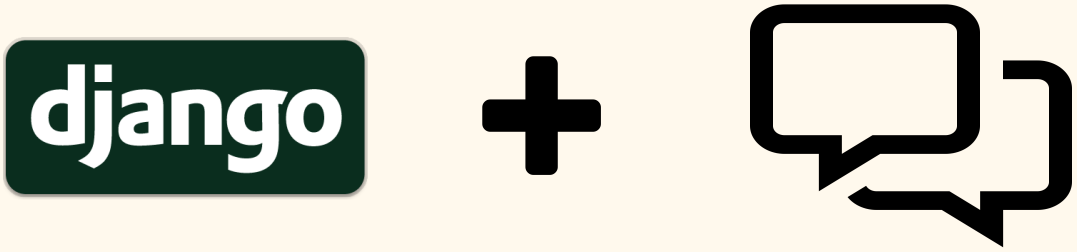
DJANGO GIRLS

WEBSOCKET WORKSHOP



Introduction

Ever sent a message on WhatsApp or Telegram and wondered how it works behind the scenes? Get ready to unlock that mystery! In this hands-on workshop, we'll build an exciting real-time chat application using Django that lets people connect and chat instantly in different rooms.

What You'll Create 💬

A fully functional chat application where users can:

- Join different chat rooms
- Send and receive messages instantly
- See message history
- Experience real-time updates

What You'll Learn 📚

1. **Websockets Explained Simply**
    - What they are and why they're perfect for chat apps
    - How they're different from regular web connections
2. **Hands-On Development**
    - Create your own WebSocket consumer
    - Set up proper routing
    - Build an interactive client interface
3. **Power Features**
    - Integrate Redis for enhanced performance
    - Make your app lightning-fast with asynchronous programming
    - Store chat history in a database

Prerequisites

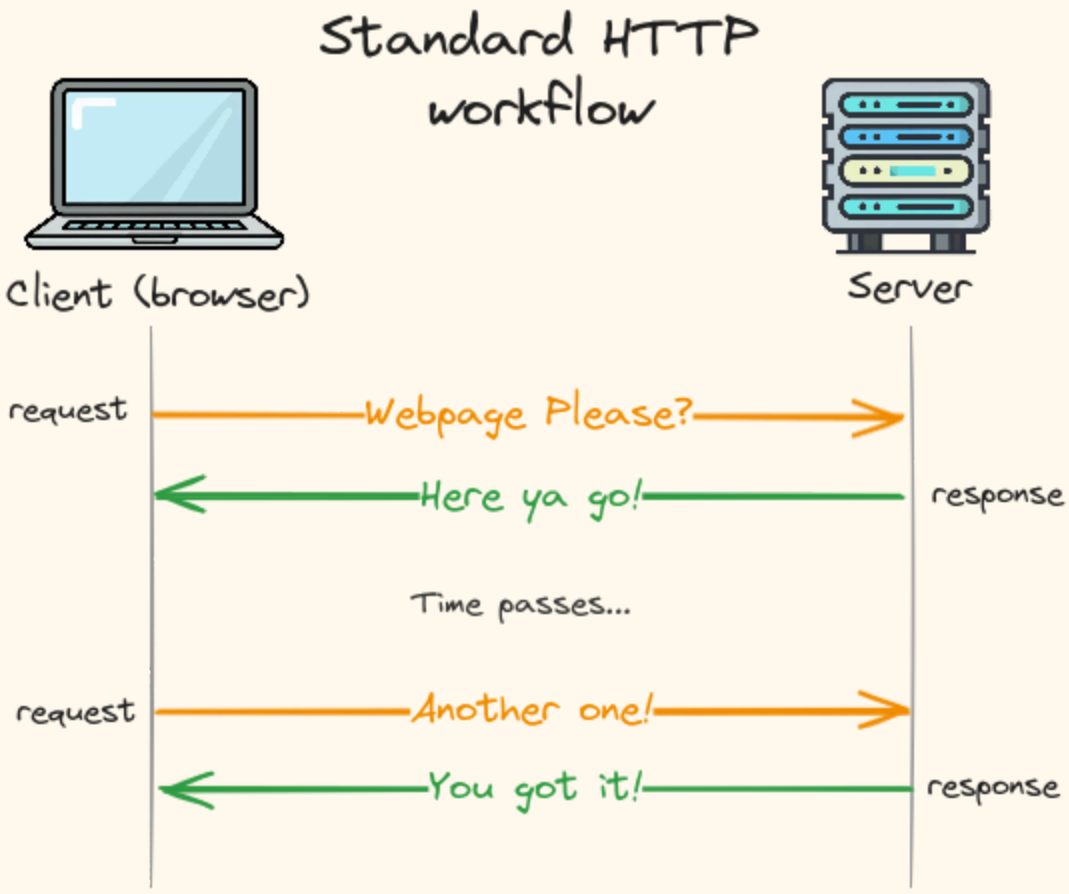
Before you start make sure you have the following installed:

- Python 3.10 and above
- Django 5

WEBSOCKETS

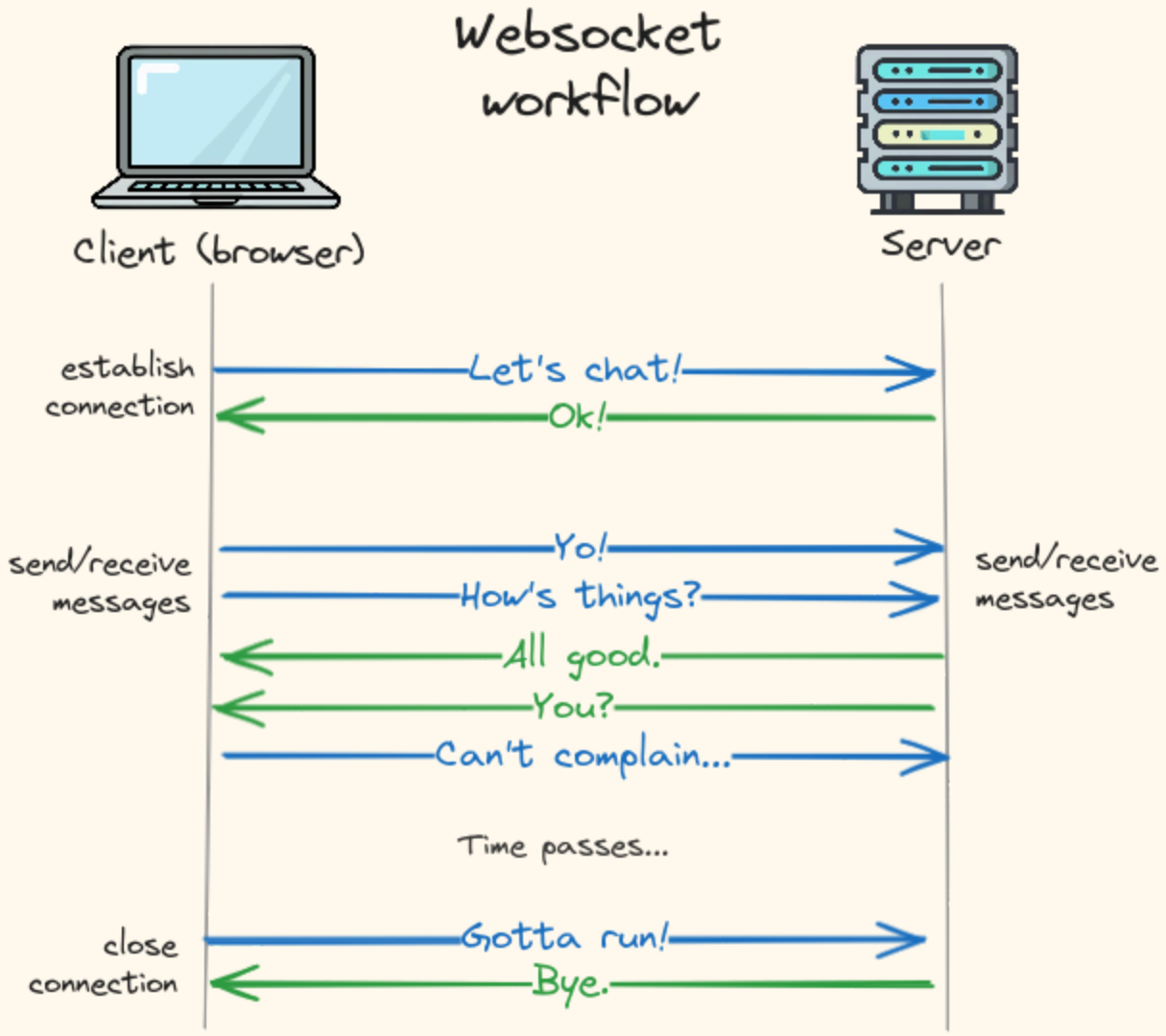
Websockets sound fancy, and anyone who has never used them will probably think of them as black magic. But, conceptually, they are quite simple!

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.

Enough talk. Let's code.

Creating A Chat Application

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

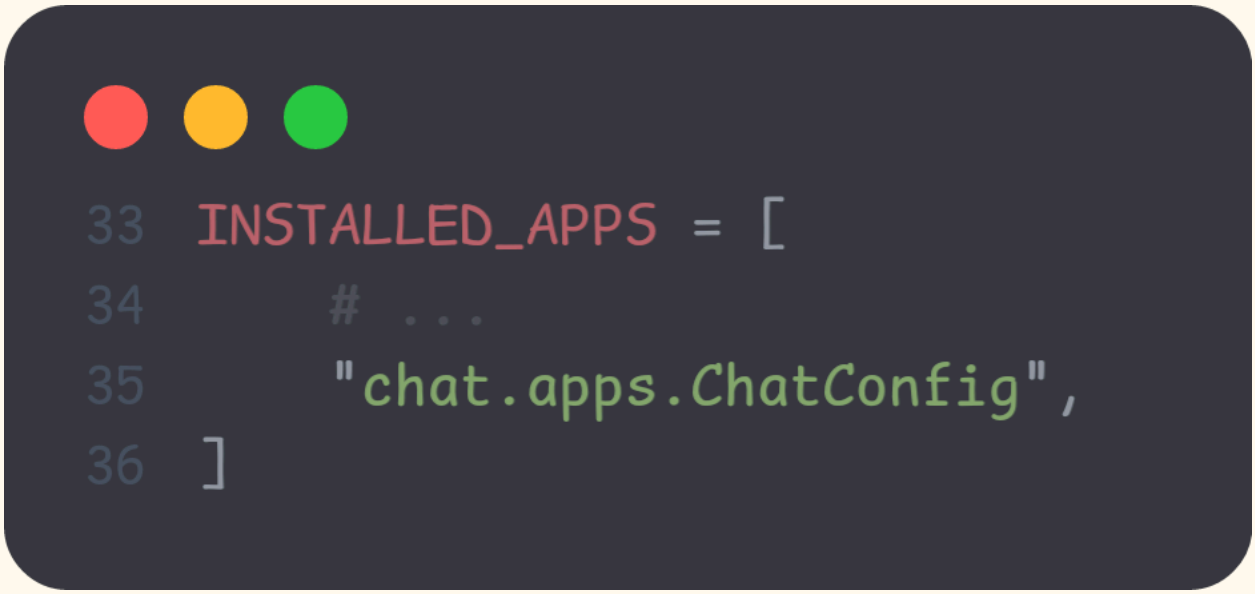
Go to your project folder and start by creating a new project with django:

`django-admin startproject config .`

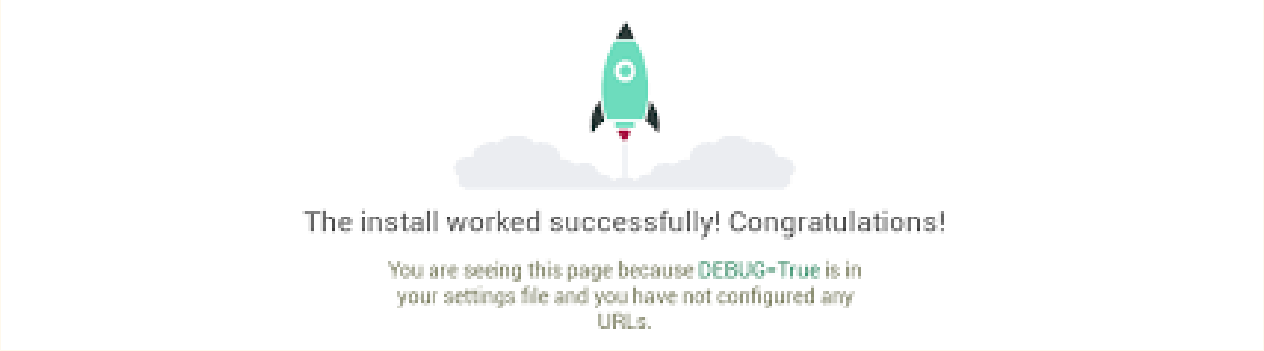
Then create an app called chat with:

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:



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:

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

app_name = 'chat'

urlpatterns = [
    path('', views.index, name='index'),
]
```

What we are basically doing is telling django that whenever someone visits the root url, then display 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:

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

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

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.

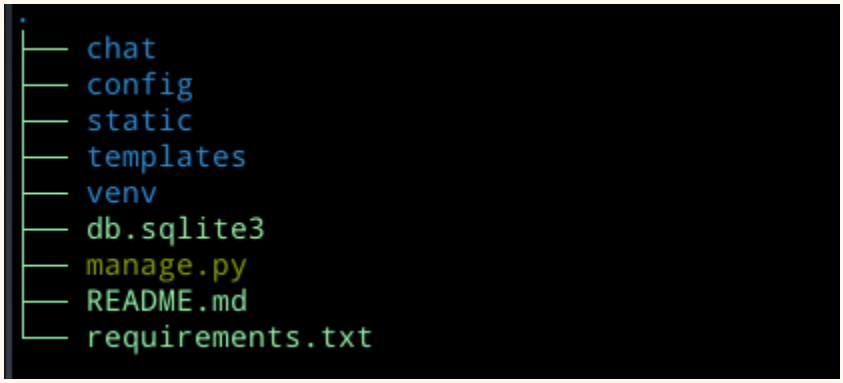
Templates allow us to create reusable HTML layouts and inject dynamic content while keeping our presentation logic separate from our business logic. By default, Django looks for templates in a 'templates' directory within each installed app.

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:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": [BASE_DIR / "templates"],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.template.context_processors.debug",
                "django.template.context_processors.request",
                "django.contrib.auth.context_processors.auth",
                "django.contrib.messages.context_processors.messages",
            ],
        },
    },
]
```

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:



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:

```
{% load static %}
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="description" content="A real-time chat application">
    <meta name="theme-color" content="#6366f1">
    <title>{% block title %}Chat App{% endblock %}</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
    <link href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0/css/all.min.css" rel="stylesheet">
    <link href="{% static 'css/base.css' %}" rel="stylesheet">
  </head>
  <body>
    <div class="container mt-4">
      {% block content %}
      {% endblock %}
    </div>
    {% block include_js %}
    {% endblock %}
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
    <script>
      document.addEventListener('DOMContentLoaded', (event) => {
        {% block domready %}
        {% endblock %}
      })
    </script>
  </body>
</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:

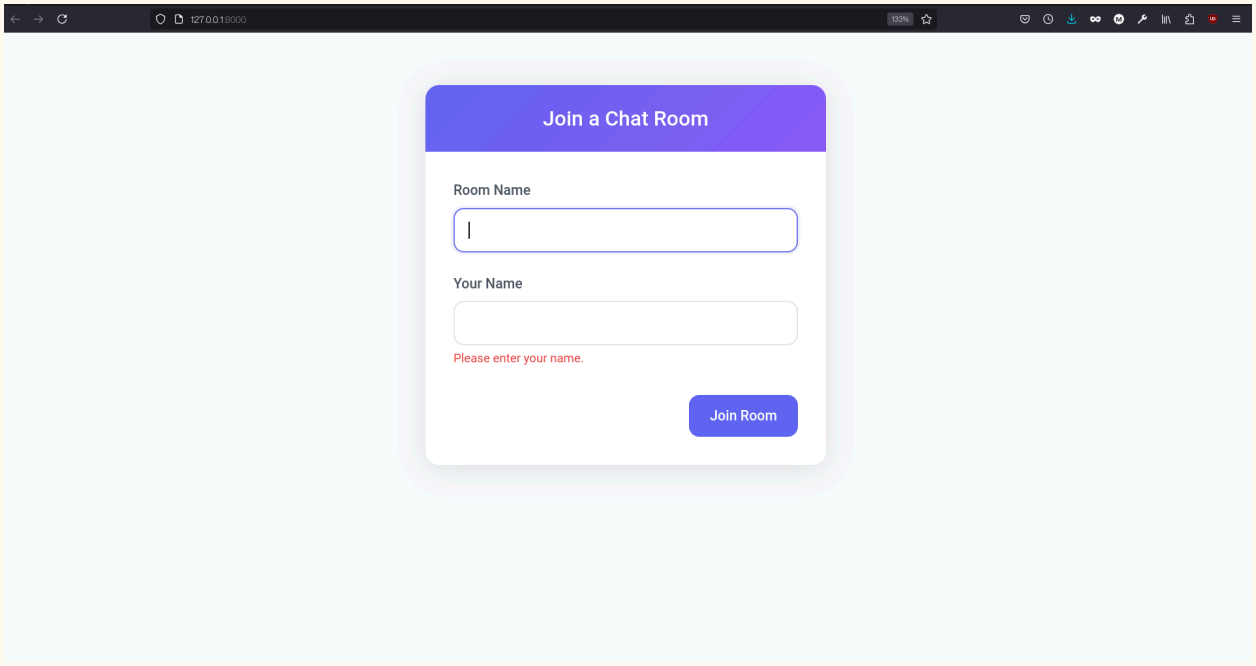
```
{% extends "base.html" %}

{% block title %}Chat Rooms{% endblock %}

{% block content %}
<div class="join-container">
    <div class="join-card">
        <div class="join-header">
            <h2>Join a Chat Room</h2>
        </div>
        <div class="join-body">
            <form method="post">
                {% csrf_token %}
                <div class="form-group">
                    <label for="room_name">Room Name</label>
                    <input type="text" id="room_name" name="room_name" required>
                    <div class="invalid-feedback">
                        Please enter a room name.
                    </div>
                </div>
                <div class="form-group">
                    <label for="username">Your Name</label>
                    <input type="text" id="username" name="username" required>
                    <div class="invalid-feedback">
                        Please enter your name.
                    </div>
                </div>
                <div class="form-submit">
                    <button type="submit" class="join-button">Join Room</button>
                </div>
            </form>
        </div>
    </div>
</div>
{% 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:

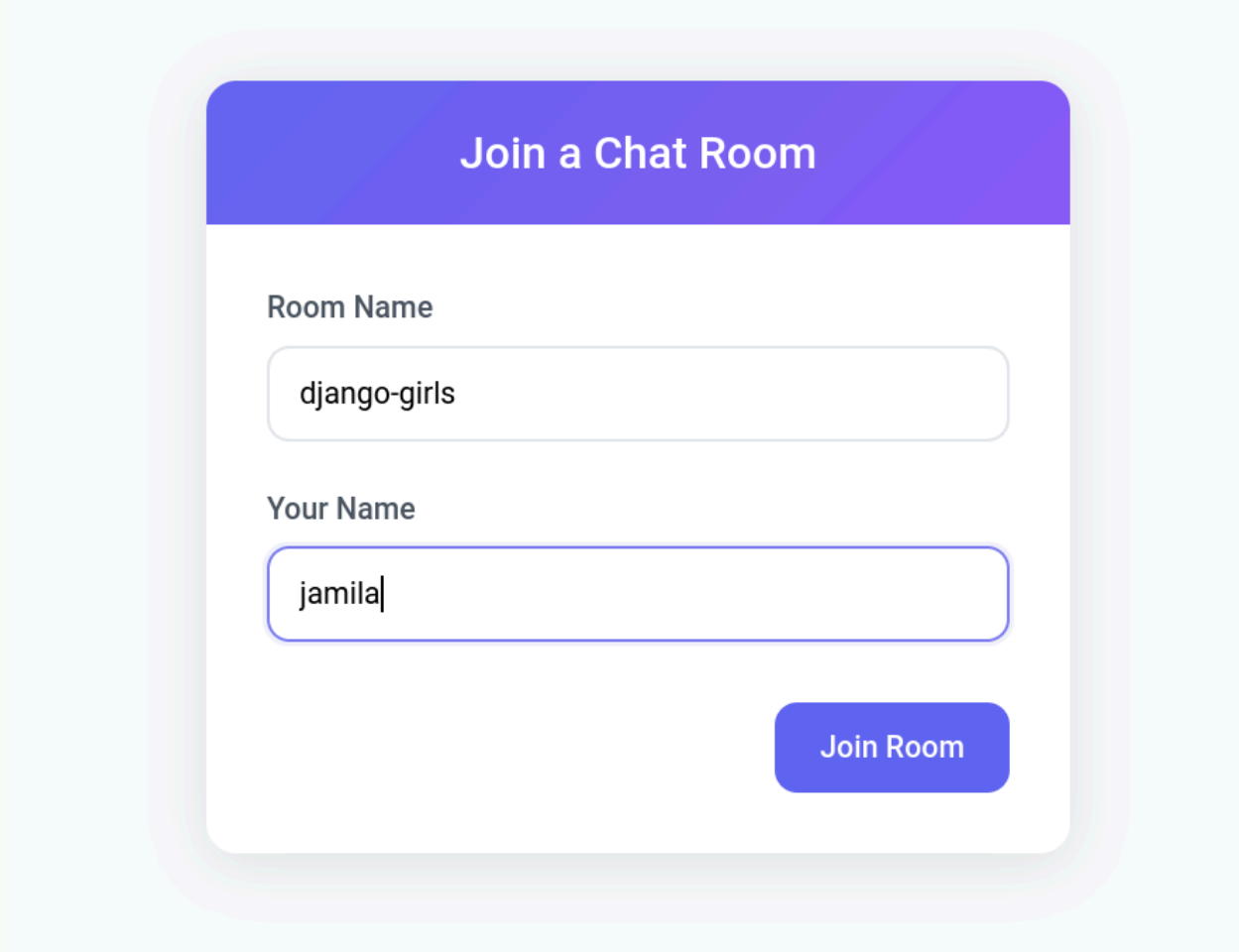
```python
from django.shortcuts import render

# Create your views here.

def index(request):

    if request.method == 'POST':
        room_name = request.POST.get('room_name')
        username = request.POST.get('username')
        print(f"We received username: {username} and room name: {room_name}")

    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:




```
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:

```
def chatroom(request, room_name, username):

    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:

```
{% extends "base.html" %}

{% block title %}
  Chat room for "{{ room_name }}"
{% endblock %}

{% block content %}
  <div class="chat-container">
    <div class="chat-header">
      <h2>💬 {{ room_name }}</h2>
      <p>Welcome, {{ username }}!</p>
    </div>
    <div id="chat" class="chat-messages"></div>

    <div id="chat-input" class="chat-input-container">
      <input
        id="chat-message-input"
        type="text"
        placeholder="Type your message..."
        class="message-input"
      >
      <button id="chat-message-submit" class="send-button">
        <i class="fas fa-location-arrow"></i>
      </button>
    </div>
  </div>
{% endblock %}

{% block include_js %}
  {{ room_name|json_script:"room_name" }}
  {{ username|json_script:"username" }}
{% endblock %}

{% block domready %}
{% 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.

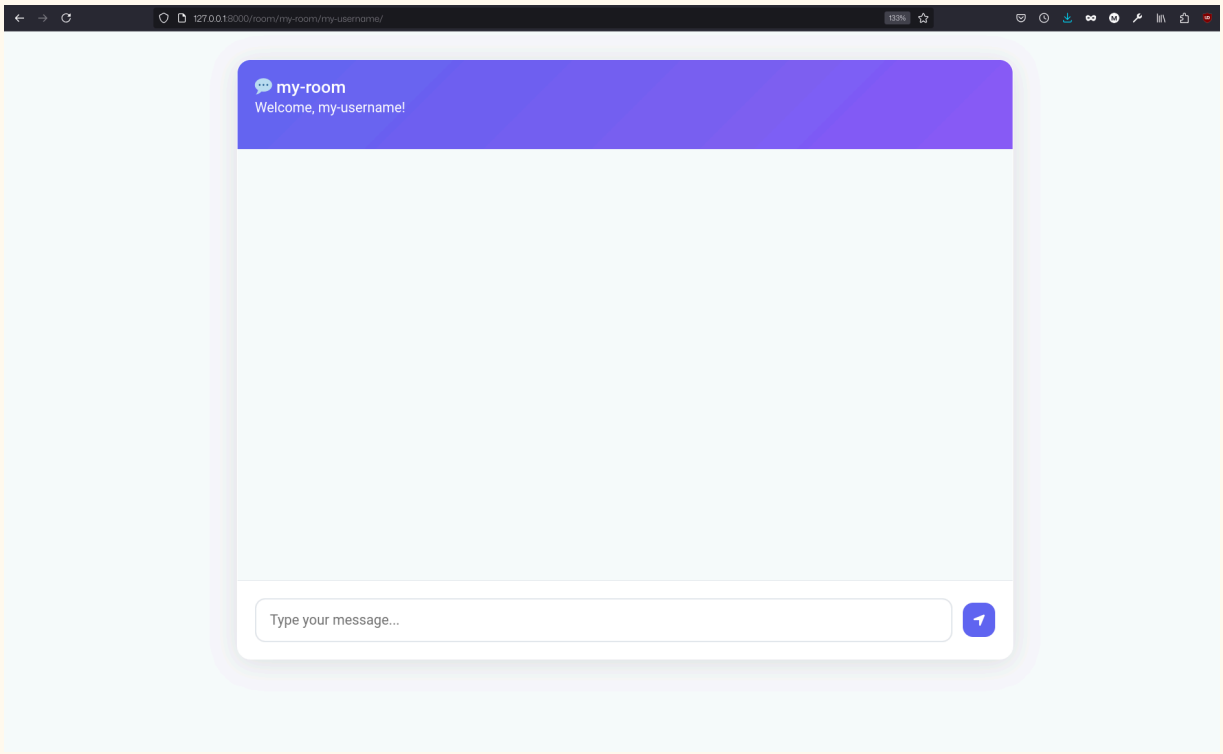
Update urls.py:

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

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

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:

```
INSTALLED_APPS = [
    "daphne",
    # ...
]
```

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:

```
import os
from channels.routing import ProtocolTypeRouter

from django.core.asgi import get_asgi_application

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "config.settings")

django_asgi_app = get_asgi_application()


application = ProtocolTypeRouter(
    {
        "http": django_asgi_app,
    }
)
```

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:

```python
import json
from channels.generic.websocket import AsyncWebsocketConsumer


class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        # accept connection
        await self.accept()

    async def disconnect(self, close_code):
        pass

    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json["message"]

        # send the message to Websocket
        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:

• connnect(): 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:

```
from django.urls import path
from . import consumers

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

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:

```
import os
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter
from chat.routing import websocket_urlpatterns

from django.core.asgi import get_asgi_application

os.environ.setdefault("DJANGO_SETTINGS_MODULE", "config.settings")

django_asgi_app = get_asgi_application()


application = ProtocolTypeRouter(
    {
        "http": django_asgi_app,
        "websocket": AuthMiddlewareStack(URLRouter(websocket_urlpatterns)),
    }
)
```

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.

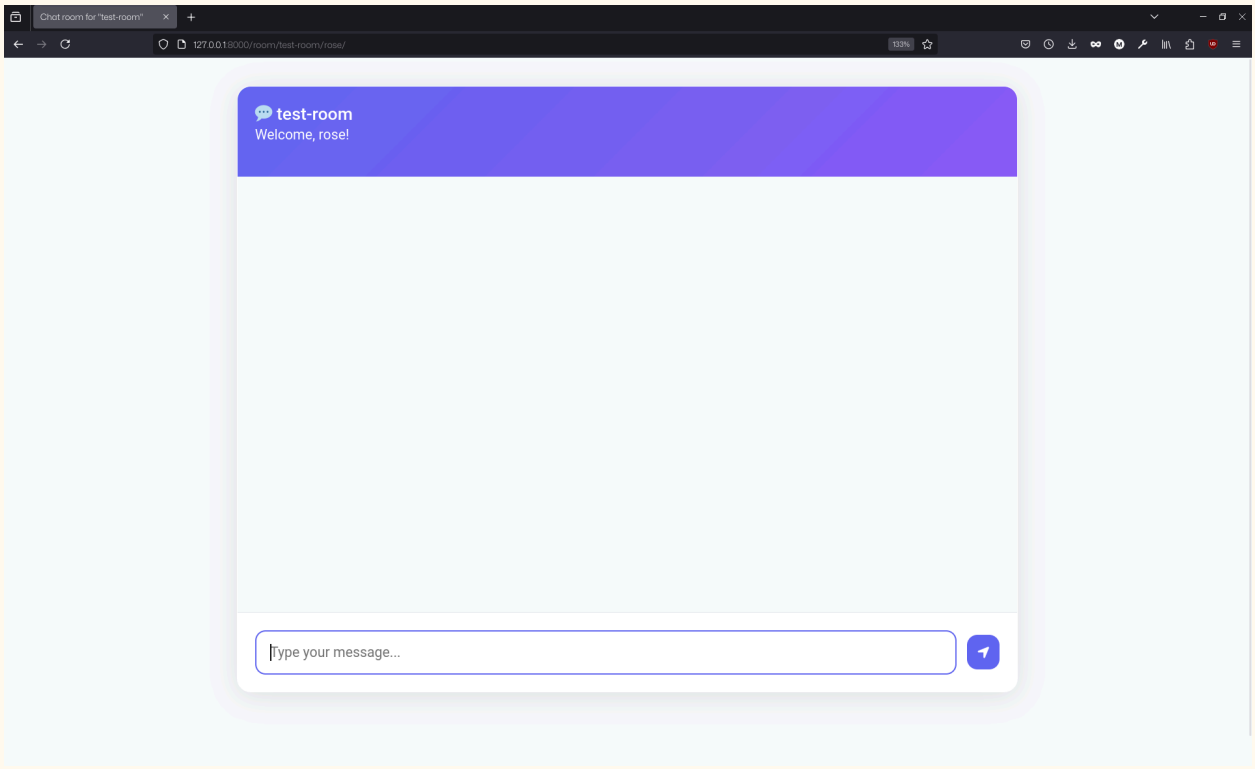
Implementing the WebSocket client

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:

```
{% block domready %}
  // Get room name and username from HTML elements
  const room_name = JSON.parse(
    document.getElementById('room_name').textContent
  );
  const username = JSON.parse(
    document.getElementById('username').textContent
  );

  // Create WebSocket connection URL and establish connection
  const url = 'ws://' + window.location.host + '/ws/room/' + room_name + '/' + username + '/';
  const chatSocket = new WebSocket(url);

  // Handle incoming messages
  chatSocket.onmessage = function(event) {
    // Parse the received JSON data
    const data = JSON.parse(event.data);
    const chat = document.getElementById('chat');

    // Format the timestamp
    const dateOptions = {hour: 'numeric', minute: 'numeric'};
    const datetime = new Date(data.datetime).toLocaleString('en', dateOptions);

    // Determine if the message is from the current user
    const isMe = data.user === username;
    const source = isMe ? 'message me' : 'message other';
    const name = isMe ? 'Me' : data.user;

    // Create and style the message element
    const messageDiv = document.createElement('div');
    messageDiv.className = `message ${source} message-new`;
    messageDiv.innerHTML = '<strong>' + name + '</strong>' +
                           '<span class="date">' + datetime + '</span><br>' + data.message;

    // Add message to chat and scroll to bottom
    chat.appendChild(messageDiv);
    chat.scrollTop = chat.scrollHeight;

    // Remove the 'new message' animation class after 300ms
    setTimeout(() => {
        messageDiv.classList.remove('message-new');
    }, 300);
  };

  // Handle WebSocket connection closure
  chatSocket.onclose = function(event) {
    console.error('Chat socket closed unexpectedly');
  };

  // Get references to input field and submit button
  const input = document.getElementById('chat-message-input');
  const submitButton = document.getElementById('chat-message-submit');

  // Handle submit button click
  submitButton.addEventListener('click', function(event) {
    const message = input.value.trim();
    if(message) {
      // Send message through WebSocket
      chatSocket.send(JSON.stringify({'message': message}));
      // Clear input and maintain focus
      input.value = '';
      input.focus();
    }
  });

  // Handle Enter key press in input field
  input.addEventListener('keypress', function(event) {
    if (event.key === 'Enter') {
      event.preventDefault();
      submitButton.click();
    }
  });

  // Set initial focus to input field
  input.focus();
{% 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 Redis as our channel layer. So let’s set it up.

Setting Up A Channel Layer With Redis

Redis is like a super-fast digital notebook that stores information using labels (keys) and their corresponding values.

Example:

- Like storing "username: john_doe" or "favorite_color: blue"

Using Redis has many advantages such as:

1. Lightning-fast access to data
2. Perfect for temporary data storage (like user sessions)
3. Great for counting things (like website visits)
4. Helps reduce database load
5. Useful for caching frequently accessed data

Think of it as a quick-access storage shelf where you can easily put and retrieve items using labels, rather than searching through a large filing cabinet (traditional database).

Redis is usually used in:

- Storing login sessions
- Saving game scores
- Caching website content
- Managing real-time data

And many other places. We will use redis to store our chat rooms.

To use the redis layer as a channel install the package as follows:

```
python -m pip install channels-redis==4.2.0
```

Then edit the settings.py file as follows:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("127.0.0.1", 6379)],
        },
    },
}
```

Here we define the configuration of the channel layer, we tell django that we will use Redis and that Redis can be accessed via port 6379.

Note that there are many ways to install redis, you can refer to the official documentation here: https://redis.io/docs/latest/operate/oss_and_stack/install/install-redis/

But the easiest way to have redis up and running is by using docker. If you have docker installed, open a new terminal and start redis with:

```
docker run -it --rm --name redis -p 6379:6379 redis:7.2.4
```

You can check if you can connect to the redis 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'}
```

The channel layer is successfully communicating with Redis. Good job!

Updating The Consumer To Broadcast Messages

Now that we have the channel layer, lets 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:

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

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:

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

Then update the receive function:

```
    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json["message"]
        now = timezone.now()

        await self.channel_layer.group_send(
            self.room_name,
            {
                "type": "chat_message",
                "message": message,
                "user": self.user,
                "datetime": now.isoformat(),
            },
        )
```

- 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:

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

This function simply sends the message.

The full consumer.py file should look like this:

```python
import json
from channels.generic.websocket import AsyncWebsocketConsumer
from django.utils import timezone


class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.user = self.scope["url_route"]["kwargs"]["username"]
        self.room_name = self.scope["url_route"]["kwargs"]["room_name"]

        await self.channel_layer.group_add(
            self.room_name, self.channel_name
        )

        await self.accept()

    async def disconnect(self, close_code):
        await self.channel_layer.group_discard(
            self.room_name, self.channel_name
        )

    async def receive(self, text_data):
        text_data_json = json.loads(text_data)
        message = text_data_json["message"]
        now = timezone.now()

        await self.channel_layer.group_send(
            self.room_name,
            {
                "type": "chat_message",
                "message": message,
                "user": self.user,
                "datetime": now.isoformat(),
            },
        )

    async def chat_message(self, event):
        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.