Izrada aplikacije za chat

Matejčić, Ivan

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:215025

Rights / Prava: In copyright/Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-18**

Sveučilište u Rijeci
FIDIT
Fakultet informatike i digitalnih tehnologija

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Informatics and Digital Technologies - INFORI Repository

zir.nsk.hr
UNIRI DIGITALNA KNJIŽNICA
dabar DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

Sveučilište u Rijeci – Fakultet informatike i digitalnih tehnologija

Sveučilišni preddiplomski studij informatike

Ivan Matejčić

Creating a chat application

Završni rad

Mentor: Doc. Dr. sc. Lucia Načinović Prskalo

Rijeka, 12.rujna 2022.

FAKULTET INFORMATIKE I DIGITALNIH TEHNOLOGIJA
SVEUČILIŠTE U RIJECI

Rijeka, 25.4.2022.

Zadatak za završni rad

Pristupnik: Ivan Matejčić

Naziv završnog rada: Izrada aplikacije za chat

Naziv završnog rada na eng. jeziku: Creating a chat application

Sadržaj zadatka: Zadatak ovog završnog rada je izraditi web aplikaciju za razgovor u koju se korisnik može prijaviti, odabrati sobu za razgovor te komunicirati s ostalim korisnicima aplikacije. U radu će taođer biti opisani svi alati koji su se koristili tijekom izrade te prikazane I opisane funkcionalnosti izrađene aplikacij

Mentor

Doc. dr. sc. Lucia Načinović Prskalo

Načinović Prskalo

____

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

[illegible]

____

Zadatak preuzet: 27.4.2022.

Ivan Matejčić
____
(potpis pristupnika)

Sažetak

Django je softver otvorenog koda pomoću kojeg je, uz korištenje programskog jezika Python, moguće jednostavno i brzo kreirati web aplikaciju. Django služi kako autor ne bi trebao izgraditi novu aplikaciju iz korijena, nego se posvetiti upravljanju backend-a i izradi frontend dizajna. Dolazi sa ugradbenim komponentama (kao npr. sustav za prijavu, sustav baze podataka itd.) pomoću kojih se autor može više skoncentrirati na izradu same aplikacije nego potrebnim dodatcima.
Cilj ovog završnog rada je iskoristiti Django i njegove prednosti pri kreiranju web aplikacije kako bi kreirali aplikaciju za chat; aplikaciju u koju se korisnik može prijaviti, odabrati sobu za chat (iliti eng. ‘*room*’ ili eng. ‘*channe*l’) te komunicirati s ostalim korisnicima aplikacije.

Ključne riječi

Django, chat, application, room, channel, Django Channels, WebSocket, template, Python.

Table of contents

1. Introduction

A chat application is an application which allows multiple users to communicate by sending and receiving text messages over the internet.
Chat applications date back to the 1980s when Compuserve released the first dedicated online chat service, CB simulator. Later, services like Skype, MSN Messenger, Facebook Messenger are more are some of the most popular chat services used throughout the years.
Today, WhatsApp is the most used chat application in the world [18].
As chat applications evolved, they became more sophisticated. Nowadays, chat applications also have features like being able to make calls (voice-only or even video calls), send voice messages, post stories (ex. WhatsApp and Facebook Messenger) and much more.
Chatting is a very common and useful feature for applications to have. Games, streaming services, technical support services, social media websites and many more services all incorporate chatting in one way or another.

This thesis consists of six main chapters: Introduction, Premise, Djat, Creating the application, Conclusion and Literature.
In the second chapter a general description of the chat application is given, what tools were used to create the application, project structure and a description of tools used to enable the chat aspect of the application itself.
The third chapter explains the application's UI and the fourth chapter goes through the code of the project to give further insight into how the application was made.

1.1 Django – what is it?

Django [12] is a web framework for the programming language Python, which means it supports development of web applications by providing built-in libraries and resources. This relieves the developer of having to create the web application from scratch which is useful when creating a web application as fast and reliable as possible.
Moreover, it controls how information within the web application itself flows as well as a template environment which helps display the information on screen, a fully functional database for safely storing the information, high-level security to ensure a safe web application experience and many more.
Django was created by two developers, Adrian Holovaty and Simon Willison, in 2003 and initially released in July 2005 [16].

1.2 Structure of a Django application

Web frameworks are based on a framework architecture. A framework architecture dictates how an app using a web framework works [2]. Django uses a Model-View-Template framework (shown in Figure 1) architecture which consists of a Model, View and a Template [15].

A Model consists of data which the developer wants to use in the application. This data is most commonly stored in a database (by default, Django uses SQL) and is the content which is to be displayed on screen. Models are located in the 'models.py' file.

A View is a Python function/request handler which, depending on the user's request, returns equivalent HTTP responses, templates and content [3].
These views are executed using a corresponding URL connected to the view.
Views are located in the 'views.py' file.

A Template is a file (usually an .HTML file) which displays all the content and data as well as information on how to display it.
Templates are located in the 'templates' folder.

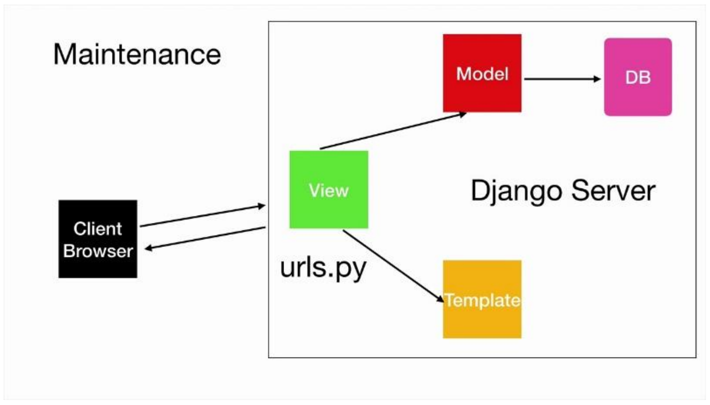


Figure 1: Model-View-Template framework architecture

1.3 How Django works

When a developer first creates a Django application and a web URL is requested by the browser, Django receives the URL and looks into the ‘urls.py’ file to find the matching view for that exact URL. Once it has done that, the equivalent model for that view is found within the ‘views.py’ file and its models are imported into it from the database. The view then sends the model information to a specific template located in the aforementioned ‘templates’ folder [12].
This is all being done inside a Django server which can either be ASGI (Asynchronous Server Gateway Interface) or WSGI (Web Server Gateway Interface) [13]. The Django chat application created for this project uses an ASGI server.
Finally, the template returns its content and data back to the browser, which is then displayed on the screen, for the user to interact with (shown in Figure 2) [14].

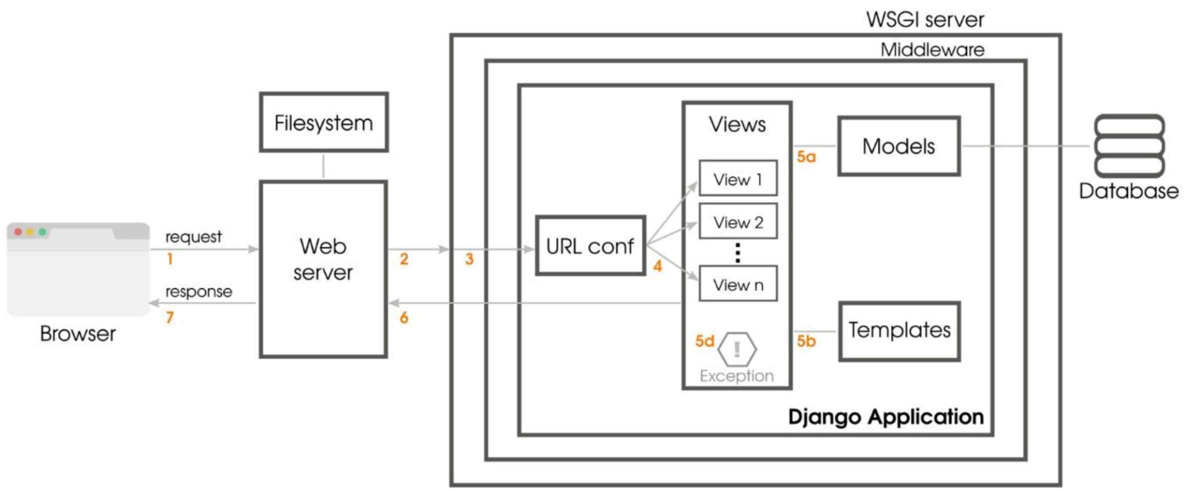


Figure 2: How Django works.

2. Premise

The application created for this project is called ‘Djat’ and is a Django chat application with a backend built with Python and a frontend using JavaScript, HTML and CSS.
This thesis consist of several parts outlining the UI and the features of the application, as well as the creation process.
In Order to have a functioning chat feature, the project uses Django Channels.
Also, this project consists of two Django applications: *room* and *chat* (shown in Figure 3). *Djat* is the main application to which the other two applications are connected (this application contains the ‘*settings.py’* file). The other two applications, *chat* and *room*, were created separately to achieve better project organization; the *chat* application contains views, URL’s and templates related to the aspect of the web application that does not include the chat functionality itself, so the log in page, sign up page and frontpage.
On the other hand, the *room* app contains views, URL’s and templates related to the chat aspect of the web application. This includes the room selection page and the chat page itself.

2.1 Django Channels

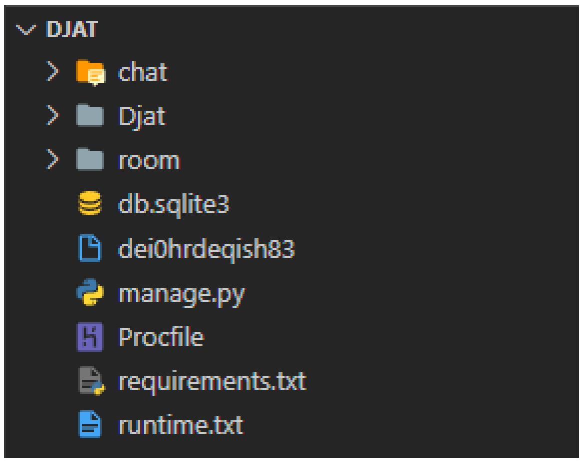
Django Channels is a Django project which extends Django’s abilities further from only HTTP, allowing it to be able to handle WebSocket’s, IoT, chat protocols and more. It does this by wrapping Django’s native asynchronous view support, all the while still preserving Django’s synchronousness and ease of use [5].
For this project, WebSocket’s and chat protocols are used.
A WebSocket is a bidirectional communication protocol used mainly because of its higher speed compared to HTTP [17].
Channels is based on an ASGI specification. For this reason, the web application uses an ASGI server rather than WSGI (see above). It also uses a special server called Daphne, which is used when deploying Django chat applications to Heroku [5].

2.1.1 How Django Channels Works

A Django Channels connection is split into two parts: a *scope* and a series of *events*.
A scope is a collection of details of an incoming connection (web request path, WebSocket IP address etc.) and an event is triggered by the user when an HTTP request or a WebSocket frame is thrown.
For HTTP, a scope lasts once per request, but for a WebSocket a scope last for the lifetime of the WebSocket itself. Events occur during the lifetime of a scope.

For example, when a user sends a message in the chat application, a scope is opened containing the user’s username and ID. After that, the app is given a *chat.recieved_message* event with the event text. Every time the user sends a message a new event is created. The scope is closed after a timeout or if the application itself is restarted.

The main component of Django Channels is a *consumer [6]*. A consumer is an event consumer; when a new WebSocket or request comes in, Channels finds the right consumer for that specific connection and make a copy of it. The developer writes code in the *'consumers.py'* file which handles various events which correspond with the app's needs. Channels itself then handles the scheduling of these events. This process will be explained in detail later.



```
DJAT
  chat
  Djat
  room
  db.sqlite3
  dei0hrdeqish83
  manage.py
  Procfile
  requirements.txt
  runtime.txt
```

Figure 3: Djat project structure.

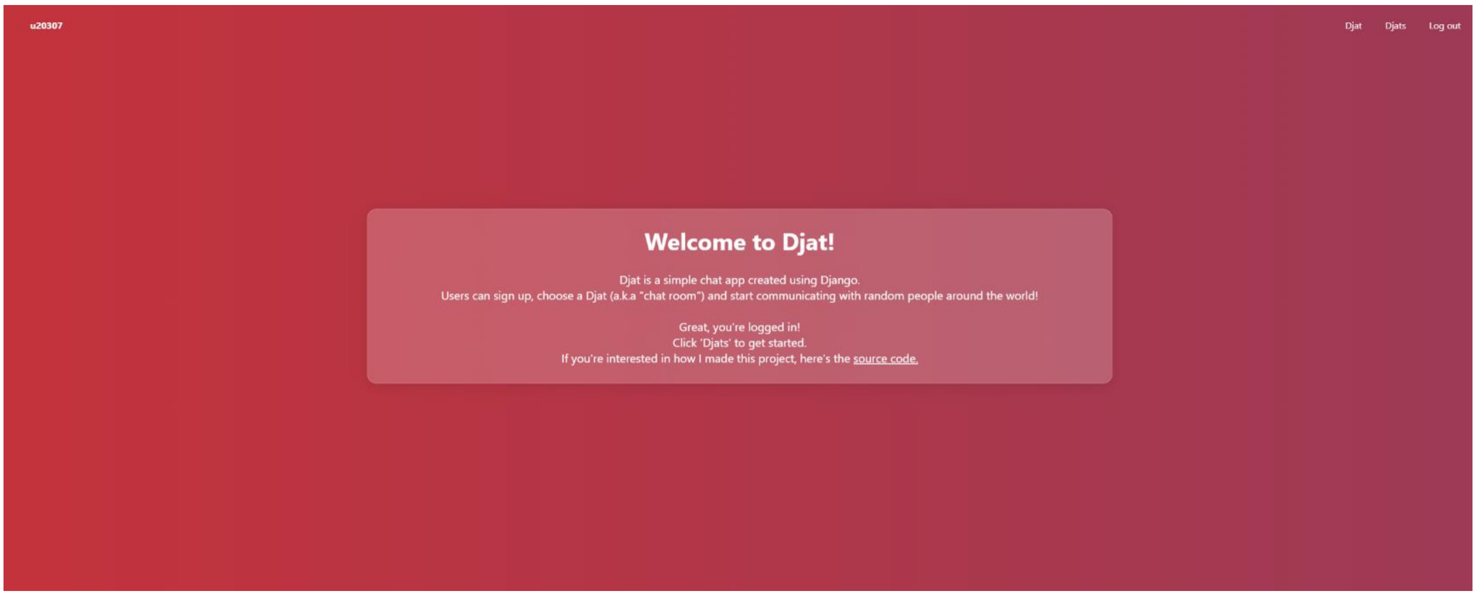
3. Djat

In this chapter, the web application interface is explained.

3.1 Djat UI and functionalities

3.1.1 Djat frontpage

When the user loads up the Djat chat web application on their browser, they are greeted by the frontpage. This page contains a simple navigation bar on top with the user's username to the left and links to other pages on the website on the right (shown in Figure 4). If the user hasn't signed up yet, the username doesn't show up (shown in Figure 5).
Moreover, the user is greeted by a floating 'Welcome to Djat!' message accompanied by a small batch of text explaining the app.

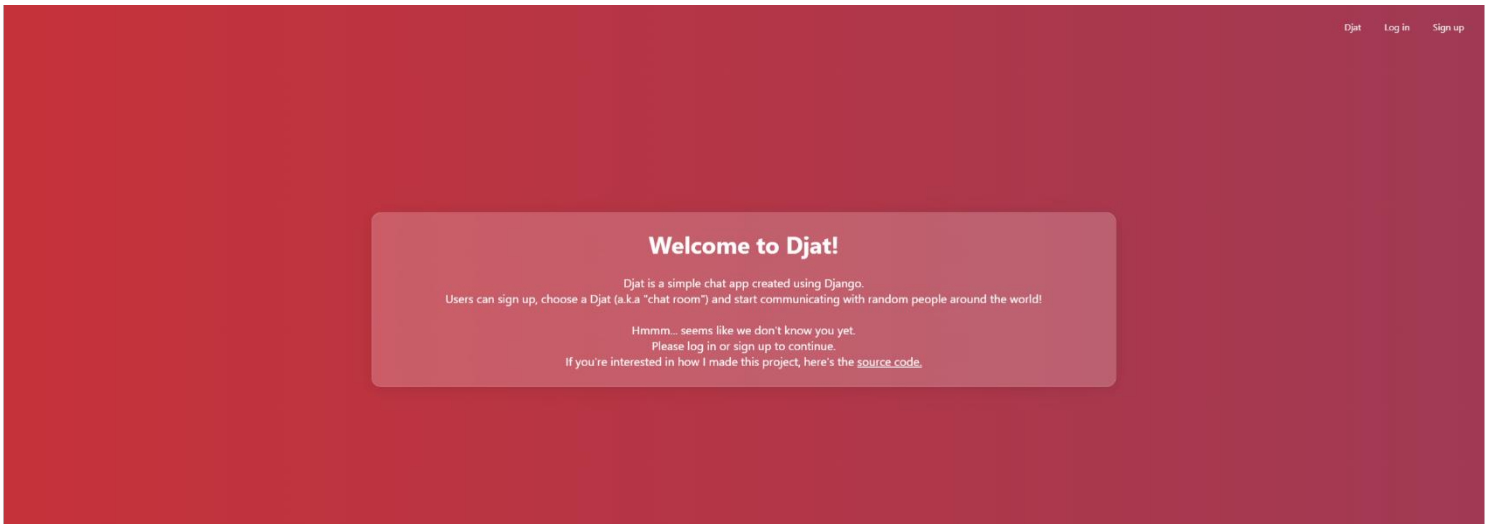


Welcome to Djat!

Djat is a simple chat app created using Django.
Users can sign up, choose a Djat (a.k.a "chat room") and start communicating with random people around the world!

Great, you're logged in!
Click 'Djats' to get started.
If you're interested in how I made this project, here's the source code.

Figure 4: Djat welcome page (user signed in).



Welcome to Djat!

Djat is a simple chat app created using Django.
Users can sign up, choose a Djat (a.k.a "chat room") and start communicating with random people around the world!

Hmmm... seems like we don't know you yet.
Please log in or sign up to continue.
If you're interested in how I made this project, here's the source code.

Figure 5: Djat welcome page (user not signed in).

3.1.2 ‘Sign up’ page

On the sign-up page, the user enters their credentials; username, password and confirm password (shown in Figure 6).
This page incorporates Django’s built-in sign-up form which contains functioning password confirm, password strength and username check features.
Upon successful sign up the user is taken back to the frontpage.

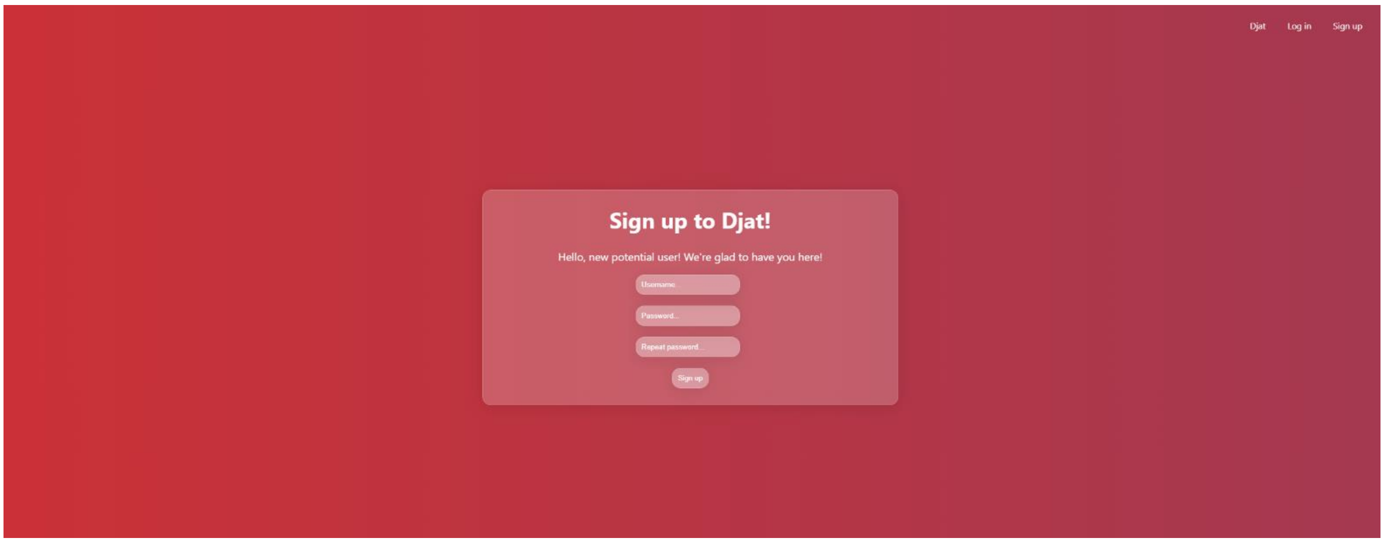
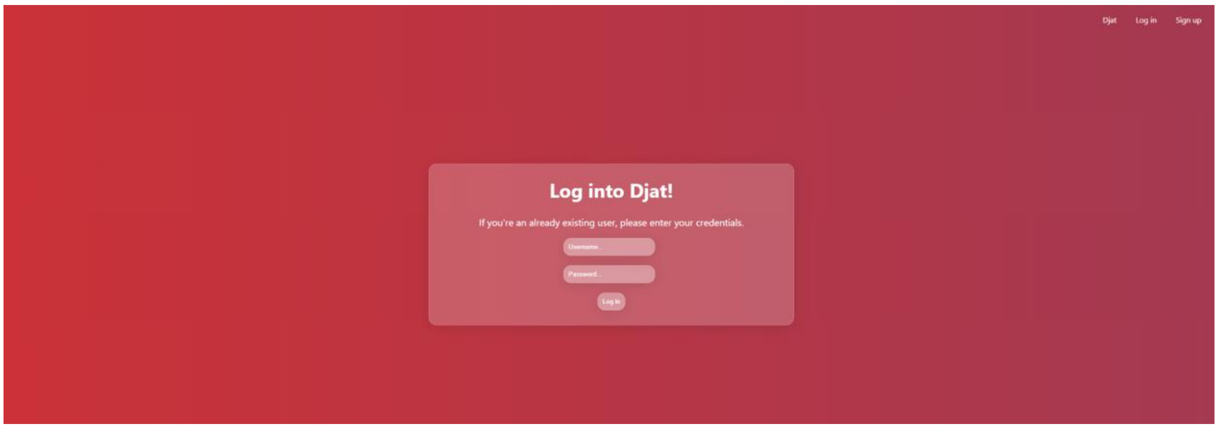


Figure 6: Djat sign up page.

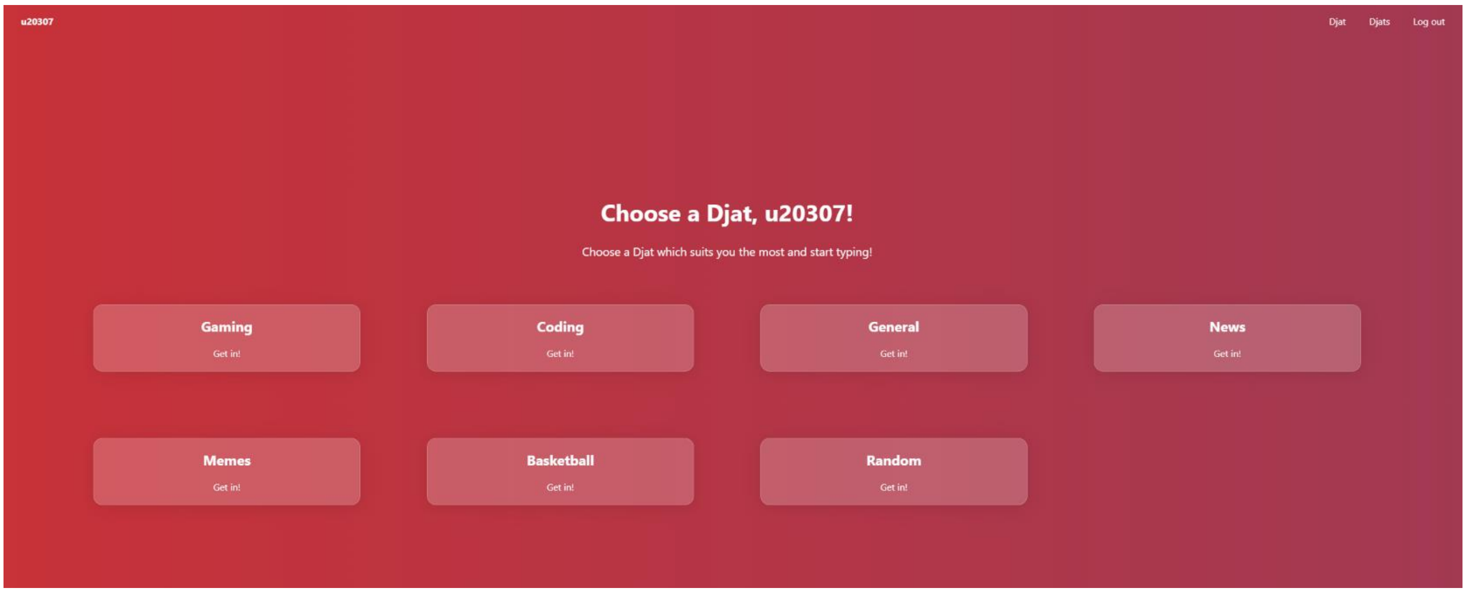
3.1.3 ‘Log in’ page

By design, the Djat log in page is similar to the sign-up page, however this time only the username and one password input field are displayed (shown in Figure 7).
This page also incorporates Django’s built-in log in form which has functioning username and password checks.
Upon successful log in the user is taken back to the frontpage.


Djat
Log in
Sign up
Log into Djat!
If you're an already existing user, please enter your credentials.
Username...
Password...
Log in

3.1.4 'Rooms' page

From this point, the user is able to click the 'Djats' link in the top right to access the list of available rooms/channels to join (shown in Figure 8).
The list itself is styled like a grid and contains the room/channel name and a link to the room/channel itself. The room/channel collection seen below is created by the admin on the Django administration page. The user cannot create a new room/channel.



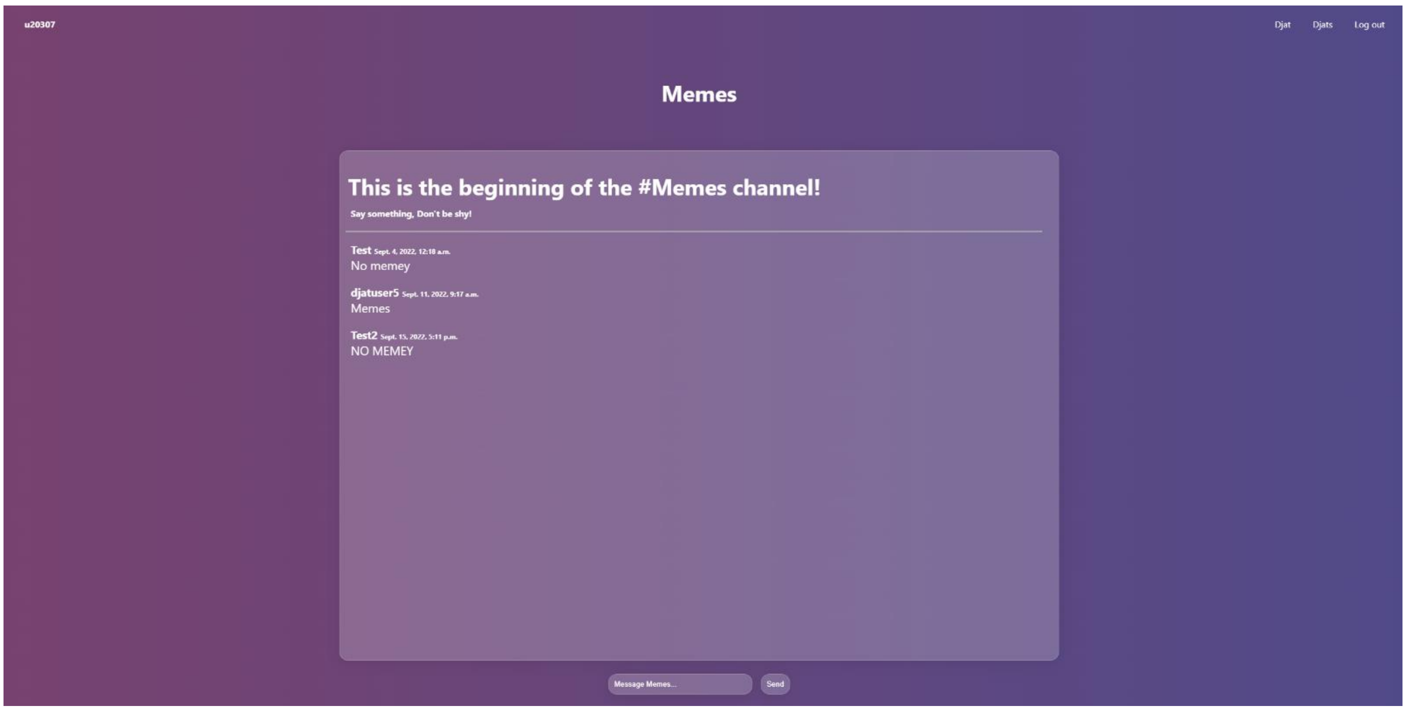
Choose a Djat, u20307!

Choose a Djat which suits you the most and start typing!

Figure 7: Djat room/channel selection page.

When the user clicks on their desired chat room/channel, they are taken to a page where they can post messages in that room/channel (shown in Figure 8).
From this point, the user is able to go back to the frontpage by clicking the 'Djat' link in the navigation bar as well as to log out from the application.

3.1.5 ‘Room’ page



Memes

This is the beginning of the #Memes channel!

Say something, Don't be shy!

Test Sept. 4, 2022, 12:18 a.m.
No memey

djatuser5 Sept. 11, 2022, 9:17 a.m.
Memes

Test2 Sept. 15, 2022, 5:11 p.m.
NO MEMEY

Figure 8: Djat chat room/channel page.

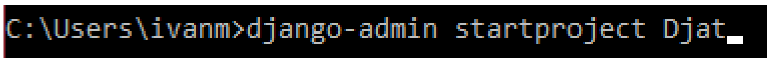
On this page, the user is greeted by a scrollable chat box containing a simple channel intro message as well as the messages themselves.
The messages consist of the sender’s username, the date the message was sent and the message content itself.
Below, there is a message input field and a ‘Send’ button.
If the user does not write anything into the input, they are unable to click the ’Send’ button. This was done in order to prevent the user being able to send ‘empty’ messages.
In this example, the room/channel name is ‘Memes’ and multiple usernames can be seen in the chat box.
From this point, the user can go back to the room/channel selection page by clicking the ‘Djats’ link, go to the frontpage by clicking the ‘Djat’ link and log out of the application.

4 Creating the application

In this chapter, the application creation process is explained.

4.1 Project setup

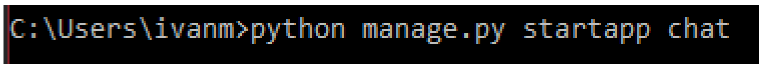
In order to successfully create the app, Python and Django themselves need to be installed. For this project, Python version 3,8.0 and Django 3.2.8 are used [4].
After they are installed, the Django project itself needs to be created.



```
C:\Users\ivanm>django-admin startproject Djat_
```

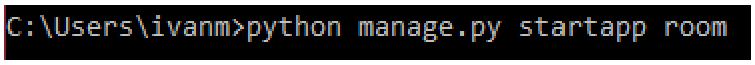
Figure 9: Creating the main 'Djat' Django project..

After this, we create the other two applications, *chat* and *room*, in a similar fashion [10]:



```
C:\Users\ivanm>python manage.py startapp chat
```

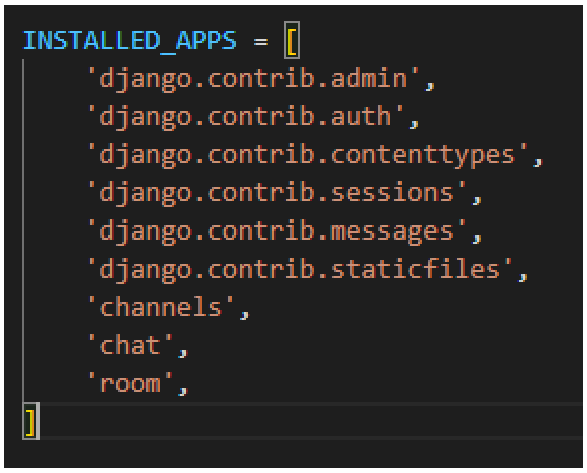
Figure 10: Creating the 'chat' app.



```
C:\Users\ivanm>python manage.py startapp room
```

Figure 11: Creating the 'room' app.

After the applications are created, we need to include them in the *'settings.py'* file in the 'INSTALLED_APPS' section in the Djat project itself (shown in Figure 12):



```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'channels',
    'chat',
    'room',
]
```

Figure 12: Including the 'room' and 'chat' applications.

With this, the setup process is complete.

4.2 Djat

This is the Django project folder itself. Links to the other two applications, project settings and Django channels configuration files are located here (shown in Figure 13).

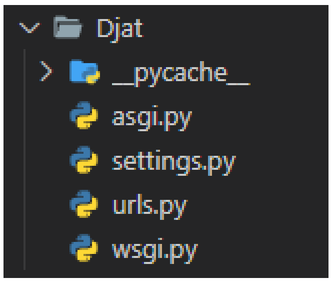
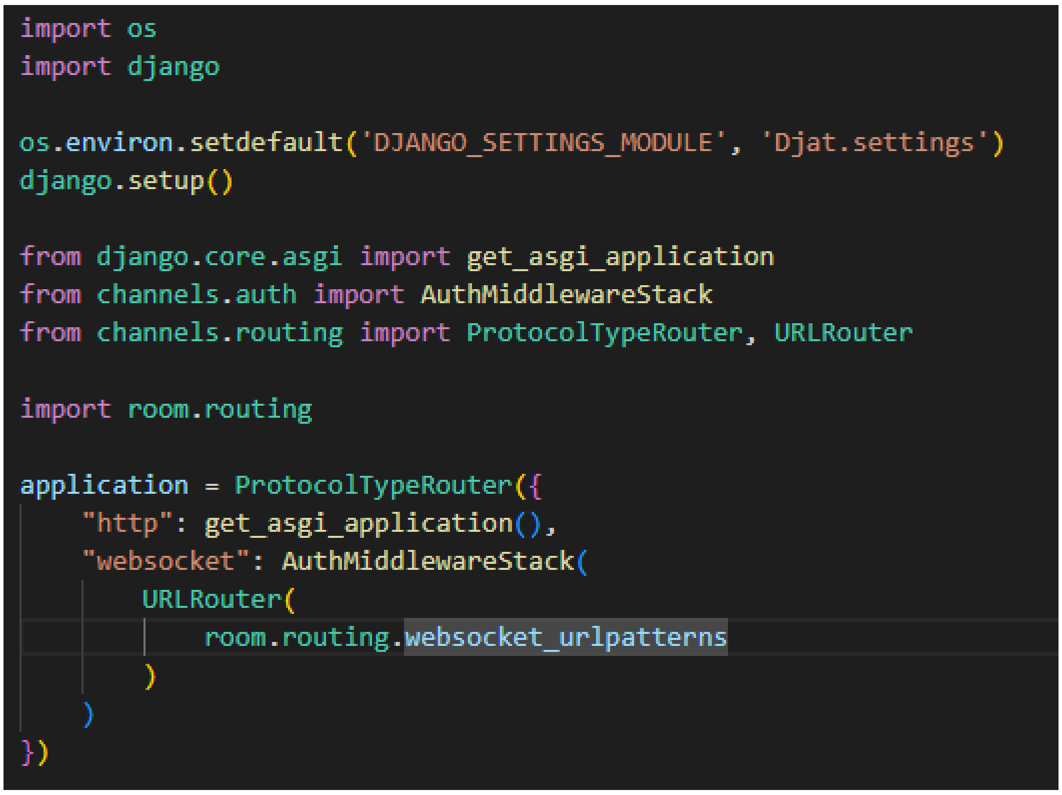


Figure 13: Djat project folder tree.

This folder contains the following files: *asgi.py*, *settings.py*, *urls.py* and *wsgi.py*.
The asgi.py file integrates the Channels library (shown in Figure 14).

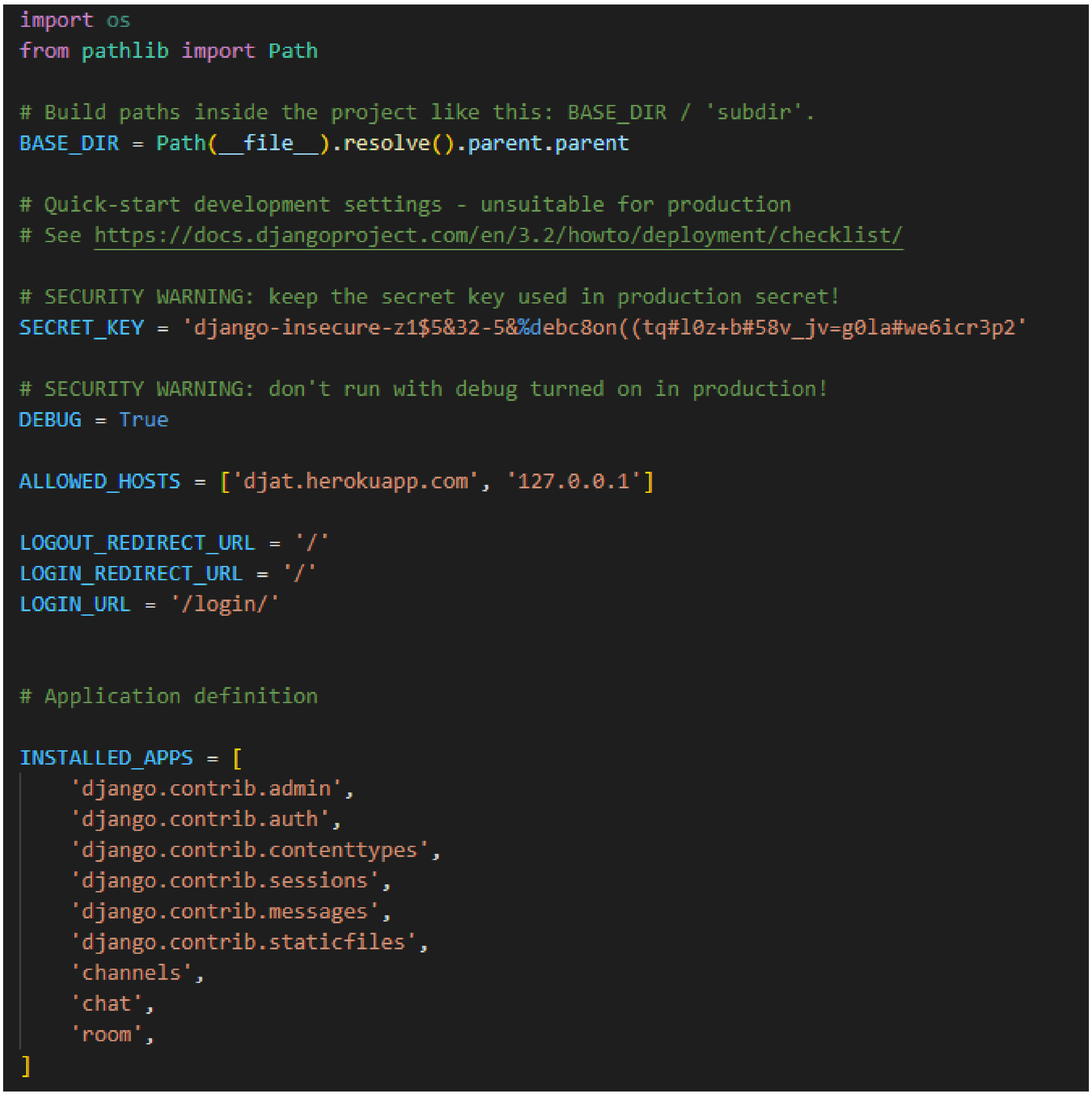


```
import os
import django

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'Djat.settings')
django.setup()

from django.core.asgi import get_asgi_application
from channels.auth import AuthMiddlewareStack
from channels.routing import ProtocolTypeRouter, URLRouter

import room.routing

application = ProtocolTypeRouter({
    "http": get_asgi_application(),
    "websocket": AuthMiddlewareStack(
        URLRouter(
            room.routing.websocket_urlpatterns
        )
    )
})
```

Figure 14: asgi.py

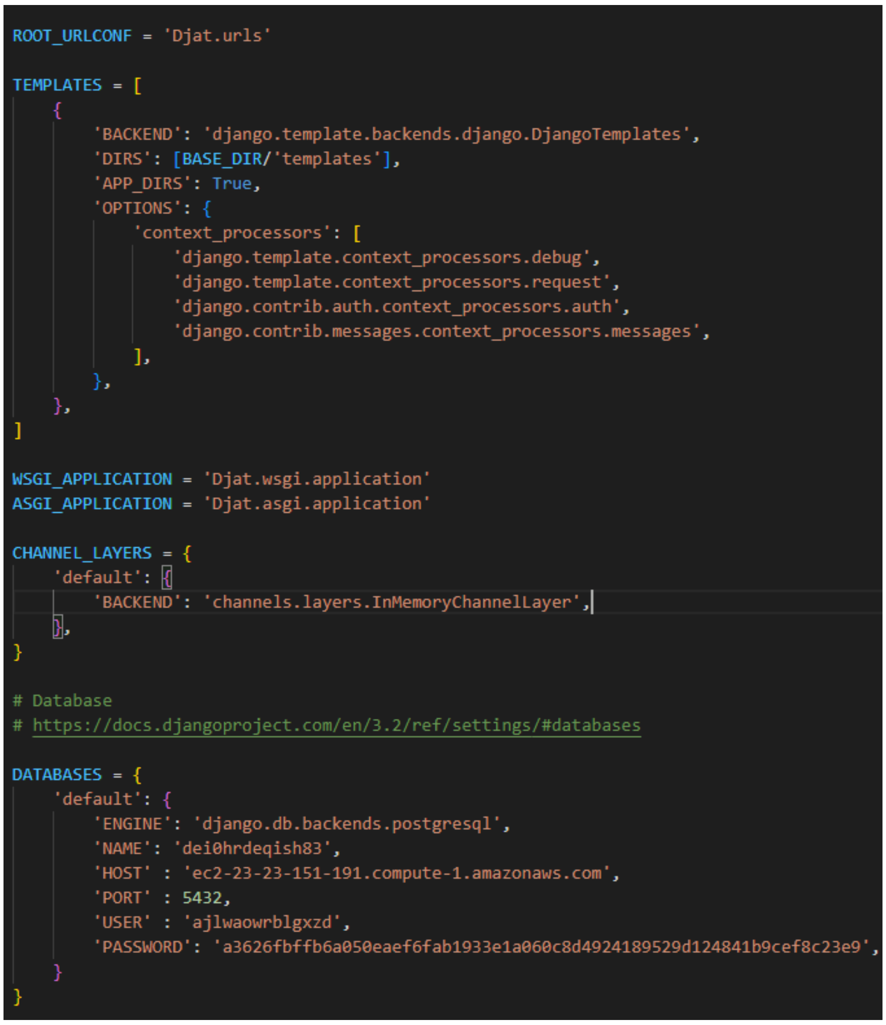
Here, we import the necessary modules needed to incorporate the Channels library.
We also import the *routing.py* file from the *room* application (will be explained later).
The code in the file defines the routing configuration. The configuration specifies that when a connection to Channels is made, *ProtocolTypeRouter* is called and inspects the type of connection established. If the connection is a WebSocket one (‘ws://’), the connection is passed to *AuthMiddlewareStack* which then passes the connection to *URLRouter* which examines the connection and routs it to a consumer, based on the provided URL patterns, in this case, *websocket_urlpatterns* which are located in the *routing.py* file as mentioned before [9].

Furthermore, the *settings.py* file contains all project settings. In this file, static file locations, database, template file locations, login redirect, login, log out URL’s, installed apps, app language, time zone and more are located (shown in Figures 15, 16 & 17).



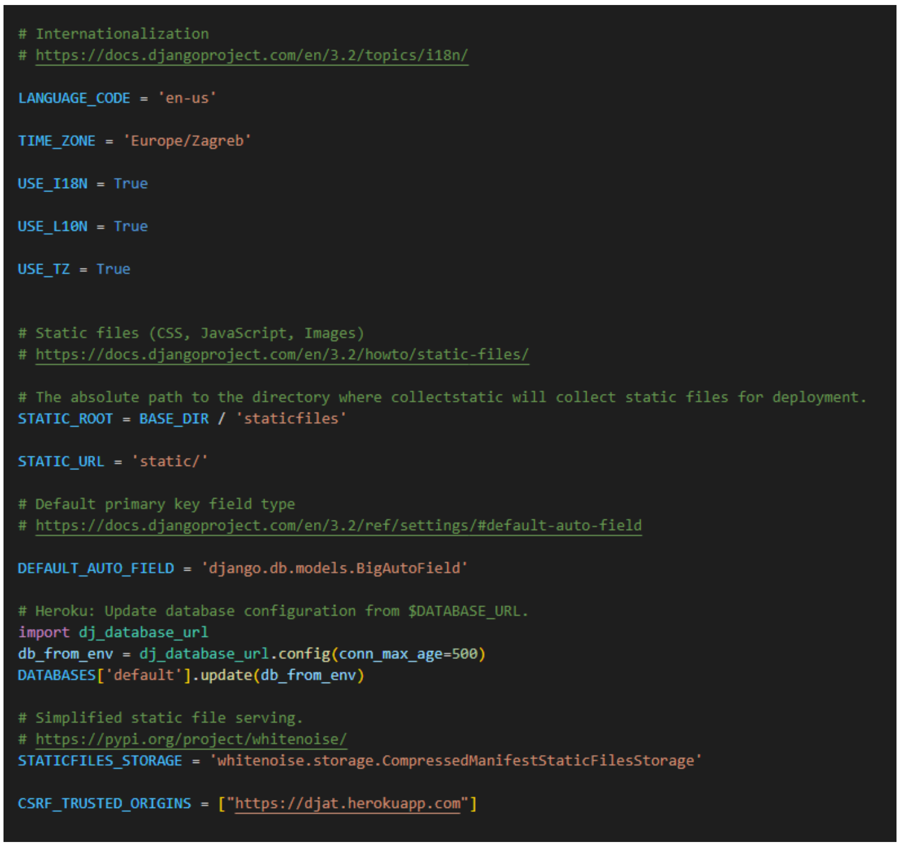
```python
import os
from pathlib import Path

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent

# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/3.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = 'django-insecure-z1$5&32-5&%debc8on((tq#l0z+b#58v_jv=g0la#we6icr3p2'

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = True

ALLOWED_HOSTS = ['djat.herokuapp.com', '127.0.0.1']

LOGOUT_REDIRECT_URL = '/'
LOGIN_REDIRECT_URL = '/'
LOGIN_URL = '/login/'


# Application definition

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'channels',
    'chat',
    'room',
]
```

Figure 15: settings.py file (1).



```python
ROOT_URLCONF = 'Djat.urls'

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',
            ],
        },
    },
]

WSGI_APPLICATION = 'Djat.wsgi.application'
ASGI_APPLICATION = 'Djat.asgi.application'

CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels.layers.InMemoryChannelLayer',
    },
}

# Database
# https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'dei0hrdeqish83',
        'HOST' : 'ec2-23-23-151-191.compute-1.amazonaws.com',
        'PORT' : 5432,
        'USER' : 'ajlwaowrblgxzd',
        'PASSWORD': 'a3626fbffb6a050eaef6fab1933e1a060c8d4924189529d124841b9cef8c23e9',
    }
}
```

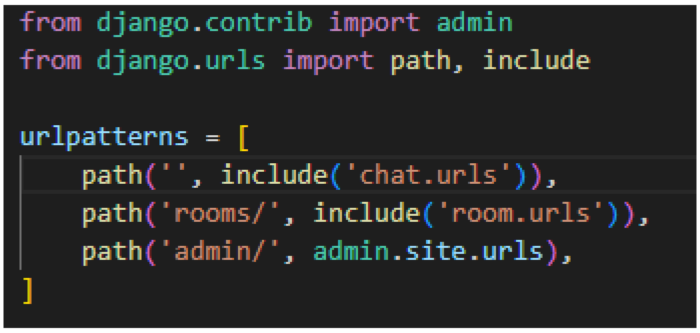
Figure 17: settings.py *file (2).*



```python
# Internationalization
# https://docs.djangoproject.com/en/3.2/topics/i18n/

LANGUAGE_CODE = 'en-us'

TIME_ZONE = 'Europe/Zagreb'

USE_I18N = True

USE_L10N = True

USE_TZ = True


# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/3.2/howto/static-files/

# The absolute path to the directory where collectstatic will collect static files for deployment.
STATIC_ROOT = BASE_DIR / 'staticfiles'

STATIC_URL = 'static/'

# Default primary key field type
# https://docs.djangoproject.com/en/3.2/ref/settings/#default-auto-field

DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'

# Heroku: Update database configuration from $DATABASE_URL.
import dj_database_url
db_from_env = dj_database_url.config(conn_max_age=500)
DATABASES['default'].update(db_from_env)

# Simplified static file serving.
# https://pypi.org/project/whitenoise/
STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'

CSRF_TRUSTED_ORIGINS = ["https://djat.herokuapp.com"]
```

Figure 16: settings.py *file (3).*

Finally, in the *urls.py* file, the main URL paths are specified (shown in Figure 18).



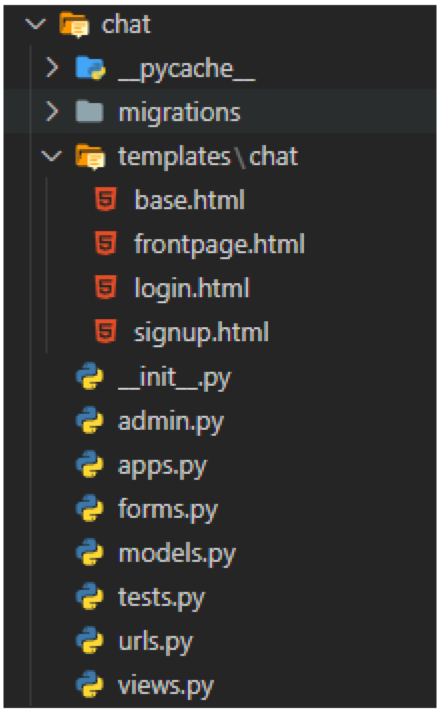
```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('', include('chat.urls')),
    path('rooms/', include('room.urls')),
    path('admin/', admin.site.urls),
]
```

Figure 18: urls.py *file.*

The first path includes and points to the URL's related to the *chat* application, the second path includes and points to the URL's related to the *room* application whilst the third path redirects to the Django administration page only accessible by superusers.
This means if the user enters the main URL of the application (Welcome | Djat), the user will be redirected to the frontpage of the *chat* application, as that is the default page of that application (will be explained in the *chat* application chapter).
Moreover, if the user enters any URL with '*rooms/*', they will be redirected to the room/channel selection page, as it is the main page of the *room* application.

4.3 ‘chat’ application

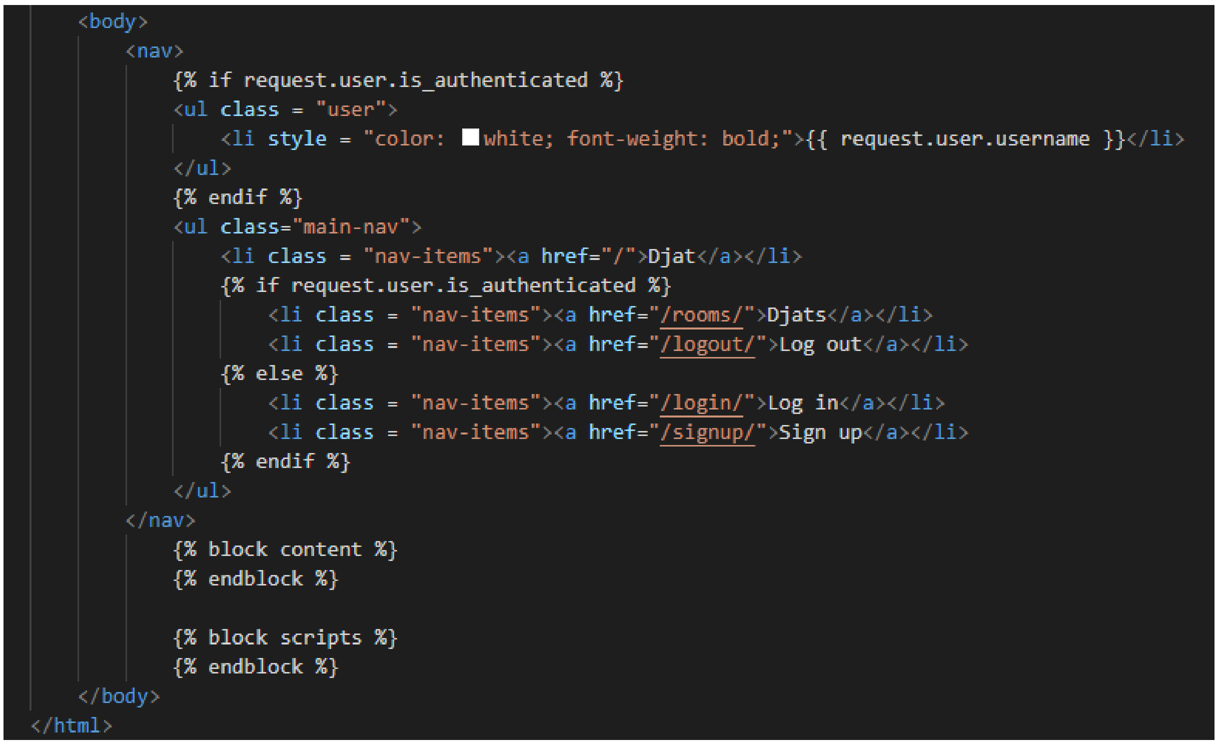
As mentioned before, this is the application containing the main, log in and sign-up pages. This application folder consists of the following files: the templates and chat folder in which the *base.html*, *frontpage.html*, *login.html* and *signup.html* are located, *admin.py, forms.py, models.py, urls.py* and *views.py* (shown in Figure 19).



```
chat
  __pycache__
  migrations
  templates\chat
    base.html
    frontpage.html
    login.html
    signup.html
  __init__.py
  admin.py
  apps.py
  forms.py
  models.py
  tests.py
  urls.py
  views.py
```

Figure 19: 'chat' *application tree.*

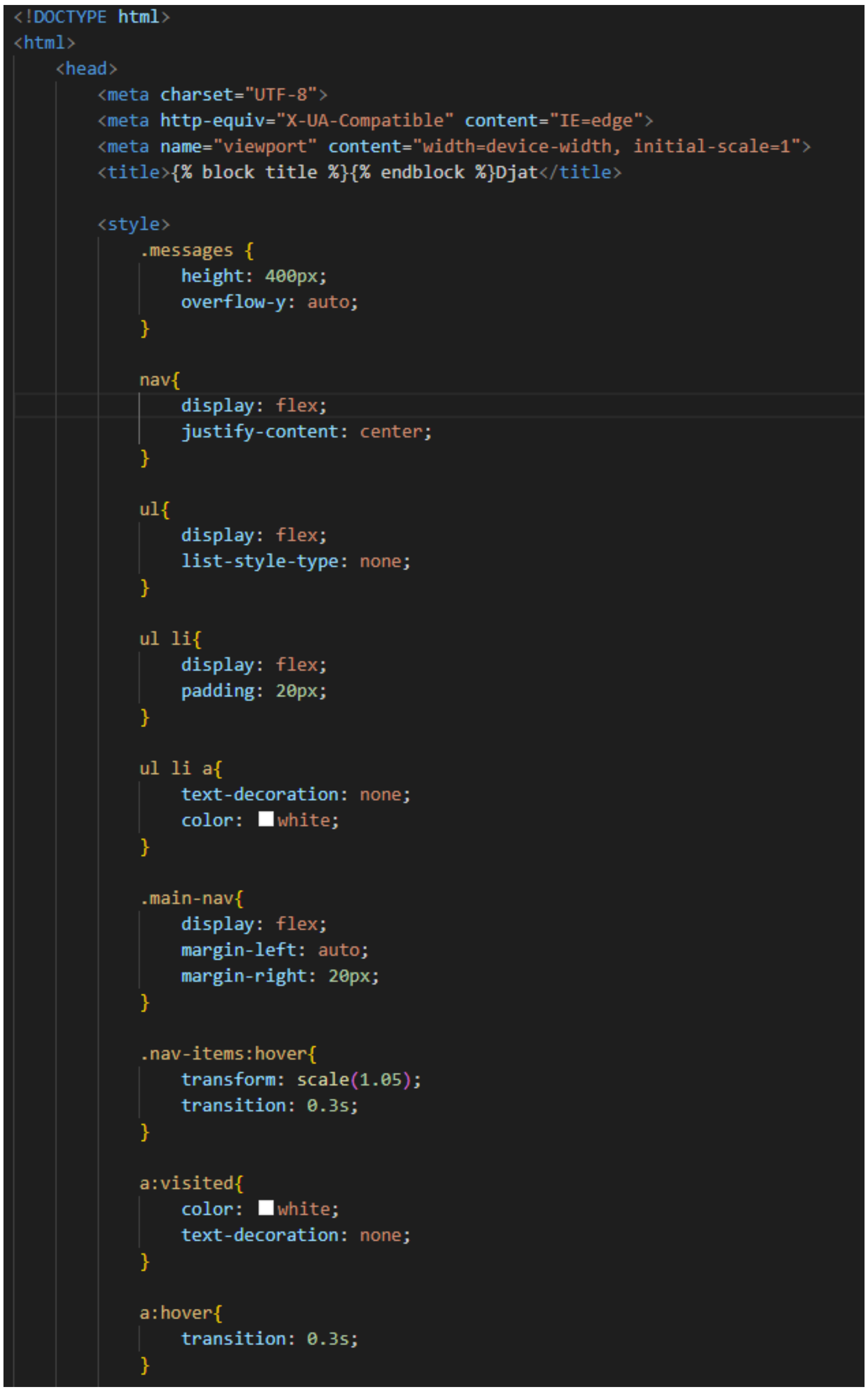
The template file *base.html* defines the basic design and appearance of each subpage of the website. The other subpages are extensions of the *base.html* file.
Also in this file the CSS code is located. This code is used to design both the *base.html* file and all the other pages (shown in Figure 21).



```
    <body>
        <nav>
            {% if request.user.is_authenticated %}
            <ul class = "user">
                <li style = "color: ■white; font-weight: bold;">{{ request.user.username }}</li>
            </ul>
            {% endif %}
            <ul class="main-nav">
                <li class = "nav-items"><a href="/">Djat</a></li>
                {% if request.user.is_authenticated %}
                    <li class = "nav-items"><a href="/rooms/">Djats</a></li>
                    <li class = "nav-items"><a href="/logout/">Log out</a></li>
                {% else %}
                    <li class = "nav-items"><a href="/login/">Log in</a></li>
                    <li class = "nav-items"><a href="/signup/">Sign up</a></li>
                {% endif %}
            </ul>
        </nav>
            {% block content %}
            {% endblock %}

            {% block scripts %}
            {% endblock %}
    </body>
</html>
```

Figure 20: base.html *template file.*

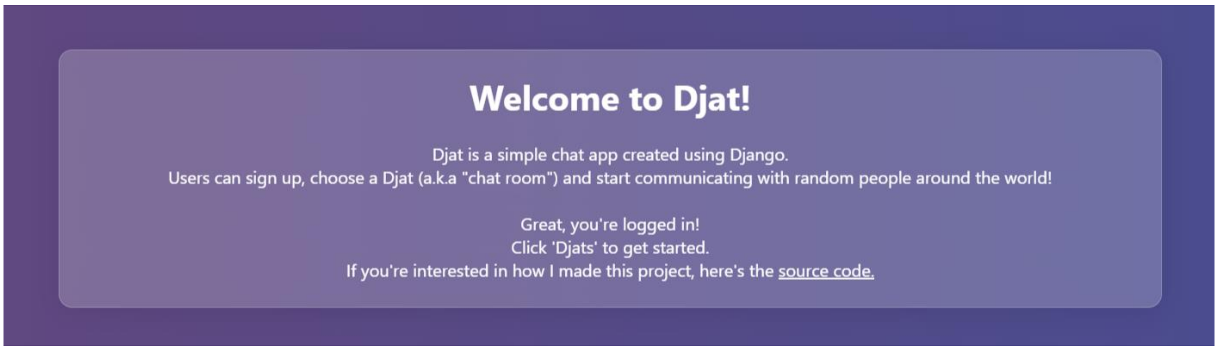
The body of the file contains the navigation bar and the navigation links (shown in Figure 20). The navigation page will display the user's username if they are signed in. This is done using the Django template language [8] which enables Django Python code to be written inside an HTML file. This code is distinctly divided using '{% %}' values for loops, and {{ }} values if a variable is entered.



```html
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <meta http-equiv="X-UA-Compatible" content="IE=edge">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>{% block title %}{% endblock %}Djat</title>

        <style>
            .messages {
                height: 400px;
                overflow-y: auto;
            }

            nav{
                display: flex;
                justify-content: center;
            }

            ul{
                display: flex;
                list-style-type: none;
            }

            ul li{
                display: flex;
                padding: 20px;
            }

            ul li a{
                text-decoration: none;
                color: white;
            }

            .main-nav{
                display: flex;
                margin-left: auto;
                margin-right: 20px;
            }

            .nav-items:hover{
                transform: scale(1.05);
                transition: 0.3s;
            }

            a:visited{
                color: white;
                text-decoration: none;
            }

            a:hover{
                transition: 0.3s;
            }
```

Figure 21: base.html CSS code.

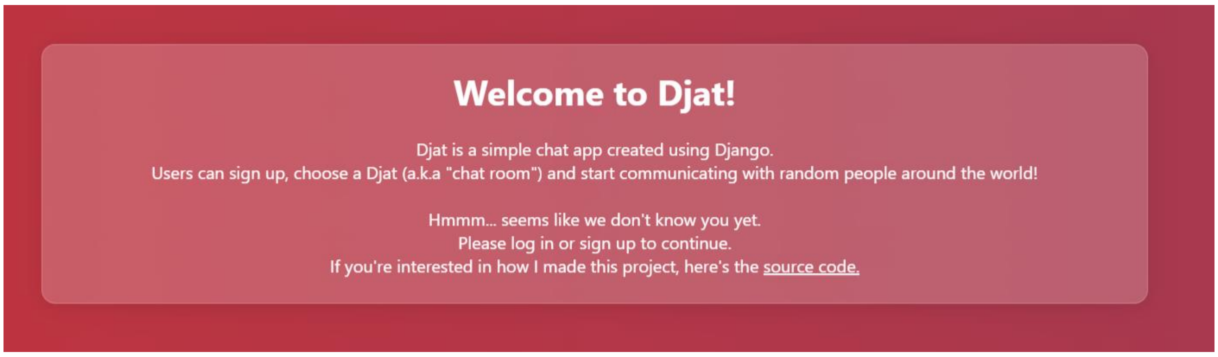
The second template file, *frontpage.html* extends the *base.html* file and displays the main greeting message using simple HTML. This page also uses the Django template language in order to determine whether the user is logged in or not.
Depending on whether the user is logged in or not, the appropriate welcome message will be displayed (shown in Figures 22 & 23).


Welcome to Djat!

Djat is a simple chat app created using Django.
Users can sign up, choose a Djat (a.k.a "chat room") and start communicating with random people around the world!

Great, you're logged in!
Click 'Djats' to get started.
If you're interested in how I made this project, here's the source code.

Figure 22: Welcome message (user is logged in).


Welcome to Djat!

Djat is a simple chat app created using Django.
Users can sign up, choose a Djat (a.k.a "chat room") and start communicating with random people around the world!

Hmmm... seems like we don't know you yet.
Please log in or sign up to continue.
If you're interested in how I made this project, here's the source code.

Figure 23: Welcome message (user is not logged in).

The third template page, *login.html* displays the built-in Django log in form (shown in Figure 24). The body of the page contains one main div element containing a simple greeting message underneath which the log in form is located.
As mentioned before, the user enters their username and password. Then, using the Django template language, we check for any possible errors with the credentials and Django will not log us in if the credentials are incorrect.

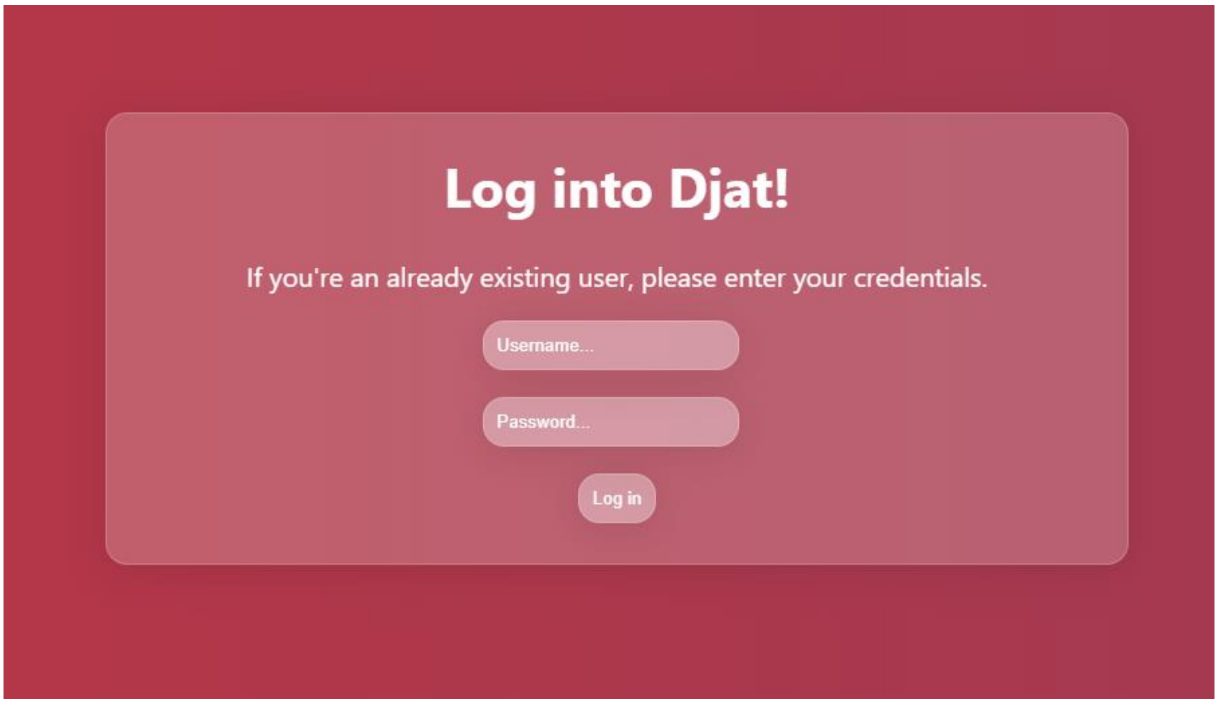
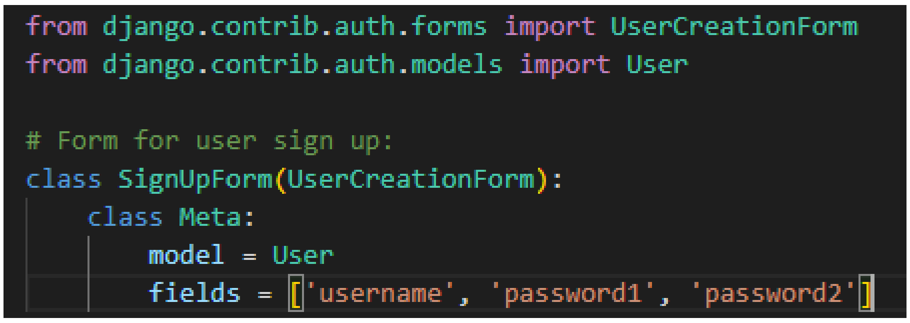

Log into Djat!
If you're an already existing user, please enter your credentials.
Username...
Password...
Log in

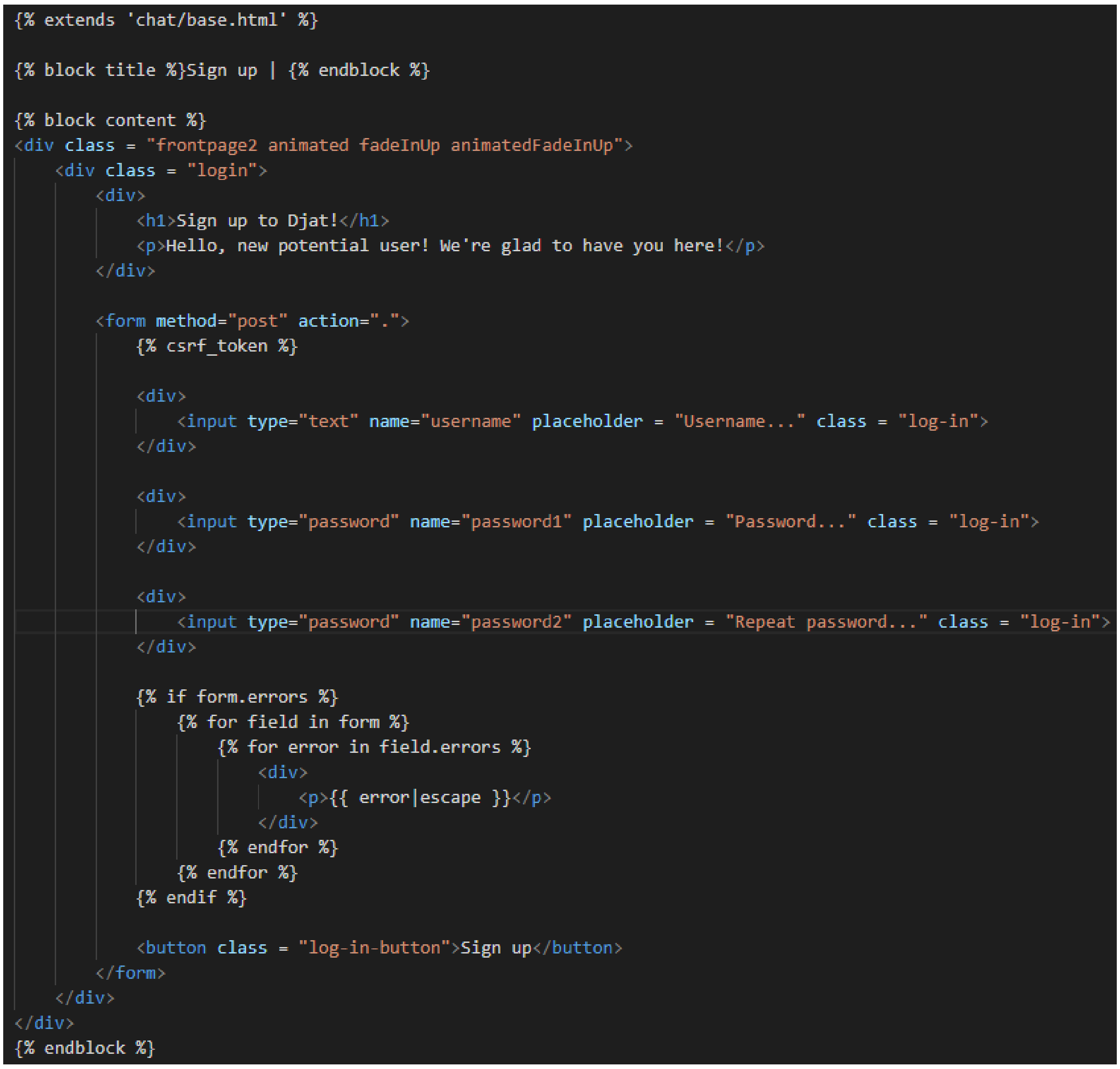
Figure 24: Log in page.

The last template page in the *chat* application is the *signup.html* template file. This file is similar to the *login.html* file, but here we use the user creation form from the *forms.py* file in which we create a new class called SignUpForm using the UserCreationForm module imported from django.contrib.auth.forms as well as the built-in User model imported from django.contrib.auth.models.
We use the User model inside the class and define the fields needed to allow the application to create a new user (*username*, *password1* and *password2*) (shown in Figure 25).



```
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

# Form for user sign up:
class SignUpForm(UserCreationForm):
    class Meta:
        model = User
        fields = ['username', 'password1', 'password2']
```

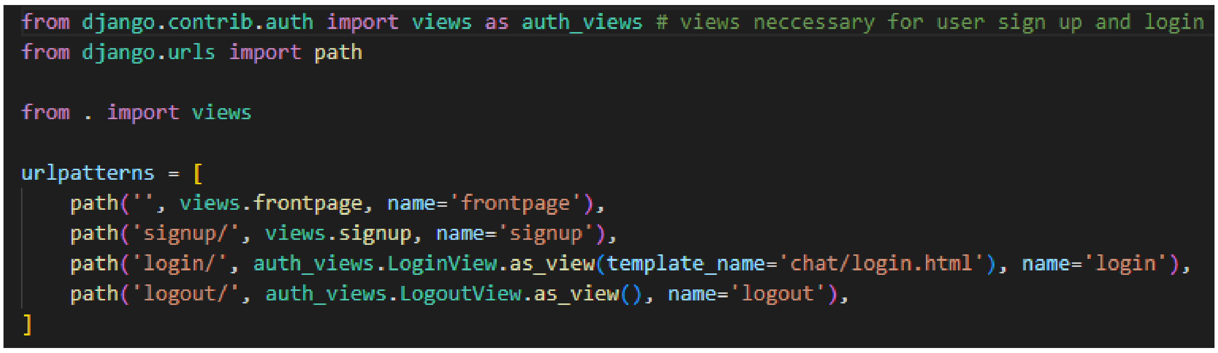
Figure 25: forms.py file.



```
{% extends 'chat/base.html' %}

{% block title %}Sign up | {% endblock %}

{% block content %}
<div class = "frontpage2 animated fadeInUp animatedFadeInUp">
    <div class = "login">
        <div>
            <h1>Sign up to Djat!</h1>
            <p>Hello, new potential user! We're glad to have you here!</p>
        </div>

        <form method="post" action=".">
            {% csrf_token %}

            <div>
                <input type="text" name="username" placeholder = "Username..." class = "log-in">
            </div>

            <div>
                <input type="password" name="password1" placeholder = "Password..." class = "log-in">
            </div>

            <div>
                <input type="password" name="password2" placeholder = "Repeat password..." class = "log-in">
            </div>

            {% if form.errors %}
                {% for field in form %}
                    {% for error in field.errors %}
                        <div>
                            <p>{{ error|escape }}</p>
                        </div>
                    {% endfor %}
                {% endfor %}
            {% endif %}

            <button class = "log-in-button">Sign up</button>
        </form>
    </div>
</div>
{% endblock %}
```

Figure 26: signup.html.

Furthermore, inside the *urls.py,* all the necessary URL's for the *chat* application are defined (shown in Figure 27).

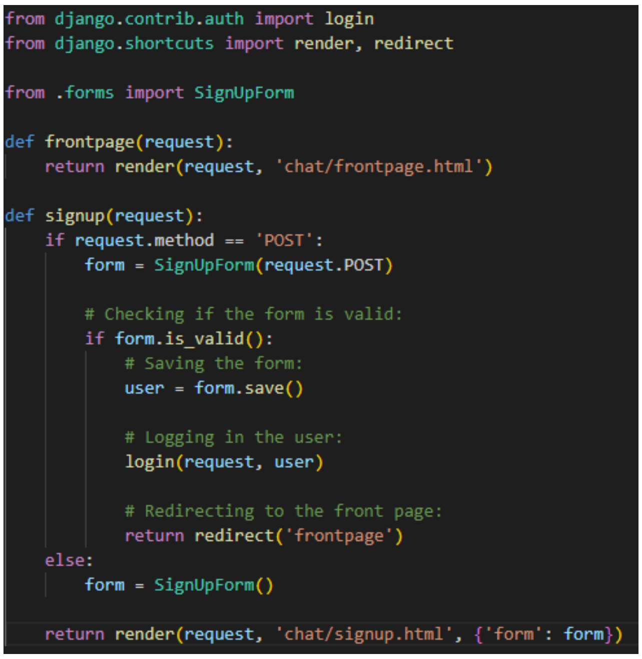


```
from django.contrib.auth import views as auth_views # views neccessary for user sign up and login
from django.urls import path

from . import views

urlpatterns = [
    path('', views.frontpage, name='frontpage'),
    path('signup/', views.signup, name='signup'),
    path('login/', auth_views.LoginView.as_view(template_name='chat/login.html'), name='login'),
    path('logout/', auth_views.LogoutView.as_view(), name='logout'),
]
```

Figure 27: urls.py

The first URL path loads the frontpage view and takes the user to the frontpage. This goes back to the *urls.py* file from the Djat project where this same path was defined, however it pointed to all these *chat* application URL's.
The second URL loads the built-in sign-up view and takes the user to the sing-up page, the third URL loads the built in log in view and takes the user to the log in page and the last URL logs the user out of the application.

Finally, the *views.py* file contains functions which are called on request (shown in Figure 28).



```python
from django.contrib.auth import login
from django.shortcuts import render, redirect

from .forms import SignUpForm

def frontpage(request):
    return render(request, 'chat/frontpage.html')

def signup(request):
    if request.method == 'POST':
        form = SignUpForm(request.POST)

        # Checking if the form is valid:
        if form.is_valid():
            # Saving the form:
            user = form.save()

            # Logging in the user:
            login(request, user)

            # Redirecting to the front page:
            return redirect('frontpage')
    else:
        form = SignUpForm()

    return render(request, 'chat/signup.html', {'form': form})
```

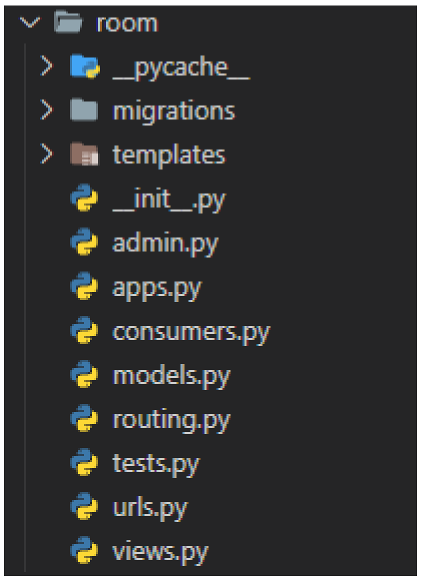
Figure 28: views.py file.

The first function, *frontpage*, takes one parameter, *request*, and with it renders the *frontpage.html* template file.
The second function, *signup*, also takes one argument, *request*, and checks if the form entered is valid. If the form is valid then the information entered is stored into the database, the user is logged in and redirected to the frontpage.
If the form isn't valid, a new form is called.
This function redirects the user to the *signup.html* page.

The *models.py* and *tests.py* files were not needed because in this application there is no data we need to create to be viewed, as we are mainly using Django's built in models and forms.

4.4 ‘room’ application

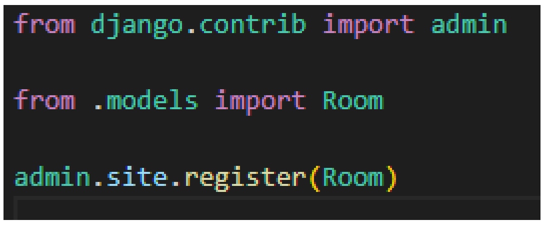
In this application the chat functionality is defined. Also, the *consumers.py* and *routing.py* files necessary for using Django Channels are located here (shown in Figure 29).
This application contains template files, views, URL’s, models and model registration files.



```
room
  __pycache__
  migrations
  templates
  __init__.py
  admin.py
  apps.py
  consumers.py
  models.py
  routing.py
  tests.py
  urls.py
  views.py
```

Figure 29: room application tree.

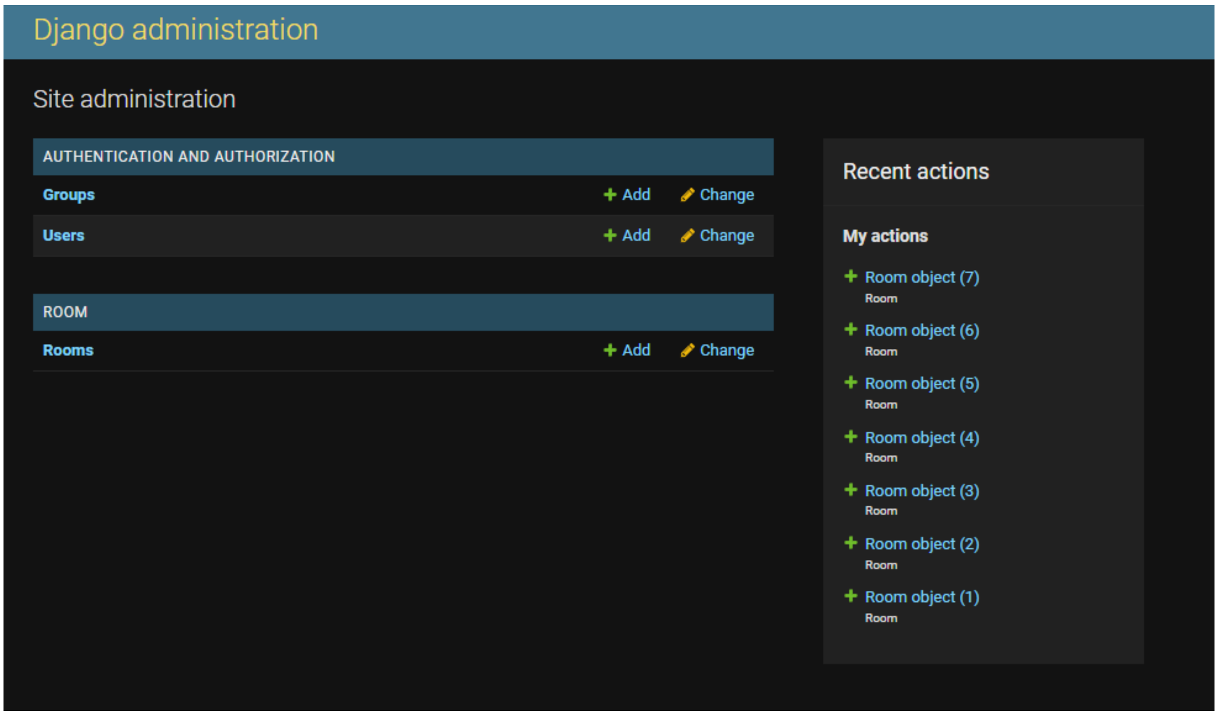
Firstly, in the *admin.py* file, the models used for this application are registered in the database. In this case, this is the *Room* model (shown in Figure 30).



```python
from django.contrib import admin

from .models import Room

admin.site.register(Room)
```

Figure 30: admin.py.

As a result, on the Django administration page, admins are able to add new rooms, edit or delete them (shown in Figures 31 & 32).


Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups	+ Add	Change
Users	+ Add	Change

ROOM

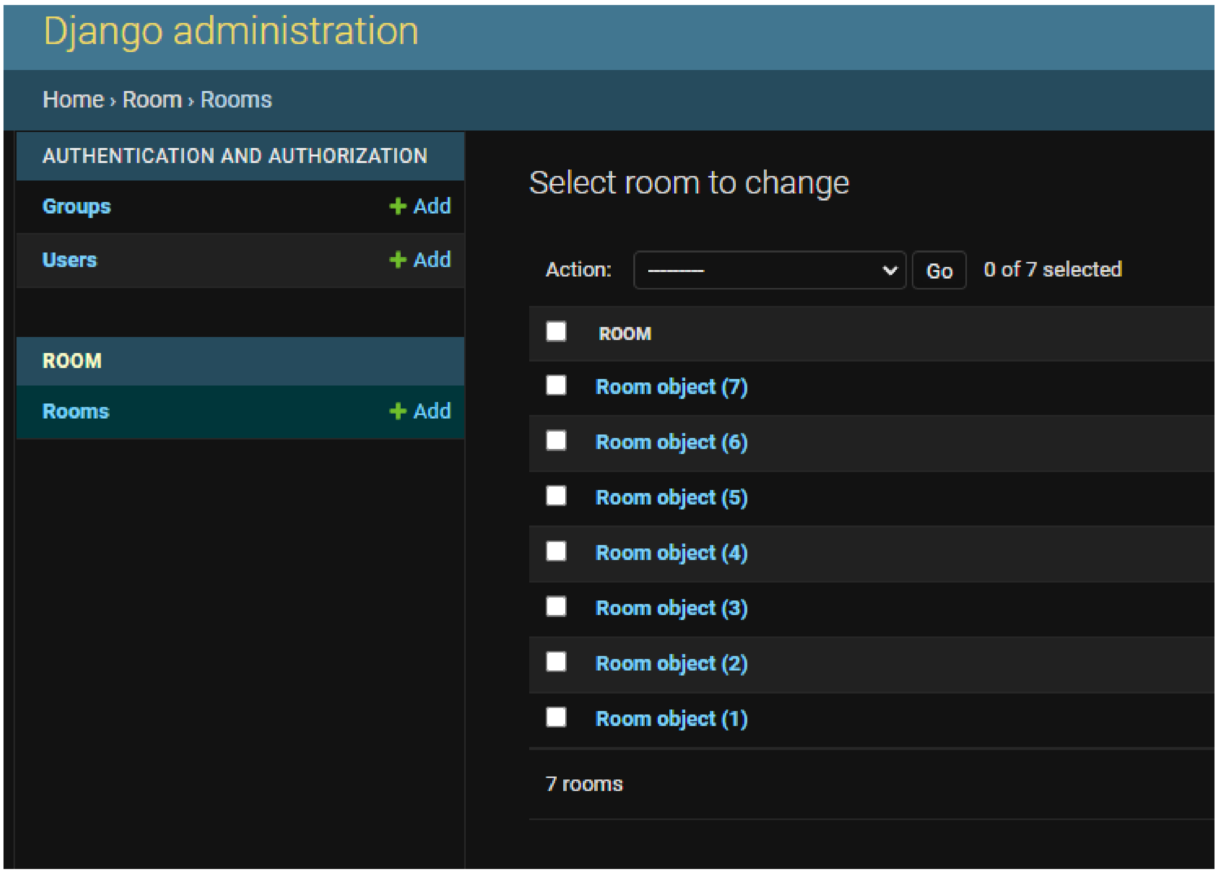
Rooms	+ Add	Change

Recent actions

My actions

+ Room object (7)
Room
+ Room object (6)
Room
+ Room object (5)
Room
+ Room object (4)
Room
+ Room object (3)
Room
+ Room object (2)
Room
+ Room object (1)
Room

Figure 31: Django administration page.


Django administration

Home › Room › Rooms

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

ROOM

Rooms + Add

Select room to change

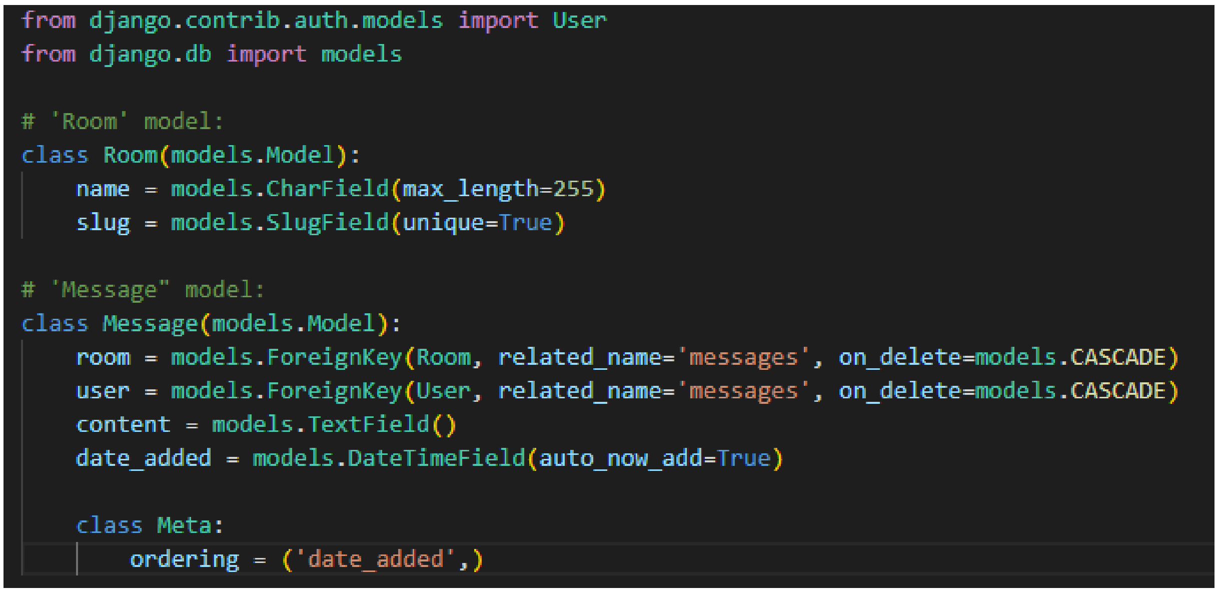
Action: --------- Go 0 of 7 selected

ROOM
Room object (7)
Room object (6)
Room object (5)
Room object (4)
Room object (3)
Room object (2)
Room object (1)

7 rooms

Figure 32: All rooms.

Moreover, inside the *models.py* file the necessary models are defined (shown in Figure 33).
The models are *Room* and *Message*. These models are defined as classes and contain attributes which represent a database field.



```python
from django.contrib.auth.models import User
from django.db import models

# 'Room' model:
class Room(models.Model):
    name = models.CharField(max_length=255)
    slug = models.SlugField(unique=True)

# 'Message" model:
class Message(models.Model):
    room = models.ForeignKey(Room, related_name='messages', on_delete=models.CASCADE)
    user = models.ForeignKey(User, related_name='messages', on_delete=models.CASCADE)
    content = models.TextField()
    date_added = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ('date_added',)
```

Figure 33: models.py.

The *Room* class is defined first. Its attributes are *name* and *slug.*
Name is a CharField with a max length of 225, while *slug* is a unique SlugFIeld which is used to create a valid URL for each room/channel [11]. This will be evaluated upon in the *urls.py* file later.
The *Message* class has *room, user, content* and *date_added* attributes.

Room is a ForeignKey [7]referring to the *Room* model, *user* is also a ForeignKey, but from the built-in Django User model. *Content* is a TextField and *date_added* is a DateTimeField which automatically specifies the time and date a message was sent.

Furthermore, inside the *urls.py* file all the necessary URL's related to the *room* application are located (shown in Figure 34).

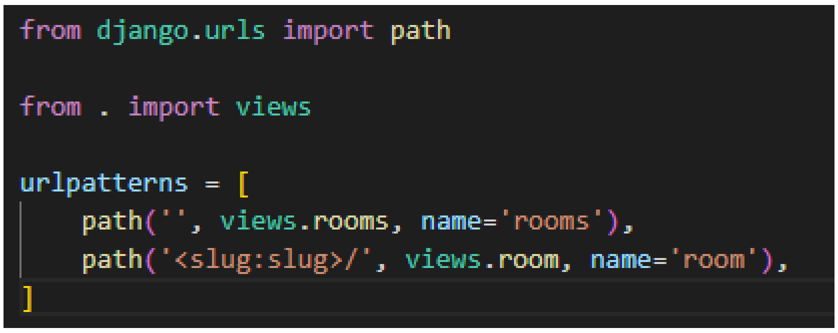

```python
from django.urls import path

from . import views

urlpatterns = [
    path('', views.rooms, name='rooms'),
    path('<slug:slug>/', views.room, name='room'),
]
```

Figure 34: urls.py.

The first URL path loads the *rooms* view and takes the user to the room/channel selection page, while the second URL path uses the *slug* attribute from the *Room* model to generate a valid URL to the specific room.

The *views.py* file contains all the used views inside the *urls.py* file (shown in Figure 35).

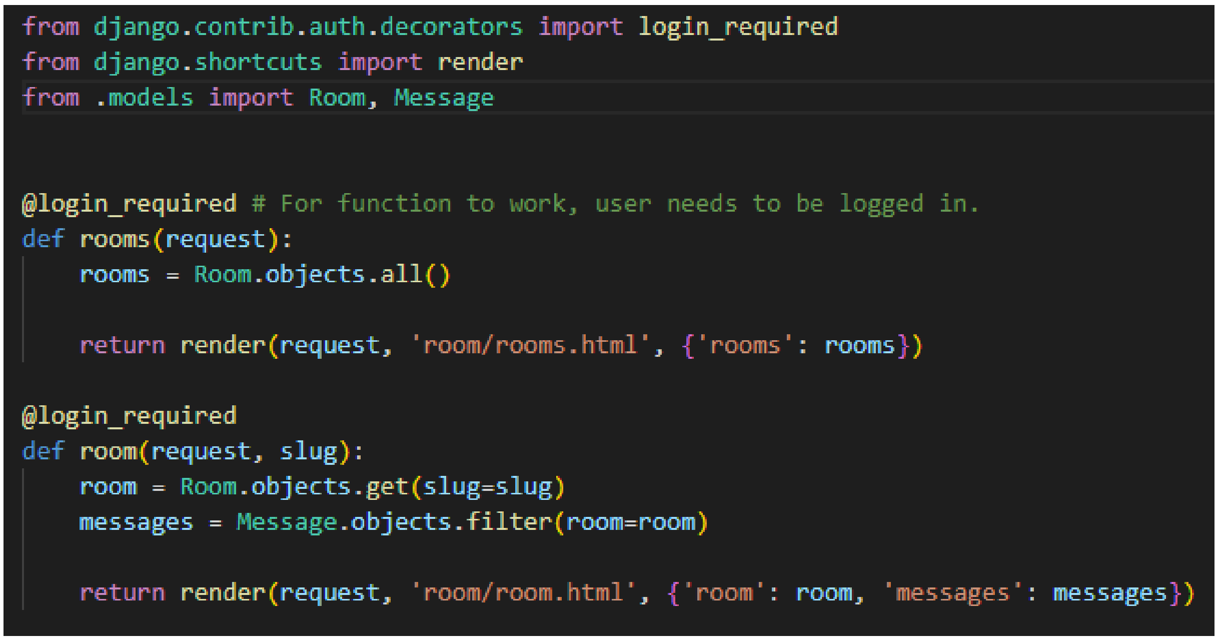
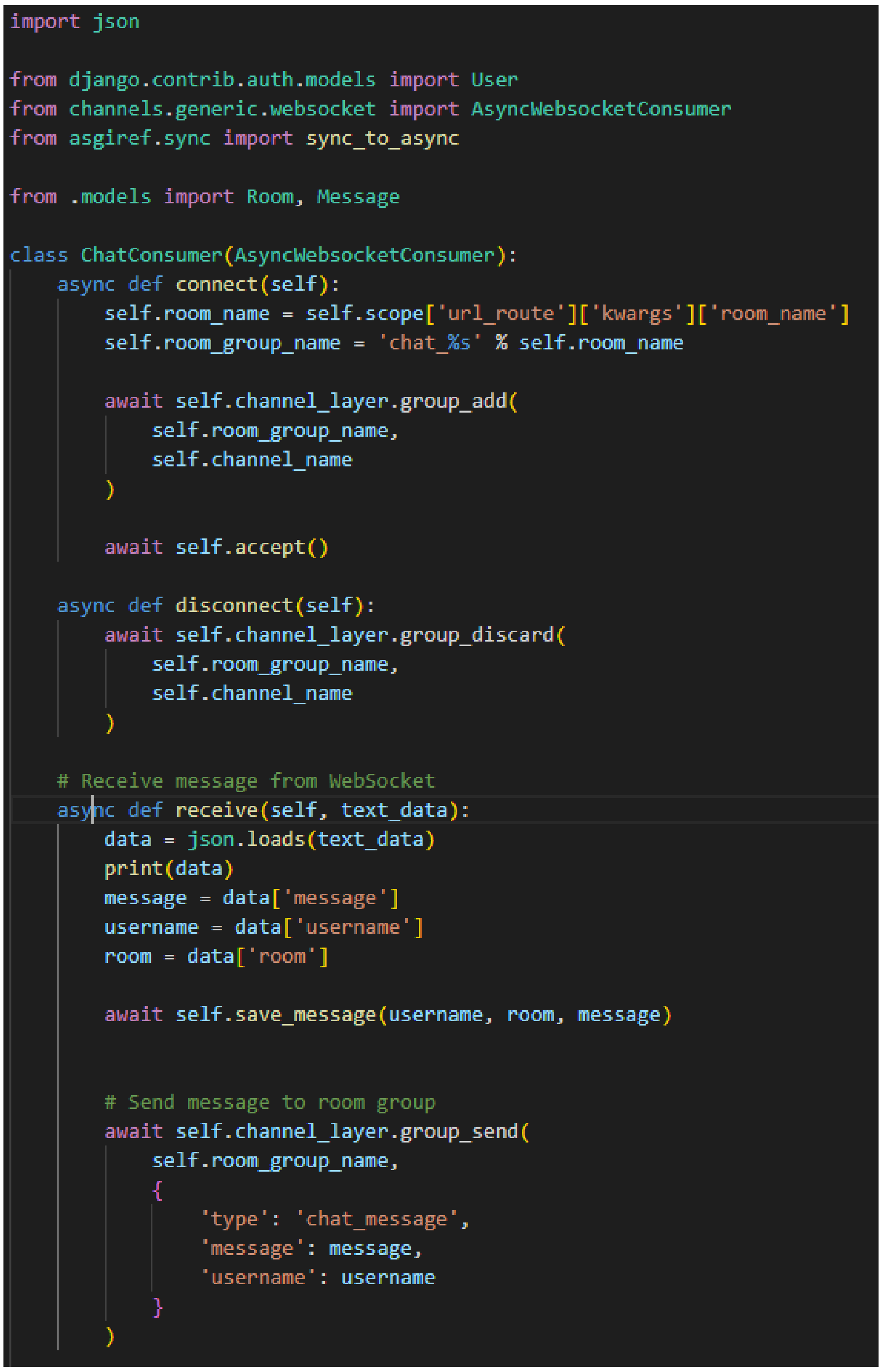

```python
from django.contrib.auth.decorators import login_required
from django.shortcuts import render
from .models import Room, Message


@login_required # For function to work, user needs to be logged in.
def rooms(request):
    rooms = Room.objects.all()

    return render(request, 'room/rooms.html', {'rooms': rooms})

@login_required
def room(request, slug):
    room = Room.objects.get(slug=slug)
    messages = Message.objects.filter(room=room)

    return render(request, 'room/room.html', {'room': room, 'messages': messages})
```

Figure 35: views.py.

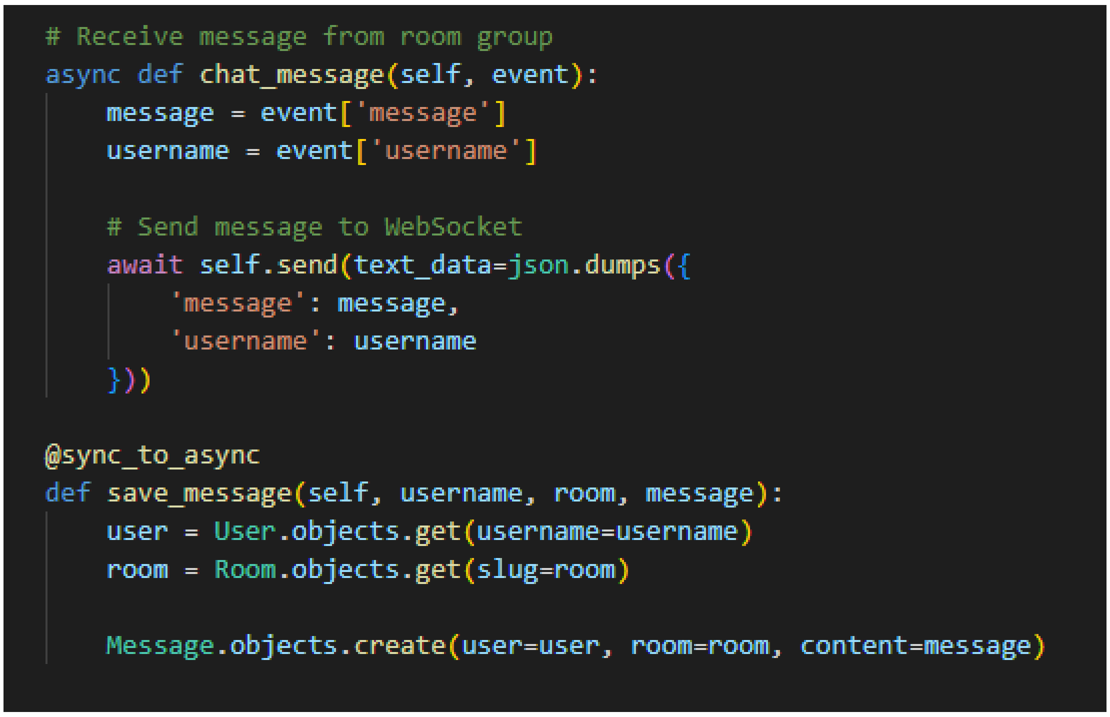
In this file we import all the necessary modules and models, and we define the two functions that are locked, that is, that can only be accessed when the user is logged in.
The first view, *rooms*, takes one parameter which is *request*, gets all the values from the *Room* model and renders them on the *rooms.html* template page.
The second view, *room*, takes the same parameter and gets the specific object from the *Room* model identified by its slug, as well as the accompanying messages which are filtered to display only the messages for that specific room and renders them in the *room.html* template file.

Finally, in this application, the *consumers.py* and *routing.py* files are located. As mentioned before, these are necessary files needed for Django Channels to work.
In the *consumers.py* file, a main class called ChatConsumer is created.
Inside this class, asynchronous functions are defined which help Channels when a specific event is called (shown in Figures 36 & 37).



```python
import json

from django.contrib.auth.models import User
from channels.generic.websocket import AsyncWebsocketConsumer
from asgiref.sync import sync_to_async

from .models import Room, Message

class ChatConsumer(AsyncWebsocketConsumer):
    async def connect(self):
        self.room_name = self.scope['url_route']['kwargs']['room_name']
        self.room_group_name = 'chat_%s' % self.room_name

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

        await self.accept()

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

    # Receive message from WebSocket
    async def receive(self, text_data):
        data = json.loads(text_data)
        print(data)
        message = data['message']
        username = data['username']
        room = data['room']

        await self.save_message(username, room, message)


        # Send message to room group
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'chat_message',
                'message': message,
                'username': username
            }
        )
```

Figure 36: consumers.py (1).



```python
# Receive message from room group
async def chat_message(self, event):
    message = event['message']
    username = event['username']

    # Send message to WebSocket
    await self.send(text_data=json.dumps({
        'message': message,
        'username': username
    }))

@sync_to_async
def save_message(self, username, room, message):
    user = User.objects.get(username=username)
    room = Room.objects.get(slug=room)

    Message.objects.create(user=user, room=room, content=message)
```

Figure 37: consumers.py (2).

Inside the class *ChatConsumer*, first we create an asynchronous view, *connect,* which is called anytime the app is connected to the WebSocket.
Inside this view, based on the URL we are trying to connect to, we are getting the room name from it as well as setting the *room_group_name* to have the prefix 'chat' before the room name itself. After this, we join the two variables, *self.channel_name* and *self.room_group_name* into the *channel_layer* using the predefined *group_add()* method*.* Finally, we use *await self.accept()* to connect to the server.

After this we define a *disconnect* function and pass in the two aforementioned variables into it using the built-in *group_discard()* method.

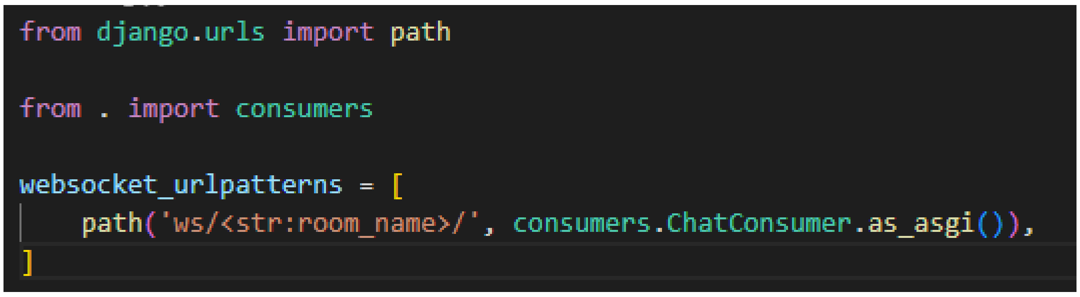
Next up, the function to receive text messages from the WebSocket is defined.
In this asynchronous function called *receive* which takes two parameters, *self* and *text_data,* JSON is used to load the JSON content of the variable *text_data.* From this we can get the *message, username* and *room* properties.

Then, this information is sent to the room group using the built-in method *group_send()*. We are sending the contents of the *self.room_group_name* variable, and the object we are sending is of a 'chat-message' type, and we are sending the contents of the *message*, *username* and *room* variables.

Furthermore, a new method is created called *chat_mesasage* which takes two parameters, *self* and *event* which contains the three variables sent in the previous function*.* This method receives the content from the *message* and *username* variables and sends them to the WebSocket by converting them to a JSON format.
Finally, a decorator *@sync_to_async* is defined which enables the ability to store all the information

into the database. Inside this decorator, the function *save_message* is defined which takes the parameters *self, username, room* and *message.* This function gets the User and Room objects from the database based on what is sent from the frontend. For the User, the username is sent from the frontend, whilst *slug* is sent from the frontend for the Room.
Finally we create a new *Message* object which is connected to the user, the room and the *content* variable from the model is equal to the *message* variable.

Lastly, inside the *routing.py* file the consumer is imported and the URL patterns are defined (shown in Figure 38).

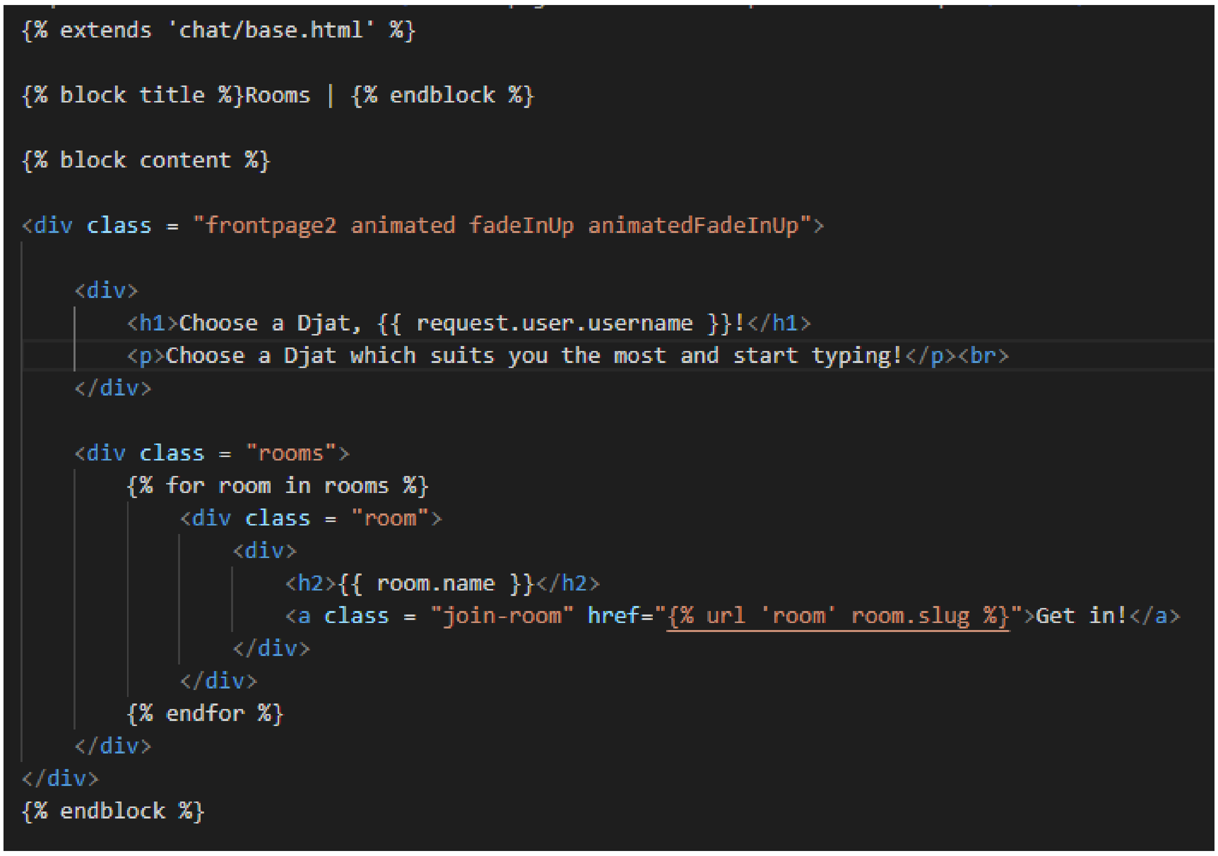


```
from django.urls import path

from . import consumers

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

Figure 38: routing.py.

A variable *websocket_urlpatterns* is defined, which is a list of paths. The path we are accessing is using the WebSocket (as seen by 'ws') and *room_name*, using our previously created consumer *ChatConsumer* and doing so as an Asynchronous Server Gateway Interface.

The two template files used in the *room* application are *room.html* and *rooms.html*.
Rooms.html is the template file which displays the room/channel list, whilst *room.html* is the template file which displays the chat aspect of each room (shown in Figure 39).

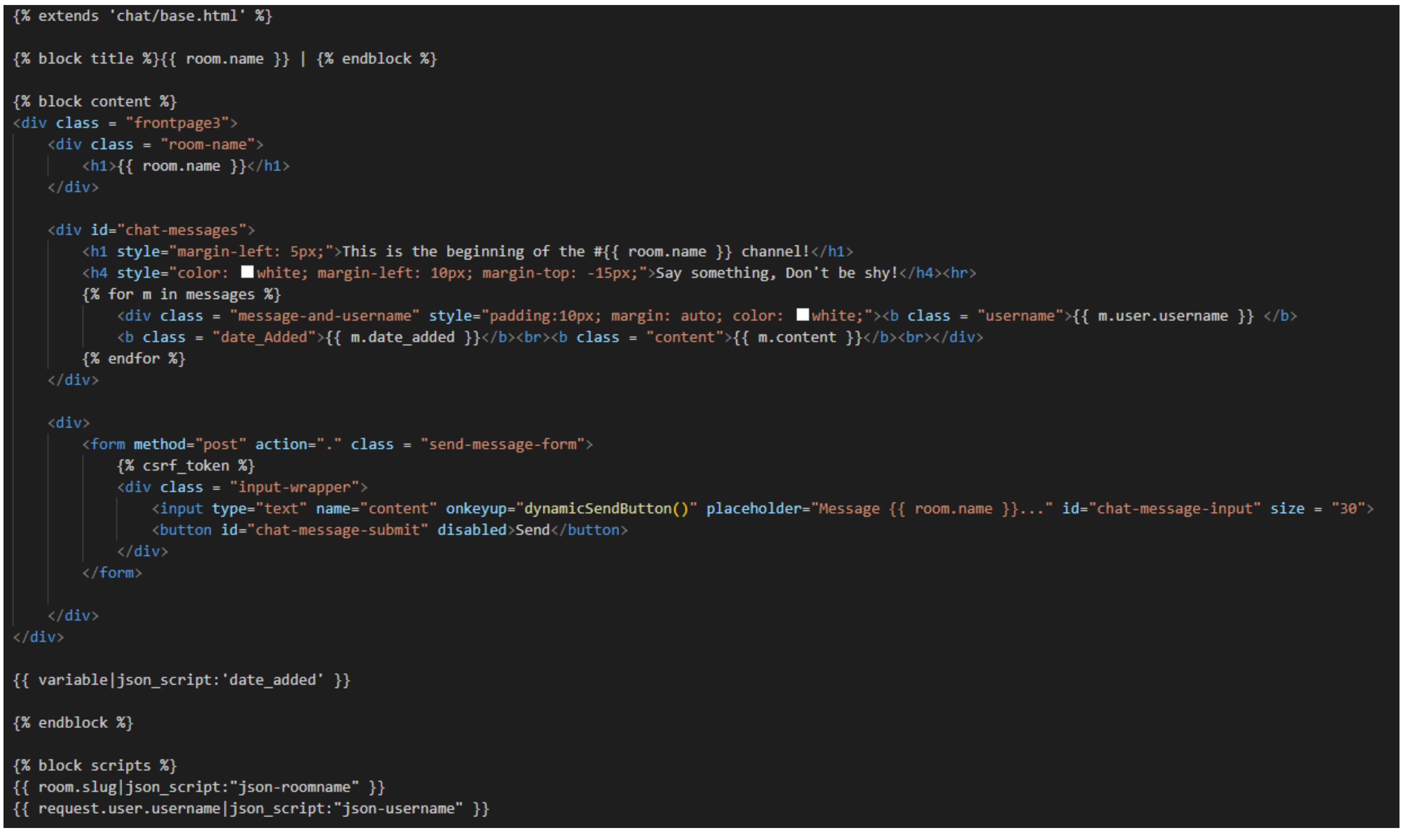


```
{% extends 'chat/base.html' %}

{% block title %}Rooms | {% endblock %}

{% block content %}

<div class = "frontpage2 animated fadeInUp animatedFadeInUp">

    <div>
        <h1>Choose a Djat, {{ request.user.username }}!</h1>
        <p>Choose a Djat which suits you the most and start typing!</p><br>
    </div>

    <div class = "rooms">
        {% for room in rooms %}
            <div class = "room">
                <div>
                    <h2>{{ room.name }}</h2>
                    <a class = "join-room" href="{% url 'room' room.slug %}">Get in!</a>
                </div>
            </div>
        {% endfor %}
    </div>
</div>
{% endblock %}
```

Figure 39: rooms.html.

On the *rooms.html* page, the user is greeted by a short welcome message underneath which is a list of available rooms/channels for the user to join.
This is done using the Django template language by looping through the values in the *Room* model.
Also, the Django template language is used to display the room name under which is a link to the specific room.

Depending on the slug of the room, when the user clicks the link they are taken to the specific room. Here, the room name is displayed again, however underneath it is a chat box filled with messages. Under the chat box is an input field for typing a message and a ‘Send’ button (shown in Figure 40).

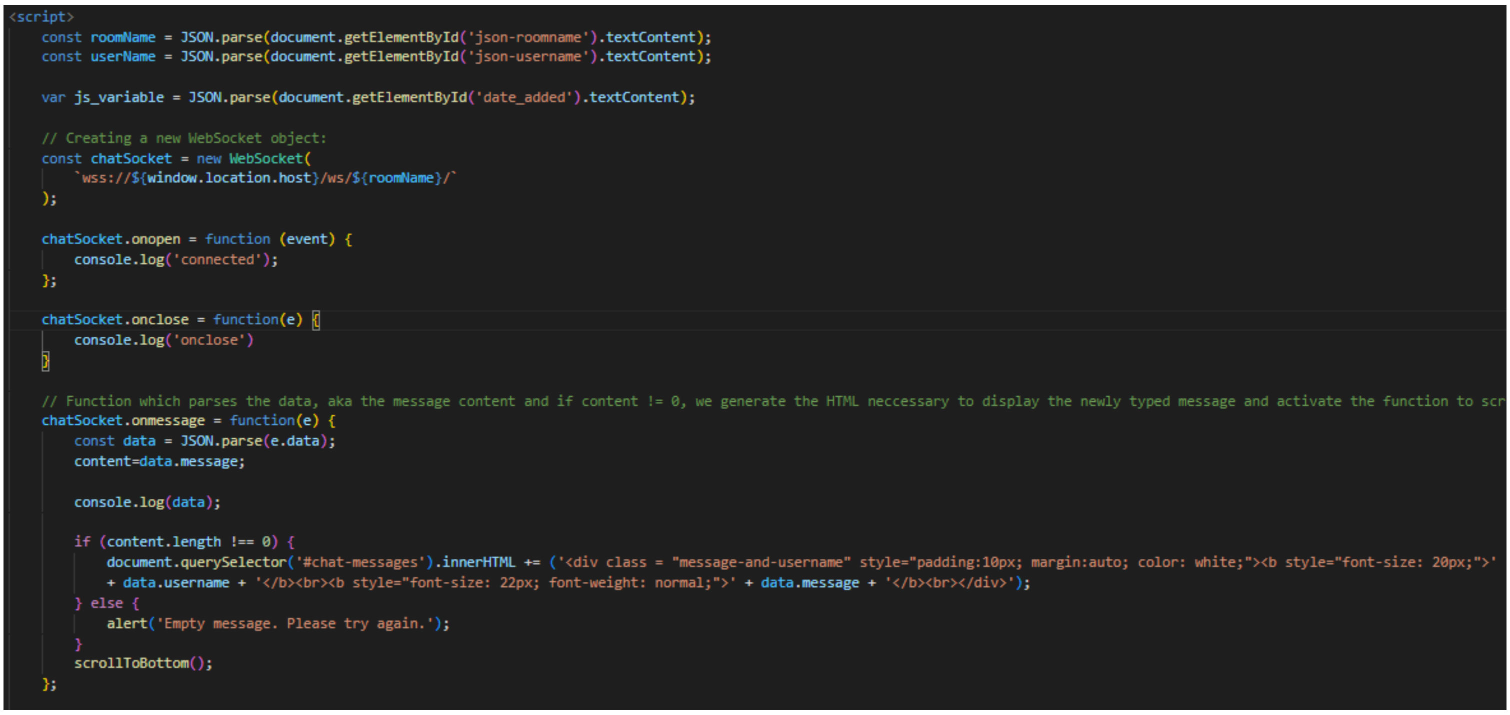


```
{% extends 'chat/base.html' %}

{% block title %}{{ room.name }} | {% endblock %}

{% block content %}
<div class = "frontpage3">
    <div class = "room-name">
        <h1>{{ room.name }}</h1>
    </div>

    <div id="chat-messages">
        <h1 style="margin-left: 5px;">This is the beginning of the #{{ room.name }} channel!</h1>
        <h4 style="color: white; margin-left: 10px; margin-top: -15px;">Say something, Don't be shy!</h4><hr>
        {% for m in messages %}
            <div class = "message-and-username" style="padding:10px; margin: auto; color: white;"><b class = "username">{{ m.user.username }} </b>
            <b class = "date_Added">{{ m.date_added }}</b><br><b class = "content">{{ m.content }}</b><br></div>
        {% endfor %}
    </div>

    <div>
        <form method="post" action="." class = "send-message-form">
            {% csrf_token %}
            <div class = "input-wrapper">
                <input type="text" name="content" onkeyup="dynamicSendButton()" placeholder="Message {{ room.name }}..." id="chat-message-input" size = "30">
                <button id="chat-message-submit" disabled>Send</button>
            </div>
        </form>

    </div>
</div>

{{ variable|json_script:'date_added' }}

{% endblock %}

{% block scripts %}
{{ room.slug|json_script:"json-roomname" }}
{{ request.user.username|json_script:"json-username" }}
```

Figure 40: room.html.

Similarly to the *rooms.html* file, using the Django template language the messages are displayed by looping through the values of the *Message* model, as well as the accompanying username and date sent.

At the bottom, the Django template language is used again to define the JSON scripts used so that the JavaScript script can be successfully written (shown in Figures 41 & 42).



```
<script>
    const roomName = JSON.parse(document.getElementById('json-roomname').textContent);
    const userName = JSON.parse(document.getElementById('json-username').textContent);

    var js_variable = JSON.parse(document.getElementById('date_added').textContent);

    // Creating a new WebSocket object:
    const chatSocket = new WebSocket(
        `wss://${window.location.host}/ws/${roomName}/`
    );

    chatSocket.onopen = function (event) {
        console.log('connected');
    };

    chatSocket.onclose = function(e) {
        console.log('onclose')
    }

    // Function which parses the data, aka the message content and if content != 0, we generate the HTML neccessary to display the newly typed message and activate the function to scr
    chatSocket.onmessage = function(e) {
        const data = JSON.parse(e.data);
        content=data.message;

        console.log(data);

        if (content.length !== 0) {
            document.querySelector('#chat-messages').innerHTML += ('<div class = "message-and-username" style="padding:10px; margin:auto; color: white;"><b style="font-size: 20px;">'
            + data.username + '</b><br><b style="font-size: 22px; font-weight: normal;">' + data.message + '</b><br></div>');
        } else {
            alert('Empty message. Please try again.');
        }
        scrollToBottom();
    };
```

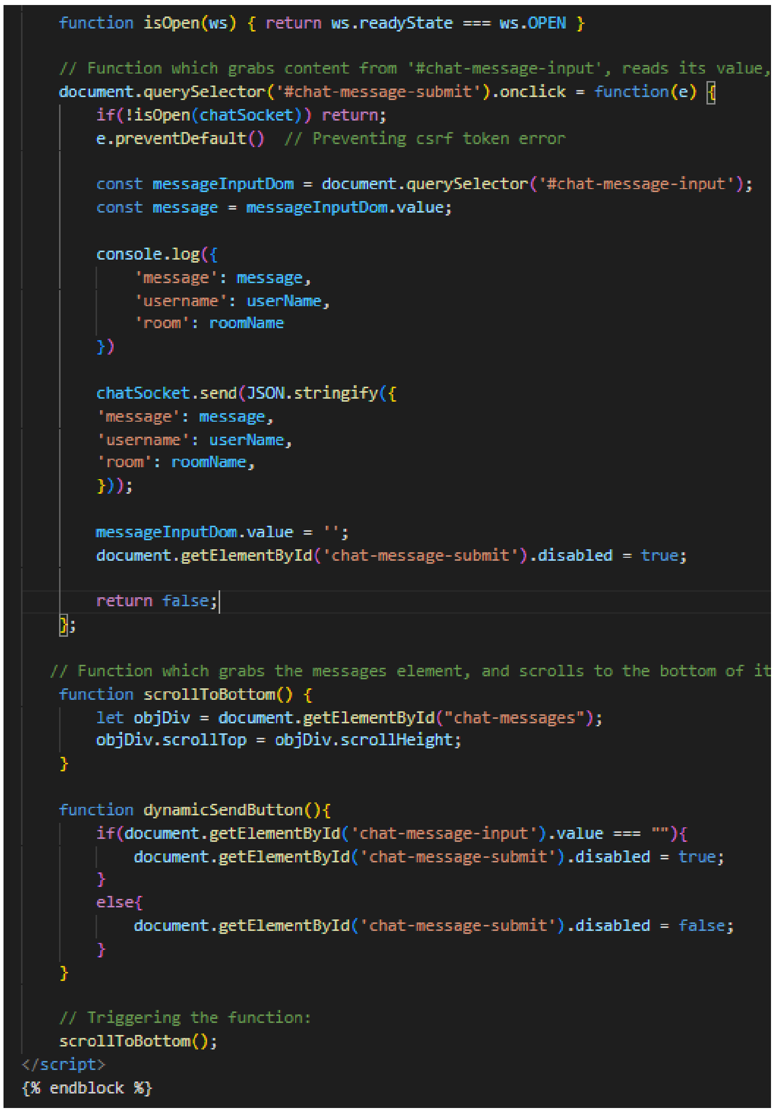
Figure 41: room.html script (1).

Inside the script, two variables are defined, *roomName* and *username.* These variables, using 'JSON.parse()' parse through our JSON scripts containing the room, message and username.

After that, a new WebSocket object is created. The connection uses an URL containing the given address using the bidirectional connection between the server and client.

Then we print into the console the status of the WebSocket. If the WebSocket is open, 'connected' is printed into the console. If the WebSocket is not connected, 'onclose' is printed into the console.

Next, we define a function which is called every time a message is sent. This function parses through the value of the variable *data* and if the length of the variable isn't null, the content of the *message*, *username* and *date_sent* variables are generated. This is done by creating a simple HTML div containing the values.



```
    function isOpen(ws) { return ws.readyState === ws.OPEN }

    // Function which grabs content from '#chat-message-input', reads its value,
    document.querySelector('#chat-message-submit').onclick = function(e) {
        if(!isOpen(chatSocket)) return;
        e.preventDefault()  // Preventing csrf token error

        const messageInputDom = document.querySelector('#chat-message-input');
        const message = messageInputDom.value;

        console.log({
            'message': message,
            'username': userName,
            'room': roomName
        })

        chatSocket.send(JSON.stringify({
        'message': message,
        'username': userName,
        'room': roomName,
        }));

        messageInputDom.value = '';
        document.getElementById('chat-message-submit').disabled = true;

        return false;
    };

   // Function which grabs the messages element, and scrolls to the bottom of it
    function scrollToBottom() {
        let objDiv = document.getElementById("chat-messages");
        objDiv.scrollTop = objDiv.scrollHeight;
    }

    function dynamicSendButton(){
        if(document.getElementById('chat-message-input').value === ""){
            document.getElementById('chat-message-submit').disabled = true;
        }
        else{
            document.getElementById('chat-message-submit').disabled = false;
        }
    }

    // Triggering the function:
    scrollToBottom();
</script>
{% endblock %}
```

Figure 42: room.html script (2).

Moreover, when the user clicks on the 'Send' button, a function is called. This function grabs the element using *document.querySelector* and its value.
Then, the message content, username and room name are printed into the console as well as sent to the WebSocket using *JSON.stringify().* After that, the value inside the input field is reset and the 'Send' button is disabled.

Lastly, two functions are defined, *scrollToBottom()* and *dynamicSendButton().*
The *scrollToBottom()* function grabs the HTML div element containing the messages and scrolls to the bottom of it offsetting the top scroll position and height.
This function is called after every newly sent message and at every page load.

The second function, *dynamicSendButton()* changes the state of the 'Send' button depending on whether the contents of the input field are empty or not.
If the content is empty, the button is disabled. If the content is not empty, i.e the user has entered something in the input field, the button is enabled.

5 Conclusion

The topic of this thesis was the creation of a Django chat application. An insight into Django itself, its structure, usage, advantages and disadvantages was given. The created application ‘Djat’ was also described, as well as its UI, functions and code.

Django is a useful, lightweight tool that can be used to create fast and reliable web applications. The abundance of modules and packages makes it easy to write simple, easy-to-understand Python code.

This paper shows how to create a simple Django chat application. Knowledge of URL’s, WebSocket’s and networks is needed to create a stable connection capable of sending, receiving and storing messages.

for the frontend, JavaScript, HTML and CSS were used.

This project successfully demonstrated why Django is a useful tool for web application development.

Link to GitHub repo: IvanMatejcic/Djat: Finals project made using Django. (github.com)

6 Literature

[1] Wikipedia, *Web framework, 23rd August 2022*. Accessed August 2022:
https://en.wikipedia.org/wiki/Web_framework

[2] Noel Greene, Evolve, *What is a web framework?* Accessed August 2022:
What is a web framework? - evolve evolve

[3] W3Schools, *Django Views.* Accessed August 2022:
Django Views (w3schools.com)

[4] W3Schools, *Django Getting Started.* Accessed September 2022:
https://www.w3schools.com/django/django_getstarted.php

[5] Andrew Godwin, Channels Documentation, *Introduction*. Accessed August 2022:
Introduction — Channels 3.0.5 documentation

[6] Rubin Raithel, Channels Documentation, *Consumers.* Accessed August 2022:
https://channels.readthedocs.io/en/stable/topics/consumers.html

[7] Django Documentation, *Models.* Accessed September 2022:
Models | Django documentation | Django (djangoproject.com)

[8] Django Documentation, *The Django template Language.* Accessed August 2022:
The Django template language | Django documentation | Django (djangoproject.com)

[9] Collin Anderson, Channels Documentation, *Tutorial 2: Implement a Chat Server.* Accessed August 2022:
Tutorial Part 2: Implement a Chat Server — Channels 3.0.5 documentation

[10] Django Documentation, *Writing your first Django app, part 1.* Accessed September 2022:
Writing your first Django app, part 1 | Django documentation | Django (djangoproject.com)

[11] 13 years ago, StackOverflow, *What is a 'slug' in Django?* Accessed August 2022:
python - What is a "slug" in Django? - Stack Overflow

[12] W3Schools, *Django Introduction.* Accessed September 2022:
Introduction to Django (w3schools.com)

[13] Raoof Naushad, Medium, *Difference between WSGI and ASGI?* Accessed September 2022:
Difference between WSGI and ASGI ? | by Raoof Naushad | Analytics Vidhya | Medium

[14] Packtpub, *How does Django work?* Accessed September 2022:
How does Django work? | Django Design Patterns and Best Practices - Second Edition (packtpub.com)

[15] Bharath thippireddy, Youtube.com, *Model View Template Design Pattern.* Accessed September 2022:
Model View Template Design Pattern - YouTube

[16] Packtpub, *The story of Django.* Accessed September 2022:
The story of Django | Django Design Patterns and Best Practices - Second Edition (packtpub.com)

[17] Wikipedia, *WebSocket.* Accessed September 2022:
WebSocket - Wikipedia

[18] Statista, *Most popular global mobile messenger apps as of January 2022, based on number of monthly active users. Accessed September 2022:*
Most popular messaging apps 2022 | Statista

8. List of images