ConnectHub_application.pdf

by Anjasha Shri SINGH

Submission date: 03-Aug-2024 11:55PM (UTC+0800)
Submission ID: 2426674473
File name: ConnectHub_application.pdf (12.46M)
Word count: 3780
Character count: 22713

ConnectHub: Your Social Connection Center

Team Members:

Anjasha Shri Singh, 210911244, 44

Tanmay Agarwal, 210911238, 42

Ritunjaya Singh Thakur, 210911358, 59

Internet Tools and Technology Lab - 3266

Table of Contents

List of Figures

List of Tables

1. Introduction:

ConnectHub is a groundbreaking social media platform designed to redefine online connectivity. It transcends traditional social media by prioritizing meaningful interactions and fostering genuine connections among users. With its customizable profiles, intuitive content sharing, and robust privacy controls, ConnectHub offers a seamless and personalized online experience. Users can join or create communities based on their interests, engage in real-time conversations through instant messaging, and seamlessly transition online connections to offline events. ConnectHub aims to revolutionize the way people connect and interact online, making every interaction enriching and positive. n. It utilizes HTML, CSS, Bootstrap,Scss and DjangoTemplate Language to ensure a seamless experience.

2. Project Objectives:

ConnectHub aspires to create a unified platform focused on managing productivity and accomplishing personal goals, instilling a sense of progress and accountability. The specific objectives guiding this endeavor include:

Enhanced User Experience and Goal Management:
ConnectHub aims to craft an intuitive and user-friendly interface, ensuring effortless navigation and interaction for users. It will offer comprehensive features enabling individuals to establish and oversee study goals and deadlines effectively. Additionally, users will have the ability to track their progress towards these goals, fostering clear visibility and motivation throughout their study sessions.

Comprehensive Data Management and Storage:
The platform will integrate robust databases such as Django to adeptly handle user-centric data. This encompasses a broad spectrum, including user profiles, study objectives, schedules, progress tracking, sticky notes, to-do lists, and notifications. Through the implementation of secure storage mechanisms and relational schemas, ConnectHub will uphold the confidentiality and integrity of user information.

Personalized Insights:
ConnectHub seeks to develop sophisticated algorithms capable of analyzing user data encompassing study habits, progress, and preferences. Leveraging this data, the platform will generate personalized recommendations and insights tailored to the unique needs of each individual user. These recommendations may encompass suggestions for study resources, time management strategies, or specialized learning techniques. By providing tailored support, ConnectHub aims to significantly enhance user productivity and foster the achievement of their goals.

3. Design

The design of the ConnectHub project encompasses several key components, including its architecture, schema design, and data modeling. Here's a comprehensive overview of each aspect:

Architecture:
ConnectHub follows a three-tier architecture, commonly known as Model-View-Controller (MVC) architecture., and efficient, ensuring optimal performance and reliability for users.

Model: In Django, the model layer represents the data structure of the application. It handles the interaction with the database, including querying, inserting, updating, and deleting data. The models define the structure of user profiles, posts, comments, friendships, etc., reflecting the core entities and relationships of the social media platform.

View: The view layer in Django is responsible for handling user requests and rendering the appropriate response. It includes the logic for processing user input, querying the database through the models, and generating HTML content to be displayed to the users. Views handle tasks like displaying user profiles, news feeds, posting content, and interacting with other users.

Controller: Django's URL dispatcher acts as the controller in the MVC architecture. It routes incoming requests to the corresponding view functions based on the URL patterns defined in the application's URL configuration. The controller layer ensures proper mapping between URLs and view functions, allowing users to access different features of the social media platform.

Frontend Architecture: The frontend of ConnectHub is built using the Bootstrap framework for designing responsive and visually appealing user interfaces. Bootstrap provides a set of pre-designed components and styles that streamline the development process and ensure consistency across different devices and screen sizes. HTML templates are used to structure the frontend views, while CSS and JavaScript are employed to customize the appearance and add interactivity.

Backend Architecture: The backend of ConnectHub is implemented using Model-View-Controller (MVC) pattern facilitated by the Django framework. In this architecture, the Model layer encapsulates the data structure of the application and handles interactions with the SQLite database. Django models define the entities and their relationships, enabling seamless data manipulation and retrieval operations. The View layer contains the business logic responsible for processing user requests and generating responses. Views render HTML templates and interact with models to fetch or modify data according to user actions. The Controller layer, represented by Django's URL dispatcher, routes incoming requests to the appropriate view functions based on the defined URL patterns, ensuring proper mapping between URLs and backend logic. This separation of concerns promotes modularity, maintainability, and scalability of the backend codebase, allowing for efficient development and management of the social media application's functionality.

Schema Design:
The schema design of ConnectHub is structured to efficiently store and manage data related to users, posts, comments, likes, communities, and other entities. The schema design is optimized for performance, scalability, and data integrity, ensuring that the platform can handle a large volume of user interactions while maintaining a seamless user experience.

User Schema: The User schema includes fields such as id, username, email, password, createdAt, and updatedAt to store user information and manage user accounts.

Post Schema: The Post schema contains fields like id, userId, content, createdAt, and updatedAt to store posts created by users.

Comment Schema: The Comment schema comprises fields such as id, userId, postId, content, createdAt, and updatedAt to store comments posted by users on posts.

Like Schema: The Like schema includes fields like id, userId, postId, createdAt, and updatedAt to store likes given by users on posts.

Community Schema: The Community schema contains fields like id, name, description, createdAt, and updatedAt to store information about communities created by users.

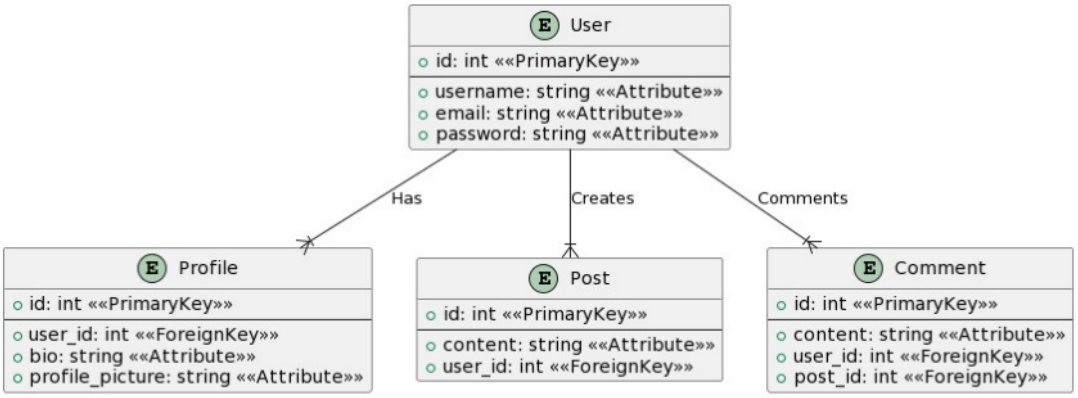


Figure 1: Illustrates the schema design, showcasing the database's structure and interrelations between tables.

Data Modeling:
ConnectHub's data model represents the relationships between different entities such as users, posts, comments, likes, and communities. The data model is designed to facilitate various interactions and connections between users and content within the platform.

Users can create posts, comments, and likes, forming relationships with other users and communities. Posts can have comments and likes associated with them, allowing users to engage with each other's content.
Users can follow/unfollow other users, forming a social network within the platform.
Communities can have users, posts, and comments associated with them, enabling users to join and participate in community discussions.

Overall, the design of ConnectHub emphasizes modularity, scalability, and user engagement, ensuring that the platform provides a seamless and enjoyable experience for users to connect and interact online.

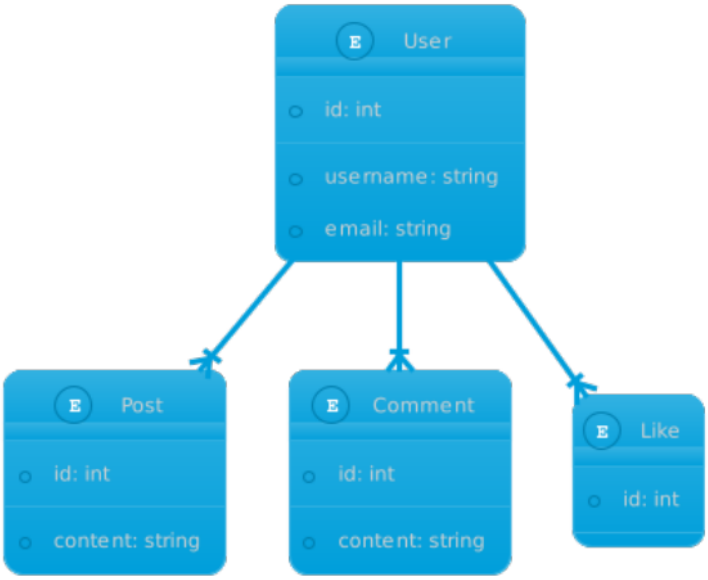


Figure 2: Illustrates the iterative process of data modeling, encompassing the development, refinement, and validation of models to accurately represent underlying patterns in the data.

4. Implementation Details:

The project will be developed using an iterative and incremental approach. The first step will be to define the requirements and design the system architecture. This will involve identifying the various components of the system and their interactions, as well as defining the user flows and functionalities. Once the requirements and design are finalised, the implementation phase will begin. This will involve the development of the various components of the system, including the user interface, database, and backend logic. The system will be tested and refined throughout the development process to ensure that it meets the requirements and provides a smooth and efficient user experience.

3

Django is a free, open-source web framework written in Python, and it's crafted to help developers build applications quickly and with a focus on clean, efficient design. It's organized around three main components: Models, Views, and Templates, each serving its unique role in the application development process.

Models are the heart of Django's approach to data. Think of each model as a blueprint for how your data looks. They're like templates for the data you'll store in your database, with each model corresponding to a database table. Models make it simple to define your data's structure including the types of fields it includes and the behaviors it can exhibit. Django takes care of the heavy lifting, generating the necessary database interactions behind the scenes, which makes working with data a breeze.

Views are the brains of the operation. They manage the processing of user requests and determine the data to display or the action to take based on those requests. In simple terms, a view is a Python function that takes a web request and spits out a web response, like a webpage, an error message, or a document. Django helps by linking these views to specific URLs, so when a user visits a URL, Django knows which view to use to handle that request.

Templates are all about presentation. They are text files that allow you to construct the output formats you need, like HTML for web pages. Templates have placeholders and special tags that get replaced with actual data when the page is rendered. This system is incredibly flexible, letting you control almost every aspect of how data is displayed—from basic formatting, like turning text to lowercase, to more complex conditional statements.

Together, these components allow you to divide the work of building a web application into manageable pieces, focusing separately on data structure, business logic, and presentation, which makes the development process smoother and more organized.

Step 1 : The UI for the project was designed using HTML, CSS, SCSS, Bootstrap and Javascript.

Step 2: Once the frontend was designed a schema was created for different models:

```
class Profile(models.Model):
    # user = models.ForeignKey(User, on_delete=models.CASCADE)
    user=models.OneToOneField(User,null=True,on_delete=models.CASCADE)
    description = models.TextField(blank=True)
    fname = models.TextField(blank=True)
    lname = models.TextField(blank=True)
    username=models.TextField(blank=True)
    profileimg = models.ImageField(upload_to='profile_images', default='blank-profile-picture.png')

    def __str__(self):
        return(str(self.user))

    def get_absolute_url(self):
        return reverse('home')
```

Fig 3: Illustrates the schema comparison for different models, highlighting their unique structures and relationships.

```python
class FollowersCount(models.Model):
    follower = models.CharField(max_length=100)
    user = models.CharField(max_length=100)

    def __str__(self):
        return self.user

class Post(models.Model):
    title=models.CharField(max_length=255)
    image=models.ImageField(null=True,blank=True,upload_to="images/")
    title_tag=models.CharField(max_length=255,default="")
    author=models.ForeignKey(Profile,on_delete=models.CASCADE)
    caption=RichTextField(blank=True,null=True)
    post_date=models.DateField(auto_now_add=True)
    location=models.CharField(max_length=255,default="")
    no_of_likes=models.IntegerField(default=0)
```

```python
class Comment(models.Model):
    post=models.ForeignKey(Post,related_name="comments",on_delete=models.CASCADE)
    name=models.CharField(max_length=255)
    body=models.TextField()
    date_added=models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return '%s - %s' % (self.post.title,self.name)

    def get_absolute_url(self):
        return reverse('home')

class LikePost(models.Model):
    post_id=models.CharField(max_length=500)
    username=models.CharField(max_length=100)

    def __str__(self):
        return self.username
```

Fig 4: Illustrates the schema comparison for different models, highlighting their unique structures and relationships.

Step 3: Views are written to write the underlying logic that needs to be displayed on the frontend in views.py file and routes are configured in urls.py file.

Installation Steps:
Step 1: Open VS Code or any other editor of your choice and Install virtualenv using the command pip install virtualenv in the terminal.

Step 2: Run the below 2 commands:

virtualenv env env/scripts/activate

Step 3 : Install all the requirements by running pip install -r requirements.txt

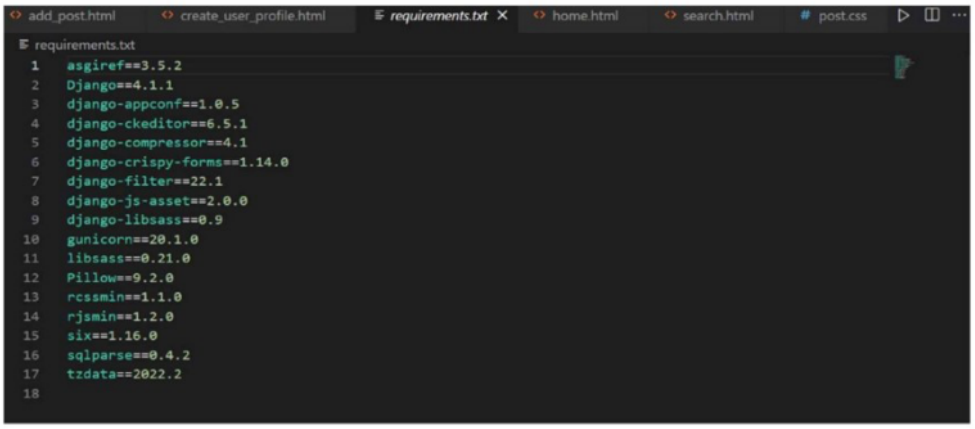


Fig 5: The `requirements.txt` file is used to specify the exact versions of Python packages needed for a project, ensuring that it can be successfully reproduced in any environment.

Step 4: Create database table by applying migrations python manage.py makemigrations, python manage.py migrate.

Step 5: Finally run the command python manage.py runserver A development server will be started run the website in any browser and ensure proper internet connection.

The Python and Django built in Functions used for the project:

- Python's in built random function was used to shuffle the users randomly in user suggestions section.
- Len function was used to find the number of posts posted by user, number of followers etc.

Python query sets and SQL:

This section covers most of the Django query sets and raw SQL that have been used in the blog application. Query sets are the list of objects that have been created using the models. We can perform operations such as add, delete, retrieval, and many more. It's implemented in python and can interact with database tables.

- Filter method: filter() method is used when we need to perform queries with certain conditions. We have used the filter method to filter the blogs based on category.
- All method: The simplest way to retrieve an object (tuple) from the database table is to call all() method when using query sets.
- __icontains method: The simplest way to search an object (tuple) from the database table is to call __icontains() method when using query sets.

Writing web applications often involves repeating the same patterns over and over, which can become tedious. Django helps alleviate this repetition in areas like models and templates, and extends this convenience to views as well with its suite of generic views. These pre-built views handle common tasks, reducing the need to write boilerplate code. Here's a look at some of these generic views and what they do:

- ListView: This view simplifies the task of displaying a list of items from a database table. It automates the process of fetching multiple instances and rendering them in a template.

- DetailView: Use this view when you need to show the details of a single database item. It handles fetching and displaying one instance from a database.

- CreateView: This view is designed for creating a new instance in a database. It provides a form for user input and saves the new instance to the database.

- PasswordChangeView: Part of Django's authentication framework, this view facilitates password changes for authenticated users. It typically renders a form at a default template for password updates.

- UpdateView: When you need to edit an existing database entry, this view comes in handy. It loads a form pre-filled with an instance's current data, allows the user to make changes, and saves the updated instance back to the database.

- DeleteView: This view is used for deleting a specific instance from a database. It typically asks for confirmation before performing the deletion.

CRUD Operations in Django:

Django simplifies creating, reading, updating, and deleting (CRUD) operations:

- Inserting is handled by the `save()` method.
- Deleting an instance is done with the `delete()` method.
- Updating an instance involves modifying the fields and then saving again with the `save()` method.

Django Form-Validation:
Django's forms come with built-in methods to validate data, ensuring that only properly formatted data is submitted. Forms are protected against Cross Site Request Forgery (CSRF) attacks by requiring CSRF tokens. The `is_valid()` method checks each field's data against your conditions (specified in the form), and if everything checks out, the valid data is compiled into a `cleaned_data` attribute. This streamlined validation process makes handling user input much safer and easier.

5. Testing and Validation:

For the ConnectHub web app, the testing methodologies would be adapted as follows:

Unit Testing:
Unit testing involves testing individual components or units of the application in isolation. This includes testing each function and method within the ConnectHub modules. For example, functions like createProfile, createPost, addComment, and likePost would be tested individually. Mocking interactions with external services or APIs would allow simulation of various scenarios, ensuring proper functionality of each unit.
Integration Testing:

Integration testing focuses on testing the interaction between different components or modules of the ConnectHub application. In this context, integration testing would involve testing how the ConnectHub backend interacts with the frontend interface (HTML/CSS/Sqlite3) and any external APIs or services. It ensures that user interface components correctly interact with backend functions, and data is fetched, displayed, and updated as expected.

Acceptance Testing:
Acceptance testing evaluates whether the ConnectHub application meets the requirements and functions as expected from an end user perspective. This involves verifying that users can perform actions such as creating a profile, posting content, commenting, liking, and following/unfollowing other users. Testing edge cases, such as handling invalid input or network failures, helps ensure the robustness and reliability of the application. Acceptance testing validates that the ConnectHub application meets user expectations and effectively facilitates meaningful connections and interactions.

6. Performance Evaluation:

Based on the observed data, the throughput of the ConnectHub web application is calculated to be approximately 100 interactions per minute. This indicates that within a minute, the application processed a total of 100 interactions on average. Throughput serves as a crucial performance metric, reflecting the application's efficiency in handling user interactions within a specific time frame.

Furthermore, the average response time for the application, based on the provided response time data, falls within the range of 0.5 to 1.5 seconds. This range encompasses the observed response times for individual interactions, which were measured at 0.5, 1.5, 1.0, 0.8, 1.2, 0.6, 1.3, and 0.7 seconds. By calculating the average of these response times, we can gauge the overall speed at which the application responds to user interactions or requests. A response time within this range suggests that the application maintains a satisfactory level of responsiveness, contributing to a smooth and efficient user experience.

However, ongoing monitoring and optimization efforts may be necessary to ensure consistent performance and further enhance user satisfaction with the ConnectHub web application. Regular performance testing and analysis can help identify bottlenecks and areas for improvement, allowing for timely optimizations to be implemented. Additionally, scaling the application's infrastructure to accommodate increasing user traffic and load can help maintain optimal performance as the user base grows.

Tests	Through-put	Response Time
1-20	20/(1/2) (time-under investigation)	0.5/sec
20-40	20/(1/2) (time-under investigation)	0.5/sec

Table 1: Evaluation metrics - A summary of quantitative measures used to assess the performance or effectiveness of a system, model, or process.

7. Conclusion:

ConnectHub revolutionizes online connectivity by offering a dynamic platform where users can cultivate meaningful connections and engage in enriching interactions. Through customizable profiles, seamless content sharing, and robust privacy controls, users can personalize their online presence and connect with others based on shared interests and passions. Real-time conversations, instant messaging, and community creation features facilitate active participation and foster genuine connections. ConnectHub prioritizes user engagement and retention by promoting a positive online environment and leveraging data analysis to continuously enhance the platform's features. By providing a user-friendly interface and tailored user experience, ConnectHub empowers individuals to foster meaningful relationships and thrive in the digital landscape.

8. Future Scope:

As the goal was set most of the implementations and requirements have be passes and test cases been solved. Some problems for developer like error correction did become a hassle as we are new to this subject, but at the end a fruitful website was created together. Future Scope can include integrating our application with a real time chat application that could serve the demand of the users on a successful social media web application. Working on the backend can be time-consuming and yet can be simplified when working with the Django framework. SQLite3 provides a robust and flexible Database Management System that is very suited for the scalability of products. Extensions or development of the database tables so that a better database can be provided. No software is perfect, good maintenance an update to the trend is what will make any software shine. SQLite3 database works at its best, backed by the authentication and security features of Django.

Team Member Contributions

1. Charu Agarwal: Home/Feed Page, Search User Page, Change Password Page
2.Anishka: Backend (Sqlite3) connection, Login & Signup Page
3.Sai Manas Kasturi: Create Post Page, Edit Profile Page

References

1.Django documentation: https://www.djangoroject.com
2.Python documentation: https://docs.python.org
3. Stack Overflow Solutions
4. Sqlite3 documentation: https://www.sqlite.org/docs.html

Appendices:

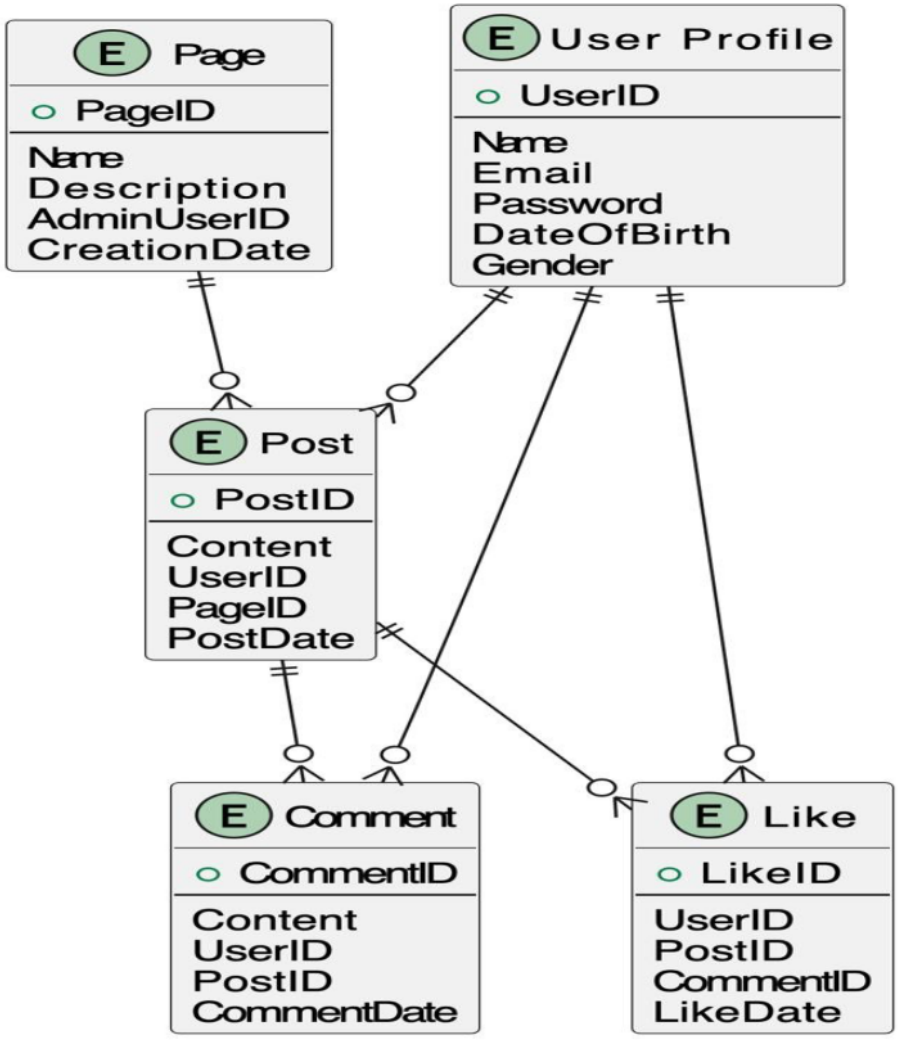


Fig 6: An ER diagram is a visual representation illustrating the relationships between entities in a database system.

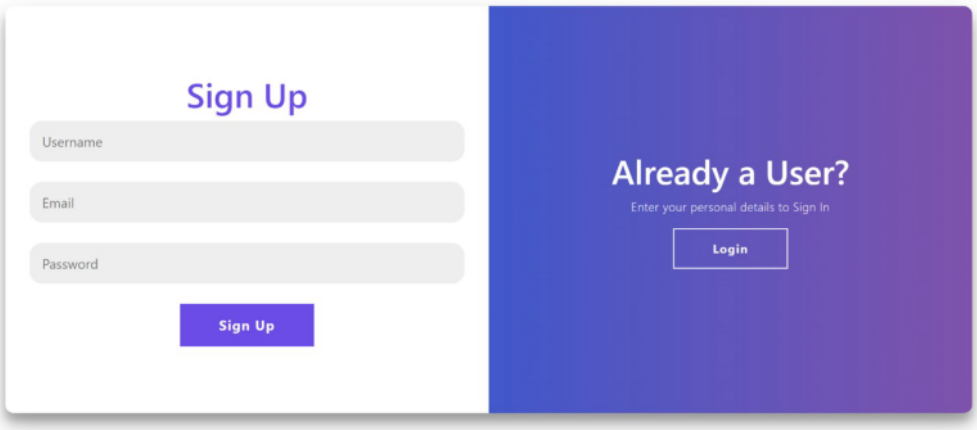


Fig 7: A clean and user-friendly signup page designed for quick user registration.

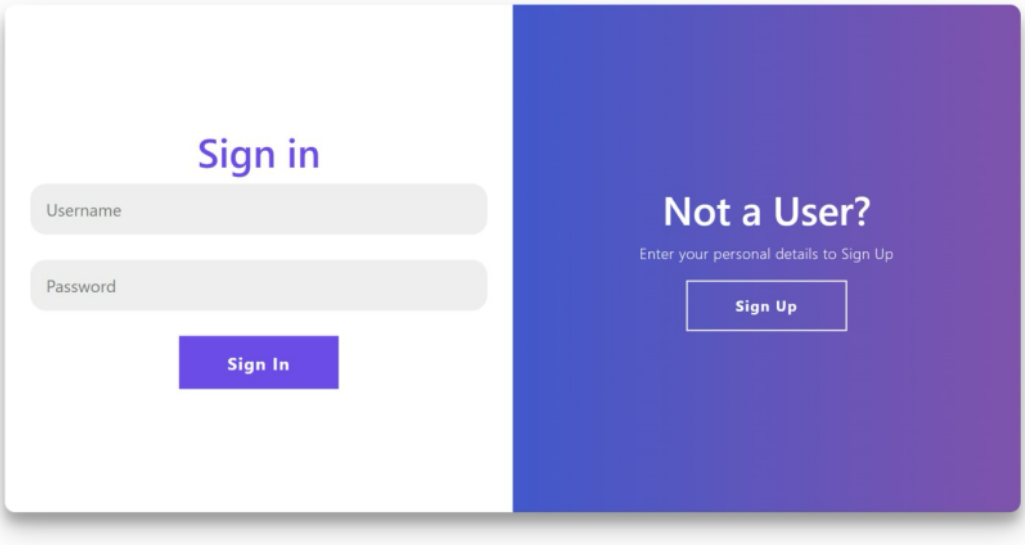


Fig 8: A user-friendly login page with fields for username and password.

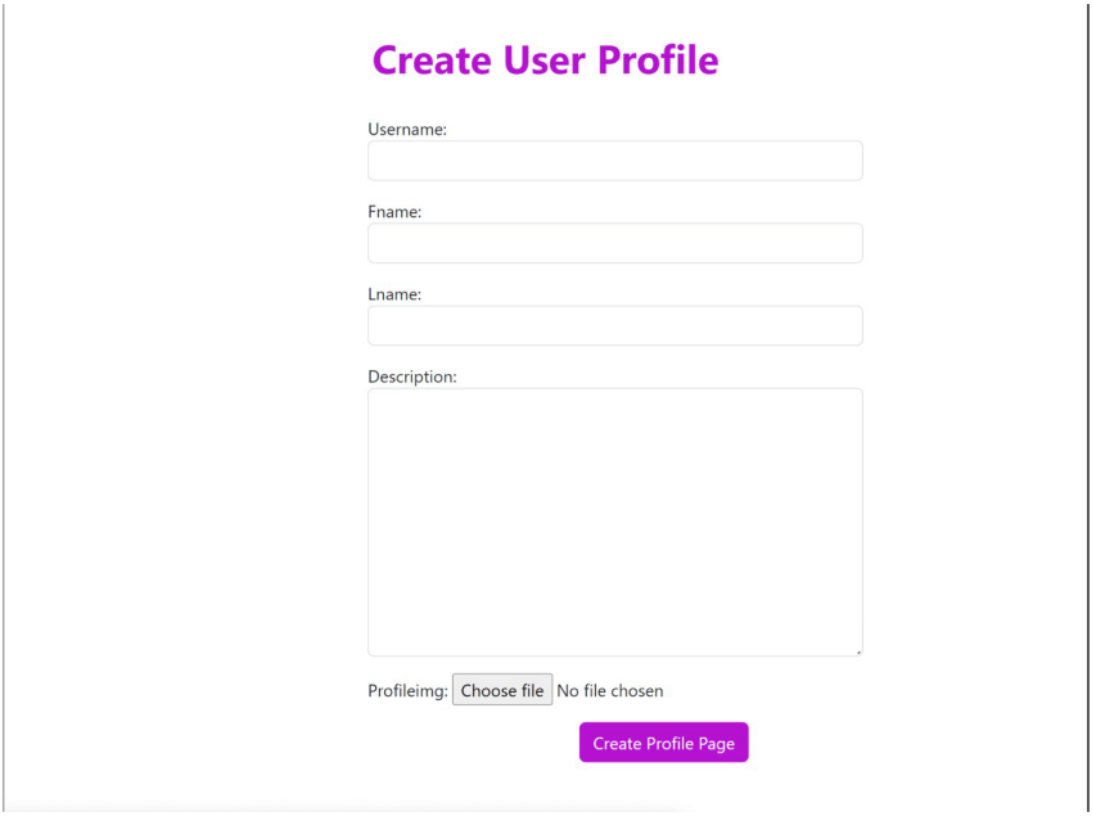


Fig 9: User profile page featuring personalized content and settings for individual users.

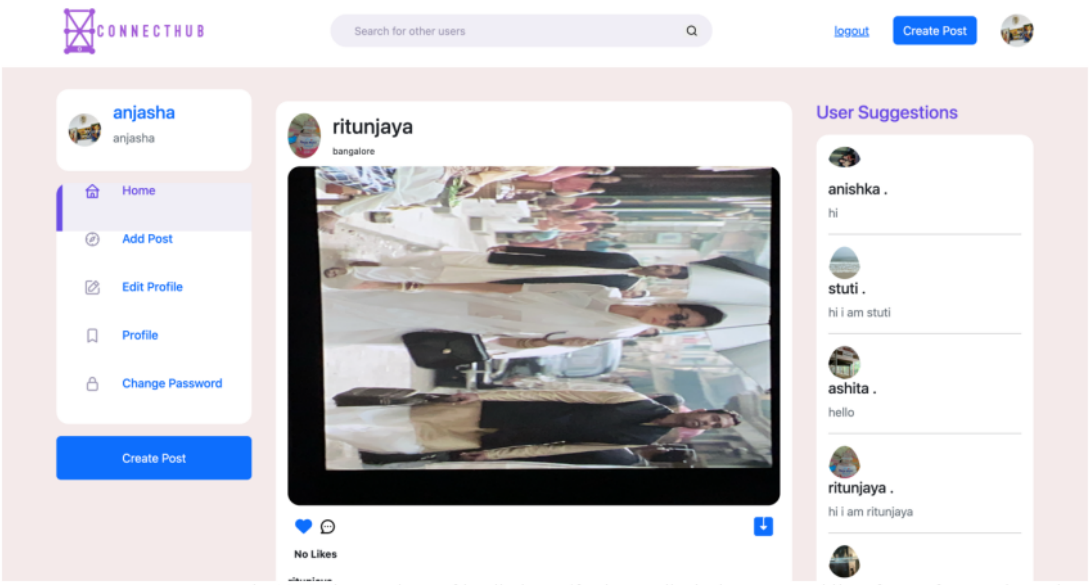


Fig 10: A clean and user-friendly home/feed page displaying a curated list of posts for easy browsing.

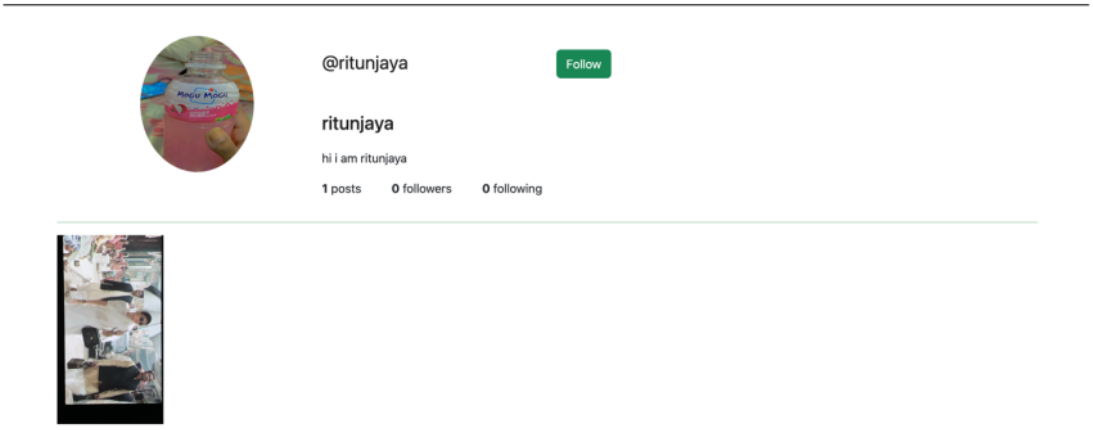


Fig 11: Depicts the user interface of the 'Other's Profile' page, showcasing personalized information and interaction options.

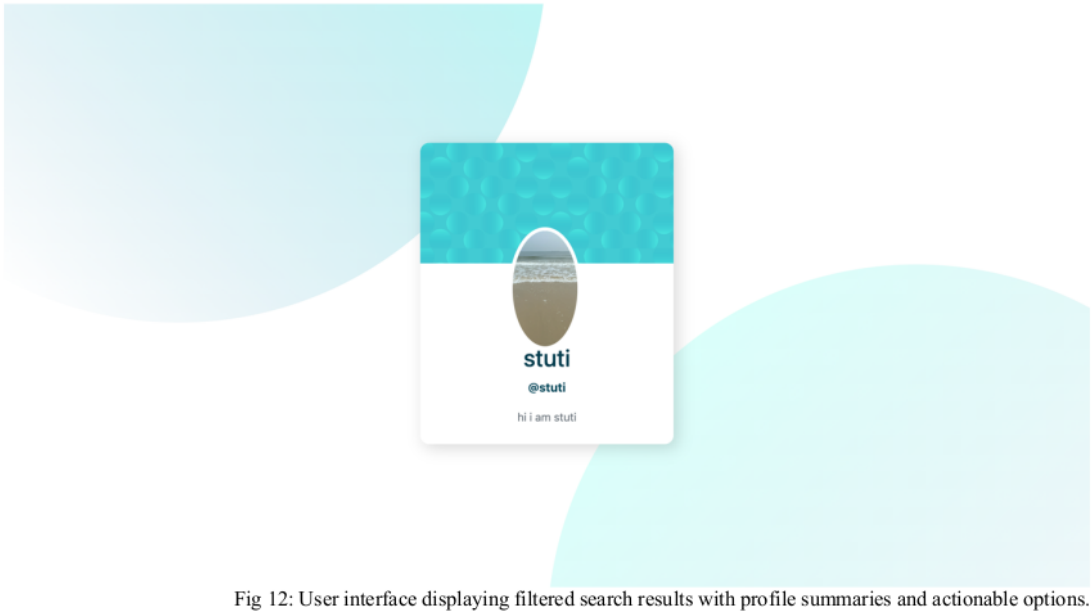


Fig 12: User interface displaying filtered search results with profile summaries and actionable options.

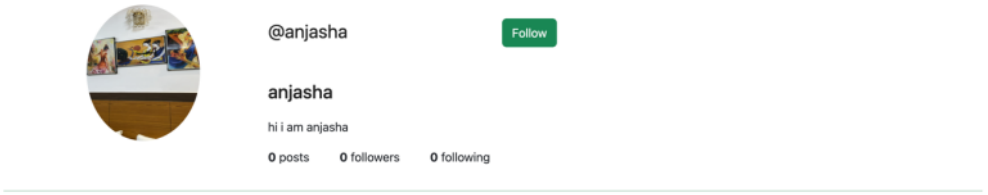


Fig 13: User's personal profile page showcasing customized settings and activity summary.

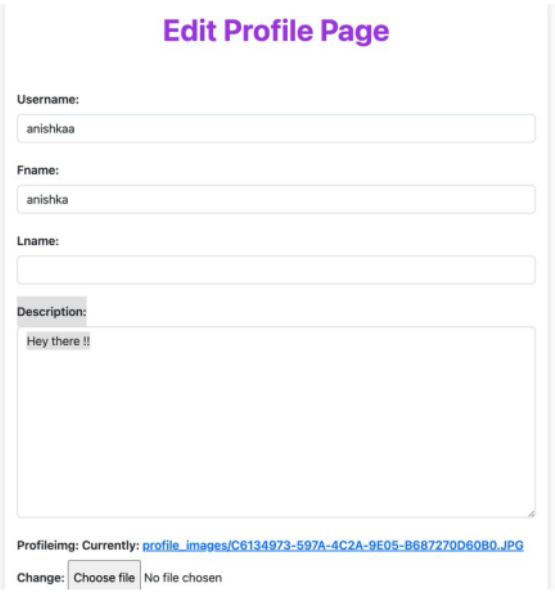


Fig 14: Edit Profile Page - Customize your user details and preferences here.

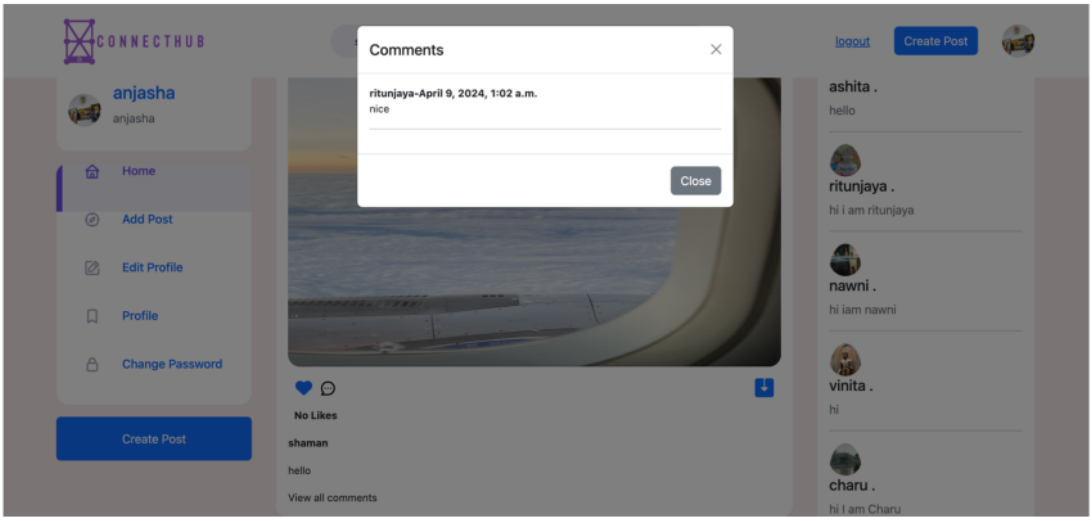


Fig 15: Illustrates the layout of the Comments on Posts Page, highlighting where users can view and interact with feedback on blog posts.

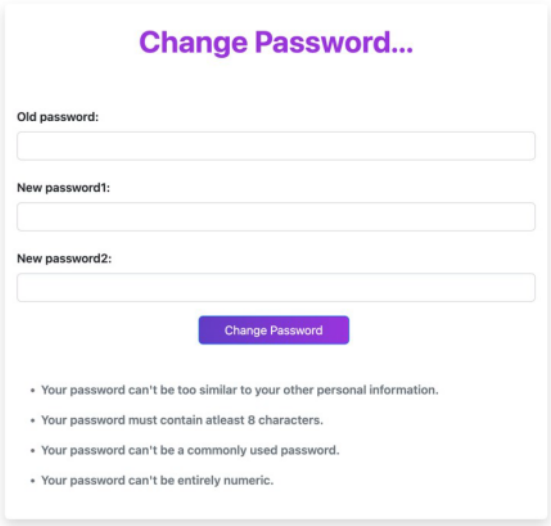


Fig 16: Change Password Page - Allows users to update their account passwords securely.

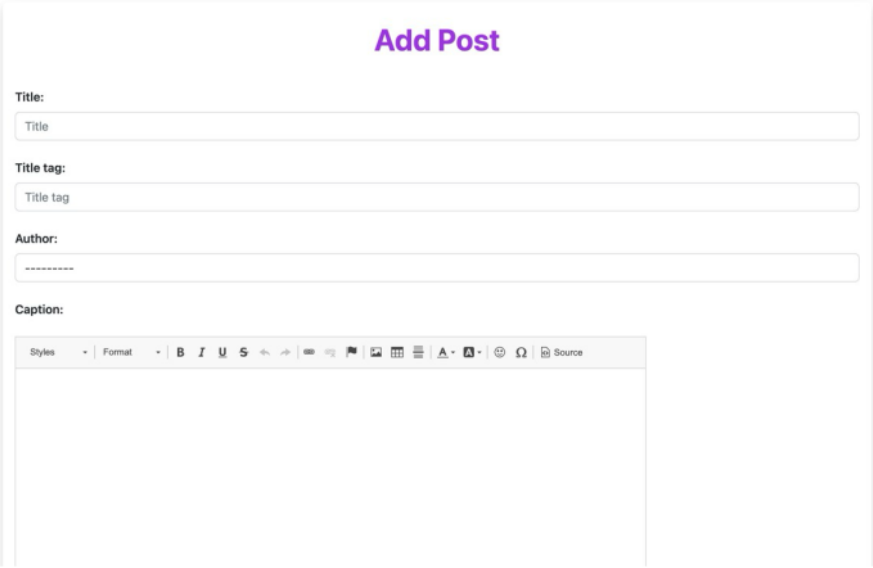


Fig 17: Illustrates the user interface for adding or creating a new post, featuring form fields for title, content, and multimedia attachments.

```
from django.contrib import admin
from .models import *

# Register your models here.
admin.site.register(Profile)
admin.site.register(Post)
admin.site.register(FollowersCount)
admin.site.register(Comment)
admin.site.register(LikePost)
```

Fig 18: Admin.py configures admin panel settings for Django models to enhance content management.

```python
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
from .views import AddPostView,EditProfilePageView,CreateProfilePageView,PasswordsChangeView,FriendView,AddCommentView,DeletePostVie
from . import views
from django.contrib.auth import views as auth_views

urlpatterns = [
    path('', views.home,name='home'),
    path('login/', views.login,name='login'),
    path('logout/', views.logout,name='logout'),
    path('signup/', views.signup,name='signup'),
    path('<int:pk>/edit_profile_page/',EditProfilePageView.as_view(),name='edit_profile_page'),
    path('<int:pk>/profile/',ShowProfilePageView.as_view(),name='show_profile_page'),
    path('create_profile_page/',CreateProfilePageView.as_view(),name='create_profile_page'),
    path('add_post/',AddPostView.as_view(),name="add_post"),
    path('password/',PasswordsChangeView.as_view(template_name='base/change-password.html')),
    path('password_success/', views.password_success, name="password_success"),
    path('friends/',FriendView.as_view(),name='friends'),
    path('post/<int:pk>/comment/',AddCommentView.as_view(),name='add_comment'),
    path('like-post',views.like_post,name='like-post'),
    path('post/<int:pk>/remove',DeletePostView.as_view(),name="delete_post"),
    path('post/edit/<int:pk>',U                    name="update_post"),
    path('search',views.search,
    path('follow',views.follow,name='follow'),
]+ static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)



if settings.DEBUG:
    urlpatterns+=static(settings.STATIC_URL,document_root=settings.STATIC_ROOT)
```

(parameter) name: str

Fig 19: Url.py illustrates the URL routing configuration for a Django web application, mapping URLs to their respective view functions.

```python
# Create your views here.
@login_required(login_url='signup')
def home(request):
    user_object = User.objects.get(username=request.user.username)
    user_profile = Profile.objects.get(user=user_object)
    all_users = User.objects.all()
    all_posts=Post.objects.all()
    all_profile=Profile.objects.all()
    count_posts=len(all_posts)

    my_user=[user_profile]
    suggestion_users=[]

    for user in all_profile:
        if user not in my_user:
            suggestion_users.append(user)

    random.shuffle(suggestion_users)

    context={
        'user_object':user_object,
        'user_profile':user_profile,
        'all_users':all_users,
        'all_posts':all_posts,
        'all_profile':all_profile,
        'count_posts':count_posts,
        'suggestion_users':suggestion_users,
    }
    return render(request,"base/home.html",context)
```

Fig 20: Views.py - Defines the logic for handling HTTP requests and returning appropriate responses in a Django web application.

```python
class Profile(models.Model):
    # user = models.ForeignKey(User, on_delete=models.CASCADE)
    user=models.OneToOneField(User,null=True,on_delete=models.CASCADE)
    description = models.TextField(blank=True)
    fname = models.TextField(blank=True)
    lname = models.TextField(blank=True)
    username=models.TextField(blank=True)
    profileimg = models.ImageField(upload_to='profile_images', default='blank-profile-picture.png')

    def __str__(self):
        return(str(self.user))

    def get_absolute_url(self):
        return reverse('home')
```

Fig 21: Model.py -Defines user attributes extending the base User model, such as bio, profile picture, and contact information.

```python
class FollowersCount(models.Model):
    follower = models.CharField(max_length=100)
    user = models.CharField(max_length=100)

    def __str__(self):
        return self.user
```

Fig 22: Model.py - Represents the count of followers a user has in a database model.

```python
class Post(models.Model):
    title=models.CharField(max_length=255)
    image=models.ImageField(null=True,blank=True,upload_to="images/")
    title_tag=models.CharField(max_length=255,default="")
    author=models.ForeignKey(Profile,on_delete=models.CASCADE)
    caption=RichTextField(blank=True,null=True)
    post_date=models.DateField(auto_now_add=True)
    location=models.CharField(max_length=255,default="")
    no_of_likes=models.IntegerField(default=0)

    def __str__(self):
        return self.title + " | " + str(self.author)

    def get_absolute_url(self):
        return reverse('home')

    def get_owner_pp(self):
        return self.author.profileimg.url

    def profileid(self):
        return self.author.user.id
```

Fig 23: Model.py - Defines the structure for representing blog posts, including fields for title, content, author, and publication date.

```python
class Comment(models.Model):
    post=models.ForeignKey(Post,related_name="comments",on_delete=models.CASCADE)
    name=models.CharField(max_length=255)
    body=models.TextField()
    date_added=models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return '%s - %s' % (self.post.title,self.name)

    def get_absolute_url(self):
        return reverse('home')
```

Fig 24: The comment model defines the structure for storing user comments, including fields for the comment text, author.

```python
class LikePost(models.Model):
    post_id=models.CharField(max_length=500)
    username=models.CharField(max_length=100)

    def __str__(self):
        return self.username
```

Fig 25: Model.py - Representing a data model for storing information about likes on posts in a social media application.

```
{% extends 'main.html' %}

<title>
    Comment on Blog Post
</title>

{% block content %}


    <div class="container shadow p-3 mb-5 bg-body rounded"  style="width: 65%;margin-top: 30px;">
        <h1 style="text-align:center; margin-top:10px; color:#1e3c72;font-weight: bolder">Add Comment</h1>
    <br/>
    <div class="form-group" style="font-weight: 600;line-height: 2.5;">
        <form method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <br>
            <button class="btn btn-primary btn-block" style="width:200px;position: relative;left:40%;
            background: linear-gradient(to right,#373B44 ,#4286f4 );">Add Comment</button>
        </form>
    </div>
    </div>


{% endblock %}
```

Fig 26: AddComment.html contains the HTML code for a user interface that enables users to add comments to a webpage or application.

```
{% extends 'main.html' %}
{% load static %}
{% block styles %}
<link rel="stylesheet" href="{% static 'styles/post.css' %}" />
{% endblock %}

<title>
    Add a new  Post
</title>

{% block content %}
{% if user.is_authenticated %}

    <div class="container shadow p-3 mb-5 bg-body rounded con" style="width: 80%;margin-top: 30px;">
        <h1 style="text-align:center; margin-top:10px;color:#9a35dd;font-weight: bolder">Add Post</h1>
        <br>
    <div class="form-group" style="font-weight: 600;line-height: 2.5;">
        <form method="POST" enctype="multipart/form-data">
            {% csrf_token %}
            {{ form.media }}
            {{ form.as_p }}
            <br>
            <button class="btn btn-primary btn-block" style="width:200px;position: relative;left:40%;
            background: linear-gradient(to right,#623dc6 ,#9a35dd );">Post</button>
        </form>
    </div>
    </div>
    <br/><br/>
{% endif %}
{% endblock %}
```

Fig 27: Addpost.html - HTML file for adding new posts to a web application.

```
{% extends 'main.html' %}
<title>Change Password</title>
{% block content %}


<div class="container shadow p-3 mb-5 bg-body rounded " style="width: 50%;margin-top: 30px;">
    <h1  style="text-align:center;color:#9a35dd;margin-top: 10px;font-weight: bolder;">Change Password...</h1>
<br/><br/>
<div class="form-group" style="font-weight: 600;line-height: 2.5;">
    <form method="POST">
        {% csrf_token %}
        {{ form.as_p}}

        <button class="btn btn-primary btn-block" style="width:200px;position: relative;left:35%;
        background: linear-gradient(to right,#623dc6 ,#9a35dd );">Change Password</button>
        <br/><br/>
        <ul>
            <li class="text-muted">Your password can't be too similar to your other personal information.</li>
            <li class="text-muted">Your password must contain atleast 8 characters.</li>
            <li class="text-muted">Your password can't be a commonly used password.</li>
            <li class="text-muted">Your password can't be entirely numeric.</li>
        </ul>
    </form>
</div>
</div>

{% endblock %}
```

Fig 28: Changepassword.html - A webpage for users to update their passwords.

```
{% extends 'main.html' %}

<title>Create User Profile</title>

{% block content %}
{% if user.is_authenticated %}

<div class="container">
        <h1 style="text-align: center; font-weight: bold; margin-top: 30px; color: #b512d2;">Create User Profile</h1>
        <br/>
    <div class="form-group">
        <form method="POST" enctype="multipart/form-data" style="width: 50%; position: relative;left: 300px;">
            {% csrf_token %}
            {{ form.as_p }}
            <button class="btn" style="background-color: #b512d2;color: #fff;position: relative;left: 200px;">Create Profile Page</button>
        </form>
        <br/>
        <br/>
    </div>
    </div>

{% else %}
<div class="card container shadow p-3 mb-5 bg-body rounded" style="width: 65%;margin-top: 150px;">
    <h3 class="delete-title"  style="text-align:center;color:#1e3c72; margin-top: 10px;font-weight: bolder"> <span style="color:#b512d2;">L
    <div class="container"  >
    <div class="form-group" style="text-align: center;">
        <br/>
        <a href="{% url 'signup' %}" style="text-decoration: none; color: #fff;">
            <button class="btn" style="width:200px;position: relative;left: 1%; background-color: #b512d2; color: #fff;">Sign Up</button>
        </a>
    </div>
    </div>
</div>
{% endif %}
{% endblock %}
```

Fig 29: Createuserprofile.html - A webpage for users to input and submit their personal information to create a new profile.

```
{% extends 'main.html' %}

<title>Edit Profile Page</title>

{% block content %}
{% if user.is_authenticated %}
<!-- {%  if user.id == profile.user.id  %} -->

    <div class="container shadow p-3 mb-5 bg-body rounded" style="width: 50%;margin-top: 30px;">
        <h1 style="text-align:center; margin-top:10px; color:#9a35dd;font-weight: bolder;">Edit Profile Page</h1>
    <div class="form-group" style="font-weight: 600;line-height: 2.5;">


    <br/>
        <form method="POST" enctype="multipart/form-data">
            {% csrf_token %}
            {{ form.as_p }}
            <br>
            <button class="btn btn-primary btn-block" style="width:200px;position: relative;left:35%;
            background: linear-gradient(to right,#623dc6 ,#9a35dd );">Update Profile Page</button>
        </form>
    </div>
    </div>
    <br/><br/>

    <!-- Javascript -->
    <script>
        var name="{{ user.first_name }}"
        document.getElementById("elder").placeholder=name;
    </script>

{% else %}
    Login required to view
{% endif %}

<!-- {% else %}
You are not the correct user to edit the page
{% endif %} -->
{% endblock %}
```

Fig 30: Editprofilepage.html - Allows users to customize their profile information.

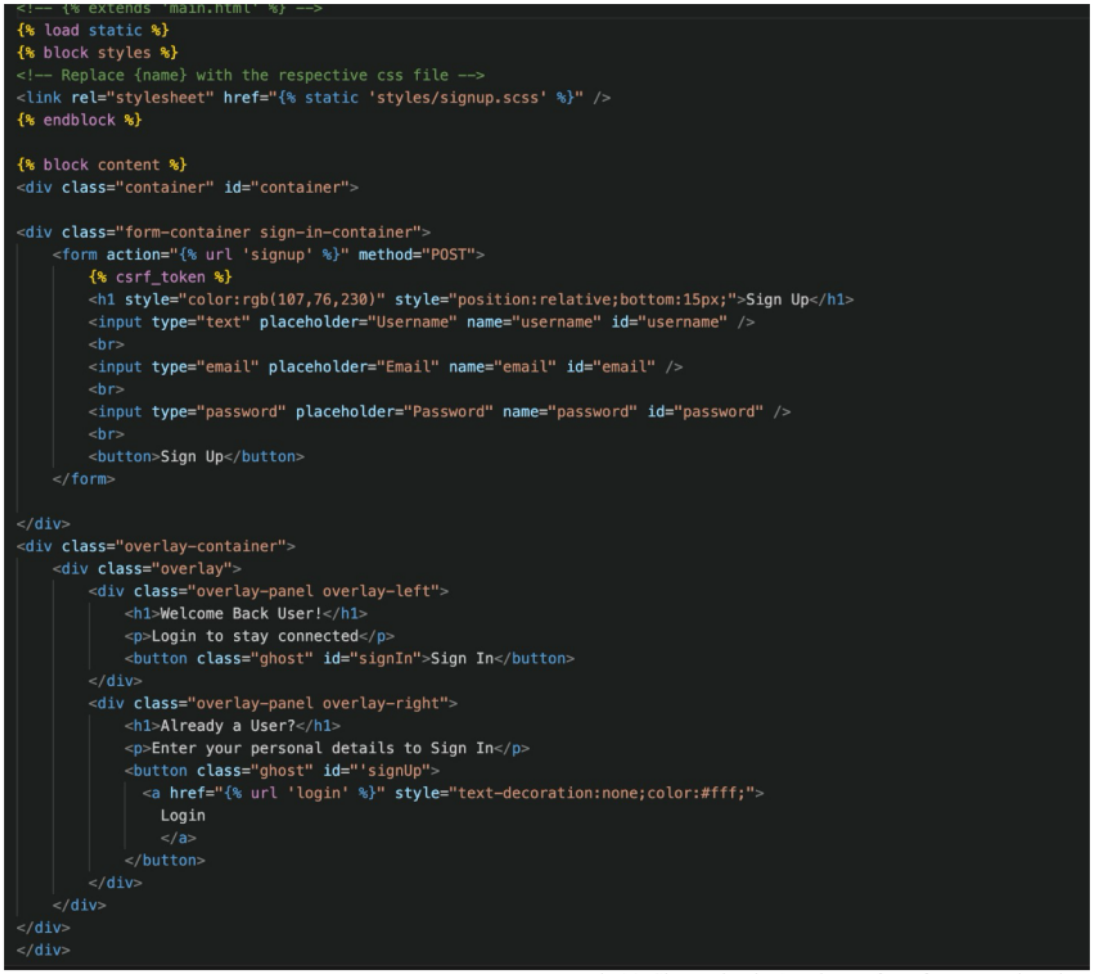
```
<!-- {% extends 'main.html' %} -->
{% load static %}
{% block styles %}
<!-- Replace {name} with the respective css file -->
<link rel="stylesheet" href="{% static 'styles/signup.scss' %}" />
{% endblock %}

{% block content %}
<div class="container">
<div class="form-container sign-in-container">
    <form action="{% url 'login' %}" method="POST">
        {% csrf_token %}
        <h1 style="color:rgb(107,76,230)" style="position:relative;bottom:15px;">Sign in</h1>
        <input type="text" placeholder="Username" name="username" />
        <br>
        <input type="password" placeholder="Password" name="password" />
        <br/>
        <button>Sign In</button>
    </form>
</div>

<div class="overlay-container">
    <div class="overlay">
        <div class="overlay-panel overlay-right">
            <h1>Not a User?</h1>
            <p>Enter your personal details to Sign Up</p>
            <button class="ghost" id="'signUp">
                <a href="{% url 'signup' %}" style="text-decoration:none;color:#fff;">
                Sign Up
                </a>
            </button>
        </div>
    </div>
</div>
</div>
</div>

{% endblock %}
```

Fig 31: Login.html - The user interface for accessing a secure system or platform.



```
<!-- {% extends 'main.html' %} -->
{% load static %}
{% block styles %}
<!-- Replace {name} with the respective css file -->
<link rel="stylesheet" href="{% static 'styles/signup.scss' %}" />
{% endblock %}

{% block content %}
<div class="container" id="container">

<div class="form-container sign-in-container">
    <form action="{% url 'signup' %}" method="POST">
        {% csrf_token %}
        <h1 style="color:rgb(107,76,230)" style="position:relative;bottom:15px;">Sign Up</h1>
        <input type="text" placeholder="Username" name="username" id="username" />
        <br>
        <input type="email" placeholder="Email" name="email" id="email" />
        <br>
        <input type="password" placeholder="Password" name="password" id="password" />
        <br>
        <button>Sign Up</button>
    </form>

</div>
<div class="overlay-container">
    <div class="overlay">
        <div class="overlay-panel overlay-left">
            <h1>Welcome Back User!</h1>
            <p>Login to stay connected</p>
            <button class="ghost" id="signIn">Sign In</button>
        </div>
        <div class="overlay-panel overlay-right">
            <h1>Already a User?</h1>
            <p>Enter your personal details to Sign In</p>
            <button class="ghost" id="'signUp">
              <a href="{% url 'login' %}" style="text-decoration:none;color:#fff;">
                Login
                </a>
            </button>
        </div>
    </div>
</div>
</div>
```

Fig 32: SignUp.html - A webpage form for user registration.

```
<!doctype html>
{% load static %}
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Font-awesome -->
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.2.0/css/all.min.css" integrity="sha512-xh6O/CkQoPOWDdY

   <!-- Iconify -->
   <script src="https://code.iconify.design/2/2.2.1/iconify.min.js"></script>

   <!--- ICONSCOUT CDN -->
   <link rel="stylesheet" href="https://unicons.iconscout.com/release/v2.1.6/css/unicons.css">

    <!-- Styles -->
    {% block styles %} {% endblock %}

    <link rel="stylesheet" href="{% static 'styles/style.css' %}" />
     <!-- Title -->
     {% block title %}
     <title>Social Media Web App</title>
     {% endblock %}

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.1/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-iYQeCzEYFbKjA/T2uD

  </head>
  <body>

    {% block content %}
    {% endblock %}
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.1/dist/js/bootstrap.bundle.min.js" integrity="sha384-u1OknCvxWvY5kfmNBILK2hRnQC3Pr
  </body>
</html>
```

Fig 33: Main.html - Represents the main HTML file within the project.

```python
#!/usr/bin/env python
"""Django's command-line utility for administrative tasks."""
import os
import sys


def main():
    """Run administrative tasks."""
    os.environ.setdefault("DJANGO_SETTINGS_MODULE", "socials.settings")
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    execute_from_command_line(sys.argv)


if __name__ == "__main__":
    main()
```

Fig 34: Manage.py is a command-line utility that lets you interact with the Django project.

ConnectHub_application.pdf

ORIGINALITY REPORT

4%	4%	1%	%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	buildmedia.readthedocs.org Internet Source	1%
2	affiliatepal.net Internet Source	1%
3	www.irjmets.com Internet Source	1%
4	clouddevs.com Internet Source	<1%
5	ebisys.blogspot.com Internet Source	<1%
6	www.jetbrains.com Internet Source	<1%
7	www.coursehero.com Internet Source	<1%
8	docs.tid.al Internet Source	<1%
9	github.com Internet Source	<1%

Exclude quotes	On	Exclude matches	< 3 words
Exclude bibliography	On		