**Project structure**

**Project consists of two parts: django and express. Let’s start by talking about the django part. Basically inside django part we have three apps:**

1. **chat- it includes starting page and info endpoint, it also serves as entry point for the entire app**
2. **log_reg- this part is responsible for logging and registration**
3. **main- the biggest app, it includes chat between users, deleting accounts(teacher only), modifying accounts, viewing all accounts info(teacher only).**

**express part:**

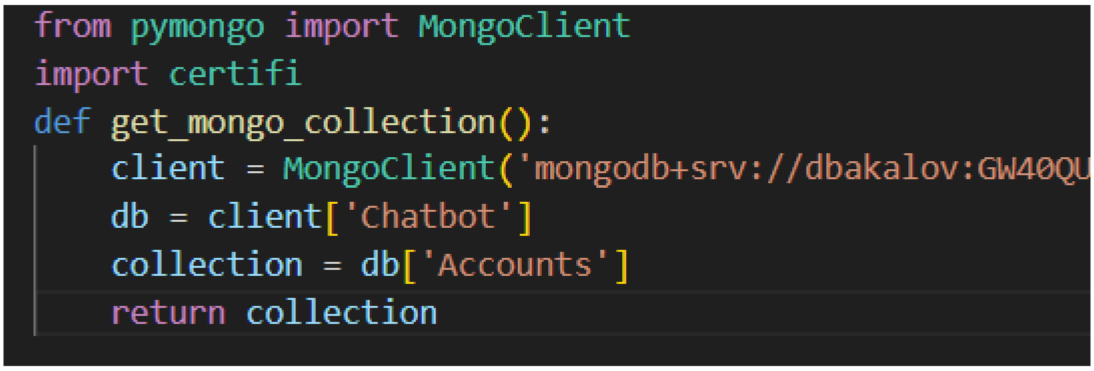
**It is only used as a microservice. The purpose of this part is only to send information about students to the django part**

**Launching app:**

**In order to launch the app you need to run django app, redis server([https://redis.io/docs/install/install-redis/install-redis-on-windows/](https://redis.io/docs/install/install-redis/install-redis-on-windows/)), in order to have access to all students info(microservice) you also need to run express project**

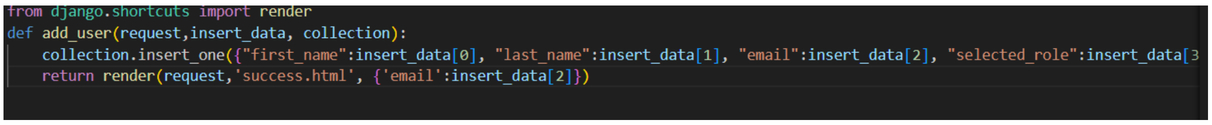
# Guide for future devs

In this file let’s talk about important parts of the project:



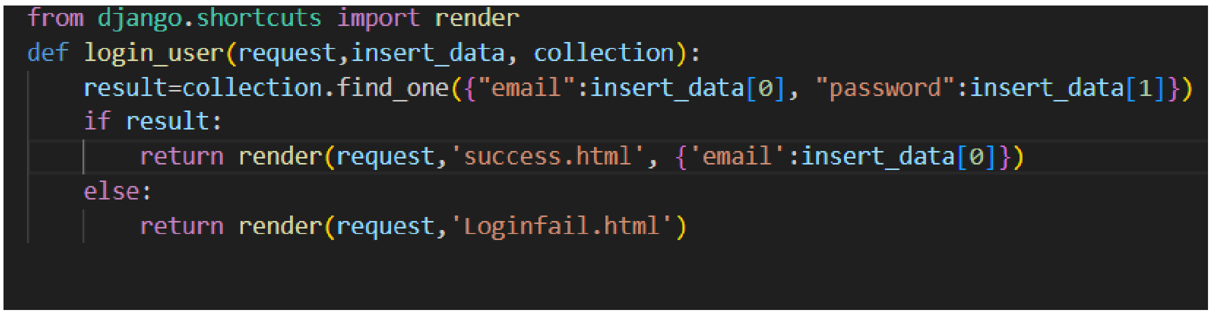
```python
from pymongo import MongoClient
import certifi
def get_mongo_collection():
    client = MongoClient('mongodb+srv://dbakalov:GW40QU
    db = client['Chatbot']
    collection = db['Accounts']
    return collection
```

This python code just returns the collection of accounts, so you can manipulate the accounts of the users



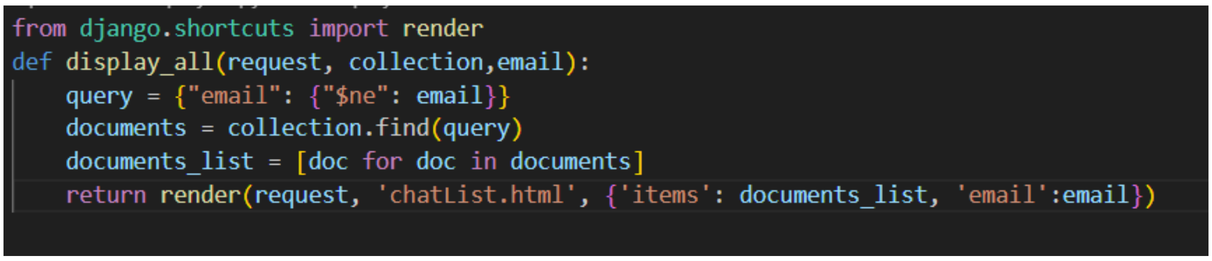
```python
from django.shortcuts import render
def add_user(request,insert_data, collection):
    collection.insert_one({"first_name":insert_data[0], "last_name":insert_data[1], "email":insert_data[2], "selected_role":insert_data[3
    return render(request,'success.html', {'email':insert_data[2]})
```

This code creates new object in accounts based on the insert_data parameter



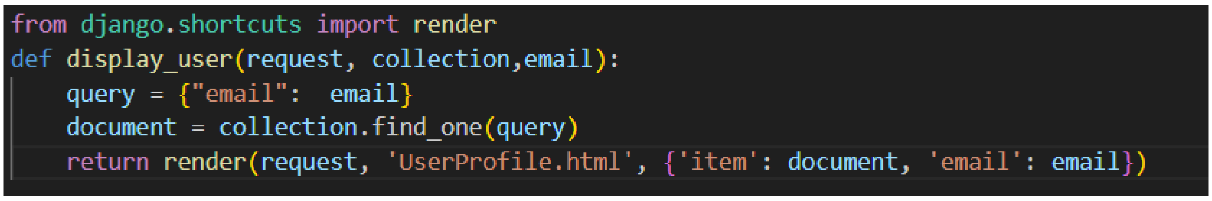
```python
from django.shortcuts import render
def login_user(request,insert_data, collection):
    result=collection.find_one({"email":insert_data[0], "password":insert_data[1]})
    if result:
        return render(request,'success.html', {'email':insert_data[0]})
    else:
        return render(request,'Loginfail.html')
```

This function serves for logging the user , if it can’t find user in db, it just returns fail page



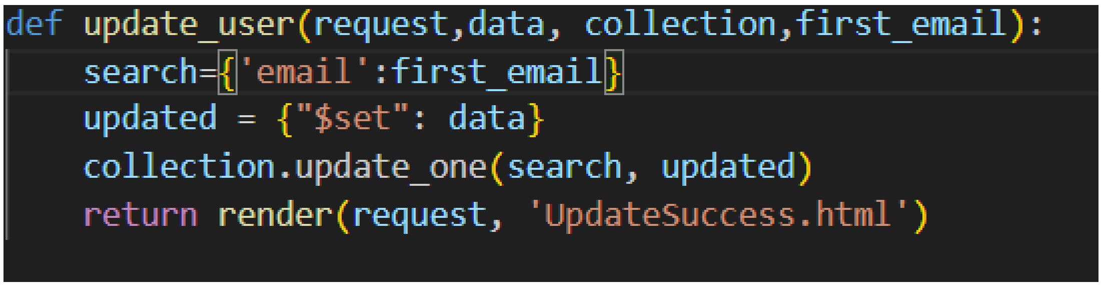
```python
from django.shortcuts import render
def display_all(request, collection,email):
    query = {"email": {"$ne": email}}
    documents = collection.find(query)
    documents_list = [doc for doc in documents]
    return render(request, 'chatList.html', {'items': documents_list, 'email':email})
```

This code is responsible for getting all available for chatting users except the user who accesses the chat, so you couldn’t chat with yourself



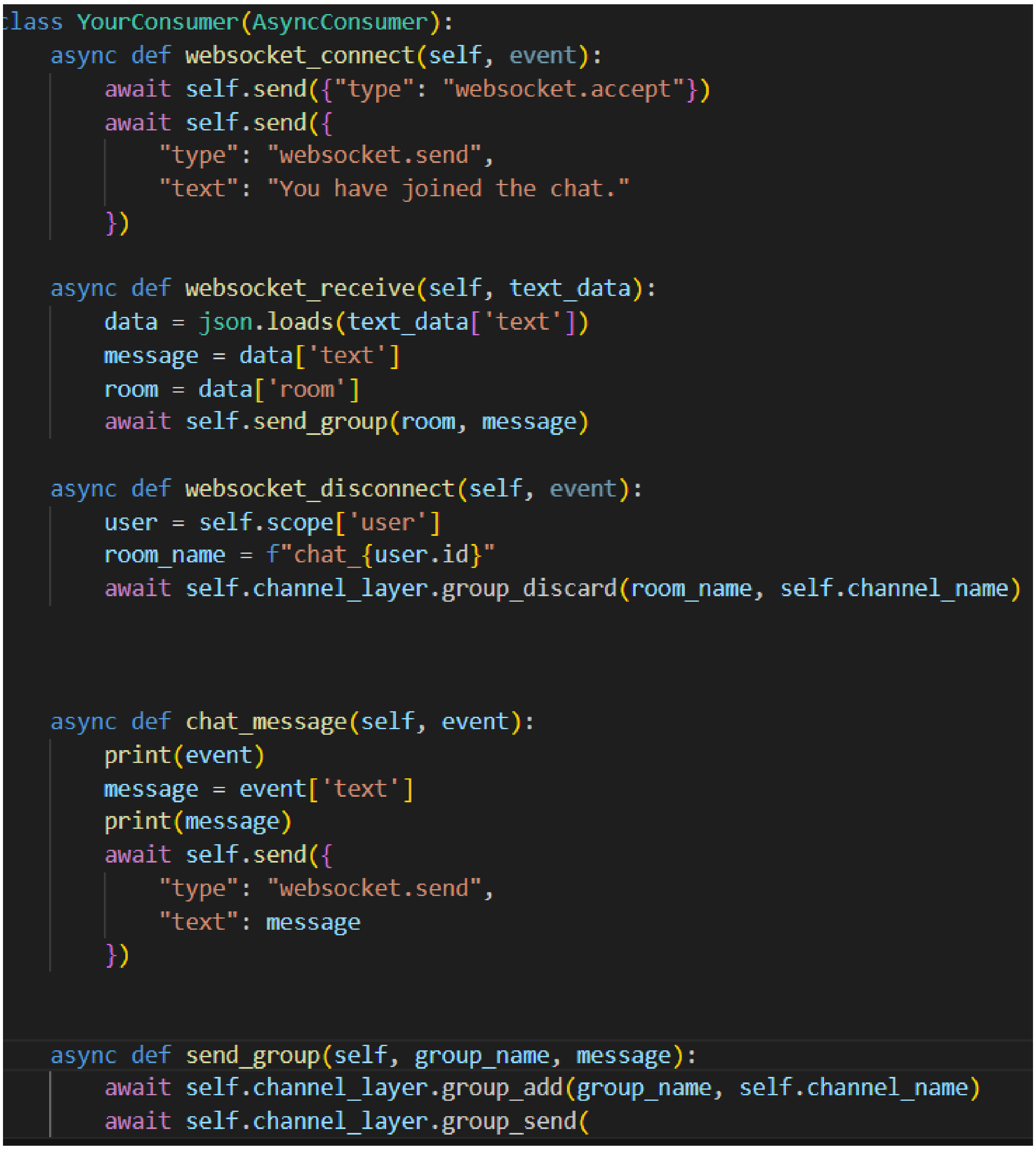
```python
from django.shortcuts import render
def display_user(request, collection,email):
    query = {"email":  email}
    document = collection.find_one(query)
    return render(request, 'UserProfile.html', {'item': document, 'email': email})
```

This code just gets user based on his email and and gives all the data about him



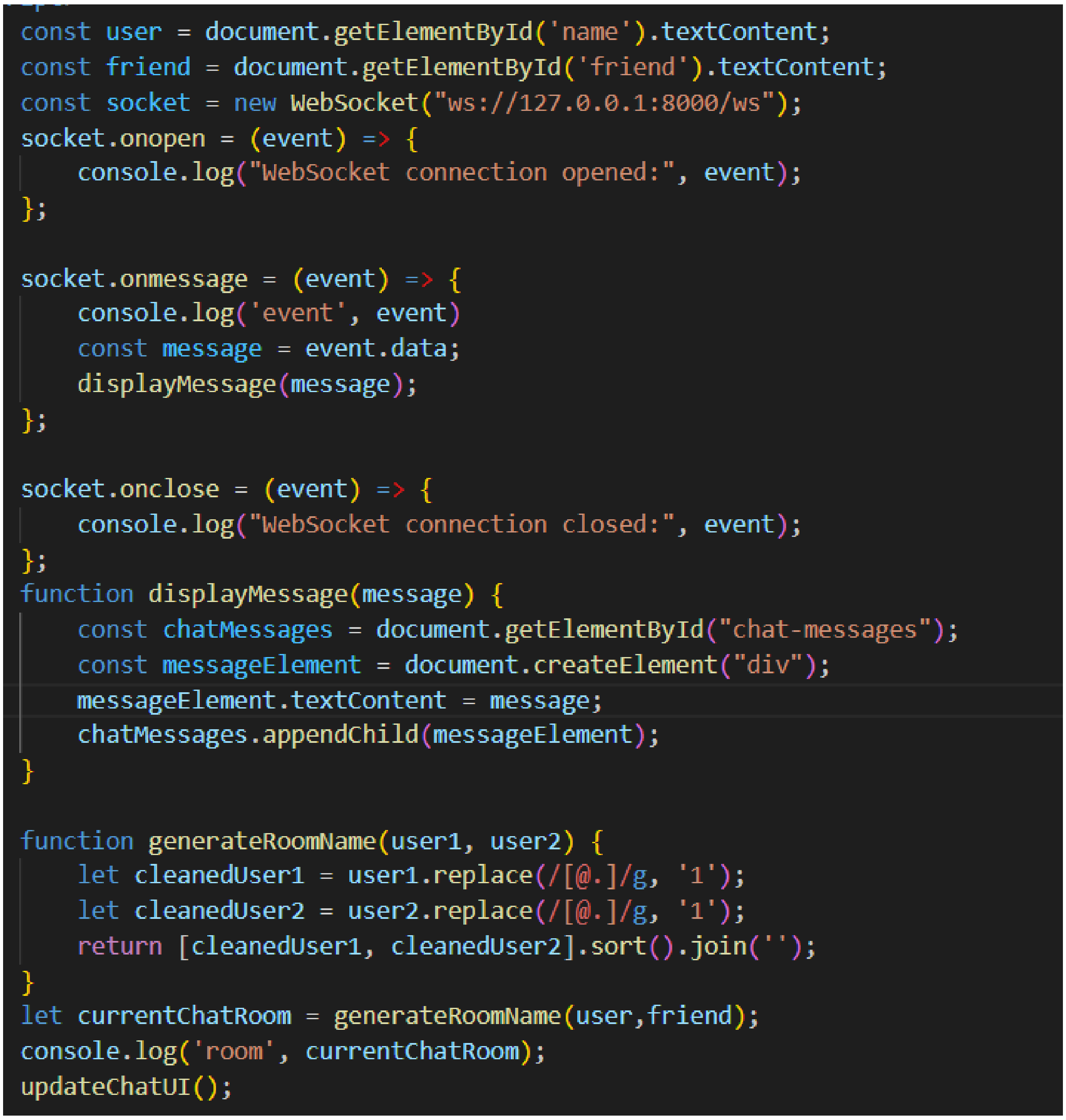
```python
def update_user(request,data, collection,first_email):
    search={'email':first_email}
    updated = {"$set": data}
    collection.update_one(search, updated)
    return render(request, 'UpdateSuccess.html')
```

This function updates the user based on the data parameter

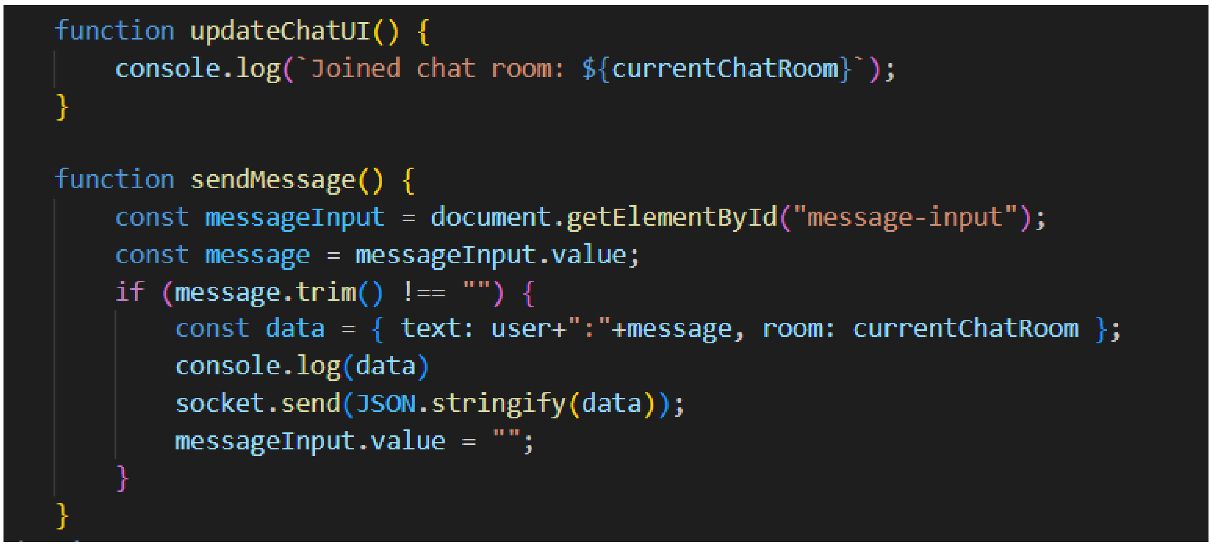


```python
lass YourConsumer(AsyncConsumer):
    async def websocket_connect(self, event):
        await self.send({"type": "websocket.accept"})
        await self.send({
            "type": "websocket.send",
            "text": "You have joined the chat."
        })

    async def websocket_receive(self, text_data):
        data = json.loads(text_data['text'])
        message = data['text']
        room = data['room']
        await self.send_group(room, message)

    async def websocket_disconnect(self, event):
        user = self.scope['user']
        room_name = f"chat_{user.id}"
        await self.channel_layer.group_discard(room_name, self.channel_name)



    async def chat_message(self, event):
        print(event)
        message = event['text']
        print(message)
        await self.send({
            "type": "websocket.send",
            "text": message
        })


    async def send_group(self, group_name, message):
        await self.channel_layer.group_add(group_name, self.channel_name)
        await self.channel_layer.group_send(
```

Here is the consumer for this project .
As we can see it when it receives text it finds message and room info to broadcast it to other users

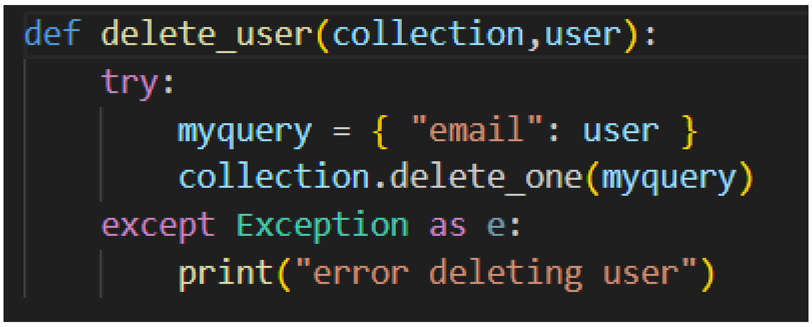


```
const user = document.getElementById('name').textContent;
const friend = document.getElementById('friend').textContent;
const socket = new WebSocket("ws://127.0.0.1:8000/ws");
socket.onopen = (event) => {
    console.log("WebSocket connection opened:", event);
};

socket.onmessage = (event) => {
    console.log('event', event)
    const message = event.data;
    displayMessage(message);
};

socket.onclose = (event) => {
    console.log("WebSocket connection closed:", event);
};
function displayMessage(message) {
    const chatMessages = document.getElementById("chat-messages");
    const messageElement = document.createElement("div");
    messageElement.textContent = message;
    chatMessages.appendChild(messageElement);
}

function generateRoomName(user1, user2) {
    let cleanedUser1 = user1.replace(/[@.]/g, '1');
    let cleanedUser2 = user2.replace(/[@.]/g, '1');
    return [cleanedUser1, cleanedUser2].sort().join('');
}
let currentChatRoom = generateRoomName(user,friend);
console.log('room', currentChatRoom);
updateChatUI();
```



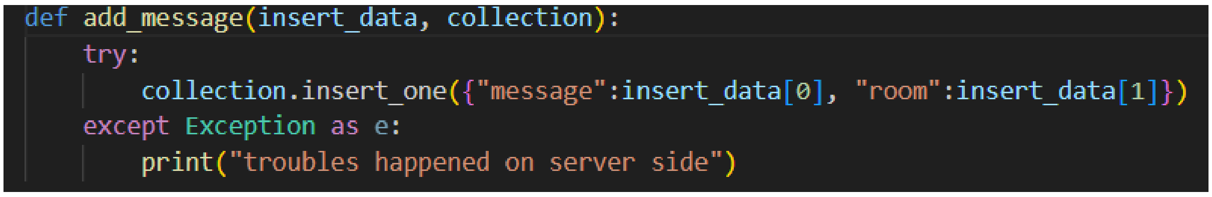
```
function updateChatUI() {
    console.log(`Joined chat room: ${currentChatRoom}`);
}

function sendMessage() {
    const messageInput = document.getElementById("message-input");
    const message = messageInput.value;
    if (message.trim() !== "") {
        const data = { text: user+":"+message, room: currentChatRoom };
        console.log(data)
        socket.send(JSON.stringify(data));
        messageInput.value = "";
    }
}
```

This is the js code which is responsible for sending messages to the consumer so it could broadcast them further.



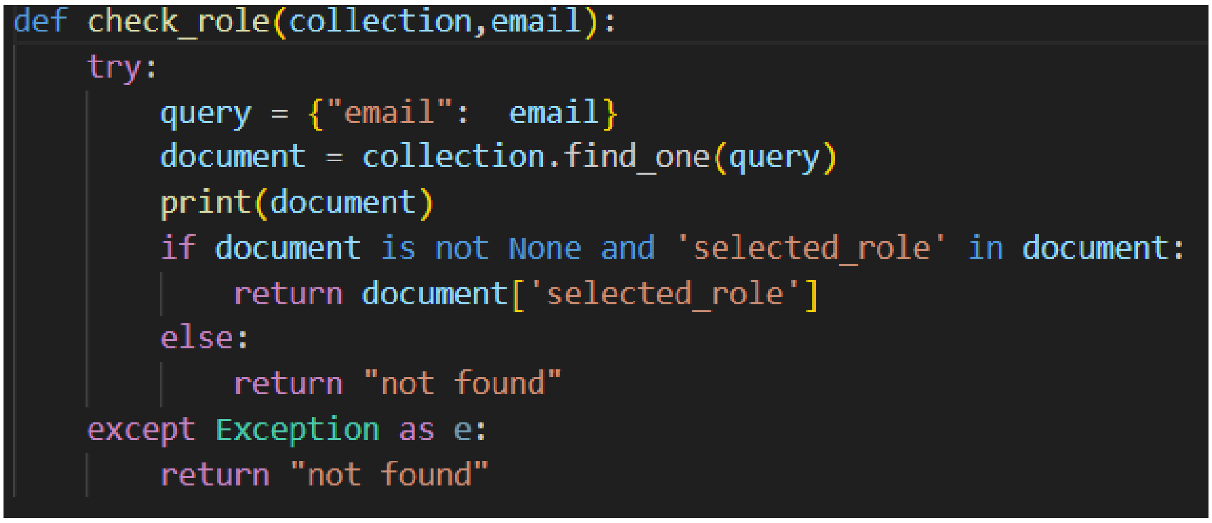
```python
def delete_user(collection,user):
    try:
        myquery = { "email": user }
        collection.delete_one(myquery)
    except Exception as e:
        print("error deleting user")
```

This code is responsible for deleting a user by his email



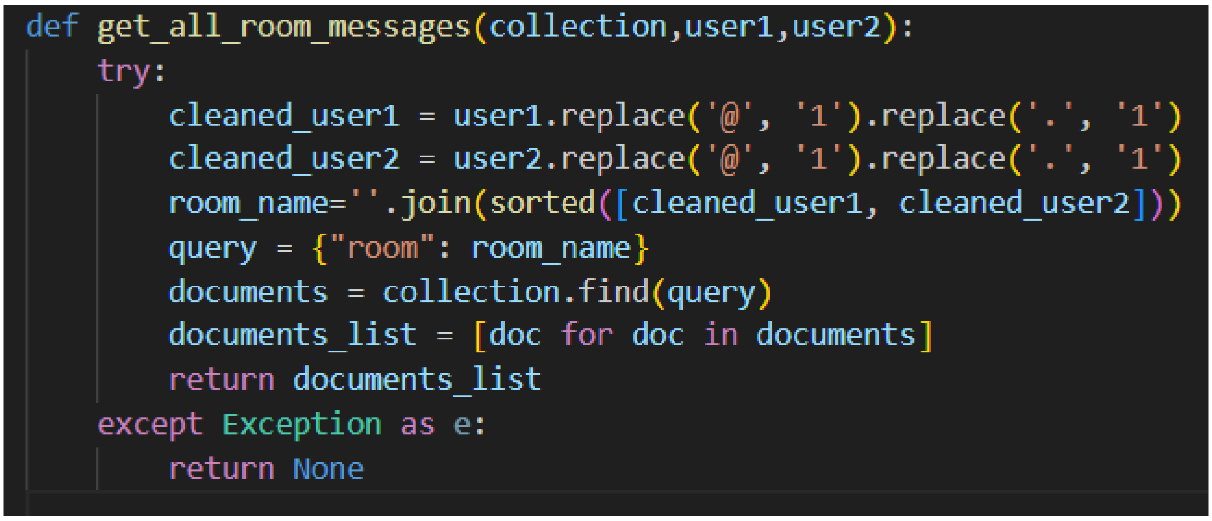
```python
def add_message(insert_data, collection):
    try:
        collection.insert_one({"message":insert_data[0], "room":insert_data[1]})
    except Exception as e:
        print("troubles happened on server side")
```

This code adds message to the db



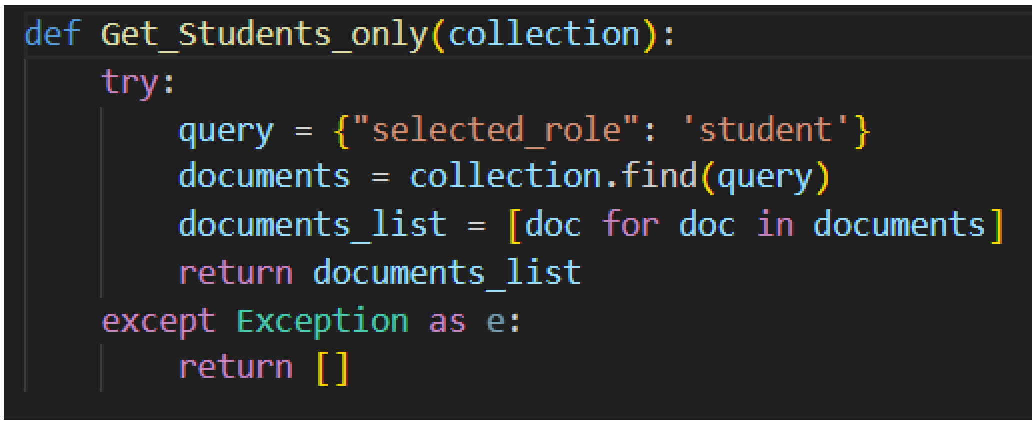
```python
def check_role(collection,email):
    try:
        query = {"email":  email}
        document = collection.find_one(query)
        print(document)
        if document is not None and 'selected_role' in document:
            return document['selected_role']
        else:
            return "not found"
    except Exception as e:
        return "not found"
```

this code check role of the user by his email



```python
def get_all_room_messages(collection,user1,user2):
    try:
        cleaned_user1 = user1.replace('@', '1').replace('.', '1')
        cleaned_user2 = user2.replace('@', '1').replace('.', '1')
        room_name=''.join(sorted([cleaned_user1, cleaned_user2]))
        query = {"room": room_name}
        documents = collection.find(query)
        documents_list = [doc for doc in documents]
        return documents_list
    except Exception as e:
        return None
```

This code retrieves all messages from db

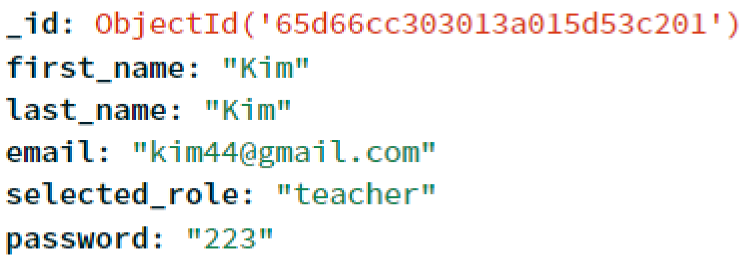


```python
def Get_Students_only(collection):
    try:
        query = {"selected_role": 'student'}
        documents = collection.find(query)
        documents_list = [doc for doc in documents]
        return documents_list
    except Exception as e:
        return []
```

this code returns only users with student role

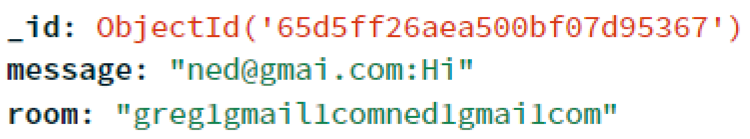
**Models examples:**

user example:



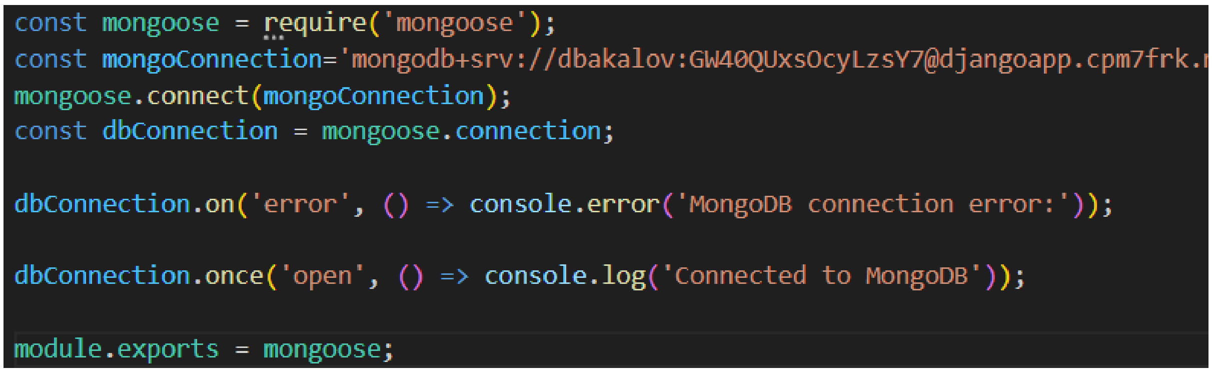
```
_id: ObjectId('65d66cc303013a015d53c201')
first_name: "Kim"
last_name: "Kim"
email: "kim44@gmail.com"
selected_role: "teacher"
password: "223"
```

message example:



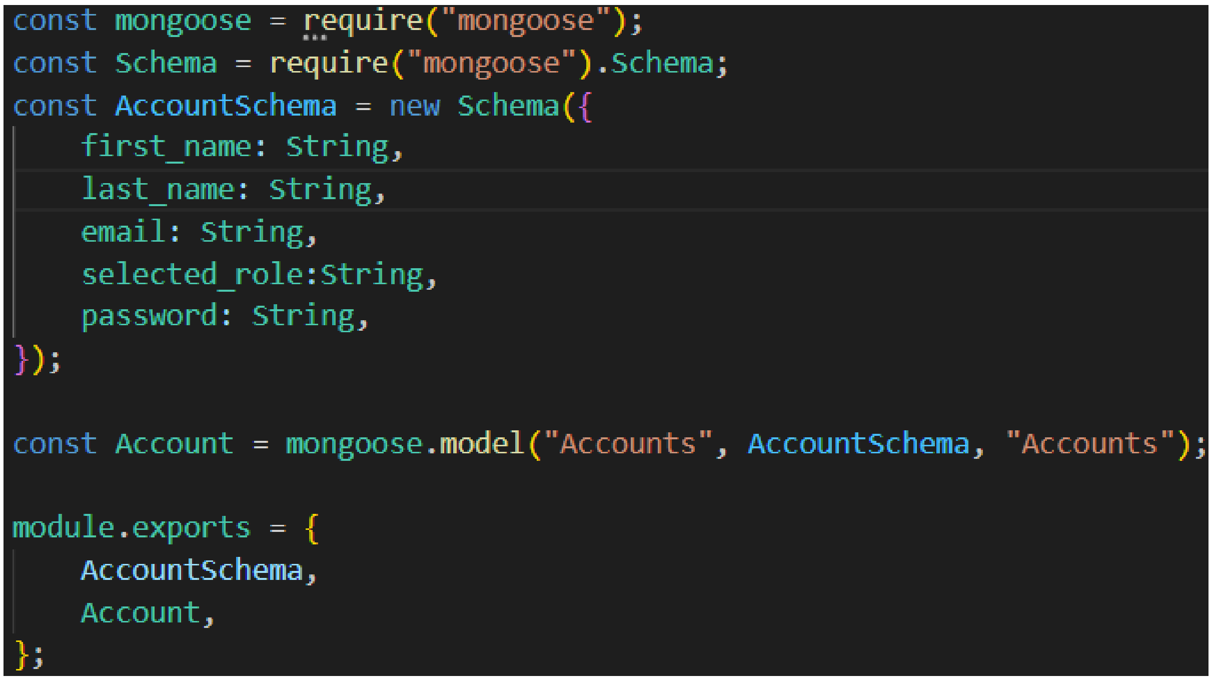
```
_id: ObjectId('65d5ff26aea500bf07d95367')
message: "ned@gmai.com:Hi"
room: "greg1gmail1comned1gmail1com"
```

## Microservice db integration:



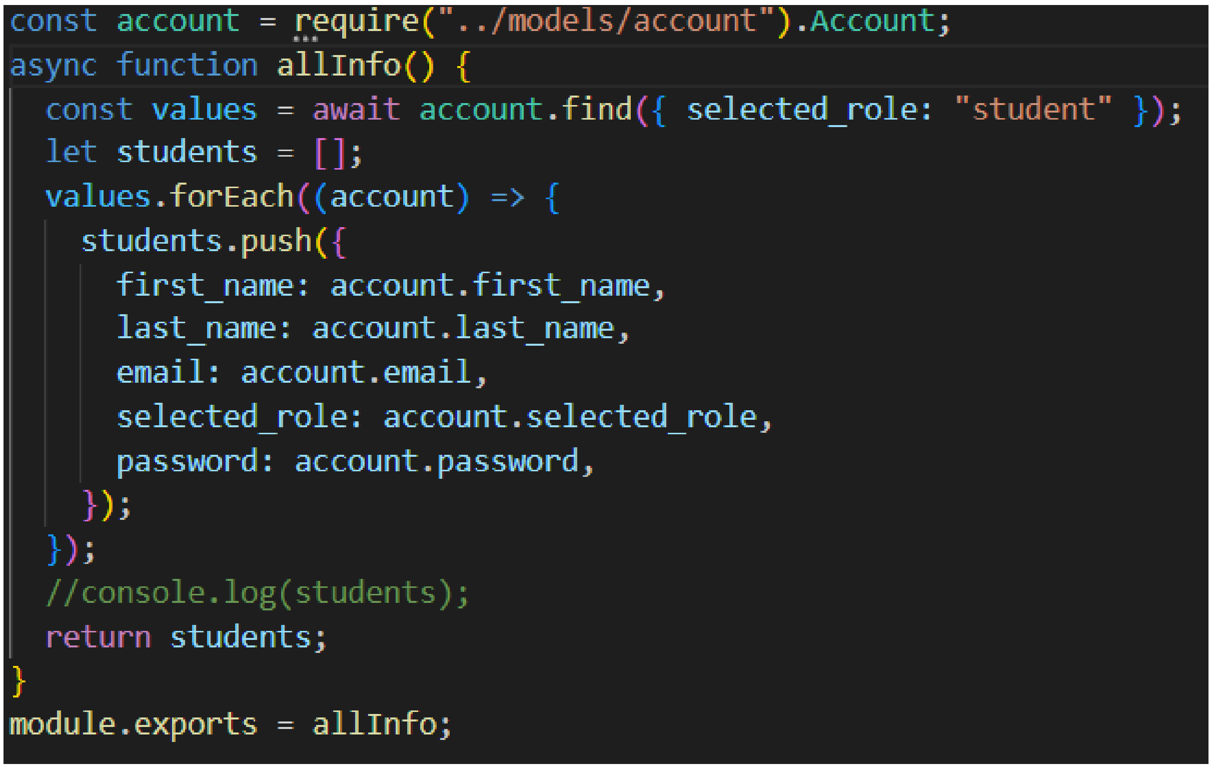
```
const mongoose = require('mongoose');
const mongoConnection='mongodb+srv://dbakalov:GW40QUxsOcyLzsY7@djangoapp.cpm7frk.
mongoose.connect(mongoConnection);
const dbConnection = mongoose.connection;

dbConnection.on('error', () => console.error('MongoDB connection error:'));

dbConnection.once('open', () => console.log('Connected to MongoDB'));

module.exports = mongoose;
```

mongo connection



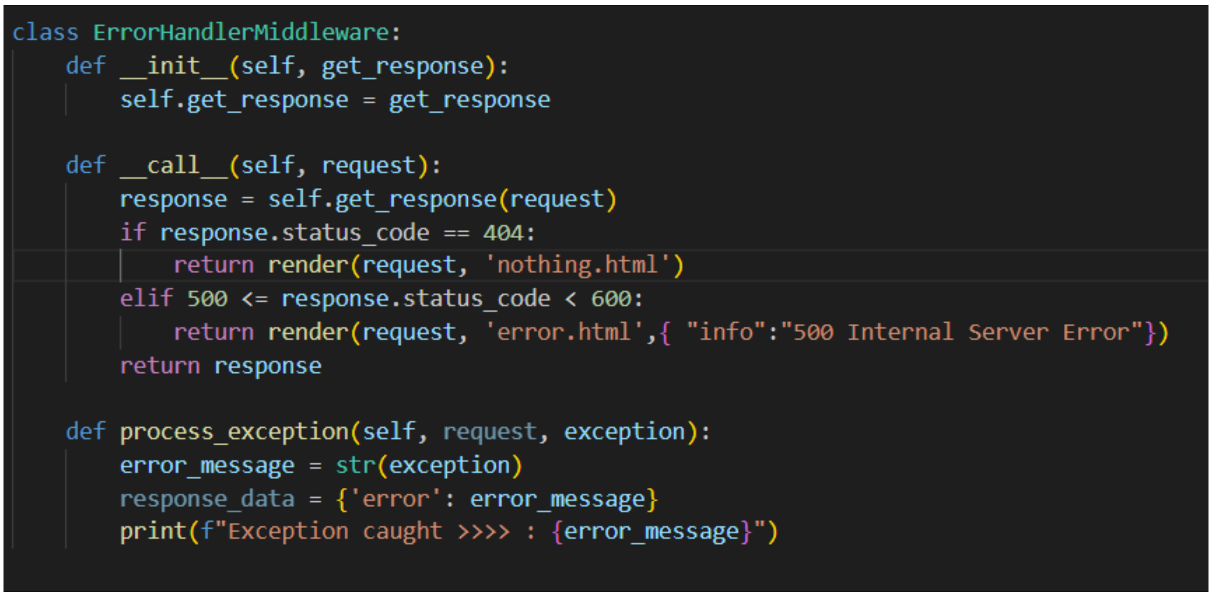
```
const mongoose = require("mongoose");
const Schema = require("mongoose").Schema;
const AccountSchema = new Schema({
    first_name: String,
    last_name: String,
    email: String,
    selected_role:String,
    password: String,
});

const Account = mongoose.model("Accounts", AccountSchema, "Accounts");

module.exports = {
    AccountSchema,
    Account,
};
```

creating user schema



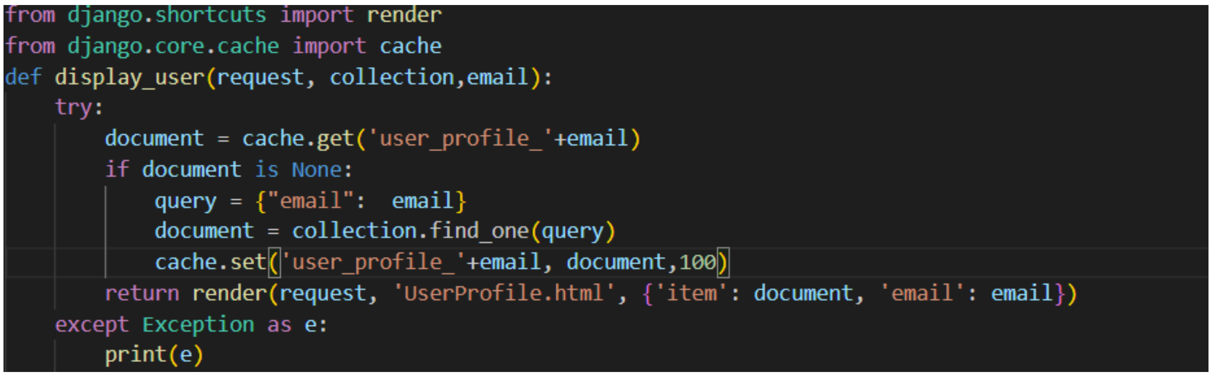
```
const account = require("../models/account").Account;
async function allInfo() {
  const values = await account.find({ selected_role: "student" });
  let students = [];
  values.forEach((account) => {
    students.push({
      first_name: account.first_name,
      last_name: account.last_name,
      email: account.email,
      selected_role: account.selected_role,
      password: account.password,
    });
  });
  //console.log(students);
  return students;
}
module.exports = allInfo;
```

getting all data about students



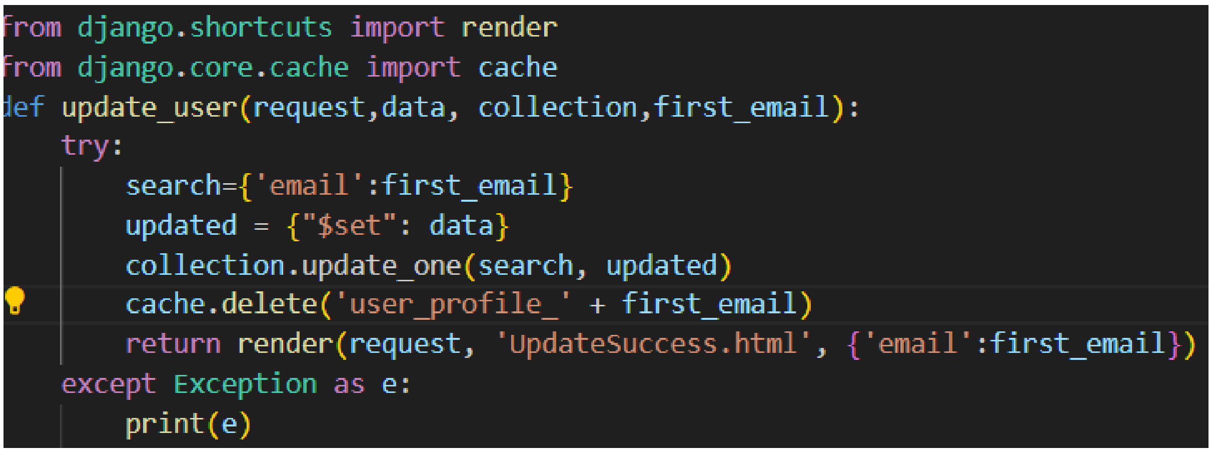
```
class ErrorHandlerMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        response = self.get_response(request)
        if response.status_code == 404:
            return render(request, 'nothing.html')
        elif 500 <= response.status_code < 600:
            return render(request, 'error.html',{ "info":"500 Internal Server Error"})
        return response

    def process_exception(self, request, exception):
        error_message = str(exception)
        response_data = {'error': error_message}
        print(f"Exception caught >>>> : {error_message}")
```

this class is responsible for sending error pages to the user in case of any error



```python
from django.shortcuts import render
from django.core.cache import cache
def display_user(request, collection,email):
    try:
        document = cache.get('user_profile_'+email)
        if document is None:
            query = {"email":  email}
            document = collection.find_one(query)
            cache.set('user_profile_'+email, document,100)
        return render(request, 'UserProfile.html', {'item': document, 'email': email})
    except Exception as e:
        print(e)
```

This helper was enhanced to use cached information instead of getting data from mongo db, which much faster



```python
rom django.shortcuts import render
rom django.core.cache import cache
ef update_user(request,data, collection,first_email):
    try:
        search={'email':first_email}
        updated = {"$set": data}
        collection.update_one(search, updated)
        cache.delete('user_profile_' + first_email)
        return render(request, 'UpdateSuccess.html', {'email':first_email})
    except Exception as e:
        print(e)
```

however when user modifies an account, then cached information will be deleted, so there will be no issues with updating data on the template

!!! Attention, in the github most of the provided code will have try except statements, however the logic is still the same !!!