9. L'ORM Django et les QuerySets

Dans ce chapitre, nous allons apprendre comment Django se connecte à la base de données et comment il y enregistre des choses. On respire un grand coup et on y va !

Qu'est-ce qu'un QuerySet ?

Un QuerySet est, par essence, une liste d'objets d'un modèle donné. C'est ce qui vous permet de lire, trier et organiser des données présentes dans une base de données.

Il est plus simple d'apprendre avec un exemple. Et si nous nous intéressions à celui-ci ?

Le shell Django

Ouvrez la console de votre ordinateur (et non celle de PythonAnywhere) et tapez la commande suivante : command-line

```
~/djangology$ python manage.py shell
```

Ceci devrait maintenant s'afficher dans votre console : command-line

```
(InteractiveConsole)
>>>
```

Vous êtes maintenant dans la console interactive de Django. C'est comme celle de Python, mais avec toute la magie qu'apporte Django :). Les commandes Python sont aussi utilisables dans cette console.

Lister tous les objets

Essayons tout d'abord d'afficher tous nos billets. Vous pouvez le faire à l'aide de cette commande : command-line

```
>>> Billet.objects.all()
Traceback (most recent call last):
     File "<console>", line 1, in <module>
NameError: name 'Billet' is not defined
```

Oups ! Voilà que ça nous renvoie une erreur qui nous dit qu'il n'existe pas de Billet. En effet, nous avons oublié de commencer par un "import" !

command-line

```
>>> from blog.models import Billet
```

Nous importons le modèle Billet depuis notre blog.models. Essayons à nouveau la commande précédente :

command-line

```
>>> Billet.objects.all()
<QuerySet [<Billet: my post title>, <Billet: another post title>]>
```

Cela nous permet d'obtenir une liste des billets que nous avons créés tout à l'heure ! Rappelez-vous : nous avions créé ces billets à l'aide de l'interface d'administration de Django. Cependant, nous aimerions maintenant créer de nouveaux billets à l'aide de python : comment allons-nous nous y prendre ?

Créer des objets

Voici comment créer un nouveau objet Billet dans la base de données :

command-line

```
>>> Billet.objects.create(author=me, title='Sample title', text='Test')
```

Cependant, il nous manque un petit quelque chose : moi. Nous avons besoin de lui passer une instance du modèle User en guise d'auteur (author). Comment faire ?

Tout d'abord, il nous faut importer le modèle User :

command-line

```
>>> from django.contrib.auth.models import User
```

Avons-nous des utilisateurs dans notre base de données ? Voyons voir :

command-line

```
>>> User.objects.all()
<QuerySet [<User: ola>]>
```

Il s'agit du superutilisateur que nous avons créé tout à l'heure ! Sauvegardons une instance de cet utilisateur (modifie la ligne suivante avec ton nom d'utilisateur) :

command-line

```
>>> me = User.objects.get(username='ola')
```

Comme vous pouvez le constater, nous avons maintenant obtenu (get) un utilisateur (User) avec un nom d'utilisateur username qui est égal à "ola". Très bien !

Nous allons enfin pouvoir créer notre Billet :

command-line

```
>>> Billet.objects.create(author=me, title='Sample title', text='Test')
<Billet: Sample title>
```

Et voilà ! Vous aimeriez voir si ça a vraiment marché ?

command-line

```
  >>> Billet.objects.all()
```

<QuerySet [<Billet: my post title>, <Billet: another post title>, <Billet: Sample title>]> Et voilà : un Billet de plus dans la liste !

Ajouter plus de billets

Amusez-vous à ajouter d'autres billets pour vous entraîner un peu. Essayez d'ajouter deux ou trois billets en plus puis passez à la partie suivante.

Filtrer les objets

L'intérêt des QuerySets, c'est que l'on peut les filtrer. Disons que nous aimerions retrouver tous les billets écrits par l'utilisateur Ola. Pour cela, nous allons utiliser filter à la place de all dans Billet.objects.all(). Les parenthèses vont nous servir à préciser quelles sont les conditions auxquelles un billet de blog doit se conformer pour être retenu par notre queryset. Dans notre exemple, la condition est que author soit égal à me. La manière de le dire en Django c'est : author=me. Maintenant, votre bout de code doit ressembler à ceci:

command-line

```
  >>> Billets.objects.filter(author=me)
  <QuerySet [<Billet: Sample title>, <Billet: Post number 2>, <Billet: My
3rd post!>, <Billet: 4th title of post>]>
```

Et si nous voulions chercher les billets qui contiennent uniquement le mot "titre" ("title" en anglais) dans le champ title?

command-line

```
  >>> Billet.objects.filter(title__contains='title')
  <QuerySet [<Billet: Sample title>, <Billet: 4th title of post>]>
```

Il y a deux tirets bas (_) entre title et contains. L'ORM de Django utilise cette règle afin de séparer les noms de champ ("title") et les opérations ou les filtres ("contains"). Si vous n'utilisez qu'un seul tiret bas, vous allez obtenir une erreur du type : "FieldError: Cannot resolve keyword title_contains".

Vous pouvez aussi obtenir une liste de tous les billets publiés. Pour cela, nous allons filtrer les billets qui possèdent une date de publication (published_date) dans le passé :

command-line

```
  >>> from django.utils import timezone
  >>> Billet.objects.filter(published_date__lte=timezone.now())
```

```
  <QuerySet []>
```

Malheureusement, le billet que nous avons créé dans la console Python n'est pas encore publié. Allons corriger ce problème ! Dans un premier temps, nous aimerions obtenir une instance du billet que nous voulons publier :

command-line

```
  >>> billet = Billet.objects.get(title="Sample title")
```

Ensuite, publions-le grâce à notre méthode publish :

command-line

```
  >>> billet.publish()
```

Maintenant, essayez d'obtenir à nouveau la liste des billets publiés (appuyez trois fois sur la flèche du haut, puis entrée) :

command-line

```
  >>> Billet.objects.filter(published_date__lte=timezone.now())
  <QuerySet [<Billet: Sample title>]>
```

Classer les objets

Les QuerySets permettent aussi de trier la liste des objets. Essayons de les trier par le champ created_date :

command-line

```
  >>> Billet.objects.order_by('created_date')
  <QuerySet [<Billet: Sample title>, <Billet: Post number 2>, <Billet: My
3rd post!>, <Billet: 4th title of post>]>
```

On peut aussi inverser l'ordre de tri en ajoutant - au début :

command-line

```
  >>> Billet.objects.order_by('-created_date')
  <QuerySet [<Billet: 4th title of post>,  <Billet: My 3rd post!>, <Billet:
Post number 2>, <Billet: Sample title>]
```

Requêtes complexes grâce au chaînage des méthodes

Comme vous l'avez vu, quand on applique certaines méthodes à Billet.objects on obtient un QuerySet en résultat. Les mêmes méthodes peuvent également être appliquées sur un QuerySet, ce qui ensuite donnera lieu à un nouveau QuerySet. Ainsi, vous pouvez combiner leur effet en les enchaînant l'une après l'autre :

```
  >>>
Billet.objects.filter(published_date__lte=timezone.now()).order_by('publishe
d_date')
  <QuerySet [<Billet: Post number 2>, <Billet: My 3rd post!>, <Billet: 4th
title of post>, <Billet: Sample title>]
```

C'est un outil très puissant qui va vous permettre d'écrire des requêtes complexes.

Génial ! Vous êtes maintenant prête à passer à l'étape suivante ! Pour fermer le shell, tapez ceci :

command-line

```
  >>> exit()
  $
```

11. Données dynamiques

From:
https://wiki.centrale-med.fr/informatique/ - **WiKi informatique**

Permanent link:
https://wiki.centrale-med.fr/informatique/public:appro-s7:td_web:orm?rev=1697631547

Last update: **2023/10/18 14:19**