Rendez vos endpoints plus performants

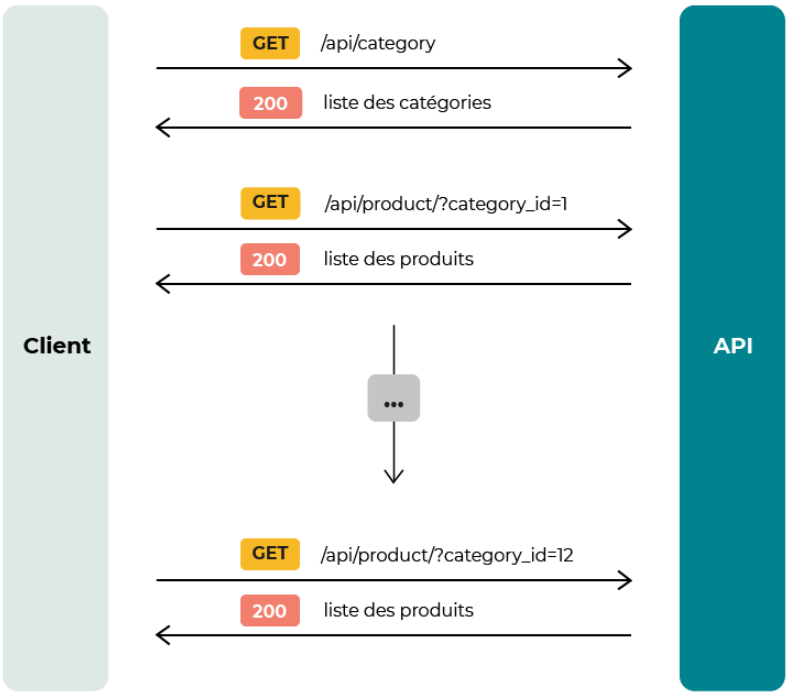
Ibrahim ALAME

18 janvier 2024

1 Minimisez les appels de votre API grâce aux serializers

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.

1

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.

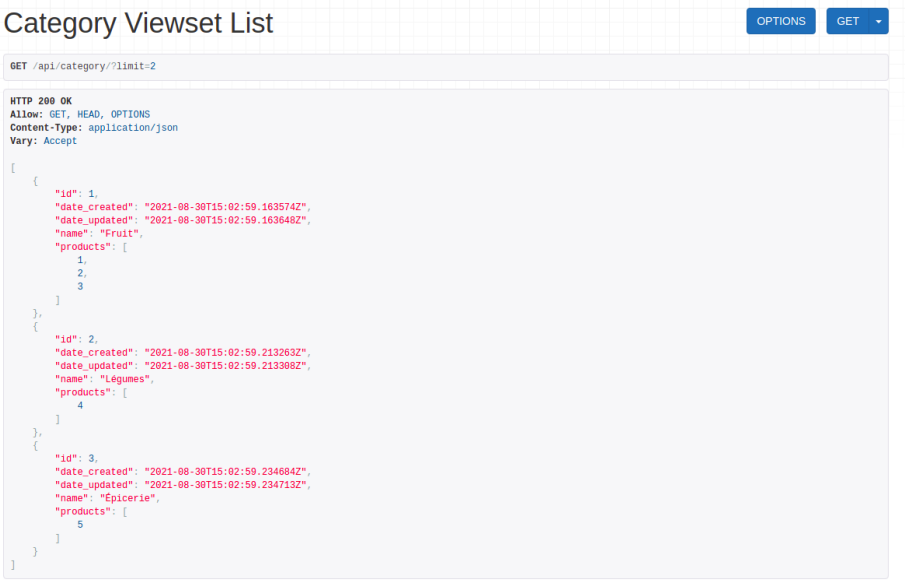
1.2 Imbriquez les serializers

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']
```

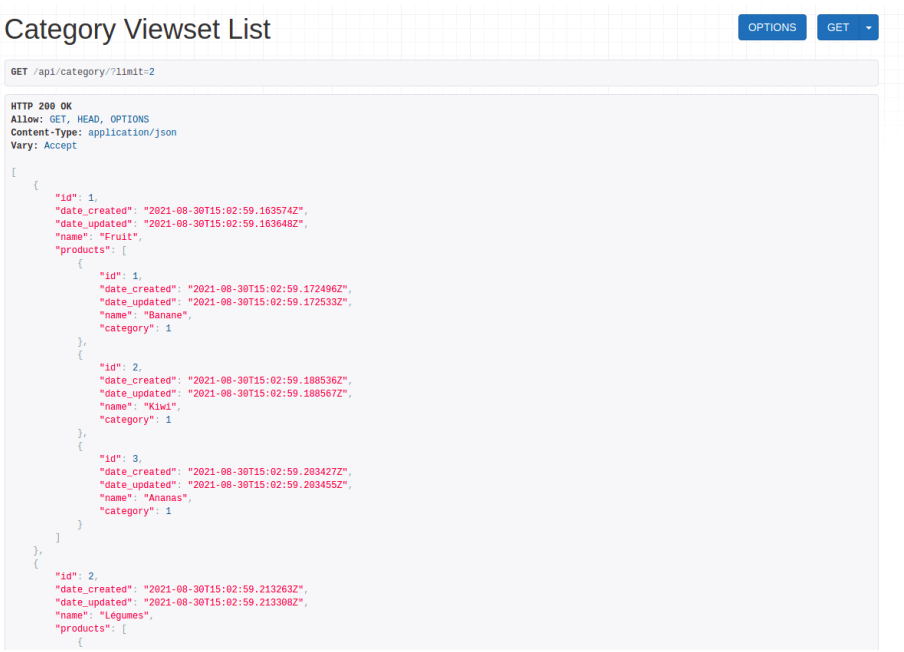
2

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*.

```python
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 !

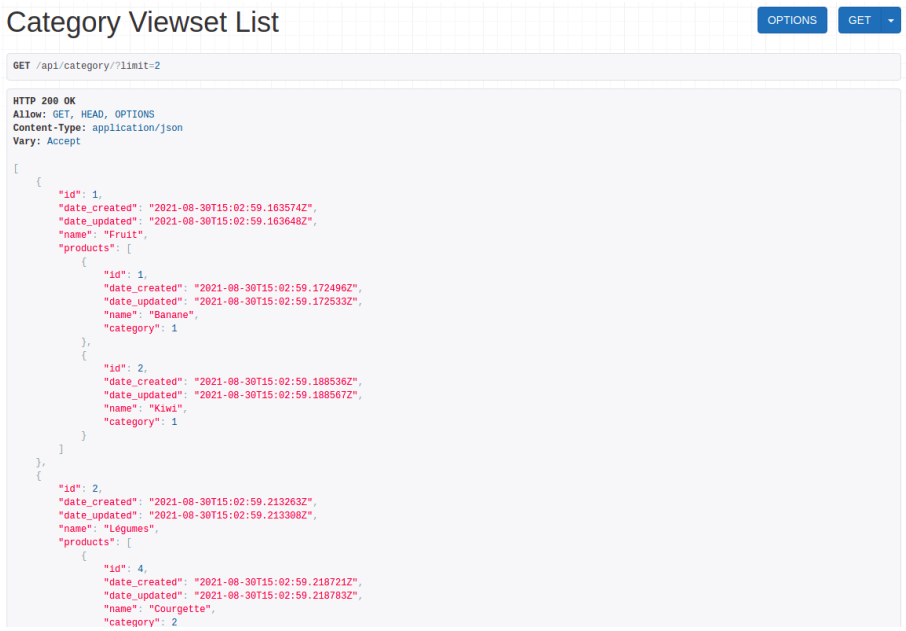
1.3 Ajoutez de la pagination

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.

Il suffit d'jouter dans `settings.py` :

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
    'rest_framework.pagination.LimitOffsetPagination',
    'PAGE_SIZE': 10
}
```



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.

3

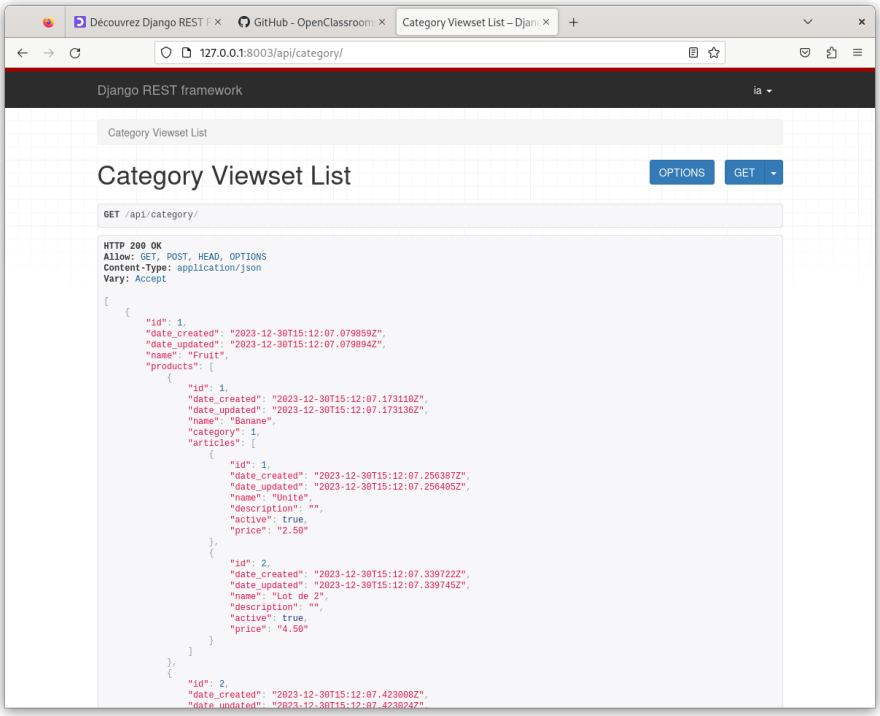
Lorsque les applications clientes sont réalisées en interne, il est important de bien communiquer avec ses équipes afin de savoir quelles sont les données qui leurs sont importantes, et dans quelles conditions. Le résultat de ces échanges vous donnera les meilleures pistes pour savoir quelles données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

2 Différenciez les informations de liste et de détail

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.

2.2 Améliorez le rendu du détail d'un endpoint

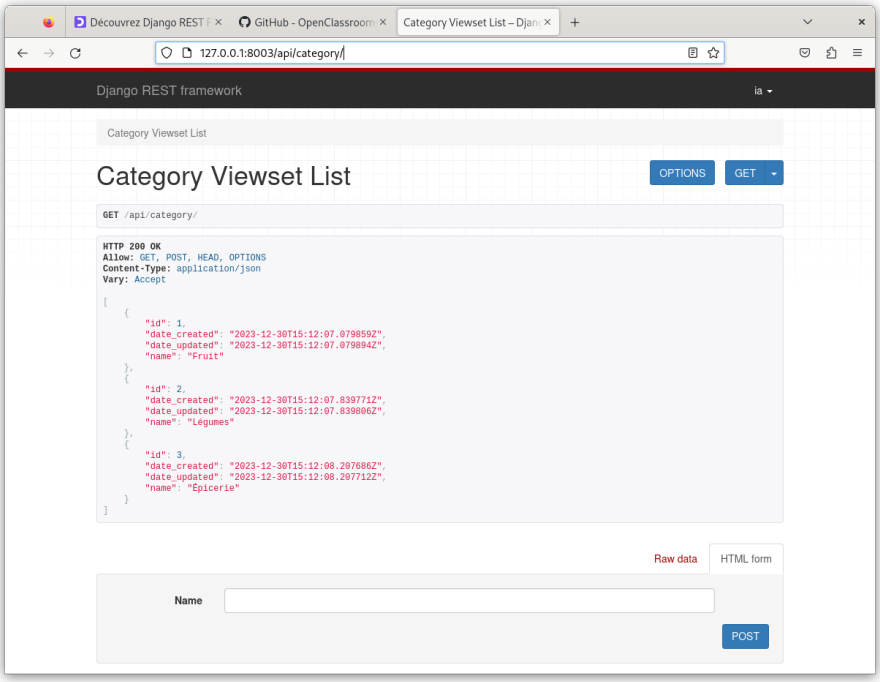
Améliorons notre endpoint de catégories en ne retournant que les informations *minimales*, dans le cas d'une liste, et *détaillées*, lorsque nous consultons une catégorie spécifique.

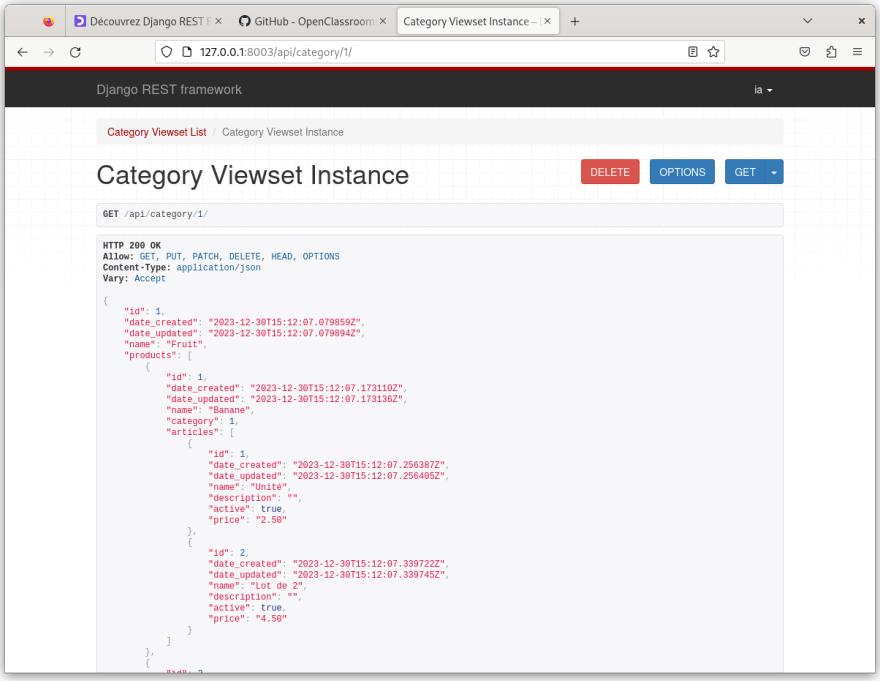
DRF nous permet au travers de ses viewsets de redéfinir la méthode `get_serializer_class` qui, elle, détermine le serializer à utiliser. Par défaut, le serializer retourné est celui défini sur l'attribut de classe `serializer_class` du viewset.

Lorsqu'une requête entre dans notre API, les viewsets définissent un attribut action qui correspond à l'*action* que l'*application client* 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()
```


Découvrez Django REST F
GitHub - OpenClassroom
Category Viewset List – Djan
127.0.0.1:8003/api/category/
Django REST framework
ia
Category Viewset List
Category Viewset List
OPTIONS
GET
GET /api/category/
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
{
"id": 1,
"date_created": "2023-12-30T15:12:07.079859Z",
"date_updated": "2023-12-30T15:12:07.079894Z",
"name": "Fruit"
},
{
"id": 2,
"date_created": "2023-12-30T15:12:07.839771Z",
"date_updated": "2023-12-30T15:12:07.839806Z",
"name": "Légumes"
},
{
"id": 3,
"date_created": "2023-12-30T15:12:08.207686Z",
"date_updated": "2023-12-30T15:12:08.207712Z",
"name": "Épicerie"
}
]
Raw data
HTML form
Name
POST


Découvrez Django REST F
GitHub - OpenClassroom
Category Viewset Instance –
127.0.0.1:8003/api/category/1/
Django REST framework
ia
Category Viewset List / Category Viewset Instance
Category Viewset Instance
DELETE
OPTIONS
GET
GET /api/category/1/
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
"id": 1,
"date_created": "2023-12-30T15:12:07.079859Z",
"date_updated": "2023-12-30T15:12:07.079894Z",
"name": "Fruit",
"products": [
{
"id": 1,
"date_created": "2023-12-30T15:12:07.173110Z",
"date_updated": "2023-12-30T15:12:07.173136Z",
"name": "Banane",
"category": 1,
"articles": [
{
"id": 1,
"date_created": "2023-12-30T15:12:07.256387Z",
"date_updated": "2023-12-30T15:12:07.256405Z",
"name": "Unité",
"description": "",
"active": true,
"price": "2.50"
},
{
"id": 2,
"date_created": "2023-12-30T15:12:07.339722Z",
"date_updated": "2023-12-30T15:12:07.339745Z",
"name": "Lot de 2",
"description": "",
"active": true,
"price": "4.50"
}
]
},
{

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 Ajoutez de l'interaction avec les actions

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. . . ;)

3.2 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'endpointhttp ://127.0.0.1 :8000/api/catego
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/.

4

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` :

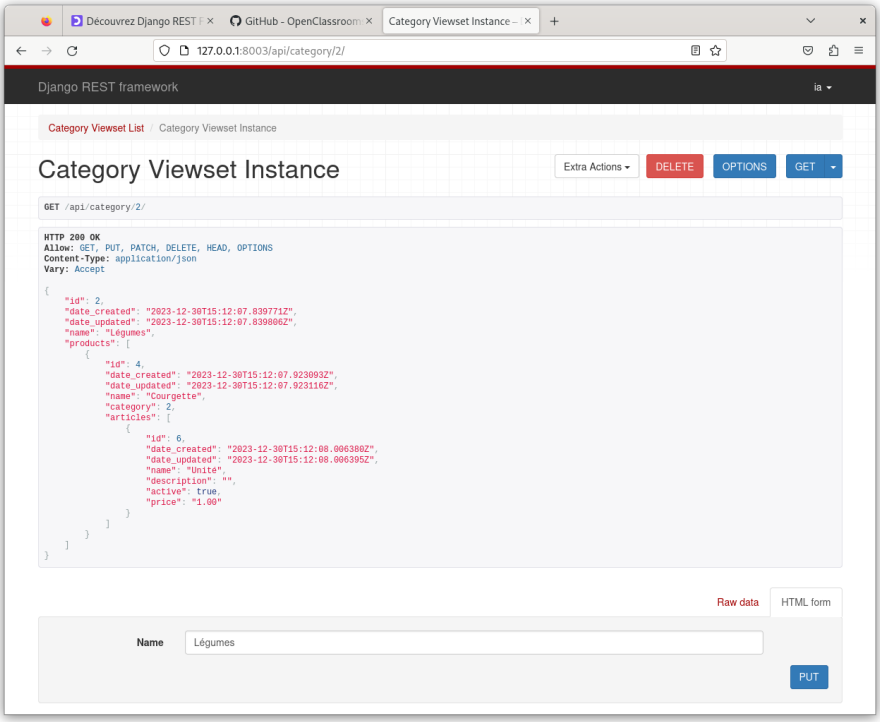
```
class Category(models.Model):

    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
```

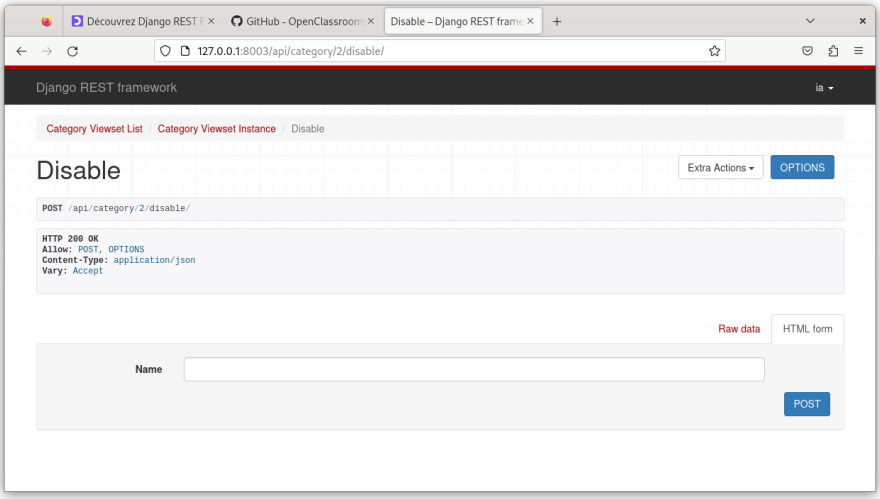
```
        self.active = False
        self.save()
        self.products.update(active=False)
```

5

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()
```





6

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 !

4 Validez les données

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. ;)

4.2 Validez les données d'un champ

Commençons tout de suite par la mise en place du nouvel endpoint sur l'URL qui, 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

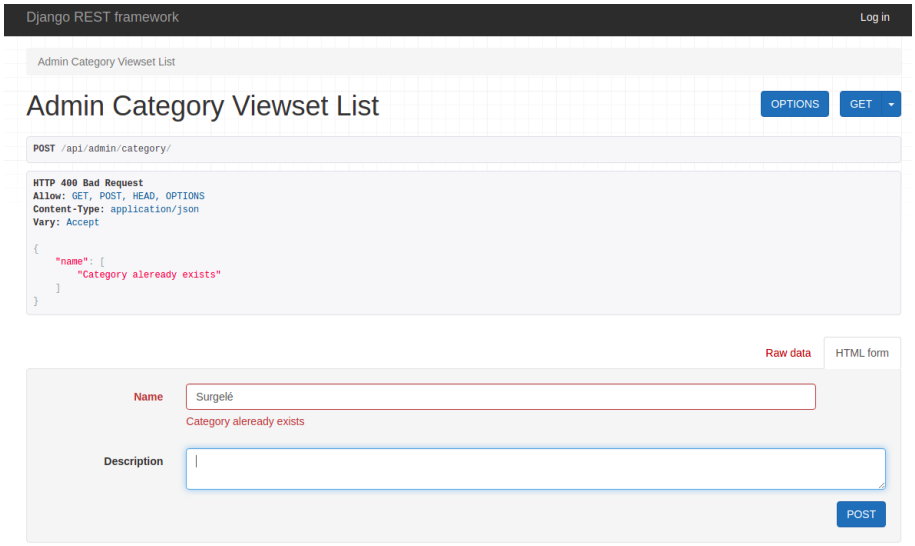
Maintenant que nous utilisons un `Viewset`, la création de catégorie est possible. C'est à la création que sert ce formulaire en nous permettant d'effectuer des actions POST. Les actions de mise à jour et de suppression sont disponibles sur les URL de détail des catégories.

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.

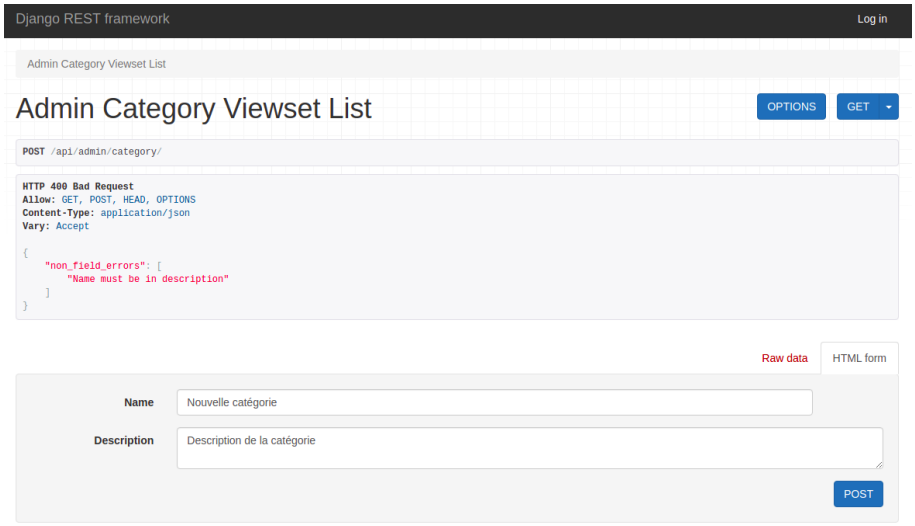
4.3 Mettez en place une validation multiple

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.



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 !