**1. Introduction à Django**

* Qu'est-ce que Django ?
* Installation de Django
* Introduction à l'architecture MVT (Model –View- Template)

**2. Mise en place d’un projet**

* Création d'un projet Django
* Structure d'un projet Django
* Exécution d’un projet Django (serveur de développement)

**3. Premiers pas avec Django**

* Création et gestion d'une application
* Concepts de base des vues et URL en Django
* Création de vues simples
* Manipulation des routes et URLs dans Django
* Navigation entre les applications

**4. Modèles et ORM de Django**

* Qu'est-ce que l'ORM de Django ?
* Création de modèles Django
* Champs de modèles courants (CharField, TextField, DateField, etc.)
* Relations entre modèles (OneToOne, ForeignKey, ManyToMany)
* Migrations avec makemigrations et migrate
* Utilisation de l'API de l'ORM pour interagir avec la base de données (CRUD)

**5. Les formulaires Django**

* Introduction aux formulaires Django
* Création et gestion de formulaires
* Validation des données dans les formulaires
* Utilisation des formulaires pour créer et modifier des objets de la base de données
* Modèles de formulaire (ModelForm)

**6. Templates Django**

* Moteur de template Django : Syntaxe de base
* Utilisation de balises et filtres dans les templates
* Héritage et inclusion de templates
* Gestion des fichiers statiques et des fichiers médias (CSS, JavaScript, images)

**7. Authentification et gestion des utilisateurs**

* Système d’authentification de Django
* Création et gestion des utilisateurs
* Gestion des sessions et des cookies
* Implémentation du système de connexion/déconnexion
* Gestion des permissions et groupes
* Réinitialisation du mot de passe et gestion des e-mails

**8. Django Admin**

* Introduction à l'interface d'administration de Django
* Gestion des utilisateurs et des groupes via l'admin
* Ajout de modèles dans l'admin

**9. Django Rest Framework (DRF)**

* Introduction à Django Rest Framework
* Création d'une API REST avec Django
* Serializers dans DRF
* Vues basées sur les classes (CBV) et vues fonctionnelles dans DRF
* Authentification et permissions dans DRF
* Gestion des requêtes GET, POST, PUT, DELETE avec DRF

### 1. Introduction à Django

#### Qu'est-ce que Django ?

**Django** est un framework web open-source en Python qui permet de développer des applications web rapidement et efficacement. Il est conçu pour encourager les bonnes pratiques de développement, telles que la réutilisation de code, la sécurité et la rapidité.

##### Exemple :

Django est utilisé dans des projets variés comme :

* **Instagram** : pour gérer des millions de photos partagées.
* **Pinterest** : pour organiser des images en collections.
* **Mozilla** : pour des applications liées à Firefox.

#### • Installation de Django

Pour commencer à utiliser Django, il faut d'abord installer Python, puis utiliser **pip** pour installer Django.

##### Étapes d'installation :

1. Installez Python (si nécessaire) en le téléchargeant depuis [python.org](https://www.python.org/).
2. Vérifiez l'installation de Python : **python –version**
3. Créer un environnement virtuel : **python -m venv env**
4. Activer l’environnement virtuel : **env\Scripts\activate**
5. Installez Django avec pip : **pip install django**
6. Vérifiez l'installation de Django : **python -m django --version**

#### Introduction à l'architecture MVT (Model –View- Template)

Django suit l'architecture **MVT** :

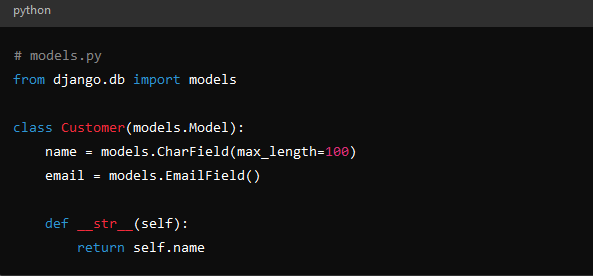
 **Modèle (Model)** : C'est la structure des données. Il définit la base de données via des classes Python.

 **Vue (View)** : C'est la logique qui relie le modèle et le template. Elle traite les requêtes et retourne les réponses.

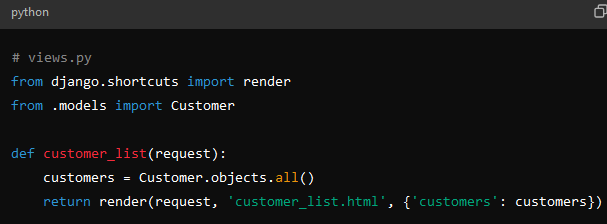
 **Template** : C'est l'interface utilisateur. Les templates sont des fichiers HTML qui affichent les données.

#### Exemple simple :

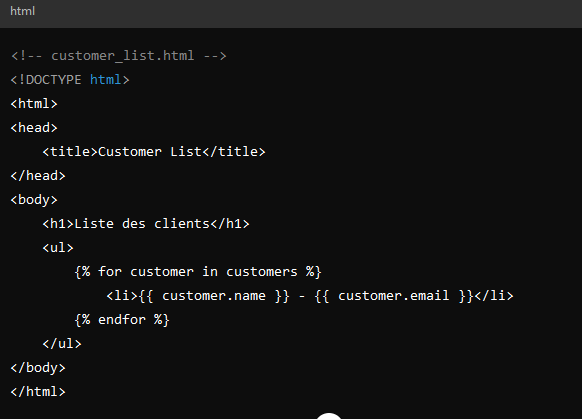
1. **Modèle :** Définition d'un modèle Customer dans models.py.



**Vue :** Création d'une vue qui affiche tous les clients dans views.py.



**Template :** Affichage des données dans un template customer\_list.html.



### Mise en place d’un projet Django

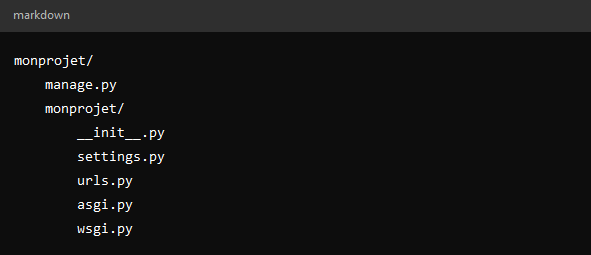
#### Créer un projet Django :

* Exécutez la commande suivante pour créer un nouveau projet :

**django-admin startproject monprojet :** Cela créera un répertoire monprojet contenant la configuration de base du projet.

#### • Structure d'un projet Django

La commande précédente génère une structure comme celle-ci :

****

 **manage.py** : Un script pour gérer le projet (lancer le serveur, faire des migrations, etc.).

 **settings.py** : Le fichier de configuration contenant les paramètres du projet.

 **urls.py** : Gère le routage des URLs vers les vues correspondantes.

###  **wsgi.py/asgi.py** : Fichiers d'entrée pour déployer le projet sur des serveurs web (WSGI/ASGI).

#### • Exécution d’un projet Django (serveur de développement)

1. **Lancer le serveur de développement :**
   * Pour démarrer le serveur de développement, accédez au répertoire du projet (là où se trouve manage.py) et tapez : **python manage.py runserver**
2. **Accéder au projet :**

* Par défaut, le serveur est accessible à l'adresse http://127.0.0.1:8000/. Ouvrez un navigateur et entrez cette URL pour voir la page d'accueil de Django.

### 3. Premiers pas avec Django

#### • Création et gestion d'une application

Dans Django, une **application** est un module Python qui contient des fonctionnalités spécifiques au sein d'un projet Django plus large. Elle est généralement composée de modèles, de vues, de templates, d'URL, de tests, et d'autres éléments nécessaires pour implémenter une fonctionnalité donnée.

1. **Créer une application Django :**
   * Dans le répertoire principal du projet (où se trouve manage.py), exécutez :

**python manage.py startapp monapplication :** Cela crée un répertoire monapplication avec une structure de base pour une application Django.

1. **Ajouter l'application au projet :**

* Ouvrez settings.py dans le répertoire du projet (monprojet/monprojet/).
* Ajoutez monapplication à la liste INSTALLED\_APPS :



#### • Concepts de base des vues et URL en Django

* **Vues (Views) :** Les vues traitent les requêtes et renvoient des réponses. Elles sont définies dans le fichier views.py de l'application.
* **URLs :** Les URL définissent comment les requêtes HTTP sont acheminées vers les vues. Elles sont configurées dans urls.py.

#### • Création de vues simples

**Définir une vue :**

* Ouvrez views.py dans votre application (monapplication/views.py).
* Créez une vue simple qui retourne une réponse HTTP :

from django.http import HttpResponse

def hello\_world(request):

    return HttpResponse("Bonjour, monde !")

**Configurer l'URL pour la vue :**

* Créez ou ouvrez le fichier urls.py dans le répertoire de l'application (monapplication/urls.py). Si le fichier n'existe pas, créez-le.
* Configurez les routes pour vos vues :

from django.urls import path

from . import views

urlpatterns = [

    path('hello/', views.hello\_world),

]

**Inclure les URLs de l'application dans le projet :**

* Ouvrez urls.py dans le répertoire du projet (monprojet/monprojet/urls.py).
* Ajoutez une ligne pour inclure les URLs de l'application :

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('monapplication/', include('monapplication.urls')),

]

#### • Manipulation des routes et URLs dans Django

1. **Utiliser des paramètres dans les URLs :**
   * Vous pouvez ajouter des paramètres dans les URLs pour les passer aux vues :

# monapplication/urls.py

from django.urls import path

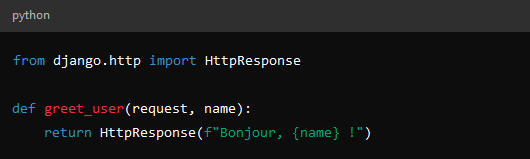
from . import views

urlpatterns = [

    path('hello/<str:name>/', views.greet\_user),

]

* Dans views.py, créez une vue pour gérer les paramètres :



**Utiliser des URL nommées :**

* Vous pouvez donner un nom aux URLs pour une utilisation facile dans les templates :

# monapplication/urls.py

from django.urls import path

from . import views

urlpatterns = [

    path('hello/<str:name>/', views.greet\_user, name='greet\_user'),

]

* Dans les templates, vous pouvez utiliser le nom de l'URL pour générer des liens :

<a href="{% url 'greet\_user' name='Alice' %}">Greet Alice</a>

#### Navigation entre les applications

**Ajoutons 3 app1, app2 et app3 dans le projet**

* 1. **Création de trois applications :**

python manage.py startapp app1

python manage.py startapp app2

python manage.py startapp app3

* 1. **Ajout des applications dans settings.py :** Dans le fichier monprojet/settings.py, ajoute chaque application dans la section INSTALLED\_APPS :

INSTALLED\_APPS = [

    'app1',  # Ajoute l'application 1

    'app2',  # Ajoute l'application 2

    'app3',  # Ajoute l'application 3

]

* 1. **Création des vues simples pour chaque application :** Dans chaque application, nous allons créer des vues qui renverront des pages HTML statiques.
* **app1/views.py**

from django.shortcuts import render

# Create your views here.

def home\_app1(request):

    return render(request, 'home\_app1.html')

* **app2/views.py**

from django.shortcuts import render

# Create your views here.

def home\_app2(request):

    return render(request, 'home\_app2.html')

* **app3/views.py**

from django.shortcuts import render

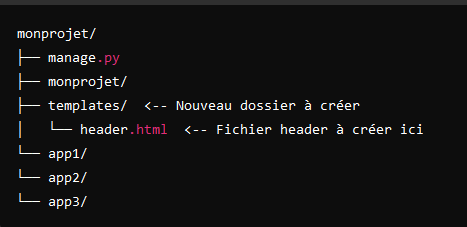
# Create your views here.

def home\_app3(request):

    return render(request, 'home\_app3.html')

**Création des templates HTML :** Pour chaque vue, on va créer un fichier HTML simple.

On va d’abord créer un fichier header.html dans le dossier templates a la racine du projet qui sera partagé par toutes les pages via le mécanisme d'inclusion de Django.



**Configurer Django pour reconnaître le dossier templates :**

Dans le fichier monprojet/settings.py, assurez-vous que Django sait où chercher les templates globaux. Cela se fait en ajoutant le chemin vers le dossier templates dans la configuration TEMPLATES :

TEMPLATES = [

    {

        'BACKEND': 'django.template.backends.django.DjangoTemplates',

        'DIRS': [BASE\_DIR / 'templates'],  # Ajoute le chemin vers le dossier templates

        'APP\_DIRS': True,

        'OPTIONS': {

            'context\_processors': [

                'django.template.context\_processors.debug',

                'django.template.context\_processors.request',

                'django.contrib.auth.context\_processors.auth',

                'django.contrib.messages.context\_processors.messages',

            ],

        },

    },

]

* **Création du template header.html :**

Crée un nouveau fichier templates/header.html qui contiendra les liens de navigation :

<header>

    <nav>

        <ul>

            <li><a href="{% url 'home\_app1' %}">Application 1</a></li>

            <li><a href="{% url 'home\_app2' %}">Application 2</a></li>

            <li><a href="{% url 'home\_app3' %}">Application 3</a></li>

        </ul>

    </nav>

</header>

**Création des templates HTML avec inclusion du header :** Chaque page HTML inclura désormais ce header.html pour afficher les liens de navigation dynamiques.

* **app1/templates/home\_app1.html**

{% include 'header.html' %}

<html>

<body>

     <h1>Bienvenue dans l'application 1</h1>

</body>

</html>

* **app2/templates/home\_app2.html**

{% include 'header.html' %}

<html>

<body>

     <h1>Bienvenue dans l'application 2</h1>

</body>

</html>

* **app3/templates/home\_app3.html**

{% include 'header.html' %}

<html>

<body>

     <h1>Bienvenue dans l'application 3</h1>

</body>

</html>

**Définition des URLs pour chaque application :** Ajoutons les routes pour chaque application.

* **app1/urls.py**

from django.urls import path

from . import views

urlpatterns = [

    path('', views.home\_app1, name='home\_app1'),

]

**Ajouter les URLs principales dans monprojet/urls.py :** Enfin, on relie chaque application à l'URL principale.

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('app1/', include('app1.urls')),

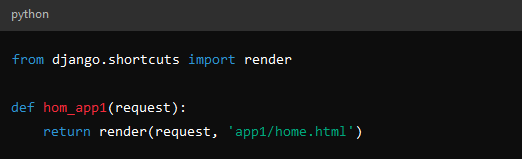
    path('app2/', include('app2.urls')),

    path('app3/', include('app3.urls')),

]

**NB:** Pour créer des vues distinctes pour plusieurs applications (app1, app2, app3) dans un projet Django, en s'assurant que chaque vue référence le bon template, même si les noms des templates sont identiques il faudra respecter cette structure :****

Et faire ceci pour toutes les vue :



**4. Modèles et ORM de Django**

**1- Qu'est-ce que l'ORM de Django ?**

L'ORM (Object-Relational Mapping) de Django est un système qui permet de manipuler les bases de données en utilisant des objets Python. Au lieu d'écrire des requêtes SQL directement, tu définis des modèles Python qui correspondent aux tables de la base de données, et Django s'occupe de la traduction entre ces objets et les tables SQL.

Créons un nouveau projet simple dans lequel nous allons appliquer les relations **OneToOneField**, **ForeignKey**, et **ManyToManyField**, ainsi que réaliser des opérations CRUD (sans utiliser de formulaires pour le moment):

* + 1. django-admin startproject library\_project
    2. **Créer trois applications Author, Book, et Publisher** :

python manage.py startapp authors

python manage.py startapp books

python manage.py startapp publishers

* + 1. **Ajouter les applications à INSTALLED\_APPS** dans library\_project/settings.py :

INSTALLED\_APPS = [

    # ...

    'authors',

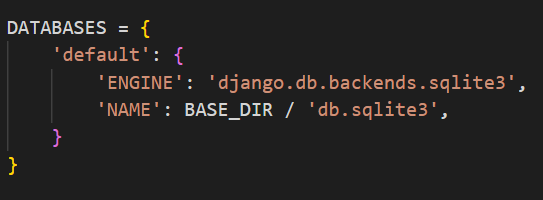
    'books',

    'publishers',

]

* + 1. Configuration de la base de donnée dans le settings

Par defaut c’est sqlit qui est utiliser



Nous allons utiliser mysql a la palce en installant ‘**pip install mysqlclient’**

DATABASES = {

    'default': {

        'ENGINE': 'django.db.backends.mysql',

        'NAME': 'libraryProject',

        'HOST': '127.0.0.1',

        'USER': 'root',

        'PASSWORD': '',

        'PORT': 3306,

    }

}

Avant de quitter le settings nous allons profiter pour configuer le chemin des finchier globaux(voir la section precedente)

* + 1. Définir les Modèles

**Modèle Author** (authors/models.py) :

from django.db import models

class Author(models.Model):

    name = models.CharField(max\_length=100)

    birthdate = models.DateField()

    biography = models.TextField()

    def \_\_str\_\_(self):

        return self.name

**Modèle Publisher** (publishers/models.py) :

from django.db import models

class Publisher(models.Model):

    name = models.CharField(max\_length=100)

    address = models.CharField(max\_length=255)

    def \_\_str\_\_(self):

        return self.name

**Modèle Book** (books/models.py) :

from django.db import models

class ISBN(models.Model):

    code = models.CharField(max\_length=20)

    # ISBN est composé de 20 caractères

    def \_\_str\_\_(self):

        return self.code

class Book(models.Model):

    title = models.CharField(max\_length=200)

    # Le titre du livre

    publication\_date = models.DateField()

    # La date de publication

    summary = models.TextField()

    # Résumé du livre

    author = models.ForeignKey(Author, on\_delete=models.CASCADE)

    # ForeignKey, plusieurs livres peuvent avoir un même auteur

    isbn = models.OneToOneField(ISBN, on\_delete=models.CASCADE)

    # OneToOneField, chaque livre a un ISBN unique

    publishers = models.ManyToManyField(Publisher)

    # ManyToManyField, un livre peut avoir plusieurs éditeurs

    def \_\_str\_\_(self):

        return self.title

* + 1. Créer et Appliquer les Migrations dans Django

Django utilise un système de migrations pour synchroniser les modèles (définis dans le code) avec la base de données. Lorsque tu modifies les modèles (comme ceux pour Book, Author, Publisher, ou ISBN), tu dois générer et appliquer des **migrations** pour que ces modifications soient reflétées dans la base de données.

Après avoir défini ou modifié les modèles dans ton projet Django, la première étape est de générer des migrations. Les migrations sont des fichiers qui enregistrent les changements apportés à la structure de la base de données en fonction des modifications dans les modèles.

#### Commande :

python manage.py makemigrations

* + 1. **Appliquer les Migrations** avec migrate

Une fois les migrations créées, la commande suivante permet d'appliquer ces changements à la base de données.

#### Commande :

python manage.py migrate

 Cette commande exécute les fichiers de migration générés et applique les changements à la base de données.

 Elle va créer les tables SQL correspondantes dans la base de données (si elles n'existent pas encore) ou les modifier selon les changements dans tes modèles.

g- **Configurer les URLs**

Pour chaque application, nous allons définir des vues pour lister, afficher les détails, et gérer le CRUD (Create, Read, Update, Delete).

**URLs pour l'application authors** (authors/urls.py) :

from django.urls import path

from . import views

app\_name = 'authors' # c'est un namespace

urlpatterns = [

    path('', views.author\_list, name='author\_list'),

    path('<int:id>/', views.author\_detail, name='author\_detail')

]

**URLs pour l'application books** (books/urls.py) :

from django.urls import path

from . import views

app\_name = 'books'

urlpatterns = [

    path('', views.book\_list, name='book\_list'),

    path('<int:id>/', views.book\_detail, name='book\_detail')

]

**URLs pour l'application publishers** (publishers/urls.py) :

from django.urls import path

from . import views

app\_name = 'publishers'

urlpatterns = [

   path('', views.publisher\_list, name='publisher\_list'),

  path('<int:id>/', views.publisher\_detail, name='publisher\_detail')

]

**Configurer les URLs du projet** (library\_project/urls.py) :

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('admin/', admin.site.urls),

    path('authors/', include('authors.urls')),

    path('books/', include('books.urls')),

    path('publishers/', include('publishers.urls'))

]

**h- Les vues pour les applications**

**Vues pour l'application authors** (authors/views.py) :

from django.shortcuts import render, get\_object\_or\_404

from .models import Author

def author\_list(request):

    authors = Author.objects.all() # select \* from Author;

    return render(request, 'authors/author\_list.html', {'authors': authors})

def author\_detail(request, id):

  author = get\_object\_or\_404(Author, id=id) # select \* from Author where id=id

  return render(request, 'authors/author\_detail.html', {'author': author})

**Vues pour l'application books** (books/views.py) :

from django.shortcuts import render, get\_object\_or\_404

from .models import Book

def book\_list(request):

    books = Book.objects.all()

    return render(request, 'books/book\_list.html', {'books': books})

def book\_detail(request, id):

    book = get\_object\_or\_404(Book, id=id)

    return render(request, 'books/book\_detail.html', {'book': book})

**Vues pour l'application publishers** (publishers/views.py) :

from django.shortcuts import render, get\_object\_or\_404

from .models import Publisher

def publisher\_list(request):

    publishers = Publisher.objects.all()

     # Récupère tous les éditeurs

    return render(request, 'publishers/publisher\_list.html', {'publishers': publishers})

    # Rendu de la liste des éditeurs

def publisher\_detail(request, id):

    publisher = get\_object\_or\_404(Publisher, id=id)

     # Récupère un éditeur par ID ou renvoie une 404 s’il n’existe pas

    return render(request, 'publishers/publisher\_detail.html', {'publisher': publisher})

    # Rendu des détails de l'éditeur

Cette fois ci nous allons changer de technique histoire de grandir un peu +

Au lieu de faire comme avant avec les inclussions, nous allons faire hériter les pages, genre créer une page de base que toutes les autres pages vont hériter pour afficher leurs contenue on parle souvent de block.

Il faut créer un dossier qui s’appelle exactement ‘**templates’**  à la racine du projet comme précédemment avec les app (1, 2 et 3) qu’on a eu à faire (voir en haut), dans ce dossier nous allons créer toutes les pages globales.

Exemple de page de base :

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>{% block title %}Mon Site Django{% endblock %}</title>

</head>

<body>

    <header>

        <h1>Bienvenue sur mon site de gestion de livres</h1>

       <nav>

        <ul>

         <li><a href="{% url 'authors:author\_list' %}">Auteurs</a></li>

         <li><a href="{% url 'books:book\_list' %}">Livres</a></li>

         <li><a href="{% url 'publishers:publisher\_list' %}">Éditeurs</a></li>

        </ul>

       </nav>

      </header>

      <main>

        {% block content %}

        <!-- contenu de la page ici -->

        {% endblock %}

      </main>

      <footer>

        <p>&copy; 2024 Mon Projet Django</p>

      </footer>

</body>

</html>

Vous remarquez qu’il y a une balise étranges ‘**block’** et ‘**endblock’**  avec un attribut ‘**content’** cette balise n’appartient pas au HTML mais à Django, son importance est d’afficher tout le contenu d’une quelconque page qui l’héritera dans le future, et tout le contenu de la page héritant est injecter dans le ‘**content’** puis Django vas se débrouiller pour nous montrer le contenue.

80% des projets fullStack Django professionnel son conçu presque de la sorte.

Voyons comment faire ce fameux héritage avec les différentes pages qu’on a besoin

**Template de Liste (author\_list.html) :**

{% extends "base.html" %}

{% block title %}Liste des Auteurs{% endblock %}

{% block content %}

  <h2>Liste des Auteurs</h2>

  <ul>

    {% for author in authors %}

<li><a href="{% url 'authors:author\_detail' author.id %}">

{{ author.name }}</a></li>

    {% endfor %}

  </ul>

{% endblock %}

Le fichier author\_list.html étend le fichier base.html pour utiliser la structure de base du site :

Cette page affiche tous les auteurs mais qui seront visible dans base.html.

**Template de detail (author\_detail.html) :**

{% extends "base.html" %}

{% block title %}Détail de l'Auteur{% endblock %}

{% block content %}

  <h2>Détails de l'Auteur</h2>

  <p><strong>Nom :</strong> {{ author.name }}</p>

  <p><strong>Biographie :</strong> {{ author.biography }}</p>

  <h3>Livres de cet auteur :</h3>

  <ul>

    {% for book in author.book\_set.all %}

      <li>{{ book.title }} ({{ book.publication\_date }})</li>

    {% endfor %}

  </ul>

  <a href="{% url 'authors:author\_list' %}">Retour à la liste des auteurs</a>

{% endblock %}

La méthode author.book\_set.all est obtenue automatiquement par Django grâce à la relation **ForeignKey** définie dans le modèle Book.

**Template de liste (**book\_list**.html) :**

{% extends "base.html" %}

{% block title %}Liste des Livres{% endblock %}

{% block content %}

  <h2>Liste des Livres</h2>

  <ul>

    {% for book in books %}

      <li><a href="{% url 'books:book\_detail' book.id %}">{{ book.title }}</a></li>

    {% endfor %}

  </ul>

{% endblock %}

Template book\_detail.html :

{% extends "base.html" %}

{% block title %}Détail du livre{% endblock %}

{% block content %}

  <h2>Détails du Livre</h2>

  <p><strong>Titre :</strong> {{ book.title }}</p>

  <p><strong>Date de publication :</strong> {{ book.publication\_date }}</p>

  <p><strong>Résumé :</strong> {{ book.summary }}</p>

  <p><strong>Auteur :</strong> <a href="{% url 'authors:author\_detail' book.author.id %}">

    {{ book.author.name }}</a></p>

  <p><strong>Éditeurs :</strong></p>

  <ul>

    {% for publisher in book.publishers.all %}

      <li>{{ publisher.name }}</li>

    {% endfor %}

  </ul>

  <a href="{% url 'books:book\_list' %}">Retour à la liste des livres</a>

{% endblock %}

 Pour accéder aux objets liés par un ManyToManyField, utilise simplement le nom du champ (comme book.publishers.all()).

 Django gère automatiquement cela sans nécessiter le suffixe set, car chaque côté de la relation peut contenir plusieurs objets.

Template publisher\_list.html :

{% extends "base.html" %}

{% block title %}Liste des Éditeurs{% endblock %}

{% block content %}

  <h2>Liste des éditeurs</h2>

  <ul>

    {% for publisher in publishers %}

      <li><a href="{% url 'publishers:publisher\_detail' publisher.id %}">

        {{ publisher.name }}</a></li>

    {% endfor %}

  </ul>

{% endblock %}

Template publisher\_detail.html :

{% extends "base.html" %}

{% block title %}Détail de l'éditeur{% endblock %}

{% block content %}

  <h2>Détails de l'Éditeur</h2>

  <p><strong>Nom :</strong> {{ publisher.name }}</p>

  <h3>Livres publiés par cet éditeur :</h3>

  <ul>

    {% for book in publisher.book\_set.all %}

      <li>{{ book.title }} ({{ book.publication\_date }})</li>

    {% endfor %}

  </ul>

  <a href="{% url 'publishers:publisher\_list' %}">Retour à la liste des éditeurs</a>

{% endblock %}

Utiliser un module Django ‘faker’ pour generer des fausses donnée dans la base de donnée

Installer faker : **pip install faker**

Puis ecrire un srcipt avec faker pour remplire les models

Lien github pour cloner le fichier : <https://github.com/darasow/populateModel.git>

Executer le fichier : python populate.py , si le fichier s’appel populate.py

Crud :

Simullons un crud complet sur les Modeles en utilisant module faker :

* 1. Authors :

Ajout des vues Add, delete, update

* Add

# Ajouter un auteur avec des données factices via Faker Sans oublier l’import

from faker import Faker

fake = Faker()# Initialiser Faker

def add\_author(request):

    author = Author(

        name=fake.name(),  # Générer un nom factice

        birthdate=fake.date\_of\_birth(),# Générer une date de naissance factice

        biography=fake.text()  # Générer une biographie factice

    )

    author.save()

    return HttpResponseRedirect('/authors/')

* Delete

# Supprimer un auteur

def delete\_author(request, id):

    author = get\_object\_or\_404(Author, id=id)

    author.delete()

    return HttpResponseRedirect('/authors/')

* Update

# Mettre à jour un auteur avec des données factices via Faker

def update\_author(request, id):

    author = get\_object\_or\_404(Author, id=id)

    author.name = fake.name()  # Nouveau nom factice

    author.birthdate = fake.date\_of\_birth()  # Nouvelle date de naissance factice

    author.biography = fake.text()  # Nouvelle biographie factice

    author.save()

    return HttpResponseRedirect('/authors/')

* 1. Configuration des routes

Metons a jour **authors/urls.py**

from django.urls import path

from . import views

app\_name = 'authors'

urlpatterns = [

    # Liste des auteurs

 path('', views.author\_list, name='author\_list'),

      # Détails d'un auteur

 path('<int:id>/', views.author\_detail, name='author\_detail'),

      # Ajouter un auteur

 path('add\_author/', views.add\_author, name='add\_author'),

      # Supprimer un auteur

 path('delete\_author/<int:id>/', views.delete\_author, name='delete\_author'),

    # Mettre à jour un auteur

 path('update\_author/<int:id>/', views.update\_author, name='update\_author'),

]

Metons a jour **authors/templates/author\_liste.html :**

{% extends "base.html" %}

{% block title %}Liste des Auteurs{% endblock %}

{% block content %}

    <h2>Liste des Auteurs</h2>

    <ul>

        {% for author in authors %}

            <li>

 <a href="{% url 'authors:author\_detail' author.id %}">{{ author.name }}</a>

                <!-- Supprimer un auteur -->

<a href="{% url 'authors:delete\_author' author.id %}">Supprimer</a>

                <!-- Mettre à jour un auteur -->

 <a href="{% url 'authors:update\_author' author.id %}">Modifier</a>

            </li>

        {% endfor %}

    </ul>

    <!-- Lien pour ajouter un auteur avec des données factices -->

    <a href="{% url 'authors:add\_author' %}">Ajouter un auteur (Faker)</a>

{% endblock %}

Mettez a jour les models pour introduire la date de creation et modification comme :

from django.db import models

from django.utils import timezone

class Author(models.Model):

    name = models.CharField(max\_length=100)

    birthdate = models.DateField()

    biography = models.TextField()

    created\_at = models.DateTimeField(null=True, blank=True)

    modified\_at = models.DateTimeField(null=True, blank=True)

    def save(self, \*args, \*\*kwargs):

        if not self.pk:  # Si l'objet est en train d'être créé

            self.created\_at = timezone.now()

            self.modified\_at = None

        else:

            self.modified\_at = timezone.now()

        super(Author, self).save(\*args, \*\*kwargs)

    def \_\_str\_\_(self):

        return self.name

**5. Formulaire en Django**

Dans ce module, nous allons explorer comment créer, gérer, et valider des formulaires Django, et utiliser ces formulaires pour créer et modifier des objets en base de données. Nous allons utiliser tous les modèles: Author, Book, Publisher, et ISBN, et montrer comment gérer les relations ForeignKey, OneToOneField, et ManyToManyField dans les formulaires.

Django propose deux types de formulaires :

* **Formulaires classiques** : Crées manuellement sans lien avec un modèle
* **Formulaires basés sur un modèle (ModelForm)** : Générés automatiquement à partir d’un modèle

### 1. Formulaire Classique pour le Modèle Author

Les formulaires classiques sont créés sans lien direct avec un modèle. Nous devons définir manuellement tous les champs et gérer l’enregistrement des données dans la vue.

#### Formulaire Classique pour Author

Créer un fichier a la racine de l’application : **authors/forms.py**

from django import forms

import datetime

class ClassicAuthorForm(forms.Form):

    name = forms.CharField(max\_length=100, label="Nom de l'auteur")

    birthdate = forms.DateField(widget=forms.SelectDateWidget( years=range(1800, datetime.date.today().year + 1)),

                                 label="Date de naissance")

    biography = forms.CharField(widget=forms.Textarea, label="Biographie", required=False)

 **CharField** : Pour le nom de l’auteur.

 **DateField** : Pour la date de naissance avec un sélecteur de date.

 **Textarea** : Pour la biographie, champ facultatif.

 **widget=forms.SelectDateWidget()** : Cela spécifie le type de widget à utiliser pour ce champ. SelectDateWidget crée trois menus déroulants permettant à l'utilisateur de sélectionner le jour, le mois et l'année de la date de naissance. Cela rend la saisie de la date plus intuitive et évite les erreurs de format.

 **label="Date de naissance"** : Cet attribut définit le texte d'étiquette affiché à côté du champ dans le formulaire. Cela guide l'utilisateur sur le type de données attendu pour ce champ, dans ce cas, la date de naissance.

**La vue d’ajout :**

def add\_author(request):

    if request.method == 'POST':

        form = ClassicAuthorForm(request.POST)

        if form.is\_valid():

            # Crée un objet Author à partir des données du formulaire

            Author.objects.create(

                name=form.cleaned\_data['name'],

                birthdate=form.cleaned\_data['birthdate'],

                biography=form.cleaned\_data.get('biography', '')

            )

            return redirect('authors:author\_list')  # Redirige après l'ajout

    else:

        form = ClassicAuthorForm()

    return render(request, 'authors/author\_form.html', {'form': form})

 **form.cleaned\_data** : Accède aux données validées du formulaire.

 **Author.objects.create** : Crée un nouvel auteur dans la base de données.

Le lien d’ajout reste le même dans le templates:

    <!-- Lien pour ajouter un auteur avec des données factices -->

    <a href="{% url 'authors:add\_author' %}">Ajouter un auteur</a>

Ajoutons un fichier **authors/templates/authors/author\_form.html :**

{% extends "base.html" %}

{% block content %}

<h2>Ajouter un Auteur</h2>

<form method="POST">

    {% csrf\_token %}

    {{ form.as\_p }}  <!--Affiche tous les champs sous forme de paragraphes-->

    <button type="submit">Ajouter l'auteur</button>

</form>

{% endblock content %}

**La modification :**

On va utiliser le même formulaire d’ajout pour la modification aussi mais il faudra mettre à jour la class **ClassicAuthorForm**

Ajouter la méthode **\_\_init\_\_()** dans la class**:**

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        # Si des données initiales sont fournies, les utiliser pour remplir le formulaire

        initial = kwargs.get('initial', {})

        super(ClassicAuthorForm, self).\_\_init\_\_(\*args, \*\*kwargs)

        if initial:

            self.fields['birthdate'].initial = initial.get('birthdate')

            self.fields['name'].initial = initial.get('name')

            self.fields['biography'].initial = initial.get('biography')

**La vue de modification :**

def update\_author(request, id):

    author = get\_object\_or\_404(Author, id=id)

    if request.method == 'POST':

        form = ClassicAuthorForm(request.POST, initial={

            'birthdate': author.birthdate,

            'name': author.name,

            'biography': author.biography

        })

        if form.is\_valid():

            author.name = form.cleaned\_data['name']

            author.birthdate = form.cleaned\_data['birthdate']

            author.biography = form.cleaned\_data['biography']

            author.save()

            return redirect('authors:author\_list')

    else:

        form = ClassicAuthorForm(initial={

            'birthdate': author.birthdate,

            'name': author.name,

            'biography': author.biography

        })  # Pré-remplissage pour modification

    return render(request, 'authors/author\_form.html', {'form': form})

A ce stade l’ajout et la modification utilise le même formulaire, la suppression n’a pas besoin de formulaire, alors le crud des Authors est complet sans utiliser faker

* 1. **ModelForm pour le Modèle Author**

Utilisation des formulaires basés sur les Models :

Les **ModelForms** sont plus simples à utiliser lorsqu’on veut directement lier un formulaire à un modèle Django. Django crée automatiquement les champs du formulaire en fonction du modèle.

#### ModelForm pour Author

On vas modifier le fichier **authors/forms.py** pour utiliser le **ModelForm**

from django import forms

from .models import Author

class AuthorForm(forms.ModelForm):

    class Meta:

        model = Author

        fields = ['name', 'birthdate', 'biography']

**Meta** : Spécifie le modèle Author et les champs que nous voulons inclure dans le formulaire.

**Vue pour Ajouter un Auteur avec un ModelForm**

def add\_author(request):

    if request.method == 'POST':

        form = AuthorForm(request.POST)

        if form.is\_valid():

            form.save()  # Enregistre directement l'auteur en base de données

            return redirect('authors:author\_list')

    else:

        form = AuthorForm()

    return render(request, 'authors/author\_form.html', {'form': form})

Vous remarquez que c’est plus simple qu’avant.

NB : Pour le champ Date de naissance, dans le model son type est DateField() , mais il ne va pas afficher un input de type date dans le formulaire, il va afficher un input de type text, pour éviter cela il faudra préciser à django qu’il s’agit bien de type date

class AuthorForm(forms.ModelForm):

    birthdate = forms.DateField(

        widget=forms.DateInput(attrs={'type': 'date'}),

        label="Date de naissance"

    )

    class Meta:

        model = Author

        fields = ['name', 'birthdate', 'biography']

On parle en HTML depuis django c’est super !!!

Le template HTML **author\_form.html** n’a toujours pas changé

{% extends "base.html" %}

{% block content %}

<h2>Ajouter un Auteur</h2>

<form method="POST">

    {% csrf\_token %}

    {{ form.as\_p }}  <!-- Affiche tous les champs sous forme de paragraphes -->

    <button type="submit">Ajouter l'auteur</button>

</form>

{% endblock content %}

**La modification :**

Nous allons toujours utiliser le même formulaire d’ajout pour la modification

Ici rien ne vas changer dans le **AuthorForm**  nous allons juste faire la logique dans la vue de modification.

def update\_author(request, id):

    author = get\_object\_or\_404(Author, id=id)

    if request.method == 'POST':

        # Liaison avec l'instance existante

        form = AuthorForm(request.POST, instance=author)

        if form.is\_valid():

            form.save()  # Met à jour l'auteur existant

            return redirect('authors:author\_list')

    else:

         # Pré-remplissage pour la modification

        form = AuthorForm(instance=author)

    return render(request,'authors/author\_form.html',{'form': form})

**Qu'est-ce que CSRF ?**

* **Cross-Site Request Forgery** (CSRF) est une attaque où un utilisateur malveillant incite un utilisateur authentifié à effectuer des actions non désirées sur une application web dans laquelle il est connecté.
* Par exemple, un attaquant peut envoyer un lien ou une image à un utilisateur, et si cet utilisateur clique sur ce lien alors qu'il est connecté à une application, cela peut entraîner des modifications non autorisées dans l'application.

**Rôle de {% csrf\_token %}**

* **Sécurisation des formulaires** : Lorsqu'un formulaire est soumis, Django vérifie si le jeton CSRF envoyé avec la requête correspond à celui qui a été généré lors du chargement de la page. Si les jetons ne correspondent pas, Django renvoie une erreur 403 (Forbidden) pour empêcher l'action.
* **Génération d'un jeton unique** : Lorsque le template est rendu, Django génère un jeton CSRF unique pour cette session et l'inclut dans le formulaire. Ce jeton est stocké dans la session de l'utilisateur.

On va booster notre projet en créant des formulaires pour les autres.

**ModelForm pour ISBN**

**Dans books/forms.py:**

from django import forms

from .models import ISBN

class ISBNForm(forms.ModelForm):

    class Meta:

        model = ISBN

        fields = ['code']

Comme le model ISBN est rattacher au Model Books, tout le config de ISBN est sera dans l’appli Books:

Mettons a jour Books/urls.py :

from django.urls import path

from . import views

app\_name = 'books'

urlpatterns = [

    # Les route pour les livres

    path('', views.book\_list, name='book\_list'),

    path('<int:id>/', views.book\_detail, name='book\_detail'),

    # Les route pour les codes

    path('codes/', views.code\_list, name='code\_list'),

    path('add\_code/', views.add\_code, name='add\_code'),

    path('code\_delete/<int:id>', views.code\_delete, name='code\_delete'),

    path('code\_update/<int:id>', views.code\_update, name='code\_update'),

]

Les vue Pour les codes toujours dans books/views.py:

def code\_list(request):

    codes = ISBN.objects.all()

    return render(request, 'books/code\_list.html', {'codes': codes})

def add\_code(request):

    if request.method == 'POST':

        form = ISBNForm(request.POST)

        if form.is\_valid():

            form.save()  # Enregistre le code ISBN

         # Redirige vers la liste des codes

            return redirect('books:code\_list')

    else:

        form = ISBNForm()

    return render(request, 'books/code\_form.html', {'form': form})

def code\_delete(request, id):

    code = get\_object\_or\_404(ISBN, id=id)

    code.delete()

    return redirect('books:code\_list')

def code\_update(request, id):

    code = get\_object\_or\_404(ISBN, id=id)

    if request.method == 'POST':

        # Liaison avec l'instance existante

        form = ISBNForm(request.POST, instance=code)

        if form.is\_valid():

            form.save()  # Met à jour l'auteur existant

            return redirect('books:code\_list')

    else:

         # Pré-remplissage pour modification

        form = ISBNForm(instance=code)

    return render(request, 'books/code\_form.html', {'form': form})

Inutile de rappeler qu’il faudra créer tous les fichier nécessaire dans books/templates/books

Et mettre à jour notre fichier de base, monprojet/templates/base.html pour inclure le lien vers la liste des codes :

  <ul>

        <li><a href="{% url 'authors:author\_list' %}">Auteurs</a></li>

        <li><a href="{% url 'books:book\_list' %}">Livres</a></li>

        <li><a href="{% url 'books:code\_list' %}">Codes</a></li>

        <li><a href="{% url 'publishers:publisher\_list' %}">Éditeurs</a></li>

 </ul>

Le template d’ajout et modification des codes :

{% extends "base.html" %}

{% block content %}

<h2>Ajouter un Code ISBN</h2>

<form method="POST">

    {% csrf\_token %}

    {{ form.as\_p }}  <!-- Affiche le champ du code ISBN -->

    <button type="submit">Ajouter</button>

</form>

{% endblock  %}

Le template **books/templates/books/code\_list.html:**

{% extends "base.html" %}

{% block title %}Liste des Code{% endblock %}

{% block content %}

    <h2>Liste des Codes</h2>

    <ul>

        {% for code in codes %}

        <li>

                <span >{{ code.code }}</span>

            <a style="color : red" href="{% url 'books:code\_delete' code.id %}">Supprimer</a>

            <!-- Mettre à jour un auteur -->

            <a style="color : green" href="{% url 'books:code\_update' code.id %}">Modifier</a>

        </li>

        {% endfor %}

    </ul>

        <!-- Lien pour ajouter un code -->

        <a href="{% url 'books:add\_code' %}">Ajouter un code</a>

{% endblock %}

En mettant à jour le projet ainsi, on a fait le crud des codes en utilisant les ModelForm

**On continue sur les Livres :**

Commençons par la class du formulaire dans **books/forms.py**:

class BookForm(forms.ModelForm):

    class Meta:

        model = Book

        fields = ['title', 'publication\_date', 'summary', 'author', 'isbn', 'publishers']

        widgets = {

            'title': forms.TextInput(attrs={'placeholder': 'Titre du livre'}),

            'publication\_date': forms.DateInput(attrs={'type': 'date'}),

            'summary': forms.Textarea(attrs={'placeholder': 'Résumé du livre'}),

        }

Apres les routes pour le crud dans **books/urls.py**:

from django.urls import path

from . import views

app\_name = 'books'

urlpatterns = [

    # Les routes pour les livres

    path('', views.book\_list, name='book\_list'),

    path('<int:id>/', views.book\_detail, name='book\_detail'),

    path('add\_book/', views.add\_book, name='add\_book'),

    path('update\_book/<int:id>/', views.update\_book, name='update\_book'),

    path('delete\_book/<int:id>/', views.delete\_book, name='delete\_book'),

    # Les routes pour les codes

    path('codes/', views.code\_list, name='code\_list'),

    path('add\_code/', views.add\_code, name='add\_code'),

    path('code\_delete/<int:id>', views.code\_delete, name='code\_delete'),

    path('code\_update/<int:id>', views.code\_update, name='code\_update'),

]

Apres les vue dans **books/views.py :**

def add\_book(request):

    if request.method == 'POST':

        form = BookForm(request.POST)

        if form.is\_valid():

            form.save()  # Enregistre le livre en base de données

          # Redirige vers la liste des livres après ajout

            return redirect('books:book\_list')

    else:

        form = BookForm()

    return render(request, 'books/book\_form.html', {'form': form})

def delete\_book(request, id):

    book = get\_object\_or\_404(Book, id=id)

    book.delete()

    return redirect('books:book\_list')

def update\_book(request, id):

    book = get\_object\_or\_404(Book, id=id)

    if request.method == 'POST':

        # Liaison avec l'instance existante

        form = BookForm(request.POST, instance=book)

        if form.is\_valid():

            form.save()  # Met à jour l'auteur existant

            return redirect('books:book\_list')

    else:

         # Pré-remplissage pour modification

        form = BookForm(instance=book)

    return render(request, 'books/book\_form.html', {'form': form})

Le fichier book\_form.html doit être créer dans **books/templates/books/book\_form.html**

{% extends "base.html" %}

{% block content %}

<h2>Ajouter un Livre</h2>

<form method="POST">

    {% csrf\_token %}

    <!-- Affiche tous les champs du formulaire sous forme de paragraphes -->

    {{ form.as\_p }}

    <button type="submit">Ajouter</button>

</form>

{% endblock %}

Et metre a jour **books/templates/books/book\_list.html**  pour inclure les liens necessaire:

{% extends "base.html" %}

{% block title %}Liste des Livres{% endblock %}

{% block content %}

    <h2>Liste des Livres</h2>

    <ul>

        {% for book in books %}

            <li>

         <a href="{% url 'books:book\_detail' book.id %}">{{ book.title }}</a>

            {% comment %} Supprimer {% endcomment %}

<a style="color:red href="{% url 'books:delete\_book' book.id %}">Supprimer</a>

            <!-- Mettre à jour un auteur -->

<a style="color:green" href="{% url 'books:update\_book' book.id %}">Modifier</a>

           </li>

        {% endfor %}

    </ul>

    <a href="{% url 'books:add\_book' %}">Ajouter un livre</a>

{% endblock %}

Le crud des livres aussi est complet.

Continuons sur les Editions(Publishers)

Commençons par la class du formulaire dans **publishers/forms.py:**

from django import forms

from .models import Publisher

class PublisherForm(forms.ModelForm):

    class Meta:

        model = Publisher

        fields = ['name', 'address']

Après le **publishers/urls.py:**

from django.urls import path

from . import views

app\_name = 'publishers'

urlpatterns = [

    path('', views.publisher\_list, name='publisher\_list'),

    path('<int:id>/', views.publisher\_detail, name='publisher\_detail'),

path('delete\_publisher/<int:id>/', views.delete\_publisher, name='delete\_publisher'),

    path('update\_publisher/<int:id>/', views.update\_publisher, name='update\_publisher'),

    path('add\_publisher/', views.add\_publisher, name='add\_publisher'),

]

Après le fichier **publishers/ templates/ publishers/publisher\_form.html**

{% extends "base.html" %}

{% block content %}

<h2>Ajouter une edition</h2>

<form method="POST">

    {% csrf\_token %}

    <!-- Affiche tous les champs sous forme de paragraphes -->

    {{ form.as\_p }}

    <button type="submit">Ajouter l'edition</button>

</form>

{% endblock content %}

Après le vues :

def add\_publisher(request):

    if request.method == 'POST':

        form = PublisherForm(request.POST)

        if form.is\_valid():

            form.save()  # Enregistre le livre en base de données

          # Redirige vers la liste des publishers après ajout

            return redirect('publishers:publisher\_list')

    else:

        form = PublisherForm()

    return render(request, 'publishers/publisher\_form.html', {'form': form})

def delete\_publisher(request, id):

    publisher = get\_object\_or\_404(Publisher, id=id)

    publisher.delete()

    return redirect('publishers:publisher\_list')

def update\_publisher(request, id):

    publisher = get\_object\_or\_404(Publisher, id=id)

    if request.method == 'POST':

        # Liaison avec l'instance existante

        form = PublisherForm(request.POST, instance=publisher)

        if form.is\_valid():

            form.save()  # Met à jour l'edition existante

            return redirect('publishers:publisher\_list')

    else:

         # Pré-remplissage pour modification

        form = PublisherForm(instance=publisher)

    return render(request, 'publishers/publisher\_form.html', {'form': form})

Et mettre à jour **pulishers/templates/publishers/publisher\_list.html** pour inclure les liens

{% extends "base.html" %}

{% block title %}Liste des Éditeurs{% endblock %}

{% block content %}<h2>Liste des Éditeurs</h2>

    <ul>

        {% for publisher in publishers %}

            <li>

<a href="{% url 'publishers:publisher\_detail' publisher.id %}">{{ publisher.name }}</a>

<a style="color : red" href="{% url 'publishers:delete\_publisher' publisher.id %}">Supprimer</a>

 <a style="color : green" href="{% url 'publishers:update\_publisher' publisher.id %}">Modifier</a>

            </li>

        {% endfor %}

    </ul>

<a href="{% url 'publishers:add\_publisher' %}">Ajouter une edition</a>

{% endblock %}

En pratiquant tout ceci vous devriez comprendre le fonctionnement des formulaires en Django, merci d’approfondir vos compétences d’avantage.