Digital Career Institute

Python Course - Databases - ORM

Digital Career Institute

Model Relationships

Model Relationships

Models, like tables, can be related to each other using foreign keys.

Pseudo-code

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Picture(models.Model):
    """The User Pictures Model."""
    user = Foreign Key references User
    file = models.FileField(...)
```

Defining Relationships

Two models can be related using the **`ForeignKey`** field available in the **`models`** module.

users/models.py

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Picture(models.Model):
    """The User Pictures Model."""
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE
    )
    file = models.FileField()
```

Defining Relationships

Two models can be related using the **`ForeignKey`** field available in the **`models`** module.

users/models.py

```
class User(models.Model):
    """The User Mo
    email = models
    password = mod

class Picture(models.Model):
    """The User Pictures Model."""
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE
    )
    file = models.FileField()
```

A positional argument with the related model (as a callable or string) is required.

Defining Relationships

Two models can be related using the **`ForeignKey`** field available in the **`models`** module.

users/models.py

```
class User(models.Model):
    """The User Mo
    email = models
    password = moc

class Picture(models.Model):
    """The User Pictures Model."""
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE
    )
    file = models.FileField()
```

The keyword argument **`on_delete`** is also required.

On Delete

The `models` module offers named constants to reference the different options.

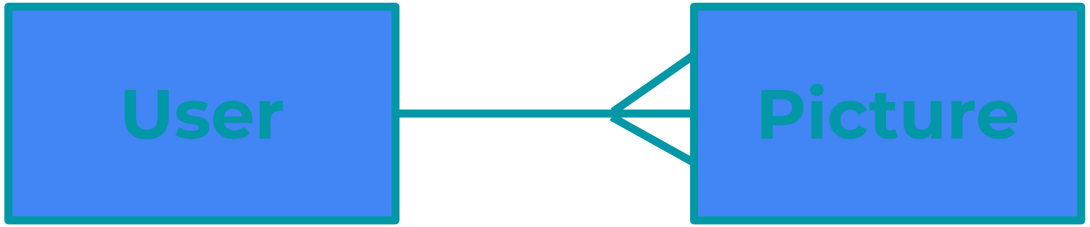
models.CASCADE	models.PROTECT
models.SET_NULL	models.RESTRICT
models.SET_DEFAULT	models.DO_NOTHING

Requires a default value.

Model Relationships *One-to-Many*

Digital Career Institute

One-to-Many Relationships



A user may have many pictures, but a picture can only belong to one user.

Defining Relationships

A **`ForeignKey`** field defines a **1:M** relationship and by default references the primary key in the related table.

Main Table		
User Table		
id	**email**	**password**
1	mary@...	1234
2	lou@...	pass
3	john@...	4321

1 ←------→ M

"Child" Table		
Picture Table		
id	**user_id**	**file**
1	3	cutie.png
2	1	holidays.png
3	1	home.png

A **`ForeignKey`** model field will create a database field named **`{related_model}_id`**.

Defining Relationships

The **`to_field`** argument will make the relationship reference another **unique** field.

users/models.py

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255,
                              unique=True)
    password = models.CharField(max_length=32)

class Picture(models.Model):
    """The User Pictures Model."""
    user = models.ForeignKey(
        User, on_delete=models.CASCADE,
        to_field="email"
    )
    file = models.FileField()
```

Defining Relationships

The `ForeignKey` field may reference any field with a unique constraint.

Main Table		
User Table		
id	**email**	**password**
1	mary@...	1234
2	lou@...	pass
3	john@...	4321

1 ←------→ M

“Child” Table		
Picture Table		
id	**user_id**	**file**
1	john@...	cutie.png
2	mary@...	holidays.png
3	mary@...	home.png

Defining Foreign Key Values

To create “child” objects, we can simply pass the parent objects as arguments of the foreign key field.

```
>>> # Create a new user
>>> mary = User.objects.create(email="mary@...", pass="1234")
>>> # Add pictures to mary
>>> Picture.objects.create(user=mary, file="home.png")
```

user is a **reference to the <u>object</u>** and requires an **object** to be assigned to it.

Defining Foreign Key Values

To create child objects, we can also use the identifiers.

```
>>> # Create a new user
>>> mary = User.objects.create(email="mary@...", pass="1234")
>>> # Add pictures to mary
>>> Picture.objects.create(user_id=mary.pk, file="home.png")
```

`user_id` is a **reference to the <u>table field</u>** and requires an **integer** to be assigned to it.

Querying Foreign Keys

Foreign keys can also be queried (`get`, `filter`, `exclude`,...) using object arguments.

```
>>> # Create a new user
>>> mary = User.objects.get(email="mary@...", pass="1234")
>>> # Get Mary's picture named home.png
>>> Picture.objects.get(user=mary, file="home.png")
```

`user` is still a **reference to the __object__** and it still requires an **object**.

Querying Foreign Keys

Query lookups can span to the related tables.

```
>>> # Getting all pictures from users whose
>>> # email starts with mary@
>>> Picture.object.filter(user__email__startswith="mary@")
<QuerySet [<Picture: home.png>, <Picture: holidays.png>]>
```

A model can be filtered based on any field in any related table.

Querying Foreign Keys

Query lookups can span to any level of related tables.

```
>>> # Getting all postal addresses in Germany
>>> Address.object.filter(
...     street__city__country__name="Germany")
...
```

In this example, the table `address` has a field pointing to a `street` table, which has a field pointing to a `city` table, with a field pointing to a `country` table that has a `name` field.

Backwards Relation

Related **new objects** can also be **created** from the main object.

```
>>> # Create a new user
>>> mary = User.objects.create(email="mary@...", pass="1234")
>>> # Add pictures to mary using its picture_set model manager
>>> pic = mary.picture_set.create(file="home.png")
```

A one-to-many relationship will automatically define a property named **`{model_name}_set`** as a model manager.

The ORM will automatically use Mary's identifier as a foreign key of the table row.

Creating Related Objects

Related **existing objects** can also be **added** from the main object.

```
>>> # Create a new user
>>> mary = User.objects.create(email="mary@...", pass="1234")
>>> # Add pictures to mary
>>> pic1 = Picture.objects.create(file="home.png")
>>> pic2 = Picture.objects.create(file="holidays.png")
>>> mary.picture_set.add(pic1, pic2)
```

As opposed to `create`, the `add` method requires objects that need to be already created at the database level.

Removing Related Objects

Related objects can also be **removed** from the main object.

```
>>> # Create a new user
>>> mary = User.objects.create(email="mary@...", pass="1234")
>>> # Set Mary's pictures
>>> home = Picture.objects.create(file="home.png")
>>> mary.picture_set.remove(home)
```

Querying Related Objects

Related objects can also be **queried** using the related manager.

```
>>> # Create a new user
>>> mary = User.objects.get(email="mary@digitalcar...")
>>> # Getting Mary's pictures which name starts with h
>>> mary.picture_set.filter(file__startswith="h")
<QuerySet [<Picture: home.png>, <Picture: holidays.png>]>
```

As with the standard model manager, the **`filter`** method can be used to query the related objects.

Selecting Related Data

When data from multiple tables is selected the **`select_related`** method
Is sometimes used to optimize the SQL query.

```
>>> # Get a picture. Hits the database
>>> pic = Picture.objects.get(pk=1)
>>> # Getting its user. Hits the database again.
>>> pic.user.name
Mary
>>> # Get the picture and the user. Hits the database.
>>> pic = Picture.objects.select_related("user").get(pk=1)
>>> # Showing its user. Does NOT hit the database again.
>>> pic.user.name
Mary
```

Related Managers

The related manager can be renamed using the argument `related_name`.

users/models.py

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Picture(models.Model):
    """The User Pictures Model."""
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name="pictures"
    )
    file = models.FileField()
```

Related Manager Name

The related manager can be renamed using the argument `related_name`.

users/models.py

```
>>> # Create a new user
>>> mary = User.objects.get(email="mary@digitalcar...")
>>> # Getting Mary's pictures which name starts with h
>>> mary.pictures.filter(file__startswith="h")
<QuerySet [<Picture: home.png>, <Picture: holidays.png>]>
```

```
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name="pictures"
    )
    file = models.FileField()
```

Related Managers

The parent object can also be filtered based on fields in the child table.

users/models.py

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Picture(models.Model):
    """The User Pictures Model."""
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name="pictures"
    )
    file = models.FileField()
```

Reverse Filters

The parent object can also be filtered based on fields in the child table.

users/models.py

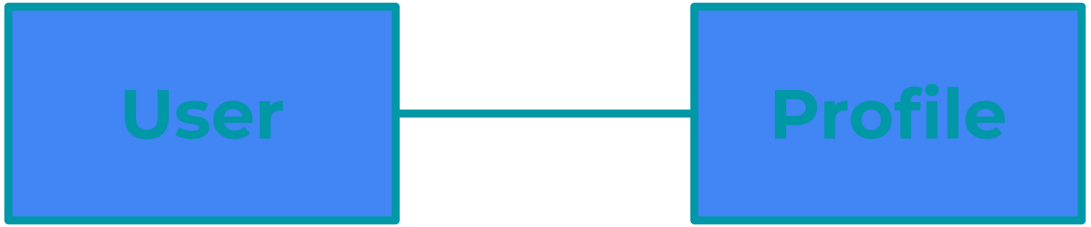
```
>>> # Getting users who have pictures which name starts with h
>>> User.objects.filter(pictures__name__startswith="h")
<QuerySet [<User: Mary>, <User: Louise>]>
```

```
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        related_name="pictures"
    )
    file = models.FileField()
```

Model Relationships
One-to-One

Digital Career Institute

One-to-One Relationships



A user may have only one profile, and a profile can only belong to a user.

Defining Relationships

One-to-one relationships can be implemented with standard foreign keys and a unique constraint.

users/models.py

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Profile(models.Model):
    """The User Profile Model."""
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        unique=True
    )
    data = models.JSONField()
```

A foreign key that has a unique constraint is limited to only one-to-one relationships between rows of the two tables.

Defining Relationships

A `ForeignKey` field with a **unique** constraint defines a **1:1** relationship.

User Table

id	email	password
1	mary@...	1234
2	lou@...	pass
3	john@...	4321

1 ←------→ 1

Profile Table

id	user_id	data
1	3	{“heading”:...
2	1	{“heading”:...
3	2	{“heading”:...

Defining Relationships

One-to raint.

```
(env)$ python manage.py makemigrations
WARNINGS:
shop.Picture.user: (fields.W342) Setting unique=True on a ForeignKey
has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a
OneToOneField.
```

d

between rows of the two tables.

```
class Profile(models.Model):
    """The User Profile Model."""
    user = models.ForeignKey(
        User,
        on_delete=models.CASCADE,
        unique=True
    )
    data = models.JSONField()
```

Defining Relationships

Django's ORM provides a specific field for these relationships.

users/models.py

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Profile(models.Model):
    """The User Profile Model."""
    user = models.OneToOneField(
        User,
        on_delete=models.CASCADE,
        primary_key=True
    )
    data = models.JSONField()
```

A `OneToOneField` is the recommended way of defining one-to-one relationships.

Arguments are essentially the same as the `ForeignKey`.

Defining Relationships

Two models can be related using a **`OneToOneField`**.

users/models.py

```
class User(models.Model):
    """The User Pr
    email = models
    password = mod

class Profile(models.Model):
    """The User Profile Model."""
    user = models.OneToOneField(
        User,
        on_delete=models.CASCADE,
        primary_key=True
    )
    data = models.JSONField()
```

The foreign key, in a 1:1 relationship, is a natural primary key, but it is not required.

Using the One-to-One Field

The one-to-one field can be used like the foreign key field.

```
>>> # Create a new user
>>> mary = User.objects.create(email="mary@...", pass="1234")
>>> # Create Mary's profile
>>> profile = Profile.objects.create(user=mary,
...                                  data={"empty": True})
...
>>> # Or create the profile using the identifiers
>>> profile = Profile.objects.create(user_id=mary.pk, …)
```

Backwards Relation

The **`OneToOneField`** also provides backwards relation.

```
>>> # Create a new user
>>> mary = User.objects.create(email="mary@...", pass="1234")
>>> # Create Mary’s profile
>>> Profile.objects.create(user=mary, data={"empty": True})
>>> # Get Mary’s profile data
>>> print(mary.profile.data)
{'empty': True}
```

Related Name

The related name can also be changed in a **`OneToOneField`**.

users/models.py

```
class User(models.Model):
    """The User Model."""
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Profile(models.Model):
    """The User Profile Model."""
    user = models.OneToOneField(
        User,
        on_delete=models.CASCADE,
        related_name="data"
    )
    company = models.CharField(max_length=100)
```

A **`OneToOneField`** is the recommended way of defining one-to-one relationships.

Arguments are essentially the same as the **`ForeignKey`**.

Related Name

The related name can also be changed in a **`OneToOneField`**.

users/models.py

```
>>> # Get Mary's profile company
>>> print(mary.data.company)
Digital Career Institute
```

```
5)
=32)

class Profile(models.Model):
    """The User Profile Model."""
    user = models.OneToOneField(
        User,
        on_delete=models.CASCADE,
        related_name="data"
    )
    company = models.CharField(max_length=100)
```

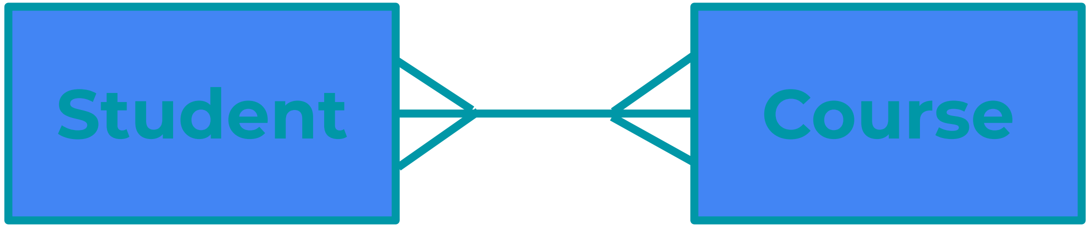
A **`OneToOneField`** is the recommended way of defining one-to-one relationships.

Arguments are essentially the same as the **`ForeignKey`**.

Model Relationships
Many-to-Many

Digital Career Institute

Many-to-Many Relationships



Each student may attend many courses, and each course can be attended by many students.

Defining Relationships

Two **`ForeignKey`** fields define a **M:N** relationship.

Student Table

id	email	password
1	mary@...	1234
2	lou@...	pass
3	john@...	4321

1 ↔ M

Intermediate Table

student_id	course_id
1	1
1	2
3	1

M ↔ 1

Course Table

id	title
1	Computer Basics
2	Intro to Programming
3	Django Framework

The combination of the two **`ForeignKey`** fields is the natural primary key of the intermediate table.

Defining Relationships in Django

Many-to-many relationships can be implemented with the **`ManyToManyField`**.

courses/models.py

```
class Student(models.Model):
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)

class Course(models.Model):
    students = models.ManyToManyField(
        Student,
        related_name="attends"
    )
    data = models.JSONField()
```

The name of the backwards relation can also be customized with **`related_name`**.

Using Many-to-Many Relationships

Many-to-many relationships always work with related managers.

```
>>> # Create a new student
>>> mary = Student.objects.create(email="mary@...", pass=...)
>>> # Create a new course
>>> django = Course.objects.create(title="Django")
>>> # Assign student to the course
>>> django.students.add(mary)
>>> # Assign course to the student
>>> mary.attends.add(django)
```

Using Many-to-Many Relationships

Many-to-many relationships always work with related managers.

```
>>> # Get a student
>>> mary = Student.objects.get(email="mary@...", pass=...)
>>> # Get a course
>>> django = Course.objects.get(title="Django")
>>> # Get mary’s courses
>>> mary.attends.all()
>>> # Get course’s students
>>> django.students.all()
```

Custom Intermediate Tables

Intermediate tables can be explicitly defined.

courses/models.py

```
class Course(models.Model):
    students = models.ManyToManyField(
        Student, through="Registration"
    )

class Registration(models.Model):
    student = models.ForeignKey(Student,...)
    course = models.ForeignKey(Course,...)
    enrolled_on = models.DateTimeField()
    passed_on = models.DateTimeField()
    score = models.PositiveSmallIntegerField()
```

This is often done when the intermediate table must hold additional details related to the relationship.

Using Many-to-Many Relationships

To store the information on the additional fields,
the query can be done directly on the intermediate model.

```
>>> # Create a new student
>>> mary = Student.objects.create(email="mary@...", pass=...)
>>> # Create a new course
>>> django = Course.objects.create(title="Django")
>>> # Register mary to the Django course
>>> Registration.objects.create(student=mary, course=django,
...                             start=1997, end=1998)
...
```

Using Many-to-Many Relationships

It can also be done from one model, using **`through_defaults`**.

```
>>> # Create a new student
>>> mary = Student.objects.create(email="mary@...", pass=...)
>>> # Create a new course
>>> django = Course.objects.create(title="Django")
>>> # Register mary to the Django course
>>> django.students.add(mary,
...     through_defaults={"start": 1997, "end": 1998}
... )
...
```

Recursive Many-to-Many Relationships

A **`ManyToManyField`** can reference its own table.

courses/models.py

```
class Student(models.Model):
    email = models.EmailField(max_length=255)
    password = models.CharField(max_length=32)
    friends = models.ManyToManyField(
        Student,
        symmetrical=False
    )
```

A self-reference can be symmetrical or asymmetrical.

By default, they are created as symmetrical.

Recursive Many-to-Many Relationships

A symmetrical relation will define the relationship in both directions.

```
>>> # Get two students
>>> mary = Student.objects.get(email="mary@...", pass=...)
>>> lou = Student.objects.get(email="lou@...", pass=...)
>>> # Add lou as mary's friend
>>> mary.friends.add(lou)
>>> # Get lou's friends
>>> lou.friends.all()
<QuerySet [{mary}]>
```

Recursive Many-to-Many Relationships

An asymmetrical relation will define the relationship in only one direction.

```
>>> # Get two students
>>> mary = Student.objects.get(email="mary@...", pass=...)
>>> lou = Student.objects.get(email="lou@...", pass=...)
>>> # Add lou to the following list of mary
>>> mary.follows.add(lou)
>>> # Get lou's following list
>>> lou.follows.all()
<QuerySet []>
```

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute
DCI