Web Applications and Services

Oline Payment System

2024 April

Contents

Introduction

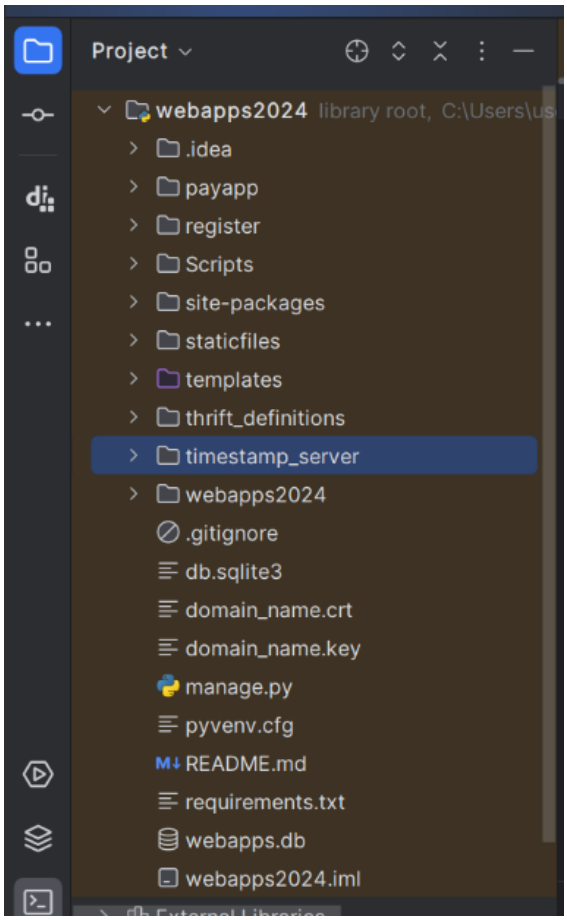
This Django project is for designing an abstracted and simplified model of PayPal. A user is able to send money to other registered users, request money from other registered users, and manage their account through the web interface, such as viewing recent transactions. We will go through introducing the presentation layer, business logic layer, data access layer, security layer, web services, RPC, and Cloud implementation. Each step will be detailly described of how it's been done.

The key functions in the system:

- **Virtual Account Management:** Allow users the ability to manage money in a virtual environment, emulating the facilities of the real world's payment service provider, PayPal.
- **Initial Funds and Currency Selection:** When new users sign up, they should be credited with a given initial amount (e.g., £1000) and enabled to select the currency in which they want to make their transactions during registration.
- **User Registration:** Requires details of username, which is the unique key in the database, email, and password for user registration.

- **Direct Payments:** Allow users to send money directly to others within the system, ensuring proper validation of the recipient and availability of money.
- **Payment Requests:** Let users request payments from other users, and the particular user could receive such a payment request and accept or decline it.
- **Transaction history:** This allows access to the detailed transaction logs, keeping track of sent and received payments, and payment requests.
- **Currency Conversion:** Create a RESTful web service that allows one to convert from one currency to another using hard-coded rates.
- **Security Measures:** Strong security must be implemented to ensure the safety of user data and transactions. Use HTTPS, CSRF, SQL Injection, and XSS security.

Project structure (screenshot1):



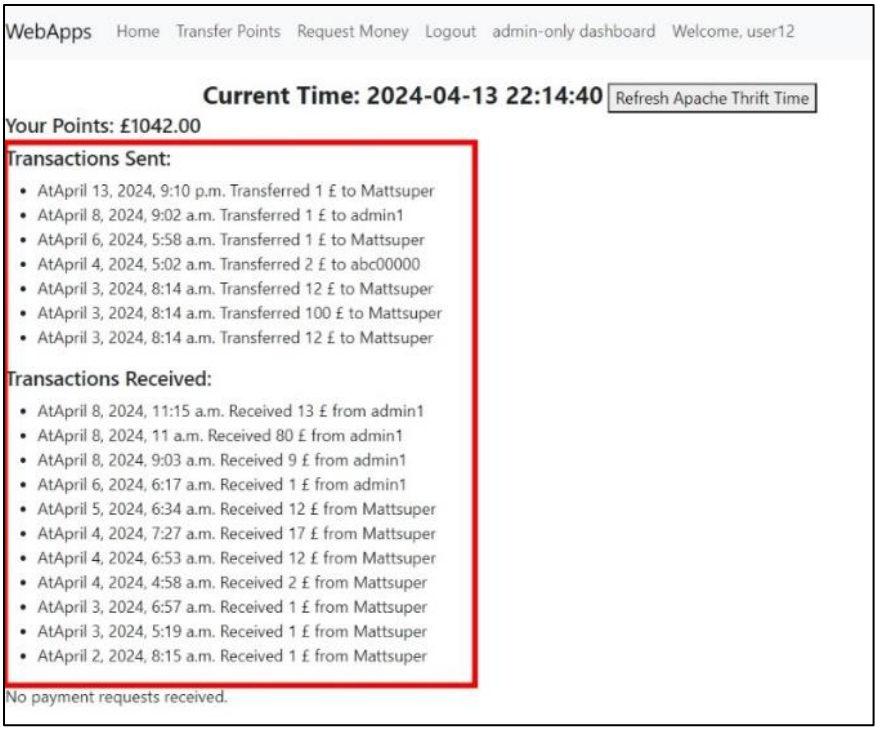
(screenshot1)

Presentation Layer

The presentation layer consists of a set of templates through which users and administrators interact with the web application.

User's view

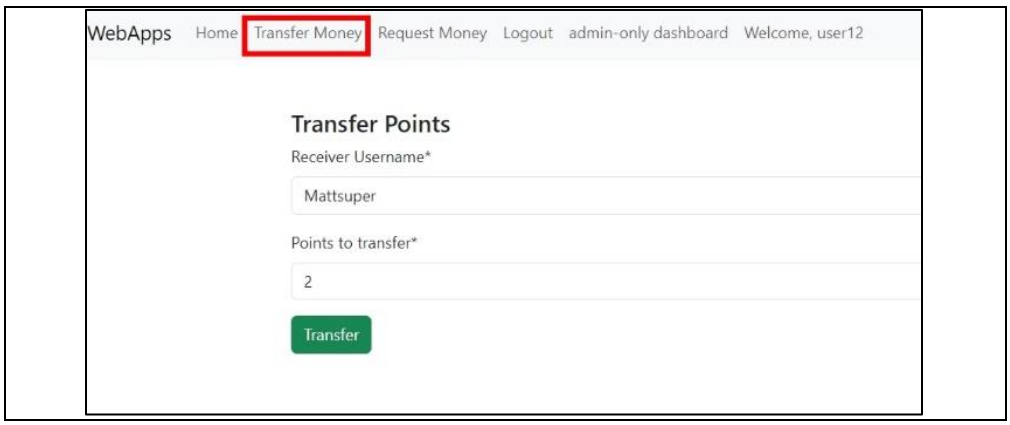
1. Users can see all their transactions (screenshot2)

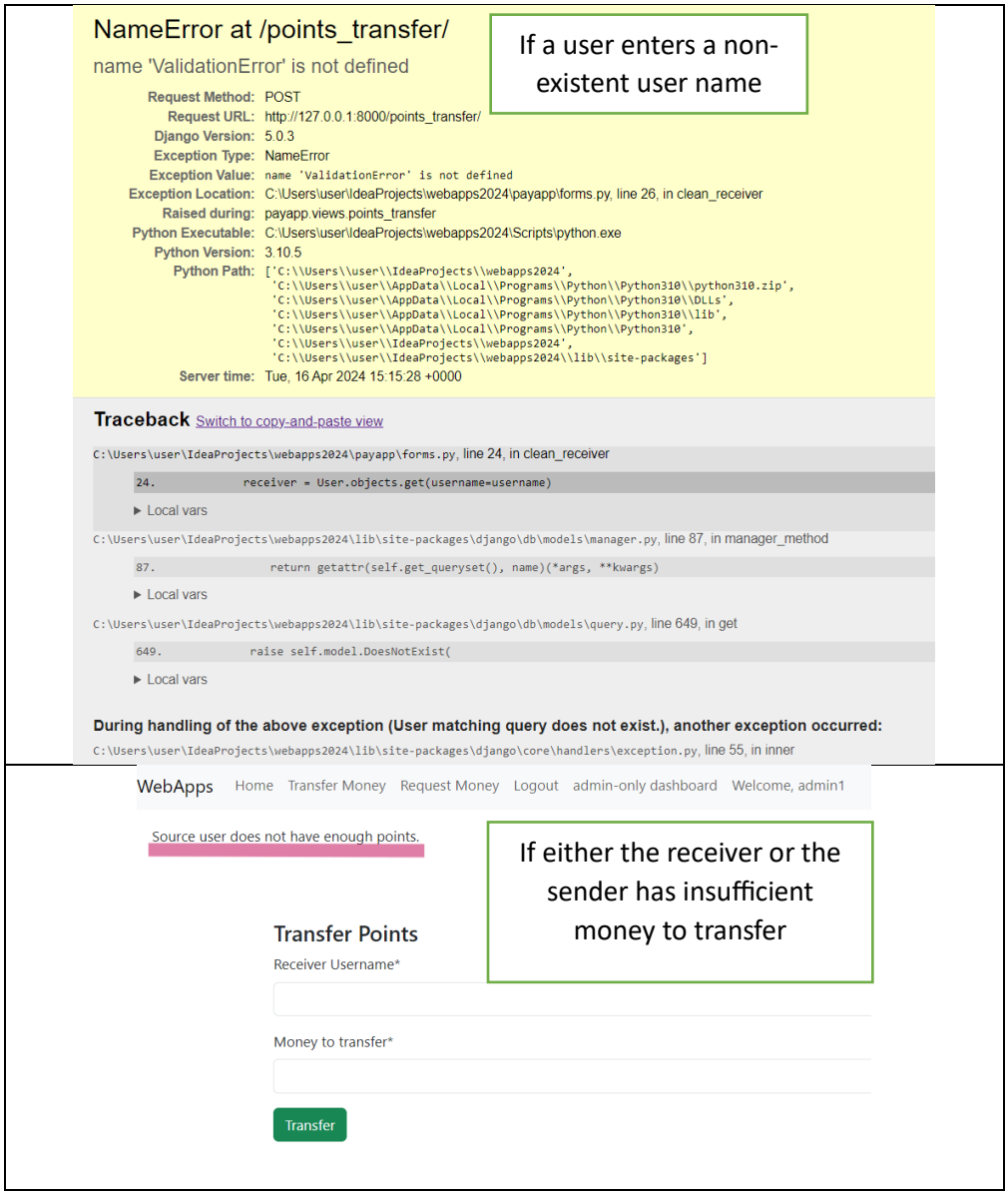


(screenshot2)

2. Users can make direct payments to other registered users (screenshot 3).

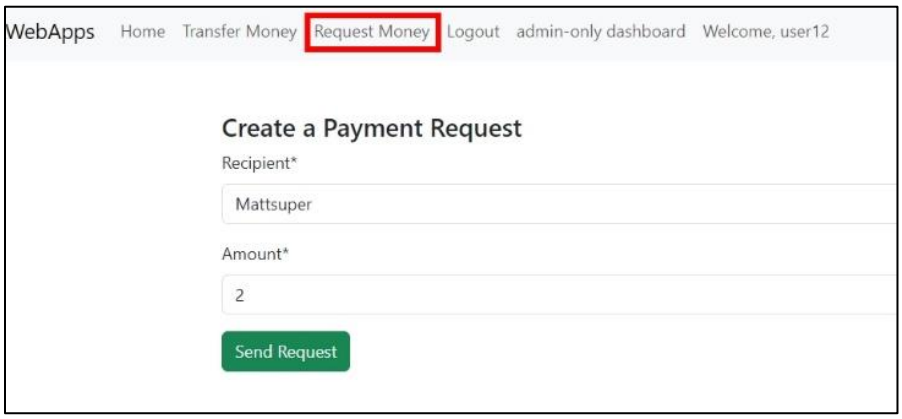
If a user tries to:
2.1 Transfer money to himself
2.2 Request money from himself
2.3 Enter the wrong user name
2.4 Transfer or request money beyond the existing amount
the website will run error messages to prevent misconduct. (screenshoot3)





(screenshot3)

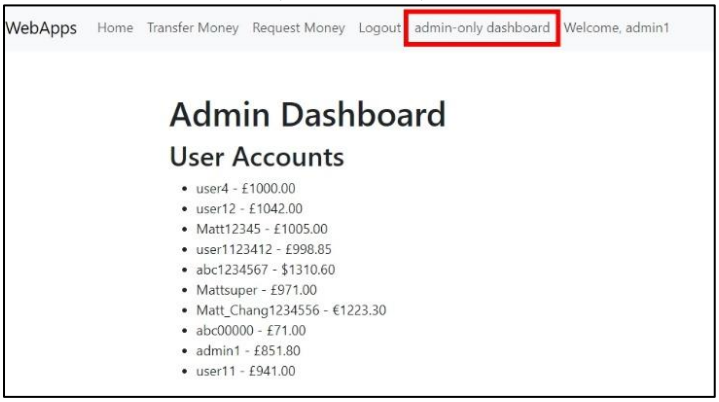
3. Users can request payments from registered users (screenshot 4).
The same error will be shown in screenshot 3 if users try to request payments from themselves or vice versa.



(screenshot4)

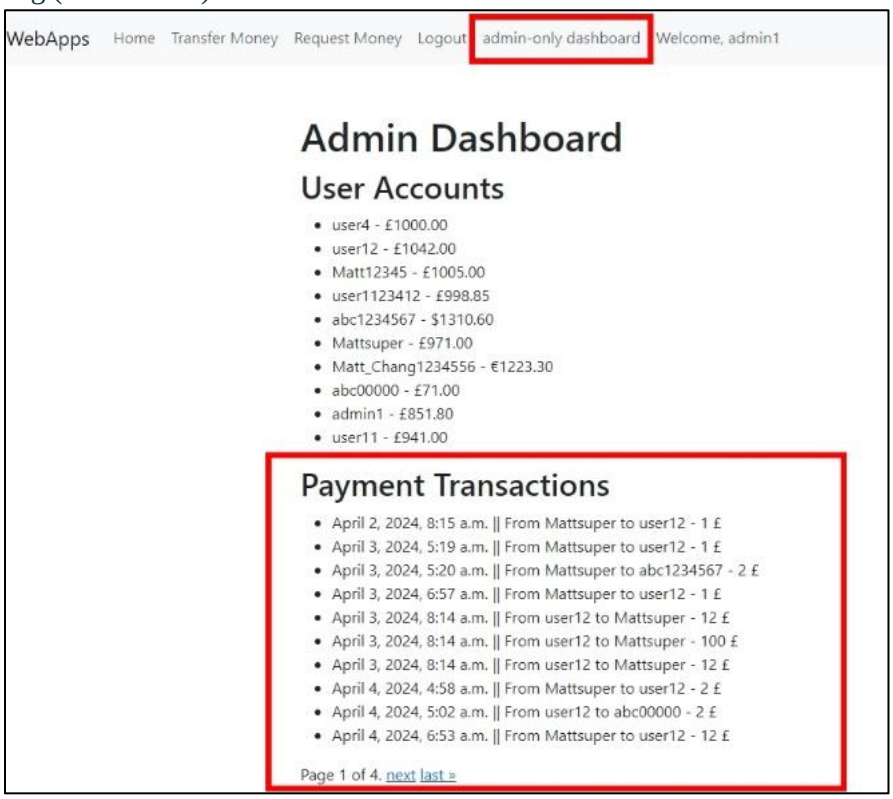
Administrators' view

4. Only administrators can see all user accounts (screenshot5)



(screenshot5)

5. Administrators can see all payment transactions. Transaction records will be organized on different pages if they are too long.(screenshot6)



(screenshot6)

Business Logic Layer

The business logic layer consists of views containing the logic that accesses the model(s) and defers to the appropriate template(s). Views support transactions so that data integrity is preserved.

1.2 User's view

All the functions for the user's view to achieve the following three points will be discussed below:

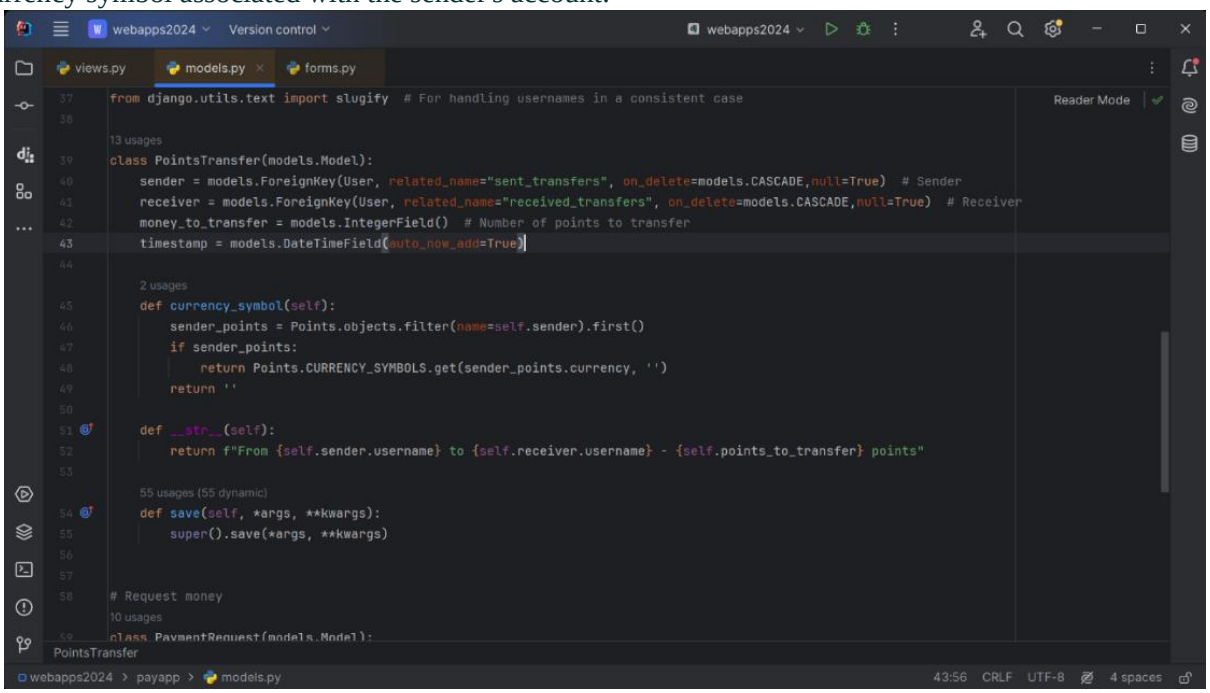
4.1.1 View all their transactions

4.1.2 Make direct payments to other registered users

4.1.3 Request payments from registered users

Model:

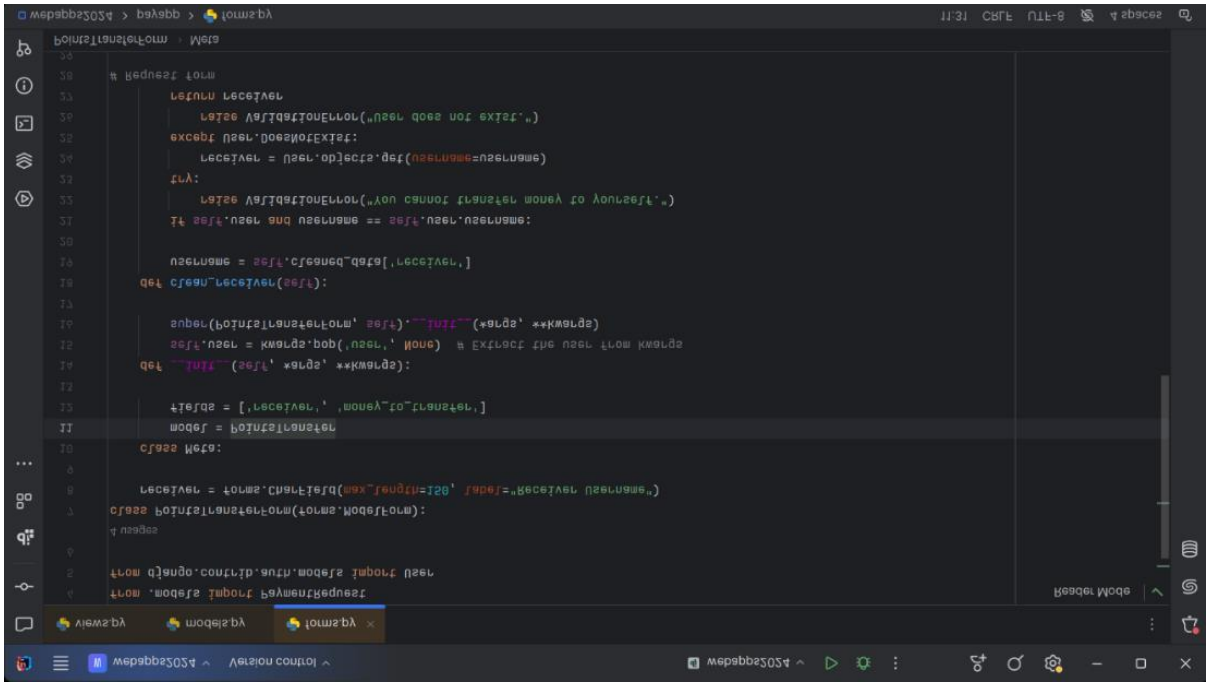
Here we create a model called the PointsTransfer model for the database. There are four fields, receiver, money_to_transfer shown as the amount of money that's been transferred, and timestamp to record the time of the transaction (screenshot7). The currency_symbol method in the PointsTransfer model serves to retrieve and return the currency symbol associated with the sender's account.



(screenshot7)

Form:

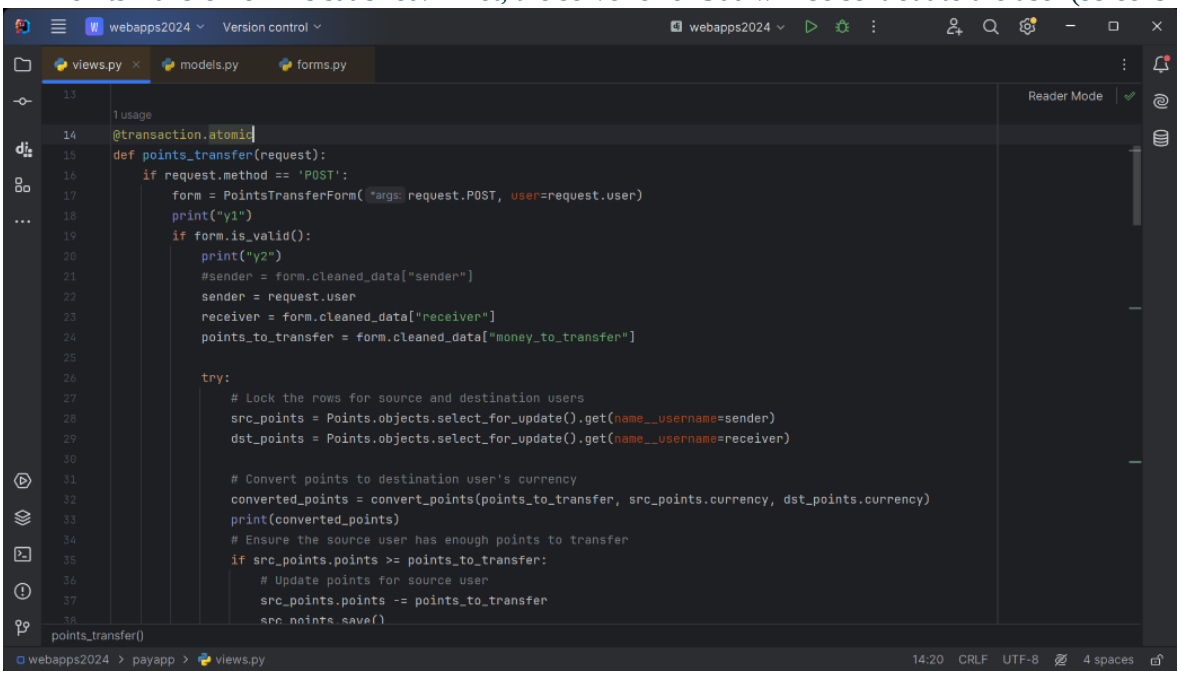
PointstransferForm is created using Django form to save the data from the users' input (screenshot 8). There are two fields needed to be filled in for the form – receiver and the amount of money to transfer. The checking function is also included in the class. The function checks for two anomalies: 1. If the sender and the receiver are the same (You can't transfer money to yourself.) 2. If the user exists in the database.



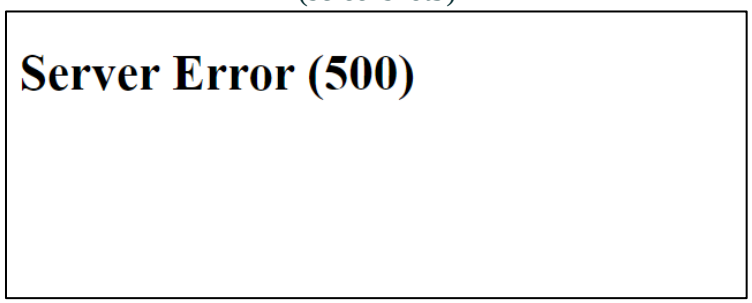
(screenshot8)

View:

Once the user interacts with the form shown on the HTML page, the following code will create the form from the PointsTransferForm if the request method is from POST (screenshot9). Then, it checks if the clean_receiver function in PointsTransferForm is satisfied. If not, the server error 500 will be sent out to the user (screenshot 10).



(screenshot9)



(screenshot10)

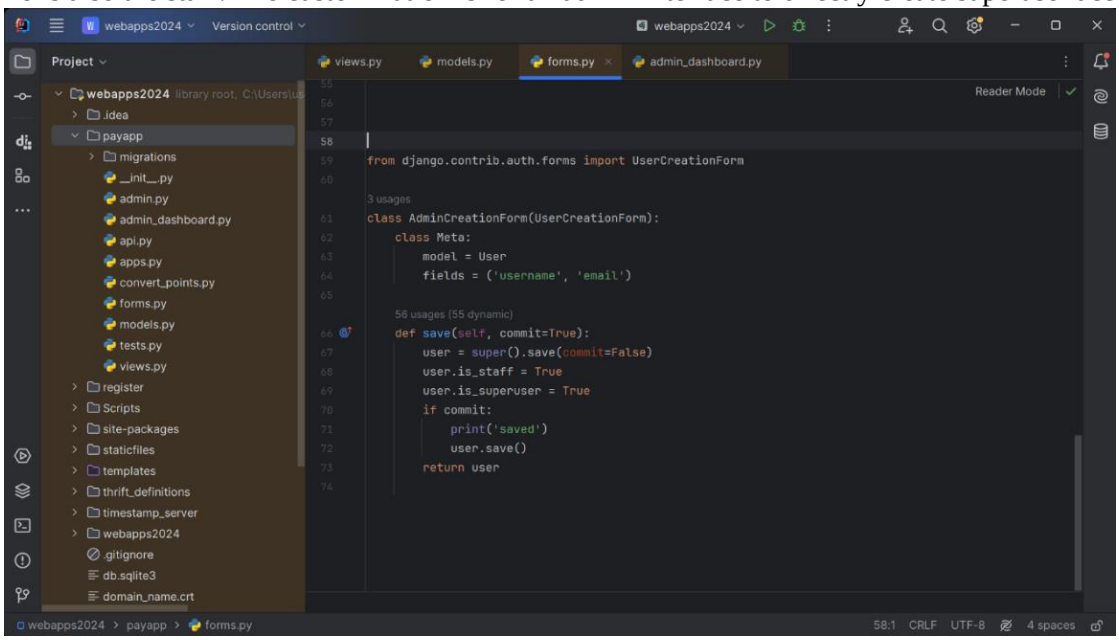
4.2 Administrators' View

The administrator view will be illustrated based on the three criteria below:

- 4.2.1 View all user accounts and balances
- 4.2.2 View all payment transactions
- 4.2.3 Register more administrators

Form:

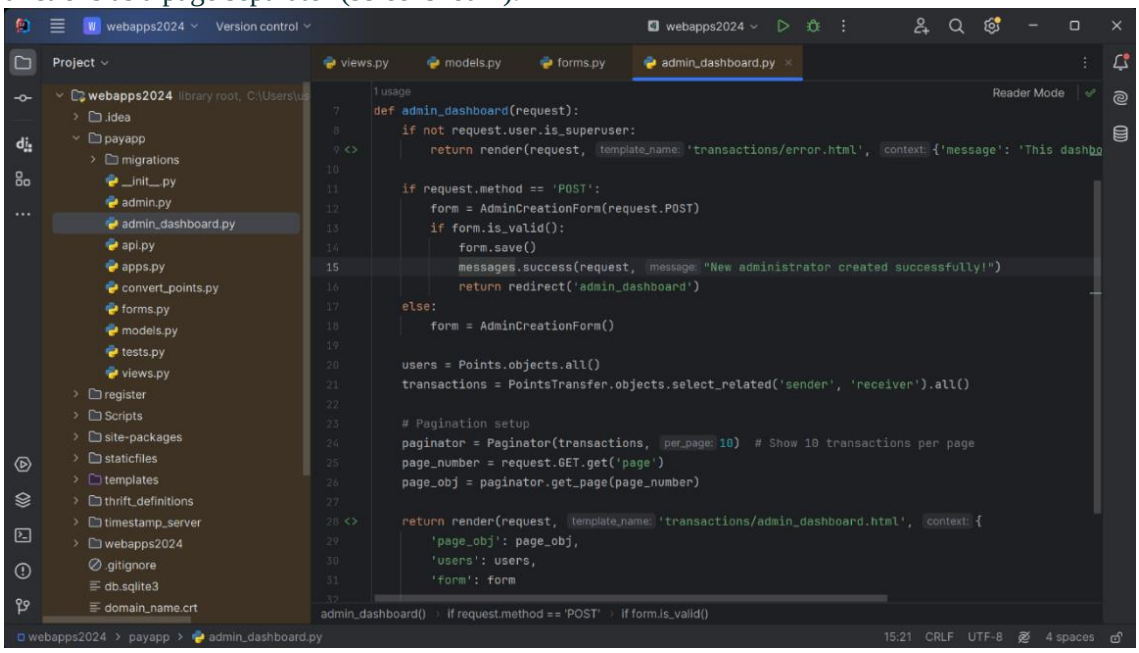
AdminCreationForm is created using Django UserCreationForm (screenshot 11). The form extends UserCreationForm with two fields - username and email. The save method in the form modifies the behavior of the default save method. It sets the is_staff and is_superuser attributes of the user to True, effectively creating a new superuser who is also the staff. The customization is for an admin interface to directly create superuser accounts.



(screenshot11)

admin_dashboard.py:

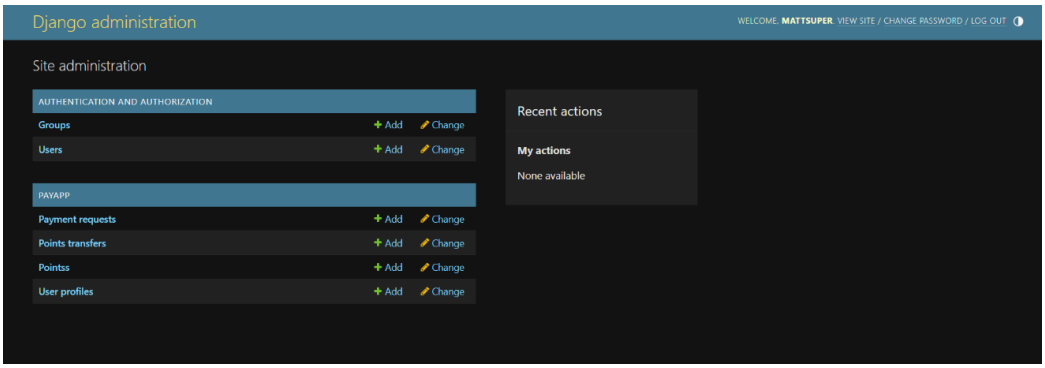
It first checks if the user is a super user in the database. Then create a form from the AdminCreationForm if the request is from POST. Form.is_valid() checks if the form's validation. Save the information for the new administrator into the database. Next, the function retrieves all the users' data and stores them using the django.core.paginator Paginator functions as a page separator (screenshot12).



(screenshot12)

Data Access Layer

The data access layer consists of models containing everything about the data. To simplify deployment and configuration, I use SQLite as my Relational DataBase Management System (RDBMS). SQLite is an RDBMS that is included in Python (screenshot13). It can be used as a Python database-access API for accessing the objects.



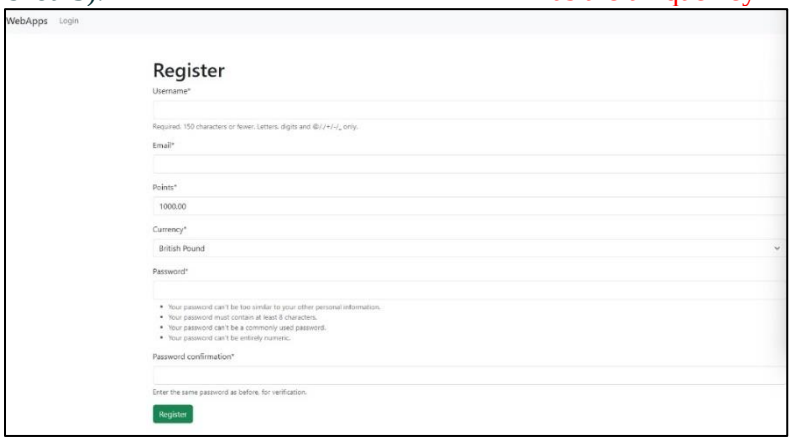
(screenshot13)

Security Layer

This online payment service is a multi-user web application. A user can log in to interact with the system. Users are not able to see other users' information nor access pages and functionality for administrators. Administrators access their own set of pages, through which can have access to all users' information. Users and administrators can log out from the web application.

4.3 Authentication functionality

6.1.1 registration(screenshot15). Users can't have the same user name as the unique key in the database.



(screenshot15)

6.1.2 login (screenshot16)

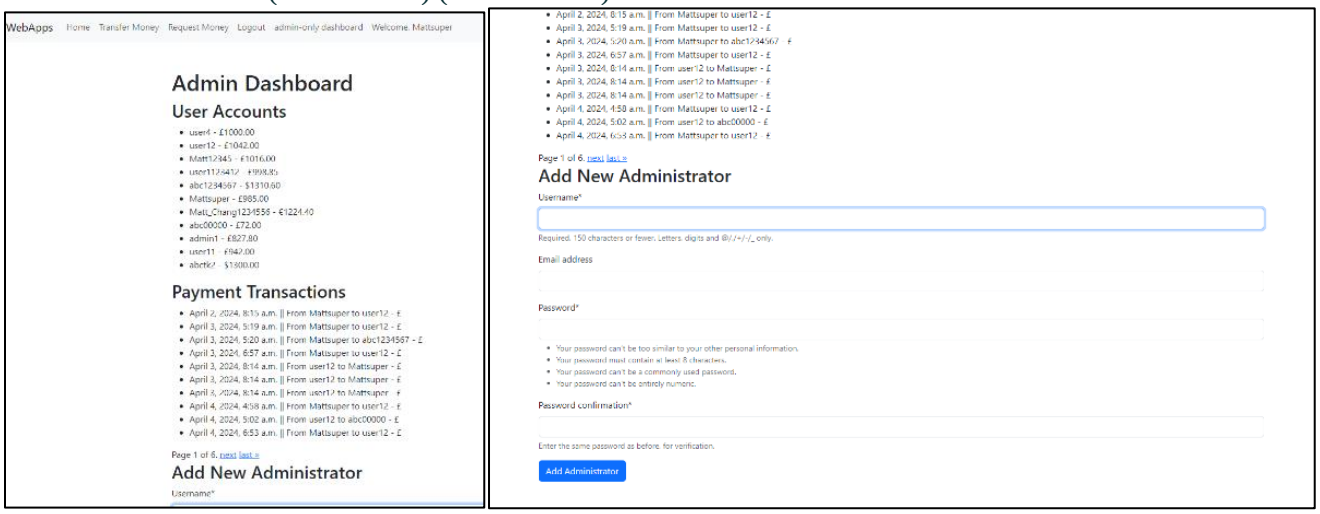
(screenshot16)

6.1.3 logout (screenshot17)

(screenshot17)

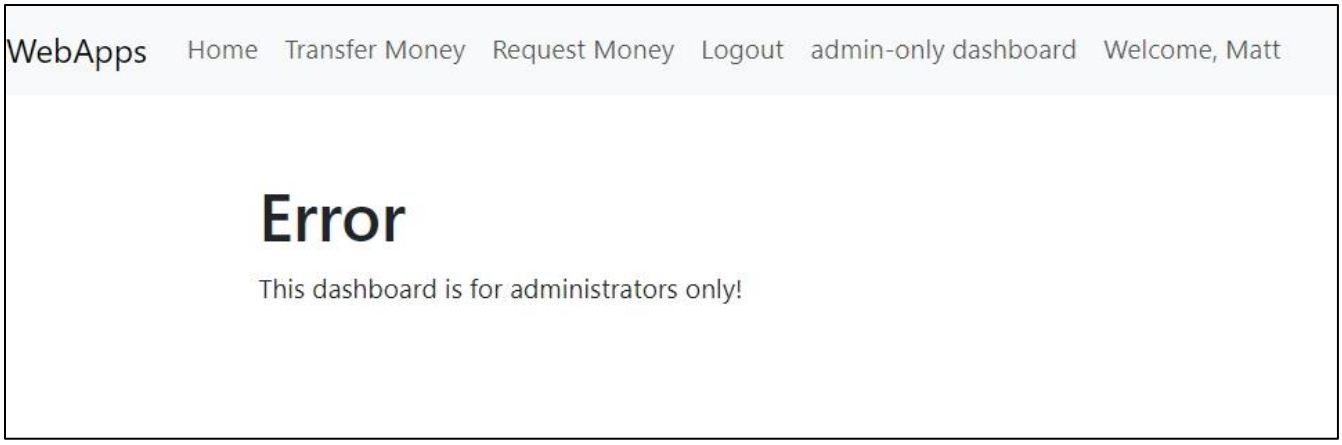
4.4 Access control to restrict access to web pages to non-authorised users

6.2.1Authorized users (Administers) (screenshot18):



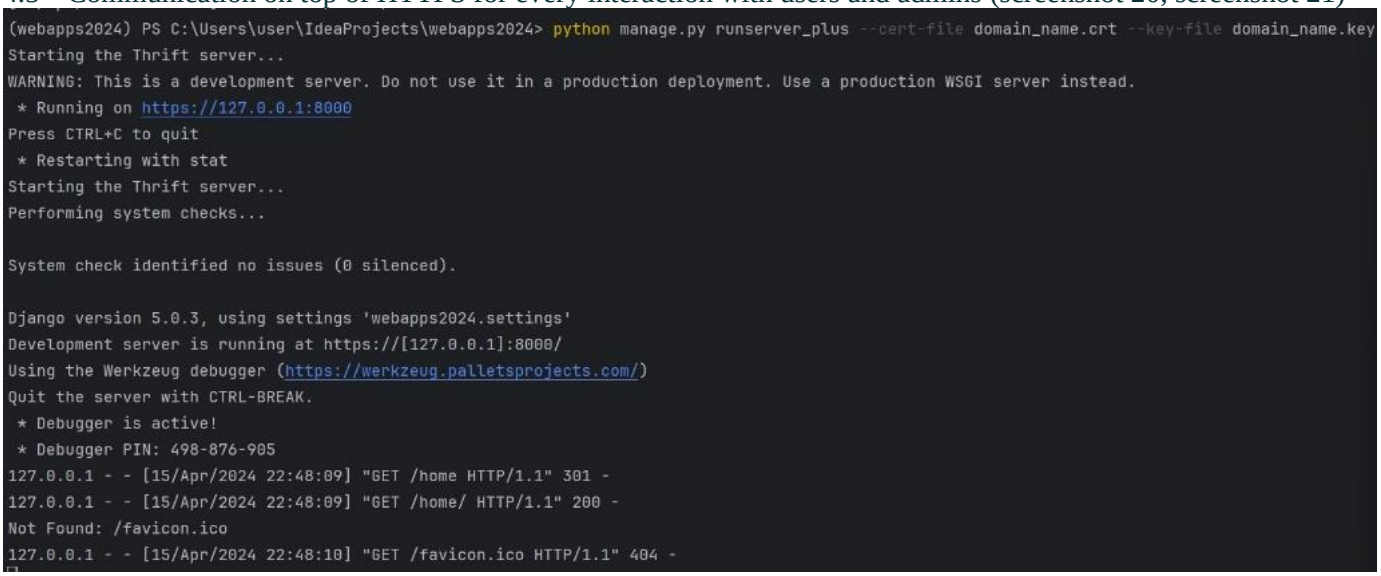
(screenshot18)

6.2.2 Unauthorized users (screenshot19):



(screenshot19)

4.5 Communication on top of HTTPS for every interaction with users and admins (screenshot 20, screenshot 21)

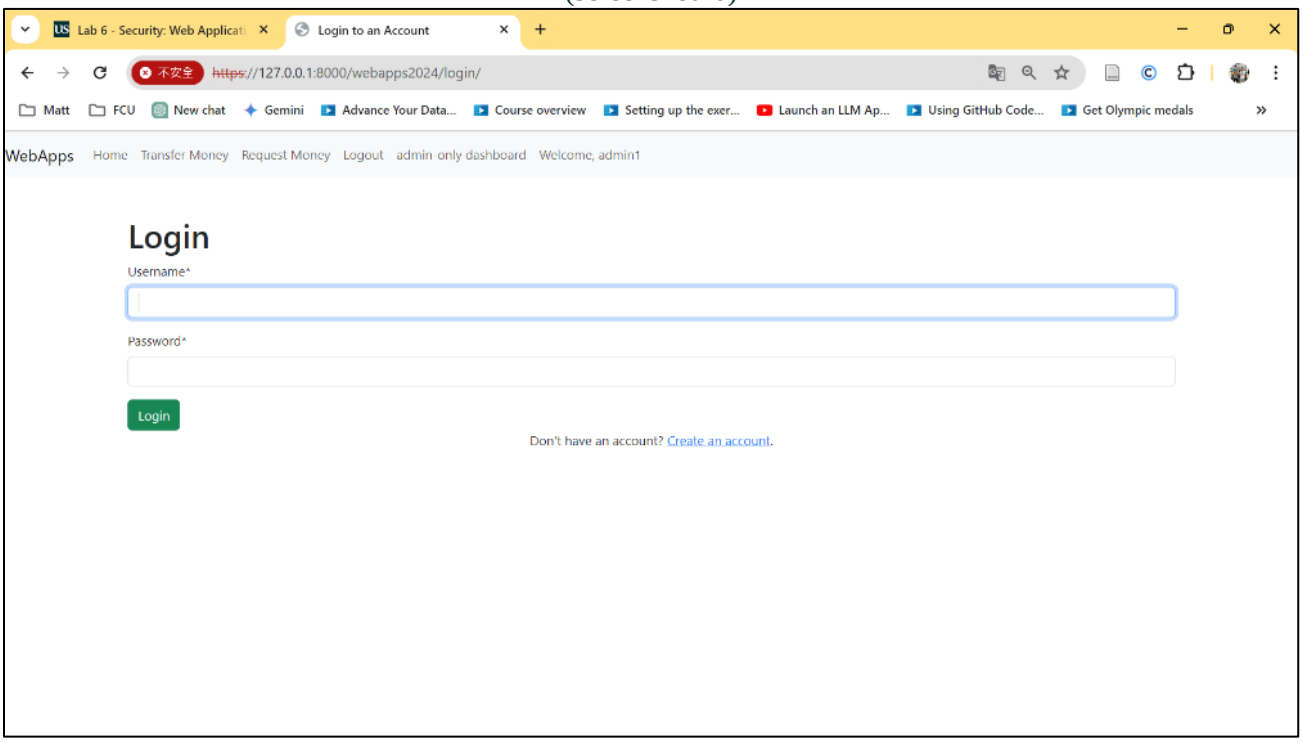


```
(webapps2024) PS C:\Users\user\IdeaProjects\webapps2024> python manage.py runserver_plus --cert-file domain_name.crt --key-file domain_name.key
Starting the Thrift server...
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on https://127.0.0.1:8000
Press CTRL+C to quit
 * Restarting with stat
Starting the Thrift server...
Performing system checks...

System check identified no issues (0 silenced).

Django version 5.0.3, using settings 'webapps2024.settings'
Development server is running at https://[127.0.0.1]:8000/
Using the Werkzeug debugger (https://werkzeug.palletsprojects.com/)
Quit the server with CTRL-BREAK.
 * Debugger is active!
 * Debugger PIN: 498-876-905
127.0.0.1 - - [15/Apr/2024 22:48:09] "GET /home HTTP/1.1" 301 -
127.0.0.1 - - [15/Apr/2024 22:48:09] "GET /home/ HTTP/1.1" 200 -
Not Found: /favicon.ico
127.0.0.1 - - [15/Apr/2024 22:48:10] "GET /favicon.ico HTTP/1.1" 404 -
```

(screenshot20)



(screenshot21)

Cross-site scripting (XSS), Cross-site request forgery (CSRF), SQL injection, and Clickjacking protection are implemented to enforce the security layer. (screenshot 22).

Cross-site scripting (XSS)

```
{% extends 'webapps2024/base.html' %}

{% block content %}
    <h1>Points Transfer Summary</h1>
    {% if src_points and dst_points %}
        <p>Transfer successful!</p>
        <div>
            <h2>Source User:</h2>
            <p>Username: {{ src_points.name.username }}</p>
            <p>Remaining Points: {{ src_points.points }}</p>
        </div>
        <div>
            <h2>Destination User:</h2>
            <p>Username: {{ dst_points.name.username }}</p>
            <p>New Points Total: {{ dst_points.points }}</p>
        </div>
    {% else %}
        <p>No points transfer information available.</p>
    {% endif %}
    <a href="{% url 'points_transfer' %}">Transfer More Point
{% endblock %}
```

Cross-site request forgery (CSRF)

```
1 usage
@csrf_protect
def logout_user(request):
    logout(request)
    messages.success(request, message: 'You are now logg out.')
    return redirect('login')
```

Cross-site request forgery (CSRF)

```
1 usage
@login_required
def home(request):
    user_points = None
    transactions_sent = transactions_received = []  # Initialize as empty lists

    # No need for try-except block for Points.DoesNotExist due to get_or_create usage
    user_points, created = Points.objects.get_or_create(name=request.user)
    # Adjusting queries to use ForeignKey relationships
    transactions_sent = PointsTransfer.objects.filter(sender=request.user).order_by('-timestamp')
    transactions_received = PointsTransfer.objects.filter(receiver=request.user).order_by('-timestamp')
    # Fetch pending payment requests received by the current user
    payment_requests_received = PaymentRequest.objects.filter(recipient=request.user, status='pending').order_by('-create
    # Call get_timestamp to get the current time from the Thrift service
    current_time = get_timestamp()
    context = {
        'user_points': user_points.points if user_points else 0,  # Ensure there's a fallback if user_points is None
        'transactions_sent': transactions_sent,
        'transactions_received': transactions_received,
        'formatted_points': user_points.formatted_points() if user_points else "0",
        'payment_requests_received': payment_requests_received,  # Add pending payment requests to the context
```

Cross-site request forgery (CSRF)

```
1 usage
@csrf_protect
def login_user(request):
    if request.method == 'POST':
        form = AuthenticationForm(request, data=request.POST)
        if form.is_valid():
            username = form.cleaned_data.get('username')
            password = form.cleaned_data.get('password')
            user = authenticate(username=username, password=password)
            if user is not None:
                login(request, user)
                messages.success(request, message: 'You are now logged in.')
                return redirect('home')
            else:
                messages.error(request, message: 'invalid username or password.')
        else:
            messages.error(request, message: 'login failed. Please correct the errors below.')
    else:
        form = AuthenticationForm()
    return render(request, template_name: 'login/login.html', context: {'form': form})
```

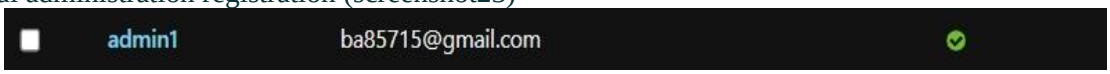
Clickjacking protection

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',

]
X_FRAME_OPTIONS = 'DENY'
```

(screenshot22)

4.6 Initial administration registration (screenshot23)



(screenshot23)

Web Services

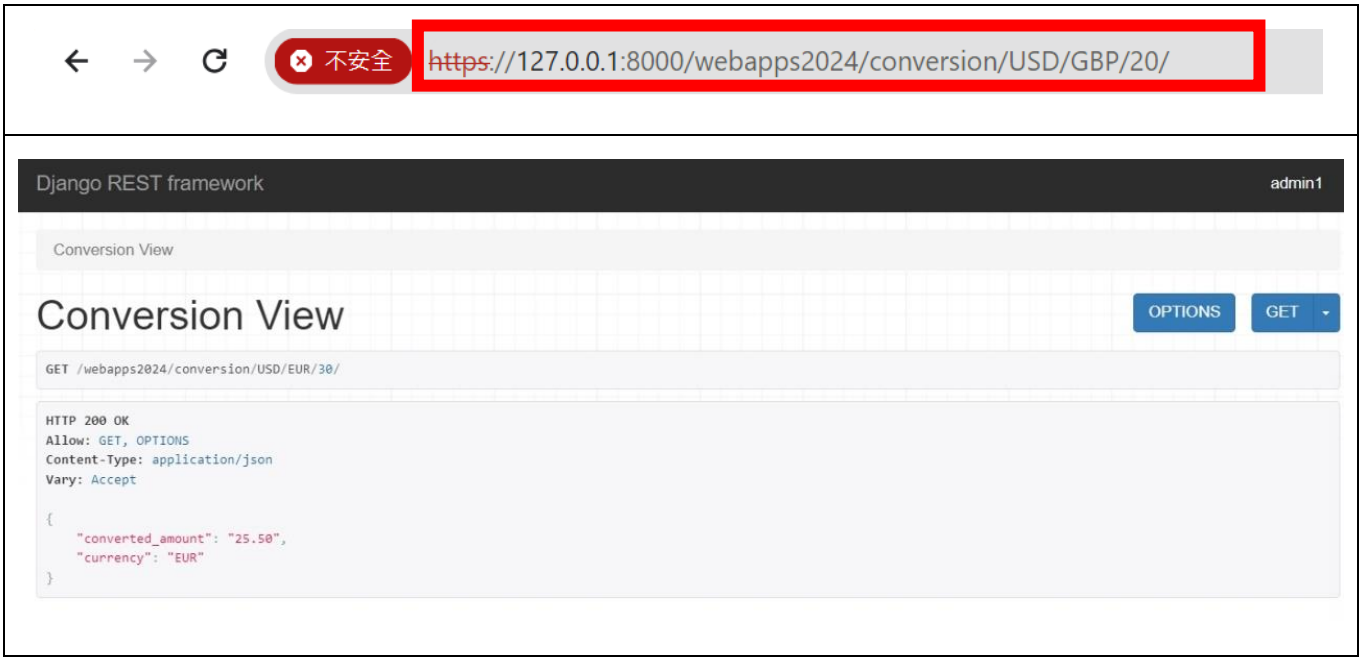
REST Service is implemented to be accessed by the business logic layer. The service will be deployed on the same server and accessed from the business logic layer in the standard way (screenshot 24).

A currency conversion RESTful web service that responds only to GET requests.

The exported resource is named *conversion* in a path such as the following

baseURL/conversion/{currency1}/{currency2}/{amount_of_currency1} (screenshot24)

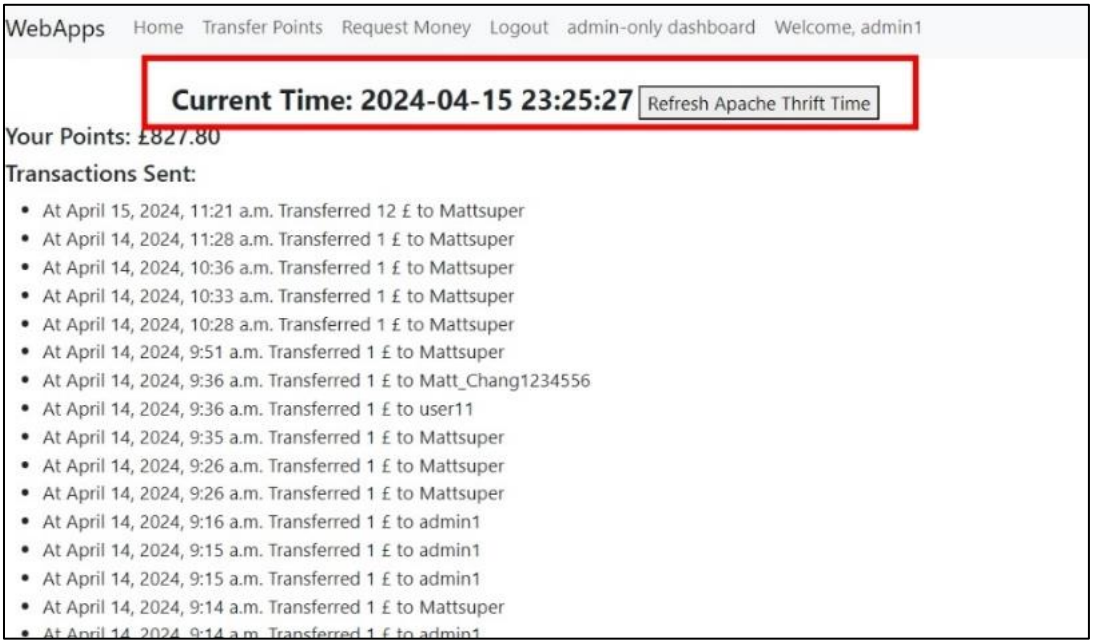
The RESTful web service will return an HTTP response with the conversion rate (currency1 to currency2).



(screenshot24)

RPC

All transactions are timestamped by accessing a 'remote' Thrift timestamp service. When requested, the service returns the current date and time to the system. The Thrift server is implemented as a Django application, which uses a separate thread to accept time-stamping requests at port 10000.



Initiate the Thrift server

```
1 usage
class TimestampServer:
    2 usages (2 dynamic)
    def getCurrentTimestamp(self):
        now = datetime.datetime.now()
        return now.strftime("%Y-%m-%d %H:%M:%S")

handler = TimestampServer()
processor = TimestampService.Processor(handler)
transport = TSocket.TServerSocket(host='127.0.0.1', port=10000)
tfactory = TTransport.TBufferedTransportFactory()
pfactory = TBinaryProtocol.TBinaryProtocolFactory()

server = TServer.TSimpleServer( *args: processor, transport, tfactory, pfactory)

3 usages
def start_server():
    print('Starting the Thrift server...')
    server.serve()
    print('Done.')

if __name__ == '__main__':
    server_thread = threading.Thread(target=start_server)
    server_thread.start()
```

get getCurrent Timestamp from the Thrift server

```python
def get_timestamp():
    try:
        transport = TSocket.TSocket( host: '127.0.0.1',  port: 10000)
        transport = TTransport.TBufferedTransport(transport)
        protocol = TBinaryProtocol.TBinaryProtocol(transport)
        client = TimestampService.Client(protocol)

        transport.open()
        timestamp = client.getCurrentTimestamp()
        transport.close()

        return timestamp
    except Thrift.TException as tx:
        print(f'Thrift exception: {tx.message}')
        return None
```

Call get_timestamp()

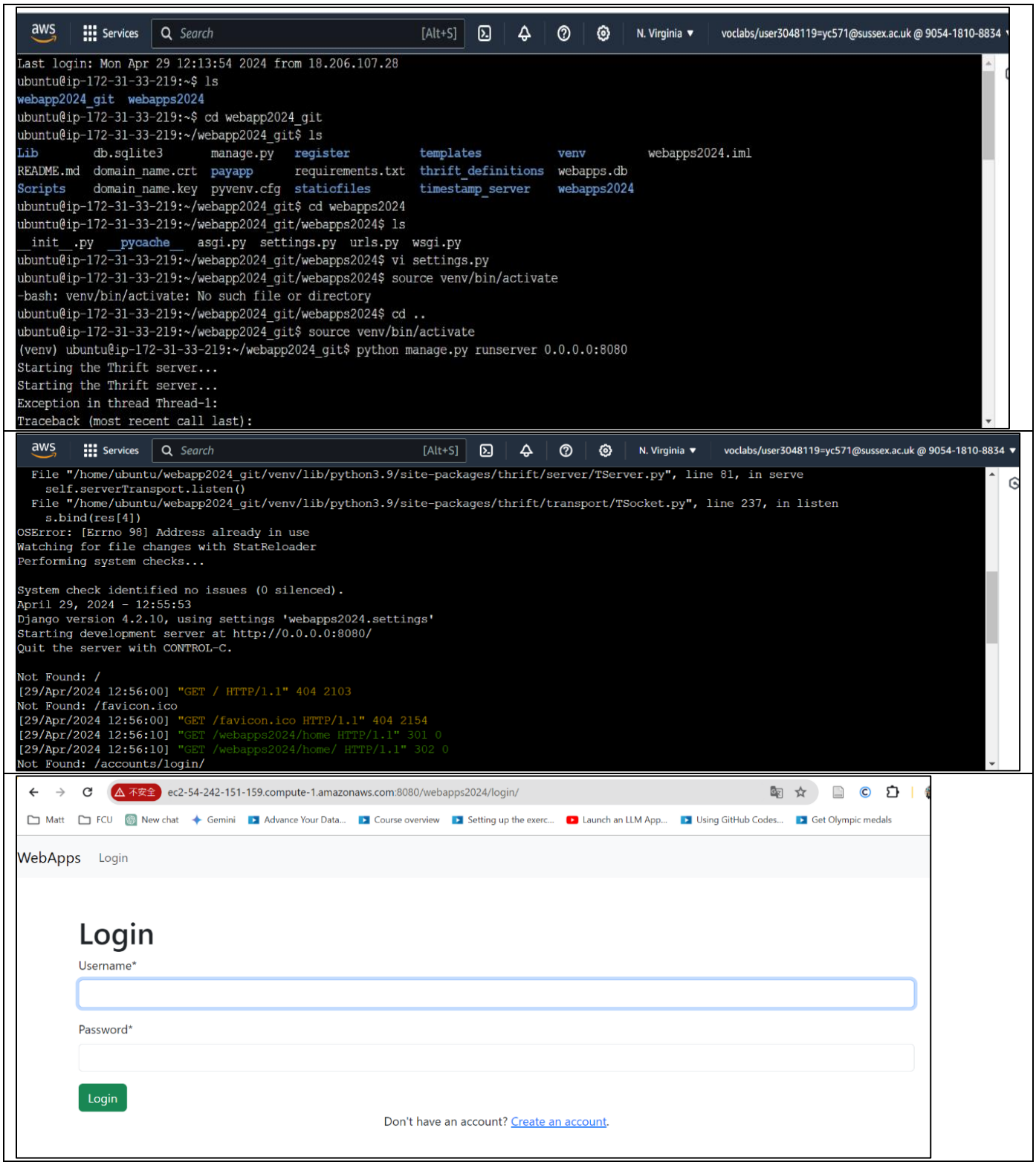
```python
from django.http import JsonResponse

1 usage
def current_timestamp(request):
    # Call the get_timestamp function that uses Thrift to fetch the current timestamp
    current_time = get_timestamp()
    if current_time:
        return JsonResponse({'current_time': current_time})
    else:
        # In case of an error, return a default error message or handle as needed
        return JsonResponse( data: {'error': 'Could not fetch the timestamp'}, status=500)
```

(screenshot25)

Cloud

The application on an Amazon EC2 virtual machine has been successfully deployed. Screenshots of the commands issued on the console to run the Django web application and screenshots of the application running on the cloud where the URI of the application are shown in (screeshot26).



(screenshot26)