Sécurisez votre API avec l'authentification

Ibrahim ALAME

4 janvier 2024

1 Ajoutez l'authentification des utilisateurs

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.

> **1**
>
> 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.eyJ0b2tlbl90eXBlIjoicmVmcmVzaCIsImV
4cCI6MTYyODk0ODE0NSwianRpIjoiODBmZTA4MDkxM2UxNDBjYmEwMDU4YWY4YmM5NjllZjY
iLCJ1c2VyX2lkIjoxfQ.l31cs5pTApiR9R9s8pZaeyHIJuTmjcs07fxSqSpj1fQ
```

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 `djangorestframework-simplejwt` :

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

1.2 Installez et configurez djangorestframework-simple-jwt

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

```
djangorestframework-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.

2

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.JWTAuthenticatio
}
```

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.

3

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.

1.2.1 En résumé

1. Les JWT permettent de transférer des informations du client au serveur en plus des jetons d'authentification.
2. `djangorestframework-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 !

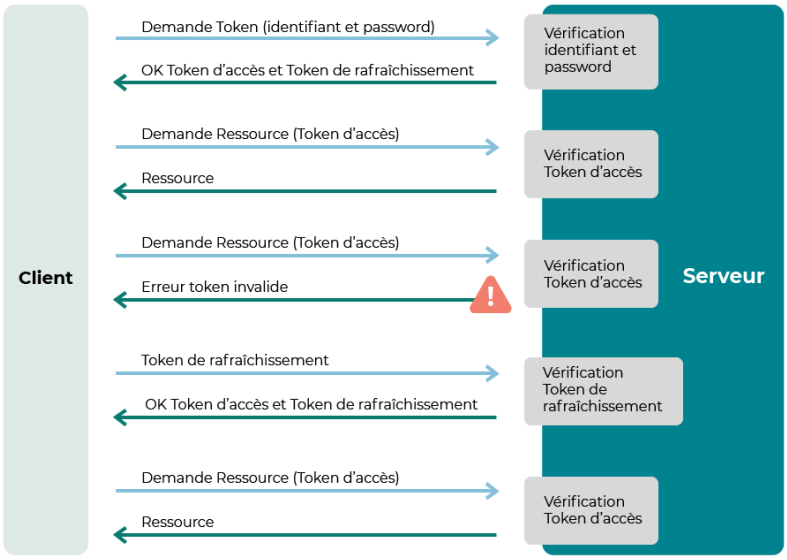
2 Donnez des accès avec les tokens

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.

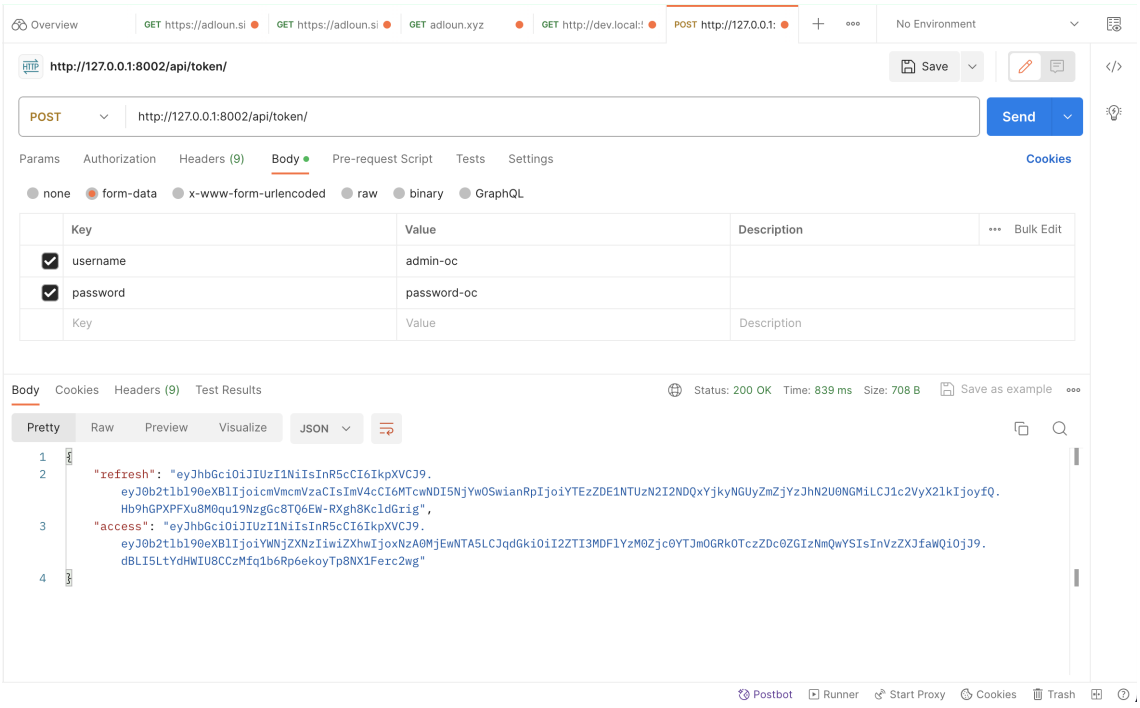
Cette chaîne dure tout le temps où l'utilisateur reste authentifié sur l'application.



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

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.

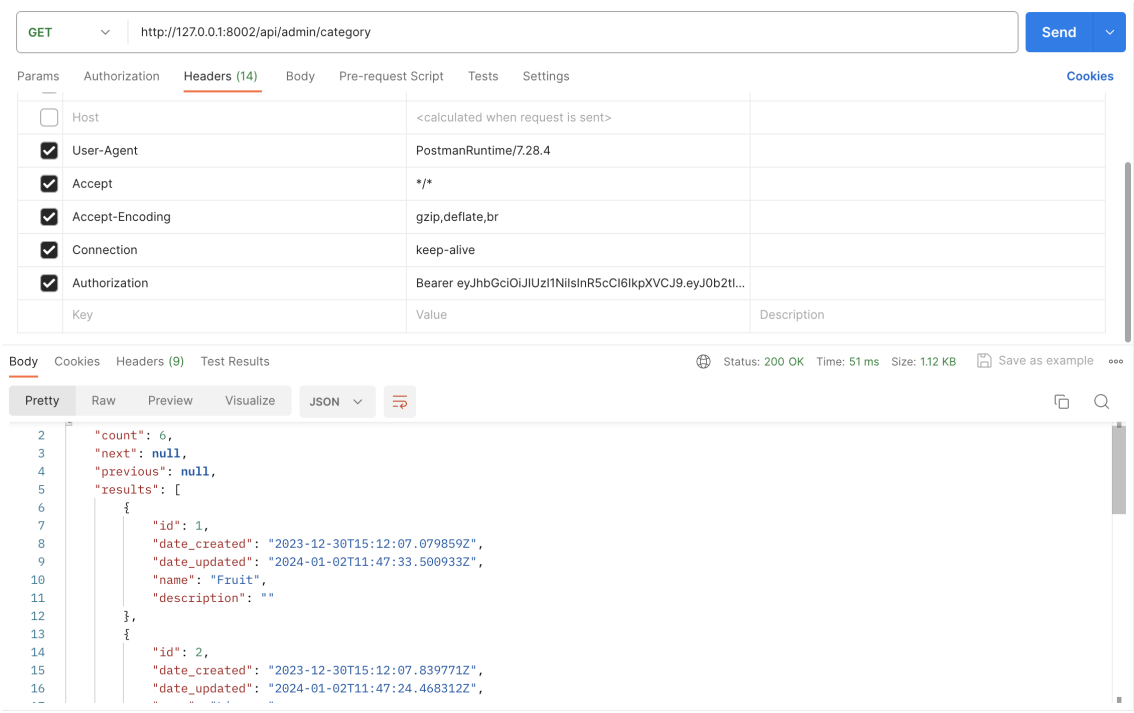


En entrant l'identifiant admin-oc et le mot de passe password-oc, nous obtenons bien deux tokens refresh et access, une paire de tokens sont retournés

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.

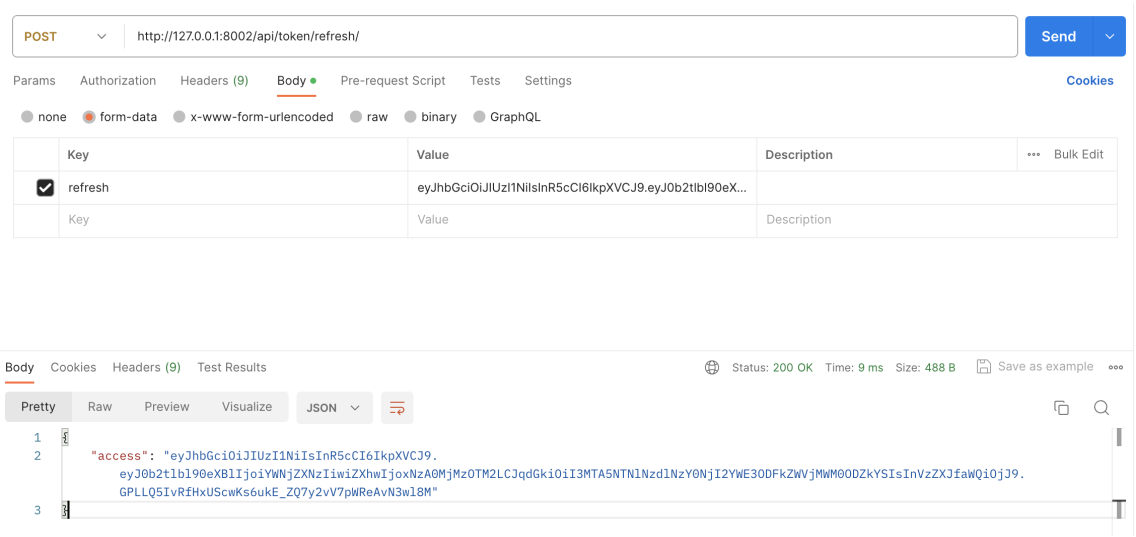
Est-il possible de modifier la durée de vie des tokens ?

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),
}
```

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.



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.

3 Restreignez l'accès à certains endpoints

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` :

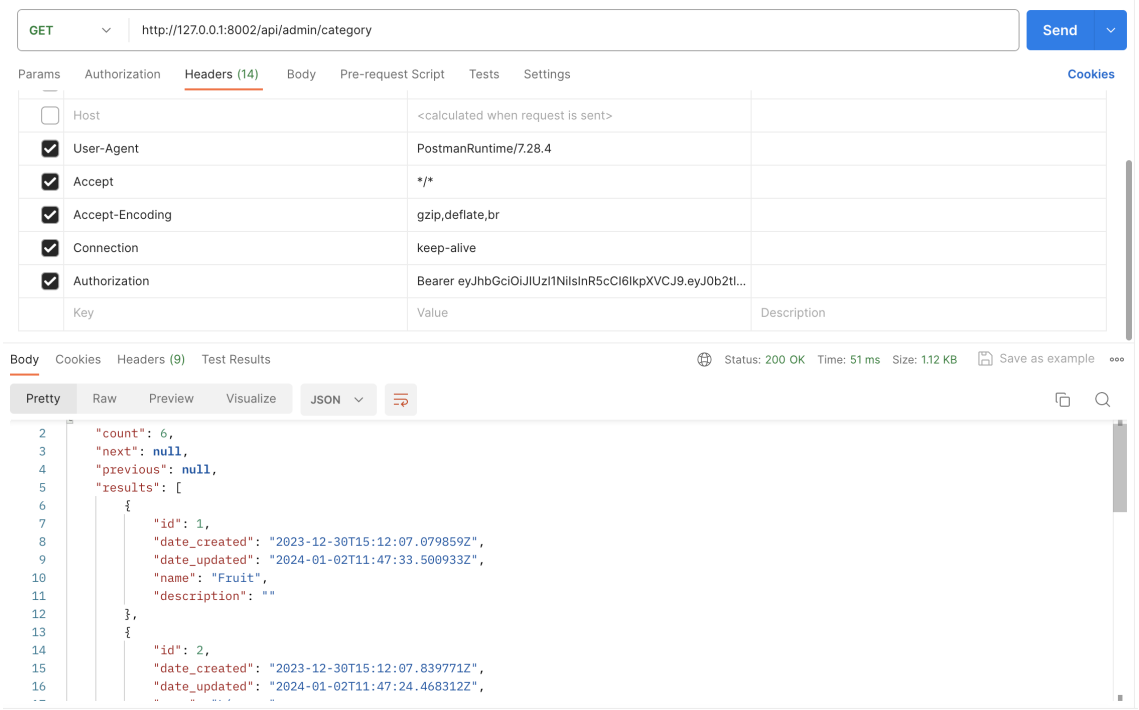
```
from rest_framework.permissions import IsAuthenticated

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 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 :

4

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.

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
```

```python
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 :

```python
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 !