**https://www.linkedin.com/posts/pavanmehta-developer\_django-rest-framework-drf-guide-activity-7365340620925612032-6Ko2?utm\_source=share&utm\_medium=member\_desktop&rcm=ACoAAFCh8yIB8k8v0fVVR1wqWrd7WCNb0JwzN6E**

**https://www.linkedin.com/posts/afoke-reality-igho-523041235\_webdevelopment-django-react-activity-7367368241473683456-jOkB?utm\_source=share&utm\_medium=member\_desktop&rcm=ACoAAFCh8yIB8k8v0fVVR1wqWrd7WCNb0JwzN6E**

**https://www.linkedin.com/posts/madhuban-khatri\_django-reactjs-fullstackdevelopment-activity-7362737972019937280-fwIn?utm\_source=share&utm\_medium=member\_desktop&rcm=ACoAAFCh8yIB8k8v0fVVR1wqWrd7WCNb0JwzN6E**

**https://github.com/kishore-c-bit/Bus-Booking-app**

**Alright, buckle up, because I’m going to nerd out hard on this Django ecosystem for you. We’re going full “under-the-hood” mode, so you’ll know exactly what every file does when you create a Django project. I’ll also cover the databases Django can use, how they’re configured, and the philosophy behind Django’s structure. Let’s go layer by layer.**

**1. Starting a Django Project**

**When you run:**

**django-admin startproject myproject**

**Django creates a folder structure like this:**

**myproject/**

**manage.py**

**myproject/**

**\_\_init\_\_.py**

**settings.py**

**urls.py**

**asgi.py**

**wsgi.py**

**1.1 manage.py**

* **This is your command-line utility for interacting with your Django project.**
* **Think of it as your “remote control” for Django.**
* **You use it to:**
  + **Run the server: python manage.py runserver**
  + **Make migrations: python manage.py makemigrations**
  + **Apply migrations: python manage.py migrate**
  + **Open shell: python manage.py shell**
  + **Create apps: python manage.py startapp appname**

**Behind the scenes: It sets up the Python path, loads the settings, and calls Django’s command-line machinery.**

**1.2 The inner myproject/ folder**

**This is the actual Python package for your project. It contains the settings, URLs, and ASGI/WSGI hooks.**

**1.2.1 \_\_init\_\_.py**

* **Marks this folder as a Python package.**
* **Usually empty. It’s just there to let Python treat this folder like a module.**

**1.2.2 settings.py**

* **The brain of your project.**
* **Contains all configurations.**
* **Key parts:**

**Basics**

**BASE\_DIR = Path(\_\_file\_\_).resolve().parent.parent**

**SECRET\_KEY = "your-secret-key"**

**DEBUG = True**

**ALLOWED\_HOSTS = []**

* + **BASE\_DIR: Root folder of your project.**
  + **SECRET\_KEY: Secret for cryptography (never share this in production!).**
  + **DEBUG: Dev mode toggle.**
  + **ALLOWED\_HOSTS: Hosts/domains your app can serve.**

**Installed apps**

**INSTALLED\_APPS = [**

**'django.contrib.admin',**

**'django.contrib.auth',**

**'django.contrib.contenttypes',**

**'django.contrib.sessions',**

**'django.contrib.messages',**

**'django.contrib.staticfiles',**

**]**

* + **These are Django’s built-in apps.**
  + **Each one adds features:**
    - **auth: user authentication system**
    - **sessions: session management**
    - **admin: admin panel**
    - **staticfiles: serves CSS/JS**
    - **contenttypes: allows dynamic model relationships**

**Middleware**

**MIDDLEWARE = [**

**'django.middleware.security.SecurityMiddleware',**

**'django.contrib.sessions.middleware.SessionMiddleware',**

**'django.middleware.common.CommonMiddleware',**

**'django.middleware.csrf.CsrfViewMiddleware',**

**'django.contrib.auth.middleware.AuthenticationMiddleware',**

**'django.contrib.messages.middleware.MessageMiddleware',**

**'django.middleware.clickjacking.XFrameOptionsMiddleware',**

**]**

* + **Middleware = layers that wrap every request/response.**
  + **Example: CsrfViewMiddleware protects against cross-site request forgery.**

**Database**

**DATABASES = {**

**'default': {**

**'ENGINE': 'django.db.backends.sqlite3',**

**'NAME': BASE\_DIR / 'db.sqlite3',**

**}**

**}**

* + **Django supports multiple databases:**
    - **sqlite3 (default, file-based, zero config, dev-friendly)**
    - **postgresql**
    - **mysql**
    - **oracle**
  + **You can even use multiple databases in one project.**

**Templates**

**TEMPLATES = [**

**{**

**'BACKEND': 'django.template.backends.django.DjangoTemplates',**

**'DIRS': [BASE\_DIR / 'templates'],**

**'APP\_DIRS': True,**

**'OPTIONS': {...},**

**}**

**]**

* + **Configures Django’s template engine.**
  + **DIRS: folder for project-wide templates.**
  + **APP\_DIRS=True: Django also looks in each app’s templates/ folder.**

**Static files**

**STATIC\_URL = '/static/'**

* + **URL prefix for serving CSS/JS/images.**

**1.2.3 urls.py**

* **Maps URLs to views (routes to functions/classes that return responses).**
* **Default file:**

**from django.urls import path**

**from django.contrib import admin**

**urlpatterns = [**

**path('admin/', admin.site.urls),**

**]**

* **Think of it as Django’s traffic controller.**

**1.2.4 asgi.py & wsgi.py**

* **Interfaces between Django and web servers.**
* **wsgi.py = traditional web servers (Gunicorn, uWSGI)**
* **asgi.py = modern async servers (Daphne, Uvicorn)**
* **They load Django settings and expose callable objects:**
* **application = get\_wsgi\_application()**

**2. Apps: Where the real magic happens**

**When you run:**

**python manage.py startapp blog**

**You get:**

**blog/**

**\_\_init\_\_.py**

**admin.py**

**apps.py**

**models.py**

**tests.py**

**views.py**

**migrations/**

**2.1 models.py**

* **Defines database structure (like tables in SQL).**
* **Example:**

**class Post(models.Model):**

**title = models.CharField(max\_length=100)**

**content = models.TextField()**

**published = models.DateTimeField(auto\_now\_add=True)**

* **Django automatically generates SQL and migrations from these.**

**2.2 admin.py**

* **Registers models to Django admin.**
* **Example:**

**from django.contrib import admin**

**from .models import Post**

**admin.site.register(Post)**

* **Makes your models editable in the default admin UI.**

**2.3 views.py**

* **Handles logic for what happens when a URL is accessed.**
* **Returns HttpResponse or renders templates.**
* **Example:**

**from django.shortcuts import render**

**from .models import Post**

**def home(request):**

**posts = Post.objects.all()**

**return render(request, 'home.html', {'posts': posts})**

**2.4 urls.py (inside app)**

* **Optional, but good practice to make app-specific routes:**

**from django.urls import path**

**from . import views**

**urlpatterns = [**

**path('', views.home, name='home'),**

**]**

* **Then include this in project’s urls.py:**

**path('', include('blog.urls'))**

**2.5 apps.py**

* **Registers app config.**
* **Usually default:**

**class BlogConfig(AppConfig):**

**default\_auto\_field = 'django.db.models.BigAutoField'**

**name = 'blog'**

* **Useful if you want custom app name, label, or ready hooks.**

**2.6 migrations/**

* **Auto-generated migration files for models.**
* **Each file tracks changes to your DB schema.**
* **Example: 0001\_initial.py for first model creation.**
* **\_\_init\_\_.py makes it a package.**

**2.7 tests.py**

* **Place to write unit tests.**
* **Django comes with its own test framework (unittest-based).**

**3. Databases in Django**

**Django abstracts DBs so you can switch engines with minimal effort. Supported:**

| **Engine** | **Use Case** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **SQLite** | **Dev, prototyping** | **No setup, file-based** | **Not for high concurrency** |
| **PostgreSQL** | **Production, advanced features** | **Transactions, JSON fields, indexing** | **Needs server** |
| **MySQL** | **Production** | **Fast for reads** | **Less feature-rich than Postgres** |
| **Oracle** | **Enterprise** | **Mature, reliable** | **Heavy, paid** |

**You configure DB in settings.py, and then Django’s ORM handles all SQL behind the scenes.**

**4. Django Philosophy**

1. **“Batteries included” – everything you need is in the core (auth, admin, forms, sessions, ORM, middleware, templating).**
2. **MTV (Model-Template-View) pattern – similar to MVC:**
   * **Model = DB schema**
   * **Template = HTML**
   * **View = Logic**
3. **DRY (Don’t Repeat Yourself) – reusable apps, centralized settings.**
4. **Migrations – automated DB schema versioning.**
5. **Scalable – supports multiple apps, databases, and async workflows.**

**5. Other Files You Might See**

* **\_\_pycache\_\_/ – compiled Python files.**
* **.env – environment variables (SECRET\_KEY, DB passwords)**
* **requirements.txt – pip dependencies**
* **static/ – CSS/JS/images**
* **media/ – uploaded files**
* **templates/ – project-wide templates**

**6. How a Request Flows**

1. **Client hits a URL.**
2. **Django matches URL in urls.py.**
3. **Middleware wraps request (security, sessions, etc.).**
4. **View function is called.**
5. **View interacts with Model if needed.**
6. **Template renders response.**
7. **Middleware wraps response.**
8. **Response is sent back to client.**

**If you want, I can draw a full diagram of the Django ecosystem showing files, flow, DB, middleware, apps, templates, so you can visualize it all in one shot.**

**Do you want me to make that diagram next?**

**🔹 1. What is Django REST Fra**

**mework (DRF)?**

* **Django** → A Python web framework for building web applications (mainly server-rendered HTML).
* **Django REST Framework (DRF)** → A toolkit built on Django to **easily create REST APIs** (JSON/XML instead of HTML).
* With DRF, you can expose your database and business logic as **RESTful endpoints** that can be consumed by web apps, mobile apps, or other systems.

**🔹 2. Setup**

First, install Django + DRF:

pip install django djangorestframework

**1. Purpose**

* **Django** → Full-stack web framework mainly used for building **websites** (with HTML templates, forms, admin panel, etc.).
* **DRF** → A library built **on top of Django** to make it easy to build **REST APIs** (data in JSON/XML instead of HTML).

**🔹 2. Output**

* **Django** → By default returns **HTML** (using templates).
* **DRF** → Returns **JSON/XML** (consumable by mobile apps, React, Angular, etc.).

**🔹 3. Views**

* **Django views** → HttpResponse, render() → return HTML pages.
* **DRF views** → APIView, ViewSet → return Response objects (JSON).

Example:

# Django

from django.shortcuts import render

def home(request):

return render(request, 'index.html')

# DRF

from rest\_framework.views import APIView

from rest\_framework.response import Response

class Hello(APIView):

def get(self, request):

return Response({"message": "Hello, API!"})

**🔹 4. Data Handling**

* **Django** → Uses **Forms** to validate & handle user input.
* **DRF** → Uses **Serializers** to convert models ↔ JSON.

**🔹 5. Authentication**

* **Django** → Session-based authentication (good for websites).
* **DRF** → Supports session auth + token-based auth + JWT (good for APIs, mobile apps, 3rd party integration).

**🔹 6. Use Cases**

* **Django** → Blogs, e-commerce sites, dashboards, traditional web apps.
* **DRF** → Mobile app backends, microservices, exposing data for frontend (React, Angular, Vue).

**🔹 7. Admin**

* **Django** → Comes with **Admin panel** out-of-the-box.
* **DRF** → Comes with a **Browsable API** for testing APIs easily.

✅ **In short**:

* Use **Django** if you’re building a **website** with pages.
* Use **DRF** if you’re building an **API** (backend for mobile apps or SPAs).
* You can (and often do) use them **together**: Django for web + DRF for API.

**🔹 1. End Users**

* **Django (traditional)** → End user is a **customer in a browser**.
* **DRF (API-based)** → End user is a **frontend app (React, mobile, Vue, Angular, etc.)**, not directly the human.

**🔹 2. Output**

* **Django website** → Renders **HTML pages** (home, products, cart, checkout).
* **DRF API** → Returns **JSON responses** (structured data).

Api

It stand for application programming language

It acts as two way communication bridge between two applications

Rest api

Representational state transfer

It organizes how web application talk to each other ,separating front end and back end

**REST = Representational State Transfer**

It’s an **architectural style** (a set of rules/guidelines) for designing **web services** (APIs).  
DRF follows these rules to make building APIs easier.

**🔹 Key Principles of REST (and how DRF uses them)**

1. **Client-Server Architecture**
   * The **client** (React, mobile app, etc.) and **server** (Django + DRF) are separate.
   * Server only provides **data (JSON)**, client handles the **UI**.
2. **Stateless**
   * Each API request is independent.
   * Server doesn’t remember client’s previous request.
   * Example: Every request must include authentication (like JWT token).
3. **Uniform Interface (Standard URLs & Methods)**
   * REST uses HTTP methods with clear meaning:
     + **GET** → Fetch data
     + **POST** → Create new data
     + **PUT/PATCH** → Update data
     + **DELETE** → Remove data

Example in DRF (Products API):

* + GET /api/products/ → list all products
  + POST /api/products/ → create new product
  + GET /api/products/1/ → get product with ID 1
  + PUT /api/products/1/ → update product 1
  + DELETE /api/products/1/ → delete product 1

1. **Resource-Based**
   * Everything is treated as a **resource** (user, product, order, etc.).
   * Each resource has a **unique URL** (endpoint).

Example:

* + /api/users/ → users resource
  + /api/products/ → products resource
  + /api/orders/ → orders resource

1. **Representation**
   * A resource can have different representations (JSON, XML, etc.).
   * DRF by default gives JSON, but can also give XML, browsable API (HTML).
2. **Stateless Communication**
   * Client must send all required info (auth token, params) with each request.
   * Server doesn’t keep session like Django normally does with HTML sites.

In **Django REST Framework (DRF)**, a **serializer** is like a **translator** between Django models (Python objects) and JSON (or other formats like XML).

**🔹 Why do we need Serializers?**

* Django models = Python objects (not directly readable by frontend / API clients).
* APIs need data in JSON format.
* Serializers convert:
  + **Model → JSON** (for sending data in API response).
  + **JSON → Model** (for creating/updating objects from API requests).

**🔹 Types of Serializers**

1. **ModelSerializer**
   * Easiest way. Connects directly to a Django model.
   * Automatically creates fields from model.
2. **Serializer (Manual)**
   * More flexible, define each field manually.

Great question 👍 — you already know some **core DRF features** like **serializers, viewsets, permissions, filters, JWT**. Let’s expand on that.

Django REST Framework has a **lot of powerful tools**, but some are considered **core must-know features** when building real APIs:

**🔹 Core Features of DRF**

**1. Serializers**

* Convert Django models/objects → JSON (and back).
* Types:
  + ModelSerializer (most common).
  + Nested serializers (Hotel → Rooms → Reviews).
  + Custom serializers (for complex logic like booking availability).

**2. ViewSets & Routers**

* Simplify CRUD operations.
* Instead of writing multiple views (list, create, update), you define a **ViewSet** and register it with a **Router**, which auto-generates endpoints.

**3. Authentication**

* Who is making the request?
* Common methods:
  + Session Authentication (default Django).
  + Token Authentication.
  + **JWT Authentication** (djangorestframework-simplejwt).
  + OAuth2 (with django-oauth-toolkit).

**4. Permissions**

* What can the user do?
* Examples:
  + IsAuthenticated → Only logged-in users.
  + IsAdminUser → Only admins.
  + **Custom permissions** (e.g., only hotel owners can edit their own hotel).

**5. Filtering & Pagination**

* **Filtering** → django-filter for queries like:
* /api/rooms/?location=mumbai&price\_\_lte=5000
* **Pagination** → Break large results into chunks (PageNumber, LimitOffset, Cursor-based).

**6. Throttling & Rate Limiting**

* Limit requests per user/IP to prevent abuse.
* Example:
  + AnonRateThrottle → Limit for anonymous users.
  + UserRateThrottle → Limit per logged-in user.

**7. Versioning**

* Maintain multiple API versions:
* /api/v1/hotels/
* /api/v2/hotels/

**8. Exception Handling**

* Custom error responses for bad requests.
* Example:
* {
* "error": "Room not available for these dates"
* }

**9. Browsable API**

* DRF comes with a **web UI** for exploring APIs.
* Very useful for testing during development.

**10. Content Negotiation**

* Serve different formats: JSON (default), XML, etc.
* DRF decides based on Accept header.

**🔹 Advanced but Important Features**

* **Nested Routers** → For APIs like /hotels/{id}/rooms/.
* **Hyperlinked Serializers** → Use hyperlinks instead of IDs in responses.
* **Generic Views** → ListCreateAPIView, RetrieveUpdateDestroyAPIView.
* **Schema & Documentation** → Auto-generate API docs using drf-yasg or drf-spectacular (Swagger/OpenAPI).
* **Signals & Hooks** → Trigger actions after bookings/payments.

👉 So in summary, **core DRF features** you’ll use in most projects are:  
✅ Serializers  
✅ ViewSets & Routers  
✅ Authentication (esp. JWT)  
✅ Permissions  
✅ Filtering & Pagination  
✅ Throttling  
✅ Exception Handling

Great choice 👌 Building a **Travel & Hotel Booking API** with DRF is perfect — it’s realistic, heavily backend-driven, and uses many DRF concepts. Let’s make a **step-by-step roadmap** to get you started.

**📌 Travel & Hotel Booking API (DRF Project Roadmap)**

**🔹 Step 1: Project Setup**

1. Create Django project + app
2. django-admin startproject hotel\_booking
3. cd hotel\_booking
4. python manage.py startapp bookings
5. Install DRF + JWT
6. pip install djangorestframework djangorestframework-simplejwt django-filter
7. Add to settings.py:
8. INSTALLED\_APPS = [
9. 'rest\_framework',
10. 'rest\_framework\_simplejwt',
11. 'django\_filters',
12. 'bookings',
13. 'django.contrib.auth',
14. ]

**🔹 Step 2: User Roles (Admin, Hotel Owner, Traveler)**

* Extend Django’s User model with a **role field** (admin, owner, traveler).
* Example:
* from django.contrib.auth.models import AbstractUser
* from django.db import models
* class User(AbstractUser):
* ROLE\_CHOICES = (
* ('admin', 'Admin'),
* ('owner', 'Hotel Owner'),
* ('traveler', 'Traveler'),
* )
* role = models.CharField(max\_length=20, choices=ROLE\_CHOICES, default='traveler')

**🔹 Step 3: Models**

1. **Hotel**
   * Owner (FK to User)
   * Name, location, description, rating
2. **Room**
   * Hotel (FK)
   * Room type, price, capacity, availability flag
3. **Booking**
   * Traveler (FK to User)
   * Room (FK)
   * Check-in, Check-out dates
   * Status: *Pending / Confirmed / Cancelled*
4. **Review**
   * Traveler (FK)
   * Hotel (FK)
   * Rating, comment

**🔹 Step 4: Serializers**

* **Nested Serializer Example** (Hotel → Rooms → Reviews).
* Use DRF ModelSerializer with relationships.

**🔹 Step 5: Views & Endpoints**

Use **ViewSets + Routers**. Example routes:

POST /api/auth/register/ → Register (traveler/owner)

POST /api/auth/login/ → JWT login

GET /api/hotels/ → List all hotels

POST /api/hotels/ → Add hotel (owner only)

GET /api/hotels/{id}/ → Hotel details + rooms + reviews

POST /api/hotels/{id}/rooms/ → Add room (owner only)

GET /api/rooms/?location=mumbai → Filter rooms by location

POST /api/bookings/ → Traveler books a room

GET /api/bookings/my/ → Traveler’s bookings

POST /api/reviews/ → Add review (only if traveler booked)

GET /api/admin/reports/ → Revenue, occupancy (admin only)

**🔹 Step 6: DRF Concepts to Use**

* **JWT Authentication** → Secure endpoints.
* **Permissions** → Owners manage only their hotels, travelers book rooms, admins see reports.
* **Filtering & Pagination** → django-filter for price, location, capacity.
* **Custom Business Logic** → Check availability before booking.
* **Nested Relationships** → Hotel → Rooms, Reviews.

**🔹 Step 7: Extra Features (Optional, Advanced)**

* Cancel booking with refund logic.
* Revenue report for Admin → sum of all confirmed bookings.
* Occupancy report → % of booked vs available rooms.
* Image upload for hotels/rooms.

⚡ This roadmap ensures you’ll use **core DRF features (serializers, viewsets, permissions, filters, JWT)** while building something practical.

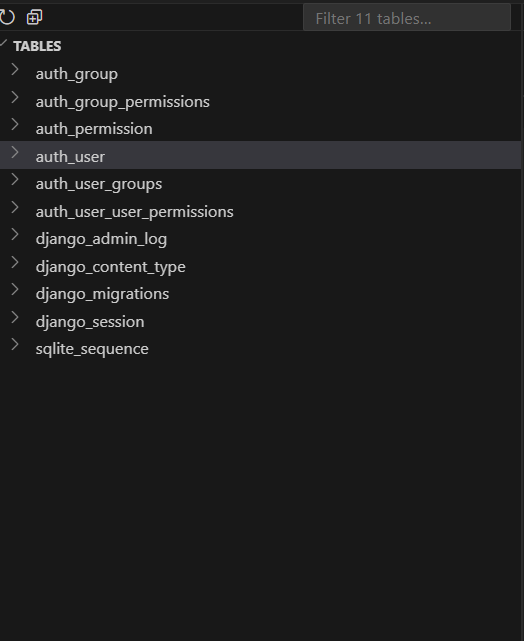
👉 Do you want me to **start with the models.py code (Hotel, Room, Booking, Review, User)** so you can begin coding right away?

📂 hotel\_booking → Django project  
📂 bookings → Django app

* If we migrate before creating any custome user then djnago by default create auth\_user table . In that case, it will keep using auth\_user, and won’t create bookings\_user.

👉 That’s why Django docs strongly recommend deciding the custom user model at the very start.

By default database will be :



Here we use AUTH\_USER\_MODEL = 'bookings.User' why we do this

By default, Django uses the built-in User model from **django.contrib.auth** (with fields like username, email, password, is\_staff, etc.).

But in project, I create **own custom User model**:

This adds a **new field (role)** to classify whether the user is an Admin, Hotel Owner, or Traveler.

So we must tell Django:

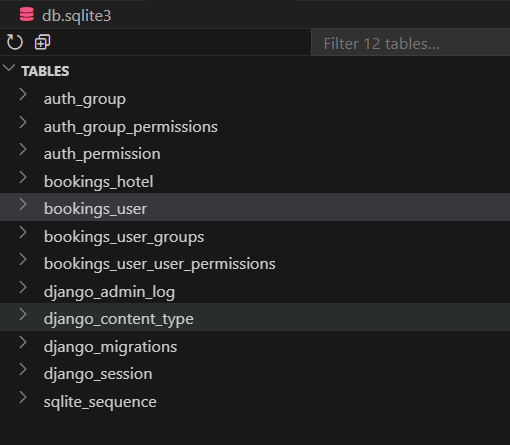
"Don’t use the default auth.User. Instead, use my custom user model (bookings.User)."

That’s what AUTH\_USER\_MODEL = 'bookings.User' does.

**Why is it important?**

1. **Flexibility**  
   You can add extra fields (role, phone\_number, profile\_picture, etc.) directly to your user model.
2. **Relationships**  
   Whenever you create a ForeignKey to the user model (like owner = models.ForeignKey(User, ...) in Hotel), Django knows which user model to use.
3. **Best Practice**  
   Django docs recommend defining a custom user model **at the start of the project**.  
   (Changing it later is painful because migrations already created the default auth\_user table).

After adding AUTH\_USER\_MODEL = 'bookings.User' in setting.py it will create database like



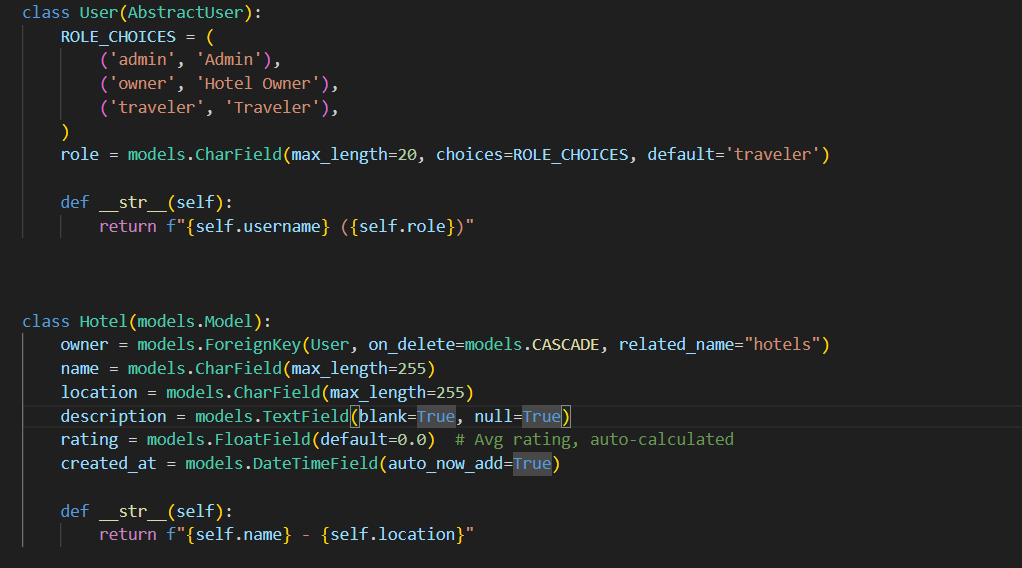
from django.db import models

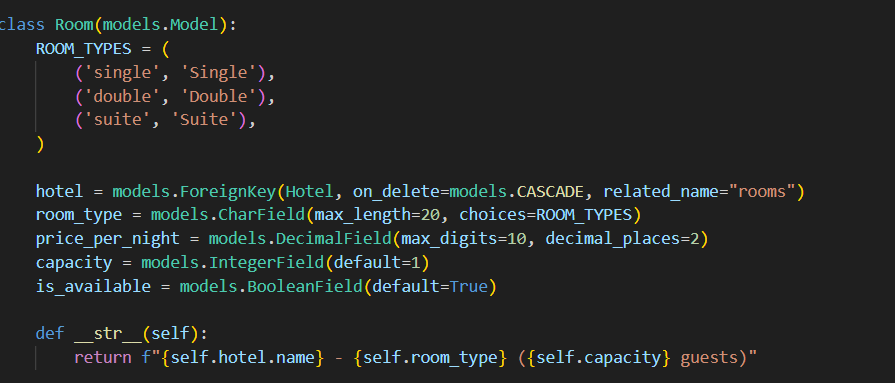
# Create your models here.

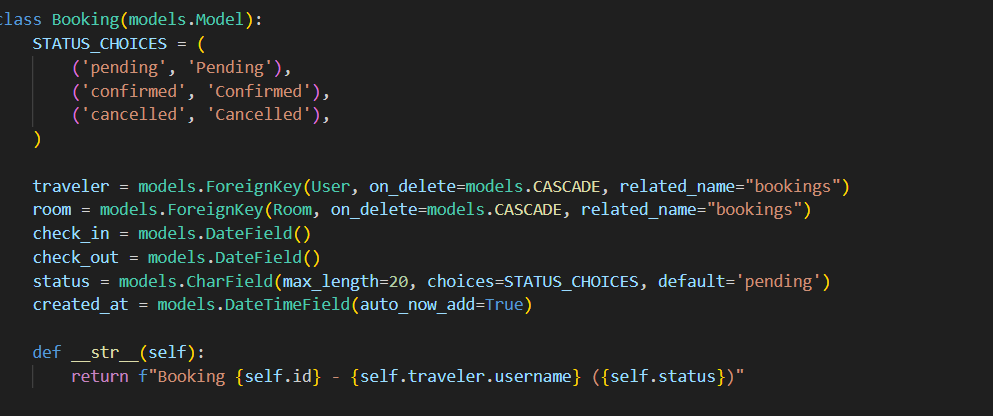
from django.db import models

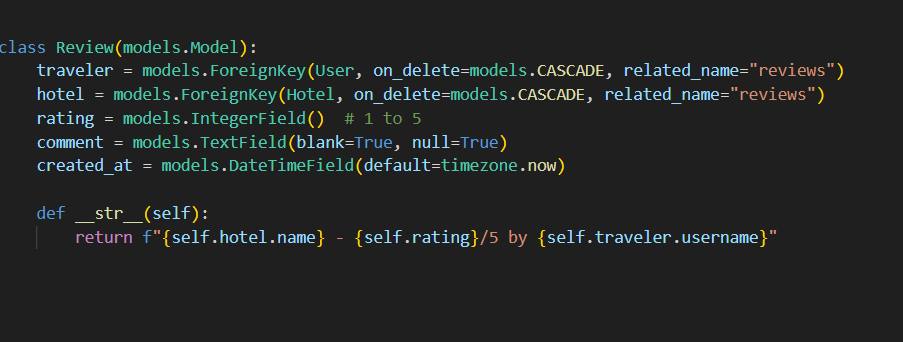
from django.contrib.auth.models import AbstractUser

from django.utils import timezone









Now if we create super user then it will by default be an traveler not and admin . because right now if you run createsuperuser, Django will create the account but your **custom role field** defaults to "traveler". That means your superuser will technically look like a traveler, even though it has Django admin privileges.

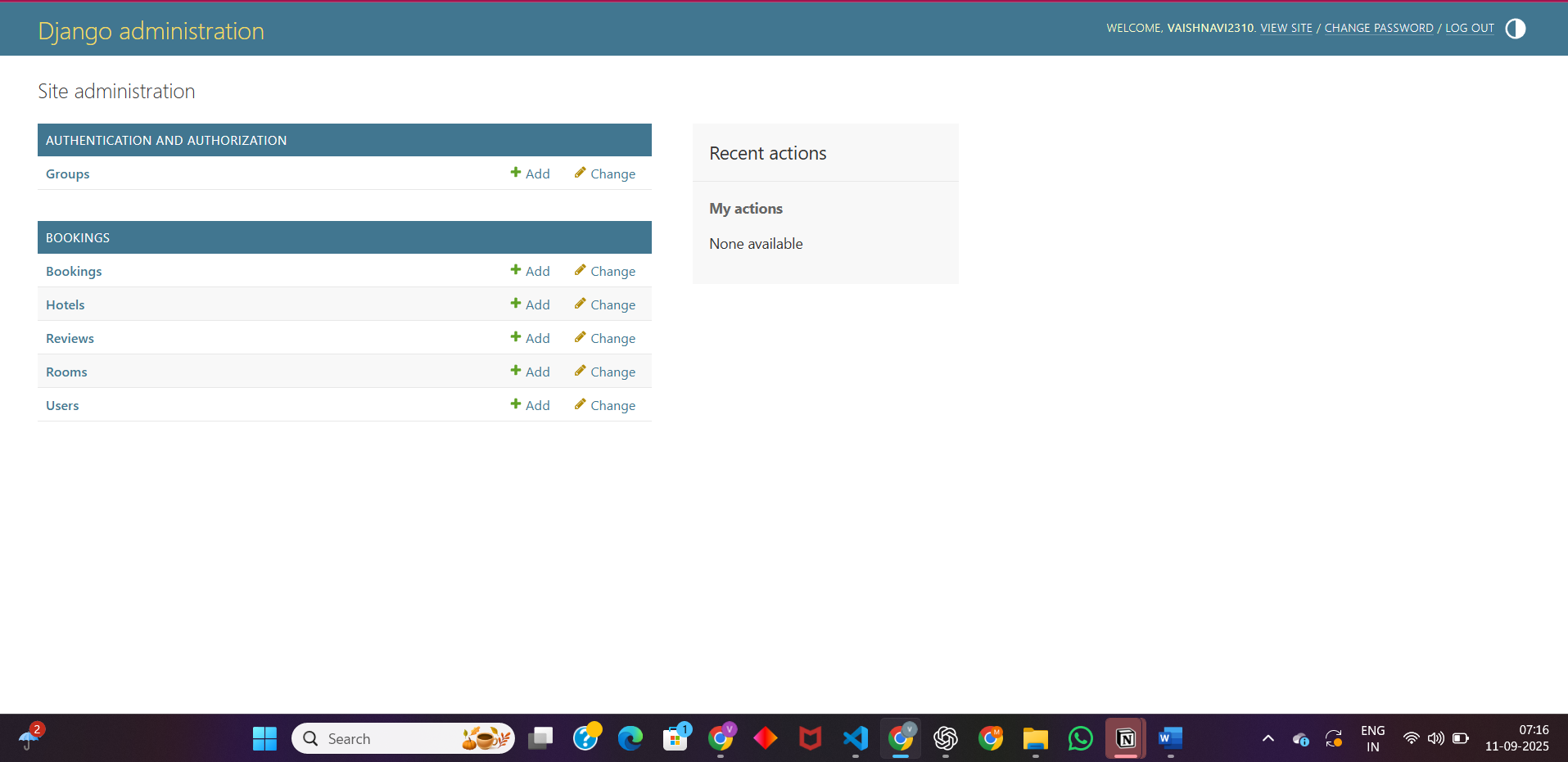
We can fix this by overriding create\_superuser in a **custom UserManager**.

* **Use AbstractUser** → when you want to **add fields** but keep Django’s default User behavior.
* **Use BaseUserManager** → when you want to **customize how users are created** (especially for create\_user and create\_superuser).

💡 Most projects use **both together**:

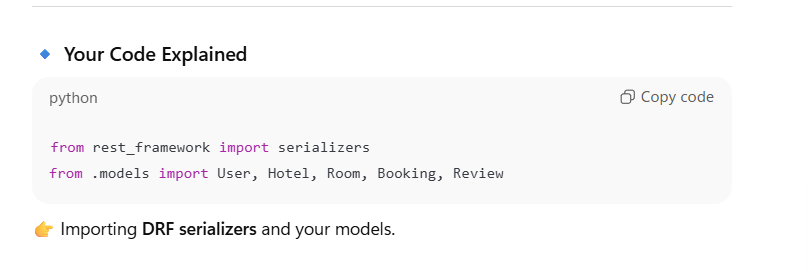
* AbstractUser → defines the User schema.
* BaseUserManager → controls how new users/superusers are created.

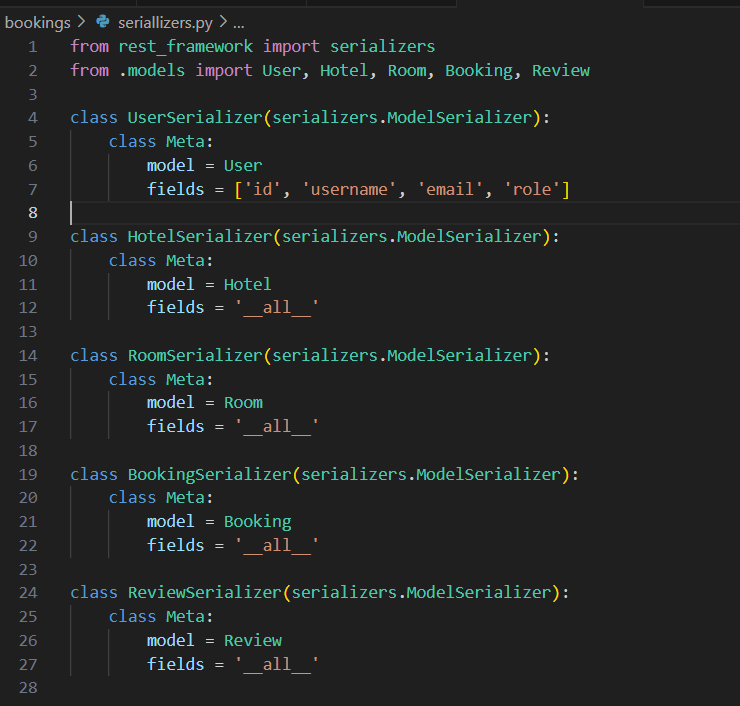
If you created the superuser and logged into the **Django Admin**, but you don’t see your models (Hotel, Room, Booking, Review, etc.), it’s because you haven’t **registered them in admin.py** yet.



**What is a Serializer in DRF?**

A **serializer** converts your **Django model objects** into **JSON** (so APIs can send them as responses) and also validates JSON data when **creating/updating** objects from API requests.





**🔹 class UserSerializer(serializers.ModelSerializer):**

This defines a **serializer class**.

* UserSerializer → your custom serializer for the User model.
* serializers.ModelSerializer → a special DRF class that automatically creates serializer fields from a Django model.

So instead of writing each field manually (like id = serializers.IntegerField()), DRF will generate them from your **User model**.

**🔹 class Meta:**

The Meta **inner class** is where you tell DRF **which model** the serializer is for, and **which fields** to include.

* model = User  
  👉 This says: "This serializer is based on the User model."
* fields = ['id', 'username', 'email', 'role']  
  👉 This says: "Only include these fields when converting model ↔ JSON."

So **Meta is just configuration for the serializer**.

**What is a ViewSet?**

A **ViewSet** is like a controller for a model in DRF.  
Instead of writing multiple views (list, retrieve, create, update, delete) manually, DRF’s ModelViewSet gives you all **CRUD operations** automatically.



* viewsets.ModelViewSet → a DRF class that provides **list, retrieve, create, update, delete** out of the box.
* queryset = User.objects.all() → tells DRF: “Work with all User objects in the database.”
* serializer\_class = UserSerializer → tells DRF: “Use UserSerializer to convert model ↔ JSON.”

👉 Now, without writing extra code, you get endpoints like:

* GET /users/ → List all users
* GET /users/1/ → Get a single user
* POST /users/ → Create a new user
* PUT /users/1/ → Update user
* DELETE /users/1/ → Delete user

**🔹 In Django (Normal Web App)**

* Views return **HTML responses** (usually rendered from templates).
* Used for traditional web applications (not APIs).
* Example:

👉 Here:

* hotel\_list view takes a **request**.
* Fetches hotels from DB.
* Passes them to **template (HTML page)**.
* Returns an **HttpResponse (HTML)**.

**🔹 In DRF (API Development)**

* Views return **JSON / API responses** (not HTML).
* Used for **REST APIs**.
* Instead of writing everything manually, you use **APIView / ViewSets**.

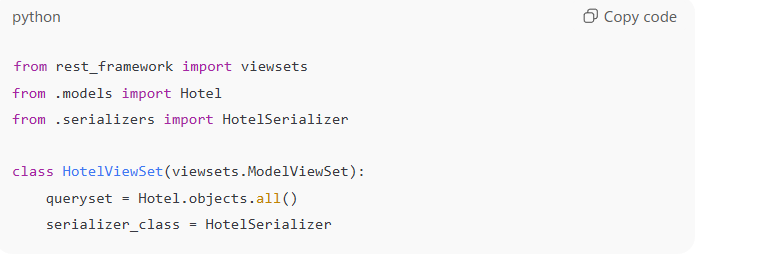
Example with APIView:

S👉 Here:

* Returns **JSON data** (not HTML).
* Uses **serializers** to convert model objects → JSON.

**🔹 Using ViewSets (shortcut in DRF)**

Instead of writing get, post, etc. manually, DRF gives us **ViewSets**:

👉 This one class provides:

* GET /hotels/ → list
* POST /hotels/ → create
* GET /hotels/{id}/ → retrieve
* PUT /hotels/{id}/ → update
* DELETE /hotels/{id}/ → delete