Chat Lounge

A chat room web application, built in Python using Django.

By Todor Popov

Documentation

Contents

[Technical Overview 2](#_Toc119763566)

[Structure of the application 3](#_Toc119763567)

[Local Installation 3](#_Toc119763568)

Installation Overview………………………………………………………………………………………………………………….4

1. Setting up Locally…………………………………………………………………………………………………………………….4

2. Editing the Project Settings……………………………………………………………………………………….………….…4

3. Redis and "InMemoryChannelLayer"………………………………………………………………………………….……4

4. Installing the Required Dependencies…………………………….………………………………………….…………..5

5. Running the Server and Testing the Application….…………………………………………………………….……5

[Functionality 5](#_Toc119763569)

1. User authentication…………………………………………………………………………………………………………………6

2. Chat Rooms…………………………………………………………………………………………………………………………….9

3. Messages and Chat Consumers…………………………………………………………………………………………….10

4.Sending and Receiving Messages…………………………………………………………………………………………..12

5. Counting the Number of Users Online………………………………………………………………………………….14

6. The User Interface………………………………………………………………………………………………………………..15

[Using the Application 14](#_Toc119763570)

# Technical Overview

Chat Lounge is a web-based chat room application, built in Python using the Django framework.

Additionally, it uses Django’s Channels module, used for extending Django’s protocol capabilities beyond HTTP. In this case, it is used to handle long-running connections, between Django and the Web Sockets API in the JS script, on the client-side.

In order to communicate with multiple Web Socket connections simultaneously in a group-like setting, the Channel Layers setting has been enabled in the settings.py file.

A Redis server is used for caching the Channel Layers’ data in key-value pairs. The server runs locally, in a Docker container.

An SQLite3 database is used for storing the information about the Rooms, Users and Messages.

On the client-side, the application uses HTML for the structure,

Tailwind CSS for styling, and JavaScript for handling logic and Web Sockets.

The application uses the Asynchronous Server Gateway Interface, meaning it needs a web server, running in an asynchronous manner. For this, it depends on the Daphne web server, a popular choice for executing such tasks.

Finally, the project has been deployed to Heroku for hosting at: <https://django-chat-room-web-app.herokuapp.com/>. And uploaded to a GitHub repository at: <https://github.com/todorpopov/Chat-Room-Application/> . Please, keep in mind that the deployed version may not be up-and-running constantly!

# Structure of the application

Graphical user interface, text, application

Description automatically generated This is what the application’s directory looks like. The chat and room folders are called apps, and the chat\_project is the project folder. They are automatically generated by Django; they contain all the code needed for the app to run. The db.sqlite3 is the database file, and the manage.py file is used for bringing everything together. The requirements.txt file is used for installing the dependencies. The Dockerfile hasn’t been tested thoroughly enough to be used, yet. The Procfile is used by Heroku for managing the web server. Both files couldn’t be ignored inside the .gitignore file, because they have no extensions. Please ignore them for now.

# Local Installation

Installation overview:

* Create a new directory on your machine.
* Download the repository from GitHub and unzip it into this new directory.
* Edit the settings.py file inside the chat\_project directory.
* Decide on whether to use a local Redis server or the “InMemoryChannelLayer” setting.
* Install the dependencies, using the requirements.txt file.
* Run the web server and go to localhost.
  1. Setting up Locally.

Choose where you’d like to set up the project locally, create a new directory, and head to GitHub, following the link in the Technical Overview. There you can download the repository as a zipped file. Then you can unzip the folder inside the new directory and proceed with the next steps.

* 1. Editing the Project Settings

For the app to work correctly, you’ll need to edit a couple of variables inside the settings.py file, which can be found in chat\_project/settings.py.

First, set the downloaded\_from\_github variable to True. By doing this, the app will use a preset SECRET\_KEY, instead of looking for it in a file. You can manually set a different key on your own by following the comments.

The next setting that needs to be changed is the redis\_caching variable. When this variable is set to True, the app will try to connect to a Redis server at localhost and port 6379. This means that if no server is found, for the app to connect to, the messaging feature will not work. I will explain more about the Redis server in the next step.

* 1. Redis and “InMemoryChannelLayer”

There are two ways of enabling the use of the Channel Layers. First is to use the “InMemoryChannelLayer”, and the second is to use a separate Redis server, to which the app can connect to. I will explain both, but I personally recommend using the “InMemoryChannelLayer”, because of the easier setup.

* To use the “InMemoryChannelLayer”, you can just leave the redis\_caching variable set to False. The official documentation suggests that this is not recommended for production use, but for development and testing only.
* To use the “RedisChannelLayer”, you’ll have to set up a Redis server locally. The easiest way to do that is with Docker.

You’ll have to install the Docker desktop application and confirm it runs correctly by opening it. Also, make sure to enter:

docker

in an empty terminal in order to confirm that the CLI works correctly.

Then, open a terminal and enter the following command:

docker run -p 6379:6379 -d redis:5

This will create a container using the Redis image. Now set the redis\_caching variable, located in setting.py, to True.

* 1. Installing the Required Dependencies

Installing the dependencies can be done through the “requirements.txt” file. Open a terminal and enter the following command:

pip install -r requirements.txt

* 1. Running the Server and Testing the Application

To run the web server, open a terminal and use the cd command to enter the directory where the project is located. Once inside the project’s directory, you can run the following command:

daphne chat\_project.asgi:application

This will run the development server at your localhost (usually: 127.0.0.1:8000). Then, you can copy the displayed address into a browser, and the application should load.

# Functionality

1. User Authentication

Django has many built-in features, which can aid developers to make user

authentication easier and cleaner.

* Background pattern

  Description automatically generatedSign Up

This is what the sign-up page looks like.

Text

Description automatically generated

Text

Description automatically generatedThis is what the form.py file, inside the chat app, looks like. The sign-up form is defined here.

This is how the form gets processed, inside the views.py file. Once an HTTP request is made, signup will check whether it is a “POST” request or different. If it is, the form is validated using the built in is\_valid() function, a new user gets created, saved and logged in with, then the client gets redirected to the frontpage, logged in with the details they created.

* Background pattern

  Description automatically generatedLog In and Log Out

The log in window looks like this. Its functionality and the functionality

A screenshot of a computer

Description automatically generated with low confidencebehind the log out are built-in in Django and can be set in the settings.py.

Text

Description automatically generatedDefining the URL redirect.

In the urls.py, the app uses the built-in class views, and its LoginView and LogoutView methods to log users in and out automatically.

1. Graphical user interface, application

   Description automatically generatedChat Rooms

Text

Description automatically generatedWhen users enter the Rooms page, they are presented with the following page.

The list of rooms is loaded with the following code. It uses a Python for

loop to dynamically load all of the saved rooms.

Text

Description automatically generated

Rooms are saved to the database using models. The Room model is

Text

Description automatically generateddefined as follows. Each room has a name, a slug, and number of online users. Currently, only the superuser can create, edit and delete rooms.

This is the views.py file, located in the room app, where all the

rooms’ data is queried from the database and sent to the client-side to be loaded.

1. Messages and Chat Consumers

* Text

  Description automatically generatedMessages

Messages are saved to the database using the following model in the

models.py file. They are sorted by date/time added, so they can be loaded in the correct order afterwards.

* Text

  Description automatically generatedChat Consumers

In order for the Django Channels to work correctly, the app needs to have a chat consumer class. This class is defined in the consumers.py file, inside the room app. The ChatConsumer has 4 asynchronous and 3 synchronous methods. The sync ones have to be called from within the async ones, so to allow that, the @sync\_to\_async decorator is used before defining them.

The connect method, adds the chat consumer to the channel layer group with the room name and increments the number of online users by 1.

The “disconnect” method works in the same manner, except it discards the chat consumer from the “channel layer group” and decrements the online users by 1.

The receive takes a text message as a parameter, creates a Message instance, saves it to the database, and sends it for the rest of the channel layer members in the group to see.

The chat\_message method sends the text messages from the chat consumer, to the client-side.

1. Sending and Receiving Messages

Now, that the server-side message handling has been explained, let’s see

how messages are sent and received for the users to see on the client-side.

* Web Sockets API

Web Sockets are used on the client side to establish a long running

connection to the server-side. They are used in the JavaScript located in the room.html file inside the room app.



Graphical user interface, text

Description automatically generatedInitiating a new Web Socket looks like this. Web Sockets take one parameter – the URL, at which they are to expect a connection at.

To set the URL at which the Web Sockets are expected to connect to, in

the backend, the app uses a routing.py file, located inside the room app.

* Sending the users’ messages to Django

Now, that the Web Sockets’ routing has been specified, the application

can communicate with them.

Text

Description automatically generatedThis is the JavaScript function that handles the sending of users’ messages to Django. Once the user submits their message, it gets converted to a JSON, along with the username and the name of the chat room. If the message is not empty, it is sent to the server-side, else, the user is alerted. After sending the message, the function resets the message field’s content for the next message. The e.preventDefault() is used for preventing the user’s CSRF token from being reset.

Graphical user interface, text

Description automatically generatedOnce a message gets sent from the client to the server, the ChatConsumer’s receive method gets called. It saves the message to the database, as a Message instance, and sends it to the channel layer group, for other chat consumers to receive.

When a new user enters the room, or when a user reloads their page, messages need to be sent from Django, and loaded onto the page. This happens by calling the chat\_message method of the ChatConsumer class. It sends a JSON, containing a message, username, room name, and current time. Once sent to the client-side. The JavaScript code snippet above, takes the data, parses it into a variable, checks if the message is empty – if it isn’t, it gets appended to the DOM, otherwise, the user gets alerted (this shouldn’t be able to happen, but is used just in case).

1. Counting the Number of Users Online

There are multiple ways to count the number of active connections

within a chat room, including one where the “InMemoryChannelLayer” is used as additional cache, parallel to the Redis caching server.

Text

Description automatically generated For the sake of simplicity, the application increments an integer every time a new connection, to the channel layer has been made. Logically, it gets decremented every time a connection is discarded.

Text

Description automatically generatedAs mentioned before, this is what the Room model looks like. Active users are saved to the users\_online field.

Once a new user is connected to the channel layer, the synchronous increment\_users\_online method is called. It creates a new Room object, where the slug is the same as the room\_name, used for connecting and disconnecting users, within the connect and disconnect methods of the ChatConsumer class. Then the room’s users\_online field is incremented by 1, and the data is saved to the database.

The decrement\_users\_online method is almost the same, the only difference being that it decrements the field by 1.

1. The User Interface

The UI has been built using Tailwind CSS, a modern CSS framework, and

Flowbite (link: <https://flowbite.com/>), an online tool for open-source Tailwind components.

The application uses the Tailwind CDN, meaning there is no installation needed for the framework to work. This is NOT the recommended way for production use, because the browser has to load many unnecessary classes, thus making the application slower. Even though, this is not optimal for production, it is suitable for local development and testing.

# Using the Application

Currently, it has only been optimized for large screens. Opening it on a phone may not work properly!

The application has been designed to be as intuitive as possible, with simple and professional-looking UI. Follow these steps to get started:

* Once the application in a browser
* Sign Up for a new user or Log In into an existing one
* Enter a chat room
* Enter a text message into the filed below and press Submit
* You can change the chat room by pressing the Rooms button, in the navigation bar
* To log out, simply press the Log Out button, located in the navigation bar

Also, the application has an admin superuser set up. You can access the

admin panel by adding “/admin” to the base URL. Then, you can use these details to log in:

username: admin

password: superuserpass1234

From there, you can create and delete different rooms, users, and messages.