APPRENTISSAGE DJANGO

Pour commencer un nouveau projet django :

* On créer le dossier de notre projet
* On se place dans le dossier
* On installe Django : **pipenv install django**

**Pipenv : Crée un environnement virtuel pour le projet**

**Un environnement virtuel est un endroit ou toutes les parties de notre projet sont stockées,**

**Donc avant de commencer un nouveau projet, nous devons toujours créer un environnement virtuel pour notre projet**

* On fait **code .** pour ouvrir le projet dans l’environnement virtuel
* On utilise ensuite la commande **pipenv shell** pour activer un environnement virtuel pour le projet
* Pour créer un nouveau projet, on utilise la commande : **django-admin startproject nom\_projet**
* On peut aussi utiliser la commande **django-admin startproject nom\_projet .** pour éviter que django créer un nouveau répertoire pour mettre le projet à l’intérieur
* On lance ensuite le serveur django avec la commande : **python manage.py runserver** (S’assurer que django est installé déjà)

1. Using the integrated terminal

VS Code a un interpréteur intégré, et nous pouvons l’activer :

* Aller dans le menu view (En haut) et choisir : Command palette
* Rechercher **Python interpreter** et mettre le chemin qui mène à notre environnement virtuel
* Pour connaitre le chemin, on tape la commande : **pipenv –venv**

1. WRITTING VIEWS

* Dans les fichiers views, on créer des fonctions de views (Ce sont des fonctions qui prennent en entrée une requête et qui retournent une réponse

Exemple de fichier **view** :

from django.shortcuts import render  
from django.http import HttpResponse  
  
  
# Create your views here.  
def sey\_hello(request):  
 return HttpResponse("Hello World")

Note\_\_\_ Il faut penser à inclure from django.http import HttpResponse

Après avoir créer une view, nous devons la relier à une URL (Uniform Ressource Locator)

1. MAPPING URL TO VIEWS

* On se place dans le répertoire de l’application concernée et on crée un fichier : **urls.py**
* Etant dans ce fichier là, introduit :
* from django.urls import path
* Inclure le fichier views.py qui se trouve dans le même répertoire (Application concernée) que le fichiers urls.py

from . import views

On fait ceci pour pouvoir relier la Route à la fonction views qui se trouve dans le fichier views

* On créer une liste nomée urlpatters(Par convention) : qui va contenir les urls liées aux fonctions views
* urlpatterns = \  
   [  
   path('playground/hello', views.sey\_hello)  
   ]
* Il faut ensuite aller dans le fichier urls.py de l’application principale pour faire le lien entre notre fichier urls et celui de l’application principale

Note\_\_\_

1. Pour inclure un nouvel URLconf, on importe d’abord la fonction include

* *from django.urls import include, path*

1. On ajouter notre url à **urlspatterns**
2. **TEMPLATES**

Note\_\_\_ Ce que nous appelons VIEWS dans les autres frameWork, c’est ça que nous appelons en django TEMPLATES…. Pour les utiliser nous allons :

* Créer un nouveau répertoire appelé template dans le répertoire de l’application
* Créer des fichiers HTML dans ce répertoire et on pourra ensuite les retourner grâce à la fonction render

Nouvelle structure du fichier views.py :

*from* django.shortcuts *import* render  
*from* django.http *import* HttpResponse  
  
*# Create your views here.  
  
  
def* say\_hello(request):  
 *return* render(request, 'hello.html')

Note\_\_\_ Nous pouvons ajouter un troisième paramètre à la fonction render (Le paramètre **context)** qui est de type Any (Donc il peut prendre n’importe quelle valeur)

Dans notre cas ici, nous allons lui passer un dictionnaire et l’afficher sur la page WEB

*def* say\_hello(request):  
 *return* render(request, 'hello.html', {'name': 'Joe Wilfred AFANA'})

Et pour afficher la Valeur dans la page HTML, nous utilisons les doubles accolades :

<!DOCTYPE *html*>  
<html *lang*="en">  
<head>  
 <meta *charset*="UTF-8">  
 <title>Title</title>  
</head>  
<body>  
<h1>Hello world</h1>  
<h3>Bienvenue {{ *name* }}</h3>  
</body>  
</html>

Note\_\_\_ Nous pouvons utiliser les balises de template (dans le HTML) pour poser des conditions (par exemple), Nous pouvons les utiliser par exemple pour vérifier si une variable a été fournie :

{% if *name* %}  
<h3>Bienvenue {{ *name* }}</h3>  
  
{% else %}  
<h1>Hello world</h1>  
  
{% endif %}

1. DJANGO DEBUG TOOL
2. BUIDING DATA MODEL

**DEUXIEME TUTORIEL:**

Pour installer l’environnement virtuel sur notre machine, on tape la commande :

**pip install virtualenvwrapper**

* **Pour créer un environnement virtuel pour notre projet, on tape la commande :**

**mkvirtualenv nom\_projet (On peut mettre autre chose)**

1. **ROUTING And Django Apps**

Une application **Django** est comme un sous site dans un site… Une application **Django** est constituée de plusieurs applications…

Par exemple Instagram est constitué de plusieurs sections

Chaque application a une chose particulière qu’elle fait

* C’est dans le fichier models.py que nous allons écrire notre base de données

1. **TEMPLATES IN DJANGO**

Pour ajouter des **templates**, on fait :

* On créer un nouveau répertoire que nous appelons **templates**
* **On va dans le fichier settings.py, au niveau de TEMPLATES , ou il y’a écrit : DIRS : on ajoute :**
* 'DIRS': [os.path.join(BASE\_DIR, 'templates')],

Ceci pour spécifier que c’est là que seront les templates de notre page

1. **SENDING DYNAMIC DATA TO MY TEMPLATES FILES**

Nous pouvons envoyer des données à nos templates (Comme des valeurs de variable par exemple) :

*def* index(request):  
 name = 'Joe'  
 *return* render(request, 'index.html', {'name': name})

Nous pouvons, dans le HTML accéder à la variable **name** en utilisant le nom de sa clé (**name** ici) et en ajoutant les doubles accolades

<h1>How do you do today {{name}} ?  
</h1>

* Vu que le troisième paramètre de la fonction **render ()** est un dictionnaire, nous pouvons directement créer un dictionnaire qui va prendre les informations de l’utilisateur dans notre fonction index et le retourner :
* *def* index(request):  
   context = \  
   {  
   'name': 'Joe',  
   'age': 25,  
   'city': 'New York',  
   'nationality': 'Cameroonian',  
   }  
   *return* render(request, 'index.html', context=context)

1. **BUILDING A WORD COUNTER IN DJANGO**

Nous voulons créer ici une petite appli avec un champs de type **textarea** ou l’utilisateur pourra entrer une phrase et nous allons afficher le nombre de mots de la phrase dans un autre fichier

* Nous allons dans le fichier urls.py de notre app pour y ajouter le lien vers notre nouveau fichier :
* urlpatterns = \  
   [  
   path('', views.index, name='index'),  
   path('counter', views.counter, name='counter'),  
   ]

On crée un formulaire HTML et dans **action** nous allons mettre le nom du fichier dans lequel nous voulons envoyer les données récupérées :

<!doctype *html*>  
<html *lang*="en">  
<head>  
 <meta *charset*="UTF-8">  
 <meta *name*="viewport"  
 *content*="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">  
 <meta *http-equiv*="X-UA-Compatible" *content*="ie=edge">  
 <title>Form</title>  
</head>  
<body>  
 <form *method*="" *action*="counter">  
 <label>  
 <textarea *name*="text" *cols*="60" *rows*="10"></textarea>  
 </label>  
 <br>  
 <input *type*="submit" *value*="Submit">  
 </form>  
</body>  
</html>

NB : En Django, dans la l’attribut **ACTION** de la balise **form**, on n’a pas besoin de mettre l’index du fichier html… On met juste le nom du fichier

* On va dans le fichier **views.py** de notre app pour implémenter le fonctionnement de notre page :
* Nous allons créer une nouvelle fonction **counter()** qui va permettre de récupérer l’entrée utilisateur et de compter le nombre de mots… Nous allons ensuite renvoyer le résultat dans la page counter.html :
* *def* counter(request):  
   text = request.GET['text']  
   number = *len*(text.split())  
   result = \  
   {  
   'number\_of\_words': number  
   }  
   *return* render(request, 'counter.html', result)
* Nous récupérons ensuite le résultat dans le fichier counter.html :
* <!DOCTYPE *html*>  
  <html *lang*="en">  
  <head>  
   <meta *charset*="UTF-8">  
   <title>Number of words</title>  
  </head>  
  <body>  
   <h1>Here is the number of words in your project: </h1><br>  
   <b>{{number\_of\_words}}</b> words  
  </body>  
  </html>

**V- GET AND POST REQUEST**

* **POST METHOD**

Quand nous utilisons la méthode POST, nous devons penser à éviter les attaques CSRF

Il faut penser à mettre une ligne de code dans notre formulaire pour ça :

<form *method*="POST" *action*="counter">  
 {% csrf\_token %}

La méthode POST est beaucoup plus appropriée lorsque nous voulons protéger les données des utilisateurs lors de l’envoie d’un formulaire

1. **STATIC FILE IN DJANGO**

Quand on parle de fichier statiques, on fait allusion aux fichiers externes que nous allons utiliser dans notre **template Django**

Tous les fichiers que nous allons utiliser (CSS,JS,etc) doivent être stockés dans un nouveau répertoire que nous allons créer et qu’on va appeler : **STATIC**

Après avoir créer le répertoire, nous allons dans le fichier settings.py :

* **On importe la bibliothèque os**
* **Juste en dessous de STATIC\_URL, on ajoute la ligne suivante :**
* STATICFILES\_DIRS = (os.path.join(BASE\_DIR, 'static'),)

* Nous pouvons maintenant mettre un fichier CSS dans le répertoire STATIC
* Pour lier le fichier CSS à notre/nos templates HTML, il faut penser à charger le répertoire STATIC (En haut du fichier HTML) :

{% load static %}  
 <link *rel*="stylesheet" *href*="{% static 'style.css' %}">  
<h1>Hi ! Welcome to my first Django web site</h1>

1. **INTRODUCTION TO DJANGO MODELS**

**Bien sûr ! Les modèles (models) dans Django représentent la structure de votre base de données. Ils définissent les champs et les comportements des données que vous stockez. Chaque modèle est une classe Python qui hérite de django.db.models.Model, et les attributs de la classe représentent les champs de la base de données.**

**Pourquoi utiliser des modèles ?**

**Les modèles permettent de :**

1. **Définir les données : Vous définissez les types de données que vous allez stocker (texte, nombre, date, etc.).**
2. **Créer des tables dans la base de données : Django crée automatiquement les tables en fonction des modèles définis.**
3. **Interagir avec la base de données : Vous pouvez ajouter, modifier, supprimer et interroger des données.**

**Structure de base d'un modèle**

**Voici un exemple simple d'un modèle représentant un livre dans une bibliothèque :**

**python**

**Copier le code**

**from django.db import models class Book(models.Model): title = models.CharField(max\_length=100) author = models.CharField(max\_length=50) published\_date = models.DateField() isbn = models.CharField(max\_length=13, unique=True) pages = models.IntegerField() cover = models.CharField(max\_length=100, choices=[('H', 'Hardcover'), ('P', 'Paperback')]) available = models.BooleanField(default=True)**

**Explication des champs**

* **title: Un champ de type CharField pour le titre du livre, avec une longueur maximale de 100 caractères.**
* **author: Un champ de type CharField pour l'auteur du livre, avec une longueur maximale de 50 caractères.**
* **published\_date: Un champ de type DateField pour la date de publication du livre.**
* **isbn: Un champ de type CharField pour le numéro ISBN du livre, avec une longueur maximale de 13 caractères et une contrainte d'unicité.**
* **pages: Un champ de type IntegerField pour le nombre de pages du livre.**
* **cover: Un champ de type CharField avec des choix prédéfinis pour la couverture du livre (couverture rigide ou souple).**
* **available: Un champ de type BooleanField pour indiquer si le livre est disponible, avec une valeur par défaut de True.**

**Note\_\_\_** Si nous avons plusieurs éléments dans notre vue qui doivent afficher des valeurs venants du même modèle, au lieu de le réécrire plusieurs fois, nous pouvons faire ceci :

* **Dans le models.py, on créer le modèle de notre objet :**
* *class* Feature:  
   name: *str* price: *float* description: *str*
* *Dans le views.py, on créer une fonction dans laquelle on va créer de nouvelles instances de notre modèle :*

***def* index(request):  
 feature1 = Feature()  
 feature1.name = 'Product One'  
 feature1.description = 'I\'m the first feature'  
 feature1.price = 23000  
  
 feature2 = Feature()  
 feature2.name = 'Product Two'  
 feature2.description = 'I\'m the second feature'  
 feature2.price = 58000  
  
 feature3 = Feature()  
 feature3.name = 'Product Three'  
 feature3.description = 'I\'m the third feature'  
 feature3.price = 25000  
  
 features = [feature1, feature2, feature3]  
 *return* render(request, 'index.html', {'features': features})**

* **On a créé ici 4 instances de notre modèle feature (Pour les 4 box qui seront affichés sur la page )**
* **On créer une liste dans laquelle on va garder les quatre instances**
* **On retourne notre tableau d’instances sous forme de clé-valeur d’un dictionnaire**
* **Dans le template (concerné), on se place juste au début de notre box et on ouvre une boucle for qui va permettre de parcourir notre liste :**
* {% for feature in features %}  
   <div *class*="card-body">  
    
   <div *class*="row no-gutters align-items-center">  
   <div *class*="col mr-2">  
   <div *class*="text-xs font-weight-bold text-primary text-uppercase mb-1">  
   {{feature.name}}</div>  
   <div *class*="h5 mb-0 font-weight-bold text-gray-800">${{feature.price}}</div>  
   </div>  
   <div *class*="col-auto">  
   <i *class*="fas fa-calendar fa-2x text-gray-300"></i>  
   </div>  
   </div>  
   </div>  
    
  </div>  
  {% endfor %}

NB : Il faut penser à mettre un **endfor** à la fin de notre box (Celui que nous voulons afficher un certain nombre de fois)

Note\_\_\_ Dans les balises Django en HTML, nous pouvons aussi utiliser les structures conditionelles IF pour vérifier si une condition est vraie et afficher quelque chose en conséquence

<div *class*="h5 mb-0 font-weight-bold text-gray-800">${{feature.price}}</div>  
 {% if feature.is\_true == True %}  
 <p>This feature is true</p>  
 {% endif %}

1. **Django Admin Panel & Manipulation Of Data**

Pour changer une classe définie dans le models.py en une table dans la base de données, il faut ajouter des parenthèses devant le nom de la classe et mettre dans les parenthèses : **models.Modes**

* **MIGRATIONS**

Les migrations nous permettent de créer ou de modifier les structures des tables au sein d’une base de données

Toutes ces modifications sont faites à l’aide de l’ORM de Django

* On utilise la commande **python manage.py makemigrations** pour créer les migrations
* L’interface Django Admin est en faite une interface utilisateur ***CRUD (Create Read Update and Delete)*** qui permet aux administrateurs de site de créer, lire, mettre à jour et supprimer des données directement depuis une interface utilisateur conviviale. Vous pouvez gérer toutes les données de vos modèles sans avoir à écrire de code supplémentaire.
* Nous devons ensuite inclure des modèles membre :
* Pour inclure le modèle Member dans l'interface d'administration, nous devons indiquer à Django que ce modèle doit être visible dans l'interface d'administration.
* Cela se fait dans un fichier appelé admin.py, et se trouve dans le dossier de votre application,
* Insérez quelques lignes ici pour rendre le modèle de membre visible dans la page d'administration :

my\_tennis\_club/members/admin.py:

from django.contrib import admin

from .models import Member

# Register your models here.

admin.site.register(Member)

Pour accéder au panel, on fait :

* [**http://127.0.0.1:8000/admin/**](http://127.0.0.1:8000/admin/)
* Pour accéder à l’interface du panel, il faudra s’identifier (S’authentifier) et pour ça, nous allons créer un super utilisateur grâce à la commande suivante :
* **python manage.py createsuperuser**

Les migrations sont la manière par laquelle Django propage des modifications que vous apportez à des modèles (ajout d’un champ, suppression d’un modèle, etc.) dans un schéma de base de données. Elles sont conçues pour être quasiment automatiques, mais vous aurez besoin de savoir quand créer les migrations, quand les exécuter, et les problèmes courants que vous pourriez rencontrer.

Les commandes

***Il y a plusieurs commandes utiles pour interagir avec les migrations et manipuler le schéma de base de données avec Django :***

*Note\_\_Importante :*

*Nous pouvons ensuite accéder facilement à tous les éléments gardés dans notre base de données et les récupérer comme étant une liste :*

*def* show\_posts(request):  
 number = 0  
 posts = Posts.objects.all()  
  
 *return* render(request, 'show\_posts.html', {'posts': posts, 'number': number})

* [**migrate**](https://docs.djangoproject.com/fr/5.0/ref/django-admin/#django-admin-migrate), qui est responsable de l’exécution et de l’annulation des migrations.
* [**makemigrations**](https://docs.djangoproject.com/fr/5.0/ref/django-admin/#django-admin-makemigrations), qui est responsable de la création de nouvelles migrations en fonction des modifications que vous avez apportées aux modèles.
* [**sqlmigrate**](https://docs.djangoproject.com/fr/5.0/ref/django-admin/#django-admin-sqlmigrate), qui affiche les instructions SQL correspondant à une migration.
* [**showmigrations**](https://docs.djangoproject.com/fr/5.0/ref/django-admin/#django-admin-showmigrations), qui affiche la liste des migrations d’un projet ainsi que leur état.

Vous devez imaginer les migrations comme un système de contrôle de versions pour un schéma de base de données. **makemigrations** se charge de regrouper vos changements de modèle dans des fichiers de migration individuels - comme des commits - et **migrate** est chargé d’appliquer les changements à la base de données.

Note\_\_\_ Chaque fois que nous

1. **USER REGISTRATION/AUTHENTIFICATION IN DJANGO**

Nous allons apprendre ici à gérer l’enregistrement des utilisateurs sur notre site web :

* Nous allons tout d’abord créer un formulaire simple pour récupérer les informations de l’utilisateur qui veut s’inscrire

Note\_\_\_ La fonction **redirect** dans Django est utilisée pour rediriger l'utilisateur vers une autre URL.

Elle peut parfois prendre des paramètres si nous voulons que la vue renvoie des données supplémentaires au template

*def* my\_view(request):  
 *# Traitement de la vue  
 return* redirect('nom\_de\_l\_url', argument1, argument2)

Bien sûr ! Voici une explication simple de ces deux lignes :

* 1. from django.contrib.auth.models import User, auth`

- User:

- C'est un modèle intégré de Django qui représente un utilisateur dans votre application. Il contient des champs comme le nom d'utilisateur, le mot de passe, l'email, etc.

- Vous utilisez `User` pour créer, modifier et gérer les utilisateurs de votre application.

- auth:

- C'est un module qui fournit des fonctions et des outils pour gérer l'authentification des utilisateurs.

- Par exemple, `**auth**` inclut des fonctions comme `**login()**`, `**logout()**`, et `**authenticate()**` qui sont utilisées pour connecter, déconnecter et vérifier les utilisateurs.

* 1. from django.contrib import messages`

- \*\*`messages`\*\* :

- C'est un Framework de Django qui vous permet de stocker et de récupérer des messages temporaires dans votre application web.

- Ces messages sont souvent utilisés pour afficher des notifications aux utilisateurs, comme des messages de succès, d'erreur, ou d'information.

- Par exemple, après qu'un utilisateur soumette un formulaire, vous pouvez utiliser `messages` pour afficher un message de confirmation ou une alerte d'erreur.

### Exemple d'utilisation

Voici un exemple concret où ces deux importations sont utilisées dans une vue pour gérer la connexion d'un utilisateur et afficher des messages :

*from* django.shortcuts *import* render, redirect  
*from* django.contrib.auth.models *import* User, auth  
*from* django.contrib *import* messages  
  
*def* login\_view(request):  
 *if* request.method == 'POST':  
 username = request.POST['username']  
 password = request.POST['password']  
 user = auth.authenticate(username=username, password=password)  
 *if* user *is not None*:  
 auth.login(request, user)  
 messages.success(request, 'You have successfully logged in!')  
 *return* redirect('profile')  
 *else*:  
 messages.error(request, 'Invalid credentials, please try again.')  
 *return* redirect('login')  
 *else*:  
 *return* render(request, 'login.html')

Dans cet exemple :

- `auth.authenticate` vérifie les informations de connexion de l'utilisateur.

- `auth.login` connecte l'utilisateur si les informations sont correctes.

- `messages.success` affiche un message de succès après une connexion réussie.

- `messages.error` affiche un message d'erreur si la connexion échoue.

1. **USER LogIn & LogOut in DJANGO**

**Nous allons apprendre ici à gérer la connexion et la déconnexion des utilisateurs sur notre site web**

* 1. **LOGIN**
* **Nous créons une vue (Une page HTML) et une fonction pour la connexion (Login)**
* **Nous créons une nouvelle URL pour le login**
* path('login', views.login\_, name='login')
* Nous créons une nouvelle fonction pour le login

*def* login(request):  
 *return* render(request, 'login.html')

* Nous créons le fichier HTML pour le login (Fichier dans lequel il y’aura le formulaire pour que l’utilisateur puisse mettre ses informations de connexion)
* Dans la fonction, nous récupérons les données reçues du formulaire grâce à la méthode POST
* *def* login(request):  
   *if* request.method == 'POST':  
   username = request.POST['username']  
   password = request.POST['password']  
    
   user = auth.authenticate(username=username, password=password)  
   *return* render(request, 'login.html')

Note\_\_\_:

**auth.authenticate**

La fonction **auth.authenticate** de Django est utilisée pour vérifier les informations d'identification fournies par un utilisateur (comme le nom d'utilisateur et le mot de passe).

**Comment ça marche**

* **username=username** : Cela signifie que vous passez le nom d'utilisateur saisi par l'utilisateur à la fonction **authenticate**.
* **password=password** : Cela signifie que vous passez le mot de passe saisi par l'utilisateur à la fonction **authenticate**.

**Ce que fait cette ligne**

* **Validation des informations** : **auth.authenticate** prend le nom d'utilisateur et le mot de passe que l'utilisateur a saisis et vérifie si ces informations correspondent à un utilisateur existant dans la base de données.
* **Retourne un objet utilisateur ou None** :
  + Si les informations sont correctes, **auth.authenticate** retourne un objet **User** représentant l'utilisateur.
  + Si les informations sont incorrectes (par exemple, le nom d'utilisateur ou le mot de passe ne correspond pas), la fonction retourne **None**.
* Nous vérifions ensuite si les données entrées par l’utilisateur figurent dans notre base de données

(Pour ça, il faut vérifier si la variable user est None, si c’est le cas alors l’utilisateur n’est pas dans la base de données.

*if* user *is not None*:  
 auth.login(request, user)  
 *return* redirect('register')

Explications\_\_\_: Si la variable **user** n’est pas None, ça veut dire que l’utilisateur existe bien dans la base de données…

La ligne :

auth.login(request, user)

permet de connecter l’utilisateur

Enfin, on redirige l’utilisateur vers sa page de profil (Facultatif)

Nous pouvons configurer le même template pour plusieurs cas et selon certaines conditions.

Par exemple on peut adapter le template en fonction que l’utilisateur soit connecter ou non : {% if *user*.*is\_authenticated* %}  
<h1 *class*="display-3 fw-bolder mb-5"><span *class*="text-gradient d-inline">Welcome {{ *user*.*username* }} you'r online</span></h1>  
{% else %}  
 <h1 *class*="display-3 fw-bolder mb-5"><span *class*="text-gradient d-inline">Welcome, You'r not online</span></h1>  
 {% endif %}

**user.authentificated** permet de vérifier si l’utilisateur a été authentifier et donc on va agir en conséquence dans le template

LOGOUT

1. Dynamic URL Routing in Django