# TRAVAUX PRATIQUES
# DJANGO : RÉALISATION D'UN CRUD

## I. INTRODUCTION

le but de ce TP est de réaliser un site web permettant de gérer une bibliothéque de livre. Pour cela nous allons nous appuyer sur un modèle de données de **Livre** et créer les pages d'un CRUD. Le CRUD correspond aux actions de :

— **Create** : création d'un **Livre**,
— **Read** : lecture et affichage des données d'un **Livre**,
— **Update** : mise à jour des données d'un **Livre**,
— **Delete** : effacer un **Livre**.

L'ensemble des données sera sauvegardé dans une base sqlite, créée automatiquement par Django. A ces actions liées à un **Livre** nous ajouterons un affichage de l'ensemble des données sous le terme de template **All**

Vous allez donc ajouter une nouvelle application **bibliotheque** dans votre projet créé initialement. Pensez ensuite à l'ajouter à la liste des applications (*Setting.py*) et à ajouter la route de l'application au niveau du projet (*url.py* au niveau du projet)

## II. CRÉATION DU MODÈLE DE DONNÉES

Dans un premier temps, nous allons définir notre modèle de données, correspondant aux colonnes d'une table de stockage dans la base de données. Nous allons définir qu'un textbfLivre est défini par :

— le titre
— l'auteur
— la date de parution
— le nombre de pages
— le résumé

Pour cela, vous allez ajouter une classe de données dans le fichier *models.py*

**Code Python 1: fichier models.py**

```python
from django.db import models

class Livre(models.Model): #déclare la classe Livre héritant de la classe Model, classe de base des modèles
    titre = models.CharField(max_length=100) # défini un champs de type texte de 100 caractères maximum
    auteur = models.CharField(max_length = 100)
    date_parution = models.DateField(blank=True, null = True) # champs de type date, pouvant être null ou ne pas être rempli
    nombre_pages = models.IntegerField(blank=False) # champs de type entier devant être obligatoirement rempli
    resume = models.TextField(null = True, blank = True) # champs de type text long

    def __str__(self):
            chaine = f"{self.titre} écrit par {self.auteur} édité le {self.date_parution}"
            return chaine
```

les types de champs sont référencés sur https ://docs.djangoproject.com/fr/4.0/ref/models/fields/. Vous avez aussi la liste des options possibles que vous pouvez indiqué comme arguments. Vous avez ainsi votre modèle de données. Nous pourrons ensuite ajouter des contraintes sur les champs pour la validations des données saisies dans les formulaires.

## III. Création du formulaire de saisie

Afin de faire un formulaire qui associe les champs de saisie aux différents attributs de la classe de données pour créer un objet nous allons créer un classe dans le ficher forms.py qui effectue cette association.

**Code Python 2: fichier forms.py**

```python
from django.forms import ModelForm
from django.utils.translation import gettext_lazy as _
from . import models

class LivreForm(ModelForm):
    class Meta:
        model = models.Livre
        fields = ('titre', 'auteur', 'date_parution', 'nombre_pages','resume')
        labels = {
            'titre' : _('Titre'),
            'auteur' : _('Auteur') ,
            'date_parution' : _('date␣de␣parution'),
            'nombre_pages' : _('nombres␣de␣pages'),
            'resume' : _('Résumé')
        }
```

Ce formulaire est lié à la classe Livre au travers de la première ligne de la class Meta. On indique ensuite les champs à afficher dans le formulaire puis les labels qui y seront associé au travers d'un dictionnaires. Les champs qui ne serait pas listés dans ce dictionnaire auront un label égal au nom du champs.

Une fois le modèle créé, il faut créer la table associée dans la base de données. Pour cela, nous allons utiliser les commandes Django pour créer cette table.

**Console pyCharm 1: commandes de création des tables dans le terminal venv**

```
# Cette première commande permet d'indiquer que des modèles ont été modifié et que le
    stockage doit être prévu
python manage py makemigrations bibliotheque
# Cette seconde commande éxécute les requêtes de créations des tables
python manage py migrate
```

## IV. création de la page de saisie

A présent, nous allons créer les éléments nécessaire à la production d'une page de formulaire html. Nous allons don créer une action (dans *views.py*, une route ajoutée dans *urls.py* et un gabarit sous la forme du fichier *ajout.html* dans le répertoire *templates/nom-application* de votre application.

IV.1. **l'action.** Dans le fichier *views.py*, nous allons ajouter une action (une fonction) pour envoyer le formulaire au gabarit de rendu en fonction de l'appel de cette page

**Code Python 3: fichier views.py**

```python
from django.shortcuts import render
from .forms import LivreForm
from . import  models

def ajout(request):
    if request.method == "POST": # arrive en cas de retour sur cette page après une
    saisie invalide on récupère donc les données. Normalement nous ne devrions pas
    passer par ce chemin la pour le traitement des données
        form = LivreForm(request)
        if form.is_valid(): # validation du formulaire.
            livre = form.save() # sauvegarde dans la base
            return render(request,"/bibliotheque/affiche.html",{"livre" : livre}) #
    envoie vers une page d'affichage du livre créé
        else:
            return render(request,"bibliotheque/ajout.html",{"form": form})
    else :
        form = LivreForm() # création d'un formulaire vide
        return render(request,"bibliotheque/ajout.html",{"form" : form})
```

Cette fonction vérifie si nous avons déjà envoyé des données ou non (POST signifie que des données ont été encapsulé dans la requête). Si oui, on procède à la validation du formulaire avant l'envoi à la page de traitement, si non, on envoi vers le gabarit du formulaire avec un form vide. Vous pouvez voir qu'on transmet un objet form de type LivreForm au gabarit *ajout.html*

IV.2. **la route.** il faut créer le lien entre l'url et l'action en ajoutant une ligne dans le fichier *urls.py*

**Code Python 4: ligne à ajouter dans fichier urls.py**

```python
 path('ajout', views.ajout),
```

IV.3. **le gabarit.** Pour finir, il faut créer le gabarit qui va permettre d'afficher le formulaire, gabarit indiqué dans l'action sous le nom formulaire.html. Pour cela nous allons donc créer le répertoire *templates* et le sous répertoire livre (le nom de l'application), puis y ajouter un nouveau fichier, *ajout.html*

**Code html 1: fichier ajout.html**

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
    <form method="post" action="/bibliotheque/traitement/">
        {% csrf_token %}
        {{ form.as_ul }} <!-- permet d'afficher tous les champs sous forme d'une liste
-->
        <input type="submit" value="envoyer les données"\>
    </form>
</body>
</html>
```

Vous avez ainsi accès au formulaire de saisie au travers de l'url

`http://127.0.0.1:8000/bibliotheque/ajout/`

Il faut à présent créer l'action de traitement des données qui permettra de récupérer les données des champs de saisie pour remplir un objet **Livre** et le sauvegarder dans la base de données. Nous allons créer l'action, la route et le gabarit associé.

## V. le traitement du formulaire

V.1. **l'url.** dans le formulaire l'url de traitement indiquée est *bibliotheque/traitement*, donc nous allons ajouter cette route au fichier *urls.py*

**Code Python 5: ligne à ajouter dans fichier urls.py**

```python
path('traitement', views.traitement), # ajouter la route traitement associé à l'action
    traitement du fichier views.py
```

V.2. **l'action.** dans le fichier *views.py* nous allons ajouter l'action permettant de récupérer les données du formulaire et de créer l'objet Livre à sauvegarder.

**Code Python 6: Méthode à ajouter dans fichier views.py**

```python
def traitement(request):
    lform = LivreForm(request.POST)
    if lform.is_valid():
        livre = lform.save()
        return render(request,"/bibliotheque/affiche.html",{"livre" : livre})
    else:
        return render(request,"bibliotheque/ajout.html",{"form": lform})
```

Cette action récupère les données transmises par le formulaire sous la forme d'un formulaire de **Livre** rempli. On vérifie la validité des données, et si c'est le cas, on sauvegarde les données et base et on les récupère sous forme d'un objet **Livre**.

V.3. **gabarit.** le gabarit est assez simple, il reprend ce qui a été fait dans le TD3, mais à partir du modèle **Livre** pour lequel nous allons récupéré les attributs

**Code html 2: fichier affiche.html**

```html
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <title>ajout d'un livre</title>
</head>
<body>
  <p> vous avez ajouté le livre suivant {{livre}} {{livre.id}} </p>
</body>
</html>
```

Vous pouvez remarquer qu'un champs supplémentaire l'id est disponible dans l'objet **Livre**. Ce champs correspond à la clé primaire un entier auto-incrémenté ou basé sur une séquence, identifiant de façon unique chaque entrée dans la table Livre. Cet Id servira dans les 3 autres composants du CRUD.

## VI. La liste des éléments

Afin de visualiser l'ensemble des éléments présents dans la base et pouvoir ensuite interagir avec eux, nous allons créer ce qu'on nomme la vue All. Comme usuellement, vous allez créer une route, une action et un gabarit.

L'action va faire appel à la méthode *all()* de la class Livre au travers de *list(models.Livre.objects.all())*. Cette méthode retourne un ensemble d'éléments transformés en liste avec la méthode list(). Cette liste doit être récupérée dans une variable qu'il faudra transmettre au gabarit d'affichage.

Dans le gabarit, nous allons simplement faire une boucle sur l'ensemble des éléments de la liste pour les afficher, soit avec juste leur titre, soit avec la chaine d'affichage définie dans la méthode __str__()

**Code html 3: Rappel code python dans les fichiers html**

```
{%  code python  %}
exemple
{% if livre.nombre_pages > 500 :%}
        <p> gros livre </p>
{%else%}
        <p> moyen livre </p>
{%endif%}
```

chaque bloc possède un élément de terminaison car la tabulation n'est pas prise en compte

## VII. la lecture des informations d'un Livre ou **Read**

dans cette section, nous allons interroger la base afin d'extraire les informations d'un ouvrage en particulier à l'aide de son id. Nous allons à nouveau créer le trio route, action gabarit mais en incluant comme paramètre/argument l'id

VII.1. **la route.** Pour cela, il va falloir construire une route qui comporte un paramètre (l'id) permettant d'informer l'action de cette valeur.

**Code Python 7: ligne à ajouter dans fichier urls.py**

```
path('/affiche/<int:id>/',views.read), # ajouter la route traitement associé à
  l'action traitement du fichier views.py. bien faire attention qu'il n'y ai pas
  d'espace dans les balises <>, sinon cela génère une erreur.
```

vous voyez ici apparaitre un paramètre représenté par <int :id> indiquant le type et le nom de la variable qui sera donnée en argument de l'action L'id est important car c'est lui qui va permettre de récupérer le bon livre dans la base de données.

VII.2. **l'action.** comme l'Id est transmis par la requête, l'action va le récupérer pour interroger la base de données et récupérer l'entrée correspondante.

**Code Python 8: Méthode à ajouter dans fichier views.py**

```
def affiche(request, id):
        livre = models.Livre.objects.get(pk=id) # méthode pour récupérer les données
    dans la base avec un id donnée
         return render(request,"bibliotheque/affiche.html",{"livre": livre})
```

VII.3. **le gabarit d'affichage.** ce gabarit récupère le livre et peut ainsi afficher les données nécessaires, ou simplement utiliser le l'affichage par défaut défini avec la méthode **__str__()** de la classe.

## VIII. mise à jour d'une entrée ou **update**

La mise à jour d'une entrée de la base va nécessiter d'obtenir l'entrée à modifier au travers de son id, et doit permettre de charger un formulaire déjà rempli avec les anciennes valeurs. Lors de la sauvegarde, il faudra utiliser l'id pour ne pas ajouter une nouvelle valeur, mais modifier celle déjà existante. Nous allons donc créer 2 routes, 2 actions, et 2 gabarits, comme pour l'insertion d'une nouvelle entrée.

VIII.1. **route du formulaire de mise à jour.** La route du formulaire de mise à jour, nécessite l'id donc nous avons comme la lecture l'id qui arrive dans la requête.

**Code Python 9: ligne à ajouter dans fichier urls.py**

```
path('/update/<int:id>/',views.update),
```

VIII.2. **l'action de mise à jour.** Cette action doit récupérer le livre avec son id et créer un formulaire pré-rempli avec les données. Pour cela, vous allez récupérer le livre comme dans l'action d'affichage. Ensuite, vous allez créer un formulaire pré-rempli. Pour cela, il faut créer un dictionnaire avec toutes les données de la classe avec les clés correspondant au nom des attributs et le passer comme argument au créateur du formulaire.

```
lform = LivreForm(dictionnaire) # ou le dictionnaire est celui qui contient toutes les valeurs
```

VIII.3. **gabarit.** Ensuite, il suffit de transmettre le formulaire au gabarit pour l'affichage, comme dans le formulaire d'insertion. Par contre dans le formulaire présent dans le gabarit, il faudra aussi transmettre l'id à la route de traitement afin de pouvoir faire la mise à jour.

**Code html 4: fichier update.html**

```
<form method="post" action="/bibliotheque/traitementupdate/{{livre.id}}">
    {% csrf_token %}
    {{ form.as_ul }}
     <input type="submit" value="envoyer les données"\>
</form>
```

## IX. traitement de la mise à jour

Comme pour l'ajout d'un nouveau livre, la mise à jour doit récupérer les données d'un formulaire, les valider puis les sauvegarder dans la base. Par contre dans notre cas, il faut tenir compte de l'id qui n'est pas dans les données de formulaire afin de ne pas créer un nouvelle entrée.

la route doit donc contenir l'Id comme le montre le code html du formulaire dans l'attribut action de la balise form. Ensuite dans l'action, vous devez récupérer les données du formulaire et faire appel à la fonction save mais en passant comme argument *commit=False* afin de ne pas enregistrer directement dans la base de données. vous modifiez ensuite l'id du livre créé avec les données du formulaire en le remplaçant par l'id qui a été transmis à l'action et enfin vous pouvez faire appel à la méthode save() de la classe.

**Code Python 10: Méthode à ajouter dans fichier views.py**

```
def traitementupdate(request, id):
    lform = LivreForm(request.POST)
    if lform.is_valid():
        livre = lform.save(commit=False) # création d'un objet livre avec les données du
    formulaire mais sans l'enregistrer dans la base.
        livre.id = id; # modification de l'id de l'objet
        livre.save() # mise à jour dans la base puisque l'id du livre existe déja.
        return HttpResponseRedirect("/bibliotheque/") # plutot que d'avoir un gabarit
    pour nous indiquer que cela c'est bien passé, nous repartons sur une autre action
    qui renvoie vers la page d'index de notre site (celle avec la liste des entrées)
    else:
        return render(request, "bibliotheque/update.html", {"form": lform, "id": id})
```

Attention, l'usage de HttpResponseRedirect nécessite d'importer la méthode du module django.http

## X. effacer une entrée ou **delete**

De la même façon que pour la mise à jour, il nous faut connaitre l'id d'une entrée pour pouvoir l'effacer. La route doit donc contenir cette information. l'action associée, récupère l'id, récupère l'objet livre de la même façon que dans l'action de lecture et ensuite il suffit de faire appel à la méthode delete() sur le livre puis de faire une redirection vers la page d'accueil pour remplir cette fonction.

## XI. ergonomie du site

Les différentes routes ne sont pas évidentes à gérer. Il est donc préférable à partir de la page d'accueil de pouvoir

- accéder à l'ajout un nouveau livre au travers d'un lien vers le formulaire de saisie.
- acceder pour chaque livre de la liste à la fiche détaillée du livre (au travers d'un lien)
- dans la fiche détaillée, pouvoir faire la mise à jour ou la suppression de l'ouvrage (au travers de lien)