Vue3

Ibrahim ALAME

4 janvier 2024

1 Découvrez Django REST Framework

1.1 Découvrez pourquoi utiliser une API

Grosso modo, c'est quelque chose qui permet de communiquer facilement avec tout un système d'information. On actionne un levier et bim ça fait ce qu'on veut derrière (il s'annonce bien ce cours !). Mais on va quand même voir ce qu'est un levier, et surtout comment on l'actionne, rassurez-vous !

API signifie *Application Programming Interface* (ou *interface de programmation applicative*). C'est une grosse interface qui permet d'interagir avec un système d'information au travers de ce qu'on appelle des endpoints. Chaque endpoint permet d'exécuter différentes actions dans le système d'information, sans avoir à en comprendre le fonctionnement. Ces actions peuvent être la consultation d'un panier, l'envoi d'un e-mail, l'authentification auprès d'un service... les possibilités sont nombreuses.

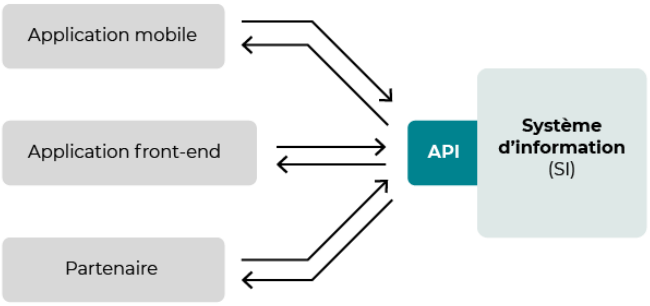
> **1**
>
> En gros, c'est un peu comme utiliser une classe fournie par une librairie tierce. C'est-à-dire qu'on utilise des méthodes d'objets de cette librairie que nous n'avons pas écrites nous-même. Dans notre cas, la classe est le système d'information, et la méthode est l'endpoint.

Mais concrètement, un endpoint, c'est quoi ?

Un endpoint est une URL sur laquelle on réalise différents appels. Selon la méthode HTTP utilisée (GET, POST, PATCH, DELETE), une partie de code va être exécutée et retourner un résultat. Ce résultat est constitué :

- D'un status code (200, 201, 400, etc.) qui indique le succès ou non de l'appel ;
- D'un contenu qui est en JSON dans la majorité des cas (peut également être du XML dans certains cas), et qui va contenir des informations soit de succès soit d'erreur. Ces appels peuvent ensuite être paramétrés avec des filtres dans les paramètres d'URL ou dans le corps de la requête.

Les API sont aujourd'hui devenues un standard. Elles permettent de centraliser toute l'information en ayant moins de données et de logique métier à gérer dans les terminaux. Une seule application Backend proposant une API peut être mutualisée et être consommée par une application Front, des applications mobiles Android/iOS, mais aussi d'autres systèmes d'information, des partenaires, etc.



Une API peut être créée dès le début d'un projet, mais aussi être mise en place sur un projet existant pour le faire évoluer. C'est ce dernier que nous allons ensuite voir ensemble, en **adaptant un projet Django existant pour lui intégrer une API avec Django REST Framework** (DRF).

1.2 Découvrez le cas concret du cours

Tout au long de ce cours, nous allons *mettre en place une API* sur un projet de boutique en ligne qui permet à des utilisateurs de consulter un catalogue de produits rangé par catégories, produits et articles.

Nous permettrons ensuite aux administrateurs de la plateforme de gérer la boutique en ligne, en ajoutant de nouveaux produits à mettre en vente, et en masquant certains produits s'ils ne sont plus disponibles. Nous devrons alors sécuriser certains endpoints pour qu'ils ne soient pas accessibles publiquement.

Le code de l'application est disponible sur ce repository. À la fin de certains chapitres, des exercices seront proposés en partant d'une branche Git, et une solution sera proposée dans une autre branche.

1.3 Installez et configurez DRF

Django REST Framework est une librairie permettant la mise en place d'une API pour Django (vous pouvez regarder sa documentation sur le site officiel, en anglais). Basée sur le framework, elle propose la mise en place des endpoints d'une façon similaire à la mise en place des URL, Views et Forms de Django.

Pourquoi utilise-t-on cette librairie ?

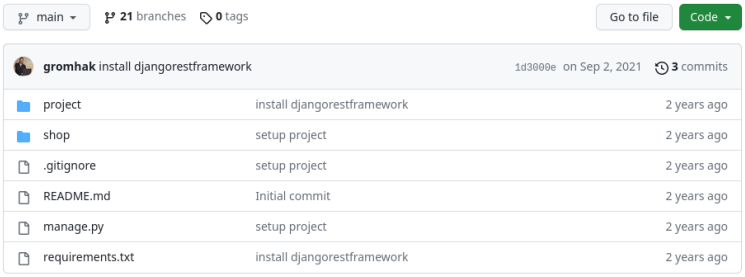
Écrire une API à la main est possible, on pourrait écrire nous-mêmes nos endpoints en nous basant sur les Views de Django, et retourner du JSON.

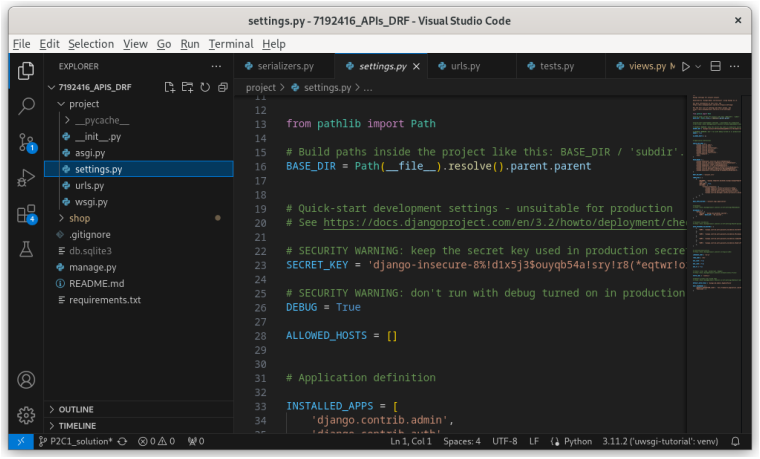
Mais c'est en réalité une quantité de travail phénoménale, et d'autres développeurs se sont déjà penchés sur le sujet pour simplifier grandement l'écriture d'API.

Dans ce chapitre, nous allons faire simple et :

- Mettre en place la librairie, en l'ajoutant aux dépendances de notre projet ;
- Déclarer DRF parmi les applications du projet Django pour permettre son utilisation ;
- Configurer une première URL fournie par DRF pour s'assurer du bon fonctionnement de la librairie.

En partant de la branche main de notre projet,





commençons par **ajouter la dépendance à DRF** dans notre fichier requirements.txt :

```
djangorestframework==3.12.4
```

Un petit coup d' install dans notre environnement virtuel :

```
pip install -r requirements.txt
```

Puis, déclarons DRF dans la liste des applications installées du fichier settings.py du projet Django :

```
INSTALLED_APPS = [
'rest_framework',
]
```

Pour nous assurer que notre API est fonctionnelle, nous allons activer l'authentification fournie par DRF pour nous connecter. Éditons notre fichier urls.py :

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

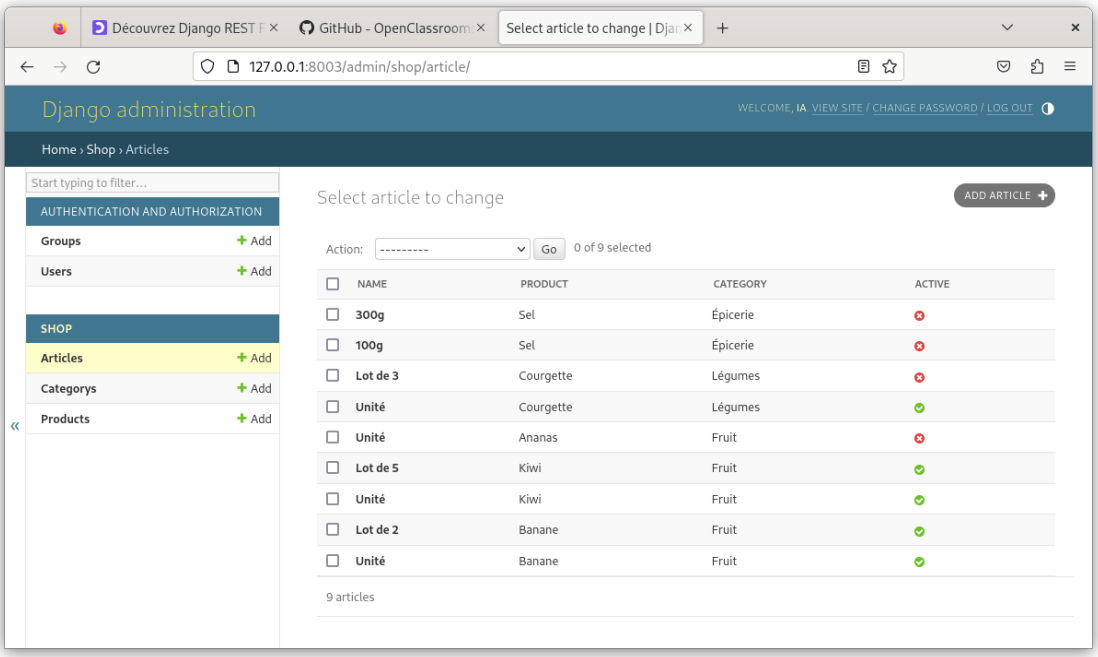
Démarrons à présent notre serveur de développement et allons nous connecter sur notre API à présent en place :

```
python manage.py runserver
```

Nous devrions maintenant pouvoir nous rendre sur notre URL et nous connecter. Le projet contient une base de données SQLite qui contient déjà un compte administrateur dont voici les identifiants :

- Nom d'utilisateur : admin-oc
- Mot de passe : password-oc

Le projet étant en local, nous pouvons nous connecter à l'URL http://127.0.0.1:8000/admin.
Nous avons installé et configuré DRF !



En résumé

- Les API sont des interfaces permettant l'échange et le traitement d'informations entre un système d'information et toute autre application.
- Un endpoint est une URL sur laquelle on peut effectuer des opérations en fonction de la méthode HTTP utilisée.
- Django Rest Framework est une librairie permettant de mettre en place une API sur un projet Django.

Maintenant, on est prêts pour la suite, et sans attendre on va mettre en place notre premier endpoint !

2 Gérez des données avec un endpoint

2.1 Créez un premier endpoint

En avant, mettons en place ce premier endpoint en permettant à nos visiteurs d'accéder à la liste des catégories de nos produits. La toute première chose à faire lors de la réalisation d'un endpoint est de se demander **quelles sont les informations importantes que nous souhaitons en tirer**.

Dans notre cas, pour afficher la liste des catégories, nous allons avoir besoin de leur nom, mais également de leur identifiant. L'identifiant sera utile aux clients (application front, mobile...) pour identifier de manière claire et unique les catégories s'ils ont des actions à faire dessus, comme afficher les produits qui constituent une catégorie.

DRF met à notre disposition des serializers qui permettent de transformer nos models Django en un autre format qui, lui, est exploitable par une API. Lorsque notre API sera consultée, le *serializer* va nous permettre de transformer notre objet en un JSON et, inversement, lorsque notre API va recevoir du JSON, elle sera capable de le transformer en un objet.

Pour le développeur qui les utilise, ils fonctionnent à la manière des formulaires proposés par Django. C'est-à-dire qu'ils sont paramétrés en précisant le model à utiliser et les champs de ce model à sérialiser.

2

Pour rappel, un `modelDjango` est une classe matérialisée dans la base de données, et qui permet donc l'utilisation de l'Object-Relational Mapping (ORM) de Django. Il sert à représenter et faire persister les données métier de notre système d'information. Dans notre cas, il s'agit des catégories, produits et articles mis en vente sur notre boutique.

Créons un fichier `serializers.py` et *écrivons notre premier serializer*, que nous nommerons `CategorySerializer` pour rester clair et précis. ;)

Il est nécessaire de définir sa classe `Meta` et la liste des champs, exactement comme pour un formulaire. De plus, les noms des attributs sont identiques.

```
from rest_framework.serializers import ModelSerializer
from shop.models import Category

class CategorySerializer(ModelSerializer):
    class Meta:
        model = Category
        fields = ['id', 'name']
```

Occupons-nous à présent de la View. **DRF** nous propose une APIView qui a également un fonctionnement similaire aux `Views` de Django.

Nous devons réécrire la méthode get qui réalisera les actions suivantes :

- Récupérer toutes les catégories en utilisant l'ORM de Django ;
- Sérialiser les données à l'aide de notre serializer ;
- Renvoyer une réponse qui contient les données sérialisées.

```python
from rest_framework.views import APIView
from rest_framework.response import Response
from shop.models import Category
from shop.serializers import CategorySerializer

class CategoryAPIView(APIView):
    def get(self, *args, **kwargs):
        categories = Category.objects.all()
        serializer = CategorySerializer(categories, many=True)
        return Response(serializer.data)
```

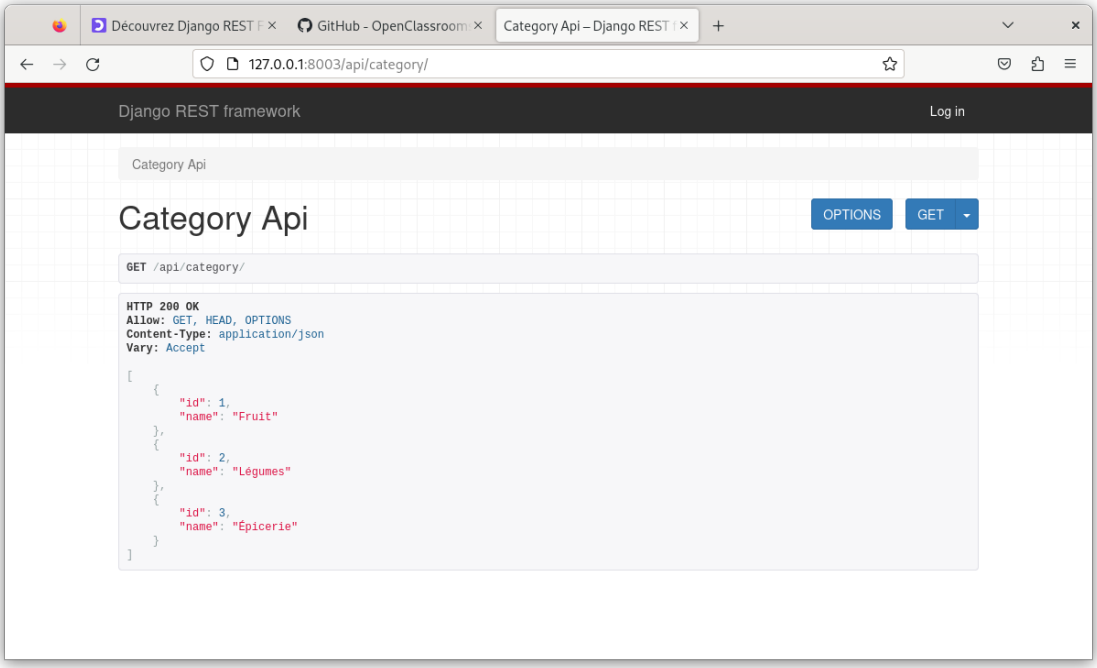
Le paramètre many permet de préciser au Serializer qu'il va devoir générer une liste d'éléments à partir de l'itérable (notre queryset) qui lui est transmis.

Et si je n'ai qu'un seul élément à sérialiser ?

Dans ce cas, il n'est pas nécessaire de préciser le paramètre many, et l'objet peut directement être donné au Serializer sans le mettre dans un *itérable*. Enfin, pour obtenir les données sérialisées, nous appelons la propriété `data` de notre serializer. Ce sont ces données qui sont utilisées pour construire la réponse. Il ne reste plus qu'à créer l'URL correspondante et l'associer à notre View :

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

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

Notre endpoint est maintenant accessible sur l'URL http://127.0.0.1:8000/api/category/ . L'interface proposée par DRF nous permet de voir également le status code obtenu. Dans notre cas, 200, qui indique un succès.



N'hésitez pas à jouer avec notre nouveau serializer, lui ajouter des champs, comme les dates de création et de modification. Tentez l'écriture pour les autres méthodes HTTP, l'expérimentation est un levier !

2.2 Découvrez les méthodes d'un endpoint

Seul GET est accessible sur notre endpoint, car nous n'avons redéfini que la méthode get dans la View. Ne pas réécrire de méthodes permet de ne pas les rendre accessibles. Nous pourrions réécrire également les méthodes post, patch et delete.

2.2.1 `GET` et `POST` servent avec Django à créer de nouvelles entités au travers de formulaires, mais `PATCH` et `DELETE` servent à quoi ?

Reprenons-les une par une pour voir réellement à quoi servent ces méthodes HTTP dans le cadre d'une API, et les différences avec un site Internet classique :

- GET : Permet la lecture d'informations. Un peu comme un site Internet classique, les appels en GET renvoient des données qui sont généralement du HTML qui est lu et rendu dans le navigateur. Dans le cas d'une API, il s'agit de JSON.
- POST : Permet la création d'entités. Alors que sur un site les appels en POST peuvent être utilisés pour modifier une entité, dans le cas d'une API, les POST permettent la création.
- PATCH : Permet la modification d'une entité. PATCH permet la modification de tout ou partie des valeurs des attributs d'une entité pour une API, alors que pour un site classique, nous utilisons un formulaire et un POST.
- DELETE : Permet la suppression d'une entité. DELETE permet la suppression d'une entité pour une API, alors que pour un site classique, nous utilisons un formulaire et généralement un POST.

- PUT : Permet également la modification d'une entité. Il est peu utilisé en dehors de l'interface de DRF, que nous verrons ensuite. Pour un site classique, l'action de modification entière d'une entité passe également par un POST au travers d'un formulaire.

Pas de panique, tout n'est pas à retenir tout de suite. Durant ce cours, nous allons aborder chacune de ces méthodes dans divers cas pratiques.

Ce qui est important à retenir est que chaque méthode correspond à une action du CRUD (Create, Read, Update, Delete), et c'est cela qui va nous permettre d'agir directement sur nos models.

Exercice

Je vous propose de prendre la main et de mettre en place un nouvel endpoint qui va permettre de lister tous les produits de notre boutique en ligne. Utilisons l'URL suivante : http://127.0.0.1:8000/api/product/. Cet endpoint doit retourner les informations suivantes des produits :

- Son identifiant `id`.
- Sa date de création `date_created`.
- Sa date de modification `date_updated`.
- Son nom `name`.
- L'identifiant de la catégorie à laquelle il appartient `category`.

Pour réaliser cela, vous pouvez partir de la branche P1C3_exercice. Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche P1C3_solution.

En résumé

1. Le fonctionnement des serializers proposé par DRF est similaire à celui des formulaires de Django.
2. Les serializers permettent de faire une représentation de nos models en JSON.
3. Les méthodes HTTP utilisables sont la représentation du CRUD (Create, Read, Update, Delete).
4. DRF nous propose une interface web qui permet de visualiser notre API.

N'hésitez pas à modifier les données que retournent le serializer. Imaginez le but de l'endpoint que vous êtes en train de mettre en place pour déterminer quelles sont les données les plus pertinentes à retourner.

3 Rendez les Views plus génériques avec un ModelViewset

4 Mettez en place un router

Plutôt que de créer nos URL et de redéfinir nos méthodes une à une, DRF propose des classes héritables pour nos vues, qui sont les ModelViewsets. Elles permettent la mise en place rapide de toutes les actions du CRUD pour un model donné. Utiliser un ModelViewset nécessite d'utiliser une autre façon de définir nos URL. Cela se fait au travers d'un router.

Un router, c'est quoi ?

Un router permet de définir automatiquement toutes les URL accessibles pour un endpoint donné. Il va à la fois permettre de définir :

- L'URL `/api/category/`, qui permet de réaliser des actions globales qui ne concernent pas directement une entité précise, comme la récupération de la liste des entités ou la création d'une nouvelle.
- L'URL `/api/category/<pk>/`, qui en acceptant un paramètre correspondant à l'identifiant d'une entité, va permettre de réaliser des actions sur celle-ci, comme obtenir des informations, la modifier ou la supprimer.

Mettons en place notre router dans notre fichier `urls.py` et supprimons notre ancien endpoint :

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers
from shop.views import CategoryViewset

# Ici nous créons notre routeur
router = routers.SimpleRouter()
# Puis lui déclarons une url basée sur le mot clé 'category' et notre view
# afin que l'url générée soit celle que nous souhaitons '/api/category/'
router.register('category', CategoryViewset, basename='category')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
    path('api/', include(router.urls))  # Il faut bien penser à ajouter les
    # urls du router dans la liste des urls disponibles.
]
```

Le router se définit en amont de la définition de `urlpatterns`. Les URL sont incluses avec un `include` au travers de la propriété `router.urls`. Notons également qu'il n'est plus nécessaire d'appeler `.as_view()`, le router le fait pour nous lorsqu'il génère les URL.

3

Le paramètre `basename` permet de retrouver l'URL complète avec la fonction `redirect`, comme le propose Django. Cela sera utile lors de l'écriture des tests que nous aborderons ensuite.

4.1 Transformez une ApiView en ModelViewset

À présent, nous devons transformer notre ApiView en un ModelViewset pour que notre vue puisse être connectée à notre routeur. Un ModelViewset est comparable à une super vue Django qui regroupe à la fois CreateView, UpdateView, DeleteView, ListView et DetailView. Il faut impérativement lui *définir deux attributs de classe* :

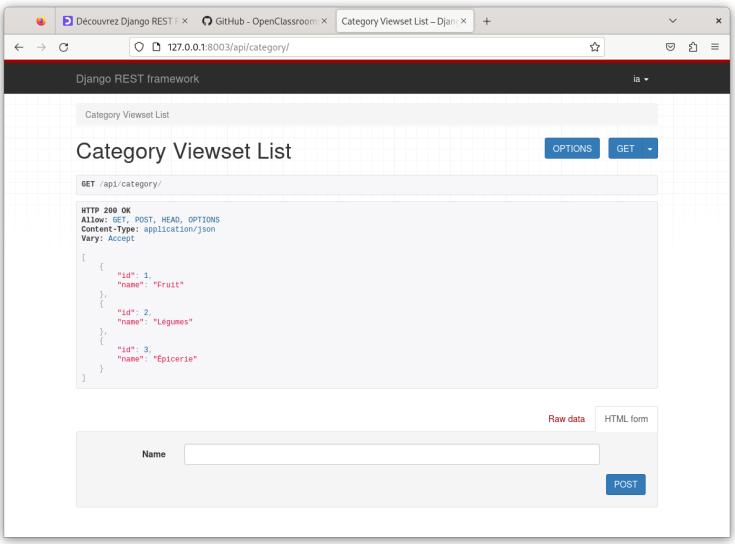
1. `serializer_class` qui va déterminer le `serializer` à utiliser ;
2. `queryset`, ou réécrire la méthode `get_queryset` qui doit retourner un Queryset des éléments à retourner.

4

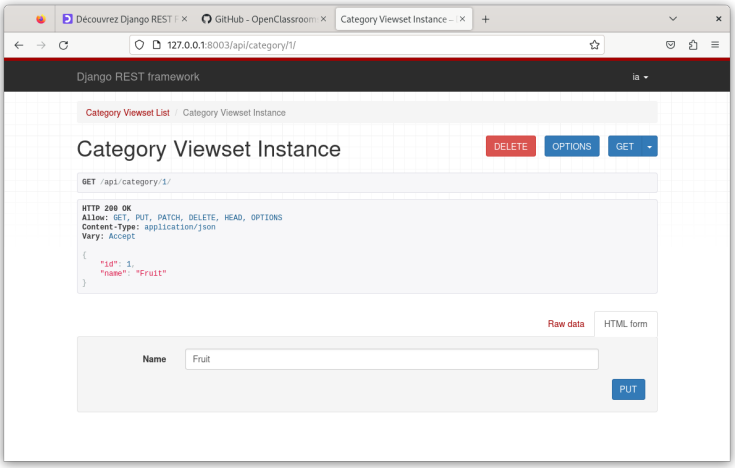
Redéfinir seulement l'attribut de classe `queryset` permet principalement de faire des tests rapides. À l'usage, redéfinir `get_queryset` est souvent la solution à adopter car elle permet d'être plus fin sur les éléments à retourner.

```python
from rest_framework.viewsets import ModelViewSet
from shop.models import Category
from shop.serializers import CategorySerializer

class CategoryViewset(ModelViewSet):

    serializer_class = CategorySerializer

    def get_queryset(self):
        return Category.objects.all()
```

Nous pouvons dès à présent retester notre API à l'adresse http://127.0.0.1:8000/api/category/. Le résultat est identique à ce que nous avions précédemment. La différence étant qu'il nous est maintenant possible de réaliser directement les autres opérations du CRUD. Le formulaire en bas de page vous invite directement à réaliser un POST pour créer une nouvelle catégorie.



Le détail d'une catégorie est également visible en ajoutant son identifiant dans l'URL, par exemple http://127.0.0.1:8000/api/category/1/. La page vous propose alors de réaliser les actions PUT, PATCH (dans l'onglet « Raw data ») et DELETE sur l'entité consultée.



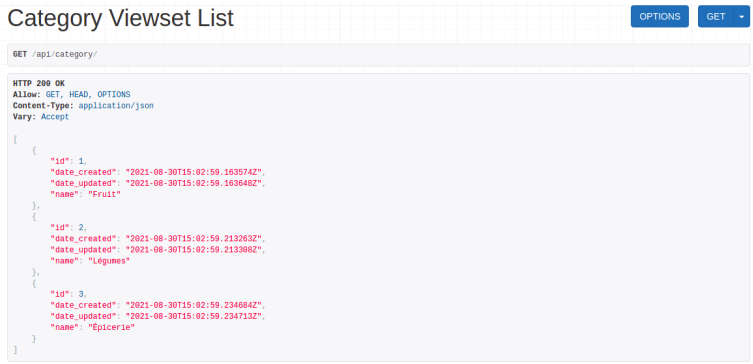
4.2 Ne permettez que la lecture

Vous l'aurez sûrement remarqué, nous pouvons créer de nouvelles catégories ; dans la pratique, permettre la création, la modification et la suppression sur un endpoint public comme le nôtre n'est pas conseillé. Nous allons donc faire en sorte que notre endpoint *ne permette que la lecture*, car son but est d'afficher à nos utilisateurs la liste des catégories disponibles sur la boutique. Pour cela, DRF nous propose un autre type de ModelViewset. Il s'agit du ReadOnlyModelViewset qui, comme son nom l'indique, ne permet que la lecture.

Modifions notre vue pour limiter les opérations disponibles. Nous allons faire étendre la vue CategoryViewset avec le `viewset` ReadOnlyModelViewset au lieu du ModelViewset actuel.

```
from rest_framework.viewsets import ReadOnlyModelViewSet
from shop.models import Category
from shop.serializers import CategorySerializer

class CategoryViewset(ReadOnlyModelViewSet):
    serializer_class = CategorySerializer
    def get_queryset(self):
        return Category.objects.all()
```

Si nous consultons à présent notre API, nous pouvons constater que toutes les options autres que la lecture ne sont plus permises.



5

Il est également possible d'utiliser l'attribut `read_only_fields` sur le serializer pour préciser les champs en lecture seule. Mais cela n'empêchera pas les actions de création, modification et suppression.

Exercice

Notre endpoint de consultation de produits ne doit également permettre que la lecture. Entraînez-vous dans l'utilisation des Viewsets pour passer cet endpoint en « lecture seule ».

- Pour réaliser cela, vous pouvez partir de la branche P1C4_exercice. Elle contient déjà ce que nous venons de faire ensemble.
- Une solution est proposée sur la branche P1C4_solution.

En résumé

1. Un router permet de définir en une seule fois toutes les opérations du CRUD sur un endpoint.
2. Utiliser un `ModelViewset` impose d'utiliser un router pour définir ses URL.
3. Lors de l'utilisation d'un `ModelViewset`, il faut définir :
   (a) le serializer à utiliser avec l'attribut de classe `serializer_class` ;
   (b) le jeu de données qui sera retourné en réécrivant la méthode `get_queryset`.
4. `ReadOnlyModelViewset` permet de limiter les accès à la lecture seule.

Les Viewsets sont très souvent les vues à privilégier, nous allons dès maintenant voir comment les personnaliser et les adapter avec plus de précision à nos besoins. Suivez-moi au prochain chapitre !

5 Filtrez les résultats d'un endpoint

5.1 Appliquez un filtre sur les données retournées

Lorsque nos utilisateurs naviguent sur notre boutique, il se peut qu'ils souhaitent récupérer les produits d'une catégorie grâce à un appel fait à notre API. Sauf que pour le moment, notre

endpoint de produits ne permet pas de *filtrer par catégorie*. Pour réaliser cela, nous allons accepter un paramètre dans l'URL qui sera l'identifiant de la catégorie, pour ne renvoyer que les produits correspondants. Le paramètre sera nommé `category_id`, ce qui donnera une URL sous le format

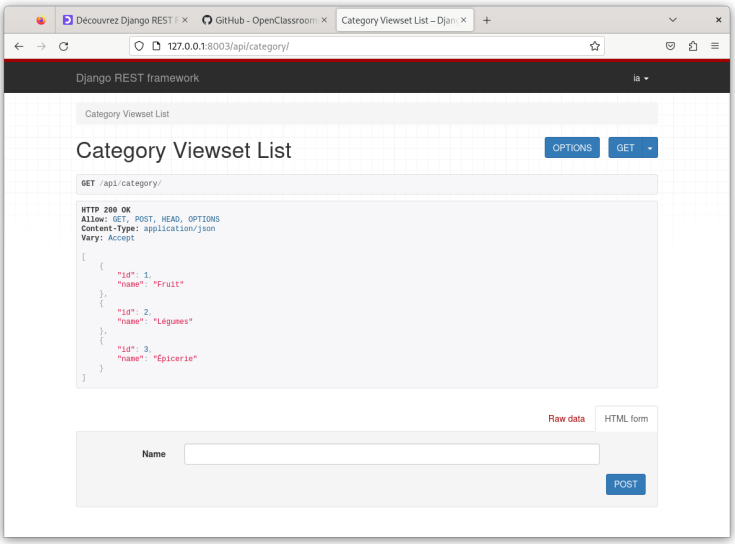
http://127.0.0.1:8000/api/product/?category_id=1.

Éditons notre classe `ProductViewset` pour qu'elle applique un nouveau filtre si le paramètre est présent. Profitons-en pour également appliquer le filtre sur les produits actifs :

```
class ProductViewset(ReadOnlyModelViewSet):
    serializer_class = ProductSerializer
    def get_queryset(self):
    # Nous récupérons tous les produits dans une variable nommée queryset
        queryset = Product.objects.filter(active=True)
        # Vérifions la présence du paramètre 'category_id' dans l'url et si oui alors appliquons
        category_id = self.request.GET.get('category_id')
        if category_id is not None:
            queryset = queryset.filter(category_id=category_id)
        return queryset
```

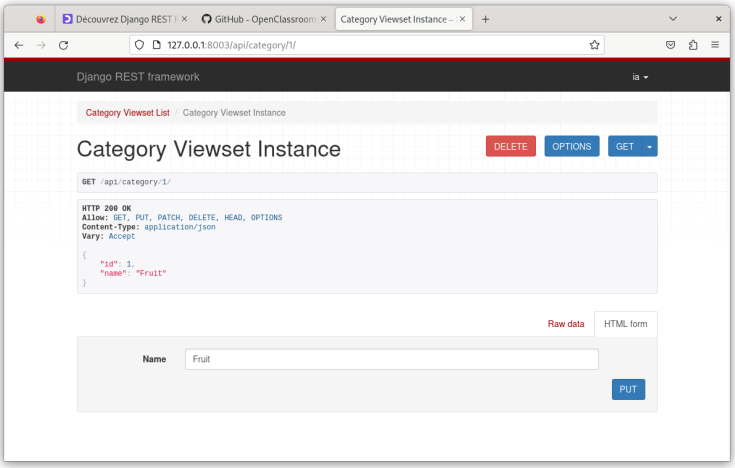
À présent, nous pouvons constater que le filtre est bien appliqué si nous consultons la liste des produits de la catégorie 1 avec l'URL

http://127.0.0.1:8000/api/product/?category_id=1.



Et si nous demandons une catégorie qui n'existe pas, alors une liste de produits vide est retournée sans faire planter notre application :

http://127.0.0.1:8000/api/product/?category_id=7777777.



Ainsi, les paramètres d'URL peuvent servir à appliquer *toutes sortes de filtres sur les données retournées*. N'hésitez pas à jouer avec, en permettant par exemple de forcer l'affichage des produits inactifs.

6

Nous aurions pu améliorer encore notre endpoint, en lui ajoutant un paramètre permettant d'inclure les catégories non disponibles, par exemple. N'hésitez pas à les utiliser, ils sont souvent une solution qui évite la création d'un nouvel endpoint.

Exercice

À votre tour ! Pour parfaire notre boutique, nous souhaitons mettre en place un endpoint de récupération des articles sur l'URL http ://127.0.0.1 :8000/api/article/. Il ne doit retourner que les articles actifs, et permettre de filtrer les articles retournés sur une catégorie avec un paramètre `product_id`. Cet endpoint doit retourner les informations suivantes :

- L'identifiant de l'article ;
- La date de création et de modification de l'article ;
- Le nom de l'article ;
- Le prix de l'article ;
- L'identifiant du produit auquel appartient l'article.

Pour réaliser cela, vous pouvez partir de la branche P1C5_exercice. Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche P1C5_solution.

En résumé

1. Redéfinir la méthode `get_queryset` permet de définir les entités à prendre en compte dans l'endpoint.
2. Il est possible d'utiliser des paramètres d'URL pour apporter des précisions sur l'action à réaliser (comme filtrer sur un critère particulier).

C'est maintenant l'heure d'écrire les premiers tests pour notre API ! Rendez-vous au prochain chapitre.

6 Écrivez des tests pour votre API

6.1 Découvrez le TestCase de DRF

Les tests sont un élément important de tout projet de développement. Ils permettent de garantir la pérennité du projet, sa maintenance, et surtout de déployer la conscience plus tranquille.

Une nouvelle classe de test APITestCase est fournie par DRF. Elle est faite pour fonctionner de la même façon que la classe TestCase. Sa principale différence étant d'utiliser un client qui permet une utilisation plus simple des appels. Il est donc recommandé de l'utiliser pour ne pas avoir à définir plusieurs paramètres pour réaliser nos appels lors des tests.

6.2 Écrivez des tests avec APITestCase

Créons notre fichier de test dans l'application `shop` et mettons en place un test qui *vérifie l'endpoint de récupération des catégories actives*. Nous allons pour cela :

- Créer deux catégories dont une inactive ;
- Réaliser notre appel ;
- Vérifier que le status code de la réponse est bien un succès : 200 ;
- Nous assurer que le contenu de la réponse est bien celui attendu, et qu'il ne comprend pas la catégorie désactivée.

Nous écrirons aussi un test qui s'assurera que la création d'une catégorie n'est pas possible, et tombe bien en erreur.

7

Il est très facile d'oublier d'écrire des tests pour les cas positifs en erreur. Notre API étant pour le moment publique, il est important de tester qu'un utilisateur ne puisse pas créer de nouvelles catégories.

Créons d'abord un fichier `tests.py` pour nos tests. Puis, c'est parti pour nos deux tests :

```
from django.urls import reverse_lazy
from rest_framework.test import APITestCase

from shop.models import Category

class TestCategory(APITestCase):
    # Nous stockons l'url de l'endpoint dans un attribut de classe pour pouvoir
    # l'utiliser plus facilement dans chacun de nos tests
    url = reverse_lazy('category-list')

    def format_datetime(self, value):
        # Cette méthode est un helper permettant de formater une date en chaine de caractères
        # sous le même format que celui de l'api
        return value.strftime("%Y-%m-%dT%H:%M:%S.%fZ")

    def test_list(self):
        # Créons deux catégories dont une seule est active
        category = Category.objects.create(name='Fruits', active=True)
```

```
        Category.objects.create(name='Légumes', active=False)

        # On réalise l'appel en GET en utilisant le client de la classe de test
        response = self.client.get(self.url)
        # Nous vérifions que le status code est bien 200
        # et que les valeurs retournées sont bien celles attendues
        self.assertEqual(response.status_code, 200)
        excepted = [
            {
                'id': category.pk,
                'name': category.name,
                'date_created': self.format_datetime(category.date_created),
                'date_updated': self.format_datetime(category.date_updated),
            }
        ]
        self.assertEqual(excepted, response.json())

    def test_create(self):
        # Nous vérifions qu'aucune catégorie n'existe avant de tenter d'en créer une
        self.assertFalse(Category.objects.exists())
        response = self.client.post(self.url, data={'name': 'Nouvelle catégorie'})
        # Vérifions que le status code est bien en erreur et nous empêche de créer une catégorie
        self.assertEqual(response.status_code, 405)
        # Enfin, vérifions qu'aucune nouvelle catégorie n'a été créée malgré le status code 405
        self.assertFalse(Category.objects.exists())
```

8

Lors d'un développement, il vaut mieux prendre le temps d'écrire les tests au fil de l'eau, ce qui est moins décourageant que de les écrire tous à la fin. Parfois il peut aussi être intéressant de commencer par écrire le test avant de développer la fonctionnalité.

Exercice

Écrivez vous aussi des tests pour l'endpoint de produits. Il peut également être intéressant de créer une classe de test pour notre projet, contenant notre méthode d'aide au formatage de date, qui sera étendu par nos classes de tests.

- Mettez en place une classe `ShopAPITestCase`.
- Refactorisez `TestCategory` pour qu'elle étende notre nouvelle classe `ShopAPITestCase`.
- Écrivez la classe de test `TestProduct` qui va tester :
  - l'endpoint de liste et de détail d'un produit ;
  - que le filtre sur la liste fonctionne correctement ;
  - qu'il n'est pas possible de créer, modifier et supprimer un produit.

Pour réaliser cela, vous pouvez partir de la branche P1C6_exercice. Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche P1C6_solution.

En résumé

1. DRF met à disposition une classe de tests qu'il faut utiliser pour tester une API.

2. Il est important de tester le status code de retour ainsi que le contenu de la réponse.
3. Ne pas oublier de tester les cas d'erreur.
4. Les tests ne sont pas la partie la plus intéressante à écrire, mais ils garantissent la stabilité d'un projet dans le temps.

Nous voilà avec une API DRF simple et validée par nos premiers tests ! Avant de passer à l'étape suivante, validez vos acquis dans le quiz de la partie 1 – je vous attends dans la partie 2 !