Django Models

Django models are Python classes that define the structure and behavior of your database tables. Django's Object-Relational Mapping (ORM) system allows you to interact with the database using Python code, making it easy to perform database operations like creating, reading, updating, and deleting records.

1. Creating and Managing Database Models

In Django, each model maps to a single database table. Here's how you create and manage models:

a. Defining a Model

To define a model, you create a class that inherits from `django.db.models.Model`. Each attribute of the class represents a database field.

Example:

python

```
from django.db import models

class Author(models.Model):
    name = models.CharField(max_length=100)
    birthdate = models.DateField()

    def __str__(self):
        return self.name

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(Author, on_delete=models.CASCADE)
    published_date = models.DateField()
    isbn = models.CharField(max_length=13)
```

```
    def __str__(self):
        return self.title
```

- **Fields:** Each field in a Django model is represented by an instance of a `Field` class. Common field types include `CharField`, `DateField`, `IntegerField`, `BooleanField`, and `ForeignKey`.
- **ForeignKey:** Establishes a many-to-one relationship between models. In this example, each `Book` is linked to a single `Author`.
- **`__str__()` method:** Provides a human-readable representation of the model instance, which is useful when displaying objects in the Django admin or shell.

b. Working with Field Options

Fields in a Django model can have options such as `max_length`, `default`, `null`, and `blank`.

Example:

python

```
class Publisher(models.Model):
    name = models.CharField(max_length=100)
    established = models.IntegerField(default=2000)
    is_active = models.BooleanField(default=True)

    def __str__(self):
        return self.name
```

- `max_length=100`: Limits the length of the `name` field to 100 characters.
- `default=2000`: Sets a default value for the `established` field.
- `is_active`: A boolean field that defaults to `True`.

2. Using Django ORM for Database Operations

Django's ORM allows you to interact with your database using Python. Here are the basics of using the ORM:

a. Creating and Saving Objects

To create a new record in the database, instantiate a model and call the `save()` method.

Example:

python

```
# Create and save a new Author
author = Author(name='George Orwell', birthdate='1903-06-25')
author.save()

# Create and save a new Book
book = Book(title='1984', author=author,
published_date='1949-06-08', isbn='9780451524935')
book.save()
```

b. Querying the Database

Django provides a powerful query API for retrieving data from the database.

Example:

python

```
# Get all books
all_books = Book.objects.all()

# Get a single book by primary key (id)
book = Book.objects.get(pk=1)

# Filter books by author
orwell_books = Book.objects.filter(author__name='George Orwell')

# Get books published after 1950
books_after_1950 =
Book.objects.filter(published_date__gt='1950-01-01')

# Order books by published date
```

```
ordered_books = Book.objects.all().order_by('published_date')
```

- **`filter()`**: Returns a queryset of objects that match the given criteria.
- **`get()`**: Returns a single object that matches the query. If no match is found or multiple matches are found, it raises an exception.
- **`order_by()`**: Orders the queryset by the specified field.

c. Updating and Deleting Objects

Updating: You can update an object by modifying its attributes and calling `save()`.

python

```
book = Book.objects.get(pk=1)
book.title = 'Nineteen Eighty-Four'
book.save()
```

Deleting: To delete an object, use the `delete()` method.

python

```
book = Book.objects.get(pk=1)
book.delete()
```

d. Complex Queries

Django's ORM supports complex queries with methods like `annotate()`, `aggregate()`, `Q objects` for complex lookups, and more.

Example:

python

```
from django.db.models import Q

# Get all books by George Orwell or published before 1950
```

```
books = Book.objects.filter(Q(author__name='George Orwell') |
Q(published_date__lt='1950-01-01'))
```

3. Handling Migrations

Migrations are Django's way of propagating changes you make to your models (like adding a field) into your database schema.

a. Creating Migrations

Whenever you create or modify a model, you need to create a migration to reflect those changes in the database.

bash

```
python manage.py makemigrations
```

This command creates migration files that record the changes to your models.

b. Applying Migrations

After creating migrations, apply them to your database with the following command:

bash

```
python manage.py migrate
```

This command updates your database schema according to the migrations.

c. Viewing Migration History

You can view the migration history and the status of migrations with:

bash

```
python manage.py showmigrations
```

d. Rolling Back Migrations

To roll back migrations, you can specify a migration number to which you want to revert:

bash

```
python manage.py migrate myapp 0001
```

This command rolls back all migrations in the `myapp` application to the specified migration.

Summary

- **Models:** Define your database schema using Django models, which are Python classes that represent database tables.
- **Django ORM:** Use Django's ORM to perform database operations like creating, reading, updating, and deleting records.
- **Migrations:** Manage database schema changes with Django's migration system, which tracks and applies changes to your database.