Débutez avec le framework Django

12 heures Moyenne

Mis à jour le 28/06/2023

Sauvegardez des données dans une base de données avec un modèle et une migration

01:00

Comprenez la représentation d'une entité avec un modèle

Quel que soit le but de votre application web, il est très probable qu'elle doive stocker des données.

Les programmeurs commencent souvent par identifier les différentes entités au sujet desquelles nous devons stocker des données. Dans notre application, nous voulons suivre les différents groupes et les

annonces d'articles à vendre. Il y a donc deux entités ici : les groupes et les annonces.

Pour chaque entité pour laquelle nous voulons stocker des données, nous créons un modèle pour représenter cette entité. Commençons par un modèle de groupe !

Un **modèle** définit les caractéristiques que nous voulons stocker à propos d'une entité particulière.

Notre modèle de groupe pourrait donc avoir comme caractéristiques un titre, un genre et l'année où elle est devenue active. Ces caractéristiques sont également connues sous le nom de **champs**.

Nous pouvons ensuite utiliser le modèle pour créer des objets individuels, ou instances, de ce modèle, qui ont chacun leurs propres caractéristiques uniques.

> Attendez une minute... n'est-ce pas ce qu'on appelle une classe ?

On pourrait penser que le terme « modèle » n'est qu'un autre nom pour une « classe » ! Et vous auriez à moitié raison : vous créez un modèle dans Django en définissant une classe Python.

> Alors qu'est-ce qui différencie un modèle d'une classe ordinaire ?

En général, dans les frameworks MVC et MVT, un modèle est également capable de stocker (ou de "**persister**") ses données dans une base de données pour une utilisation ultérieure. Cela contraste avec les classes et objets ordinaires, dont les données existent ***temporairement*** : par exemple seulement pendant l'exécution de l'application.

De même, les « caractéristiques » des classes Python sont appelées ***attributs***, mais lorsqu'un modèle enregistre un attribut dans la base de données, il s'agit d'un champ.

> OK, donc pour créer un modèle, est-ce que je dois savoir comment écrire du code qui interagit avec une base de données ?

En fait, non. C'est l'un des avantages de l'utilisation d'un framework comme Django : toutes les fonctionnalités de persistance des données dans une base de données ont déjà été écrites pour vous. Tout ce que vous avez à faire est de faire en sorte que votre modèle hérite de la classe `models.Model` de Django. Votre modèle hérite ensuite de toutes les méthodes (comportements) nécessaires pour effectuer des opérations telles que la sélection et l'insertion de données dans une base de données.

Vous n'aurez donc pas à écrire de code qui interagit avec une base de données, mais vous ***devrez*** apprendre à interagir avec les modèles de Django pour faire la même chose (ne paniquez pas, c'est plus facile ! 😜).

Créons maintenant notre premier modèle. Nous allons faire simple pour commencer : ce modèle va représenter un groupe, mais nous allons seulement définir une caractéristique pour le moment : le nom

du groupe.

Ajoutez un modèle à models.py

Ouvrez le fichier models.py qui ressemble à ceci :

python

```python
# listings/models.py

from django.db import models

# Create your models here.
```

Une fois encore, Django nous a laissé un commentaire utile (Create your models here = Créez vos modèles ici). Maintenant que nous savons où vont nos modèles, remplaçons ce commentaire par le code suivant :

python

```python
# listings/models.py

class Band(models.Model):
    name = models.fields.CharField(max_length=100)
```

Nous avons défini notre classe, l'avons nommée `Band` et l'avons fait hériter de models.Model, qui est la classe de base du modèle de Django.

Ensuite, nous ajoutons un attribut de classe à notre classe `name` . À cet attribut, nous attribuons un `CharField` , qui est l'abréviation de Character Field. Il s'agira d'un champ qui stocke des données de type caractère/texte/chaîne, ce qui est le type de données approprié pour un nom.

Nous avons également fixé la longueur maximale du nom d'un `Band` à 100.

Et voilà notre premier modèle.

> Cela peut sembler différent des classes que vous avez construites auparavant.
>
> Les classes Python ont généralement un **constructeur** : la méthode `__init__` , où nous utilisons les arguments passés pour définir les valeurs des attributs d'instance.
>
> Avec les modèles Django, les choses se font différemment. Le framework examine les champs du modèle (que nous définissons comme des attributs de *classe*), puis crée le constructeur pour nous. C'est un autre exemple de la « magie » du framework.

Comme pour les classes Python, nous pouvons utiliser les modèles Django pour créer des instances de cette classe, par exemple `band = Band()` .

> On peut donc commencer à créer des objets de groupe maintenant ?

Pas tout de suite.

Comme nous l'avons dit, une des caractéristiques d'un modèle est qu'il est capable de stocker ses données dans une base de données. Nous avons créé notre base de données dans le chapitre sur la configuration. Mais cette base de données ne sait encore rien de notre modèle de bande. Et c'est là que les migrations entrent en jeu.

Comprenez la gestion de l'état de la base de données avec une migration

Si nous voulons stocker les groupes dans notre base de données, nous aurons besoin d'une nouvelle table, contenant une colonne pour chaque champ que nous avons ajouté à notre modèle de groupe, ainsi qu'une colonne id pour servir de **clé primaire** : un identifiant unique pour chaque ligne de la table.

La structure d'une base de données, en termes de tables et de colonnes, est appelée **schéma**.

Si nous construisions notre schéma de base de données manuellement, nous pourrions écrire une requête SQL ou utiliser une interface graphique de gestion de base de données, pour créer notre première table.

Mais dans Django, nous faisons les choses différemment. Nous utilisons une sous-commande de l'utilitaire de ligne de commande qui va générer des instructions pour construire la table. Et ensuite, nous utilisons une autre sous-commande pour exécuter ces instructions. Ces instructions sont appelées une *migration*.

Une **migration** est un ensemble d'instructions permettant de passer le schéma de votre base de données d'un état à un autre. Il est important de noter que ces instructions peuvent être exécutées automatiquement, comme un code.

> Mais pourquoi ne pas créer la table manuellement ?

En programmation, on parle souvent de « configuration par le code ». Il s'agit d'une philosophie qui stipule que toutes les étapes nécessaires à la construction de votre application ne doivent pas être effectuées à la main, mais plutôt inscrites dans le code. Pourquoi ? Pour plusieurs raisons :

- Les étapes manuelles peuvent facilement être oubliées, mais les étapes écrites sous forme de code peuvent être stockées dans un repository afin qu'elles soient aussi sûres que tous vos autres codes sources.
- Lorsque les étapes sont conservées dans votre repository, elles peuvent être facilement partagées avec les autres membres de l'équipe.
- Les étapes écrites sous forme de code peuvent être exécutées automatiquement par votre ordinateur. Cette méthode est rapide et fiable, surtout s'il y a plusieurs étapes.

Maintenant que nous savons pourquoi les migrations sont importantes, créons-en une pour notre modèle de groupe.

text

```
# shell

(env) ~/projects/django-web-app/merchex
→ python manage.py makemigrations
python manage.py makemigrations
Migrations for 'listings':
  listings/migrations/0001_initial.py
    - Create model Band
```

Le résultat de la CLI nous indique qu'une nouvelle migration a été enregistrée dans « listings/migrations/0001_initial.py », et que son objectif est de « Créer le modèle de groupe », ce qui signifie en fait que cette migration va créer une table dans la base de données pour notre modèle de groupe.

> Comment a-t-il su faire ça sans que je doive préciser quoi que ce soit ?

L'avantage de cette commande est qu'elle analyse notre fichier models.py pour y déceler toute modification et déterminer le type de migration à générer.

Maintenant que nous avons notre migration (nos instructions), nous devons exécuter ces instructions sur la base de données.

Appelez « migrate » avec la CLI

text

```
(env) ~/projects/django-web-app/merchex
→ python manage.py migrate
Operations to perform:
Apply all migrations: admin, auth, contenttypes, listings, sessions
Running migrations:
Applying listings.0001_initial... OK
```

Vous vous rappelez dans le chapitre sur l'installation quand nous avons ajouté « listings » au `INSTALLED_APPS` de notre projet ? Django a recherché dans chacune de ces applications installées de nouvelles migrations à exécuter, il a trouvé notre nouvelle migration et l'a « appliquée » : il a exécuté ces instructions sur la base de données.

Nous sommes enfin prêts à utiliser notre modèle pour créer des objets de groupe !

Enregistrez des objets dans la base de données dans le shell de Django

Dans cette section, nous allons écrire du code dans le shell de Django.

Le **shell de Django** est simplement un shell Python ordinaire qui exécute votre application Django. Vous pouvez le considérer comme un endroit où vous pouvez essayer du code en temps réel : chaque fois que vous appuyez sur Entrée, la ligne de code que vous venez de taper est exécutée. Ainsi, alors que le

code que vous tapez dans un module/fichier Python peut être exécuté de nombreuses fois, le code que vous tapez dans le shell Django n'est exécuté qu'une seule fois, puis oublié.

> Apprenez-en davantage sur le **shell Python**.

Utilisons le shell pour créer quelques objets de groupe, puis enregistrons ces objets dans la base de données.

Vous pouvez utiliser mes exemples, ou n'hésitez pas à ajouter vos propres groupes, musiciens ou compositeurs préférés !

Ouvrez le shell en utilisant l'utilitaire de ligne de commande :

```text
(env) ~/projects/django-web-app/merchex
→ python manage.py shell
>>>
```

À l'invite du shell (`>>>`), tapez le code suivant pour importer notre modèle de groupe :

```text
>>> from listings.models import Band
```

Appuyez sur `Entrée` pour exécuter cette ligne.

Ensuite, nous allons créer une nouvelle instance du modèle `Band` :

```text
>>> band = Band()
>>> band.name = 'De La Soul'
```

Jetez un coup d'oeil à l'état actuel de l'objet en tapant simplement `band` :

```text
>>> band
<Band: Band object (None)>
```

Le shell nous dit que nous avons un objet band, mais l'id est `None` , il n'a pas encore d'id.

Maintenant, sauvegardons cet objet dans la base de données :

```text
>>> band.save()
```

... et ensuite regardez à nouveau l'état de l'objet :

```text
>>> band
<Band: Band object (1)>
```

... maintenant l'id est `1` .

> Chaque fois que nous insérons un objet dans la base de données, un identifiant est ajouté automatiquement pour nous.

Recommençons, en donnant cette fois un nom différent à l'objet Bande. Vous pouvez réutiliser la variable `band` et lui attribuer un nouveau `Band` :

text

```
>>> band = Band()
>>> band.name = 'Cut Copy'
>>> band.save()
>>> band
<Band: Band object (2)>
```

Essayons maintenant une autre méthode. Voici une alternative en une seule ligne qui fait la même chose :

text

```
>>> band = Band.objects.create(name='Foo Fighters')
```

> Nous en apprendrons plus sur les `Band.objects` **dans la dernière partie** du cours.

Nous pouvons regarder l'objet et la valeur de son champ `name` :

text

```
>>> band
<Band: Band object (3)>
>>> band.name
'Foo Fighters'
```

Notre base de données contient maintenant 3 objets de groupe. On peut vérifier ça comme ça :

text

```
>>> Band.objects.count()
3
>>> Band.objects.all()
<QuerySet [<Band: Band object (1)>, <Band: Band object (2)>, <Band: Band object (3)>]>
```

> Pour l'instant, vous pouvez considérer qu'un `QuerySet` ressemble beaucoup à une `list` Python.

Appuyez sur `Ctrl` + `D` pour quitter le shell.

Nous avons maintenant créé un modèle, exécuté les migrations pour qu'il existe dans la base de données, puis créé et enregistré une instance de ce modèle dans la base de données. Consultez le

screencast pour vous rafraîchir la mémoire sur toutes ces étapes.

02:19

Revenons à notre vue afin d'afficher les noms de nos groupes sur l'une de nos pages.

Mettez à jour la vue pour afficher les objets de groupe

> Alors, comment faire pour que nos objets sortent de la base de données et se retrouvent dans nos pages ?

Ouvrez listings/views.py et trouvez notre fonction de vue `hello` , telle que nous l'avons laissée au dernier chapitre :

python

```python
# listings/views.py

def hello(request):
return HttpResponse('<h1>Hello Django!</h1>')
```

Dans la section précédente, nous avons vu comment obtenir tous les objets de groupe de la base de données dans le shell : `Band.objects.all()` . Faisons la même chose dans notre vue et stockons les objets dans une variable :

python

```python
# listings/views.py

...
from listings.models import Band
...

def hello(request):
```

```
    bands = Band.objects.all()
    return HttpResponse('<h1>Hello Django!</h1>')
```

N'oubliez pas d'importer votre modèle `Band` en haut !

La variable `bands` contient maintenant une liste de tous les groupes qui peuvent être trouvés dans la base de données. Cela signifie qu'il est maintenant possible d'accéder à chacun des objets `Band` individuels en utilisant la notation d'index de Python, comme ceci :

python
```python
# exemple de code

bands[0] # pour le premier objet `Band`...
bands[1] # pour le prochain..
bands[2] # et le suivant...
```

Ensuite, pour accéder au champ « name » de l'un de ces objets `Band` , nous ferons appel à l'attribut `name` de l'objet en utilisant la notation par points de Python, comme nous le ferions avec n'importe quel autre objet :

python
```python
# exemple de code

bands[0].name # renvoie « De La Soul »
```

Utilisons ces techniques pour afficher les noms de nos groupes dans notre page :

python
```python
# ~/projects/django-web-app/merchex/listings/views.py

...
from bands.models import Band
...

def hello(request):
    bands = Band.objects.all()
    return HttpResponse(f"""
        <h1>Hello Django !</h1>
        <p>Mes groupes préférés sont :<p>
        <ul>
            <li>{bands[0].name}</li>
            <li>{bands[1].name}</li>
            <li>{bands[2].name}</li>
        </ul>
""")
```

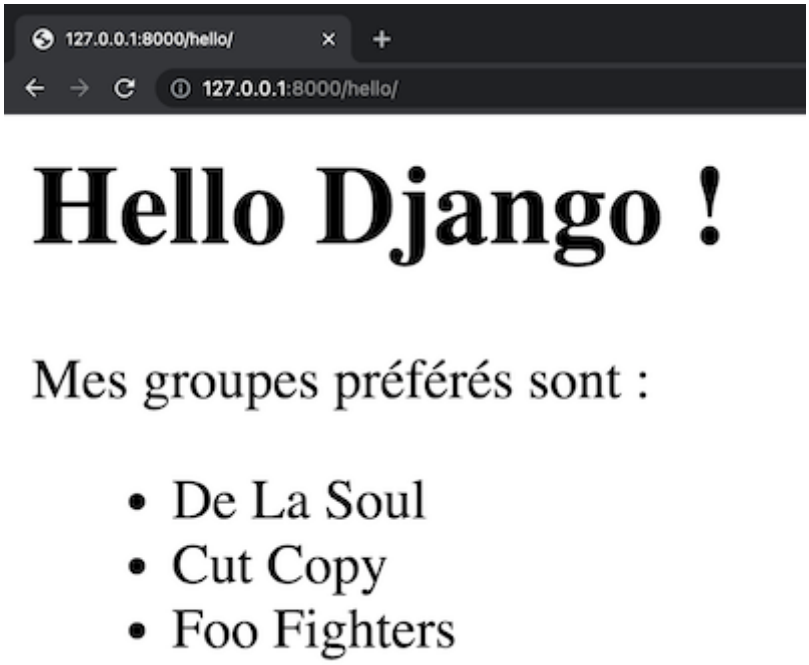
Pour aider à la lisibilité, nous avons :

- utilisé des guillemets triples (`"""`) pour répartir notre chaîne HTML sur plusieurs lignes ;

- fait de cette chaîne une « f-string » (`f"""`) afin que nous puissions injecter nos noms de groupes dans la chaîne en utilisant `{ ... }` comme placeholders.

  Si vous avez inséré moins, ou plus, de 3 groupes dans votre base de données, n'hésitez pas à mettre à jour votre code en conséquence !

Regardons à nouveau cette page dans le navigateur : http://127.0.0.1:8000/hello/



La page « hello ».

Notre vue est maintenant capable d'extraire des données de cette base de données, et de les afficher dans une de nos pages. Revoyons ces étapes dans le screencast.

01:19

Visualisons ce que nous avons fait dans ce chapitre :

01:50

Vous êtes maintenant prêt à créer vos propres modèles et migrations !

C'est à vous ! Utilisez les modèles et les migrations pour sauvegarder les données dans une base de données

C'est maintenant à votre tour d'ajouter un nouveau modèle à votre application. Ce modèle va suivre les objets `Listing`. Il ne devrait y avoir qu'un seul champ appelé `title` : le titre de l'annonce. Le `title` doit avoir une longueur maximale de 100 caractères.

Votre modèle devra être accompagné d'une migration, qui devra être exécutée sur la base de données. Vous pouvez ensuite utiliser le shell de Django pour insérer au moins 3 objets dans la base de données.

Quelques exemples de titres pour ces annonces :

- « Affiche ORIGINALE de la tournée de De La Soul - Fillmore Auditorium San Francisco novembre 2001 »
- « T-shirt du concert de Cut Copy, tournée Free Your Mind, 2013 »
- « Foo Fighters - l'affiche promo du single Big Me, fin des années 90 »
- « Beethoven - Sonate au clair de lune - manuscrit original EXTRÊMEMENT RARE »

> Si vos titres comportent des apostrophes (comme le troisième élément ci-dessus), n'oubliez pas **d'échapper les guillemets ou d'utiliser des guillemets doubles pour vos chaînes de caractères**, selon le cas.

Enfin, vous devez mettre à jour la vue `listings` que vous avez créée dans le dernier chapitre, afin qu'elle récupère les objets `Listing` de votre base de données et affiche leurs titres dans le HTML de votre page.

En résumé

- Un modèle définit les caractéristiques et les comportements d'un objet dont vous voulez garder la trace dans votre application. Il ressemble beaucoup à une classe standard, mais en plus, un modèle sait comment enregistrer (« persister ») ses données dans une base de données.
- Une migration est un ensemble d'instructions qui font passer notre base de données d'un état à un autre, par exemple en créant une nouvelle table. Nous pouvons utiliser le CLI de Django pour générer et exécuter les migrations à notre place.
- Nous pouvons utiliser le shell de Django pour insérer de nouveaux objets dans notre base de données.

- Dans une vue, nous pouvons récupérer des objets dans la base de données et afficher leurs données dans nos pages.

Grâce aux modèles, le « M » de l'architecture MVT, nous avons enregistré des données dans notre base de données et nous les affichons dans nos pages. Examinons maintenant le dernier élément de MVT : le gabarit.

Indiquer que ce chapitre n'est pas terminé

Servez du contenu à l'aide d'une vue

Séparez la logique de l'application de la présentation avec un gabarit Django

Les professeurs

Patrick Wampé

Développeur full stack et Data Scientist. Formateur dans plusieurs écoles d'informatique, il a également écrit un livre sur l'IA.

Patrick Heneghan

Software engineer in the UK, coding mostly in Python on backend systems.

Rafiq Hilali

British Software Engineer and Django expert with Lambert Labs. Currently based in BC, Canada.

OPENCLASSROOMS

OPPORTUNITÉS

AIDE

POUR LES ENTREPRISES

EN PLUS

Français

Télécharger dans l'App Store