TRAVAUX DIRIGÉS
API REST EN PYTHON

I. INTRODUCTION

le but de ce TD est de créer votre première API REST en python à l'aide du framewok REST de django (https://www.django-rest-framework.org).

II. PROCÉDURE D'INSTALLATION

Pour travailler sur les machines virtuelles débian il va falloir installer :

(1) python3 et pip, ou au moins vérifier qu'ils soient présent tous les deux.

(2) un environnement de développement, ce qui nous permettra d'avoir plusieurs fichiers édités en parallèle. Visual studio code peut faire l'affaire et permettra aussi de connecter un dépot GitHub pour versionner votre code, de même que Atom ou Geany. À vous de choisr votre éditeur.

Les procédures d'installation des logiciels sont disponibles sur https ://code.visualstudio.com/docs/setup/linux pour VSCode (bien choisir débian), https://www.linuxbuzz.com/install-atom-text-editor-in-debian/ pour Atom, et simplement un **sudo apt install geany** pour geany

une fois ces deux éléments installés, nous allons configurer notre environnement python virtuel pour développer notre API pour cela, vous allez exécuter les commandes suivantes dans un terminal sous un compte qui ne soit pas root (En terme de sécurité, développer en tant que root, créer des applications avec les droits root, ce qui n'est jamais conseillé)

Code 1: configuration de l'environnement virtual python

```
mkdir project && cd project
python3 -m venv venv
source venv/bin/activate
```

La première permet d'embarquer toute les librairies et exécutables python dans un répertoire nommé venv (le 2nd de la ligne de commande). La deuxième ligne active l'environnement

Une fois cela fait, nous allons pouvoir installer les modules django et djangorestframework.

Code 2: installation des modules django et djangorestframework

```
pip install django
pip install djangorestframework
```

III. CRÉATION DU PROJET ET DE L'API

Comme Pour Django, nous allons créer une nouveau projet puis une application. Pour rappel, la création d'un projet et d'une nouvelle application passe par les commandes suivantes

Code 3: Création d'un projet Django et d'une application

```
django-admin startproject monprojet
cd monprojet
python3 manage.py startapp monapi
```

Vous avez à présent l'arborescence de votre projet qui est créée. Il faut ajouter rest_framework et monapi dans la liste des applications de votre projet (mettre à jour le fichier *settings.py*. Créer aussi le fichier urls.py dans votre application et faire le lien dans le fichier urls.py du projet.

Code 4: modification du fichier urls.py du projet

```
from django.contrib import admin
from django.urls import path, include


urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
    path('monapi/', include("monapi.urls")),
]
```

Pour finir la configuration vous allez faire la migration initiale des tables de la base de données liées aux modèle

```
python3 manage.py migrate
```

puis créer un super utilisateur.

```
python3 manage.py createsuperuser
```

IV. Votre première API

le principe des API REST consiste à associer les types de requêtes http à des éléments d'un CRUD sur un modèle de données comme illustré par le tableau 1. Le format de retour choisi sera du Json.

élément CRUD	route	requête HTTP
All	http ://127.0.0.1 :8000/monapi/api/	GET
Create	http ://127.0.0.1 :8000/monapi/api/	POST
Read	http ://127.0.0.1 :8000/monapi/api/<id>/	GET
Update	http ://127.0.0.1 :8000/monapi/api/<id>/	PUT
Delete	http ://127.0.0.1 :8000/monapi/api/<id>/	DELETE

Table 1 – relation entre élément du CRUD et type de requête http

V. Modèle et Serializer

V.1. **modèle.** Comme dans les projets Django de première année, nous allons créer un modèle de données. Dans le fichier *models.py* vous aller ajouter la classe **Commentaire** héritant de la classe **Model** du package *model* (à importer depuis *django.db*). Les champs de cette classe seront :

— le titre ;
— le commentaire ;
— la date de publication. Ce champs est un champs DateFields avec le paramètre auto_date = True ;

Vous ajouterez une méthode **__str__** et **__repr__**. Vous penserez aussi aux commandes makemigrations et migrate pour propager votre modèle dans la base de données.

V.2. **serializer.** Dans ce projet nous n'allons pas utiliser de formulaire pour lequel il faut travailler sur un mapping entre les champs de saisie et le champs du modèle, nous allons utiliser un format Json pour échanger des données. il va donc falloir avoir une conversion entre un format JSON et un objet **Commentaire**. C'est le rôle du Serializer. Pour cela vous aller créer un fichier *serializer.py*

Code 5: fichier *Serializer.py*

```
from rest_framework import serializers
from .models import Commentaire
class CommentaireSerializer(serializers.ModelSerializer):
    class Meta:
        model = Commentaire
        fields = ["titre", "commentaire", "date\_publication"]
```

Comme le formulaire, cela définit les champs attendus et leur conversion.

VI. LE CONTRÔLEUR *VIEWS.PY*

Nous allons travailler ici avec des classes plutôt qu'avec des fonctions simple afin de faire la correspondance entre les éléments du CRUD et les méthodes d'appel. Nous allons définir 2 classes, la classe générale permettant d'afficher la liste des éléments avec ou sans filtre de sélection et l'insertion d'un nouveau Commentaire et une classe de détail pour manipuler un objet en particulier.

VI.1. **classe générale.** la classe générale posséde deux méthodes, une méthode **GET** permettant de récupérer un ensemble de données et une méthode **POST** permettant d'inserer un nouveau commentaire

Code 6: classe CommentaireListApiView

```
1  ffrom rest_framework.views import APIView
2  from rest_framework.response import Response
3  from rest_framework import status
4  from .models import Commentaire
5  from .serializers import CommentaireSerializer
6
7  class CommentaireListApiView(APIView):
8
9       def get(self, request, *args, **kwargs):
10        commentaires = Commentaire.objects.filter(user = request.user.id)
11        serializer = CommentaireSerializer(commentaires, many=True)
12        return Response(serializer.data, status=status.HTTP_200_OK)
13
14
15    def post(self, request, *args, **kwargs):
16        data = {
17            'titre': request.data.get('titre'),
18            'commentaire': request.data.get('commentaire'),
19        }
20        serializer = CommentaireSerializer(data=data)
21        if serializer.is_valid():
22            serializer.save()
23            return Response(serializer.data, status=status.HTTP_201_CREATED)
24
25        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

On peux voir quel le nom des méthodes correspondent aux type de requêtes HTTP associées, GET et POST Les différents statuts de retour HTTP peuvent être trouvé sur

https://www.django-rest-framework.org/api-guide/status-codes/

.

Pour déployer ces méthodes aux travers des routes, il faut modifier le fichier urls.py de l'application afin de déployer les routes

Code 7: fichier *urls.py*

```
1  from django.conf.urls import url
2  from django.urls import path, include
3  from .views import CommentaireListApiView
4
5  urlpatterns = [
6      path('api/', CommentaireListApiView.as_view()),
7  ]
```

VI.2. **interaction avec le serveur.** Pour interagir avec le serveur, vous allez utiliser dans un premier temps les commande **curl**. Pour récupérer l'ensemble des commentaires insérés, il faut faire une requête GET sur la route définie.

```
3  curl http://127.0.0.1:8000/monapi/api/
```

par défaut la commande curl effectue un GET. Pour insérer un commentaire, il faut effectuer une requête POST et transmettre un lot de données.

```
curl -d '{donnees au format JSON}' -H 'Content-Type:application/json' \
http://127.0.0.1:8000/monapi/api/
```

l'option -d permet d'envoyer des données sous format chaine de caractère. On peut aussi créer un fichier JSON tel que requete.json et le passer à la commande

```
curl -d @requete.json -H 'Content-Type : application./json' \
http://127.0.0.1:8000/monapi/api/
```

Une fois maitrisé vous pouvez aussi utiliser le site web mis à disposition par le framework pour tester votre API. il suffit de vous connecter sur **http ://127.0.0.1 :8000/monapi/api/**

VI.3. **classe de détail.** Dans cette second classe, nous allons nous occuper d'un objet en particulier à l'aide de son ID. Pour rappel lors de la création de la table associée au modèle, un champ id est automatiquement crée au format numérique auto-incrémenté pour servir de clé primaire de la table. La route à ajouter est de la forme

```
path('api/<int:id>/', CommentaireDetailApiView.as_view()),
```

où **CommentaireDetailApiView** est la seconde classe de contrôle. Vous voyez aussi le second paramètre dans l'URI qui correspond à l'id de l'objet.

Code 8: classe de contrôleur : récupérer un commentaire

```python
class CommentaireDetailApiView(APIView):

    def get(self, request, id, *args, **kwargs):
        commentaire = Commentaire.objects.get(pk=id) # (ou id=id qui fonctionne aussi)
        if not Commentaire:
            return Response({"res": "Object with id does not exists"},
                status=status.HTTP_400_BAD_REQUEST
            )

        serializer = CommentaireSerializer(commentaire)
        return Response(serializer.data, status=status.HTTP_200_OK)
```

Cette méthode **GET** permet de récupérer un objet à l'aide de son ID et d'en afficher les données. Pour effacer un commentaire nous utilisons la requête **DELETE**

Code 9: classe de contrôleur : effacer un commentaire

```python
class CommentaireDetailApiView(APIView):

....

    def delete(self, request, id, *args, **kwargs):

        commentaire = Commentaire.object.get(pk=id) # (ou id=id qui fonctionne aussi)

        if not commentaire:
            return Response(
                {"res": "Object with id does not exists"},
    status=status.HTTP_400_BAD_REQUEST)
        commentaire.delete()
        return Response({"res": "Object deleted!"},  status=status.HTTP_200_OK)
```

Enfin, le dernier type de requête utilisée est l'**PUT** qui permet de mettre à jour un commentaire existant.

Code 10: classe de contrôleur : Mettre à jour un commentaire

```
class CommentaireDetailApiView(APIView):

...

 def put(self, request, id, *args, **kwargs):

       commentaire = Commentaire.object.get(pk=id) # (ou id=id qui fonctionne aussi)
       if not commentaire:
            return Response(
                {"res": "Object with id does not exists"},
                status=status.HTTP_400_BAD_REQUEST
            )
        data = {
            'titre': request.data.get('titre'),
            'commentaire': request.data.get('commentaire'),
        }
        serializer = CommentaireSerializer(instance = commentaire, data=data, partial =
    True)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_200_OK)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

VII. exécution des requêtes HTTP PUT et DELETE

Pour exécuter avec curl des requêtes **PUT** et **DELETE**, vous utiliserez

```
curl -X PUT -d @requete.json -H 'Content-Type : application./json' \
http://127.0.0.1:8000/monapi/api/2/
```

et

```
curl -X DELETE http://127.0.0.1:8000/monapi/api/2/
```

VIII. authentification

pour forcer l'usage d'une authentification pour utiliser l'API, vous devez rajouter dans chacune des classes de contrôleur, au début de la classe la ligne

```
from rest_framework import permissions # ajouter pour l'authentification
class CommentaireDetailApiView(APIView):
    permission_classes = [permissions.IsAuthenticated]
    ....
```

dans ce cas, les commandes curl devront utiliser l'option –user

```
curl -X DELETE --user login:password http://127.0.0.1:8000/monapi/api/2/
```

IX. usage d'autres outils pour interroger l'API

IX.1. **application rest_framework.** Pour interroger l'API au travers du navigateur, le framework rest django embarque une application le permettant. Il faut ajouter dans la liste des applications l'application 'rest_framework'. Ensuite, il faut ajouter la route

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
    path('monapi/',include('monapi.urls')),
]
```

dans le fichier urls.py de votre projet. vous pourrez enfin utiliser le navigateur pour exécuter une requête comme illustré sur la figure 1

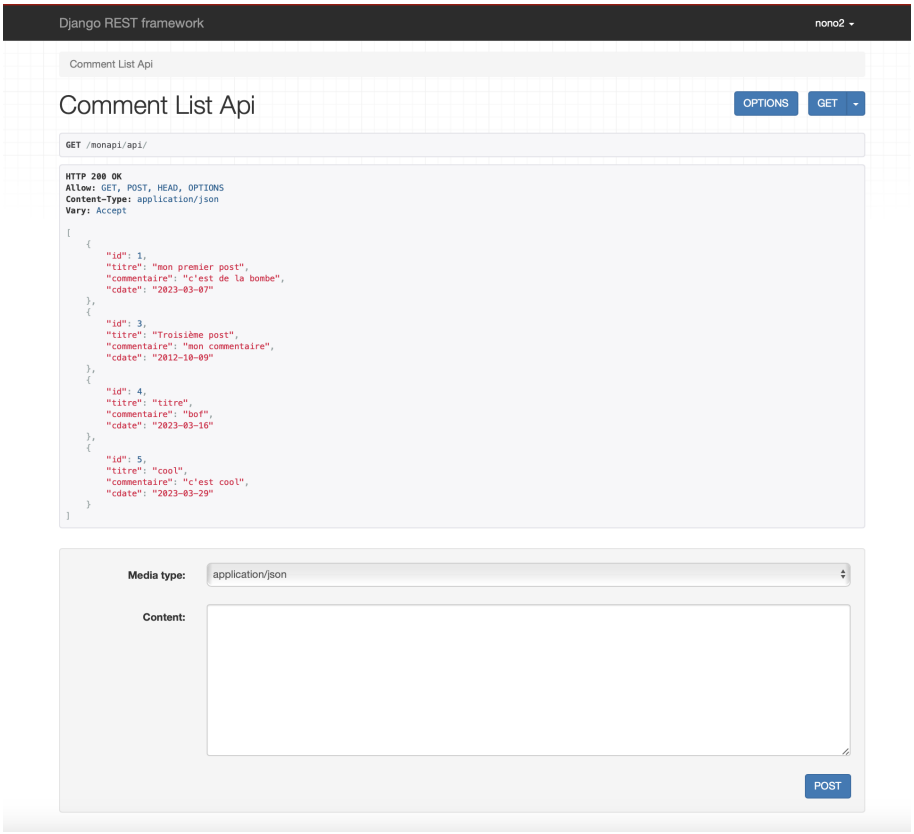

FIGURE 1 – accès API dans un navigateur avec Django Rest

IX.2. **application postman.** L'application postman (téléchargeable sur le site de postman https://www.postman.com/downloads/). Il faut créer un compte gratuit pour l'utiliser.

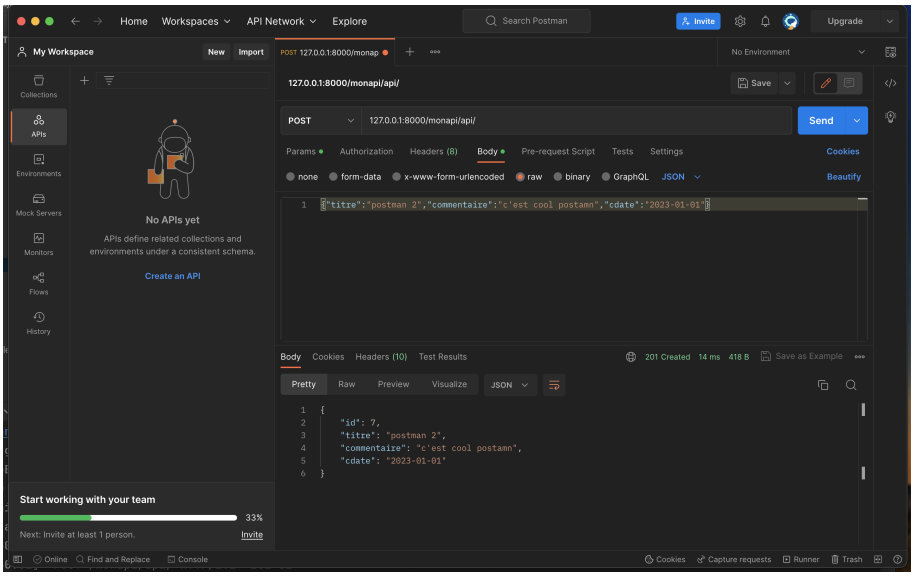

FIGURE 2 – accès API dans l'application postman

X. CONCLUSION

L'ensemble de ces méthodes permettent de définir une API REST permettant d'interroger une base de données et d'interagir avec.

XI. Exercice

Créer une API permettant de gérer un service de client. un client se définit avec son genre, nom, prénom, identifiant, mot de passe, adresse, mail et téléphone. Les valeurs obligatoire sont le nom, identifiant, mot de passe et adresse mail. l'identifiant doit être unique. vous vérifierez cela avant l'insertion d'un nouveau clients

La méthode **POST** de la classe de détail, permettra d'authentifier le client en renvoyant une valeur booléenne comme résultat. De même, la méthode **GET** ne doit pas afficher l'id ou le mot de passe.