En Django REST Framework (DRF), les classes APIView et ViewSet permettent de créer des API, mais elles sont utilisées de manières différentes selon ton besoin de contrôle vs automatisation.

APIView vs ViewSet — Résumé global

Critère	APIView	ViewSet
Niveau d'abstraction	Bas (plus de code à écrire)	Élevé (DRF génère beaucoup pour toi)
Contrôle	Maximal (tu codes chaque méthode)	Plus automatique, moins de contrôle
S Routing / URLs	Manuellement dans urls.py	Automatiquement via routers
Néthodes à définir	get, post, put, patch, etc.	list, retrieve, create, etc.

1. APIView – plus classique et explicite

Tu dois écrire **toi-même** les méthodes HTTP (get, post, put, etc.).

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
class HelloAPIView(APIView):
    def get(self, request):
        return Response({'message': 'Hello, GET!'})
    def post(self, request):
        return Response({'message': 'Hello, POST!'},
status=status.HTTP_201_CREATED)
```

🔁 Tu déclares **chaque méthode** manuellement. Tu as un **contrôle total**, mais c'est plus de travail.

2. ViewSet – plus automatisé et structuré

Avec un ViewSet, tu écris des méthodes comme list, retrieve, create et le router s'occupe de générer les routes automatiquement.

```
from rest_framework import viewsets
from rest_framework.response import Response
class HelloViewSet(viewsets.ViewSet):
```

```
def list(self, request):
    return Response({'message': 'Hello from list'})

def create(self, request):
    return Response({'message': 'Hello from create'})
```

Et dans urls.py:

```
from rest_framework.routers import DefaultRouter
from .views import HelloViewSet

router = DefaultRouter()
router.register('hello', HelloViewSet, basename='hello')

urlpatterns = router.urls
```

- DRF va créer automatiquement des routes comme :
 - /hello/ (GET → list, POST → create)
 - /hello/{id}/ (GET → retrieve, PUT/PATCH → update, DELETE → destroy)

@ Quand utiliser quoi?

Tu veux	Utilise
Un contrôle fin sur les requêtes	APIView
Écrire une API simple et rapide	ViewSet
Personnaliser à fond le comportement des routes	APIView
Générer automatiquement toutes les routes REST	ViewSet

Résumé en une phrase

APIView = tu contrôles tout manuellement. ViewSet = tu profites de la puissance du router pour suivre automatiquement les conventions REST.

On va créer un mini projet DRF avec une API pour gérer des livres, en deux versions :

- 1. Avec APIView (manuelle)
- 2. Avec ViewSet + Router (automatique)

1. Version APIView (manuelle)

✓ Objectif : Gérer une liste de livres (GET, POST)

models.py

```
from django.db import models

class Book(models.Model):
   title = models.CharField(max_length=255)
   author = models.CharField(max_length=255)

def __str__(self):
   return self.title
```

serializers.py

```
from rest_framework import serializers
from .models import Book

class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ['id', 'title', 'author']
```

views.py (avec APIView)

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status
from .models import Book
from .serializers import BookSerializer
class BookListCreateAPIView(APIView):
    def get(self, request):
        books = Book.objects.all()
        serializer = BookSerializer(books, many=True)
        return Response(serializer.data)
    def post(self, request):
        serializer = BookSerializer(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)
```

urls.py

```
from django.urls import path
from .views import BookListCreateAPIView

urlpatterns = [
   path('books/', BookListCreateAPIView.as_view(), name='book-list-create'),
]
```

2. Version ViewSet + Router (automatique)

views.py (avec ViewSet)

```
from rest_framework import viewsets
from .models import Book
from .serializers import BookSerializer

class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

urls.py

```
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import BookViewSet

router = DefaultRouter()
router.register('books', BookViewSet)

urlpatterns = [
   path('', include(router.urls)),
]
```

Résultat comparatif

Action	APIView route	ViewSet route (auto)
Liste des livres (GET)	/books/	/books/
Créer un livre (POST)	/books/	/books/

Action	APIView route	ViewSet route (auto)
Détail d'un livre	💢 (non implémenté)	/books/ <id>/</id>
Supprimer (DELETE)	X	/books/ <id>/ (DELETE)</id>
Mettre à jour (PATCH)	×	/books/ <id>/ (PATCH)</id>

Avec APIView:

Tu dois toi-même implémenter chaque méthode HTTP que tu veux supporter.

Par exemple, si tu veux gérer un livre avec :

- GET (lister ou récupérer),
- POST (créer),
- PATCH (mettre à jour),
- DELETE (supprimer),

```
class BookDetailView(APIView):
    def get(self, request, pk):
        ...

def patch(self, request, pk):
        ...

def delete(self, request, pk):
        ...
```

₹ Tu as un contrôle total, mais c'est plus de code.

Avec ViewSet:

Tu écris zéro méthode HTTP directement.

Tu définis simplement :

- le queryset (ce que tu veux manipuler),
- le serializer.

```
class BookViewSet(viewsets.ModelViewSet):
   queryset = Book.objects.all()
   serializer_class = BookSerializer
```

Ensuite:

• Django REST Framework crée automatiquement les routes REST (GET, POST, PUT, PATCH, DELETE,

• Tu peux surcharger les méthodes seulement si besoin (create, update, destroy, etc.).



◯ Tu veux	Choix recommandé
Simplicité + rapidité	ViewSet
Contrôle fin + logique personnalisée	APIView

LES ACTIONS D'UNE VIEWSET



Qu'est-ce qu'un ViewSet ?

Un ViewSet est une classe qui regroupe plusieurs vues (GET, POST, PUT, DELETE, etc.) pour gérer un modèle de manière structurée et réutilisable.

Les actions principales dans un ModelViewSet

Un ModelViewSet fournit automatiquement 6 actions principales (CRUD complet):

Action DRF	Méthode HTTP	Utilité	URL typique
list()	GET	Récupérer une liste d'objets	/api/recipes/
retrieve()	GET	Récupérer un objet précis	/api/recipes/1/
create()	POST	Créer un nouvel objet	/api/recipes/
update()	PUT	Mettre à jour un objet entièrement	/api/recipes/1/
<pre>partial_update()</pre>	PATCH	Mettre à jour partiellement un objet	/api/recipes/1/
destroy()	DELETE	Supprimer un objet	/api/recipes/1/

Exemple d'un ModelViewSet

```
from rest_framework import viewsets
from .models import Recipe
from .serializers import RecipeSerializer
class RecipeViewSet(viewsets.ModelViewSet):
    queryset = Recipe.objects.all()
    serializer_class = RecipeSerializer
```

Avec ça, tu obtiens toutes les actions automatiquement.



Personnaliser une action

Tu peux surcharger une action par son nom:

```
def create(self, request, *args, **kwargs):
   # Logique personnalisée
   return super().create(request, *args, **kwargs)
```

Ajouter une action personnalisée avec @action

Parfois tu veux une action qui n'est ni list, ni create, ni retrieve, etc.

Utilise @action:

```
from rest_framework.decorators import action
from rest_framework.response import Response
class RecipeViewSet(viewsets.ModelViewSet):
   @action(detail=True, methods=['post'])
   def upload_image(self, request, pk=None):
        recipe = self.get_object()
        recipe.image = request.FILES.get('image')
        recipe.save()
        return Response({'status': 'image uploaded'})
```

- detail=True:s'applique à un objet (ex: /recipes/1/upload_image/)
- detail=False:s'applique à la collection (ex: /recipes/stats/)

🗱 Résumé

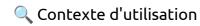
Méthode	Utilité principale
list()	Liste d'objets
retrieve()	Un objet
create()	Créer un objet
update()	Remplacer un objet
partial_update()	Modifier partiellement un objet
destroy()	Supprimer

Méthode

Utilité principale

@action

Ajouter des comportements custom



Prenons cet exemple classique dans un ViewSet:

```
def get_serializer_class(self):
    if self.action == 'list':
        return RecipeSerializer
    elif self.action == 'upload_image':
        return RecipeImageSerializer
    return RecipeDetailSerializer
```

À quoi sert self.action?

self.action est une propriété de ViewSet qui contient le nom de l'action appelée. Elle vaut :

- "list" → si tu appelles GET /recipes/
- "retrieve" → si tu appelles GET /recipes/1/
- "create" → si tu appelles POST /recipes/
- "upload_image" → si tu appelles une action personnalisée définie avec @action(name="upload_image")

Pourquoi 'upload_image'?

C'est le **nom de l'action personnalisée** que tu as définie :

```
from rest_framework.decorators import action

class RecipeViewSet(viewsets.ModelViewSet):

    @action(methods=['POST'], detail=True, url_path='upload-image')
    def upload_image(self, request, pk=None):
    ...
```

➡ Dans ce cas, self.action == 'upload_image' quand cette méthode est appelée.

V Utilité

Cela permet d'utiliser un serializer spécifique pour cette action :

```
elif self.action == 'upload_image':
    return RecipeImageSerializer
```

Ex: le RecipeImageSerializer contient seulement un champ image, pas tous les détails d'une recette complète.

Quand tu crées une action personnalisée dans un ViewSet de Django REST Framework (DRF), tu dois :

1. Utiliser le décorateur @action

Ce décorateur te permet de définir :

- les méthodes HTTP supportées (methods)
- si l'action est liée à un objet (avec detail=True) ou pas
- le nom dans l'URL (avec ur l_path)

Exemple complet :

```
from rest_framework.decorators import action
from rest_framework.response import Response
from rest_framework import status

class RecipeViewSet(viewsets.ModelViewSet):

    # Action personnalisée pour uploader une image sur une recette
existante
    @action(methods=['POST'], detail=True, url_path='upload-image')
    def upload_image(self, request, pk=None):
        recipe = self.get_object()
        serializer = RecipeImageSerializer(recipe, data=request.data)

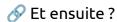
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=status.HTTP_200_0K)

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

Explication des paramètres :

Paramètre	Description
methods=['POST']	Spécifie que cette action accepte uniquement des requêtes POST
detail=True	Indique que cette action agit sur une instance spécifique (ex:
	/recipes/1/upload-image/)

Paramètre	Description
url_path='upload-	Ce sera le suffixe dans l'URL. DRF construira /recipes/ <id>/upload-</id>
image'	image/



DRF expose cette action dans la route automatiquement (si tu utilises un DefaultRouter), donc pas besoin de l'ajouter manuellement à urls.py.



💡 Pour les actions globales (non liées à un objet) :

Si tu voulais une action comme /recipes/stats/, tu utiliserais:

```
@action(methods=['GET'], detail=False, url_path='stats')
def stats(self, request):
```