Contents

[🔧 Step 1: Define Your Models (Database Structure) 1](#_Toc195661546)

[🔐 Step 2: Authentication (Login, Signup, Roles) 3](#_Toc195661547)

[🔌 Step 3: API Endpoints (Django REST Framework) 3](#_Toc195661548)

[⚡ Step 4: Real-time Chat (Django Channels) 4](#_Toc195661549)

[🧠 Step 5: Matching System 4](#_Toc195661550)

[🧰 Step 6: Admin Interface 5](#_Toc195661551)

# 🔧 Step 1: Define Your Models (Database Structure)

Models define **what kind of data** your app stores and how different pieces of data **relate** to each other.

**🧍‍♂️ 1. CustomUser Model**

We extend Django’s built-in user to support **roles** (volunteer or regular user) and add fields like expertise.

from django.contrib.auth.models import AbstractUser

from django.db import models

class CustomUser(AbstractUser):

is\_volunteer = models.BooleanField(default=False)

expertise = models.TextField(blank=True, null=True)

✅ **Why?**

* is\_volunteer: lets you distinguish between user types.
* expertise: helps in matching users to the right volunteers.

**Extra Tip:** You’ll need to tell Django to use this custom user in settings.py:

AUTH\_USER\_MODEL = 'yourapp.CustomUser'

**💬 2. ChatSession Model**

Represents a **1-on-1 conversation** between a user and a volunteer.

class ChatSession(models.Model):

user = models.ForeignKey(CustomUser, on\_delete=models.CASCADE, related\_name='user\_sessions')

volunteer = models.ForeignKey(CustomUser, on\_delete=models.CASCADE, related\_name='volunteer\_sessions')

started\_at = models.DateTimeField(auto\_now\_add=True)

ended\_at = models.DateTimeField(null=True, blank=True)

active = models.BooleanField(default=True)

✅ **Why?**

* Keeps track of who’s chatting with whom.
* Can be used to load chat history or monitor active sessions.

**✉️ 3. Message Model**

Stores each individual **chat message** sent during a session.

class Message(models.Model):

session = models.ForeignKey(ChatSession, on\_delete=models.CASCADE)

sender = models.ForeignKey(CustomUser, on\_delete=models.CASCADE)

content = models.TextField()

timestamp = models.DateTimeField(auto\_now\_add=True)

✅ **Why?**

* Makes it easy to show messages in the frontend.
* Timestamps help order messages.

**🕒 4. VolunteerAvailability Model**

Tracks whether a volunteer is **available** to take on a chat.

class VolunteerAvailability(models.Model):

volunteer = models.OneToOneField(CustomUser, on\_delete=models.CASCADE)

is\_available = models.BooleanField(default=False)

last\_updated = models.DateTimeField(auto\_now=True)

✅ **Why?**

* You need to know which volunteers are available **in real-time** for matching.
* Prevents overloading a single volunteer.

# 🔐 Step 2: Authentication (Login, Signup, Roles)

You’ll use Django’s built-in auth system, but since we have **custom roles**, we extend it with CustomUser above.

**Tools you might use:**

* For web: django.contrib.auth.views (login/logout views)
* For APIs: djoser, SimpleJWT (if using DRF)

**Signup View Logic:**

* When someone signs up, ask if they are a **volunteer or user**.
* If volunteer, collect extra data like expertise.

**Login Logic:**

* Basic username/password login.
* After login, redirect based on role (volunteer dashboard vs user chat).

# 🔌 Step 3: API Endpoints (Django REST Framework)

**Django REST Framework (DRF)** is perfect for building your API.

You’ll need APIs like:

| **Endpoint** | **Description** |
| --- | --- |
| /api/start-session/ | Start a new chat session |
| /api/send-message/ | Send a message |
| /api/get-messages/?session\_id=123 | Get messages for a session |
| /api/available-volunteers/ | Show available volunteers |
| /api/set-availability/ | Volunteer sets availability |

✅ **Why?**

* Frontend needs to talk to the backend through API calls.
* Keeps your app fast and clean.

# ⚡ Step 4: Real-time Chat (Django Channels)

**Normal Django is synchronous**, but for live chat you need **WebSockets** (real-time).

Use **Django Channels**:

* Runs alongside Django.
* Handles real-time communication like live chat or notifications.

**How it works:**

1. User connects to WebSocket: ws://yoursite/ws/chat/session\_id/
2. Django Channels accepts the connection.
3. Messages are sent instantly between users via the socket.

You’ll create:

* A routing.py file for WebSocket URLs.
* A consumers.py file to handle messages like Django views.

# 🧠 Step 5: Matching System

When a user wants to talk:

1. Backend checks VolunteerAvailability where is\_available=True.
2. Filters volunteers by matching expertise if possible.
3. Pairs the user with a volunteer.
4. Creates a new ChatSession.

You can also add:

* Load balancing (don’t overload a single volunteer).
* Timeout system if nobody responds in 1 minute.

# 🧰 Step 6: Admin Interface

Django’s admin is super useful for:

* Managing users, volunteers, and messages.
* Viewing active sessions.
* Blocking/reporting users if needed.

Customize the admin to:

* Show only key info.
* Make it easy to search/filter.

**Want to Build it Step-by-Step?**

We can do it piece by piece:

* First: Set up the CustomUser model and user signup/login.
* Then: Set up ChatSession + Message.
* Then: Add API endpoints.
* Then: Real-time chat.
* Then: Matchmaking logic.

Just tell me where you want to begin coding, boss man 🔥