In Django, **models** are classes that represent the structure of your data. Each model corresponds to a table in the database, where each attribute of the model represents a column in that table. Django models are defined in Python, making it easier to manage database structures without needing to write SQL directly.
Defining Models:

- Models are defined as classes that inherit from `django.db.models.Model`.
- Each attribute in the class represents a database field, with the field type specified by Django's field classes like `CharField`, `IntegerField`, `DateField`, etc.

Creating Fields:

- Each field in the model maps to a column in the database table.
- You define the field type by using various field classes, and you can also set constraints, default values, and more.

Django automatically sets up SQLite when you create a new project, as it is easy to configure and works well for small to medium-sized applications or for development purposes.

Key Points about SQLite in Django:

1. **Ease of Use**: SQLite is a file-based database, meaning it stores data in a single file. This simplicity makes it convenient for testing, prototyping, and small applications.

Configuration: Django sets up SQLite as the default in the `settings.py` file under the `DATABASES` setting:
python
Copy code

```
DATABASES = {
    'default': {
```

```
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

2.
3. **File Location**: The SQLite database file is usually named `db.sqlite3` and is located in the project's base directory.
4. **Switching Databases**: For production environments, developers often switch to more robust databases like PostgreSQL, MySQL, or Oracle, which are better suited for handling larger amounts of data and higher levels of traffic.

Changing the Database

To switch to another database, you would update the `DATABASES` setting in `settings.py` to configure the new database engine and credentials.

Fields in Models

In Django models, fields define the types of data each column in the database table will store. Each field type corresponds to a specific database column type (like text, integer, date, etc.), and these fields can have various attributes to customize them.

Here are some of the most common fields used in Django models, along with their frequently used attributes:

---

1. `CharField`

Used for short text fields like names, titles, etc.

- **Attributes**:
    - `max_length` (required): Maximum number of characters allowed.

    - `blank`: Allows the field to be empty in forms (`blank=True`).
    - `null`: Allows the field to store `NULL` in the database (`null=True`).
    - `unique`: Ensures that the field value is unique across all records (`unique=True`).

python
Copy code
```
name = models.CharField(max_length=100, blank=True, unique=True)
```

-

---

2. `TextField`

Used for long text fields like descriptions, content, etc.

- **Attributes**:
    - `blank`: Allows the field to be empty in forms.
    - `null`: Allows `NULL` values in the database.

python
Copy code
```
description = models.TextField(blank=True, null=True)
```

-

---

3. `IntegerField`

Used for storing integers.

- **Attributes**:
    - `blank`: Allows the field to be empty in forms.
    - `null`: Allows `NULL` values in the database.
    - `default`: Sets a default value for the field.

python
Copy code
```
age = models.IntegerField(null=True, default=0)
```

-

---

4. `FloatField`

Used for floating-point numbers.

- **Attributes**:
  - `blank`: Allows the field to be empty in forms.
  - `null`: Allows `NULL` values in the database.
  - `default`: Sets a default value for the field.

python
Copy code
```
price = models.FloatField(default=0.0)
```

-

---

5. `BooleanField`

Used for storing `True` or `False` values.

- **Attributes**:
  - `default`: Sets the default value (`True` or `False`).

python
Copy code
```
is_active = models.BooleanField(default=True)
```

-

---

6. `DateField`

Used for storing date values (year, month, day).

- **Attributes**:
  - `blank`: Allows the field to be empty in forms.
  - `null`: Allows `NULL` values in the database.
  - `auto_now`: Automatically sets the field to the current date every time the object is saved.
  - `auto_now_add`: Automatically sets the field to the current date only when the object is first created.

python
Copy code
```
published_date = models.DateField(auto_now_add=True)
```

-

---

7. `DateTimeField`

Used for storing date and time values.

- **Attributes**:
  - `blank`: Allows the field to be empty in forms.
  - `null`: Allows `NULL` values in the database.
  - `auto_now`: Automatically sets the field to the current date and time every time the object is saved.
  - `auto_now_add`: Automatically sets the field to the current date and time only when the object is created.

python
Copy code
```
last_updated = models.DateTimeField(auto_now=True)
```

-

---

8. `EmailField`

Used to store and validate email addresses.

- **Attributes**:
  - `blank`: Allows the field to be empty in forms.
  - `null`: Allows `NULL` values in the database.
  - `unique`: Ensures email addresses are unique.

python
Copy code
```
email = models.EmailField(unique=True)
```

-

---

9. `URLField`

Used to store and validate URLs.

- **Attributes**:
  - `blank`: Allows the field to be empty in forms.
  - `null`: Allows `NULL` values in the database.

python
Copy code
```
website = models.URLField(blank=True, null=True)
```

-

---

10. `ForeignKey`

Used to define a many-to-one relationship with another model.

- **Attributes**:
  - `to` (required): The related model.
  - `on_delete` (required): Defines behavior when the related object is deleted (e.g., `models.CASCADE`, `models.SET_NULL`).

    - `related_name`: Custom name for reverse relationships.
    - `null`: Allows `NULL` values in the database.

python
Copy code
```
author = models.ForeignKey(User, on_delete=models.CASCADE, related_name='books')
```

-

---

11. `ManyToManyField`

Used to define a many-to-many relationship with another model.

- **Attributes**:
    - `to` (required): The related model.
    - `related_name`: Custom name for reverse relationships.
    - `blank`: Allows the field to be empty in forms.

python
Copy code
```
categories = models.ManyToManyField(Category, related_name='books', blank=True)
```

-

---

12. `OneToOneField`

Used to define a one-to-one relationship with another model.

- **Attributes**:
    - `to` (required): The related model.
    - `on_delete` (required): Defines behavior when the related object is deleted.
    - `related_name`: Custom name for reverse relationships.
    - `null`: Allows `NULL` values in the database.

python
Copy code
```
profile = models.OneToOneField(Profile,
on_delete=models.CASCADE, null=True)
```

-

---

13. `SlugField`

Used to store slugs, typically for URLs (e.g., "my-first-post"),seo

- **Attributes**:
  - `max_length`: Maximum length of the slug (default is 50).
  - `unique`: Ensures the slug is unique.
  - `blank`: Allows the field to be empty in forms.
  - `null`: Allows `NULL` values in the database.

python
Copy code
```
slug = models.SlugField(max_length=100, unique=True)
```

-

---

14. `ImageField`

Used to store image file paths (requires `Pillow` library).

- **Attributes**:
  - `upload_to`: Specifies the folder where images will be saved.
  - `blank`: Allows the field to be empty in forms.
  - `null`: Allows `NULL` values in the database.
- python
  Copy code

```
image = models.ImageField(upload_to='images/', blank=True, null=True)
```

Project

ProjectName: employeeproject
Project App:modelsapp

Step 1: Register in setting.py + os setting
Import os and at line59 `'DIRS': [os.path.join(BASE_DIR, 'templates')],`

Step 2: Define Model

```
from django.db import models

# Create your models here.
class EmployeeModel(models.Model):
    eno = models.IntegerField()
    name = models.CharField(max_length=70)
    esal =  models.FloatField()
    eaddr = models.CharField(max_length=70)

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

Then python manage.py make migrations
python manage.py make migrate
makemigrations: Creates migration files that represent changes to your database schema.
migrate: Applies those changes to the database.

Step 3:

```
from django.contrib import admin
from .models import EmployeeModel
# Register your models here.
# admin.site.register(EmployeeModel)

class EmployeeAdmin(admin.ModelAdmin):
```

```
    list_display = ['eno', 'name', 'esal','eaddr']

admin.site.register(EmployeeModel, EmployeeAdmin)
```

Create superuser

Step 4:

```
from django.shortcuts import render
from .models import EmployeeModel

# Create your views here.

def employeeview(request):
    employee = EmployeeModel.objects.all()
    return render(request, 'employee.html', {'employee': employee})
```

Step 5:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Document</title>
</head>
<body>
    <h1>Employee data</h1>
    {% if employee %}
    <table>
        <thead>
            <th>Eno</th>
            <th>E Name</th>
            <th>E salary</th>
            <th>E address</th>
        </thead>

        <tbody>
            {% for emp in employee %}
            <tr>
                <td>{{ emp.eno }}</td>
```

```
                <td>{{ emp.name }}</td>
                <td>{{ emp.esal }}</td>
                <td>{{ emp.eaddr }}</td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
    {% else %}
    <p>No employee data found</p>
    {% endif %}
</body>
</html>
```

Step 6:

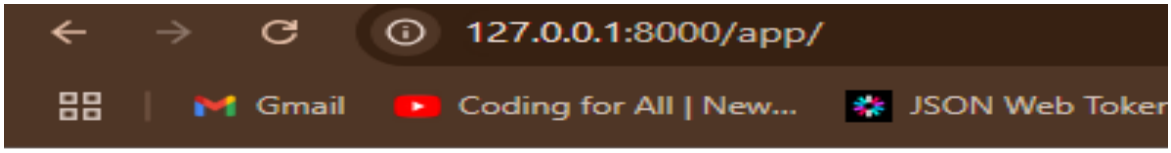
App urls

```
from django.urls import path
from . import views


urlpatterns = [
    path("app/", views.employeeview),
]
```

Step 7: Project urls

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include("modelsapp.urls")),
]
```

Step 8: Run server



Employee data

Eno	E Name	E salary	E address
101	Deepika	35000.0	Banglore
102	Shreya	40000.0	Mysore
103	Vijay	70000.0	AndraPradesh
104	Gowtham	50000.0	Hyderbad
105	Rajesh	90000.0	Btm