

Vue3

Ibrahim ALAME

4 janvier 2024

Chapitre 1

Introduction

Les API sont aujourd’hui incontournables dans le développement. Cependant, les écrire à la main dans des programmes représente un travail important !

Quand il s’agit d’un programme Django, la librairie Django REST Framework est là pour faciliter l’implémentation des API.

Dans ce cours, vous verrez pas à pas comment mettre en place une API avec Django REST Framework. D’abord, vous allez installer ce framework et créer un premier endpoint. Puis, vous ajouterez plus d’endpoints en les optimisant. Enfin, vous implémenterez l’authentification pour sécuriser l’application.

Chapitre 2

Mettez en place une API simple

2.1 Découvrez Django REST Framework

2.1.1 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

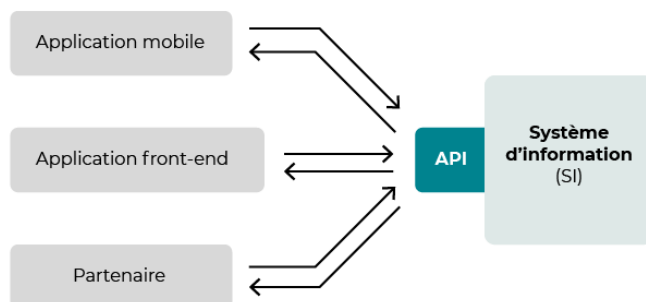
En gros, c'est un peu comme utiliser une classe fournie par une librairie tierce. C'est-à-dire qu'on utilise des méthodes d'objets de cette librairie que nous n'avons pas écrites nous-même. Dans notre cas, la classe est le système d'information, et la méthode est l'endpoint.

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).

2.1.2 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 devons 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.

2.1.3 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 ?

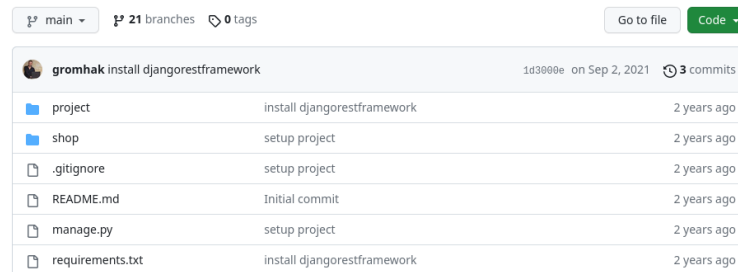
Écrire une API à la main est possible, on pourrait écrire nous-mêmes nos endpoints en nous basant sur les Views de Django, et retourner du JSON.

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.

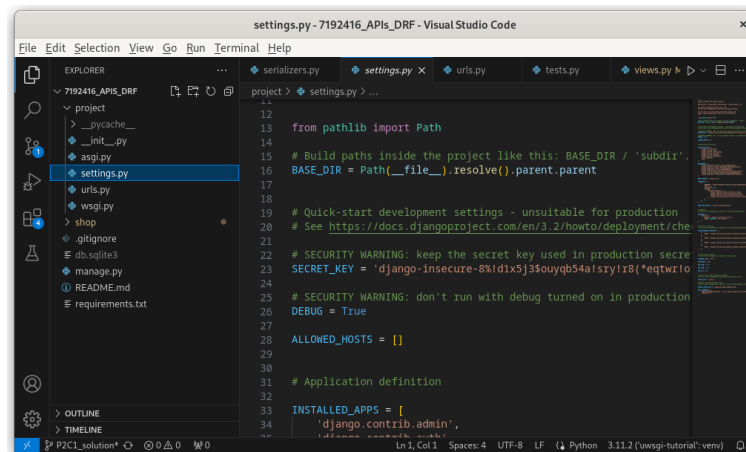
Dans ce chapitre, nous allons faire simple et :

- 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](#),



gromhak install djangoestframework 1d3900e on Sep 2, 2021 3 commits		
project	install djangoestframework	2 years ago
shop	setup project	2 years ago
.gitignore	setup project	2 years ago
README.md	Initial commit	2 years ago
manage.py	setup project	2 years ago
requirements.txt	install djangoestframework	2 years ago



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

```
djangoestframework==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. Éditez 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
```

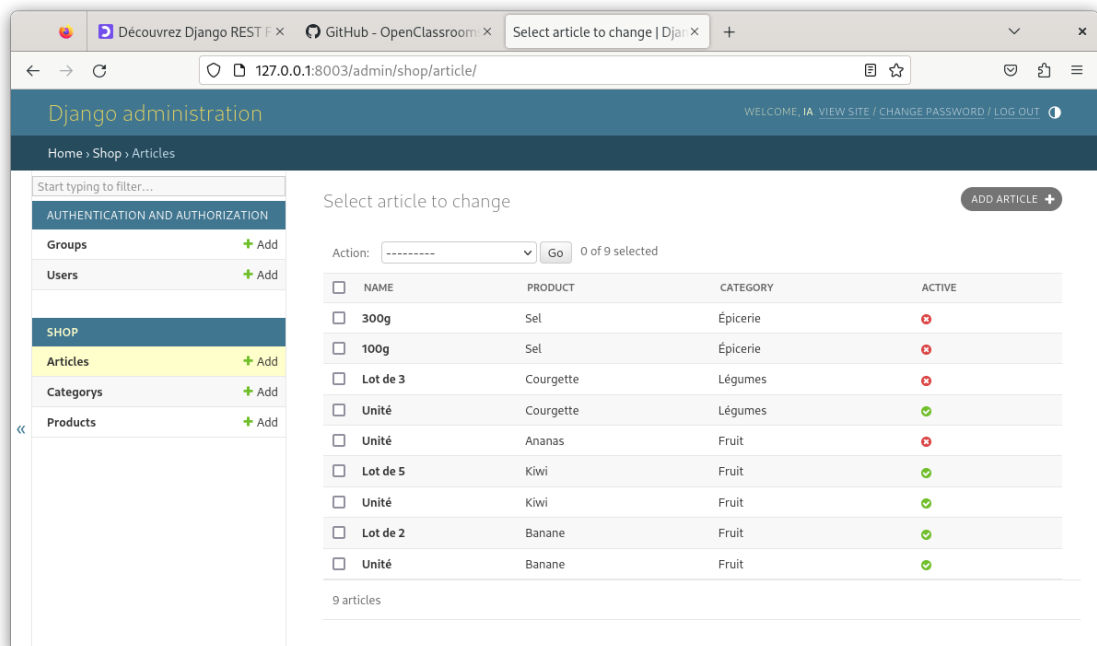
Nous devrions maintenant pouvoir nous rendre sur notre URL et nous connecter. Le projet contient une base de données SQLite qui contient déjà un compte administrateur dont voici les identifiants :

- 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 !

2.2 Gérez des données avec un endpoint

2.2.1 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.

```
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.

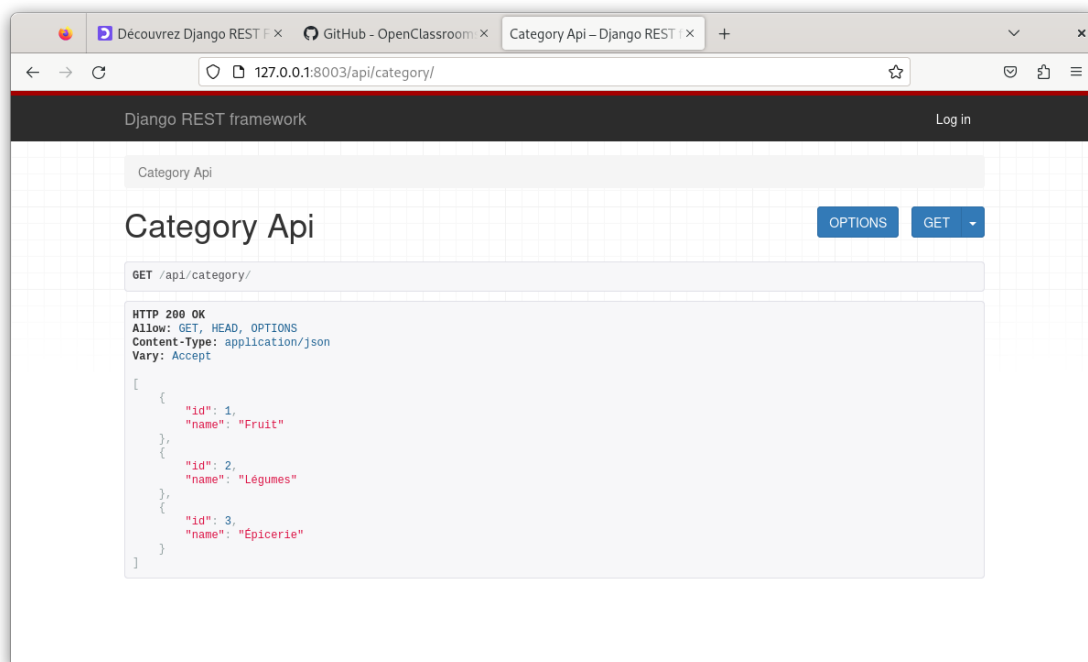
Et si je n'ai qu'un seul élément à sérialiser ?

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** :

```
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 !

2.2.2 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**.

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.

2.3 Rendez les Views plus génériques avec un ModelViewSet

2.4 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.

2.4.1 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.

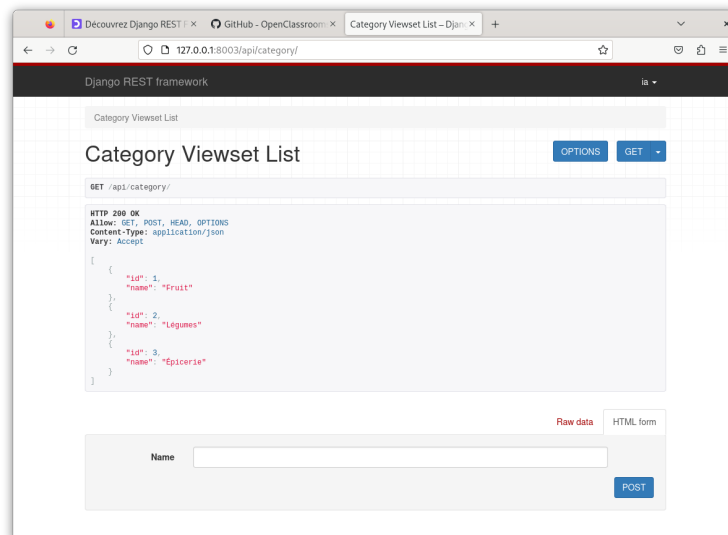
```
from rest_framework.viewsets import ModelViewSet
from shop.models import Category
from shop.serializers import CategorySerializer

class CategoryViewSet(ModelViewSet):

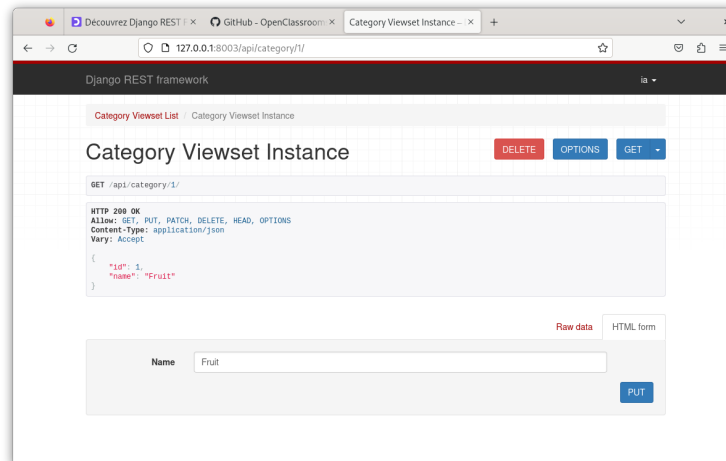
    serializer_class = CategorySerializer

    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 avons 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.



2.4.2 Ne permettez que la lecture

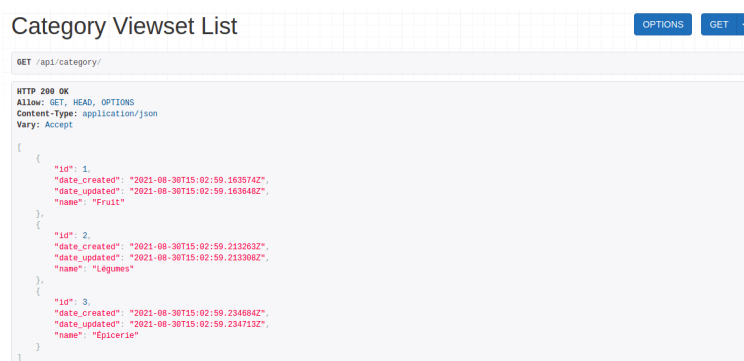
Vous l'aurez sûrement remarqué, nous pouvons créer de nouvelles catégories ; dans la pratique, permettre la création, la modification et la suppression sur un endpoint public comme le nôtre n'est pas conseillé. Nous allons donc faire en sorte que notre endpoint *ne permette que la lecture*, car son but 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 !

2.5 Filtrez les résultats d'un endpoint

2.5.1 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

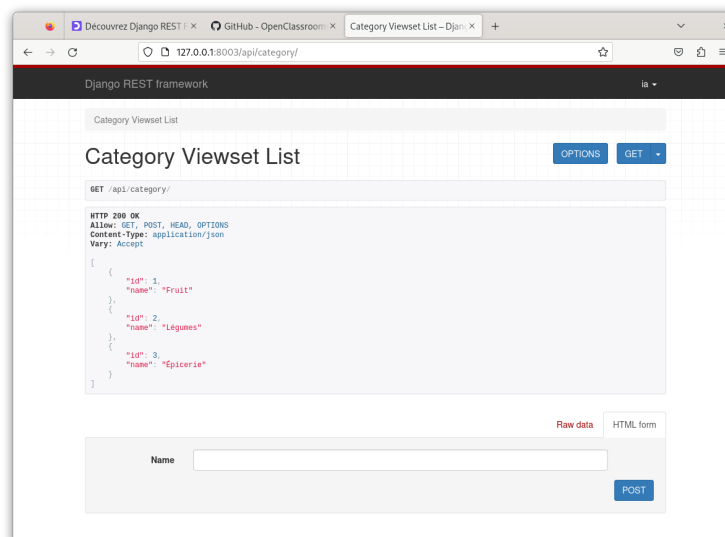
http://127.0.0.1:8000/api/product/?category_id=1.

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 no
        category_id = self.request.GET.get('category_id')
        if category_id is not None:
            queryset = queryset.filter(category_id=category_id)
        return queryset
```

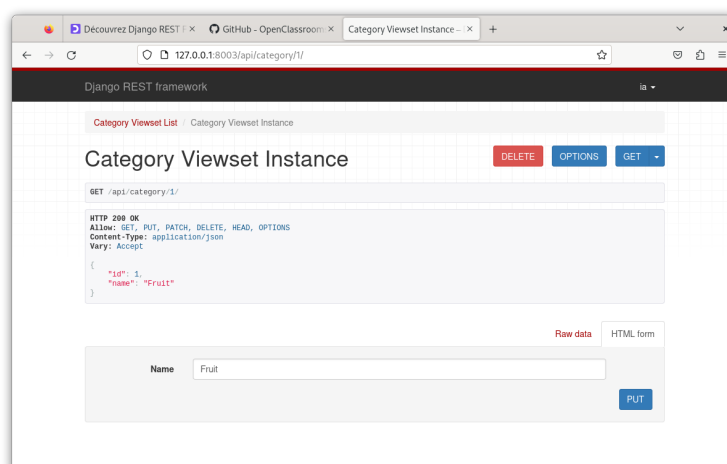
À présent, nous pouvons constater que le filtre est bien appliqué si nous consultons la liste des produits de la catégorie 1 avec l'URL

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.

2.6 Écrivez des tests pour votre API

2.6.1 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.

2.6.2 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 chaîne 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 !

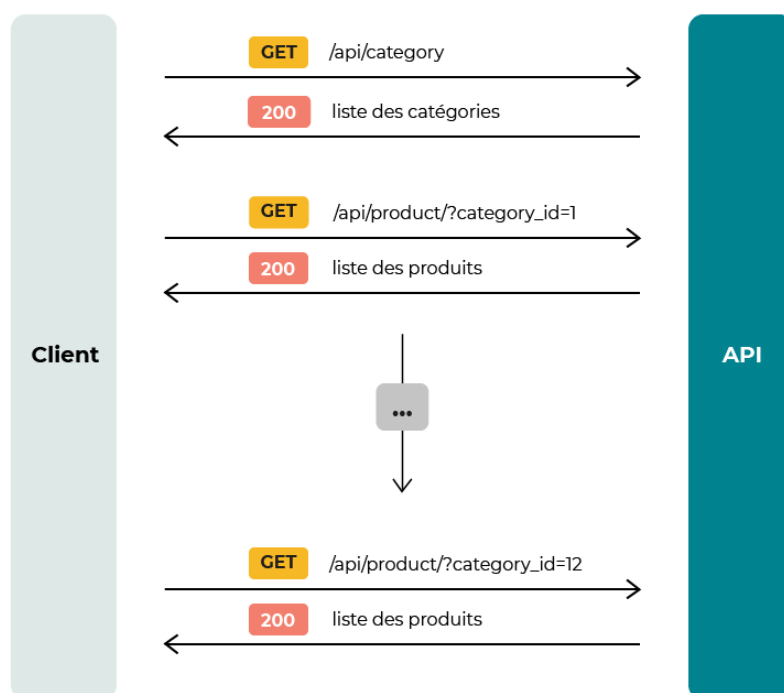
Chapitre 3

Rendez vos endpoints plus performants

3.1 Minimisez les appels de votre API grâce aux serializers

3.1.1 Retournez plus d'informations

En l'état, les clients (application front, mobile, etc.) de notre API doivent réaliser *plusieurs appels* pour obtenir la liste des catégories et la liste des produits qui composent chacune d'entre elles.



Donc autant d'appels qu'il y a de catégories sont nécessaires, ce qui n'est pas très optimal. Pour résoudre cette problématique, il est possible d'*imbriquer des serializers*, afin que le serializer des catégories renvoie directement la liste des produits qui composent la catégorie consultée. Et c'est ce que nous allons voir tout de suite ensemble.

9

Attention tout de même à ne pas imbriquer trop de serializers car beaucoup d'appels en base de données peuvent être faits, à moins d'optimiser ces appels à l'aide de `prefetch_related` et de `select_related`. Certains calculs de valeurs d'attributs peuvent aussi être coûteux, il faut donc utiliser l'imbrication avec parcimonie, selon les cas qui se présentent à nous.

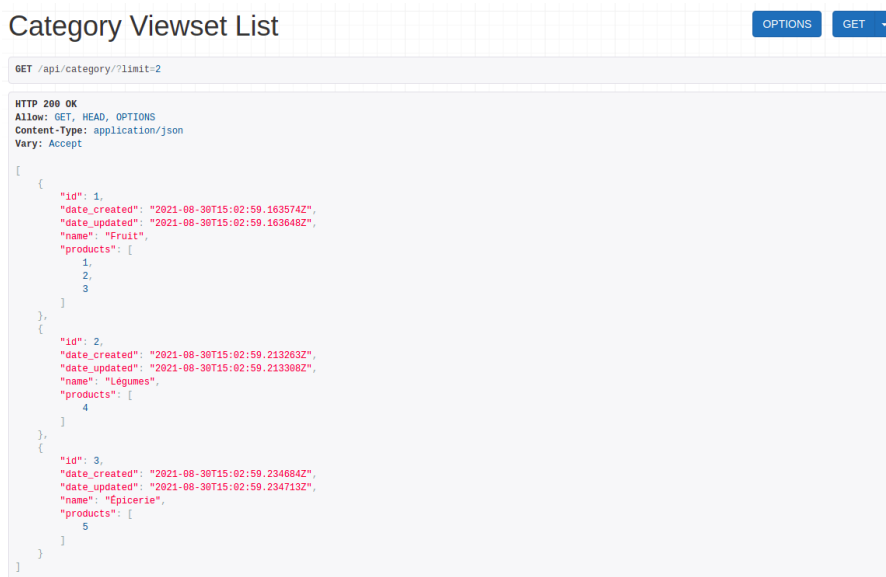
3.1.2 Imbriquez les serializers

Nous allons faire en sorte que notre endpoint de catégorie renvoie également la liste des produits qui le composent. Pour cela, *éditons notre serializer de catégorie* en ajoutant dans la liste des fields le `related_name` vers les produits définis dans notre model :

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

10

Lorsque nous ajoutons un `related_name` dans un serializer, DRF va ajouter la liste des identifiants distants dans un attribut portant le même nom que le `related_name`.



Cependant, un problème persiste, DRF affiche tous les produits de chaque catégorie, alors que nous voudrions n'afficher que ceux qui sont *actifs*. Pour cela, nous pouvons redéfinir notre attribut de classe `products` avec un `SerializerMethodField` qui nous donne alors la possibilité de *filtrer les produits à retourner*.

```

class CategorySerializer(serializers.ModelSerializer):

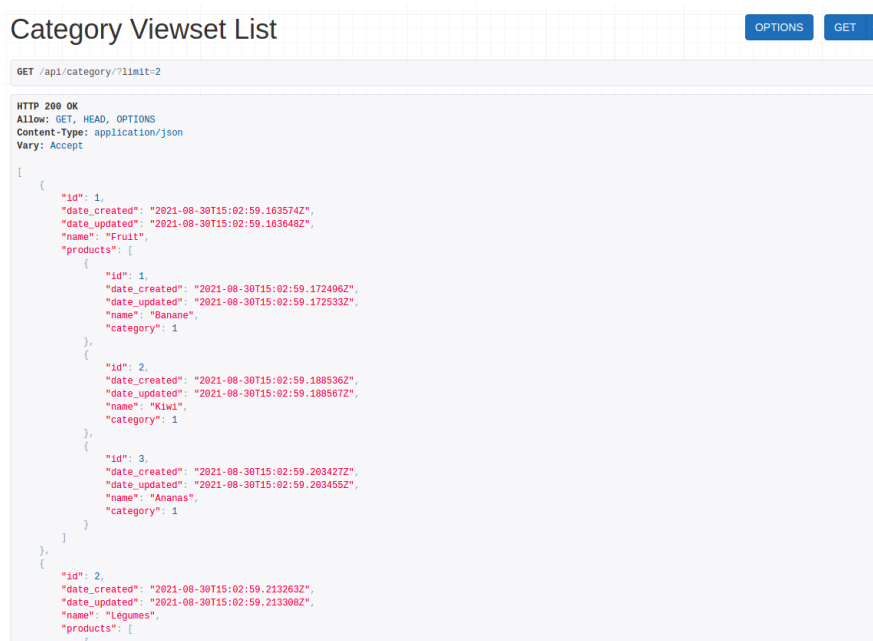
    # En utilisant un `SerializerMethodField`, il est nécessaire d'écrire une méthode
    # nommée 'get_XXX' où XXX est le nom de l'attribut, ici 'products'
    products = serializers.SerializerMethodField()

    class Meta:
        model = Category
        fields = ['id', 'date_created', 'date_updated', 'name', 'products']

    def get_products(self, instance):
        # Le paramètre 'instance' est l'instance de la catégorie consultée.
        # Dans le cas d'une liste, cette méthode est appelée autant de fois qu'il y a
        # d'entités dans la liste

        # On applique le filtre sur notre queryset pour n'avoir que les produits actifs
        queryset = instance.products.filter(active=True)
        # Le serializer est créé avec le queryset défini et toujours défini en tant que many=True
        serializer = ProductSerializer(queryset, many=True)
        # la propriété '.data' est le rendu de notre serializer que nous retournons ici
        return serializer.data

```



Nous pouvons constater qu'à présent les produits retournés sont bien seulement des produits actifs.

Et si on lançait nos tests après le développement de cette feature ?

Oups, c'est cassé ! Et c'est tout à fait **normal**.

Une API se doit d'être minutieusement **testée**, car les retours de ces endpoints sont souvent garants du bon fonctionnement des applications clientes. L'ajout d'un attribut dans un endpoint ne pose en général aucun problème, mais le **retrait** d'un attribut peut avoir des conséquences plus importantes.

Imaginons que l'application mobile de notre site utilise l'attribut **price** de nos articles, et que nous décidions de le **retirer** pour le déplacer ailleurs. Les applications mobiles ne pourraient pas alors afficher de prix tant qu'ils ne déploient pas eux-mêmes une nouvelle version de leur application qui utilise le **nouvel attribut**.

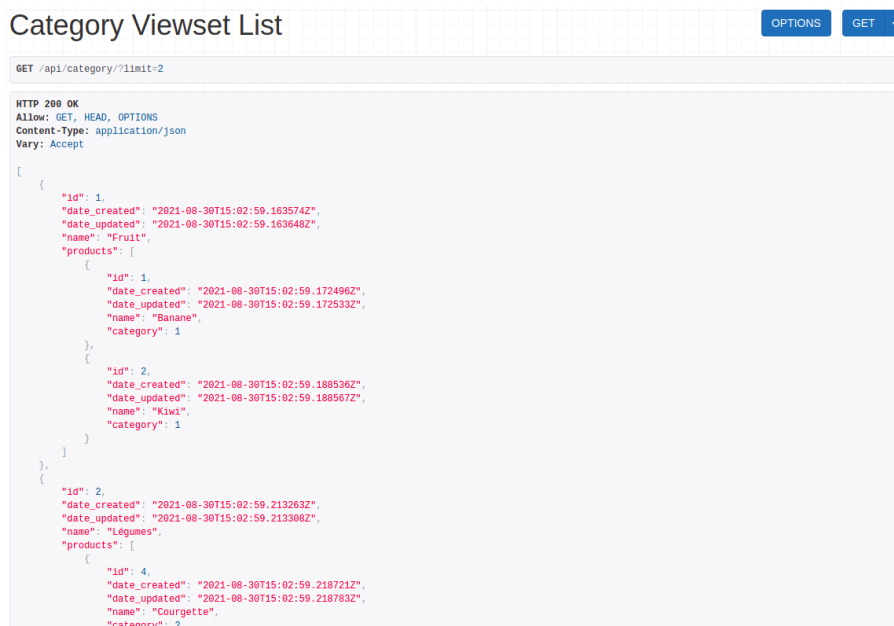
Mais ça fait beaucoup de données tout ça pour un seul endpoint, non ?

Pour limiter les données retournées, il est possible de mettre en place une **pagination**. Faisons-le ensemble !

3.1.3 Ajoutez de la pagination

Mettre en place une pagination dès la création d'une API est une bonne pratique, car en limitant le nombre d'entités retournées, cela permet :

- De réduire le temps de réponse, surtout si le calcul de certains attributs est coûteux ;
- Aux applications clientes de ne pas récupérer toutes les informations si elles ne se servent que d'une seule partie ;
- D'éviter la modification des applications clientes par la suite, car la pagination impose certains attributs.



La pagination indique les informations suivantes :

- **count** : le nombre total d'éléments ;
- **next** : l'URL de l'endpoint pour obtenir la page suivante ;
- **previous** : l'URL de l'endpoint pour obtenir la page précédente ;
- **results** : les données réelles utilisables.

11

Lorsque les applications clientes sont réalisées en interne, il est important de bien communiquer avec ses équipes afin de savoir quelles sont les données qui leurs sont importantes, et dans quelles conditions. Le résultat de ces échanges vous donnera les meilleures pistes pour savoir quelles données retourner, et sous quel format.

Je vous propose de mettre en place le même mécanisme d'imbrication sur le serializer de produits, pour que les applications clientes puissent récupérer d'un coup les produits et leurs articles actifs. Pour réaliser cela, vous pouvez partir de la branche [P2C1_exercice](#). Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche [P2C1_solution](#).

```
from rest_framework import serializers
from shop.models import Category, Product, Article

class ArticleSerializer(serializers.ModelSerializer):
    class Meta:
        model = Article
        fields = ['id', 'date_created', 'date_updated', 'name', 'description', 'active', 'price']

class ProductSerializer(serializers.ModelSerializer):
    articles = serializers.SerializerMethodField()
```

```

class Meta:
    model = Product
    fields = ['id', 'date_created', 'date_updated', 'name', 'category', 'articles']

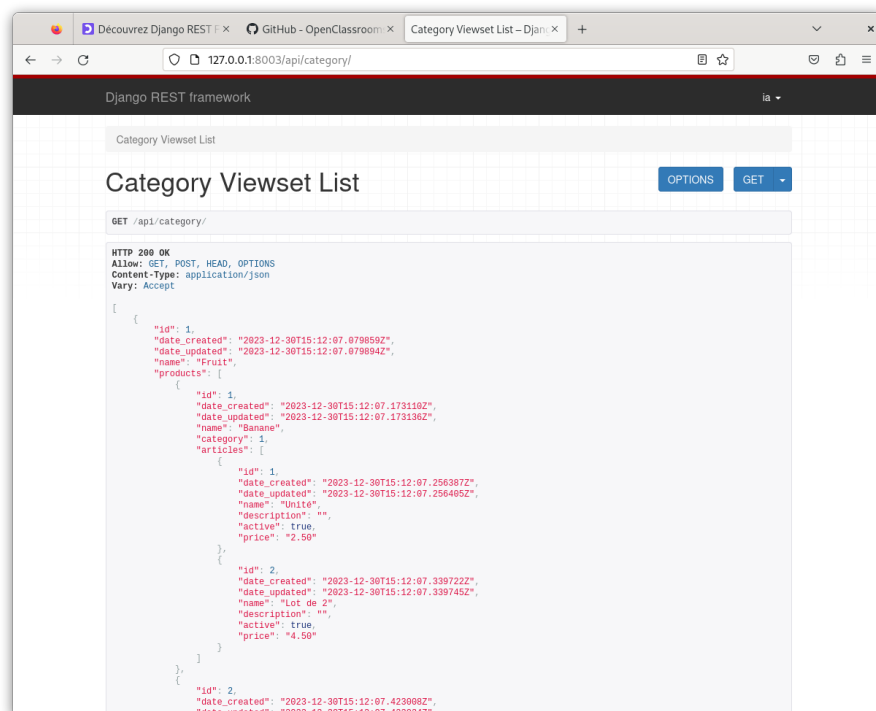
def get_articles(self, instance):
    queryset = instance.articles.filter(active=True)
    serializer = ArticleSerializer(queryset, many=True)
    return serializer.data

class CategorySerializer(serializers.ModelSerializer):
    products = serializers.SerializerMethodField()

    class Meta:
        model = Category
        fields = ['id', 'date_created', 'date_updated', 'name', 'products']

    def get_products(self, instance):
        queryset = instance.products.filter(active=True)
        serializer = ProductSerializer(queryset, many=True)
        return serializer.data

```



En résumé

1. Imbriquer des serializers permet d'obtenir plus d'informations en un seul appel.
2. Il est possible d'appliquer des filtres sur un attribut du serializer en utilisant un `SerializerMethodField`.
3. Toute modification d'un endpoint entraîne l'adaptation d'un test.
4. Il est bien de rapidement mettre en place une pagination sur une API.

Maintenant que nous avons optimisé le nombre d'appels de notre API, voyons comment différencier les informations retournées en liste ou en détail – vous me suivez au prochain chapitre ? C'est parti !

3.2 Différenciez les informations de liste et de détail

3.2.1 Retournez plus d'informations dans le détail

Très souvent, les données en retour sont différentes selon qu'on consulte un endpoint de liste ou un endpoint de détail. En règle générale, les listes sont appelées pour être affichées. Lorsque l'utilisateur sélectionne un des éléments qui la composent, alors un autre appel est réalisé par l'application cliente afin d'obtenir plus d'informations. Cela permet de *réduire les temps de réponse* en ne retournant que les informations utiles.

Dans notre cas, la liste des catégories est bien trop complète pour l'usage qui doit en être fait, et nous allons donc réduire les informations de liste, en conservant toutes nos données actuelles dans l'endpoint de détail d'une catégorie.

3.2.2 Améliorez le rendu du détail d'un endpoint

Améliorons notre endpoint de catégories en ne retournant que les informations *minimales*, dans le cas d'une liste, et *détaillées*, lorsque nous consultons une catégorie spécifique.

DRF nous permet au travers de ses viewsets de redéfinir la méthode `get_serializer_class` qui, elle, détermine le serializer à utiliser. Par défaut, le serializer retourné est celui défini sur l'attribut de classe `serializer_class` du viewset.

Lorsqu'une requête entre dans notre API, les viewsets définissent un attribut `action` qui correspond à l'action que l'application cliente est en train de réaliser. Elle peut être :

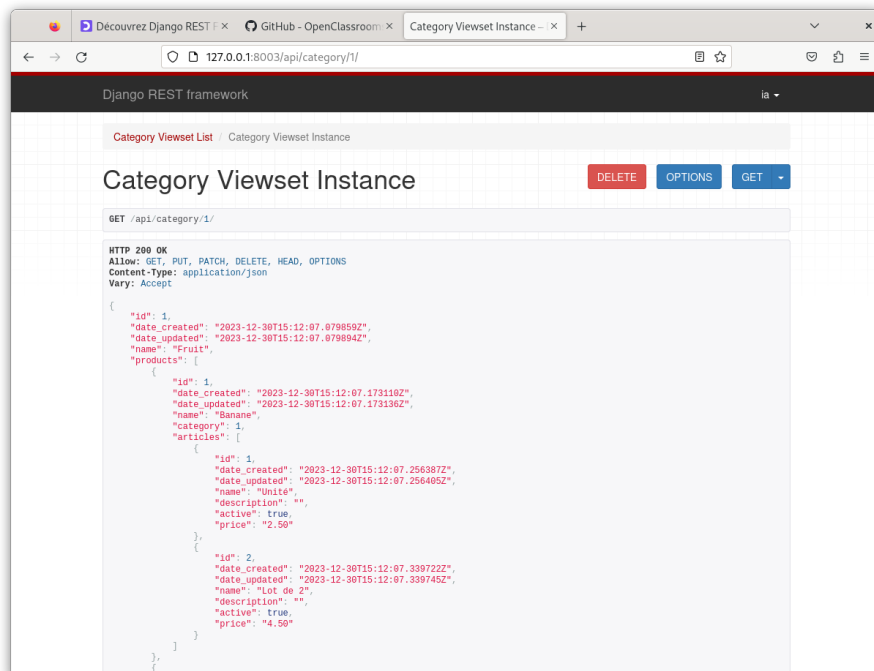
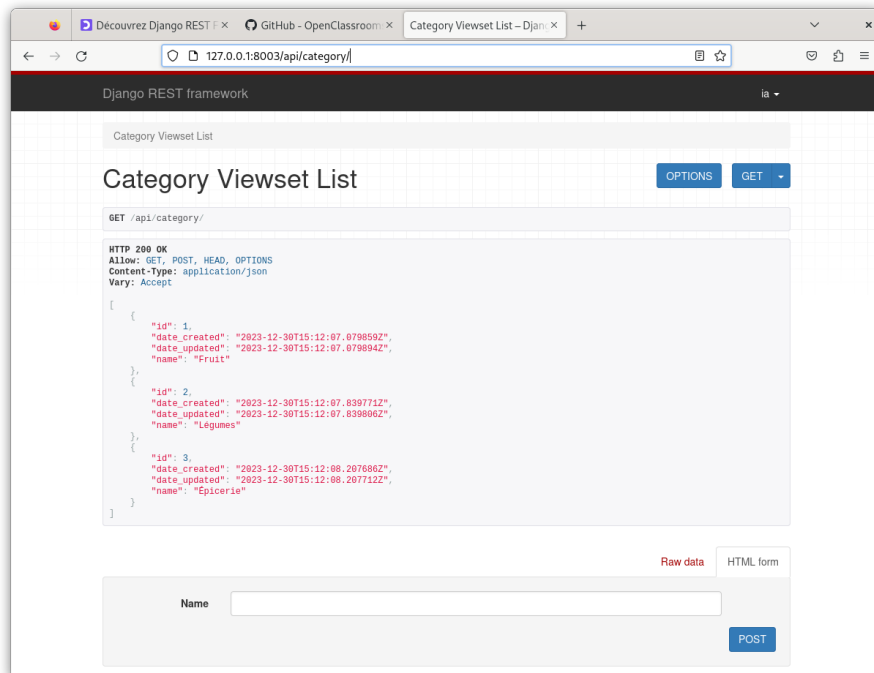
- `list` : appel en GET sur l'URL de liste ;
- `retrieve` : appel en GET sur l'URL de détail (qui comporte alors un identifiant) ;
- `create` : appel en POST sur l'URL de liste ;
- `update` : appel en PUT sur l'URL de détail ;
- `partial_update` : appel en PATCH sur l'URL de détail ;
- `destroy` : appel en DELETE sur l'URL de détail.

```
class CategoryViewSet(ReadOnlyModelViewSet):

    serializer_class = CategoryListSerializer
    # Ajoutons un attribut de classe qui nous permet de définir notre serializer de détail
    detail_serializer_class = CategoryDetailSerializer

    def get_queryset(self):
        return Category.objects.filter(active=True)

    def get_serializer_class(self):
        # Si l'action demandée est retrieve nous retournons le serializer de détail
        if self.action == 'retrieve':
            return self.detail_serializer_class
        return super().get_serializer_class()
```



Exemple

À vous à présent, faites de même et améliorez l'endpoint de produits pour que la liste reste succincte, et que le détail retourne plus d'informations. Pour réaliser cela, vous pouvez partir de la branche [P2C2_exercice](#). Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche [P2C2_solution](#).

En résumé

1. L'action réalisée est définie dans l'attribut `action` des viewsets, et permet de savoir précisément l'action en cours.
2. Il est possible de définir un serializer différent pour chaque action.
3. Les mixins sont une bonne façon de gérer les différents serializers à utiliser, car cette opération est très courante.
4. Utiliser différents serializers contribue à améliorer les performances de l'API.

Dissocier les serializers de liste et de détail nous permettrait par exemple d'inclure les produits et articles dans le détail d'une catégorie. N'hésitez pas à expérimenter.

3.3 Ajoutez de l'interaction avec les actions

3.3.1 Ne soyez pas trop actif!

Une action, on a déjà parlé de ça, non ?

Nous avons parlé du paramètre `action` des Viewsets, mais dans ce chapitre nous allons voir le décorateur `action`, fourni par DRF, qui permet de réaliser d'autres types d'actions que les classiques du CRUD, comme *Demander en ami*, *S'abonner à un fil d'actualité* ou *Publier un article*.

Vous devez penser `Action` chaque fois qu'un besoin fait référence à une entité, mais que le verbe ne correspond pas à un élément du CRUD. Par exemple, dans *Nous souhaitons que nos visiteurs puissent liker des publications*, l'entité est la publication et l'action est liker.

Une action se crée dans DRF en mettant en place le décorateur `action` sur une méthode d'un Viewset. Les paramètres suivants sont disponibles :

- `methods` est la liste des méthodes HTTP qui appellent cette action, parmi GET, POST, PATCH, PUT, DELETE.
- `detail` est un booléen qui précise si l'action est disponible sur l'URL de liste ou de détail.
- `url_path` permet de déterminer l'URL qui sera ajoutée à la fin de l'endpoint de liste ou de détail. S'il n'est pas précisé, alors le nom de la méthode est utilisé.

Pour notre boutique en ligne, on pourrait imaginer une action qui permette d'activer ou de désactiver une catégorie.

Mais on ne pourrait pas juste faire un PATCH sur la catégorie pour mettre active à False ?

On pourrait effectivement, mais nous aimerions également désactiver tous les produits qui composent cette catégorie. Plutôt que de laisser les applications clientes faire tous ces appels, mettons-leur à disposition un seul endpoint qui réalise cela. D'ailleurs, vous savez quoi ? On va le faire tout de suite... ;)

3.3.2 Soyez actif quand même !

Les actions se mettent en place sur les Viewsets, alors allons modifier notre `CategoryViewSet` pour lui ajouter une action `disable` que nous souhaitons accessible en POST.

```
@transaction.atomic
@action(detail=True, methods=['post'])
def disable(self, request, pk):
    # Nous avons défini notre action accessible sur la méthode POST seulement
    # elle concerne le détail car permet de désactiver une catégorie

    # Nous avons également mis en place une transaction atomique car plusieurs requêtes vont être exécutées
    # en cas d'erreur, nous retrouverions alors l'état précédent

    # Désactivons la catégorie
    category = self.get_object()
    category.active = False
    category.save()

    # Puis désactivons les produits de cette catégorie
    category.products.update(active=False)

    # Retournons enfin une réponse (status_code=200 par défaut) pour indiquer le succès de l'action
    return Response()
```

Ainsi, la réalisation d'un POST sur l'endpoint `aura` pour effet de :

1. Désactiver la **catégorie**, ce qui ne rendra plus visible la catégorie sur l'endpoint `http://127.0.0.1:8000/api/ca`
2. Désactiver **les produits de cette catégorie**, ce qui ne rendra plus visible ces produits sur l'endpoint `http://127.0.0.1:8000/api/product/`.

12

Il est toujours préférable de rapprocher le code métier du model afin de pouvoir l'exécuter plus facilement. Faisons cela, et notre action en sera alors simplifiée.

Ajoutons une méthode `disable` à notre model `Category` :

```
class Category(models.Model):

    date_created = models.DateTimeField(auto_now_add=True)
    date_updated = models.DateTimeField(auto_now=True)

    name = models.CharField(max_length=255)
    active = models.BooleanField(default=False)

    @transaction.atomic
    def disable(self):
        if self.active is False:
            # Ne faisons rien si la catégorie est déjà désactivée
            return
```



```
self.active = False
self.save()
self.products.update(active=False)
```

13

Vous remarquerez que nous avons déplacé la transaction atomique sur la méthode du model, elle n'est donc plus nécessaire sur l'action du Viewset.

```
class MultipleSerializerMixin:
    # Un mixin est une classe qui ne fonctionne pas de façon autonome
    # Elle permet d'ajouter des fonctionnalités aux classes qui les étendent

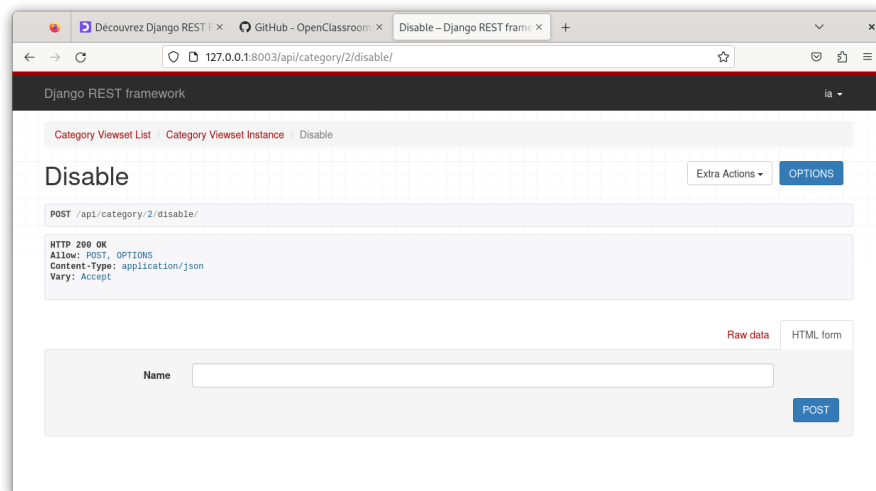
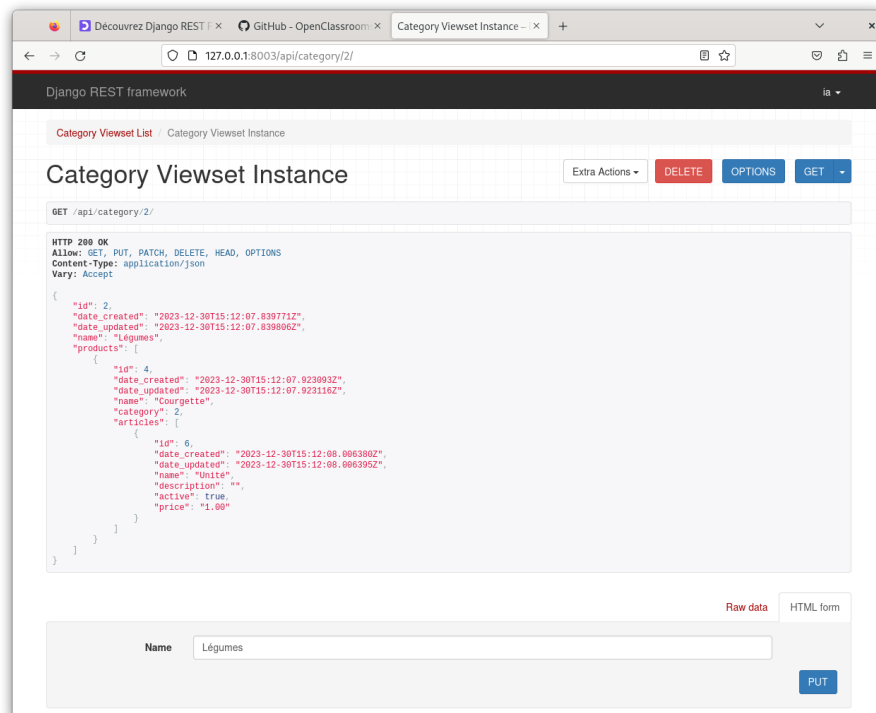
    detail_serializer_class = None

    def get_serializer_class(self):
        # Notre mixin détermine quel serializer à utiliser
        # même si elle ne sait pas ce que c'est ni comment l'utiliser
        if self.action == 'retrieve' and self.detail_serializer_class is not None:
            # Si l'action demandée est le détail alors nous retournons le serializer de détail
            return self.detail_serializer_class
        return super().get_serializer_class()

class CategoryViewSet(MultipleSerializerMixin, ReadOnlyModelViewSet):

    serializer_class = CategoryListSerializer
    detail_serializer_class = CategoryDetailSerializer

    @action(detail=True, methods=['post'])
    def disable(self, request, pk):
        # Nous pouvons maintenant simplement appeler la méthode disable
        self.get_object().disable()
        return Response()
```



14

Cette modification ne change en rien le fonctionnement de notre API, mais permet une **lecture** plus claire du code. L'action ne fait qu'appeler une méthode de notre model `Category`.

Exercice

Mettons en place le même système pour **désactiver un produit** qui **désactive également les articles associés**. Nous pourrions également améliorer la désactivation d'une catégorie en désactivant les articles de chaque produit.

Pour réaliser cela, vous pouvez partir de la branche [P2C3_exercice](#). Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche [P2C3_solution](#).

En résumé

1. Il est possible de créer d'autres actions en dehors de celles du CRUD.
2. DRF met à disposition un décorateur `action` qui permet de créer de nouvelles actions.
3. Les actions peuvent être mises en place sur les URL de liste et de détail d'un endpoint.
4. Les actions peuvent utiliser n'importe quelle méthode HTTP (GET, POST, PATCH, DELETE...).

Les actions servent à gérer des morceaux de logique métier, et ainsi à éviter aux applications clientes de les gérer, car elles pourraient les gérer différemment. Le fait d'ajouter un article au panier d'un utilisateur est un bon exemple d'action qui ne peut être réalisé que d'une seule façon, son traitement doit donc être réalisé par le serveur et non les applications clientes. Dans le chapitre suivant, vous découvrirez comment valider les données transmises par ces actions !

3.4 Validez les données

3.4.1 Ne permettez pas la création de tout et n'importe quoi

Attaquons-nous à présent à un autre sujet : **la création d'entités** et plus précisément de catégories, dans notre cas.

Notre endpoint actuel permet la lecture seulement, car il est destiné aux visiteurs de notre boutique. Nous allons très vite en mettre un second en place, qui sera dédié aux **administrateurs** qui, eux, auront la possibilité de **créer, modifier et supprimer** des données. La création d'entité impose d'effectuer certains contrôles qui sont généralement de deux types :

1. Les contrôles **sur un champ**, comme par exemple vérifier que le nom d'une catégorie n'existe pas déjà, et ainsi éviter les doublons.
2. Les contrôles **multichamps**, comme la vérification que les deux mots de passe saisis à l'inscription sont les mêmes.

DRF nous permet de réaliser ces deux types de contrôles au travers de la réécriture de méthodes sur le `serializer` :

- `validate_XXX` où XXX est le nom du champ à valider ;
- `validate` qui permet un contrôle global sur tous les champs du `serializer`.

N'attendons pas plus longtemps pour mettre en place l'endpoint et les contrôles qui l'accompagnent. ;)

3.4.2 Validez les données d'un champ

Commençons tout de suite par la mise en place du nouvel endpoint sur l'URL [q](#), comme son nom l'indique, servira à administrer les catégories.

Pourquoi ne pas utiliser l'endpoint déjà existant ?

Dans les endpoints d'administration :

- Des serializers différents sont utilisés et les données retournées diffèrent ;
- Certains accès peuvent également être limités à certaines personnes authentifiées.

Dans le cadre de notre boutique, nous avons décidé de **définir nos endpoints** en fonction des **acteurs** qui les utilisent. Créons **notre nouvel endpoint d'administration** qui cette fois-ci étend `ModelViewSet` et non plus `ReadOnlyViewSet`. Celui-ci ne doit pas avoir de limitation sur les catégories actives, car il s'agit d'un endpoint d'administration.

```
class AdminCategoryViewSet(MultipleSerializerMixin, ModelViewSet):

    serializer_class = CategoryListSerializer
    detail_serializer_class = CategoryDetailSerializer

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

Puis définissons notre nouvel endpoint en le **déclarant auprès de notre routeur**.

```
router.register('admin/category', AdminCategoryViewSet, basename='admin-category')
```

Vérifions que notre endpoint est bien fonctionnel sur l'URL

<http://127.0.0.1/api/admin/category>

Maintenant que nous utilisons un `ViewSet`, **la création de catégorie est possible**. C'est à la création que sert ce formulaire en nous permettant d'effectuer des actions POST. Les actions de mise à jour et de suppression sont disponibles sur les URL de détail des catégories.

Validons à présent nos données, sans oublier que la création d'un doublon de catégorie ne doit pas être permis. Il nous faut pour cela modifier notre serializer de liste, car c'est lui qui est utilisé pour l'action create .

La **validation d'un champ** unique se fait en écrivant la méthode `validate_XXX` où XXX est le nom du champ. Dans notre cas, `validate_name` :

```
class CategoryListSerializer(serializers.ModelSerializer):

    class Meta:
        model = Category
        fields = ['id', 'date_created', 'date_updated', 'name']

    def validate_name(self, value):
        # Nous vérifions que la catégorie existe
        if Category.objects.filter(name=value).exists():
            # En cas d'erreur, DRF nous met à disposition l'exception ValidationError
            raise serializers.ValidationError('Category already exists')
        return value
```

Si nous tentons à présent de créer une catégorie qui existe déjà, une réponse en 400 avec des données contenant la nature de l'erreur sont renvoyées.

La validation de champ unique permet d'effectuer plein de **contrôles**, tant qu'ils sont effectués sur ce champ précis. On pourrait imaginer un système de filtre de mots pour un forum, par exemple.

3.4.3 Mettez en place une validation multiple

Pour notre boutique, nous souhaitons optimiser le référencement et avoir un rappel du nom de la catégorie également présent dans la description. Nous pouvons effectuer automatiquement ce contrôle au travers d'une validation multiple.

La validation entre champs se fait au travers de la méthode `validate`. Vérifions que le nom est bien présent dans la description :

```
class CategoryListSerializer(serializers.ModelSerializer):

    class Meta:
        model = Category
        # Pensons à ajouter << description >> à notre liste de champs
        fields = ['id', 'date_created', 'date_updated', 'name', 'description']

    def validate_name(self, value):
        if Category.objects.filter(name=value).exists():
            raise serializers.ValidationError('Category already exists')
        return value

    def validate(self, data):
        # Effectuons le contrôle sur la présence du nom dans la description
        if data['name'] not in data['description']:
            # Levons une ValidationError si ça n'est pas le cas
            raise serializers.ValidationError('Name must be in description')
        return data
```

Nous pouvons alors constater que notre validation fonctionne si le nom de la catégorie n'est pas présent dans sa description.

The screenshot shows the Django REST framework interface for the 'Admin Category Viewset List'. At the top, there's a header with 'Django REST framework' and a 'Log in' link. Below the header, the title 'Admin Category Viewset List' is displayed with 'OPTIONS' and 'GET' buttons. The main content area shows a POST request to '/api/admin/category/' with an 'HTTP 400 Bad Request' error. The error details are: 'Allow: GET, POST, HEAD, OPTIONS', 'Content-Type: application/json', 'Vary: Accept', and a JSON response: {'non_field_errors': [{'Name must be in description'}]}. At the bottom, there's a form with two fields: 'Name' (containing 'Nouvelle catégorie') and 'Description' (containing 'Description de la catégorie'). There are 'Raw data' and 'HTML form' tabs, and a 'POST' button.

Exercice

Mettons en place un endpoint d'administration des articles pour les administrateurs de la boutique. Certains contrôles doivent être effectués :

- Le **prix** doit être supérieur à 1€.
- Le produit associé doit être **actif**.

Pour réaliser cela, vous pouvez partir de la branche [P2C4_exercice](#). Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche [P2C4_solution](#).

En résumé

1. Utiliser un ModelViewSet permet l'utilisation de l'ensemble des actions du CRUD.
2. La validation d'un champ se fait au travers de la méthode `validate_XXX` du serializer.
3. La validation multichamp se fait au travers de la méthode `validate` du serializer.

La validation des données lors de la création et/ou la modification est un facteur clé pour assurer la cohérence des données, n'hésitez pas à valider toutes les données importantes. Maintenant, voyons comment tester les API externes avec les mocks – suivez-moi au prochain chapitre !

Chapitre 4

Sécurisez votre API avec l'authentification

4.1 Ajoutez l'authentification des utilisateurs

4.1.1 Découvrez JSON Web Token

L'authentification permet de rendre certains endpoints **privés** et accessibles seulement aux utilisateurs authentifiés. Ce qui est bien le cas pour notre boutique en ligne !

Nous souhaitons que les endpoints utiles aux visiteurs soient publics, cependant nous souhaitons disposer d'**endpoints d'administration** auxquels seuls les administrateurs de la plateforme pourront avoir accès. Nous ne souhaitons pas permettre aux visiteurs de modifier le prix d'un article avant de l'acheter, par exemple.

Nous allons donc mettre en place un système d'authentification basé sur JWT et la librairie [Simple JWT](#) qui est préconisée par DRF.

15

Il existe divers modes d'authentification, ceux préconisés par DRF sont disponibles sur sa [documentation](#) (en anglais).

JWT est le sigle de *JSON Web Token*, qui est un **jeton d'identification** communiqué entre un serveur et un client. Il permet l'échange de données sécurisées entre ces derniers.

Dans le cadre de l'authentification, le JWT est utilisé pour s'assurer de l'identité de la personne réalisant la requête. Lorsque le serveur reçoit une requête, il vérifie alors la validité du token et détermine l'utilisateur à l'initiative de la requête. Le JWT permet d'identifier l'**utilisateur** à l'origine de la requête et permet ainsi de **vérifier ses droits**.

Ça ressemble à quoi, un JWT ?

Un JWT est constitué de 3 parties séparées par un point. Par exemple ce JWT :

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ0b2t1b190eXB1IjoicmVmcmVzaCIzMV4cCI6MTYyODk0ODEONSwianRpIjojODBmZTA4MDkxM2UxNDBjYmEwMDU4YWY4YmM5Nj1lZjYiLCJ1c2VyX2lkIjoxfQ.131cs5pTApiR9R9s8pZaeyHIJuTmjcs07fxSqSpj1fQ
```

Chaque partie est encodée en base64, il est possible d'utiliser jwt.io pour les déchiffrer :

- Le **header**, qui est en général constitué de deux attributs, indique le type de token et l'algorithme de chiffrement utilisé. Une fois décodé :

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

- Le **payload** contient les informations utiles que nous souhaitons faire transiter entre le serveur et le client. Une fois décodé :

```
{
  "token_type": "refresh",
  "exp": 1628948145,
  "jti": "80fe080913e140cba0058af8bc969ef6",
  "user_id": 1
}
```

- La **signature** est un élément de sécurité permettant de vérifier que les données n'ont pas été modifiées entre les échanges client-serveur. La clé permettant la génération de cette signature est stockée sur le serveur qui fournit le JWT (elle est basée sur la `SECRET_KEY` de Django).

La librairie Simple JWT va nous permettre d'authentifier nos utilisateurs et de leur fournir une paire de JWT :

- Un **access_token** qui va permettre de vérifier l'identité et les droits de l'utilisateur. Sa durée de vie est limitée dans le temps ;
- Un **refresh_token** qui va permettre d'obtenir une nouvelle paire de tokens une fois que l' **access_token** sera expiré.

Deux endpoints vont donc être mis à disposition par `django-rest-framework-simplejwt` :

- Un endpoint d'authentification ;
- Un endpoint de rafraîchissement de token.

4.1.2 Installez et configurez `django-rest-framework-simple-jwt`

Prêt ? Eh bien on est parti ! Commençons sans attendre l'installation et la configuration de `django-rest-framework-simple-jwt` dans notre projet de boutique en ligne. Commençons comme d'habitude par ajouter la dépendance au fichier `requirements.txt`.

```
django-rest-framework-simplejwt==4.7.2
```

Puis, installons cette nouvelle dépendance avec la commande

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

Ajoutons la librairie dans les applications Django, et paramétrons DRF pour qu'il utilise notre librairie en tant que classe d'authentification.

16

Pour rappel, une classe d'authentification dans Django permet de définir l'utilisateur à l'origine de la requête. C'est elle qui attache le user à la requête avec l'attribut `request.user` si l'utilisateur a prouvé son authentification.

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    'rest_framework',  
    'rest_framework_simplejwt',  
    'shop',  
]  
  
REST_FRAMEWORK = {  
    'DEFAULT_PAGINATION_CLASS': 'rest_framework.pagination.LimitOffsetPagination',  
    'PAGE_SIZE': 100,  
    'DEFAULT_AUTHENTICATION_CLASSES': ('rest_framework_simplejwt.authentication.JWTAuthentication',)  
}
```

Il ne nous reste plus qu'à définir nos URL d'obtention et de rafraîchissement de tokens dans `urls.py` en les ajoutant à `urlpatterns`.

```
urlpatterns = [  
    path('admin/', admin.site.urls),  
    path('api-auth/', include('rest_framework.urls')),  
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),  
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),  
    path('api/', include(router.urls))  
]
```

L'installation et la configuration sont terminées, nous allons maintenant pouvoir jeter un œil à nos deux nouveaux endpoints.

17

Après avoir suivi ces étapes pour installer et configurer Simple JWT, vous pouvez retrouver la version du projet qui contient l'installation et la configuration de Simple JWT dans la branche [P3C1-C2](#).

En résumé

1. Les JWT permettent de transférer des informations du client au serveur en plus des jetons d'authentification.
2. `django-rest-framework-simplejwt` est une librairie permettant de gérer l'authentification, mais il en existe d'autres également conseillées par DRF.
3. Deux nouveaux endpoints d'obtention et de rafraîchissement de tokens fournis par Simple JWT permettent de gérer l'authentification de nos utilisateurs.

Maintenant que nous avons installé et configuré Simple JWT, créons des accès pour nos utilisateurs. Quand vous serez prêt, suivez-moi au prochain chapitre !

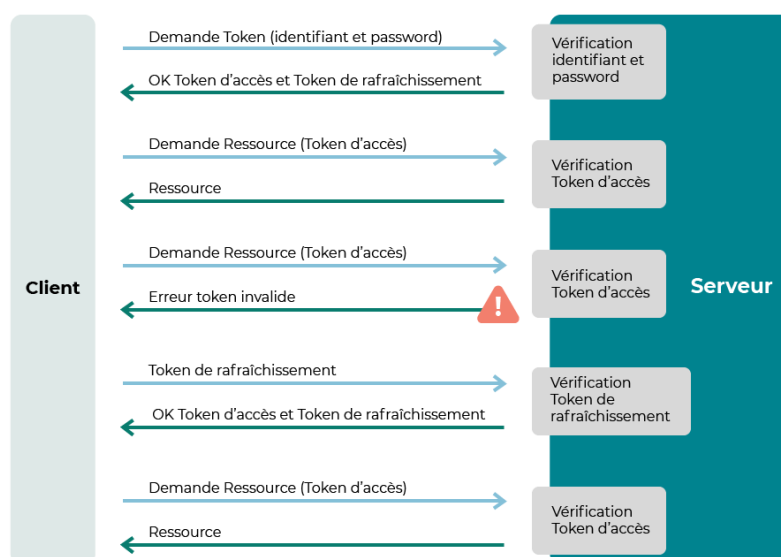
4.2 Donnez des accès avec les tokens

4.2.1 Découvrez le fonctionnement de l'échange de tokens

Voyons plus concrètement comment fonctionnent l'obtention et le rafraîchissement des JWT. Il s'agit d'un **jeu de tennis** entre le client et le serveur dans lequel un token, faisant office de balle, doit constamment transiter entre le client et le serveur :

- Le client commence toujours, et **demande une paire de tokens** au serveur en lui fournissant ses identifiants.
- Pour chaque appel suivant, le client doit fournir son **token d'accès**.
- Lorsque le token d'accès n'est plus valide, alors le client demande une **nouvelle paire de tokens** au serveur, en lui fournissant son **token de rafraîchissement**.

Cette chaîne dure tout le temps où l'utilisateur reste authentifié sur l'application.

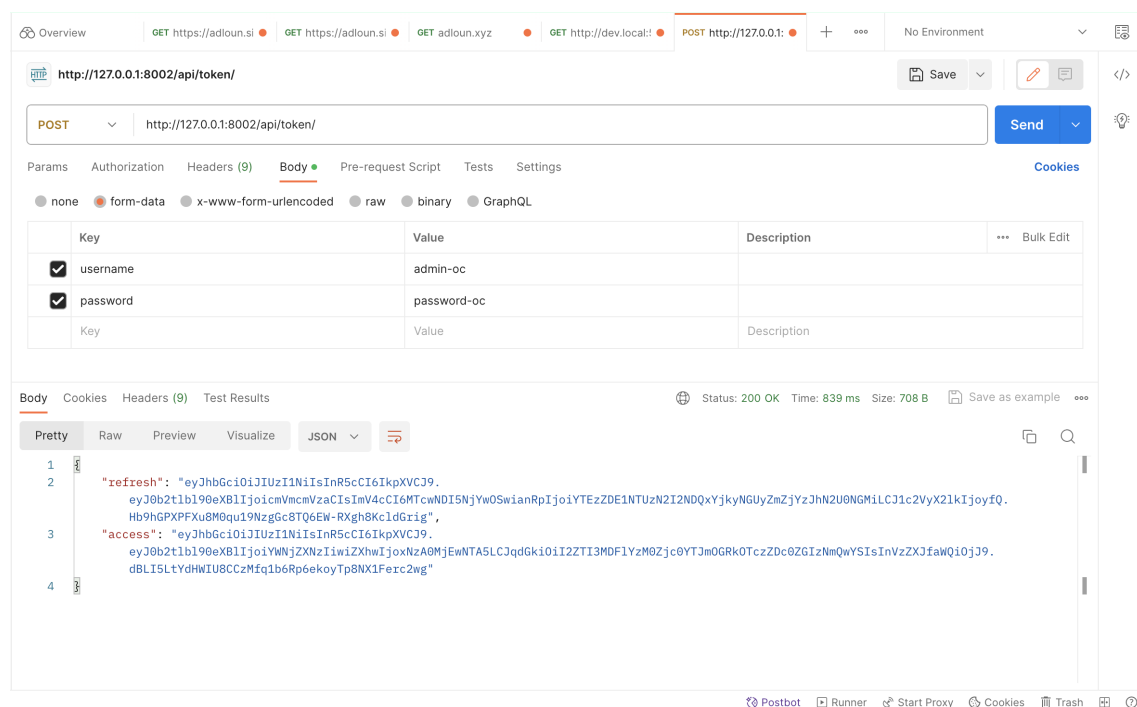


Le client envoie des demandes pour accéder aux ressources au serveur avec des tokens. Si le token est valide, le serveur envoie la ressource et un token de rafraîchissement. Le client fournit des tokens au serveur pour accéder aux ressources

4.2.2 Obtenez des tokens

La **gestion des tokens** est un processus réalisé par les applications clientes – voyons ensemble comment fonctionne ce processus. Dans ce cours, nous allons utiliser **Postman**. Il s'agit d'un outil permettant de modifier toutes les informations de la requête qui appelle notre API. Dans le cas de **l'authentification**, nous aurons besoin entre autres de modifier les headers des requêtes. Vous pouvez obtenir des tokens via un appel POST sur l'endpoint d'obtention de tokens que nous avons configuré dans le projet.

Un identifiant et un mot de passe doivent être fournis dans le corps de la requête, sous les noms **username** et **password**. Lorsque l'authentification est en succès, alors une paire de tokens est retournée.



En entrant l'identifiant `admin-oc` et le mot de passe `password-oc`, nous obtenons bien deux tokens `refresh` et `access`, une paire de tokens sont retournés

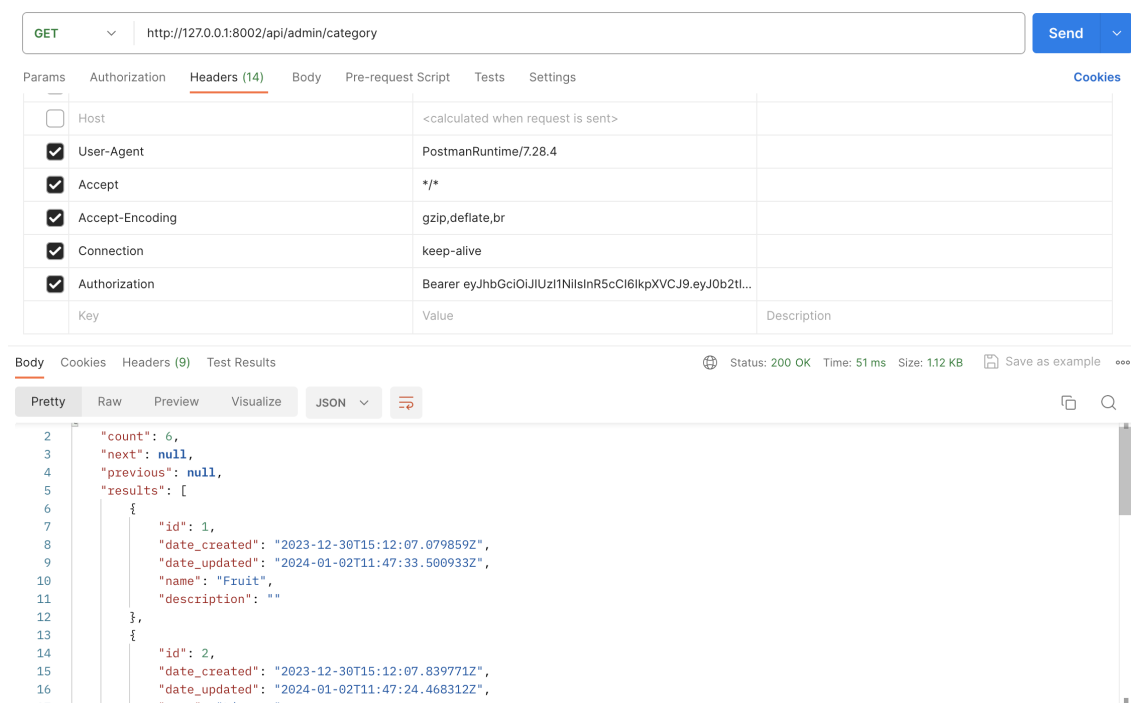
C'est quoi Postman ? Pourquoi on ne reste pas dans le navigateur ?

C'est un outil de gestion de collection d'API très utile lorsqu'on travaille en équipe, mais aussi disponible gratuitement pour un seul utilisateur. Il permet de **sauvegarder des appels API** et de réaliser plus facilement le **paramétrage de nos appels** (gestion des headers, corps de la requête, etc.).

Nous pouvons voir que chaque token est un JWT, et consiste en 3 parties séparées par des points. Chacun de ces tokens a un **rôle bien spécifique** et doit être **conservé** par l'application cliente afin de garantir la connexion de l'utilisateur.

Le token d'accès doit être transmis dans le header de chaque requête faite au serveur dans la clé nommée **Authorization**. Cette clé doit avoir pour valeur **Bearer TOKEN**, en remplaçant **TOKEN** par la valeur du token d'accès.

Dans l'onglet Headers, nous voyons la clé Authorization et le Bearer TOKEN associé. La clé Authorization et le Bearer TOKEN



De cette façon, le serveur, en recevant la requête, peut déterminer l'utilisateur réalisant la requête. Le token d'accès a une durée de vie **limitée dans le temps**; si nous décodons son payload en utilisant par exemple `jwt.io` :

```

{
  "token_type": "access",
  "exp": 1628927720,
  "jti": "63e4e6dab0494aee803fed93f24a80c1",
  "user_id": 1
}

```

... nous pouvons voir différentes informations :

- `token_type` indique le type de token ;
- `exp` est un timestamp indiquant jusqu'à quand ce token peut être utilisé ;
- `jti` signifie JWT ID, c'est un identifiant unique créé lors de la génération du token ;
- `user_id` est l'identifiant Django de l'utilisateur, il est ajouté par Simple JWT lors de la génération du token.

Est-il possible de modifier la durée de vie des tokens ?

Oui, la durée de vie des tokens peut être configurée dans les settings de Django.

```
SIMPLE_JWT = {
    'ACCESS_TOKEN_LIFETIME': timedelta(minutes=5),
    'REFRESH_TOKEN_LIFETIME': timedelta(days=1),
}
```

4.2.3 Rafraîchissez des tokens

Lorsque le token d'accès n'est plus valide, il est nécessaire d'en créer un autre. C'est à cela que sert le **token de rafraîchissement**.

Il faut pour cela réaliser un appel en POST sur l'endpoint de rafraîchissement de tokens. Le token de rafraîchissement doit être transmis dans le corps de la requête sous l'attribut **refresh**, et un nouveau token d'accès est alors retourné par le serveur.

POST ▼ | http://127.0.0.1:8002/api/token/refresh/ Send ▼

Params Authorization Headers (9) **Body** ● Pre-request Script Tests Settings Cookies

● none ● **form-data** ● x-www-form-urlencoded ● raw ● binary ● GraphQL

	Key	Value	Description	... Bulk Edit
<input checked="" type="checkbox"/>	refresh	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t1bi90eX...		
	Key	Value	Description	

Body Cookies Headers (9) Test Results Status: 200 OK Time: 9 ms Size: 488 B Save as example

Pretty Raw Preview Visualize JSON ▼ ↻

```

1
2      "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ0b2t1bi90eXB1IjoieWwNjZlZlZlZWwiOjoxNzA0MjMzMzQ0dGkiOiI3MTA5NTNlNmVldmY0NGIyZWYwMDk3ZWJjMWw0ODZKYSIsInVzZXJfaWQ0IjojJ9.GPLQ5IvRFHxUScwKs6ukE_Q7y2vV7pwReAvN3w18M"
3

```

Quand nous envoyons le token de rafraîchissement, un token d'accès nous est retourné dans Postman Le token de rafraîchissement permet d'obtenir un nouveau token d'accès Voyons comment réaliser ces appels ensemble dans le screencast ci-dessous.

N'hésitez pas à tester le fonctionnement des tokens, comment et quand ils expirent. Chaque matin en reprenant le développement du votre projet, vos tokens devront être rafraîchis, c'est signe que l'authentification fonctionne. :)

Le code du projet est disponible sur la branche P3C1-C2 du projet. Expérimentez la gestion des tokens comme le ferait une application cliente, afin que ce processus n'ait plus de secret pour vous !

En résumé

1. Les tokens JWT sont un moyen d'assurer l'authentification des utilisateurs.
2. Les tokens d'accès et de rafraîchissement sont à conserver précieusement par les applications clientes pendant la session des utilisateurs.
3. Deux endpoints différents permettent d'une part l'obtention de tokens, et d'autre part le rafraîchissement du token d'accès.

Maintenant que notre serveur nous permet d'obtenir et de rafraîchir des tokens, voyons dès le prochain chapitre comment limiter l'accès à certains endpoints.

4.3 Restreignez l'accès à certains endpoints

4.3.1 Limitez l'accès aux utilisateurs authentifiés

Nous disposons maintenant d'endpoints d'administration qui permettent de gérer pleinement des entités, mais leur accès est pour le moment **public**. Améliorons cela en imposant aux utilisateurs d'être authentifiés pour pouvoir faire des appels à nos endpoints d'administration. Commençons par l'endpoint d'**administration des catégories**.

DRF nous permet de **gérer l'accès aux endpoints** au travers des permissions, et nous propose une permission **IsAuthenticated**, exactement ce qu'il nous faut !

Les permissions se configurent au niveau des views, au travers de l'attribut de classe **permission_classes**. Cet attribut est une liste, et permet donc d'appliquer plusieurs permissions. Elles sont parcourues une à une par DRF, et l'accès à la vue n'est permis que si toutes les permissions le permettent.

Mettons en place cette permission sur notre **AdminCategoryViewSet** :

```
from rest_framework.permissions import IsAuthenticated

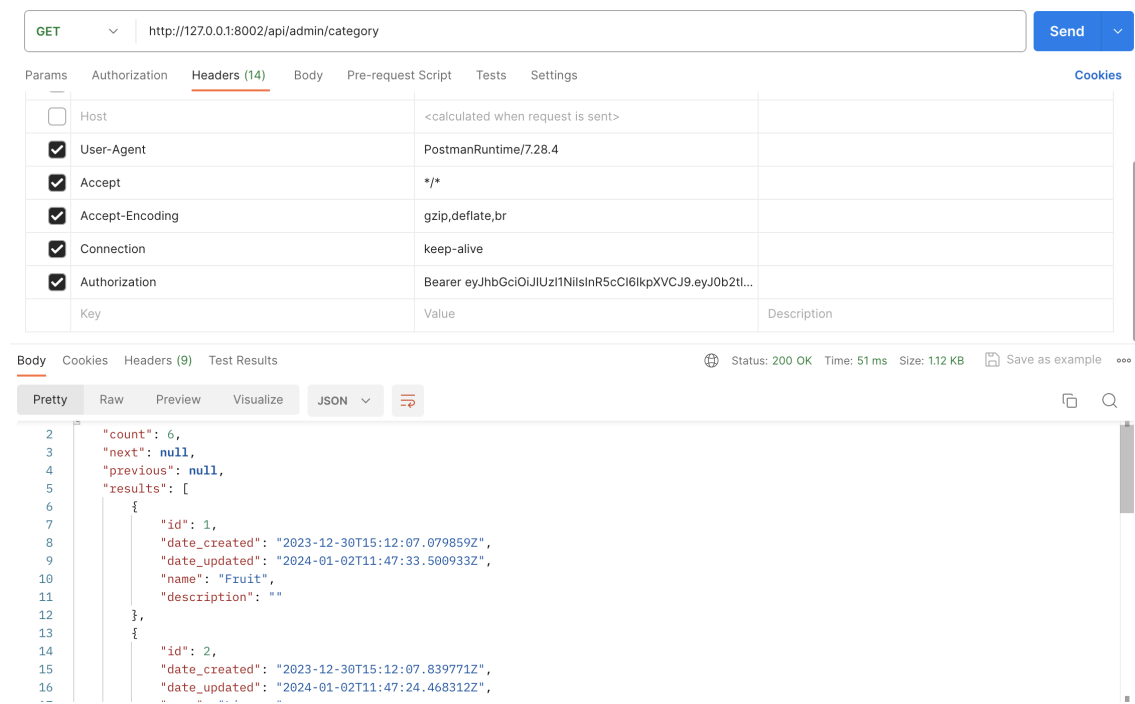
class AdminCategoryViewSet(MultipleSerializerMixin, ModelViewSet):

    serializer_class = CategoryListSerializer
    detail_serializer_class = CategoryDetailSerializer
    # Nous avons simplement à appliquer la permission sur le viewset
    permission_classes = [IsAuthenticated]

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

L'accès à l'endpoint nécessite à présent d'être authentifié en ajoutant dans les headers de la requête POST la clé **Authorization** qui a pour valeur **Bearer ACCESS**, où **ACCESS** est la valeur du token d'accès.

Appel en succès d'un endpoint sécurisé en précisant l'attribut **Authorization** dans le header de la requête



Vous verrez le status code à droite entre les KEY et la console. Ci-dessus, l'accès a été accepté et un status code 200 est retourné. Sans authentification, un status code 401 est retourné, et l'accès est **refusé**.

Un appel sans disposer d'un JWT valide ou sans en fournir sur un endpoint sécurisé retourne un status code 401

Voyons la mise en place d'authentification dans le screencast ci-dessous :

18

Lors de la création d'un endpoint, il est important de penser à son accès, car il retourne peut-être des informations **confidentielles** ou permet des **actions** qui ne doivent pas être permises à tout le monde.

4.3.2 Créez des permissions plus fines

Nous venons de limiter l'accès aux personnes authentifiées, mais est-ce suffisant ?

La réponse est non, c'est un **premier niveau de sécurité**, mais en l'état, les clients de notre boutique pourraient alors administrer nos catégories en disposant seulement d'un compte utilisateur.

DRF propose des permissions, je vous propose encore mieux ! De **créer les nôtres**. ;)

Faisons ensemble en sorte que **seuls les administrateurs** puissent accéder à cet endpoint. Créons un fichier nommé `permissions.py` dans notre application Django `shop`.

```
from rest_framework.permissions import BasePermission
```

```
class IsAdminAuthenticated(BasePermission):

    def has_permission(self, request, view):
        # Ne donnons l'accès qu'aux utilisateurs administrateurs authentifiés
        return bool(request.user and request.user.is_authenticated and request.user.is_superuser)
```

Il ne nous reste plus qu'à modifier notre vue pour utiliser notre nouvelle permission :

```
from shop.permissions import IsAdminAuthenticated

class AdminCategoryViewSet(MultipleSerializerMixin, ModelViewSet):

    serializer_class = CategoryListSerializer
    detail_serializer_class = CategoryDetailSerializer
    permission_classes = [IsAdminAuthenticated]

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

Notre endpoint est à présent **protégé** et accessible seulement aux **administrateurs authentifiés**. Un simple compte utilisateur ne suffit plus pour pouvoir y accéder.

Suivez-moi dans le screencast ci-dessous pour voir comment j'ai mis en place des permissions plus fines pour nos administrateurs :

Les permissions permettent la mise en place de règles de contrôle strictes. Avoir une permission par « acteur » (clients, partenaires, utilisateurs, administrateurs, etc.) de notre API est un bon moyen de gérer qui peut accéder à quels endpoints.

Exercice

Allons plus loin dans nos contrôles ! Pour diverses raisons, un utilisateur administrateur n'est pas membre de l'équipe de la boutique. Il faudrait qu'il puisse n'avoir accès à l'administration qu'en tant que **prestataire**, et nous ne souhaitons pas qu'il puisse ajouter de nouvelles catégories.

Je vous propose donc de mettre en place une **nouvelle permission** qui va vérifier que l'utilisateur fasse également partie de l'équipe, en vérifiant que **is_staff** du model **User** est bien à **True**. Les deux permissions pourront alors être mises en place sur le Viewset d'administration de catégorie.

Disposer de deux permissions nous permettra par la suite de donner l'accès à certains endpoints aux membres de l'équipe sans qu'ils aient besoin de disposer d'un compte administrateur, par exemple.

Pour réaliser cela, vous pouvez partir de la branche **P2C3_exercice**. Elle contient déjà ce que nous venons de faire ensemble. Une solution est proposée sur la branche **P2C3_solution**.

En résumé

1. Les permissions permettent de limiter l'accès aux endpoints.
2. DRF fournit certaines permissions, mais il est possible de créer les nôtres.

3. Disposer une permission par acteur permet de facilement savoir sur quels endpoints les permissions doivent être placées.
4. Lors de la mise en place d'un nouvel endpoint, il est important de savoir qui va l'utiliser, pour déterminer s'il doit rester public ou doit posséder certaines permissions.

Notre API est toute sécurisée. Je vous invite maintenant à valider vos acquis de cette partie dans le quiz. Après, nous reviendrons sur tout ce que vous avez accompli pendant ce cours !

Table des matières

1	Introduction	3
2	Mettez en place une API simple	5
2.1	Découvrez Django REST Framework	5
2.1.1	Découvrez pourquoi utiliser une API	5
2.1.2	Découvrez le cas concret du cours	6
2.1.3	Installez et configurez DRF	6
2.2	Gérez des données avec un endpoint	9
2.2.1	Créez un premier endpoint	9
2.2.2	Découvrez les méthodes d'un endpoint	11
2.3	Rendez les Views plus génériques avec un ModelViewSet	12
2.4	Mettez en place un router	12
2.4.1	Transformez une ApiView en ModelViewSet	13
2.4.2	Ne permettez que la lecture	15
2.5	Filtrez les résultats d'un endpoint	16
2.5.1	Appliquez un filtre sur les données retournées	16
2.6	Écrivez des tests pour votre API	19
2.6.1	Découvrez le TestCase de DRF	19
2.6.2	Écrivez des tests avec APITestCase	19
3	Rendez vos endpoints plus performants	23
3.1	Minimisez les appels de votre API grâce aux serializers	23
3.1.1	Retournez plus d'informations	23
3.1.2	Imbriquez les serializers	24
3.1.3	Ajoutez de la pagination	26
3.2	Différenciez les informations de liste et de détail	29
3.2.1	Retournez plus d'informations dans le détail	29
3.2.2	Améliorez le rendu du détail d'un endpoint	29
3.3	Ajoutez de l'interaction avec les actions	31
3.3.1	Ne soyez pas trop actif!	31
3.3.2	Soyez actif quand même!	32
3.4	Validez les données	35
3.4.1	Ne permettez pas la création de tout et n'importe quoi	35
3.4.2	Validez les données d'un champ	35
3.4.3	Mettez en place une validation multiple	37

4	Sécurisez votre API avec l'authentification	39
4.1	Ajoutez l'authentification des utilisateurs	39
4.1.1	Découvrez JSON Web Token	39
4.1.2	Installez et configurez djangorestframework-simple-jwt	40
4.2	Donnez des accès avec les tokens	42
4.2.1	Découvrez le fonctionnement de l'échange de tokens	42
4.2.2	Obtenez des tokens	43
4.2.3	Rafraîchissez des tokens	45
4.3	Restreignez l'accès à certains endpoints	46
4.3.1	Limitez l'accès aux utilisateurs authentifiés	46
4.3.2	Créez des permissions plus fines	47