esprit Se former autrement	Année Universitaire : 2022-2023 **Workshops Framework Django 4.1** **UP-Web**

Workshop : Django Rest Framework

Objectif

L'objectif de ce workshop est de manipuler JSON côté serveur dans une application Django4.1.

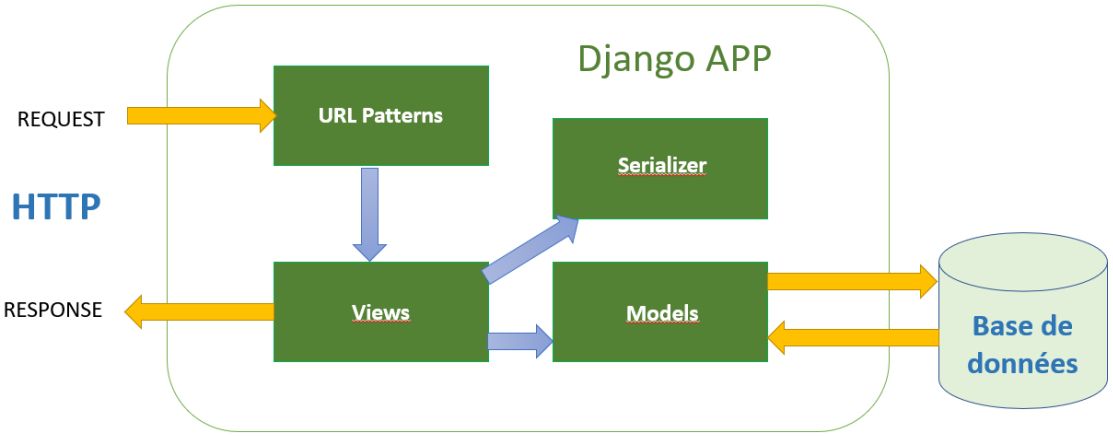


Figure 1 : Architecture de DRF

I- Le principe de sérialisation

- JSON est un format de données léger, facile à lire et à écrire et compatible avec pas mal de langages de développement.

- Le framework Python Django nous offre un composant **Serializer** pour sérialiser les objets en différents formats (json, xml ...).

- La classe Serializer est une classe fournie par Django Rest Framework qui permet de sérialiser et de désérialiser les données d'un modèle Django en JSON, XML ou tout autre format de données.

- En d'autres termes, les Serializer sont des classes qui permettent de convertir des instances de modèles Django en représentations Python pouvant être facilement transformées en JSON, XML ou d'autres formats de données. Les Serializer sont

également utilisés pour valider les données entrantes dans les vues API de Django Rest Framework.

II- CRUD de la classe model « Event » avec DRF :

1. Initialisation

- Soit le dossier « **api** » qui contient 3 fichiers :
  - ✓ Serializers.py
  - ✓ Urls.py
  - ✓ Views.py

- Taper la commande « pip install djangorestframework » pour installer le module Django REST Framework (DRF) via le gestionnaire de packages Python "pip".
- Ajouter "rest_framework" dans la liste des applications installées dans le fichier "settings.py" via la variable "INSTALLED_APPS".

2. La sérialisation :

Cette classe définit un serializer pour le modèle Event au niveau du fichier Serializers.py

```
class EventSerializer(serializers.ModelSerializer):
    class Meta:
        model = Event
        fields = '__all__'
```

- EventSerializer est définie en tant que sous-classe de serializers.ModelSerializer de DRF, qui fournit des fonctionnalités de sérialisation de base pour les modèles Django.
- La classe Meta définit les métadonnées pour le serializer, notamment le modèle sur lequel le serializer doit travailler et les champs que le serializer doit inclure dans la sortie JSON.
- En utilisant fields = '__all__', le serializer inclura tous les champs du modèle dans la sortie JSON. Cela peut également être remplacé par une liste de noms de champs pour spécifier les champs à inclure dans la sortie. Par exemple :

```
class EventSerializer(serializers.ModelSerializer):
    class Meta:
        model = Event
        fields = ['title','category']
```

3. Liste des évènements

La fonction « getEvents() » permet d'afficher la liste des évènements.

```python
@api_view(['GET'])
def getEvents(request):
      events = Event.objects.all()
      serializer = EventSerializer(events , many=True)
      return Response( serializer.data ,status=status.HTTP_200_OK )
```

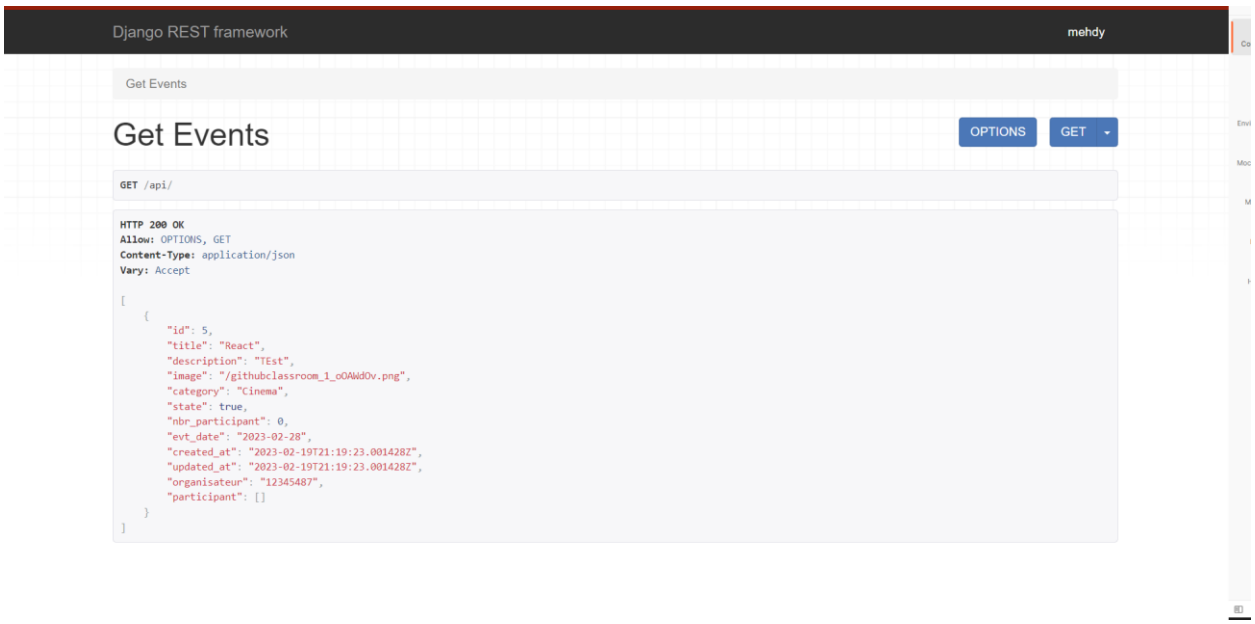
Cette fonction getEvents est une vue basée sur une fonction Django REST Framework qui renvoie tous les événements (instances du modèle Event) enregistrés dans la base de données en tant que réponse JSON.

- L'annotation @api_view(['GET']) signifie que cette vue répond uniquement aux requêtes HTTP GET. Si la vue devait également accepter les requêtes POST, PUT, DELETE ou d'autres méthodes HTTP, il faudrait spécifier ces méthodes dans la liste des méthodes HTTP autorisées.
- La vue récupère tous les événements enregistrés en base de données en utilisant Event.objects.all(), puis crée un serializer EventSerializer pour convertir ces événements en JSON.
- L'argument many=True est utilisé ici pour spécifier que plusieurs instances de modèle doivent être sérialisées.
- Enfin, la vue renvoie une réponse JSON contenant les données sérialisées, en utilisant Response(serializer.data) et le code de réponse HTTP est 200 OK.

- N'oubliez pas d'ajouter les importations des classes utilisées.

```python
from rest_framework.response import Response
from rest_framework import status
from rest_framework.decorators import api_view
from Event.models import Event
from .serializers import EventSerializer
```

En tapant le lien suivant http://127.0.0.1:8000/api/ , une liste des évènements sera affichée Avec l'interface par défaut fournie par DRF sinon on peut utiliser Postman.



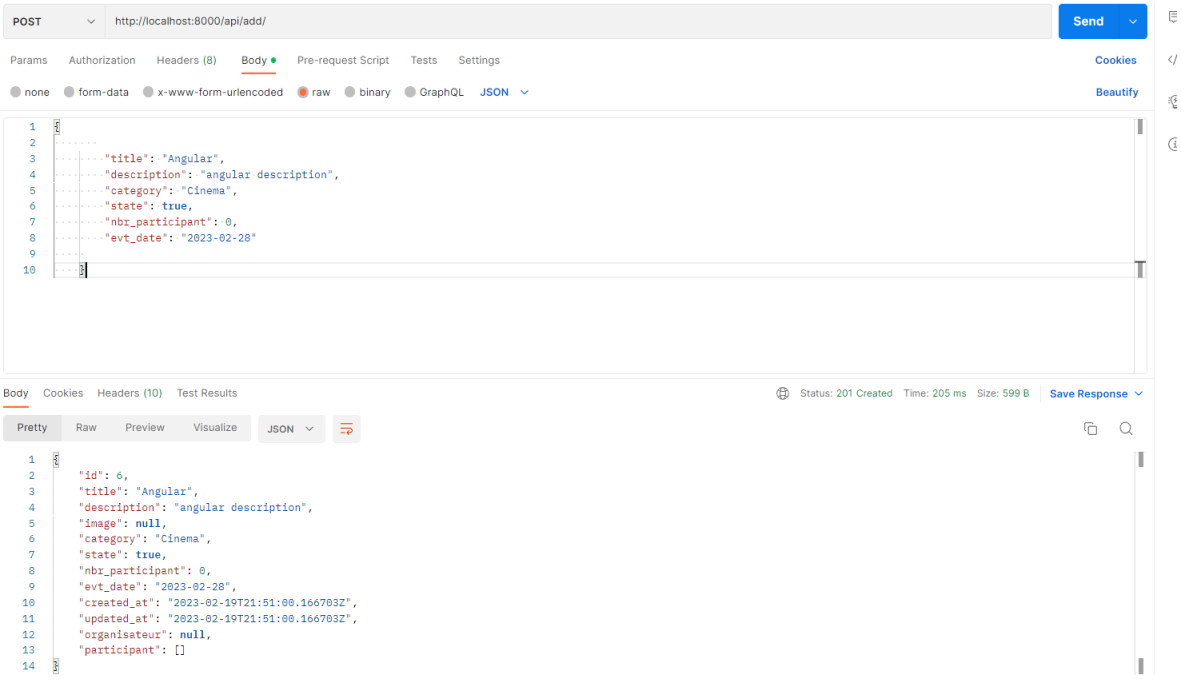
4. Ajout d'un évènement :

Le code ci-dessous permet d'ajouter un évènement

```python
@api_view(['POST'])
def addEvent(request):
    serializer = EventSerializer(data=request.data)
    if serializer.is_valid():
            serializer.save()

            return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

- Pour le test, appelez l'url suivant : http://127.0.0.1:8000/api/add



5. Modification d’un évènement :

Le code ci-dessous permet de modifier un évènement

```python
@api_view(['GET','PUT'])
def updateEvent(request , id= None):
    event = Event.objects.get(id=id)

    serializer = EventSerializer(instance=event,data=request.data)
    if serializer.is_valid():
            serializer.save()

            return Response(serializer.data, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

La fonction prend en paramètre la requête "request" et un identifiant "id" pour l'événement à mettre à jour.

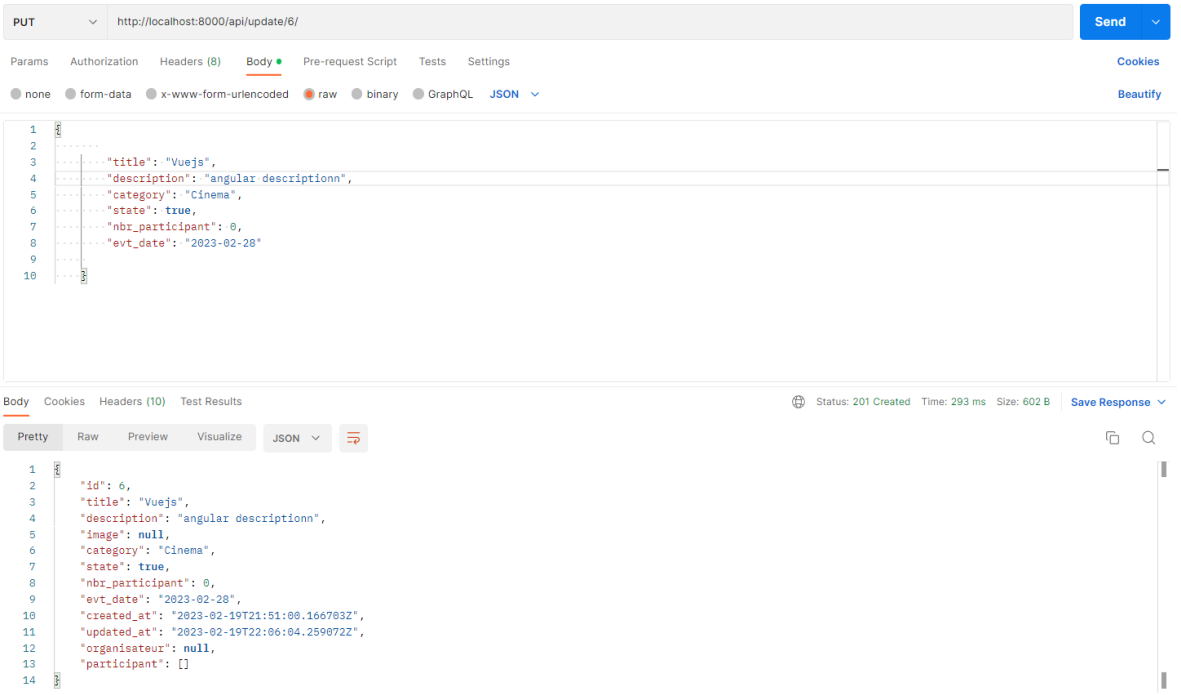
- **@api_view(['GET', 'PUT'])** est un décorateur qui indique les méthodes HTTP autorisées pour cette vue. Dans ce cas, il s'agit de la méthode GET pour récupérer les informations d'un événement existant et de la méthode PUT pour mettre à jour.

- **event = Event.objects.get(id=id)** récupère l'objet "Event" correspondant à l'identifiant "id" à partir de la base de données.

- **serializer = EventSerializer(instance=event, data=request.data)** crée une instance de la classe de sérialisation "EventSerializer" en utilisant l'objet "event" récupéré de la base de données comme instance et les données de la requête HTTP comme données.

- **if serializer.is_valid():** vérifie si les données soumises sont valides en utilisant la méthode "is_valid()" du sérialiseur. Si les données sont valides, la méthode "save()" du sérialiseur est appelée pour enregistrer les données dans la base de données. Si les données sont enregistrées avec succès.

Pour le test, appelez l’url suivante :
http://127.0.0.1:8000/api/update/6/



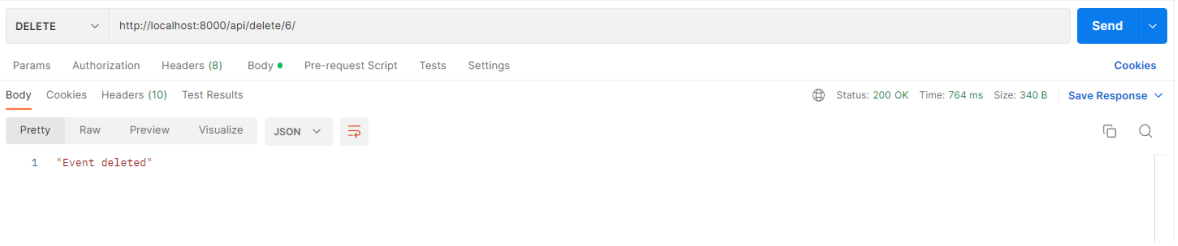
6. Suppression d'un évènement :

Le code ci-dessous permet de supprimer un évènement

```python
@api_view(['GET','DELETE'])
def deleteEvent(request, id=None):
    try:
        event = Event.objects.get(id=id)
    except event.DoesNotExist:
        return Response(status="Event not found")

    event.delete()
    return Response("Event deleted")
```

- Pour le test, appelez l'url suivante : http://127.0.0.1:8000/api/delete/6



7. Urls.py :

Les URL définies sont associées à différentes vues pour effectuer des opérations CRUD (Create, Read, Update, Delete) sur le modèle "Event".

```python
from django.urls import path
from .views import  *

urlpatterns = [
    path('' , getEvents),
    path('add/', addEvent),
    path('delete/<int:id>/', deleteEvent),
    path('update/<int:id>/', updateEvent),
]
```

- N’oubliez pas d’inclure les URL définies dans le module "urls.py" de l'application "api".

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('event/', include('Event.urls')),
    path('api/', include('api.urls'))

]
```

A bientôt