

DJANGO GIRLS

Building a Real-Time Video Chat Application with Django



django

Introduction

Ever jumped on a Zoom call or FaceTime and wondered about the magic happening behind the scenes? Get ready to uncover those secrets! In this hands-on workshop, we'll build an exciting real-time video chat application using Django that lets people connect face-to-face and communicate instantly in virtual rooms.

What You'll Create

A fully functional video and chat application where users can:

- Join different public chatrooms
- Have one on one video calls

You can think of this workshop as having two parts, the first whereby you create a chat application to understand websockets, and the second whereby you extend it by adding a video call functionality.

What You'll Learn

1. **Websockets Explained Simply**
 - What they are and why they're perfect for real time applications
 - How they're different from regular web connections
2. **WebRTC**
 - An overview of what they are
3. **Hands-On Development**
 - Create your own WebSocket consumer
 - Set up proper routing
 - Create a WebRTC client
 - Build an interactive client interface

Prerequisites

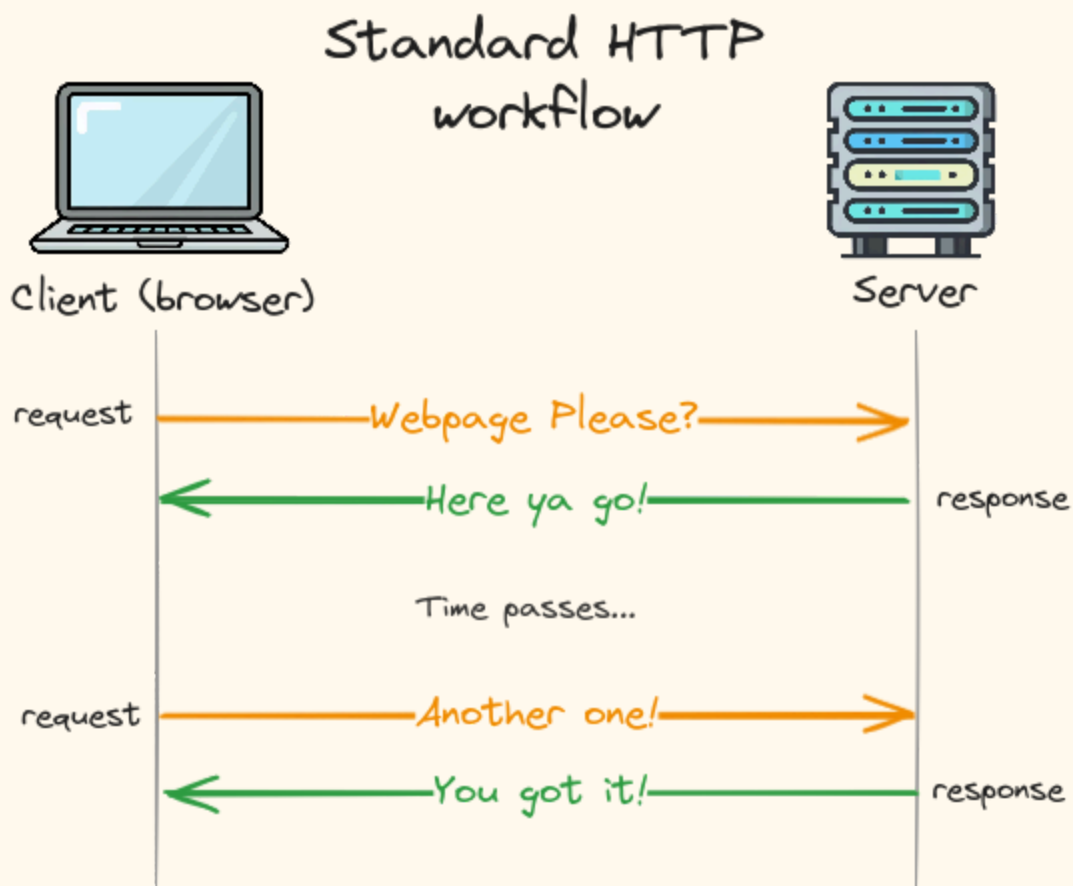
Before you start make sure you have the following installed:

- Python 3.10 and above
- Django 5

WEBSOCKETS

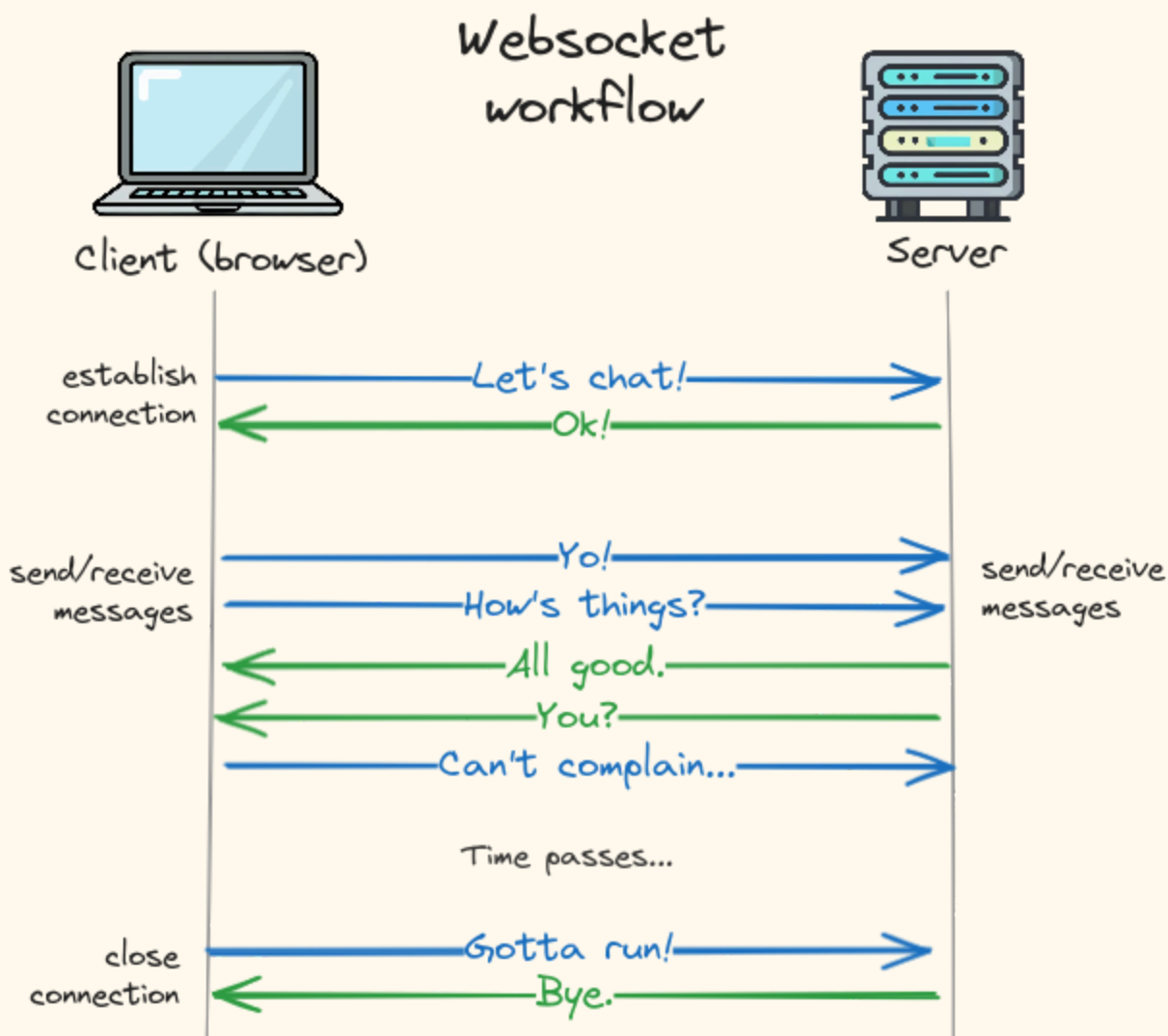
[Websockets](#) sound fancy, and anyone who has never used them will probably think of them as black magic. But, conceptually, they are quite simple!

Let's first talk about how a traditional HTTP request works. In this world you have a *client* (usually the browser), and a *server* (usually the...well, server). The *client* starts off by sending a *request* to the server (e.g. "please load this webpage"). The server then processes the request and sends a *response* to the client (e.g. "here it is!"). Once the content has been delivered to the client, the connection is closed and the request is completed.



You can think of this HTTP interaction like writing a letter to a pen pal. You send the letter, it gets delivered, your pen pal reads it, and then they send you a response. Each letter gets one and only one response, and everything happens serially.

If HTTP is like letter-writing, then websockets is like making a phone call. Instead of sending requests and getting responses, the client first *opens a connection* to the server (i.e. calls someone). Once the connection is established (call is answered), both the client and server can send messages over the channel as much as they want (the conversation). Then, at some point the channel is closed (hanging up) and the conversation is over.



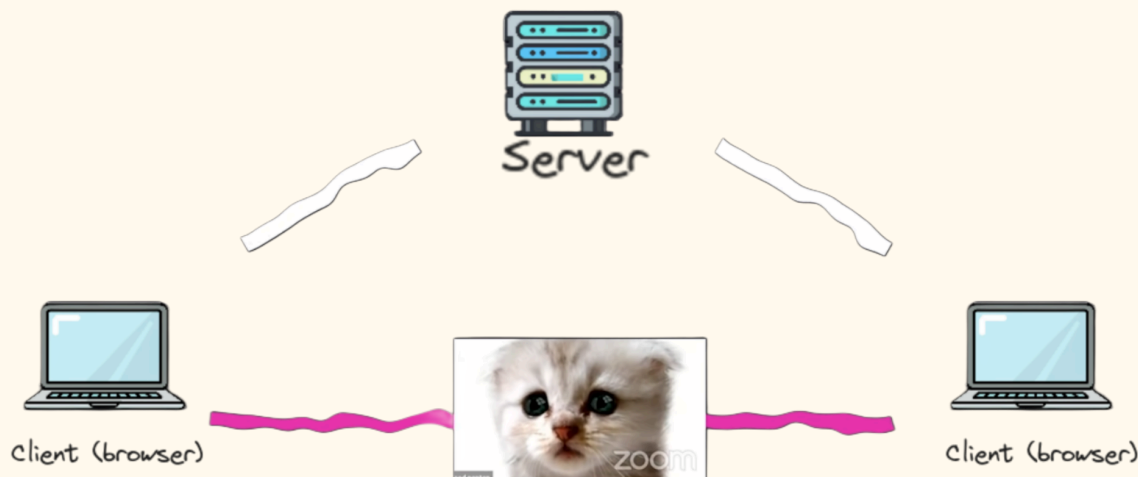
The main difference between normal HTTP requests and web sockets is the "open channel" where both client and server can send and receive messages at any time. This makes websockets a good choice when you are expecting multiple messages from the server in a row—for example in a chatting UI.

Now that we have an idea of what WebSockets are, let's discuss a little bit about WebRTC.

WebRTC

WebRTC enables direct peer-to-peer communication between browsers. Unlike WebSockets, which require a server to relay messages between clients, WebRTC allows connected devices to communicate directly with each other without an intermediary server.

It should be noted that even though WebRTC can allow devices to communicate directly, they still need a signalling server that will securely connect the two devices.



This is an overview of how WebRTC connection works:

- Lets say computer A wants to communicate with computer B via WebRTC
- Computer A will send an Offer asking for computer B to connect to them
- The offer generates a session description protocol (SDP) which will describe the peer to peer connection (video codec, timing etc)

- The SDP will be saved on the server and the server will inform computer B about the offer.
- Computer B will then accept Computer A's offer by sending an SDP answer to the server.
- The two computers will then communicate with each other directly via a peer-to-peer connection without going through the server.

So in short:

1. We will use HTTP requests for displaying regular html pages
2. We will use WebSockets for sending text messages
3. We will use WebRTC for video call

All good? Let's get coding. Please make sure you already have Python and Django installed.

Creating A Chat Application

Make sure you have Django installed in your virtual environment. If you have any difficulties, reach out to a mentor.

Go to your project folder and start by creating a new project with django:

```
django-admin startproject config .
```

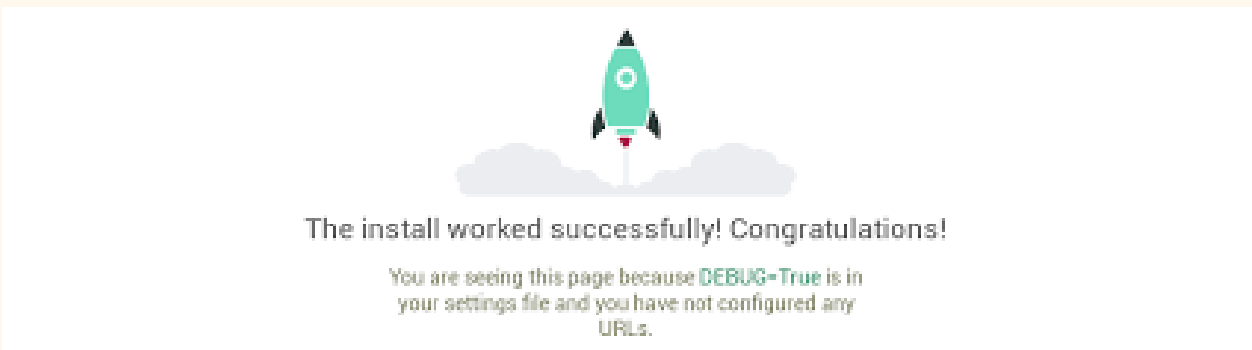
Then create an app called **chat** with:

```
django-admin startapp chat
```

Then edit the `settings.py` file in the `config` folder and activate the chat application in your project by editing the `INSTALLED_APPS` setting as follows:

```
33 INSTALLED_APPS = [  
34     # ...  
35     "chat.apps.ChatConfig",  
36 ]
```

The chat application is now active in your project. You can confirm if everything is set by running `python manage.py runserver``. You should see something like this:



Creating Our Index View:

Let's create our first view in the chat app, go to the `view.py` folder and add the following:

```
1 from django.shortcuts import render
2
3 # Create your views here.
4
5 def index(request):
6     return render(request, 'index.html')
```

We have defined a simple function called `index` and its task is to render (display) the `index.html` file. We haven't created a html file though so let's do that.

Actually, django being django, to display the page we will have to do the following:

- Create a view (which we just did)

- Add our view to urls.py
- Add our app to the main urls.py which is under the config folder (confusing I know)
- Create a static folder. This is where we will store our css files
- Create a template folder. This is where we will put our html files.

That's quite a lot of configuration to display one page.

Adding Views To URL Patterns:

Create a file called `urls.py` under the `chat` folder. Then add the following lines of code:

```
1 from django.urls import path
2 from . import views
3
4 app_name = 'chat'
5
6 urlpatterns = [
7     path('', views.index, name='index'),
8 ]
```

What we are basically doing is telling django that whenever someone visits the root url, then displays the index page. The root url is the main url without a slug (eg: 127.0.0.1:8000, www.google.com etc). URLs like 127.0.0.1:8000/chat and www.google.com/search are not root URLs as they contain slugs.

Once we add the URL pattern on the app level, we have to include it in the project level. Go to urls.py file under the config folder, and make the following changes:

```
18 from django.contrib import admin
19 from django.urls import path, include
20
21 urlpatterns = [
22     path("admin/", admin.site.urls),
23     path("", include("chat.urls", namespace="chat")),
24 ]
```

What we are doing is telling django that, 'Listen up django, we have this app called 'chat' and it has its own URL patterns that you must include in the project.' After adding these lines, django has no option but to oblige.

Adding The Static Folder

The static folder is where all static files like css, javascript, images, icons etc are located. These are files that do not change, or at least do not change often.

For our project, we will only have one static file called `styles.css`. You can download it from [here](#).

Once you've downloaded the css file, create a folder called static in the root of your project, then place the css file inside it.

Then update the settings.py file under the config folder as follows:

```
122 STATIC_URL = "static/"
123
124 STATICFILES_DIRS = [
125     BASE_DIR / "static",
126 ]
```

We are telling django that the static folder of this project is in the root directory.

Adding The Templates Folder

The templates folder is where we place our html files. They are called templates because they contain placeholders and template tags that Django can dynamically fill with data from our views and models.

Templates allow us to create reusable HTML layouts and inject dynamic content while keeping our presentation logic separate from our business logic. By default, Django looks for templates in a 'templates' directory within each installed app.

But since this is a relatively small project, we will create our template in the root directory. Then we will make the following changes on settings.py:

```
58 TEMPLATES = [
59     {
60         "BACKEND": "django.template.backends.django.DjangoTemplates",
61         "DIRS": [BASE_DIR / "templates"],
62         "APP_DIRS": True,
63         "OPTIONS": {
64             "context_processors": [
65                 "django.template.context_processors.debug",
66                 "django.template.context_processors.request",
67                 "django.contrib.auth.context_processors.auth",
68                 "django.contrib.messages.context_processors.messages",
69             ],
70         },
71     },
72 ]
```

And similar to the static folder, we are telling django that templates folder is in the root directory.

So, by now you should have a structure similar to this in your project:

```
├── chat
├── config
├── static
├── templates
├── venv
├── db.sqlite3
├── manage.py
├── README.md
└── requirements.txt
```

We are almost done. What's left is creating the html pages.

Adding HTML Files To Templates

We will add two html files:

1. The base.html file, which will be the base template for pages in this project.
2. The index.html file, this will extend the base template and will be the home page of our app.

The Base Template

Create a base.html file and add the following lines of code:

```
1 {% load static %}
2 <!DOCTYPE html>
3 <html lang="en">
4   <head>
5     <meta charset="utf-8">
6     <meta name="viewport" content="width=device-width, initial-scale=1">
7     <meta name="description" content="A real-time chat application">
8     <meta name="theme-color" content="#6366f1">
9     <title>{% block title %}Chat App{% endblock %}</title>
10    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
11    <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css" rel="stylesheet">
12    <link href="{% static 'css/base.css' %}" rel="stylesheet">
13  </head>
14  <body>
15    <div class="container mt-4">
16      {% block content %}
17      {% endblock %}
18    </div>
19    {% block include_js %}
20    {% endblock %}
21    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
22    <script>
23      document.addEventListener('DOMContentLoaded', (event) => {
24        {% block domready %}
25        {% endblock %}
26      })
27    </script>
28  </body>
29 </html>
```

What this basically does is:

- Create the head tags
- Import CSS stylings
- Create a block for content
- Create a block for Javascript

The Index Template

Now create the index.html file and add the following lines of code:


```

1  {% extends "base.html" %}
2
3  {% block title %}Chat Rooms{% endblock %}
4
5  {% block content %}
6  <div class="join-container">
7      <div class="join-card">
8          <div class="join-header">
9              <h2>Join a Chat Room</h2>
10         </div>
11         <div class="join-body">
12             <form method="post">
13                 {% csrf_token %}
14                 <div class="form-group">
15                     <label for="room_name">Room Name</label>
16                     <input type="text" id="room_name" name="room_name" required>
17                     <div class="invalid-feedback">
18                         Please enter a room name.
19                     </div>
20                 </div>
21                 <div class="form-group">
22                     <label for="username">Your Name</label>
23                     <input type="text" id="username" name="username" required>
24                     <div class="invalid-feedback">
25                         Please enter your name.
26                     </div>
27                 </div>
28                 <div class="form-submit">
29                     <button type="submit" class="join-button">Join Room</button>
30                 </div>
31             </form>
32         </div>
33     </div>
34 </div>
35 {% endblock %}

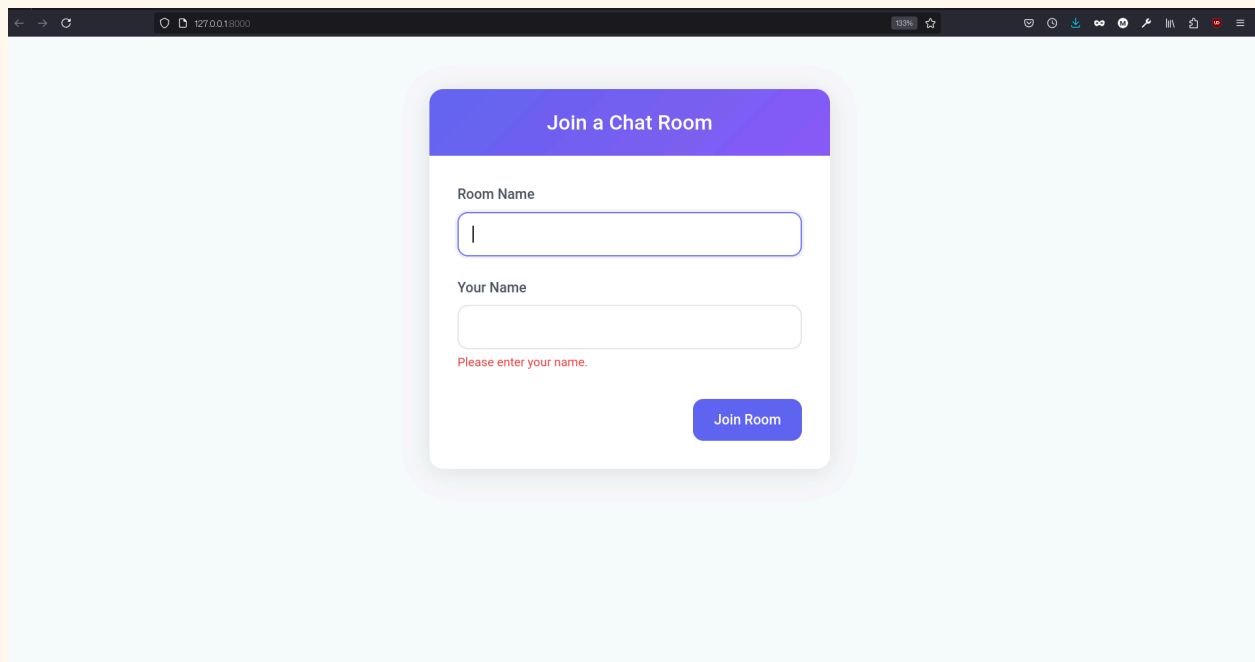
```

What this does is:

- Create a Join chat room card
- Creates form fields
- The form fields are:
 - `username`: Which is the user's name
 - `room_name`: Which is the room the user wants to join to chat

- There is also a CSRF tag, this is for preventing a hacking method called Cross-Site Scripting Attack. The tag is present by default in Django and must be used when submitting forms.

Now when you run your django app you will see a beautiful page like this:



Accepting Form Input

Now let's update our index view to accept form input:

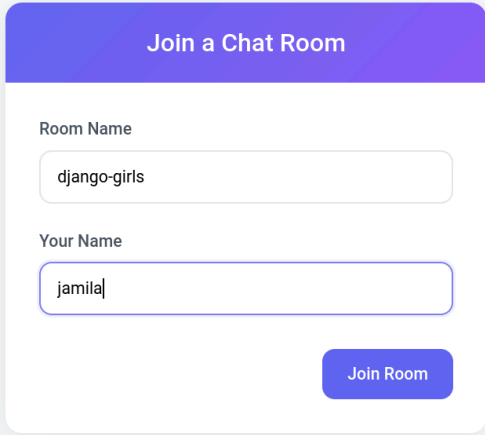
```
1 from django.shortcuts import render
2
3 # Create your views here.
4
5 def index(request):
6
7     if request.method == 'POST':
8         room_name = request.POST.get('room_name')
9         username = request.POST.get('username')
10        print(f"We received username: {username} and room name: {room_name}")
11
12    return render(request, 'index.html')
```

Whenever a form is submitted it sends a POST request. A POST request is a http method to send data to a server. Other http methods include:

- GET: Which fetches data
- DELETE: Which sends a delete request
- UPDATE: Which sends an update request

On the view, we will check if the request method is POST, and if yes we will extract the values. For now we will simply extract these values.

So if you fill in values on the index page and click Join Room, you will see the username and room name logged on the terminal:



The image shows a web form titled "Join a Chat Room". It has a purple header. Below the header, there are two input fields. The first is labeled "Room Name" and contains the text "django-girls". The second is labeled "Your Name" and contains the text "jamila". At the bottom right of the form, there is a blue button with the text "Join Room".

```
We received username: jamila and room name: django-girls  
HTTP POST / 200 [0.01, 127.0.0.1:38676]
```

Now let's create the chat view!

The Chat View

The chat view will be simple. We basically want the user's username and the room they want to join.

Go to your views.py and add the following:

```
15 def chatroom(request, room_name, username):  
16  
17     return render(request, 'room.html', {"room_name": room_name, "username": username})
```

This view accepts parameters `room_name` and `username`. Then it will render the `room.html` template and provide the `room_name` and `username` values.

We don't have a `room.html` template yet, so let's create that:

```

1 {% extends "base.html" %}
2
3 {% block title %}
4   Chat room for "{{ room_name }}"
5 {% endblock %}
6
7 {% block content %}
8   <div class="chat-container">
9     <div class="chat-header">
10      <h2>🗨️ {{ room_name }}</h2>
11      <p>Welcome, {{ username }}!</p>
12    </div>
13    <div id="chat" class="chat-messages"></div>
14
15    <div id="chat-input" class="chat-input-container">
16      <input
17        id="chat-message-input"
18        type="text"
19        placeholder="Type your message..."
20        class="message-input"
21      >
22      <button id="chat-message-submit" class="send-button">
23        <i class="fas fa-location-arrow"></i>
24      </button>
25    </div>
26  </div>
27 {% endblock %}
28
29 {% block include_js %}
30   {{ room_name|json_script:"room_name" }}
31   {{ username|json_script:"username" }}
32 {% endblock %}
33
34 {% block domready %}
35 {% endblock %}

```

This is what the room.html does:

- Extends the base.html template we already made
- Renders the title of our chat room and greets the user
- Creates a nice container for viewing messages
- Creates an input box for us to type our messages
- Notice we also render the room_name and username values with:

...

```
{% block include_js %}
```

```
  {{ room_name|json_script:"room_name" }}
```

```
  {{ username|json_script:"username" }}
```

```
{% endblock %}
```

...

What this does is creates a script tag for both `room_name` and `username`. This is so we can access it later.

Now that we have our `room.html`, let's update the `urls.py` to include the chatroom view.

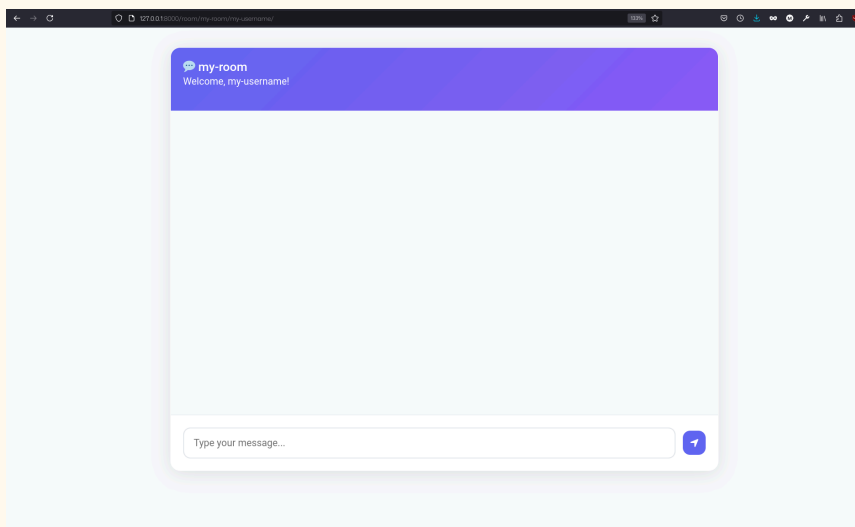
Update `urls.py`:

Go to your `urls.py` inside the `chat` folder and update the `urlpatterns` as follows:

```
6 urlpatterns = [  
7     path('', views.index, name='index'),  
8     path('room/<str:room_name>/<str:username>', views.chatroom, name='chatroom'),  
9 ]
```

We now added a room slug which will call the chatroom view we just created. Notice that the path has two variables, the `room_name` and the `username`.

Now if you go to the url `/room/my-room/my-username`, you should see something like this:



Awesome right? You did great. Take a five minute break to congratulate yourself. Next we will implement the chat feature, but first - a little theory.

Real-time Django with Channels

You are building a chat server to communicate in a room and exchange messages. This functionality requires real-time communication between the server and the client.

Think of it like this:

- Regular websites work like sending letters through mail you have to keep checking your mailbox to see if you got new mail
- But a chat room needs to work more like a phone call - you hear the other person right away when they speak

The old way of making websites (where you have to keep checking for updates) isn't good enough for a chat room. It would be like refreshing your page every few seconds to see new messages - that's slow and wastes resources!

Instead, we need a special way where:

- When someone sends a message, it appears on everyone's screen right away
- You don't need to keep checking or refreshing the page
- The server can instantly "push" new messages to everyone in the chat

This is called "asynchronous communication" - it's like having an open phone line where messages can flow freely both ways.

This makes the chat:

- Faster
- More efficient
- More like a real conversation

We're going to build this chat room using something called ASGI, which helps us create this instant, two-way communication.

Making Your Chat Room Work Better ASGI

Normally, Django works in a simple back-and-forth way:

- Your browser asks for something
- The website responds
- And that's it until you ask for something else

This way of deploying Django is called Web Server Gateway Interface (WSGI).

But for a chat room, we need something better. We need a system that can:

- Keep connections open
- Send messages instantly
- Handle many people chatting at once

This is where two important tools come in:

1. ASGI (think of it as an "upgraded version" of how Django usually works):

- Asynchronous Server Gateway interface allows for asynchronous communication. It's newer and faster
- It can handle real-time stuff
- Perfect for chat rooms!

2. Channels (an extra tool that makes ASGI even better):

- Helps manage ongoing connections
- Great for chat rooms and similar apps

- Makes everything run smoothly

So now we will upgrade our app from just using http, to giving it the ability to use both http and websockets.

Installing Channel In Django

Since we are now changing our django app from using WSGI to ASGI, we will have to install a channels package. Install it with:

```
python -m pip install -U 'channels[daphne]==4.1.0'
```

Then update the `INSTALLED_APPS` inside the `settings.py` to include 'daphne' as follows:



```
33 INSTALLED_APPS = [  
34     "daphne",  
35     # ...  
36 ]
```

Notice that 'daphne' has to be the first item in `INSTALLED_APPS`, otherwise it will not work.

Then you will have to enable asgi support in django. Go to the 'asgi.py' file inside config, and write the following:

```
10 import os
11 from channels.routing import ProtocolTypeRouter
12
13 from django.core.asgi import get_asgi_application
14
15 os.environ.setdefault("DJANGO_SETTINGS_MODULE", "config.settings")
16
17 django_asgi_app = get_asgi_application()
18
19
20 application = ProtocolTypeRouter(
21     {
22         "http": django_asgi_app,
23     }
24 )
```

We define the main ASGI application that will be executed when serving the Django project through ASGI. You use the `ProtocolTypeRouter` class provided by Channels as the main entry point of your routing system. `ProtocolTypeRouter` takes a dictionary that maps communication types like `http` or `websocket` to ASGI applications. We instantiate this class with the default application for the HTTP protocol. Later, we will add a protocol for the WebSocket.

Finally we will update the `settings.py` as follows:

```
133 ASGI_APPLICATION = "config.asgi.application"
```

Now if we run our application with:

```
python manage.py runserver
```

We should see something like this:

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
November 16, 2024 - 05:29:20
Django version 5.1.3, using settings 'config.settings'
Starting ASGI/Daphne version 4.1.2 development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Notice the `Starting ASGI/Daphne`, this tells us that our django is now running via ASGI.

Now that our django app is upgraded, let's give it the ability to accept websocket protocols. For HTTP requests, what we usually do is:

- Define the view
- Add it to the app urls
- Add the app urls to the project urls

We are going to do the same thing for websockets:

- Define a consumer
- Configure websocket urls (called routes)
- Add the app routes to the project urls

Setting Up A Consumer

Consumers are the equivalent of Django views for asynchronous applications. As mentioned, they handle WebSockets in a very similar way to how traditional views handle HTTP requests.

Consumers are ASGI applications that can handle messages, notifications, and other things. Unlike Django views, consumers are built for long-running communication. URLs are mapped to consumers through routing classes that allow you to combine and stack consumers.

Let's implement a basic consumer that can accept WebSocket connections and echoes every message it receives from the WebSocket back to it. This initial functionality will allow the student to send messages to the consumer and receive back the messages it sends.

Create a new file inside the chat application directory and name it `consumers.py`. Add the following code to it:

```
1 import json
2 from channels.generic.websocket import AsyncWebsocketConsumer
3
4
5 class ChatConsumer(AsyncWebsocketConsumer):
6     async def connect(self):
7         # accept connection
8         await self.accept()
9
10    async def disconnect(self, close_code):
11        pass
12
13    async def receive(self, text_data):
14        text_data_json = json.loads(text_data)
15        message = text_data_json["message"]
16
17        # send the message to Websocket
18        await self.send(text_data=json.dumps({"message": message}))
```

This is the `ChatConsumer` consumer. This class inherits from the Channels `AsyncWebsocketConsumer` class to implement a basic WebSocket consumer. In this consumer, you implement the following methods:

- `connect()`: Called when a new connection is received. You accept any connection with `self.accept()`. You can also reject a connection by calling `self.close()`.
- `disconnect()`: Called when the socket closes. You use `pass` because you don't need to implement any action when a client closes the connection.
- `receive()`: Called whenever data is received from the WebSocket. You expect text to be received as `text_data` (this could also be `binary_data` for binary data). You treat the text data received as JSON. Therefore, you use `json.loads()` to load the received JSON data into a Python dictionary. You access the `message` key, which you expect to be present in the JSON structure received. To

echo the message, you send the message back to the WebSocket with `self.send()`, transforming it into JSON format again through `json.dumps()`.

The initial version of your `ChatConsumer` consumer accepts any WebSocket connection and echoes to the WebSocket client every message it receives. Note that the consumer does not broadcast messages to other clients yet. You will build this functionality by implementing a channel layer later.

First, let's expose our consumer by adding it to the URLs of the project.

Routing

You need to define a URL to route connections to the `ChatConsumer` consumer you have implemented.

Channels provides routing classes that allow you to combine and stack consumers to dispatch based on what the connection is. You can think of them as the URL routing system of Django for asynchronous applications.

Create a new file inside the chat application directory and name it `routing.py`. Add the following code to it:

```
1 from django.urls import path
2 from . import consumers
3
4 websocket_urlpatterns = [
5     path(r"ws/room/<str:room_name>/<str:username>/", consumers.ChatConsumer.as_asgi())
6 ]
```

In this code, you map a URL pattern with the `ChatConsumer` class that you defined in the `chat/consumers.py` file. There are some details that are worth reviewing:

- You use Django's `path()` to define the path just as how you do it on `urls.py`
- The URL includes two parameters: `room_name` and `username`. These parameters will be available in the scope of the consumer and will allow you to identify the chat room the user is connecting to.

- You call the `as_asgi()` method of the consumer class in order to get an ASGI application that will instantiate an instance of the consumer for each user connection. This behavior is similar to Django's `as_view()` method for class-based views.

Now let's add these routes to the main route. Go to the `asgi.py` file and add the following:

```
10 import os
11 from channels.auth import AuthMiddlewareStack
12 from channels.routing import ProtocolTypeRouter, URLRouter
13 from chat.routing import websocket_urlpatterns
14
15 from django.core.asgi import get_asgi_application
16
17 os.environ.setdefault("DJANGO_SETTINGS_MODULE", "config.settings")
18
19 django_asgi_app = get_asgi_application()
20
21
22 application = ProtocolTypeRouter(
23     {
24         "http": django_asgi_app,
25         "websocket": AuthMiddlewareStack(URLRouter(websocket_urlpatterns)),
26     }
27 )
```

Note that we have defined two protocols:

1. For http we tell it to use django's default urls
2. For websocket we use the `websocket_urlpatterns` we defined in `routing.py`

Now let us give the browser the ability to connect to the websocket.

Implementing the WebSocket client

Now this is a django workshop, but unfortunately since we are dealing with the browser we are forced to write some javascript. For now don't worry too much about it, I have added comments to explain what they do but if you do not understand it, don't worry.

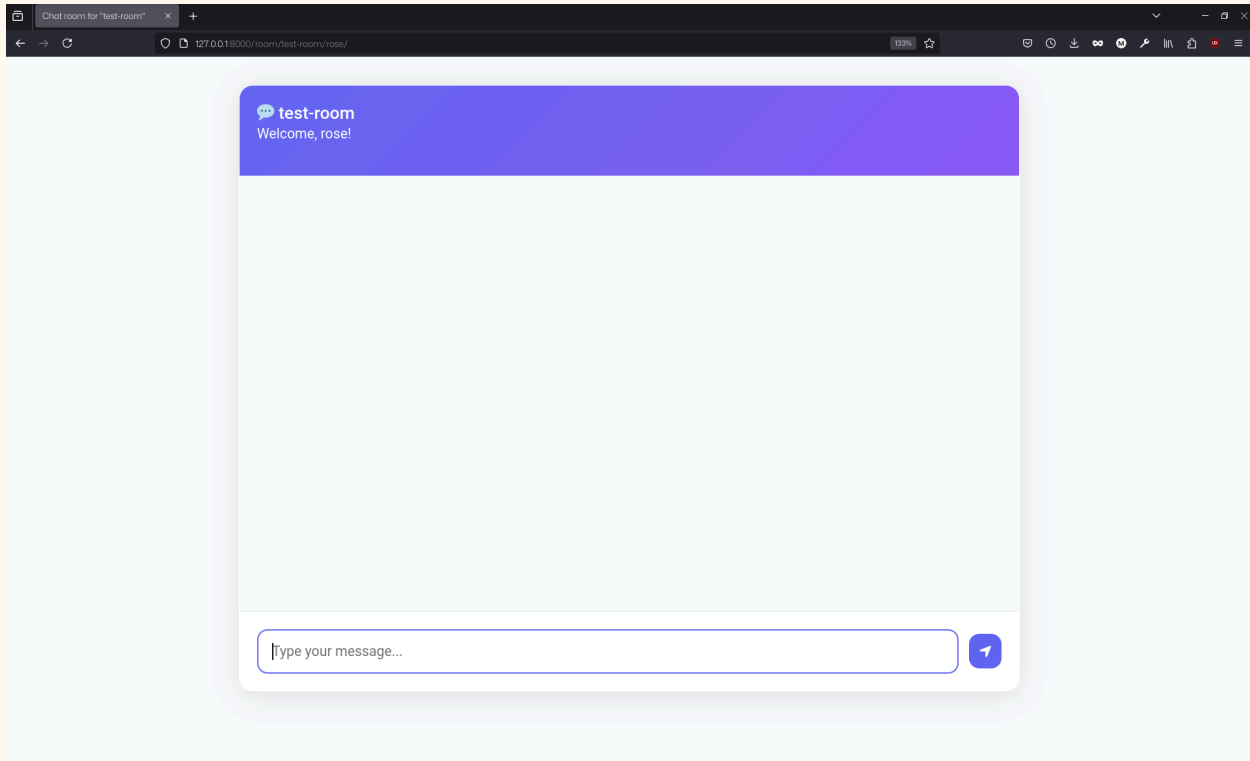
Go to your `room.html` file, and on the `domready` block add the following lines of code:

```

1  {% block domready %}
2  // Get room name and username from HTML elements
3  const room_name = JSON.parse(
4    document.getElementById('room_name').textContent
5  );
6  const username = JSON.parse(
7    document.getElementById('username').textContent
8  );
9
10 // Create WebSocket connection URL and establish connection
11 const url = 'ws://' + window.location.host + '/ws/room/' + room_name + '/' + username + '/';
12 const chatSocket = new WebSocket(url);
13
14 // Handle incoming messages
15 chatSocket.onmessage = function(event) {
16   // Parse the received JSON data
17   const data = JSON.parse(event.data);
18   const chat = document.getElementById('chat');
19
20   // Format the timestamp
21   const dateOptions = {hour: 'numeric', minute: 'numeric'};
22   const datetime = new Date(data.datetime).toLocaleString('en', dateOptions);
23
24   // Determine if the message is from the current user
25   const isMe = data.user === username;
26   const source = isMe ? 'message me' : 'message other';
27   const name = isMe ? 'Me' : data.user;
28
29   // Create and style the message element
30   const messageDiv = document.createElement('div');
31   messageDiv.className = `message ${source} message-new`;
32   messageDiv.innerHTML = `<strong>${name}</strong> +
33     '<span class="date">${datetime}</span><br> + data.message;
34
35   // Add message to chat and scroll to bottom
36   chat.appendChild(messageDiv);
37   chat.scrollTop = chat.scrollHeight;
38
39   // Remove the 'new message' animation class after 300ms
40   setTimeout(() => {
41     messageDiv.classList.remove('message-new');
42   }, 300);
43 };
44
45 // Handle WebSocket connection closure
46 chatSocket.onclose = function(event) {
47   console.error('Chat socket closed unexpectedly');
48 };
49
50 // Get references to input field and submit button
51 const input = document.getElementById('chat-message-input');
52 const submitButton = document.getElementById('chat-message-submit');
53
54 // Handle submit button click
55 submitButton.addEventListener('click', function(event) {
56   const message = input.value.trim();
57   if(message) {
58     // Send message through WebSocket
59     chatSocket.send(JSON.stringify({'message': message}));
60     // Clear input and maintain focus
61     input.value = '';
62     input.focus();
63   }
64 });
65
66 // Handle Enter key press in input field
67 input.addEventListener('keypress', function(event) {
68   if (event.key === 'Enter') {
69     event.preventDefault();
70     submitButton.click();
71   }
72 });
73
74 // Set initial focus to input field
75 input.focus();
76 {% endblock %}

```

So for now if you got to `/room/test-room/rose`` you will see something like this:



Enabling A Channel Layer

Currently with our application, if two people join the same room, they will not be able to communicate with each other. This is because they are in separate channels.

To overcome this, we will add a channel layer to enable people in the same room to be able to talk to each other.

The channel layer has two parts to manage communication:

Channels: these are like personal mailboxes:

- Each mailbox (channel) has a unique address (name)
- Anyone who knows your address can send you letters (messages)
- Only you can check and read your mailbox (consume messages)

Groups: these are like mailing lists:

- A mailing list (group) has a name
- Multiple mailboxes (channels) can be part of the list
- When someone sends a message to the mailing list, everyone on the list gets a copy
- People (channels) can join or leave the list at any time

In our use case:

- A chat application might give each user their own channel for private messages
- All users in a chat room might be part of a group, so when someone sends a message to the group, everyone in the chat room receives it

We will use an InMemory channel layer by adding this on config/settings.py:

```
135 CHANNEL_LAYERS = {  
136     "default": {  
137         "BACKEND": "channels.layers.InMemoryChannelLayer"  
138     }  
139 }
```

You can check if you can connect to the channel by first running this command:

```
python manage.py shell
```

Then run these:

```
>>> import channels.layers  
>>> from asgiref.sync import async_to_sync  
>>> channel_layer = channels.layers.get_channel_layer()  
>>> async_to_sync(channel_layer.send)('test_channel', {'message': 'hello'})  
>>> async_to_sync(channel_layer.receive)('test_channel')
```

You should get the following output:

```
{'message': 'hello'}
```

You have successfully set up the channel layer. Good job!

Updating The Consumer To Broadcast Messages

Now that we have the channel layer, let's update the consumer to use it. We want that whenever we receive a message, we update everyone connected to the channel with that message.

Modify the connect method in consumer.py as follows:

```
7     async def connect(self):
8         self.user = self.scope["url_route"]["kwargs"]["username"]
9         self.room_name = self.scope["url_route"]["kwargs"]["room_name"]
10
11        await self.channel_layer.group_add(
12            self.room_name, self.channel_name
13        )
14
15        await self.accept()
```

The code above does the following:

- We retrieve the username and room_name from the url. Remember we defined the url as:

```
5 path(r"ws/room/<str:room_name>/<str:username>/", consumers.ChatConsumer.as_asgi())
```

So it can get the username and the room name by extracting it from the url.

- Once we get the room_name and username, we just add them to a group.
- Then finally, we add a self.accept() to accept incoming websocket connections.

Let's also update the disconnect method so that we can remove a user from the channel layer. Add the following lines to disconnect method:

```
17 async def disconnect(self, close_code):
18     await self.channel_layer.group_discard(
19         self.room_name, self.channel_name
20     )
```

We remove them from the channel by using `group_discard()`.

Lastly we will update the receive method so that we can broadcast the received message to the group.

Let's import the django's time utility:

```
3 from django.utils import timezone
```

Then update the receive function:

```
22     async def receive(self, text_data):
23         text_data_json = json.loads(text_data)
24         message = text_data_json["message"]
25         now = timezone.now()
26
27         await self.channel_layer.group_send(
28             self.room_name,
29             {
30                 "type": "chat_message",
31                 "message": message,
32                 "user": self.user,
33                 "datetime": now.isoformat(),
34             },
35         )
```

- In the code above we get the data received which is json format, then we extract the message content and assign it to a variable. We get the time it was sent and assign it to the now variable.
- We then broadcast the message with `group_send`, we provide the `room_name` to broadcast to. And we include the data we want to send which are:
 - `message`: ie the message
 - `user`: which is the username
 - `datetime`: which is the time it was sent
- The type is a special field, and here we provide the function we want to use for broadcasting the message. Let's create that function:

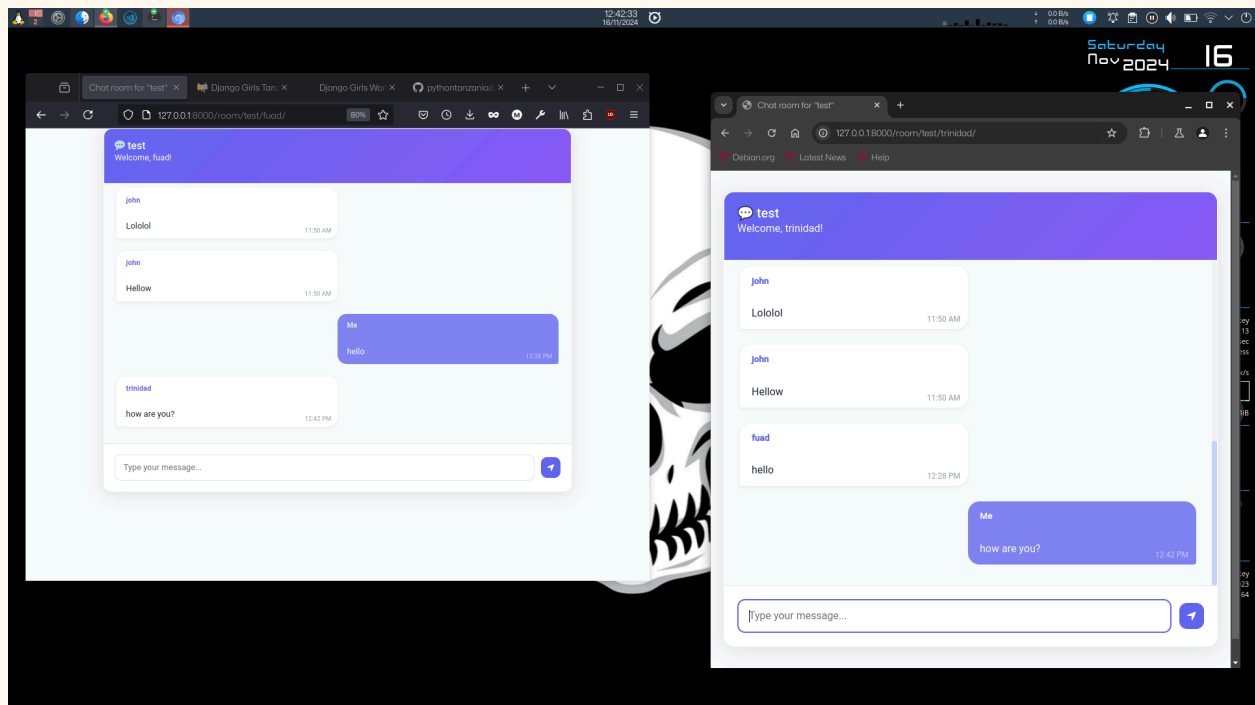
```
37 async def chat_message(self, event):
38     await self.send(text_data=json.dumps(event))
```

This function simply sends the message.

The full consumer.py file should look like this:

```
1 import json
2 from channels.generic.websocket import AsyncWebsocketConsumer
3 from django.utils import timezone
4
5
6 class ChatConsumer(AsyncWebsocketConsumer):
7     async def connect(self):
8         self.user = self.scope["url_route"]["kwargs"]["username"]
9         self.room_name = self.scope["url_route"]["kwargs"]["room_name"]
10
11         await self.channel_layer.group_add(
12             self.room_name, self.channel_name
13         )
14
15         await self.accept()
16
17     async def disconnect(self, close_code):
18         await self.channel_layer.group_discard(
19             self.room_name, self.channel_name
20         )
21
22     async def receive(self, text_data):
23         text_data_json = json.loads(text_data)
24         message = text_data_json["message"]
25         now = timezone.now()
26
27         await self.channel_layer.group_send(
28             self.room_name,
29             {
30                 "type": "chat_message",
31                 "message": message,
32                 "user": self.user,
33                 "datetime": now.isoformat(),
34             },
35         )
36
37     async def chat_message(self, event):
38         await self.send(text_data=json.dumps(event))
```

And voila, that's it. We now have a complete chat application in django. Give it a try by opening two browsers, connect to the same room and send messages.



Adding Video Call Capabilities

Now that we have successfully created a realtime chat application, let's extend it by adding a video call functionality with WebRTC.

As discussed in the WebRTC section, our django server will act as a signalling server to connect the devices that want to video call. After that the devices will communicate directly without interacting with the server.


Let's start by creating an app called call with:

```
django-admin startapp call
```

Then let's include in our settings.py:

```
33 INSTALLED_APPS = [  
34     ...  
35     "chat.apps.ChatConfig",  
36     "call.apps.CallConfig", # add this  
37 ]
```

Let's then create a video call view. Add this under `call/views.py`:



```
1 from django.shortcuts import render
2
3 def index(request):
4     return render(request, 'call.html')
```

But we do not have a `call.html` yet, so we will have to create. Before we do though, let's pause a little and think of what we are trying to do.

- We want to have a page where by someone can enter their username
- We will then give them an option of entering the username of the person they want to call
- We then want the recipient to receive the calling notification
- And if they answer, we want to stream the contents of the video call to both the caller and callee
- Since the contents of the video call itself will be sent from device to device directly without the server, it means we will have to write some javascript.

Let's start with the html, create a `call.html` file under the templates folder, and add the following lines:

```

1 {% extends "base.html" %}
2
3 {% block title %}Video Chat{% endblock %}
4
5 {% block content %}
6 <div class="call-container">
7   <!-- Login Section -->
8   <div id="userName" class="call-card">
9     <div class="text-center">
10      <input type="text" class="form-control form-control-lg mb-3" id="userNameInput"
11        placeholder="Username">
12      <button onclick="login()" class="btn btn-primary btn-lg">Login</button>
13    </div>
14  </div>
15
16  <!-- User Info Section -->
17  <div id="userInfo" class="text-center mb-4">
18    <h1 class="display-6">Hello, <span id="nameHere"></span></h1>
19  </div>
20
21  <!-- Call Section -->
22  <div id="call" class="call-card mb-4">
23    <input type="text" class="form-control form-control-lg mb-3" id="callName" placeholder="Whom to call?">
24    <div class="text-center">
25      <button class="btn btn-success btn-lg" onclick="call()">Call</button>
26    </div>
27  </div>
28
29  <!-- Answer Section -->
30  <div id="answer" class="call-card text-center">
31    <h2 class="mb-4">Incoming Call</h2>
32    
33    <h2 class="mb-4"><span id="callerName"></span></h2>
34    <button class="btn btn-success btn-lg mb-3" onclick="answer()">Answer</button>
35  </div>
36
37  <!-- Calling Section -->
38  <div id="calling" class="call-card text-center">
39    <h2 class="mb-4">Calling</h2>
40    
41    <h3><span id="otherUserNameCA"></span></h3>
42  </div>
43
44  <!-- In Call Section -->
45  <div id="inCall" class="call-card text-center">
46    <h3>On Call With</h3>
47    <h2><span id="otherUserNameC"></span></h2>
48  </div>
49
50  <!-- Videos Section -->
51  <div id="videos" class="video-container">
52    <video id="localVideo" autoplay muted playsinline></video>
53    <div id="remoteVideoDiv" class="text-center">
54      <video id="remoteVideo" autoplay playsinline></video>
55    </div>
56  </div>
57 </div>
58 {% endblock %}

```

Here is a little explanation:

- We have a login section which is where we get the **username** of the caller. When the user presses the **login** button it will call a javascript function called **login()**. We will create this function shortly.
- The **User Info Section** is basically the **Hello + username**
- The **Call Section** is where we put the user we want to call. Then when we click the call button, the button will call the javascript **call()** function. We haven't made this yet.
- The **Answer Section** is what will be displayed when a user is receiving a call. When they call the answer button, it will run the javascript function **answer()**. We haven't made this yet.
- The **Calling Section** is what will be displayed when we are calling someone.
- The **In Call Section** basically shows the **On Call With + username** message
- The **Video Section** is the part that actually shows the video. The **localVideo** is the caller and the **remoteVideo** is the one being called.

NB: Download the dj-profile.jpg image here:

<https://github.com/pythontanzania/DjangoGirlsWorkShopNov24/blob/main/static/images/dj-profile.jpg> and put the image under **static/images/**

Then add the following block in **call.html**:

```

60 {% block include_js %}
61 <script src="/static/js/call.js"></script>
62 <script>
63     // Hide all sections initially
64     document.getElementById("call").style.display = "none";
65     document.getElementById("userInfo").style.display = "none";
66     document.getElementById("answer").style.display = "none";
67     document.getElementById("inCall").style.display = "none";
68     document.getElementById("calling").style.display = "none";
69     document.getElementById("videos").style.display = "none";
70
71     function login() {
72         let userName = document.getElementById("userNameInput").value;
73         myName = userName;
74         document.getElementById("userName").style.display = "none";
75         document.getElementById("call").style.display = "block";
76         document.getElementById("nameHere").innerHTML = userName;
77         document.getElementById("userInfo").style.display = "block";
78         connectSocket();
79     }
80 </script>
81 {% endblock %}

```

Some more explanation:

- We import a `call.js` file, we haven't created this yet so do not worry about it for now.
- We then get all the sections we defined earlier except for `login` and hide them.
- Then we define the `login()` function which is run when a user presses the login button. What this function does is get the username, set the message `Hello + username`. It will then hide the login section and show the User Info and Call Sections.
- It will then call the `connectSocket()` javascript function. We haven't created this yet.

Pat yourself in the back for reaching this far! What's left to finish the video call functionality is to:

- Create the consumer in the call app
- Add websocket routing to the call app
- Add the WebRTC javascript

Adding A Consumer to the Call App

The call consumer will be very similar to the chat consumer we made earlier. Create a `consumer.py` file inside the call app and write the following:

```

1 import json
2 from channels.generic.websocket import AsyncWebsocketConsumer
3
4
5 class CallConsumer(AsyncWebsocketConsumer):
6     """
7     WebSocket consumer for handling real-time video call functionality.
8     Manages user connections, call signaling, and ICE candidate exchange.
9     """
10
11     async def connect(self):
12         """
13         Handles initial WebSocket connection.
14         Accepts the connection and sends a confirmation message to the client.
15         """
16         await self.accept()
17         await self.send(
18             text_data=json.dumps(
19                 {"type": "connection", "data": {"message": "Connected"}}
20             )
21     )

```

We create a consumer class just like in the chat app. We then create a connect function that will accept new websocket connections. We then send a response telling them they have successfully connected. We will also need a `disconnect` function to handle when a user disconnects. So let's write that:

```

23 async def disconnect(self, close_code):
24     """
25     Handles WebSocket disconnection.
26     Removes the user from their associated channel group.
27     """
28     await self.channel_layer.group_discard(self.my_name, self.channel_name)

```

What this does is remove the connected user from the channel group. Finally, just like on the chat app, we will need a `receive` function to handle incoming requests. Since we are expecting to receive different types of messages (call someone, answer calls etc) we will have to check the type of each content and handle it. Let's start with handling login:

```

30 async def receive(self, text_data):
31     """
32     Processes incoming WebSocket messages from clients.
33     Handles different types of events: login, call initiation, call answering, and ICE candidates.
34     """
35     text_data_json = json.loads(text_data)
36     eventType = text_data_json["type"]
37
38     if eventType == "login":
39         # Handle user login: Add user to their personal channel group
40         name = text_data_json["data"]["name"]
41         self.my_name = name
42         await self.channel_layer.group_add(self.my_name, self.channel_name)

```

All messages we receive will have a `type` part and a `data` part. When we receive the login type we will add whoever logged in into a channel group. We haven't written the javascript part but this is what it sends (don't write this):

```

77 // When socket connects, send login info
78 callSocket.onopen = event =>{
79     callSocket.send(JSON.stringify({
80         type: 'login',
81         data: {
82             name: myName
83         }
84     }));
85 }

```

It will tell the server that this is a `login` type, and the data is `name: myName` which is the username of the person calling. This is the name we extract and add to a group.

The next event type we expect is `call`, so let's handle that. Add the following `if-statement` to the `consumer.py` file:

```

44 if eventType == "call":
45     # Handle outgoing call: Notify the callee about the incoming call
46     name = text_data_json["data"]["name"]
47     print(self.my_name, "is calling", name)
48     await self.channel_layer.group_send(
49         name,
50         {
51             "type": "call_received",
52             "data": {
53                 "caller": self.my_name,
54                 "rtcMessage": text_data_json["data"]["rtcMessage"],
55             },
56         },
57     )

```

We extract the name of the person we want to call. We then notify the callee that someone is trying to call them. We do that by calling the `call_received` function and sending the

relevant data. We don't have a `call_received` function yet, so let's write it. Add the following function to `consumers.py`:

```
81 async def call_received(self, event):
82     """
83     Notifies a user about an incoming call.
84     """
85     print("Call received by", self.my_name)
86     await self.send(
87         text_data=json.dumps({"type": "call_received", "data": event["data"]})
88     )
```

This simply notifies the user that they are being called. This is the javascript block that handles the various types we send to it:

```
87 // Handle incoming WebSocket messages
88 callSocket.onmessage = (e) =>{
89     let response = JSON.parse(e.data);
90     let type = response.type;
91
92     // Handle different types of messages
93     if(type == 'connection') {
94         console.log(response.data.message)
95     }
96     if(type == 'call_received') {
97         onNewCall(response.data)
98     }
99     if(type == 'call_answered') {
100         onCallAnswered(response.data);
101     }
102     if(type == 'ICEcandidate') {
103         onICECandidate(response.data);
104     }
105 }
106
107 // Handler for receiving a new call
108 const onNewCall = (data) =>{
109     otherUser = data.caller;
110     remoteRTCMessage = data.rtcMessage
111     document.getElementById("callerName").innerHTML = otherUser;
112     document.getElementById("call").style.display = "none";
113     document.getElementById("answer").style.display = "block";
114 }
```

So it is similar to our `consumers.py` file, it also checks what type is being received and handles accordingly. We can see that the `onNewCall` function will hide irrelevant elements in the html and show the `answer` section we wrote on the html.

Now that we already handle calling on the `consumers.py` file, let's handle answering a call. On the list of `if-statements` inside the `receive` function, add this:

```

59 if eventType == "answer_call":
60     # Handle call answer: Forward answer to the original caller
61     caller = text_data_json["data"]["caller"]
62     await self.channel_layer.group_send(
63         caller,
64         {
65             "type": "call_answered",
66             "data": {"rtcMessage": text_data_json["data"]["rtcMessage"]},
67         },
68     )

```

What this does is extract the caller's name and then runs the `call_answered` function. We don't have a `call_answered` function yet so let's write it:

```

90 async def call_answered(self, event):
91     """
92     Notifies the caller that their call has been answered.
93     """
94     print(self.my_name, "'s call answered")
95     await self.send(
96         text_data=json.dumps({"type": "call_answered", "data": event["data"]})
97     )

```

This simply sends a message to the javascript that they have answered the call and that they can start the video call. This is what the javascript looks like (don't write this):

```

117 const onCallAnswered = (data) =>{
118     remoteRTCMMessage = data.rtcMessage
119     peerConnection.setRemoteDescription(new RTCSessionDescription(remoteRTCMMessage));
120     document.getElementById("calling").style.display = "none";
121     console.log("Call Started. They Answered");
122     callProgress()
123 }

```

We're almost done. Remember that WebRTC is a peer to peer connection, meaning that devices communicate directly with no server in between. To ensure that this connection is reliable, there is something called **Interactive Connectivity Establishment (ICE)**. The purpose of this is to represent various ways two peers can connect to each other. This is to ensure reliability. So we will have to add this on the `receive` function on our `consumer.py` file. Add the following `if` -statement:

```

70 if eventType == "ICEcandidate":
71     # Handle ICE candidate exchange between peers
72     user = text_data_json["data"]["user"]
73     await self.channel_layer.group_send(
74         user,
75         {
76             "type": "ICEcandidate",
77             "data": {"rtcMessage": text_data_json["data"]["rtcMessage"]},
78         },
79     )

```

This will receive the various **ICE** candidates and send them to the javascript via the `ICEcandidate` function. We don't have that function yet so let's write it:

```

99 async def ICEcandidate(self, event):
100     """
101     Forwards ICE candidates between peers for WebRTC connection establishment.
102     """
103     await self.send(
104         text_data=json.dumps({"type": "ICEcandidate", "data": event["data"]})
105     )

```

This basically forwards the received information as is. For the curious, this is how the javascript handles the ICE data (don't write this):

```
125     // Handler for receiving ICE candidates
126     const onICECandidate = (data) =>{
127         console.log("GOT ICE candidate");
128         let message = data.rtcMessage
129         let candidate = new RTCIceCandidate({
130             sdpMLineIndex: message.label,
131             candidate: message.candidate
132         });
133
134         if (peerConnection) {
135             console.log("ICE candidate Added");
136             peerConnection.addIceCandidate(candidate);
137         } else {
138             console.log("ICE candidate Pushed");
139             iceCandidatesFromCaller.push(candidate);
140         }
141     }
```

We're finally done with the `consumer.py` file. Your `consumer.py` file in the call app should look like this:


```

1 import json
2 from channels.generic.websocket import AsyncWebsocketConsumer
3
4
5 class CallConsumer(AsyncWebsocketConsumer):
6     """
7     WebSocket consumer for handling real-time video call functionality.
8     Manages user connections, call signaling, and ICE candidate exchange.
9     """
10
11     async def connect(self):
12         """
13         Handles initial WebSocket connection.
14         Accepts the connection and sends a confirmation message to the client.
15         """
16         await self.accept()
17         await self.send(
18             text_data=json.dumps(
19                 {"type": "connection", "data": {"message": "Connected"}}
20             )
21         )
22
23     async def disconnect(self, close_code):
24         """
25         Handles WebSocket disconnection.
26         Removes the user from their associated channel group.
27         """
28         await self.channel_layer.group_discard(self.my_name, self.channel_name)
29
30     async def receive(self, text_data):
31         """
32         Processes incoming WebSocket messages from clients.
33         Handles different types of events: login, call initiation, call answering, and ICE candidates.
34         """
35         text_data_json = json.loads(text_data)
36         eventType = text_data_json["type"]
37
38         if eventType == "login":
39             # Handle user login: Add user to their personal channel group
40             name = text_data_json["data"]["name"]
41             self.my_name = name
42             await self.channel_layer.group_add(self.my_name, self.channel_name)
43
44         if eventType == "call":
45             # Handle outgoing call: Notify the callee about the incoming call
46             name = text_data_json["data"]["name"]
47             print(self.my_name, "is calling", name)
48             await self.channel_layer.group_send(
49                 name,
50                 {
51                     "type": "call_received",
52                     "data": {
53                         "caller": self.my_name,
54                         "rtcMessage": text_data_json["data"]["rtcMessage"],
55                     },
56                 },
57             )
58
59         if eventType == "answer_call":
60             # Handle call answer: Forward answer to the original caller
61             caller = text_data_json["data"]["caller"]
62             await self.channel_layer.group_send(
63                 caller,
64                 {
65                     "type": "call_answered",
66                     "data": {"rtcMessage": text_data_json["data"]["rtcMessage"]},
67                 },
68             )
69
70         if eventType == "ICEcandidate":
71             # Handle ICE candidate exchange between peers
72             user = text_data_json["data"]["user"]
73             await self.channel_layer.group_send(
74                 user,
75                 {
76                     "type": "ICEcandidate",
77                     "data": {"rtcMessage": text_data_json["data"]["rtcMessage"]},
78                 },
79             )

```

```

80
81     async def call_received(self, event):
82         """
83         Notifies a user about an incoming call.
84         """
85         print("Call received by", self.my_name)
86         await self.send(
87             text_data=json.dumps({"type": "call_received", "data": event["data"]})
88         )
89
90     async def call_answered(self, event):
91         """
92         Notifies the caller that their call has been answered.
93         """
94         print(self.my_name, "'s call answered")
95         await self.send(
96             text_data=json.dumps({"type": "call_answered", "data": event["data"]})
97         )
98
99     async def ICEcandidate(self, event):
100         """
101         Forwards ICE candidates between peers for WebRTC connection establishment.
102         """
103         await self.send(
104             text_data=json.dumps({"type": "ICEcandidate", "data": event["data"]})
105         )

```

Adding Consumer Routes To Call App

We should then add a `routing.py` file similar to how we did in the chat app. Create a `routing.py` file inside the call app and write this:

```

1 from django.urls import re_path
2
3 from . import consumers
4
5 call_websocket_urlpatterns = [
6     re_path(r'ws/call/', consumers.CallConsumer.as_asgi()),
7 ]

```

This just creates a websocket url which is `ws/call/`. We then have to update the `asgi.py` file to include this new router:

```
10 import os
11 from channels.auth import AuthMiddlewareStack
12 from channels.routing import ProtocolTypeRouter, URLRouter
13 from chat.routing import websocket_urlpatterns
14 from call.routing import call_websocket_urlpatterns # add this
15
16 from django.core.asgi import get_asgi_application
17
18 os.environ.setdefault("DJANGO_SETTINGS_MODULE", "config.settings")
19
20 django_asgi_app = get_asgi_application()
21
22
23 application = ProtocolTypeRouter(
24     {
25         "http": django_asgi_app,
26         "websocket": AuthMiddlewareStack(
27             URLRouter(websocket_urlpatterns + call_websocket_urlpatterns) # add this
28         ),
29     }
30 )
```

And we are all set. Last but not least we have to include the `call.js` file. Go to <https://raw.githubusercontent.com/pythontanzania/DjangoGirlsWorkShopNov24/refs/heads/main/static/js/call.js> and paste its content in `call.js` file. This file should be under `static/js/call.js`.

And congratulations, you have integrated video call capabilities. Here's a screenshot of a working demo:

Hello, fuad

Incoming Call



avi

Answer

Hello, fuad

On Call With
avi

