eBook Gratuit

APPRENEZ

django-rest-framework

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#django-rest-framework

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: django-rest-framework

It is an unofficial and free django-rest-framework ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official django-rest-framework.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec django-rest-framework

Remarques

Django Framework REST est une boîte à outils pour la création d'applications Web. Cela aide le programmeur à faire *des API REST* , mais il peut gérer des niveaux d'API moins matures. Pour plus d'informations sur les niveaux de maturité de l'API, recherchez le modèle de maturité de Richardson.

En particulier, Django REST Framework n'approuve aucune disposition particulière au niveau Hypermedia, et c'est au programmeur (ou à d'autres projets, tels que srf-hal-json) s'ils souhaitent poursuivre une implémentation de l'API HATEOAS, opinions en dehors du cadre. Ainsi, il est possible d'implémenter une API HATEOAS dans Django REST Framework, mais aucun utilitaire n'est déjà disponible.

Versions

Version	Date de sortie
3.5.3	2016-11-07

Examples

Installer ou configurer

Exigences

- Python (2.7, 3.2, 3.3, 3.4, 3.5, 3.6)
- Django (1.7+, 1.8, 1.9, 1.10, 1.11)

Installer

Vous pouvez soit utiliser `pip` pour installer ou cloner le projet depuis github.

- **En utilisant `pip` :**

```
pip install djangorestframework
```

- **Utilisation du `git clone` :**

```
git clone git@github.com:tomchristie/django-rest-framework.git
```

Après l'installation, vous devez **ajouter `rest_framework` à vos paramètres `INSTALLED_APPS` .**

```
INSTALLED_APPS = (
    ...
    'rest_framework',
)
```

Si vous envisagez d'utiliser l'API de navigation, vous souhaiterez probablement également ajouter les vues de connexion et de déconnexion du framework REST. Ajoutez ce qui suit à votre fichier urls.py racine.

```
urlpatterns = [
    ...
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Exemple

Jetons un coup d'œil à un exemple rapide d'utilisation de la structure REST pour créer une API basée sur un modèle simple.

Nous allons créer une API de lecture-écriture pour accéder aux informations sur les utilisateurs de notre projet.

Tous les paramètres globaux pour une API de structure REST sont conservés dans un dictionnaire de configuration unique nommé `REST_FRAMEWORK` . Commencez par ajouter ce qui suit à votre module `settings.py` :

```
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}
```

Nous sommes prêts à créer notre API maintenant. Voici le module root `urls.py` de notre projet:

```
from django.conf.urls import url, include
from django.contrib.auth.models import User
from rest_framework import routers, serializers, viewsets

# Serializers define the API representation.
class UserSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        model = User
        fields = ('url', 'username', 'email', 'is_staff')

# ViewSets define the view behavior.
class UserViewSet(viewsets.ModelViewSet):
    queryset = User.objects.all()
    serializer_class = UserSerializer
```

```
# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'users', UserViewSet)

# Wire up our API using automatic URL routing.
# Additionally, we include login URLs for the browsable API.
urlpatterns = [
    url(r'^', include(router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework'))
]
```

Vous pouvez maintenant ouvrir l'API dans votre navigateur à l' `http://127.0.0.1:8000/` et afficher votre nouvelle API utilisateur. Si vous utilisez le contrôle de connexion dans le coin supérieur droit, vous pourrez également ajouter, créer et supprimer des utilisateurs du système.

Lire Démarrer avec django-rest-framework en ligne: https://riptutorial.com/fr/django-rest-framework/topic/1875/demarrer-avec-django-rest-framework

Chapitre 2: Authentification

Examples

Ecrire des authentifications personnalisées

à partir de django.contrib.auth.models importation Utilisateur de rest_framework importation à partir d'exceptions à l'importation rest_framework

Cet exemple d'authentification provient directement des documents officiels ici .

```
class ExampleAuthentication(BaseAuthentication):
    def authenticate(self, request):
        username = request.META.get('X_USERNAME')
        if not username:
            return None

        try:
            user = User.objects.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed('No such user')

        return (user, None)
```

Une classe d'authentification personnalisée comporte quatre parties.

1. Étendez-le à partir de la classe BaseAuthentication fournie dans `from rest_framework. importation authentification BaseAuthentication
2. Avoir une méthode appelée `authenticate` prenant `request` en premier argument.
3. Renvoie un tuple de (utilisateur, aucun) pour une authentification réussie.
4. Augmenter l'exception `AuthenticationFailed` pour une authentification échouée. Ceci est disponible dans rest_framework.authentication.

```
class SecretAuthentication(BaseAuthentication):
    def authenticate(self, request):
        app_key = request.META.get('APP_KEY')
        app_secret = request.META.get('APP_SECRET')
        username = request.META.get('X_USERNAME')
        try:
            app = ClientApp.objects.match_secret(app_key, app_secret)
        except ClientApp.DoesNotExist:
            raise AuthenticationFailed('App secret and key does not match')
        try:
           user = app.users.get(username=username)
        except User.DoesNotExist:
            raise AuthenticationFailed('Username not found, for the specified app')
        return (user, None)
```

Le schéma d'authentification renverra les réponses HTTP 403 interdites lorsqu'une demande non authentifiée se voit refuser l'accès.

Authentification par demande de base API-KEY

Vous pouvez utiliser les classes d'autorisation de `django rest framework` pour vérifier les en-têtes de demande et authentifier les demandes des utilisateurs

- Définissez votre secret_key sur les paramètres du projet

```
API_KEY_SECRET = 'secret_value'
```

note: une bonne pratique consiste à utiliser des variables d'environnement pour stocker cette valeur secrète.

- Définir une classe d'autorisation pour l'authentification API-KEY

Créez le fichier permissions.py sur votre répertoire d'application avec les codes ci-dessous:

```
from django.conf import settings

from rest_framework.permissions import BasePermission

class Check_API_KEY_Auth(BasePermission):
    def has_permission(self, request, view):
        # API_KEY should be in request headers to authenticate requests
        api_key_secret = request.META.get('API_KEY')
        return api_key_secret == settings.API_KEY_SECRET
```

- Ajoutez cette classe d'autorisation aux vues

```
from rest_framework.response import Response
from rest_framework.views import APIView

from .permissions import Check_API_KEY_Auth


class ExampleView(APIView):
    permission_classes = (Check_API_KEY_Auth,)

    def get(self, request, format=None):
        content = {
            'status': 'request was permitted'
        }
        return Response(content)
```

Ou, si vous utilisez le décorateur @api_view avec des vues basées sur les fonctions

```
from rest_framework.decorators import api_view, permission_classes
from rest_framework.response import Response

from .permissions import Check_API_KEY_Auth


@api_view(['GET'])
@permission_classes((Check_API_KEY_Auth, ))
def example_view(request, format=None):
    content = {
```

```
        'status': 'request was permitted'
    }
    return Response(content)
```

Lire Authentification en ligne: https://riptutorial.com/fr/django-rest-framework/topic/5451/authentification

Chapitre 3: Authentification

Examples

Configuration de l'authentification globalement

L'authentification dans Django REST Framework peut être configurée globalement en tant que sous-clé de la variable `REST_FRAMEWORK` dans settings.py, tout comme les autres configurations d' `REST_FRAMEWORK` par défaut.

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.BasicAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    )
}
```

Configuration de l'authentification pour un APIView spécifique

L'authentification peut être définie pour un point de terminaison APIView spécifique, à l'aide de la variable `authentication_classes` :

```
from rest_framework.authentication import SessionAuthentication, BasicAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response
from rest_framework.views import APIView

class ExampleView(APIView):
    authentication_classes = (SessionAuthentication, BasicAuthentication)
    permission_classes = (IsAuthenticated,)

    def get(self, request):
        content = {
            'user': unicode(request.user),  # `django.contrib.auth.User` instance.
            'auth': unicode(request.auth),  # None
        }
        return Response(content)
```

Utilisation de l'authentification de base à base de jeton

Django REST Framework fournit un mécanisme d'authentification basé sur des jetons de base qui doit être configuré en tant qu'application dans Django avant d'être utilisable, afin que les jetons soient créés dans la base de données et que leur cycle de vie soit géré.

Ajouter une authentification basée sur les jetons à settings.py

```
INSTALLED_APPS = (
    ...
    'rest_framework.authtoken'
)
```

Exécuter la migration de la base de données

```
./manage.py migrate
```

Créer des jetons pour vos utilisateurs

D'une manière ou d'une autre, vous devrez créer un jeton et le retourner:

```
def some_api(request):
    token = Token.objects.create(user=request.user)
    return Response({'token': token.key})
```

Il y a déjà un point de terminaison d'API dans l'application de jeton, de sorte que vous pouvez simplement ajouter ce qui suit à votre urls.py:

urls.py

```
from rest_framework.authtoken import views
urlpatterns += [
    url(r'^auth-token/', views.obtain_auth_token)
]
```

Les clients peuvent maintenant s'authentifier

Utiliser un en-tête d' `Authorization` comme:

```
Authorization: Token 123456789
```

Préfixé par un littéral "Token" et le jeton lui-même après les espaces.

Le littéral peut être modifié en sous- `TokenAuthentication` et en modifiant la variable de classe de `keyword` - `keyword` .

Si authentifié, `request.auth` contiendra l'instance `rest_framework.authtoken.models.BasicToken` .

Configuration de l'authentification OAuth2

OAuth n'est pas géré par Django REST Framework, mais quelques modules pip implémentent un client OAuth. La documentation de la structure REST propose l'un des modules suivants:

Django OAuth Toolkit

```
pip install django-oauth-toolkit
```

settings.py

```
INSTALLED_APPS = (
    ...
    'oauth2_provider',
)

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'oauth2_provider.ext.rest_framework.OAuth2Authentication',
    )
}
```

urls.py

```
urlpatterns = patterns(
    ...
    url(r'^o/', include('oauth2_provider.urls', namespace='oauth2_provider')),
)
```

Cadre de Django REST OAuth

```
pip install djangorestframework-oauth django-oauth2-provider
```

settings.py

```
INSTALLED_APPS = (
    ...
    'provider',
    'provider.oauth2',
)

REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.OAuth2Authentication',
    )
}
```

urls.py

```
urlpatterns = patterns(
    ...
    url(r'^oauth2/', include('provider.oauth2.urls', namespace='oauth2')),
)
```

Admin

Accédez au panneau d'administration et créez un nouveau `Provider.Client` pour avoir un `client_id` et un `client_secret` .

Utilisation d'une meilleure autorisation basée sur les jetons avec plusieurs jetons client

Le paquet le plus intéressant pour gérer les véritables jetons est django-rest-knox, qui prend en charge plusieurs jetons par utilisateur (et annule chaque jeton indépendamment), tout en prenant en charge l'expiration des jetons et plusieurs autres mécanismes de sécurité.

`django-rest-knox` dépend de la `cryptography` . Vous pouvez trouver plus d'informations sur la façon de l'installer à: http://james1345.github.io/django-rest-knox/installation/

Installer knox

```
pip install django-rest-knox
```

settings.py

```
INSTALLED_APPS = (
  ...
  'rest_framework',
  'knox',
  ...
)
```

Appliquer les migrations:

```
./manage.py migrate
```

Lire Authentification en ligne: https://riptutorial.com/fr/django-rest-framework/topic/6276/authentification

Chapitre 4: Authentification par jeton avec Django Rest Framework

Examples

AJOUT DE FONCTIONNALITÉ AUTH

Django REST Framework a déjà intégré certaines méthodes d'authentification, l'une d'elles étant basée sur Token, aussi la première chose à faire est de dire à notre projet que nous allons utiliser l'authentification de Rest Framework. Ouvrez le fichier `settings.py` et ajoutez la ligne en surbrillance.

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'rest_framework.authtoken',
    'test_app',
)
```

Comme nous avons ajouté une nouvelle application au projet, nous devons nous synchroniser

```
python manage.py migrate
```

CRÉER UN SUPERUSER À DJANGO

Pour utiliser l'authentification, nous pouvons nous appuyer sur le modèle des utilisateurs de django. La première chose à faire est de créer un superutilisateur.

```
python manage.py createsuperuser
```

Obtenir le jeton utilisateur

La fonctionnalité d'authentification par jeton assigne un jeton à un utilisateur. Ainsi, chaque fois que vous utilisez ce jeton, l'objet de la demande aura un attribut utilisateur contenant les informations du modèle utilisateur. Facile, n'est-ce pas?

Nous allons créer une nouvelle méthode POST pour renvoyer le jeton pour cet utilisateur, tant que la requête contient un utilisateur et un mot de passe corrects. Ouvrez **views.py** situé dans le dossier de l'application test_app.

```
from rest_framework.response import Response
```

```
from rest_framework.authtoken.models import Token
from rest_framework.exceptions import ParseError
from rest_framework import status

from django.contrib.auth.models import User


# Create your views here.
class TestView(APIView):
    """
    """

    def get(self, request, format=None):
        return Response({'detail': "GET Response"})

    def post(self, request, format=None):
        try:
            data = request.DATA
        except ParseError as error:
            return Response(
                'Invalid JSON - {0}'.format(error.detail),
                status=status.HTTP_400_BAD_REQUEST
            )
        if "user" not in data or "password" not in data:
            return Response(
                'Wrong credentials',
                status=status.HTTP_401_UNAUTHORIZED
            )

        user = User.objects.first()
        if not user:
            return Response(
                'No default user, please create one',
                status=status.HTTP_404_NOT_FOUND
            )

        token = Token.objects.get_or_create(user=user)

        return Response({'detail': 'POST answer', 'token': token[0].key})
```

UTILISER LE JETON

Créons une nouvelle vue qui nécessite ce mécanisme d'authentification.

Nous devons ajouter ces lignes d'importation:

```
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated
```

puis créez la nouvelle vue dans le même fichier `views.py`

```
class AuthView(APIView):
    """
    Authentication is needed for this methods
    """
    authentication_classes = (TokenAuthentication,)
```

```
    permission_classes = (IsAuthenticated,)

    def get(self, request, format=None):
        return Response({'detail': "I suppose you are authenticated"})
```

Comme nous l'avons fait dans le post précédent, nous devons dire à notre projet que nous avons un nouveau chemin REST qui écoute, sur `test_app/urls.py`

```
from rest_framework.urlpatterns import format_suffix_patterns
from test_app import views

urlpatterns = patterns('test_app.views',
    url(r'^', views.TestView.as_view(), name='test-view'),
    url(r'^auth/', views.AuthView.as_view(), name='auth-view'),
)

urlpatterns = format_suffix_patterns(urlpatterns)
```

Utiliser CURL

Si une boucle était exécutée sur ce noeud final

```
curl http://localhost:8000/auth/
op : {"detail": "Authentication credentials were not provided."}%
```

retournerait une **erreur 401 NON AUTORISE**

mais au cas où nous aurions un jeton avant:

```
curl -X POST -d "user=Pepe&password=aaaa"  http://localhost:8000/
{"token": "f7d6d027025c828b65cee5d38240aec60dffa150", "detail": "POST answer"}%
```

et puis nous mettons ce jeton dans l'en-tête de la requête comme ceci:

```
curl http://localhost:8000/auth/ -H 'Authorization: Token
f7d6d027025c828b65cee5d38240aec60dffa150'

op: {"detail": "I suppose you are authenticated"}%
```

Lire Authentification par jeton avec Django Rest Framework en ligne: https://riptutorial.com/fr/django-rest-framework/topic/9087/authentification-par-jeton-avec-django-rest-framework

Chapitre 5: Des filtres

Examples

Exemples de filtrage, du plus simple au plus complexe

Filtrage de vanille ordinaire

Pour filtrer une vue, remplacez sa méthode `get_queryset` pour retourner un ensemble de requêtes filtré

```
class HREmployees(generics.ListAPIView):
    def get_queryset(self):
        return Employee.objects.filter(department="Human Resources")
```

Toutes les fonctions de l'API utiliseront alors l'ensemble de requêtes filtré pour les opérations. Par exemple, la liste de la vue ci-dessus ne contiendra que des employés dont le département est Ressources humaines.

Accès aux paramètres de requête dans get_queryset

Le filtrage basé sur les paramètres de demande est facile.

`self.request` , `self.args` et `self.kwargs` sont disponibles et indiquent la requête en cours et ses paramètres pour le filtrage

```
def DepartmentEmployees(generics.ListAPIView):
    def get_queryset(self):
        return Employee.objects.filter(department=self.kwargs["department"])
```

Laisser les paramètres de l'API décider quoi filtrer

Si vous souhaitez davantage de flexibilité et permettre à l'appel de l'API de transmettre des paramètres pour filtrer la vue, vous pouvez associer des filtres backend comme Django Request Framework (installé via pip)

```
from rest_framework import filters
```

```
class Employees(generics.ListAPIView):
    queryset=Employee.objects.all()
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ("department", "role",)
```

Vous pouvez maintenant créer un appel API `/api/employees?department=Human Resources` et vous obtiendrez une liste des employés appartenant uniquement au service des ressources humaines, ou `/api/employees?role=manager&department=Human Resources` le département des ressources humaines.

Vous pouvez combiner le filtrage des ensembles de requêtes avec Django Filter Backend, sans problème. Les filtres fonctionneront sur l'ensemble de requêtes filtré renvoyé par `get_queryset`

```
from rest_framework import filters

class HREmployees(generics.ListAPIView):
    filter_backends = (filters.DjangoFilterBackend,)
    filter_fields = ("department", "role",)

    def get_queryset(self):
        return Employee.objects.filter(department="Human Resources")
```

Jeux de filtres

Jusqu'à présent, vous pouvez vous en tirer avec des correspondances de type simple dans les cas ci-dessus.

Mais que se passe-t-il si vous voulez quelque chose de plus complexe, comme une liste d'employés des RH âgés de 25 à 32 ans?

Réponse au problème: Filtres

Les ensembles de filtres sont des classes qui définissent comment filtrer les différents champs du modèle.

Définir comme ça

```
class EmployeeFilter(django_filters.rest_framework.FilterSet):
    min_age = filters.django_filters.NumberFilter(name="age", lookup_expr='gte')
    max_age = filters.django_filters.NumberFilter(name="price", lookup_expr='lte')

    class Meta:
        model = Employee
        fields = ['age', 'department']
```

`name` pointe vers le champ que vous souhaitez filtrer

`lookup_expr` fait essentiellement référence aux mêmes noms que vous utilisez lors du filtrage des ensembles de requêtes. Par exemple, vous pouvez faire une correspondance "commence par" en utilisant `lookup_expr="startswith"` qui est équivalent à
`Employee.objects.filter(department__startswith="Human")`

Ensuite, utilisez-les dans vos classes de vue en utilisant `filter_class` au lieu de `filter_fields`

```
class Employees(generics.ListAPIView):
    queryset=Employee.objects.all()
    filter_backends = (filters.DjangoFilterBackend,)
    filter_class = EmployeeFilter
```

Maintenant, vous pouvez faire `/api/employees?department=Human Resources&min_age=25&max_age=32`

Correspondances et relations non exactes

Les classes de filtre et les expressions sont très similaires à la manière dont vous spécifiez le filtrage dans les ensembles de requêtes

Vous pouvez utiliser la notation "__" pour filtrer les champs dans les relations. Par exemple, si le service était une clé étrangère de l'employé, vous pouvez ajouter

```
filter_fields=("department__name",)
```

et ensuite vous pouvez faire `/api/employees?department__name=Human Resources`

Plus élégamment, vous pouvez créer un jeu de filtres, ajouter une variable de filtre appelée `dept` et définir son nom sur `department__name` , ce qui vous permet de faire `/api/employees?dept=Human Resources`

Lire Des filtres en ligne: https://riptutorial.com/fr/django-rest-framework/topic/8144/des-filtres

Chapitre 6: Mixins

Introduction

Les classes mixin fournissent les actions utilisées pour fournir le comportement de base de la vue. Notez que les classes mixin fournissent des méthodes d'action plutôt que de définir les méthodes de gestionnaire, telles que `.get()` et `.post()` , directement. Cela permet une composition plus flexible du comportement. - Documentation du framework de repos officiel Django -

Examples

[Introduction] Liste des mixins et utilisation sur les vues / ensembles de vues

---

Liste des mixins disponibles:

- **ListModelMixin:** fournit une méthode `.list()` à la vue / vue
- **RetrieveModelMixin:** fournit une méthode `.retrieve()` à la vue / vue
- **CreateModelMixin:** fournit une méthode `.create()` à la vue / vue
- **UpdateModelMixin:** fournit une méthode `.update()` à la vue / vue
- **DestroyModelMixin:** fournit une méthode `.destroy()` à la vue / vue

---

Nous pouvons combiner les mixins dans nos vues et vues génériques, afin de leur donner l'utilitaire correspondant dont nous avons besoin:

1. Une vue API avec les `.list()` `.create()` et `.destroy()` ?
   `GenericAPIView` et combinez les mixins appropriés:

```
from rest_framework import mixins, generics

from myapp.models import MyModel
from myapp.serializers import MyModelSerializer

class MyCustomAPIView(mixins.ListModelMixin,
                      mixins.CreateModelMixin,
                      mixins.DestroyModelMixin,
                      generics.GenericAPIView):

    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer

    def get(self, request, *args, **kwargs):
        return self.list(request, *args, **kwargs)

    def post(self, request, *args, **kwargs):
        return self.create(request, *args, **kwargs)

    def delete(self, request, *args, **kwargs):
        return self.destroy(request, *args, **kwargs)
```

2. Un viewset avec seulement les `.list()` et `.update()` ?
   `GenericViewSet` et ajoutez les mixins appropriés:

```
from rest_framework import mixins

class MyCustomViewSet(mixins.ListModelMixin,
                      mixins.UpdateModelMixin,
                      viewsets.GenericViewSet):
    pass
```

   Oui, c'était aussi simple que ça !!

Pour conclure, nous pouvons combiner chaque mixin dont nous avons besoin et l'utiliser pour personnaliser nos vues et leurs méthodes dans toutes les combinaisons possibles!

[Intermédiaire] Créer des mixins personnalisés

DRF offre la possibilité de personnaliser davantage le comportement des vues / vues génériques en permettant la création de mixins personnalisés.

Comment:

Pour définir un mixin personnalisé, il suffit de créer une classe héritant d'un `object` .

Supposons que nous souhaitons définir deux vues distinctes pour un modèle nommé `MyModel` . Ces vues partageront le même ensemble de `queryset` et la même `queryset serializer_class` . Nous nous épargnerons de la répétition de code et nous placerons ce qui précède dans un seul mixin pour être hérité par nos vues:

- `my_app/views.py` (ce n'est pas la seule option de fichier disponible pour placer nos mixins personnalisés, mais c'est la moins complexe):

```
from rest_framework.generics import CreateAPIView, RetrieveUpdateAPIView
from rest_framework.permissions import IsAdminUser

class MyCustomMixin(object):
    queryset = MyModel.objects.all()
    serializer_class = MyModelSerializer

class MyModelCreateView(MyCustomMixin, CreateAPIView):
    """
    Only an Admin can create a new MyModel object
    """
    permission_classes = (IsAdminUser,)

    Do view staff if needed...

class MyModelCreateView(MyCustomMixin, RetrieveUpdateAPIView):
    """
    Any user can Retrieve and Update a MyModel object
    """
    Do view staff here...
```

Conclusion:

Les mixins sont essentiellement des blocs de code réutilisable pour notre application.

Lire Mixins en ligne: https://riptutorial.com/fr/django-rest-framework/topic/10083/mixins

Chapitre 7: Pagination

Introduction

La pagination est la division de jeux de données volumineux en pages séparées et autonomes.

Sur django rest framework, la pagination permet à l'utilisateur de modifier la quantité de données de chaque page et le style d'application du fractionnement.

Examples

[Introduction] Configuration du style de pagination à l'échelle mondiale

Afin de définir le style de pagination pour l'ensemble du projet, vous devez définir `DEFAULT_PAGINATION_CLASS` et `PAGE_SIZE` sur les paramètres du projet.

Pour ce faire, accédez à `settings.py` et sur la variable `REST_FRAMEWORK` , ajoutez les éléments suivants:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.DESIRED_PAGINATION_STYLE',
    'PAGE_SIZE': 100
}
```

Au lieu de `DESIRED_PAGINATION_STYLE` un des éléments suivants doit être placé:

- `PageNumberPagination` : accepte un numéro de `page` unique dans les paramètres de requête de la requête.

  ```
  http://your_api_url/a_table/?page=2
  ```

- `LimitOffsetPagination` : Accepte un paramètre `limit` , qui indique le nombre maximal d'éléments à renvoyer et un paramètre `offset` qui indique la position de départ de la requête par rapport au jeu de données. `PAGE_SIZE` n'est pas nécessaire de définir `PAGE_SIZE` pour ce style.

  ```
  http://your_api_url/a_table/?limit=50&offset=100
  ```

- `CursorPagination` : la pagination basée sur le curseur est plus complexe que les styles ci-dessus. Cela nécessite que le jeu de données présente un ordre fixe et ne permette pas au client de naviguer dans des positions arbitraires du jeu de données.

- Des styles de pagination personnalisés peuvent être définis à la place de ce qui précède.

[Intermédiaire] Remplacer le style de pagination et configurer la pagination

par classe

Remplacer le style de pagination:

Chaque style de pagination disponible peut être remplacé en créant une nouvelle classe qui hérite de l'un des styles disponibles, puis modifie ses paramètres:

```
class MyPagination(PageNumberPagination):
    page_size = 20
    page_size_query_param = 'page_size'
    max_page_size = 200
    last_page_strings = ('the_end',)
```

Ces paramètres (répertoriés dans la documentation officielle de pagination) sont les suivants:

PageNumberPagination

- `page_size` : Valeur numérique indiquant la taille de la page. Si défini, cela remplace le paramètre `PAGE_SIZE` . La valeur par défaut est la même que la `PAGE_SIZE` paramètres `PAGE_SIZE` .
- `page_query_param` : valeur de chaîne indiquant le nom du paramètre de requête à utiliser pour le contrôle de pagination.
- `page_size_query_param` : S'il est défini, il s'agit d'une valeur de chaîne indiquant le nom d'un paramètre de requête permettant au client de définir la taille de la page pour chaque demande. La valeur par défaut est `None` , indiquant que le client ne peut pas contrôler la taille de la page demandée.
- `max_page_size` : S'il est défini, il s'agit d'une valeur numérique indiquant la taille maximale autorisée de la page demandée. Cet attribut n'est valide que si `page_size_query_param` est également défini.
- `last_page_strings` : une liste ou un tuple de valeurs de chaîne indiquant les valeurs pouvant être utilisées avec `page_query_param` pour demander la dernière page de l'ensemble. Par défaut à `('last',)`
- `template` : nom d'un modèle à utiliser lors du rendu des contrôles de pagination dans l'API navigable. Peut être remplacé pour modifier le style de rendu ou défini sur `None` pour désactiver complètement les contrôles de pagination HTML. La valeur par défaut est `"rest_framework/pagination/numbers.html"` .

LimitOffsetPagination

- `default_limit` : Valeur numérique indiquant la limite à utiliser si une donnée n'est pas fournie par le client dans un paramètre de requête. La valeur par défaut est la même que la `PAGE_SIZE` paramètres `PAGE_SIZE` .
- `limit_query_param` : valeur de chaîne indiquant le nom du paramètre de requête "limit". Par défaut à `'limit'` .
- `offset_query_param` : Valeur de chaîne indiquant le nom du paramètre de requête "offset". La valeur par défaut est `'offset'` .
- `max_limit` : si elle est définie, il s'agit d'une valeur numérique indiquant la limite maximale autorisée pouvant être demandée par le client. La valeur par défaut est `None` .
- `template`

: Identique à *PageNumberPagination* .

CursorPagination

- `page_size` : identique à *PageNumberPagination* .
- `cursor_query_param` : Valeur de chaîne indiquant le nom du paramètre de requête "cursor". La valeur par défaut est `'cursor'` .
- `ordering` : Ce devrait être une chaîne, ou une liste de chaînes, indiquant le champ contre lequel la pagination basée sur le curseur sera appliquée. Par exemple: `ordering = 'slug'` . Par défaut à `-created` . Cette valeur peut également être remplacée en utilisant `OrderingFilter` dans la vue.
- `template` : Identique à *PageNumberPagination* .

Configuration de la pagination par classe:

En plus de la possibilité de configurer globalement le style de pagination, une configuration par classe est disponible:

```
class MyViewSet(viewsets.GenericViewSet):
    pagination_class = LimitOffsetPagination
```

Maintenant, seul `MyViewSet` a une pagination `LimitOffsetPagination` .

Un style de pagination personnalisé peut être utilisé de la même manière:

```
class MyViewSet(viewsets.GenericViewSet):
    pagination_class = MyPagination
```

Exemple d'utilisation complexe [intermédiaire]

Supposons que nous ayons une api complexe, avec de nombreuses vues génériques et des sets de vues génériques. Nous voulons activer `PageNumberPagination` sur toutes les vues, sauf une (vue générique ou vue, ne fait aucune différence) pour laquelle nous voulons une casse personnalisée de `LimitOffsetPagination` .

Pour y parvenir, nous devons:

1. Sur les `settings.py` nous placerons notre pagination par défaut afin de l'activer pour chaque vue / vue générique et nous `PAGE_SIZE` sur 50 éléments:

   ```
   REST_FRAMEWORK = {
       'DEFAULT_PAGINATION_CLASS':
           'rest_framework.pagination.PageNumberPagination',
       'PAGE_SIZE': 50
   }
   ```

2. Maintenant, dans nos `views.py` (ou dans un autre `.py` ex: `paginations.py`), nous devons remplacer la `LimitOffsetPagination` :

```
from rest_framework.pagination import LimitOffsetPagination

class MyOffsetPagination(LimitOffsetPagination):
    default_limit = 20
    max_limit = 1000
```

Un `LimitOffsetPagination` personnalisé avec `PAGE_ZISE` de 20 éléments et une `limit` maximale de 1000 éléments.

3. Dans nos `views.py` , nous devons définir la `views.py` `pagination_class` de notre vue spéciale:

```
imports ...

# ====================================
#    PageNumberPagination classes
# ====================================

class FirstView(generics.ListAPIView):
    ...

class FirstViewSet(viewsets.GenericViewSet):
    ...

...

# ====================================
#     Our custom Pagination class
# ====================================

class IAmSpecialView(generics.ListAPIView):
    pagination_class = MyOffsetPagination
    ...
```

Maintenant, toutes les vues / vues génériques de l'application ont `PageNumberPagination` , à l'exception de la classe `IAmSpecial` , qui est en effet *spéciale* et possède sa propre `LimitOffsetPagination` personnalisée.

[Avancé] Pagination sur les vues / vues non génériques

C'est un sujet avancé, n'essayez pas de comprendre d'abord les autres exemples de cette page .

Comme indiqué dans la structure de Django Rest Framework officielle sur la pagination :

> La pagination est effectuée automatiquement uniquement si vous utilisez les vues ou les vues génériques. Si vous utilisez un APIView standard, vous devrez appeler l'API de pagination vous-même pour vous assurer de recevoir une réponse paginée. Voir le code source des classes mixins.ListModelMixin et generics.GenericAPIView pour un exemple.

Mais que faire si nous voulons utiliser la pagination sur un view / viewset non générique?

Eh bien, descendons le lapin:

1. Le premier arrêt est le dépôt officiel de Django Rest Framework et plus précisément le

django-rest-framework / rest_framework / generics.py . La ligne spécifique sur laquelle pointe ce lien nous montre comment les développeurs du framework gèrent la pagination dans leurs génériques.
C'est exactement ce que nous allons utiliser à notre avis également!

2. Supposons que nous ayons une configuration de pagination globale semblable à celle présentée dans l' exemple introductif de cette page et supposons également que nous ayons un `APIView` auquel nous voulons appliquer la pagination.

3. Alors sur `views.py` :

```
from django.conf import settings
from rest_framework.views import APIView

class MyView(APIView):
    queryset = OurModel.objects.all()
    serializer_class = OurModelSerializer
    pagination_class = settings.DEFAULT_PAGINATION_CLASS # cool trick right? :)

    # We need to override get method to achieve pagination
    def get(self, request):
        ...
        page = self.paginate_queryset(self.queryset)
        if page is not None:
            serializer = self.serializer_class(page, many=True)
            return self.get_paginated_response(serializer.data)

        ... Do other stuff needed (out of scope of pagination)

    # Now add the pagination handlers taken from
    #  django-rest-framework/rest_framework/generics.py

    @property
    def paginator(self):
        """
        The paginator instance associated with the view, or `None`.
        """
         if not hasattr(self, '_paginator'):
             if self.pagination_class is None:
                 self._paginator = None
             else:
                 self._paginator = self.pagination_class()
         return self._paginator

     def paginate_queryset(self, queryset):
         """
         Return a single page of results, or `None` if pagination is disabled.
         """
         if self.paginator is None:
             return None
         return self.paginator.paginate_queryset(queryset, self.request, view=self)

     def get_paginated_response(self, data):
         """
         Return a paginated style `Response` object for the given output data.
         """
         assert self.paginator is not None
         return self.paginator.get_paginated_response(data)
```

Maintenant, nous avons un `APIView` qui gère la pagination!

[Intermédiaire] Pagination sur une vue basée sur une fonction

Nous avons vu dans ces exemples (ex_1 , ex_2) comment utiliser et remplacer les classes de pagination dans n'importe quelle vue de base de classe générique.
Que se passe-t-il lorsque nous voulons utiliser la pagination dans une vue basée sur une fonction?

Supposons également que nous souhaitons créer une vue basée sur les fonctions pour `MyModel` avec `PageNumberPagination` , répondant uniquement à une requête `GET` . Alors:

```
from rest_framework.pagination import PageNumberPagination


@api_view(['GET',])
def my_function_based_list_view(request):
    paginator = PageNumberPagination()
    query_set = MyModel.objects.all()
    context = paginator.paginate_queryset(query_set, request)
    serializer = MyModelSerializer(context, many=True)
    return paginator.get_paginated_response(serializer.data)
```

Nous pouvons également faire la même chose pour une pagination personnalisée en modifiant cette ligne:

```
paginator = PageNumberPagination()
```

pour ça

```
paginator = MyCustomPagination()
```

à condition que nous ayons défini `MyCustomPagination` pour remplacer la pagination par défaut

Lire Pagination en ligne: https://riptutorial.com/fr/django-rest-framework/topic/9950/pagination

Chapitre 8: Routeurs

Introduction

Le routage consiste à mapper la logique (méthodes d'affichage, etc.) sur un ensemble d'URL. L'infrastructure REST prend en charge le routage automatique des URL vers Django.

Syntaxe

- router = routers.SimpleRouter ()
- router.register (préfixe, viewset)
- router.urls # l'ensemble des URL générées pour le vueset enregistré.

Examples

[Introduction] Utilisation de base, SimpleRouter

Le routage automatique pour le DRF peut être réalisé pour les classes `ViewSet` .

1. Supposons que la classe ViewSet s'appelle `MyViewSet` pour cet exemple.

2. Pour générer le routage de `MyViewSet` , le `SimpleRouter` sera utilisé.
   Sur `myapp/urls.py` :

   ```
   from rest_framework import routers

   router = routers.SimpleRouter()         # initialize the router.
   router.register(r'myview', MyViewSet)   # register MyViewSet to the router.
   ```

3. Cela générera les modèles d'URL suivants pour `MyViewSet` :

   - `^myview/$` avec le nom `myview-list` .
   - `^myview/{pk}/$` avec le nom `myview-detail`

4. Enfin, pour ajouter les patterns générés dans les modèles d'URL de `myapp` , le fichier `include()` de django sera utilisé.
   Sur `myapp/urls.py` :

   ```
   from django.conf.urls import url, include
   from rest_framework import routers

   router = routers.SimpleRouter()
   router.register(r'myview', MyViewSet)

   urlpatterns = [
       url(r'other/prefix/if/needed/', include(router.urls)),
   ]
   ```

Lire Routeurs en ligne: https://riptutorial.com/fr/django-rest-framework/topic/10938/routeurs

Chapitre 9: Sérialiseurs

Examples

Accélérer les requêtes de sérialiseurs

Disons que nous avons un modèle de `Travel` avec de nombreux domaines connexes:

```
class Travel(models.Model):

    tags = models.ManyToManyField(
        Tag,
        related_name='travels', )
    route_places = models.ManyToManyField(
        RoutePlace,
        related_name='travels', )
    coordinate = models.ForeignKey(
        Coordinate,
        related_name='travels', )
    date_start = models.DateField()
```

Et nous voulons construire CRUD dans `/travels` via la vue ViewSet.
Voici la vue simple:

```
class TravelViewset(viewsets.ModelViewSet):

    queryset = Travel.objects.all()
    serializer_class = TravelSerializer
```

Le problème avec ce ViewSet est que nous avons beaucoup de champs liés dans notre modèle de `Travel` , donc Django va frapper db pour chaque instance de `Travel` . Nous pouvons appeler directement select_related et prefetch_related dans l'attribut `queryset` , mais si nous souhaitons séparer les sérialiseurs pour les actions de `list` , de `retrieve` , de `create` .. de ViewSet.
On peut donc mettre cette logique dans un mixin et en hériter:

```
class QuerySerializerMixin(object):
    PREFETCH_FIELDS = [] # Here is for M2M fields
    RELATED_FIELDS = [] # Here is for ForeignKeys

    @classmethod
    def get_related_queries(cls, queryset):
        # This method we will use in our ViewSet
        # for modify queryset, based on RELATED_FIELDS and PREFETCH_FIELDS
        if cls.RELATED_FIELDS:
            queryset = queryset.select_related(*cls.RELATED_FIELDS)
        if cls.PREFETCH_FIELDS:
            queryset = queryset.prefetch_related(*cls.PREFETCH_FIELDS)
        return queryset


    class TravelListSerializer(QuerySerializerMixin, serializers.ModelSerializer):
```

```
        PREFETCH_FIELDS = ['tags'']
        RELATED_FIELDS = ['coordinate']
        # I omit fields and Meta declare for this example


    class TravelRetrieveSerializer(QuerySerializerMixin, serializers.ModelSerializer):

        PREFETCH_FIELDS = ['tags', 'route_places']
```

`ViewSet` maintenant notre `ViewSet` avec de nouveaux sérialiseurs

```
class TravelViewset(viewsets.ModelViewSet):

    queryset = Travel.objects.all()

    def get_serializer_class():
        if self.action == 'retrieve':
            return TravelRetrieveSerializer
        elif self.action == 'list':
            return TravelListSerializer
        else:
            return SomeDefaultSerializer


    def get_queryset(self):
        # This method return serializer class
        # which we pass in class method of serializer class
        # which is also return by get_serializer()
        q = super(TravelViewset, self).get_queryset()
        serializer = self.get_serializer()
        return serializer.get_related_queries(q)
```

Sérialiseurs imbriqués pouvant être mis à jour

Les sérialiseurs imbriqués par défaut ne prennent pas en charge la création et la mise à jour. Pour supporter cela sans dupliquer la logique de création / mise à jour DRF, il est important de *supprimer* les données imbriquées de `validated_data` avant de déléguer à `super` :

```
# an ordinary serializer
class UserProfileSerializer(serializers.ModelSerializer):
    class Meta:
        model = UserProfile
        fields = ('phone', 'company')


class UserSerializer(serializers.ModelSerializer):
    # nest the profile inside the user serializer
    profile = UserProfileSerializer()

    class Meta:
        model = UserModel
        fields = ('pk', 'username', 'email', 'first_name', 'last_name')
        read_only_fields = ('email', )

    def update(self, instance, validated_data):
        nested_serializer = self.fields['profile']
        nested_instance = instance.profile
```

```
        # note the data is `pop`ed
        nested_data = validated_data.pop('profile')
        nested_serializer.update(nested_instance, nested_data)
        # this will not throw an exception,
        # as `profile` is not part of `validated_data`
        return super(UserDetailsSerializer, self).update(instance, validated_data)
```

Dans le cas de `many=True` , Django se plaindra que `ListSerializer` ne supporte pas la `update` . Dans ce cas, vous devez gérer vous-même la sémantique de la liste, mais vous pouvez toujours déléguer à `nested_serializer.child` .

Ordre de validation du sérialiseur

Dans DRF, la validation du sérialiseur est exécutée dans un ordre spécifique, non documenté

1. Désérialisation du champ appelée (`serializer.to_internal_value` et `field.run_validators`)
2. `serializer.validate_[field]` est appelé pour chaque champ.
3. Les validateurs au niveau du sérialiseur sont appelés (`serializer.run_validation` suivi de `serializer.run_validators`)
4. Enfin, `serializer.validate` est appelé pour terminer la validation.

Obtenir la liste de tous les objets enfants associés dans le sérialiseur parent

Supposons que nous implémentions une API simple et que nous ayons les modèles suivants.

```
class Parent(models.Model):
    name = models.CharField(max_length=50)


class Child(models.Model):
    parent = models.ForeignKey(Parent)
    child_name = models.CharField(max_length=80)
```

Et nous voulons retourner une réponse lorsqu'un `parent` particulier est récupéré via l'API.

```
{
'url': 'https://dummyapidomain.com/parents/1/',
'id': '1',
'name': 'Dummy Parent Name',
'children': [{
        'id': 1,
        'child_name': 'Dummy Children I'
    },
    {
        'id': 2,
        'child_name': 'Dummy Children II'
    },
    {
        'id': 3,
        'child_name': 'Dummy Children III'
    },
    ...
],
```

}

Pour cela, nous implémentons les sérialiseurs correspondants comme ceci:

```
class ChildSerializer(serializers.HyperlinkedModelSerializer):

    parent_id =
serializers.PrimaryKeyRelatedField(queryset=Parent.objects.all(),source='parent.id')

    class Meta:
        model = Child
        fields = ('url','id','child_name','parent_id')

    def create(self, validated_data):
        subject = Child.objects.create(parent=validated_data['parent']['id'],
child_name=validated_data['child_name'])

        return child


class ParentSerializer(serializers.HyperlinkedModelSerializer):
    children = ChildSerializer(many=True, read_only=True)
    class Meta:
        model = Course
        fields = ('url','id','name','children')
```

Pour faire ce travail de mise en œuvre correctement , nous devons mettre à jour notre `Child` modèle et ajouter un **related_name** au `parent` champ. La version mise à jour de notre implémentation de modèle `Child` devrait être comme ceci:

```
class Child(models.Model):
parent = models.ForeignKey(Parent, related_name='children')  # <--- Add related_name here
child_name = models.CharField(max_length=80)
```

En faisant cela, nous pourrons obtenir la liste de tous les objets enfants associés dans le sérialiseur du parent.

Lire Sérialiseurs en ligne: https://riptutorial.com/fr/django-rest-framework/topic/2377/serialiseurs

Chapitre 10: Sérialiseurs

Introduction

Selon la documentation officielle de DRF, les sérialiseurs permettent de convertir des données complexes, telles que des ensembles de requêtes et des instances de modèles, en types de données python natifs, afin de les rendre au format JSON, XML et autres types de contenu.

Les sérialiseurs dans DRF sont plus comme django **Form** et **ModelForm** . **La** classe **Serializer** nous fournit un moyen personnalisé de gérer les données comme django Form. Et la classe **ModelSerializer** nous permet de gérer facilement les données basées sur des modèles.

Examples

Introduction de base

Disons que nous avons un modèle appelé produit.

```
class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.IntegerField()
```

Nous allons maintenant déclarer un modèle de sérialiseurs pour ce modèle.

```
from rest_framework.serializers import ModelSerializer

class ProductSerializers(ModelSerializer):
    class Meta:
        model = Product
        fields = '__all__'
        read_only_fields = ('id',)
```

Par cette classe ProductSerializers, nous avons déclaré un modèle de sérialiseurs. Dans la classe Meta, par variable de modèle, nous avons indiqué au ModelSerializer que notre modèle serait le modèle de produit. Par champs variables, nous avons dit que ce sérialiseur devait inclure tout le champ du modèle. Enfin, par la variable read_only_fields, nous avons indiqué que id sera un champ en lecture seule, il ne peut pas être modifié.

Voyons ce qu'il y a dans notre sérialiseur. Au début, importez le sérialiseur dans la ligne de commande et créez une instance, puis imprimez-la.

```
>>> serializer = ProductSerializers()
>>> print(serializer)
ProductSerializers():
        id = IntegerField(label='ID', read_only=True)
        name = CharField(max_length=100)
        price = IntegerField(max_value=2147483647, min_value=-2147483648)
```

Ainsi, notre sérialiseur saisit tout le champ de notre modèle et crée tout le champ à sa manière.

Nous pouvons utiliser ProductSerializers pour sérialiser un produit ou une liste de produits.

```
>>> p1 = Product.objects.create(name='alu', price=10)
>>> p2 = Product.objects.create(name='mula', price=5)
>>> serializer = ProductSerializers(p1)
>>> print(serializer.data)
{'id': 1,'name': 'alu', 'price': 10}
```

À ce stade, nous avons traduit l'instance de modèle en types de données natifs Python. Pour finaliser le processus de sérialisation, nous rendons les données en json.

```
>>> from rest_framework.renderers import JSONRenderer
>>> serializer = ProductSerializers(p1)
>>> json = JSONRenderer().render(serializer.data)
>>> print(json)
'{"id": 1,"name": "alu", "price": 10}'
```

Lire Sérialiseurs en ligne: https://riptutorial.com/fr/django-rest-framework/topic/8871/serialiseurs

Chapitre 11: Utiliser django-rest-framework avec AngularJS comme framework frontal.

Introduction

Dans cette rubrique, nous examinerons comment utiliser Django REST Framework en tant qu'API backend pour une application AngularJS. Les principaux problèmes liés à l'utilisation conjointe de DRF et d'AngularJS concernent généralement la communication HTTP entre les deux technologies, ainsi que la représentation des données aux deux extrémités, et enfin la manière de déployer et d'architecter l'application / le système.

Examples

DRF Voir

```
class UserRegistration(APIView):
    def post(self, request, *args, **kwargs):
        serializer = UserRegistrationSerializer(data=request.data)
        serializer.is_valid(raise_exception=True)
        serializer.save()
        return Response(serializer.to_representation(instance=serializer.instance),
status=status.HTTP_201_CREATED)
```

Demande angulaire

```
$http.post("\<domain\>/user-registration/", {username: username, password: password})
     .then(function (data) {
         // Do success actions here
     });
```

Lire Utiliser django-rest-framework avec AngularJS comme framework frontal. en ligne: https://riptutorial.com/fr/django-rest-framework/topic/10893/utiliser-django-rest-framework-avec-angularjs-comme-framework-frontal-

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec django-rest-framework	Abhishek, Chitrank Dixit, Community, davyria, itmard, Majid, Rahul Gupta, Saksow, ssice
2	Authentification	iankit, itmard
3	Authentification par jeton avec Django Rest Framework	Ali_Waris, Cadmus
4	Des filtres	Bitonator
5	Mixins	John Moutafis
6	Pagination	John Moutafis
7	Routeurs	John Moutafis
8	Sérialiseurs	hnroot, Ivan Semochkin, Louis Barranqueiro, Silly Freak
9	Utiliser django-rest-framework avec AngularJS comme framework frontal.	mattjegan