Django Migrations

Summary: in this tutorial, you'll learn how to create models and use Django migrations to create database tables.

Introduction to Django migration commands

When working with Django, you don't need to write SQL to create new tables or make changes to existing tables. Instead, you use Django migrations.

Django migrations allow you to propagate the changes that you make to the models to the database via the command line.

Django provides you with some commands for creating new migrations based on the changes that you made to the model and applying the migrations to the database.

The process for making changes to models, creating migrations, and applying the changes to the database is as follows:

- First, define new models or make changes to existing models.
- Second, make new migrations by running the `makemigrations` command.
- Third, apply the changes from the models to the database by executing the `migrate` command.

Suppose that you define the `Post` models in the `blog` application like this:

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
```

```
class Post(models.Model):
    title = models.CharField(max_length=120)
    content = models.TextField()
    published_at = models.DateTimeField(default=timezone.now)
    author = models.ForeignKey(User, on delete=models.CASCADE)


    def__str__(self):
        return self.title
```

and you can create a new migration using the `makemigrations` command:

```
python manage.py makemigrations
```

The `makemigrations` command scans the `models.py` file, detects changes, and makes corresponding migrations. It'll show the following output:

```
Migrations for 'blog':
  blog\migrations\0001_initial.py
    - Create model Post
```

Behind the scene, the command creates the file `migrations\0001_initial.py` file.

To preview the SQL that Django will run to create the `blog_post` table in the database, you use the `sqlmigrate` command:

```
python manage.py sqlmigrate blog 0001
```

In this `sqlmigrate` command, the `blog` is the name of the application and `0001` is the migration number.

It'll output the following:

```
BEGIN;
--
-- Create model Post
```

```
--
CREATE TABLE "blog_post" (
  "id" integer NOT NULL PRIMARY KEY AUTOINCREMENT,
  "title" varchar(120) NOT NULL,
  "content" text NOT NULL,
  "published_at" datetime NOT NULL,
  "author_id" integer NOT NULL REFERENCES "auth_user" ("id")
   DEFERRABLE INITIALLY DEFERRED
);

CREATE INDEX "blog_post_author_id_dd7a8485"
ON "blog post" ("author id");

COMMIT;
```

To apply the changes to the database, you execute the `migrate` command:

```
python manage.py migrate
```

It'll show the following output:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
```

```
  Applying auth.0010_alter_group_name_max_length... OK
  Applying  auth.0011_update_proxy_permissions...  OK
  Applying  auth.0012_alter_user_first_name_max_length...  OK
  Applying blog.0001_initial... OK
  Applying  sessions.0001_initial...  OK
```

Note that besides applying the migration for the `Post` model, Django also applied the migrations for the built-in models used in authentication, authorization, sessions, etc.

If you execute the `migrate` command again and there are no unapplied migrations, the command will output the following:

```
Operations to perform:
  Apply all migrations: admin, auth, blog, contenttypes, sessions, users
Running migrations:
  No migrations to apply.
```

To list the project migrations and their status, you use the `showmigrations` command:

```
python manage.py showmigrations
```

Output:

```
admin
 [X] 0001_initial
 [X] 0002_logentry_remove_auto_add
 [X] 0003_logentry_add_action_flag_choices
auth
 [X] 0001_initial
 [X] 0002_alter_permission_name_max_length
 [X] 0003_alter_user_email_max_length
 [X] 0004_alter_user_username_opts
 [X] 0005_alter_user_last_login_null
 [X] 0006_require_contenttypes_0002
 [X] 0007_alter_validators_add_error_messages
 [X] 0008_alter_user_username_max_length
 [X] 0009_alter_user_last_name_max_length
```

```
 [X] 0010_alter_group_name_max_length
 [X] 0011_update_proxy_permissions
 [X] 0012_alter_user_first_name_max_length
blog
 [X] 0001_initial
contenttypes
 [X] 0001_initial
 [X] 0002_remove_content_type_name
sessions
 [X] 0001_initial
```

Summary

- Use the `makemigrations` command to make migrations based on the changes that you made to the models.
- Use the `migrate` command to apply changes from models to the database.
- Use the `sqlmigrate` command to view the generated SQL based on the model.
- Use the `showmigrations` command to list all migrations and their status in the project.