borrower = membre
        mediaLoaned.dateLoan = datetime.now()

        membreLoaning.save()
        mediaLoaned.save()
```

La première méthode pour les emprunts correspond à l'ajout d'un emprunt. Elle reprend la base d'une mise à jour de ligne dans la base de données comme vu précédemment dans les méthodes de mise à jour. On a donc une condition qui vérifie si un formulaire a été soumis ou si la vue doit rediriger l'utilisateur vers le formulaire correspondant. On ne détaillera pas ce principe ici, il fonctionne de la même façon. Cependant, attardons nous sur certaines spécificités de cette fonctionnalité...

Le formulaire d'ajout d'emprunt récupère deux id : celui d'un membre et celui d'un média.

Ensuite, on récupère la liste de tous les médias qui ont pour emprunteur (borrower) le membre correspondant afin de faire une vérification :
Pour chaque média contenu dans cette liste, dans cette sélection, on va vérifier le temps d'emprunt. Pour cela on récupère la date d'aujourd'hui, on récupère la date d'emprunt du média contenu en base de données, puis on fait une soustraction des deux afin d'avoir la durée entre les deux dates. Enfin, pour simplifier la vérification, on récupère seulement le nombre de jours de cette durée.

On pose ensuite une condition, si ce nombre de jours (durée entre la date d'emprunt et aujourd'hui) est supérieur à 7, alors l'emprunt a plus d'une semaine et donc le membre n'est plus autorisé à emprunter. On change donc la valeur de 'canLoan' à False pour signaler cette nouvelle condition, puis on enregistre cela en base de données.

Ensuite, on vérifie seulement si le membre est autorisé à emprunter, si oui on va ajouter '1' au nombre d'emprunts de ce membre (numLoan) afin de connaître combien de médias il a en sa possession. Puis on actualise les informations du média nouvellement emprunté en lui désignant le membre comme emprunteur, puis la date du jour comme date d'emprunt. On enregistre le tout en base de données via la méthode save().

Le reste n'est pas détaillé ici mais on log l'action puis on redirige l'utilisateur vers la page détaillée du membre.

Si le membre n'est pas autorisé à emprunter, on redirige vers le formulaire avec un message signalant cette erreur.

Note : on utilise 'numLoan' pour filtrer les membres sélectionnables dans le formulaire. Dans l'input select permettant de choisir le membre emprunteur, ne sont affichés que les membres ayant moins de 3 emprunts.

```
class addLoanForm(forms.Form):
  membre = forms.ModelChoiceField(queryset=Membres.objects.all().filter(numLoan__lt=3, canLoan=True), label = «Emprunteur»)
  media = forms.ModelChoiceField(queryset=Medias.objects.all().filter(dateLoan__isnull=True), label = «Média»)
```

Les méthodes d'emprunt

```
def removeLoan(request, item_id, membre_id):
    media = Medias.objects.get( pk = item_id )
    membre = Membres.objects.get( pk = membre_id )

    membre.numLoan -= 1
    membre.canLoan = True
    media.dateLoan = None
    media.timeLoan = None
    media.borrower = None

    membre.save()
    media.save()

    mediaLoans = Medias.objects.all().filter(borrower = membre_id)

    logger.info(«Suppression Emprunt « + request.user.username + « | Nom du membre : « + membre.lastname + « « + membre.firstname + « / Nom du médias : « + media.title )

    for media in mediaLoans:
        now = datetime.now().date()
        dateLoan = media.dateLoan
        timeLoan = now - dateLoan
        media.timeLoan = timeLoan.days
        media.save()
        if media.timeLoan > 7 :
            membre.canLoan = False
            membre.save()

    return render(
        request,
        «logisticMediatheque/itemDetails/membreDetails.html»,
        {
            «membre»: get_object_or_404( Membres, pk = membre_id ),
            'mediaLoans': mediaLoans
        })
```

Pour la suppression d'un emprunt, on récupère également les id du membre et du média concernés afin de les actualiser. Pour le média on 'nettoie' les informations en remettant tout à 'zéro' (la date d'emprunt, le temps d'emprunt, l'emprunteur). Pour le membre, on enlève '1' à son nombre d'emprunts, puisqu'on supprime un. Enfin, on autorise temporairement le membre à emprunter car, dans le cas d'un blocage de cette autorisation, c'est peut-être CET emprunt que l'on vient de supprimer dont la date de retour était dépassée.

On enregistre ensuite ces modifications dans la base de données avec save().

On ajoute une étape qui, comme pour l'ajout, récupère tous les médias dont l'emprunteur correspond au membre. On va, de nouveau, vérifier si l'un de ces médias à un temps d'emprunt supérieur à 7 jours. Si c'est le cas, on bloque de nouveau la capacité d'emprunt du membre et on enregistre.

On log, on redirige... comme d'habitude.

Note : en écrivant ces lignes je me rend compte qu'il est possible de créer une méthode au sein de la class media permettant de calculer le temps d'emprunt. Ainsi, au lieu de l'écrire deux fois comme c'est le cas ici dans la création ET dans la suppression d'emprunt, il suffirait d'appeler la méthode avec les bons paramètres...

Les tests unitaires

Les tests permettent de vérifier le bon fonctionnement des méthodes et le bon déroulement des actions menées par l'utilisateur. L'objectif est de vérifier chaque étape de façon automatique afin que, lors de modifications du code, l'on puisse en très peu de temps vérifier que le code existant est toujours fonctionnel et qu'il n'y a pas de conflits ou d'effets de bord avec le nouveau code.

Dans ce projet, on n'a fait que de simples vérifications de création d'objets. On vérifie que les objets créés depuis les class correspondent bien aux données que l'on souhaite. Il faudrait, idéalement, créer des tests pour les updates, les deletes, mais aussi les cas particuliers comme les emprunts.

Base de données de test

Le projet contenu dans le repository possède déjà une base de données avec des données de test permettant d'explorer les fonctionnalités. On ne garantit absolument pas la véracité des informations contenues dans cette base...