esprit
Se former autrement
HONORIS UNITED UNIVERSITIES

Django Models

UP Web

AU: 2024/2025

CONFÉRENCE DES GRANDES ÉCOLES

cdio
CONCEIVE DESIGN IMPLEMENT OPERATE

AUF Agence Universitaire de la Francophonie
IFGU

UNESCO Chair Project-based learning ESPRIT School of engineering Tunisia

Plan

1. La couche modèle
2. Types des champs
3. Options des champs
4. Associations
5. Gestion des Migrations
6. Querysets
7. Lookups
8. Requêtes complexes avec la classe Q
9. Validateurs

La couche modèle

La couche modèle dans Django est responsable de la définition de la structure des données. Les modèles représentent les tables de la base de données et définissent comment les données sont stockées et interagissent.

- La couche « modèle » concrétisée dans le fichier **models.py** représente l'unique façon de déclaration des données.
- Chaque « modèle » correspond à une seule table dans la base de données.
- **Exemple:**

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

La couche modèle

- La définition du modèle ci-dessus est équivalente à la déclaration de la table suivante:

```
Create Table myapp_person (
        "id" serial NOT NULL PRIMARY KEY,
        "first_name" varchar(30) NOT NULL,
        "last_name" varchar(30) NOT NULL
);
```

Types des champs

- Selon le type de la donnée à représenter, on déclare le type du champ adéquat.
- **Exemple:**
  - **IntegerField:** champ pour une valeur entière.
  - FloatField :Pour les nombres à virgule flottante.
  - CharField:Pour les chaînes de caractères de longueur fixe.
  - TextField:Pour les grandes chaînes de texte sans limite de longueur.
  - **DateField:** champ pour une date.
  - TimeField :Pour stocker des heures.
  - **EmailField:** champ d'Email.
  - BooleanField:Pour les valeurs booléennes (True/False
  - ChoiceField:Permet de sélectionner une option parmi une liste prédéfinie
  - **FileField:** Importation de fichiers.
  - ImageField :Pour télécharger des images.
  - Etc…

Model field reference | Django documentation | Django (djangoproject.com)

Options des champs

- Il est possible de définir certains arguments aux champs.
- Par exemple: l'argument **max_length** pour le champs **CharField** spécifie la taille de la colonne de type « Varchar » dans la base de données.
- **Exemple d'options:**
  - Null, Blank, choices, primary_key, unique, etc.

- Pour définir les labels « Prénom » et « nom » aux champs

```
first_name = models.CharField('Prénom', max_length=30)
last_name = models.CharField('Nom',max_length=30)
```

Models | Documentation de Django | Django (djangoproject.com)

Associations

- On distingue dans la déclaration du « data model » des attributs spécifiques pour les associations entre les classes.

- On distingue 3 types d'associations:

  - « One-to-One »
  - « Many-to-One »
  - « Many-to-Many »

Association « ONE-TO-ONE »

- L'association est déclarée via l'attribut **OneToOneField**.

```
class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)


class Restaurant(models.Model):
    place = models.OneToOneField(
        Place,
        on_delete = models.CASCADE,
        primary_key = True,
)
```

Model field reference | Django documentation | Django (djangoproject.com)

Association « MANY-TO-ONE »

- On peut définir une association de type « Many to One » via le mot clé **ForeignKey**.

```
class Reporter(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    email = models.EmailField()


class Article(models.Model):
    headline = models.CharField(max_length=100)
    pub_date = models.DateField()
    reporter = models.ForeignKey(
        Reporter,
        on_delete = models.CASCADE,
)
```

Model field reference | Django documentation | Django (djangoproject.com)

Association « MANY-TO-MANY »

- Pour définir une association de type « Many to Many », on utilise le champ **ManyToManyField**.
- Dans l'exemple ci-dessous, un **Article** peut être publié dans plusieurs objets de type **Publication** et une **Publication** possède plusieurs objets de type **Article**.

```
class Publication(models.Model):
    title = models.CharField(max_length=30)

    class Meta:
        ordering = ('title',)


class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication)
    class Meta:
        ordering = ('headline',)
```

Model field reference | Django documentation | Django (djangoproject.com)

Association « MANY-TO-MANY » avec classe d'association

- Pour créer une classe d'association pour les modèles Publication et Article , vous devez définir un modèle intermédiaire pour stocker des informations supplémentaires sur la relation. Voici comment procéder :
- Nous allons créer une classe **ArticlePublication** pour représenter la relation entre l'article et la publication, ce qui nous permettra d'ajouter des champs supplémentaires si nécessaire.

Model field reference | Django documentation | Django (djangoproject.com)

Association « MANY-TO-MANY » avec classe d'association

```python
class Publication(models.Model):
    title = models.CharField(max_length=30)

    class Meta:
        ordering = ('title',)

    def __str__(self):
        return self.title

class Article(models.Model):
    headline = models.CharField(max_length=100)
    publications = models.ManyToManyField(Publication, through='ArticlePublication')

    class Meta:
        ordering = ('headline',)

    def __str__(self):
        return self.headline

class ArticlePublication(models.Model):
    article = models.ForeignKey(Article, on_delete=models.CASCADE)
    publication = models.ForeignKey(Publication, on_delete=models.CASCADE)
    date_added = models.DateField(auto_now_add=True)  # Additional field for the association

    class Meta:
        unique_together = (('article', 'publication'),)  # Prevent duplicate associations

    def __str__(self):
        return f"{self.article.headline} - {self.publication.title}"
```

Gestion des Migrations

- Une fois les classes et leurs associations sont définies dans models.py, on peut générer la base de données. Pour ce faire:
  - Déclarer les ajouts réalisés sur le modèle de données

python manage.py makemigrations <Nom de l'application>

  - Générer le schéma de la base de données

python manage.py migrate

- Vérifier l'État des Migrations

```
python manage.py showmigrations
```

- - Revenir sur des Migrations

```
python manage.py migrate <Nom_de_l_application> <numéro_de_version>
```

Gestion des Migrations

- Si vous avez plusieurs applications dans votre projet Django et que vous souhaitez effectuer les migrations pour toutes les applications en une seule fois, vous pouvez simplement exécuter la commande **`makemigrations`** sans spécifier de nom d'application.

Querysets

- Il est possible d'interagir avec les données via une API fournie par Django

  python manage.py shell

- Par exemple, pour ajouter un nouvel Reporter:

```
>>> from projectsApp.models import Reporter
>>> r = Reporter(first_name='Jane',last_name='Doe',email='jd@esprit.tn')
>>> r.save()
>>>Reporter.objects.create(first_name='Emile',last_name='Jhon',email='ej@esprit.tn')
```

- Pour récupérer la liste des étudiants:

  Reporter.objects.all()

- Accéder à un attribut

  r.first_name

Querysets

- Il existe des méthodes de manipulation des données mais ne retournant pas des « QuerySet ».
- **Exemple:**
  - **get**: Retourner L'objet correspondant
  - **get_or_create**: En cas d'inexistence, créer l 'élément
  - **bulk_create**: Insertion d'une liste de valeurs
  - **count**: Retourne le nombre d'objets correspondant au QuerySet
  - …

QuerySet API reference | Django documentation | Django (djangoproject.com)

Querysets

- **Filter** est une méthode permettant de retourner un sous ensemble de données ou « QuerySet » correspondant à un critère fourni en paramètre.
- **Filter** n'est pas la méthode unique de sélection, il existe d'autres et qui retournent des « QuerySet ».
- **Exemple**
  - **exclude**: Exclure les éléments répondants au critère
  - **annotate**: Associer une annotation à chaque élément renvoyé
  - **order_by**: Trier les éléments renvoyés
  - **reverse**: Inverser l'ordre du jeu de données
  - **distinct**: Suppression des duplications

QuerySet API reference | Django documentation | Django (djangoproject.com)

Querysets

- **Soit l'exemple de models.py suivant:**

```
class Publisher(models.Model):
    name = models.CharField(max_length=30)
    address = models.CharField(max_length=50)
    website = models.URLField()

class Author(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=40)

class Book(models.Model):
    title=models.CharField(max_length=100)
    authors = models.ManyToManyField(Author)
    publisher = models.ForeignKey(Publisher)
```

Querysets

- **Publisher.objects.get**(name='Oxford University Press')

#get est utilisée pour récupérer un seul élément (lève une exception en cas de duplication ➔ penser à utiliser try/except)

- **Publisher.objects.order_by** ("name", "address")

Trier les données retournés par nom et adresse

utiliser « - » pour inverser l'ordre de tri

- **Publisher.objects.filter**(id=5).**update**(name="New Publisher")

Mise à jour de l'élément d'Id 5

Remarque: Il est possible de mettre à jour des éléments multiple avec update

Querysets

- **Publisher.objects.filter**(address='Paris').**delete**()

  Remarque: Utiliser **all()** en cas de besoin de suppression de tous les éléments.
- **Publisher.objects.filter**(name='Oxford University Press')

- **Publisher.objects.filter**(name__contains="press")

#__contains est dite un « lookup »

SELECT * FROM Publisher WHERE name like " %press% "

Lookups

- Django fournit des méthodes permettant de chercher dans les champs (L'équivalent de WHERE en SQL).
- Exemples de méthodes:
  - **__contains** : doit contenir la même valeur
  - **__exact**: Pour une équivalence exacte
  - **__iexact**: Egalité insensible à la casse
  - **__in**: Tester l 'appartenance à une liste de valeurs
  - **__gt**: Teste de supériorité
  - **__range**: Tester l'appartenance à un intervalle
  - **__regex** : Tester la correspondance à une expression régulière
  - …

QuerySet API reference | Django documentation | Django (djangoproject.com)

REQUÊTES COMPLEXES AVEC LA CLASSE Q

- Pour des filtres plus complexes avec des opérations logiques AND(&), OR (|) et NOT (~), on utilise un objet Q (django.db.models.Q). Cet objet rend possible la définition, réutilisation et la combinaison de conditions.

```
from django.db.models import Q


Publisher.objects.get(
        Q(name__startswith='EN'),
        Q(address ="Paris")) |
        Q(website__endswith=".fr")
)
```

est équivalente à:

```
SELECT * from Publisher WHERE name LIKE 'EN%' AND (adress = 'Paris' OR
website = '%.fr')
```

https://docs.djangoproject.com/en/4.1/topics/db/queries/#complex-lookups-with-q-objects

Validateurs

- Les validateurs sont des fonctions qui prennent en paramètre une valeur, testent si celle-ci répond à certains critères et génèrent une erreur (**`ValidationError`**) sinon.

- Django fournit une collection de validateur via le module **`django.core.validators`**
  - RegexValidator
  - EmailValidator
  - MaxValueValidator | MinValueValidator
  - MaxLengthValidator | MinLengthValidator
  - …
- Sinon on peut définir des validateurs personnalisés

Validators | Django documentation | Django (djangoproject.com)

Validateurs

- Exemple

```
def validate_author_firstName(value):
if not '@' in value:
        raise ValidationError('@ is not allowed in first name')
    return value

class Author(models.Model):
    first_name = models.CharField(
                        max_length=30,
                        validators = [validate_author_firstName ]
                )
    last_name = models.CharField(
                        max_length=40,
                        validators = [
                            validators.MinLengthValidator (
                                2,
                                'enter 2 or more characters')
                        ]
                )
```

Validators | Django documentation | Django (djangoproject.com)

Atelier