Model relationships

In this reading, you will explore the different types of relationships between Django models.

Primary key

In a relational database, each table that represents an entity has one column that has a unique value for each row. Such a column or field is known as the **Primary Key**.

If the primary key of one table appears as one of the fields in another table while having its own primary key, then it is called a **Foreign Key**.

For example, in a Products table, the **`ProductID`** field is its primary key.

In the Customers' table, the **`ProductID`** field, which refers to the product purchased by the customer, becomes the foreign key.

The idea behind designing related tables is to avoid **data redundancy** unnecessary repetition of the same data in many rows and ensure **data integrity.**

If the unique **`ProductID`** in the Customers' table is replaced by a longer naming convention for the product field, it will have to be entered every time a customer buys the product. This can lead to typing errors.

Similarly, if a product's ID is referred to in the Customers' table and is removed, other product details, such as price, will not be available.

Relational databases have a mechanism to prevent the deletion of the primary key if it is being used in the related table so that the data integrity is intact.

Since the Django models are mapped to the corresponding tables in the database, you can define a relationship such as this between the two model fields.

Types of Relationships

There are three types of relationships that exist:

One-to-One,

One-to-Many, and

Many-to-Many.

Let's explore these by beginning with a One-to-One relationship.

One-to-One relationship

If a primary key is in one model, and only one record exists in the other related model, the two models are said to have a one-to-one relationship.

Let's use an example of a college model and a principal model. A college can have only one principal or it can be similarly phrased as one person can be a principal of only one college.

The college model can be described as follows:

```python
class college(Model):
    CollegeID = models.IntegerField(primary_key = True)
    name = models.CharField(max_length=50)
    strength = models.IntegerField()
    website=models.URLField()
```

In the principal model, you must provide the **`CollegeID`** field as the foreign key. The **`on_delete`** option specifies the behavior in case the associated object in the primary model is deleted. The values are:

CASCADE: deletes the object containing the **`ForeignKey`**

PROTECT: Prevent deletion of the referenced object.

RESTRICT: Prevent deletion of the referenced object by raising **`RestrictedError`**

The principal model has the following field structure:

```python
class Principal(models.Model):
    CollegeID = models.OneToOneField(
                College,
                on_delete=models.CASCADE
                )
    Qualification = models.CharField(max_length=50)
    email = models.EmailField(max_length=50)
```

When you run a migration on these models, respective tables are created with equivalent SQL queries:

```
CREATE TABLE "myapp_college" ("CollegeID" integer NOT NULL PRIMARY KEY, "name" varchar(50) NOT NUL
```

```
CREATE TABLE "myapp_principal" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "Qualification" v
```

One-to-Many Relationship

In a One-to-Many relationship, one object of a model can be associated with one or more objects of another model. For example, a teacher is qualified to teach a subject, but there can be more than one teacher in a college who teaches the same subject.

The subject model is as below:

```python
class Subject(models.Model):
    Subjectcode = models.IntegerField(primary_key = True)
    name = models.CharField(max_length=30)
    credits = model.IntegerField()
```

The teacher model has its own primary key. It has a **`Foreignkey`** relating this model with the subject model.

```python
class Teacher(models.Model):
    TeacherID = models.ItegerField(primary_key=True)
    subjectcode=models.ForeignKey(
                Subject,
                on_delete=models.CASCADE
                  )
    Qualification = models.CharField(max_length=50)
    email = models.EmailField(max_length=50)
```

The following SQL queries will be executed when you run the migration:

```
CREATE TABLE "myapp_subject" ("Subjectcode" integer NOT NULL PRIMARY KEY, "name" varchar(30) NOT N
```

```
CREATE TABLE "myapp_teacher" ("TeacherID" integer NOT NULL PRIMARY KEY, "Qualification" varchar(5
```

```
CREATE INDEX "myapp_teacher_subjectcode_id_bef86dea" ON "myapp_teacher" ("subjectcode_id");
```

Many-to-Many Relationship

In a Many-to-Many relationship, multiple objects of one model can be associated with multiple objects of another model.

Let's redefine the relationship between the subject and teacher models in the above example. If more than one teacher can teach the same subject, a single teacher can teach more than one subject. So, there is a Many-to-Many relationship between the two. Django implements this with a Many-to-Many Field type. Let’s use it in the subject model.

The teacher model is straightforward.

```
class Teacher(models.Model):
    TeacherID = models.ItegerField(primary_key=True)
    Qualification = models.CharField(max_length=50)
    email = models.EmailField(max_length=50)
```

The design of the Subject model class reflects the Many-to-Many relationship.

```
class Subject(models.Model):
    Subjectcode = models.IntegerField(primary_key = True)
    name = models.CharField(max_length=30)
    credits = model.IntegerField()
    teacher = model.ManyToManyField(Teacher)
```

When the migrations are done, the following SQL queries will be executed to establish a many-to-many relationship.

CREATE TABLE "myapp_teacher" ("TeacherID" integer NOT NULL PRIMARY KEY, "Qualification" varchar(50) NOT NULL, "email" varchar(50) NOT NULL);

```
CREATE TABLE "myapp_teacher" ("TeacherID" integer NOT NULL PRIMARY KEY, "Qualification" varchar(5
```

```
CREATE TABLE "myapp_subject" ("Subjectcode" integer NOT NULL PRIMARY KEY, "name" varchar(30) NOT N
```

```
CREATE TABLE "myapp_subject_teacher" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "subject_i
```

```
CREATE UNIQUE INDEX "myapp_subject_teacher_subject_id_teacher_id_9b6a3c00_uniq" ON "myapp_subject_
```

```
CREATE INDEX "myapp_subject_teacher_subject_id_e87c76e7" ON "myapp_subject_teacher" ("subject_id"
```

```
CREATE INDEX "myapp_subject_teacher_teacher_id_359f8cce" ON "myapp_subject_teacher" ("teacher_id"
```