Model Relationship

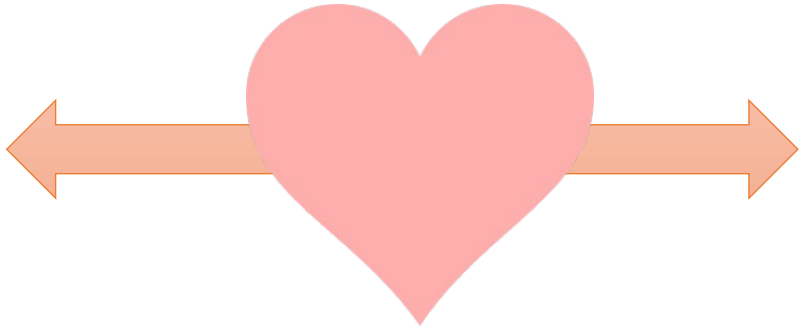
Django offers ways to define the three most common types of database relationships

- One to One Relationship
- Many to One Relationship
- Many to Many Relationships

One to One Relationship

When one row of table A can be linked to one row of table B.

Husband



Wife

One to One

User

ID	Username	Password
1	Rahul	rahul12
2	Sonam	sonam12
3	Kunal	kunal12

Page

ID	Page Name	Page Cat	Page Publish Date
1	Geekyshows	Programming	12-12-2000
2	World News	News	11-09-2003
3	DjangoLove	Programming	07-01-2001

One to One Relationship

One to One Relationship - To define a one-to-one relationship, use **OneToOneField**. You use it just like any other Field type by including it as a class attribute of your model.

OneToOneField requires a positional argument, the class to which the model is related.

Syntax:- OneToOneField(to, on_delete, parent_link=False, **options)

Where,

to - The class to which the model is related.

on_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on_delete argument. on_delete doesn't create an SQL constraint in the database.

parent_link - When True and used in a model which inherits from another concrete model, indicates that this field should be used as the link back to the parent class, rather than the extra OneToOneField which would normally be implicitly created by subclassing.

limit_choices_to - Sets a limit to the available choices for this field when this field is rendered using a ModelForm or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a Q object, or a callable returning a dictionary or Q object can be used.

One to One Relationship

related_name - The name to use for the relation from the related object back to this one. It's also the default value for related_query_name (the name to use for the reverse filter name from the target model).

If you'd prefer Django not to create a backwards relation, set related_name to '+' or end it with '+'.

related_query_name - The name to use for the reverse filter name from the target model. It defaults to the value of related_name or default_related_name if set, otherwise it defaults to the name of the model.

to_field - The field on the related object that the relation is to. By default, Django uses the primary key of the related object. If you reference a different field, that field must have unique=True.

swappable - Controls the migration framework's reaction if this ForeignKey is pointing at a swappable model. If it is True - the default - then if the ForeignKey is pointing at a model which matches the current value of settings.AUTH_USER_MODEL (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

One to One Relationship

db_constraint - Controls whether or not a constraint should be created in the database for this foreign key. The default is True, and that's almost certainly what you want; setting this to False can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

You have legacy data that is not valid.

You're sharding your database.

If this is set to False, accessing a related object that doesn't exist will raise its DoesNotExist exception.

on_delete

on_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on_delete argument. on_delete doesn't create an SQL constraint in the database.

The possible values for on_delete are found in django.db.models:

- CASCADE - Cascade deletes. Django emulates the behavior of the SQL constraint ON DELETE CASCADE and also deletes the object containing the ForeignKey.
- PROTECT - Prevent deletion of the referenced object by raising ProtectedError, a subclass of django.db.IntegrityError.
- SET_NULL - Set the ForeignKey null; this is only possible if null is True.
- SET_DEFAULT - Set the ForeignKey to its default value; a default for the ForeignKey must be set.
- SET() - Set the ForeignKey to the value passed to SET(), or if a callable is passed in, the result of calling it.
- DO_NOTHING - Take no action. If your database backend enforces referential integrity, this will cause an IntegrityError unless you manually add an SQL ON DELETE constraint to the database field.

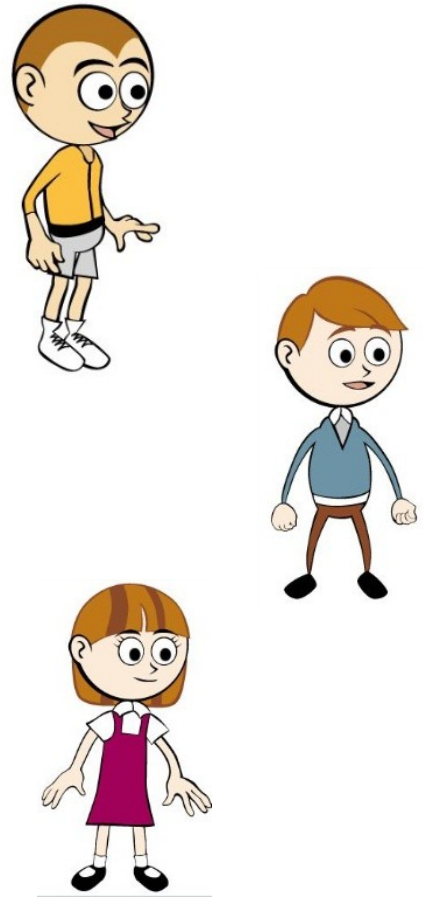
One to One Relationship

```
class User(models.Model):
 user_name = models.CharField(max_length=70)
 password = models.CharField(max_length=70)

class Page(models.Model):
 user = models.OneToOneField(User, on_delete=models.CASCADE)
 page_name = models.CharField(max_length=70)
 page_cat = models.CharField(max_length=70)
 page_publish_date = models.DateField()
```

Many to One Relationship

When one or more row of table B can be linked to one row of table A.



Many to One Relationship

Post

ID	Post Title	Post Cat	Post Publish Date	User_id
1	Title 1	django	12-12-2000	1
2	Title 2	django	11-09-2003	1
3	Title 3	python	07-01-2001	2

User

ID	Username	Password
1	Rahul	rahul12
2	Sonam	sonam12
3	Kunal	kunal12

Many to One Relationship

Many to One Relationship - To define a many-to-one relationship, use **ForeignKey**. You use it just like any other Field type: by including it as a class attribute of your model.

A many-to-one relationship requires two positional arguments: the class to which the model is related and the on_delete option.

Syntax:- ForeignKey(to, on_delete, **options)

to - The class to which the model is related.

on_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on_delete argument. on_delete doesn’t create an SQL constraint in the database.

limit_choices_to - Sets a limit to the available choices for this field when this field is rendered using a ModelForm or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a Q object, or a callable returning a dictionary or Q object can be used.

Many to One Relationship

related_name - The name to use for the relation from the related object back to this one. It’s also the default value for related_query_name (the name to use for the reverse filter name from the target model).

If you’d prefer Django not to create a backwards relation, set related_name to '+' or end it with '+'.

related_query_name - The name to use for the reverse filter name from the target model. It defaults to the value of related_name or default_related_name if set, otherwise it defaults to the name of the model.

to_field - The field on the related object that the relation is to. By default, Django uses the primary key of the related object. If you reference a different field, that field must have unique=True.

swappable - Controls the migration framework’s reaction if this ForeignKey is pointing at a swappable model. If it is True - the default - then if the ForeignKey is pointing at a model which matches the current value of settings.AUTH_USER_MODEL (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

Many to One Relationship

db_constraint - Controls whether or not a constraint should be created in the database for this foreign key. The default is True, and that's almost certainly what you want; setting this to False can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

You have legacy data that is not valid.

You're sharding your database.

If this is set to False, accessing a related object that doesn't exist will raise its DoesNotExist exception.

on_delete

on_delete - When an object referenced by a ForeignKey is deleted, Django will emulate the behavior of the SQL constraint specified by the on_delete argument. on_delete doesn't create an SQL constraint in the database.

The possible values for on_delete are found in django.db.models:

- CASCADE - Cascade deletes. Django emulates the behavior of the SQL constraint ON DELETE CASCADE and also deletes the object containing the ForeignKey.
- PROTECT - Prevent deletion of the referenced object by raising ProtectedError, a subclass of django.db.IntegrityError.
- SET_NULL - Set the ForeignKey null; this is only possible if null is True.
- SET_DEFAULT - Set the ForeignKey to its default value; a default for the ForeignKey must be set.
- SET() - Set the ForeignKey to the value passed to SET(), or if a callable is passed in, the result of calling it.
- DO_NOTHING - Take no action. If your database backend enforces referential integrity, this will cause an IntegrityError unless you manually add an SQL ON DELETE constraint to the database field.

Many to One Relationship

```
class User(models.Model):
 user_name = models.CharField(max_length=70)
 password = models.CharField(max_length=70)

class Post(models.Model):
 user = models.ForeignKey(User, on_delete=models.CASCADE)
 post_title = models.CharField(max_length=70)
 post_cat = models.CharField(max_length=70)
 post_publish_date = models.DateField()
```

Many to Many Relationship

When one row of table A can be linked to one or more rows of table B, and vice-versa.

Many to Many Relationship

User

ID	Username	Password
1	Rahul	rahul12
2	Sonam	sonam12
3	Kunal	kunal12

Song

ID	Song Name	Song Duration
1	Tum hi ho	5
2	Kuch Kuch	7
3	Dil to hai dil	8

song_user

ID	Song_id	User_id
1	1	1
2	1	2
3	2	3
4	2	1

Many to Many Relationship

Many to Many Relationships - To define a many-to-many relationship, use ManyToManyField. You use it just like any other Field type: by including it as a class attribute of your model.

ManyToManyField requires a positional argument: the class to which the model is related.

Syntax:- ManyToManyField(to, **options)

Where,

to - The class to which the model is related.

related_name - The name to use for the relation from the related object back to this one. It's also the default value for related_query_name (the name to use for the reverse filter name from the target model).

If you'd prefer Django not to create a backwards relation, set related_name to '+' or end it with '+'.

related_query_name - The name to use for the reverse filter name from the target model. It defaults to the value of related_name or default_related_name if set, otherwise it defaults to the name of the model.

Many to Many Relationship

limit_choices_to - Sets a limit to the available choices for this field when this field is rendered using a ModelForm or the admin (by default, all objects in the queryset are available to choose). Either a dictionary, a Q object, or a callable returning a dictionary or Q object can be used.

symmetrical - If you do not want symmetry in many-to-many relationships with self, set symmetrical to False. This will force Django to add the descriptor for the reverse relationship, allowing ManyToManyField relationships to be non-symmetrical. Only used in the definition of ManyToManyFields on self.

through - Django will automatically generate a table to manage many-to-many relationships. However, if you want to manually specify the intermediary table, you can use the through option to specify the Django model that represents the intermediate table that you want to use.

through_fields - Only used when a custom intermediary model is specified. Django will normally determine which fields of the intermediary model to use in order to establish a many-to-many relationship automatically. through_fields accepts a 2-tuple ('field1', 'field2'), where field1 is the name of the foreign key to the model the ManyToManyField is defined on , and field2 the name of the foreign key to the target model.

Many to Many Relationship

db_table - The name of the table to create for storing the many-to-many data. If this is not provided, Django will assume a default name based upon the names of: the table for the model defining the relationship and the name of the field itself.

db_constraint - Controls whether or not constraints should be created in the database for the foreign keys in the intermediary table. The default is True, and that's almost certainly what you want; setting this to False can be very bad for data integrity. That said, here are some scenarios where you might want to do this:

You have legacy data that is not valid.

You're sharding your database.

It is an error to pass both db_constraint and through.

swappable - Controls the migration framework's reaction if this ManyToManyField is pointing at a swappable model. If it is True - the default - then if the ManyToManyField is pointing at a model which matches the current value of settings.AUTH_USER_MODEL (or another swappable model setting) the relationship will be stored in the migration using a reference to the setting, not to the model directly.

Many to Many Relationship

Example:-

```
class User(models.Model):
 user_name = models.CharField(max_length=70)
 password = models.CharField(max_length=70)

class Song(models.Model):
 user = models.ManyToManyField(User)
 song_name = models.CharField(max_length=70)
 song_duration = models.IntegerField()
```