Article
Guillaume Rongier · Août 31, 2022 6m de lecture

Open Exchange

Découvrir Django, partie 3

Nous continuons à observer les possibilités de Django, et son utilisation avec IRIS. Dans la première partie nous avons regardé comment définir des modèles et se connecter à des tableaux déjà existants dans IRIS, dans la suite nous avons étendu le portail d'administration intégré de Django, avec la possibilité de voir quelles données nous avons dans ces modèles, avec des filtres, l'édition et même la pagination.

Il est temps de passer à l'action, nous allons maintenant créer une API REST, sur Django, basée sur les mêmes données, que nous avons utilisées auparavant à partir du paquet posts-and-tags.

Pour ce faire, nous utiliserons Django REST Framework

Le cadre REST de Django est une boîte à outils puissante et flexible permettant de créer des API Web.

Quelques raisons pour lesquelles vous pourriez vouloir utiliser le cadre REST :

- L'API navigable sur le Web est un avantage considérable pour vos développeurs.
- Politiques d'authentification comprenant des paquets pour OAuth1a et OAuth2.
- Sérialisation qui prend en charge les sources de données ORM et non ORM.
- Personnalisable jusqu'en bas - utilisez simplement des visualisations régulières basées sur des fonctions si vous n'avez pas besoin des fonctionnalités les plus puissantes.
- Une documentation complète et un support communautaire important.
- Utilisé par des entreprises de renommée internationale, telles que Mozilla, Red Hat, Heroku et Eventbrite, qui lui font confiance.

Tout d'abord, nous devons mettre à jour notre fichier requirements.txt avec les dépendances

```
# Django itself
django>=4.0.0
```

```
# Pilote InterSystems IRIS pour Django, et pilote DB-API d'InterSystems
django-iris=>0.1.13
https://raw.githubusercontent.com/intersystems-community/iris-driver-
distribution/main/DB-API/intersystems_irispython-3.2.0-py3-none-any.whl

# Cadre REST de Django et ses dépendances facultatives
djangorestframework>=3.4.4
# Support du format Markdown pour l'API navigable.
markdown>=3.0.0
# Support de filtrage
django-filter>=1.0.1
```

Et installez-les

```
python install -r requirements.txt
```

Première version de l'API

Pour cela, mettez à jour le fichier *urls.py*. Ici on dit, cela de la racine pour notre api comme ***api/***, donc toutes les demandes à http://localhost:8000/api/ seront utilisées comme racine pour nos demandes API

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers

router = routers.DefaultRouter()

urlpatterns = [
    path('api/', include(router.urls)),
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
]
```

Le cadre Django REST a une interface utilisateur intégrée pour les API, lorsque le serveur fonctionne en mode de développement avec DEBUG=True dans settings.py. Et nous pouvons ouvrir cette URL

Tout fonctionne, même sans rien définir, juste en connectant ce cadre à l'url. Il supporte l'autorisation, pour les requêtes nécessitant une autorisation.

```
$ curl http://127.0.0.1:8000/api/
{}
```

Définissez une API pour le projet, nous utiliserons au minimum quelques fonctionnalités du cadre REST.

- Les sérialiseurs permettent de convertir des données complexes, telles que des querysets et des instances de modèles, en types de données Python natifs qui peuvent ensuite être facilement convertis en JSON, XML ou d'autres types de contenu. Les sérialiseurs fournissent également une désérialisation, permettant aux données analysées d'être reconverties en types complexes, après avoir validé les données entrantes.
- ViewSet - permet de combiner la logique d'un ensemble de visualisations connexes dans une seule classe

Ajoutons un point de terminaison pour nos messages de notification. Très simple, pourtant. Regardez le contenu mis à jour de **urls.py _**

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers, serializers, viewsets
from .models import CommunityPost

router = routers.DefaultRouter()

class CommunityPostSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        # classe avec modèle
        model = CommunityPost
        # liste des champs à afficher, ou uniquement '__all__'
        fields = '__all__'
# les ViewSets définissent le comportement de la visualisation.
class CommunityPostViewSet(viewsets.ModelViewSet):
    queryset = CommunityPost.objects.all()
    serializer_class = CommunityPostSerializer

# connexion à l'API
router.register(r'posts', CommunityPostViewSet)

urlpatterns = [
    path('api/', include(router.urls)),
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
]
```

Et il apparaît maintenant dans l'interface web

Il suffit de cliquer sur le lien ici, et vous obtiendrez la réponse à cette question

Faites défiler jusqu'à la fin, et vous trouverez un formulaire généré pour les nouveaux éléments, qui peuvent être ajoutés par requête POST. Tous les champs sont adaptés au type de propriétés

Dans la liste des éléments, vous pouvez cliquer sur l'URL de n'importe quel élément, et voir ceci. Seulement cet élément comme réponse, et un formulaire d'édition avec une requête PUT

Authentication

Et vous pouvez déjà changer les données, par PUT ou POST. Le besoin d'autorisation n'est pas encore activé. Le cadre REST offre diverses combinaisons d'authentifications à utiliser. Ainsi, vous pouvez ouvrir certaines ressources pour un accès anonyme en lecture seule. Et s'authentifier pour pouvoir effectuer des modifications. Ou fermer complètement tout accès. Il suffit de configurer l'accès en lecture seule pour les anonymes, et de demander une authentification pour toute modification. Pour ce faire, il suffit d'ajouter ces lignes à ***settings.py***

```
REST_FRAMEWORK = {
    # Utilisez les permissions standard de Django `django.contrib.auth`,
    # ou autorisez l'accès en lecture seule pour les utilisateurs non authentifiés.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly',
    ],
}
```

Avec ceci, vous ne verrez plus le formulaire jusqu'à ce que vous vous connectiez, avec le nom d'utilisateur et le mot de passe d'instance créés précédemment pour l'administration de Django.

Pagination

Par défaut, il n'y a pas de pagination, mais nous pouvons facilement l'ajouter à toute requête de liste. Mettez à jour la variable **REST_FRAMEWORK** dans ***settings.py***. Configurer la classe de pagination et la taille de page par défaut

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

En conséquence, cela a légèrement modifié le JSON résultant, en ajoutant des éléments pertinents, tels que les liens vers les pages suivantes et précédentes, ainsi que la quantité de tous les éléments. Et avec l'interface Web, vous pouvez parcourir les pages.

Filtrage et recherche

Il est également assez simple d'ajouter des capacités de filtrage et de recherche. Mettez à jour la variable **REST_FRAMEWORK** dans ***settings.py***.

```
REST_FRAMEWORK = {
...
    'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.SearchFilter',
    ],
...
]
```

Et mettez à jour CommunityPostViewSet avec une liste de champs pour le filtrage et la recherche

```
class CommunityPostViewSet(viewsets.ModelViewSet):
    queryset = CommunityPost.objects.all()
    serializer_class = CommunityPostSerializer
    filterset_fields = ['posttype', 'lang', 'published']
    search_fields = ['name',]
```

Et cela fonctionne directement depuis l'interface Web

Et finalement, nous disposons d'une API REST entièrement fonctionnelle, juste pour cette seule ressource, pour le moment. C'est assez simple pour le moment, mais il est possible de faire beaucoup de personnalisation, de connecter d'autres ressources et de les lier.

#Embedded Python #Python #InterSystems IRIS
Voir l'application sur InterSystems Open Exchange