Here's a clear, practical step-by-step guide to creating a model in **Django**:

□ Step 1: Create or open a Django project

If you don't already have a project:

```
django-admin startproject myproject
cd myproject
```

Run the server to confirm everything works:

```
python manage.py runserver
```

□ Step 2: Create an app

Models live inside apps.

```
python manage.py startapp myapp
```

Then register the app in `settings.py`:

```
INSTALLED_APPS = [
    ...
    'myapp',
]
```

□ Step 3: Define your model

Go to `myapp/models.py` and create a class.

Example:

```
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)
    description = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)

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

 What's happening:

- `models.Model` → base class
- Fields define database columns
- `__str__` → readable name in admin

Step 4: Create migrations

Tell Django to prepare database changes:

```
python manage.py makemigrations
```

Step 5: Apply migrations

Actually create the table in your database:

```
python manage.py migrate
```

Step 6: Register model in admin (optional but useful)

Open `myapp/admin.py`:

```
from django.contrib import admin
from .models import Product

admin.site.register(Product)
```

Create a superuser:

```
python manage.py createsuperuser
```

Then visit:

```
http://127.0.0.1:8000/admin/
```

□ Step 7: Test your model in shell

```
python manage.py shell
from myapp.models import Product

p = Product(name="Laptop", price=999.99, description="Gaming laptop")
p.save()

Product.objects.all()
```

□ Common Field Types

- `CharField` → short text
- `TextField` → long text
- `IntegerField` → numbers
- `FloatField` / `DecimalField` → decimals
- `BooleanField` → true/false
- `DateTimeField` → date & time

Let's go beyond basic models and understand relationships in **Django**—this is where things start to feel like real-world applications.

□ 1. ForeignKey (One-to-Many)

Concept

One object can be related to many others.

□ Example:

- One **Author** → many **Books**

Define models

```
from django.db import models

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

    def __str__(self):
        return self.name


class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        Author,
        on_delete=models.CASCADE,
        related_name='books'
    )

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

Key things to understand

- `ForeignKey(Author)` → creates the relationship
- `on_delete=models.CASCADE` → if author is deleted, books are deleted
- `related_name='books'` → allows reverse access

Usage in shell

```
author = Author.objects.create(name="J.K. Rowling")

Book.objects.create(title="Harry Potter 1", author=author)
Book.objects.create(title="Harry Potter 2", author=author)

# Forward relation
book = Book.objects.first()
book.author

# Reverse relation
author.books.all()
```

---

2. ManyToMany (Many-to-Many)

Concept

Many objects relate to many others.

Example:

- One **Student** → many **Courses**
- One **Course** → many **Students**

Define models

```
class Course(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name


class Student(models.Model):
    name = models.CharField(max_length=100)
    courses = models.ManyToManyField(
        Course,
        related_name='students'
    )

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

What Django does behind the scenes

Django automatically creates a **junction table** like:

```
student_id | course_id
```

You don’t have to manage it manually.

Usage in shell:-

Open the Django shell

Run this in your project folder:

```
python manage.py shell
```

 This opens an interactive Python environment with your Django project loaded.

---

2. Import your models

```
from myapp.models import *


student = Student.objects.create(name="John")
course1 = Course.objects.create(name="Math")
course2 = Course.objects.create(name="Science")

# Add relationships
student.courses.add(course1, course2)

# Get courses of student
student.courses.all()

# Reverse relation
course1.students.all()
```

Important Differences

Feature	ForeignKey	ManyToMany
Type	One-to-Many	Many-to-Many
Field location	On "many" side	Either model
DB structure	Column in table	Separate table

□ When to use what?

- Use **ForeignKey** when:
  - One object clearly belongs to another
  - Example: Order → User
- Use **ManyToMany** when:
  - Both sides can have multiple connections
  - Example: Tags ↔ Posts

Let's complete the picture with **One-to-One relationships** in **Django**—they're simpler than they sound but very useful.

□ What is One-to-One?

Concept

Each record in one model is linked to exactly **one** record in another model.

□ Example:

- One **User** → one **Profile**
- One **Car** → one **Engine**

Think of it as extending a model without modifying it directly.

□ Define One-to-One relationship

Example: User & Profile

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

class Profile(models.Model):
    user = models.OneToOneField(
```

```
        User,
        on_delete=models.CASCADE,
        related_name='profile'
    )
    bio = models.TextField()
    profile_pic = models.ImageField(upload_to='profiles/')

    def __str__(self):
        return self.user.username
```

□ Key points

- `OneToOneField(User)` → ensures **only one profile per user**
- `on_delete=models.CASCADE` → deleting user deletes profile
- `related_name='profile'` → access via `user.profile`

□ Usage in shell

```
from django.contrib.auth.models import User
from myapp.models import Profile

user = User.objects.create(username="john")

# Create profile
profile = Profile.objects.create(user=user, bio="Hello!")

# Access profile from user
user.profile

# Access user from profile
profile.user
```

□ Important behavior

- If you try to create **another profile for the same user**, Django will raise an error □
- That's what makes it different from ForeignKey

□ OneToOne vs ForeignKey

Feature	OneToOne	ForeignKey
Relation	1 ↔ 1	1 → many
Uniqueness	Enforced	Not enforced
Use case	Extend model	Group objects

Product and Category with a ForeignKey—a very common Django pattern. Let's break it down clearly.

□ Concept: Product → Category (ForeignKey)

□ One **Category** can have many **Products**
□ Each **Product** belongs to one **Category**

This is a **one-to-many relationship**.

□ Step 1: Define models

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return self.name


class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=10, decimal_places=2)

    category = models.ForeignKey(
        Category,
        on_delete=models.CASCADE,
        related_name='products'
    )

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

□ What this means

- `category = models.ForeignKey(Category)`
  → Each product is linked to **one category**
- `on_delete=models.CASCADE`
  → If a category is deleted, all its products are deleted too

- `related_name='products'`
  → Lets you access products from a category like:

  `category.products.all()`

□ Database structure (simplified)

Category table
`id | name`

Product table
`id | name | price | category_id`

□ `category_id` is the ForeignKey column

□ Step 2: Use in Django shell

```
from myapp.models import Category, Product
```

Create category
```
c = Category.objects.create(name="Electronics")
```

Create products
```
Product.objects.create(name="Laptop", price=50000, category=c)
Product.objects.create(name="Phone", price=20000, category=c)
```

□ Query examples

Get product and its category
```
p = Product.objects.first()
p.category
```

Get all products in a category
```
c.products.all()
```

□ Real-world examples

- Electronics → Laptop, Phone, TV
- Clothing → Shirt, Jeans

- Grocery → Rice, Milk

□ Common mistakes

□ Forgetting to pass category

```
Product.objects.create(name="Laptop", price=50000)  # ERROR
```

✔ Fix:

```
Product.objects.create(name="Laptop", price=50000, category=c)
```