

Blog de Jean-Loup Adde

[Accueil](#) > [Posts](#) > Créez aisément une API rest grâce au django rest framework !

Créez aisément une API rest grâce au django rest framework !

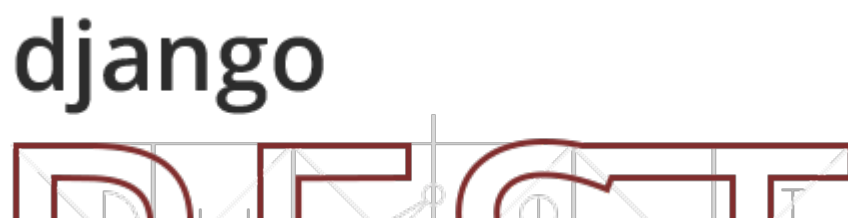
Read in english: Create easily a REST API with the [django rest framework](#)!

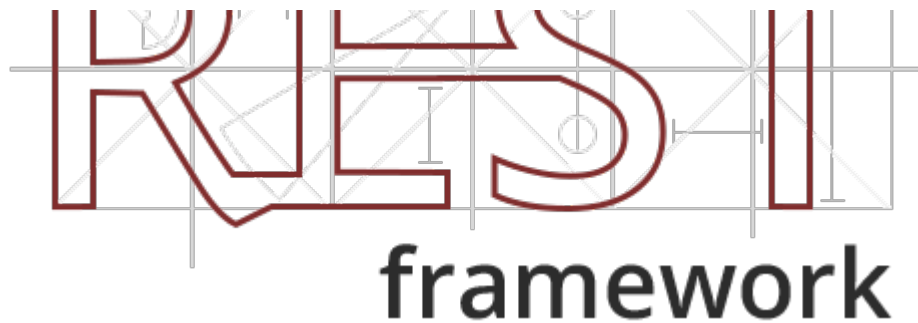
07/02/2016 - [Posts](#) - [#Django](#) [#django REST framework](#) - 2591 mots - 12min

Sommaire

- [Qu'est qu'une API REST ?](#)
 - [Pourquoi faire ?](#)
- [Le django rest framework](#)
 - [Préquel](#)
 - [Installation](#)
 - [Les viewsets](#)
 - [Les serializers](#)
 - [Les routers](#)
- [Bonus: Ajoutons de la doc à notre API](#)
- [The End](#)

Salut ! Vous êtes sur le point de créer une nouvelle API. Vous hésitez encore sur quel framework choisir ? Si cette API est un ajout à votre projet django, le django rest framework sera parfait pour créer en quelques lignes une API REST, documentée, accessible et performante.





Qu'est qu'une API REST ?

Le REST va permettre de nous orienter pûrement vers une API décrivant les ressources utilisées dans notre application Django. Le grand avantage d'utiliser une API REST est que nous avons une séparation entre Client et Serveur. Ainsi de multiples Clients peuvent être développés afin de consommer les ressources mises à disposition par l'API.

Le grand avantage de REST est qu'il permet de requêter l'API avec de simples requêtes HTTP. Les requêtes GET vont permettre de récupérer des ressources, les requêtes POST d'en modifier, les requêtes DELETE d'en supprimer, etc..

Si jamais vous voulez en savoir plus sur l'architecture REST, je vous conseille [cet article](#)

Pourquoi faire ?

Et bien, si jamais vous voulez mettre à disposition des données pour quelles soient utilisées sur d'autres plateformes ou que vos données interagissent avec d'autres. Par exemple ce qui est cool avec toutes les APIs maintenant disponibles sur le web est que par exemple si vous voulez faire interagir Trello avec twitter vous pouvez, Il vous suffit de créer le pont entre les deux. Ici, nous créerons une simple API en lecture seulement afin de récupérer toutes

les données disponibles sur le blog :

Articles

Catégories

Tags

Le django rest framework

Le django rest framework va nous permettre de créer très facilement et en très peu de lignes de code une API REST au dessus de notre application django. Donc je vous conseille éperdument ce framework si vous avez un projet django existant ! Si vous commencez from scratch, il vous suffira de créer les modèles ce qui n'est pas non plus très complexe :).

Préquel

Pour ce petit tour d'intro, nous partirons d'un projet existant (ce blog). Ce dont vous avez réellement besoin est d'avoir un `models.py` avec quelques entités et ça devrait faire l'affaire ;)









Dans mon cas j'ai trois classes dans mon `models.py`:

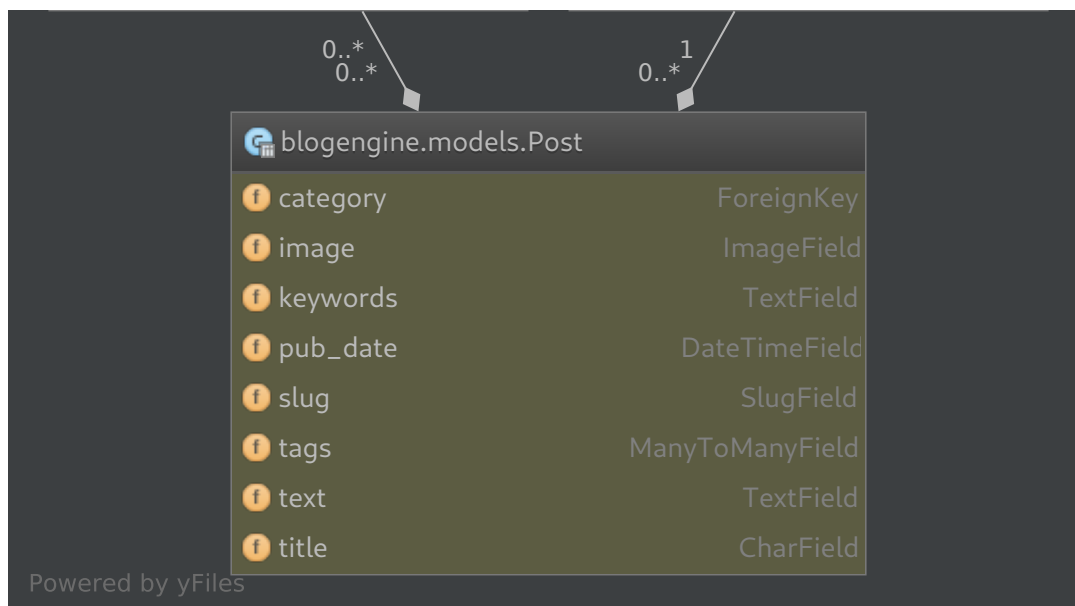
Post (Article)

Category

Tag

Si jamais vous voulez un petit schéma, ils ont cette gueule là:

blogengine.models.Tag	blogengine.models.Category
 description	 description
 name	 name
 post_set	 post_set
 slug	 slug
TextField	TextField
CharField	CharField
QuerySet	QuerySet
SlugField	SlugField



Installation

Pour l'installation vous avez deux choix, installer depuis pip ou de l'installer depuis github. On va se contenter de la première. Activer votre virtualenv, nous allons commencer !

- 1 pip install.djangorestframework
- 2 pip install markdown
- 3 pip install django-filter

Ou sinon vous pouvez les ajouter dans votre fichier requirements.txt de votre projet.

Maintenant créons une nouvelle application pour développer notre API !

Ouvrez un terminal, déplacez vous à la racine de votre projet django et tapez:

```
1python manage.py startapp api
```

Si tout c'est bien passé, vous devriez avoir un dossier avec un dossier migrations, et des fichiers models.py, views.py, apps.py, tests.py, admin.py.

Vous pouvez supprimer les fichiers `models.py`, `admin.py` et le dossier `migrations`.

Les viewsets

Les viewsets vont jouer le rôle de vues comme dans django. Elles vont permettre de requêter la base de données en fonction d'une requête donnée, on peut les comparer à des controllers dans le pattern MVC.

Dans ces viewsets vous aurez donc des méthodes telles que :

get
post
list
create

Mais ce qui est encore mieux avec les viewset, c'est que vous en avez pour tous les goûts ! `Generic`, `ModelViewSet`, `ReadOnlyModelViewSet`, etc..

Dans mon cas, j'utiliserai des `ReadOnlyModelViewSet` afin d'avoir un viewset spécifique à mon modèle et limitant les actions sur cette entrée de l'API à la lecture. (Ce serait dommage que tout le monde puisse poster des articles :/)

Passons au code !

Dans `api/views.py`

```
1 # api/views.py
2 from blogengine import models
3 from api import serializers
4
5
```

```
6 class PostViewSet(ReadOnlyModelViewSet):
7     """
8     A simple viewset to retrieve all the posts of blog.juanw
9     """
10    queryset = models.Post.objects.all()
11    serializer_class = serializers.PostSerializer
12    # On va créer le serializer juste après.
```

Les serializers

Les serializers vont nous permettre de serialiser les instances en JSON et transformer le JSON en instance python. On peut les comparer à des formulaires en django.

Nous allons ajouter au sein de notre serializer, les champs que nous avons besoin. On pourrait donc écrire un serializer de cette façon :

```
1 # api/serializers.py
2
3 from rest_framework import serializers
4 from blogengine import models
5
6
7 class PostSerializer(serializers.Serializer):
8     title = serializers.CharField(max_length=255)
9     text = serializers.TextField()
10    pub_date = serializers.DateField()
11    # Et tous les autres champs ...
```

Voilà ce que l'on aurait pu écrire si les ModelSerializer n'existaient pas ! On va juste avoir besoin de spécifier le modèle et le djrf va s'occuper de créer les

champs à la volée. Plutôt cool:

```
1 # api/serializers.py
2
3
4 from rest_framework import serializers
5 from blogengine import models
6
7
8 class PostSerializer(serializers.ModelSerializer):
9
10     class Meta:
11         model = models.Post
12         # Vous pouvez ajouter un fields pour filtrer les
13         # champs du modèle à sérialiser
14         fields = ('title', 'text', 'pub_date')
```

Ce qui rend le code tout de suite plus lisible :) (Perso, j'ai enlevé le fields afin d'avoir tous les attributs de mon modèles dans l'API.

Les Serializers imbriqués

Dans le cas du blog, on peut voir qu'un post peut être lié à plusieurs tags et est lié à une catégorie. On pourrait vouloir imbriqué cette catégorie et ces tags au sein de l'objet JSON en question. Pour cela nous pouvons :

Créer un serializer pour l'entité imbriquée que l'on ajoutera dans le serializer de Post

Utiliser un secret ancestral japonais transmis de génération en génération dans la famille REST

Commençons par le premier point :

Créer un serializer qui sera imbriqué directement dans notre PostSerializer !

```
1 class CategorySerializer(serializers.ModelSerializer):
2     class Meta:
3         model = models.Category
4         fields = ('id', 'name', 'description')
5
6
7 class PostSerializer(serializers.ModelSerializer):
8     category = CategorySerializer()
9
10    class Meta:
11        model = models.Post
12        fields = ('title', 'text', 'category')
13
```

```
1 {
2     "id": 12,
3     "pub_date": "2015-12-26T15:24:10Z",
4     "title": "Sky Force Anniversary, an old style shoot'em u
5     "text": "blablablba",
6     "category": {
7         'id': 2
8         'name': 'Video Games',
9         'description': "You'll find here all the posts [...]"
10    }
11 }
```

On reste dans le basique mais on pourra par exemple ajouter des champs

spécifiques (des [SerializerMethodField](#) par exemple).

Et la seconde et dernière méthode que vous attendez tous. La méthode qui révolutionne le monde de la création de l'API depuis maintenant, pfiou, longtemps. Je veux bien sûr parler de 'depth'.

```
1
2 class PostSerializer(serializers.ModelSerializer):
3     category = CategorySerializer()
4
5     class Meta:
6         model = models.Post
7         depth = 1
8
```

Cet attribut de classe vous permet de spécifier la profondeur de sérialization que vous voulez pour ce modèle. Ici le définir à 1 va permettre d'inclure la catégorie (avec tous ces champs) directement dans les posts.

WARNING

Faites attention ! Ici, nous parlons seulement de sérialization. Un nombre important de requête en base de données vont être exécutées afin de récupérer ces éléments imbriqués. Je vous conseille fortement de lire la documentation sur les [prefetch_related](#) et [select_related](#) de django.

Exemple: Dans l'exemple précédent où nous essayons d'intégrer la catégorie aux articles, nous allons avoir n requêtes faites en bases de plus pour n articles au sein du JSON de retour. Pourquoi ? Pour l'affichage de votre json, le django rest framework va faire:

```
1 # 1
2 nested_obj = post.category
3 # 2
4 DefaultSerializer(nested_obj)
```

Bon j'improvise un peu le truc, mais c'est comme ça que ça se passe en interne. Ce qui implique que django va exécuter une requête SQL quand la ligne `obj = post.category` va être exécutée. Pour palier à ce problème nous devons dire à django de précharger l'élément imbriqué quand nous requérons la base de données pour récupérer les articles.

Pour cela, rendons nous dans le fichier `api/views.py`.

```
1 # api/views.py
2
3 class PostViewSet(ReadOnlyModelViewSet):
4     """
5     A simple viewset to retrieve all the posts of blog.juanwo
6     """
7     # queryset = models.Post.objects.all() # La précédente q
8     queryset = models.Post.objects.all().select_related('ca
9     serializer_class = serializers.PostSerializer
```

Ainsi nous n'aurons plus n requetes mais seulement 1 car django va s'occuper de faire un JOIN SQL sur la table catégorie !

Les routers

Les routeurs vont nous permettre d'ajouter des urls aisément sans avoir à les taper une à une.

Nous aurons seulement besoin de spécifier la viewset et le nom qu'on lui attribue et le django rest framework va s'occuper de créer les urls automatiquement.

Si l'on crée au sein de notre fichier api/urls.py un routeur tel que:

```
1 api/urls.py
2
3 from rest_framework import routers
4 from api import views
5
6 router = routers.SimpleRouter()
7 router.register(r'posts', views.PostViewSet)
8
9 urlpatterns = router.urls
```

Nous allons avoir 2 urls créées automatiquement !

'^posts/\$' avec le nom 'post-list'

'^posts/{pk}\$' avec le nom 'post-detail'

On aura ainsi un fichier urls.py plus léger et plus facile à maintenir que pour un projet django normal :)

Bonus: Ajoutons de la doc à notre API

Afin que tout développeur puisse consommer (aisément) notre API, il est important d'y apporter de la documentation.

Pour cela, nous allons installer swagger qui est un générateur automatique de documentation. Il va permettre de montrer les urls disponibles pour les

requêtes HTTP, mais aussi le type de retour de chaque champs de notre JSON, etc, etc...

Pour cela, on va commencer par ajouter `django_rest_swagger` dans notre `requirements.txt` ou avec `pip` c'est comme vous voulez.

En suite on va ajouter `'rest_framework_swagger'` à notre liste d'applications `django` dans le `settings.py`.

Et pour finir, nous allons inclure les urls swagger dans notre `urls.py`

```
1 #juanwolf_s_blog/urls.py
2 urlpatterns = [
3     '',
4     r'^api/docs/', include('rest_framework_swagger.urls')),
5     r'^api/', include('api.urls'))
6 ]
```

Et bim voilà le résultat ! <https://blog.juanwolf.fr/api/docs/>

The End

C'est terminé pour notre petit tour d'horizon du `django rest framework`, j'espère que ça vous a plu ! Si jamais vous voulez entrer plus en détail sur le sujet, je vous invite à vous rendre sur [le site du django rest framework](#) qui est très bien documenté ! Sur ce, Codez bien !

[GitHub](#)[GitLab](#)[Twitch](#)[RSS](#)

Jean-Loup Adde 2020 © **CC BY 4.0**