Online Payment Service Report

---

1. Presentation Layer

The presentation layer was implemented using **Django templates**, enabling efficient interaction for both users and administrators.

Implemented Features:

- **User Authentication:**
  - Register, login, and logout functionality handled by a dedicated register app.
  - Templates located at templates/register and templates/login.
- **Transaction Overview:**
  - Users can view their sent and received transaction history via their profile.
- **Send & Request Payments:**
  - Users can send payments to other registered users.
  - Users can request payments, which appear as notifications. Currency conversions are handled automatically.
- **Admin Panel:**
  - Admins can view all user accounts and transaction records.
  - Admins can promote users to admin and view data in different currencies.
- **Navigation:**
  - Seamless page transitions implemented.
- **Enhanced UI/UX:**
  - Styled using Bootstrap and Crispy Forms for responsive and elegant design.

---

2. Business Logic Layer

Business logic is handled in Django views (primarily in payment/views.py) to maintain separation of concerns and ensure transaction atomicity. Logic is distributed based on user role (standard user vs. admin) and action (send/request/payment). When making migrations we upload some currency conversions as well as admin user to the

Implemented Features:

- **User-Level Views:**
  - pay_to_user: Sends payments and displays transaction history.
  - request_payment: Allows users to request payments from others.
- **Admin Views:**
  - promote_to_admin: Promotes a user to admin.
  - check_transactions: Allows filtering and reviewing of all user transactions.
- **Authentication & Routing:**
  - register, login_view, and logout_view handle user session flow.
- **Transaction Management:**
  - Payment operations are wrapped in Django's transaction.atomic block to ensure ACID compliance.
- **Currency Conversion Logic:**
  - Automatically triggered during transactions via models and views.

Form Logic Integration:

- Forms like PaymentForm, RequestPaymentForm, and CheckTransaction are closely tied to these views to handle input and validation.
- Admin actions and transaction requests are tied with real-time model updates to maintain business logic consistency.

---

3. Data Access Layer

Utilizes **SQLite** and Django ORM to ensure data integrity and scalability. The logic is divided into structured models that encapsulate both user data and transaction data.

Models Used:

- Profile:
  - Extends the default Django user.
  - Fields: balance, admin flag, currency
- Pay:

    - Logs a payment transaction.
    - Fields: payer, payee, amount, timestamp, success
- RequestPayment:
    - Represents a payment request between users.
    - Fields: requester, payer, amount, timestamp, success
- Currency:
    - Stores static exchange rates for conversions.
    - Fields: code, name, value

Forms for Data Handling:

- RegisterForm: Collects user data during sign-up.
- PaymentForm: Initiates a payment.
- RequestPaymentForm: Initiates a request for payment.
- CheckTransaction: Admin form to filter transactions.

Data Flow:

- Django ORM handles all read/write operations.
- ForeignKey relationships maintain referential integrity between users, profiles, and transactions.

---

4. Security Layer

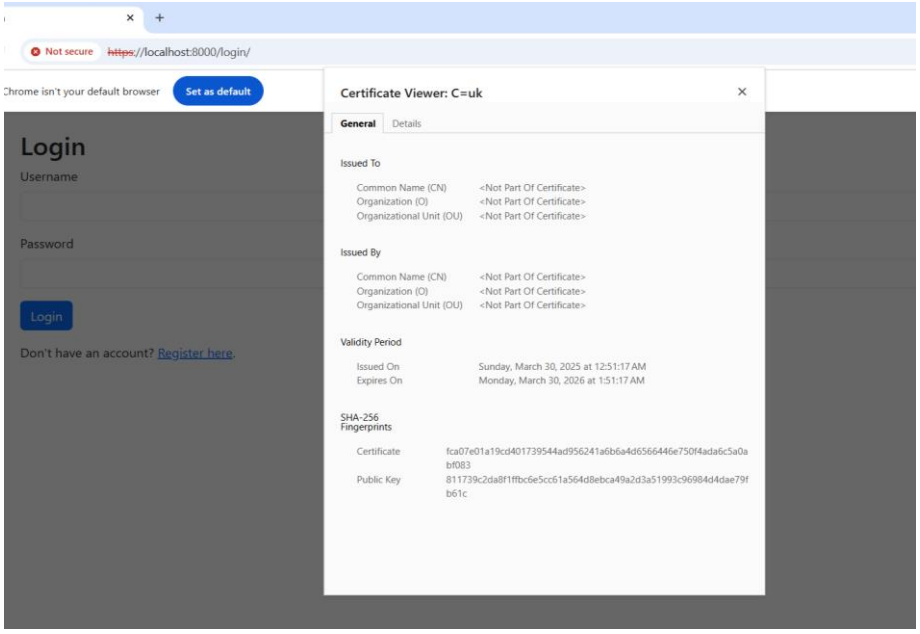
Robust security practices are applied across the application. Secure connection setup using generated public/private keys and certificates.

Implemented Features:

- **Authentication**: Register, login, logout**.**
- **Access Control:** Admin access controlled via profile fla**gs.**
- **Secure Server Login and Key Generation:**
    - Log into Unix server:
      ssh kb655@unix.sussex.ac.uk
    - Generate private key:
      openssl genrsa -out localhost.key 2048

- Create Certificate Signing Request (CSR)
- Generate Certificate:
  openssl x509 -req -days 365 -in localhost.csr -signkey localhost.key -out localhost.crt
- Use FileZilla or SCP to securely download the generated .key and .crt files to your local machine.

- **HTTPS Support:**
  - Enabled with *localhost.crt* and *localhost.key*.
  - Run the server securely: *python manage.py runserver_plus --cert-file localhost.crt --key-file localhost.key*



- **Security Protections:**
  - XSS: Certificate validation, ALLOWED_HOSTS configuration.
  - CSRF: {% csrf_token %} included in all templates.
  - SQL Injection: Prevented via Django ORM.
  - Clickjacking: Prevented using security header**s.**
- **Default Admin Account: admin1/admin1 initialized by default.**

---

5. Web Services (REST API)

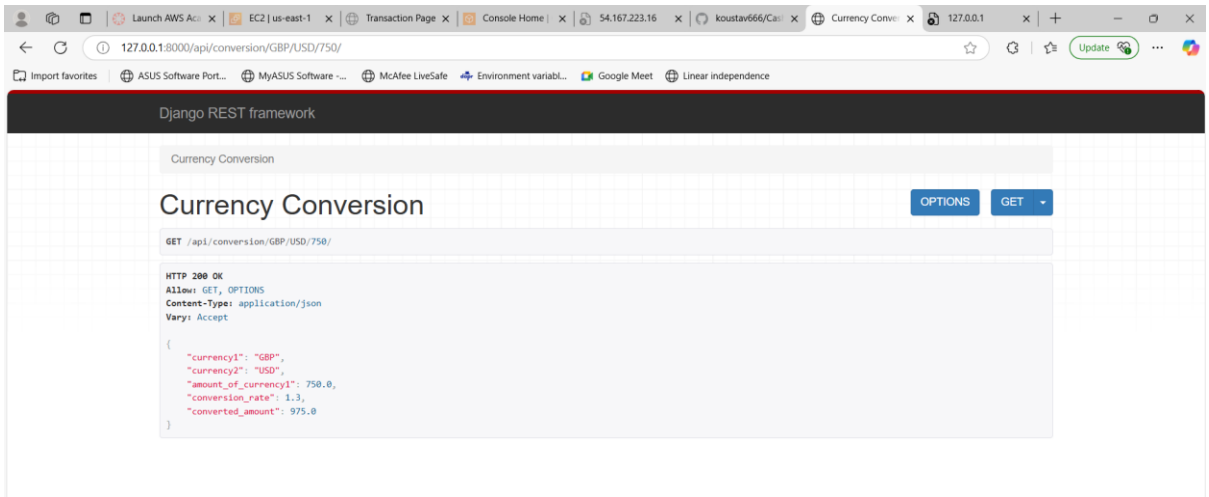
A RESTful **Currency Conversion API** is implemented in a **separate app** called currency_conversion.

Endpoint:

/conversion/{currency1}/{currency2}/{amount}

Features:

- Supports conversion between **GBP**, **USD**, and **EUR**.
- Exchange rates are statically loaded to the database during migration.



---

6. RPC with Apache Thrift

RPC service built using **Apache Thrift** to timestamp transactions.

Features:

- The interface is defined in a timestamp_service.thrift file.
- Thrift server is started using:

  thrift --gen py timestamp_service.thrift

  python timestampserver.py
- The server listens on **port 9090** and client timestampclient.py provides the method get_remote_timestamp().
- The client is integrated into the Django app and called during each payment or request transaction to provide accurate, real-time timestamps.
- This enhances traceability and trust for all financial operations.

---

7. Cloud Deployment (AWS EC2)

Deployed using **AWS EC2 instance** (created in Lab 2).

Deployment Steps:

1. Clone GitHub repository:

   git clone https://github.com/koustav666/Cash-app.git

2. Navigate:

   cd Cash-app

3. Create & activate virtual environment:

   python3 -m venv env

   source env/bin/activate

4. Install required packages.
5. Initialize database:

   python3 manage.py makemigrations

   python3 manage.py migrate

6. Run server:

   python3 manage.py runserver 0.0.0.0:8000

7. Ensure port 8000 is open in EC2 security group and inbound rule is defined.
8. Add the EC2 Public IP 98.81.241.55 to ALLOWED_HOSTS in settings.py.
9. Access the application from any browser at:

http://98.81.241.55:8000/{endpoints}

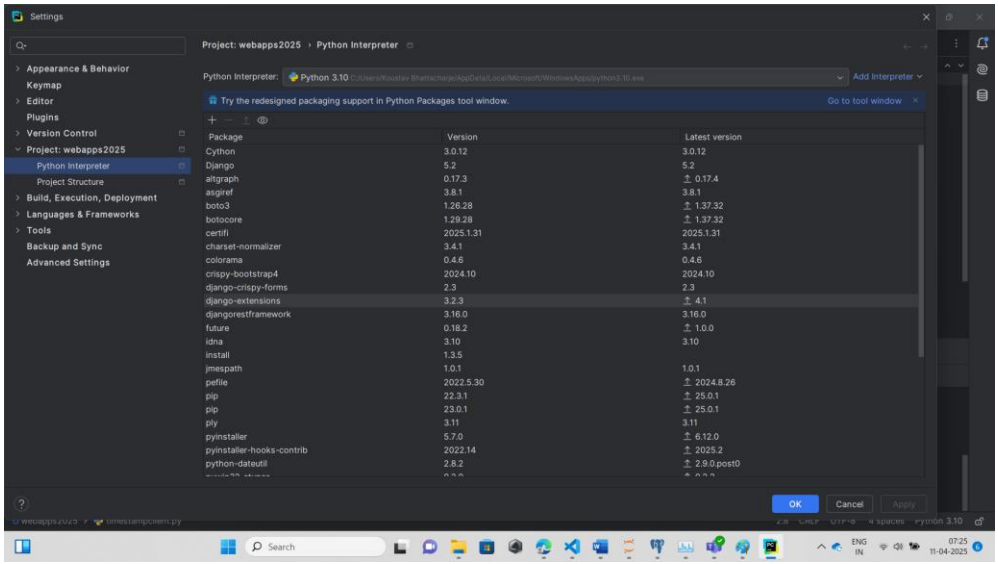
---

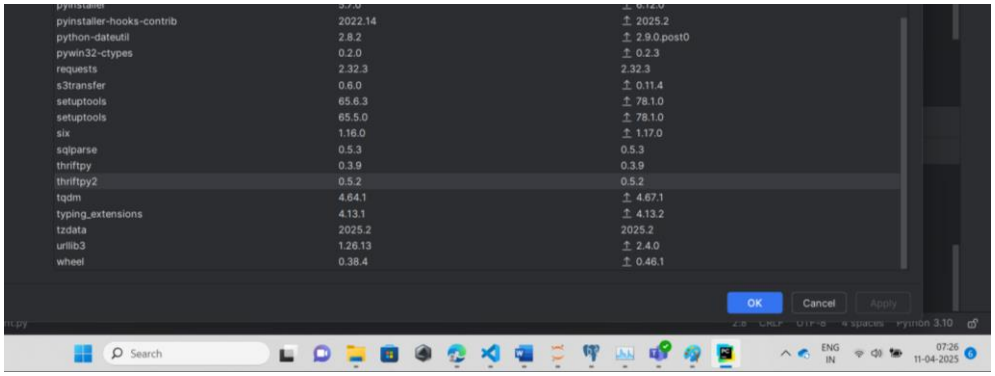
8. Used Packages(Install before running launching the app)

- djangorestframework
- thriftpy or thriftpy2(used here because deployment server could not install thriftpy)
- pyOpenSSL
- crispy-bootstrap4
- crispy-forms
- requests

- django-extensions (for runserver_plus)

Screenshot of packages present





---

9. Forms, Models, and Views Breakdown

🔹 Register App

Views:

- register: Registers user and creates profile.
- login_view: Authenticates and redirects.
- logout_view: Logs user out.

Forms:

- RegisterForm (extends UserCreationForm):
  - Fields: username, email, password1, password2

- **Payment App**

Views:

1. pay_to_user: Send payments and display transactions.
2. promote_to_admin: Admin feature.
3. check_transactions: Admin-only filtered view.
4. request_payment: Users request payments.

Forms:

1. PaymentForm: ModelForm for Pay (fields: payee, amount).
2. CheckTransaction: Regular form (fields: payee, currency).
3. RequestPaymentForm: ModelForm for RequestPayment (fields: payer, amount).

Models:

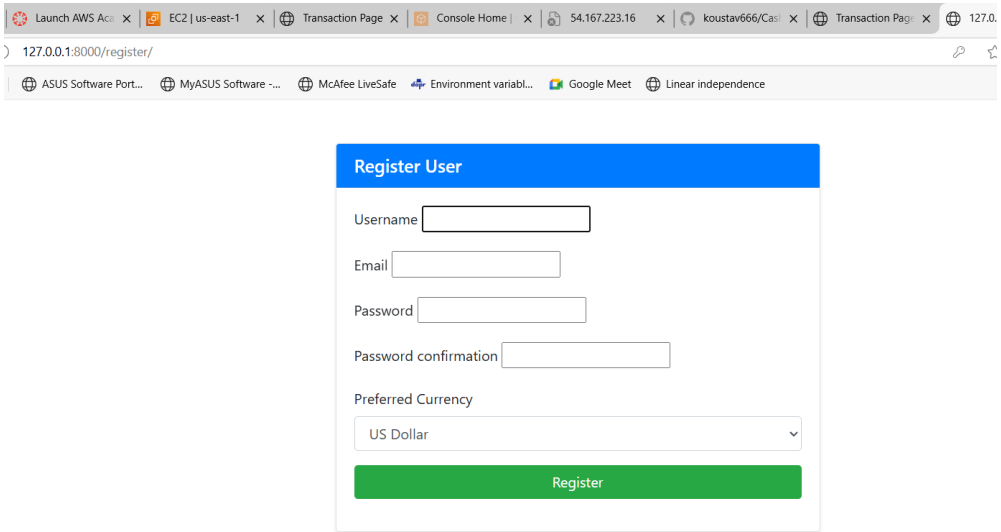
1. Currency: Stores exchange rates.
2. Profile: Extends user with balance, admin flag, currency.
3. Pay: Represents a payment.
4. RequestPayment: Stores payment requests.

---

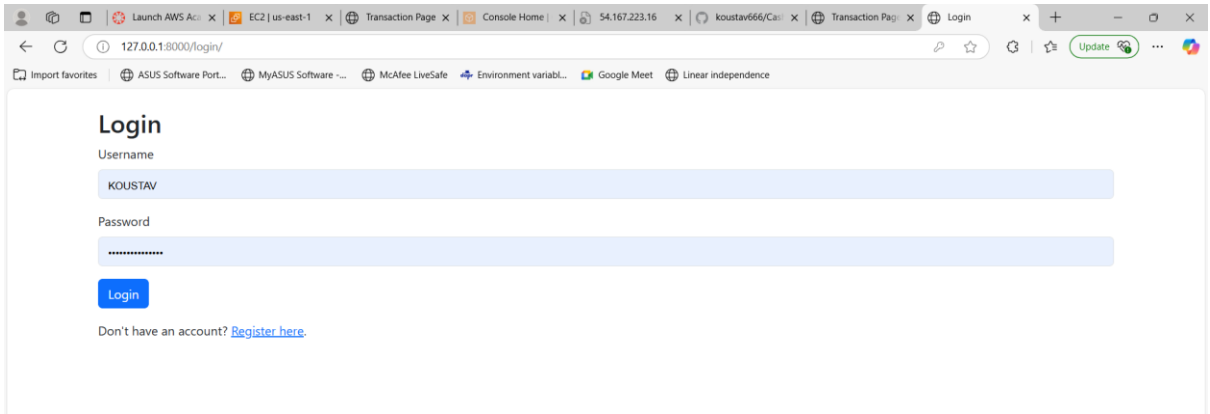
10. User Manual (Endpoints are mentioned in the screen shots)

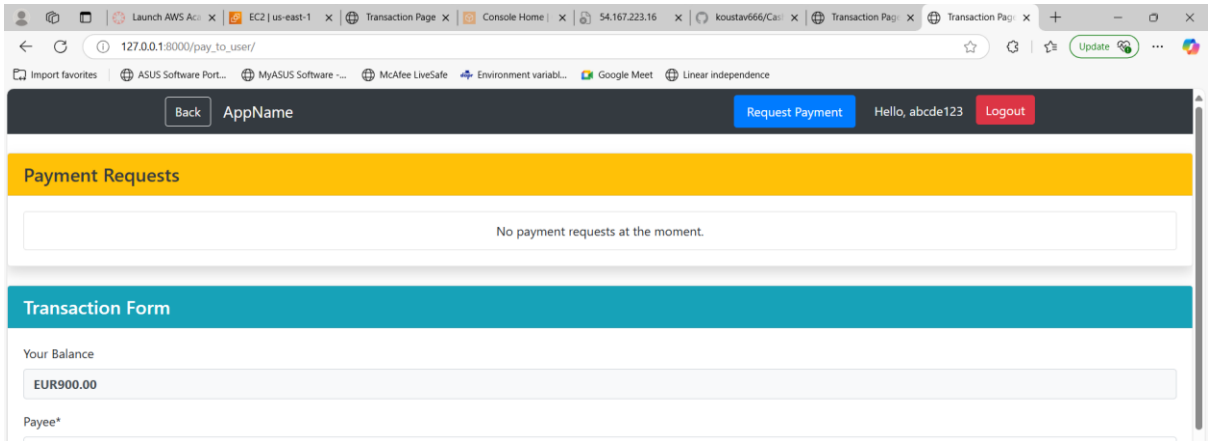
For Users:

1. Register with details.

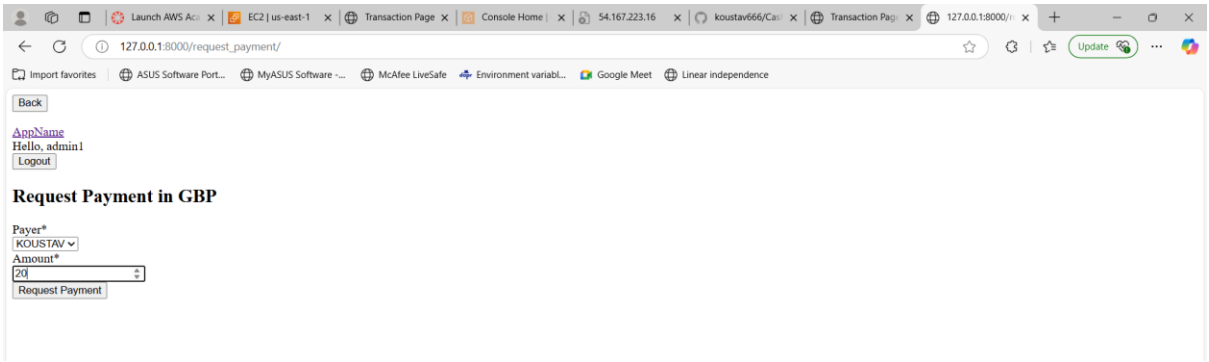


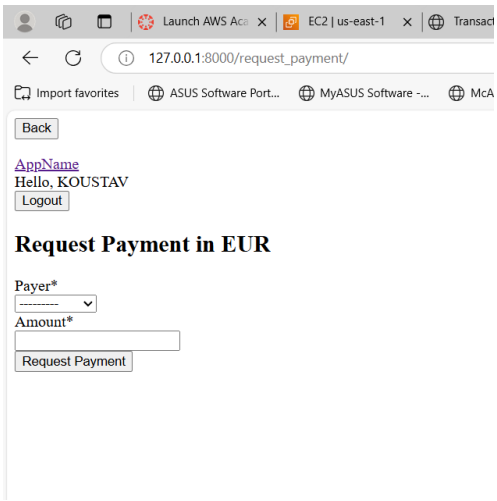
2. Log in to dashboard.



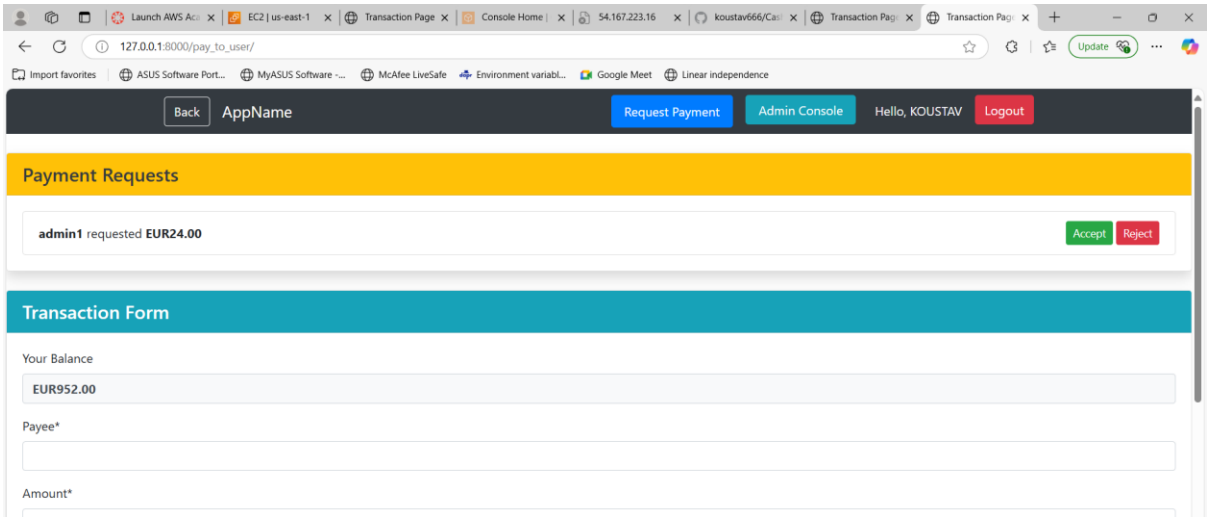


3. Request payments.

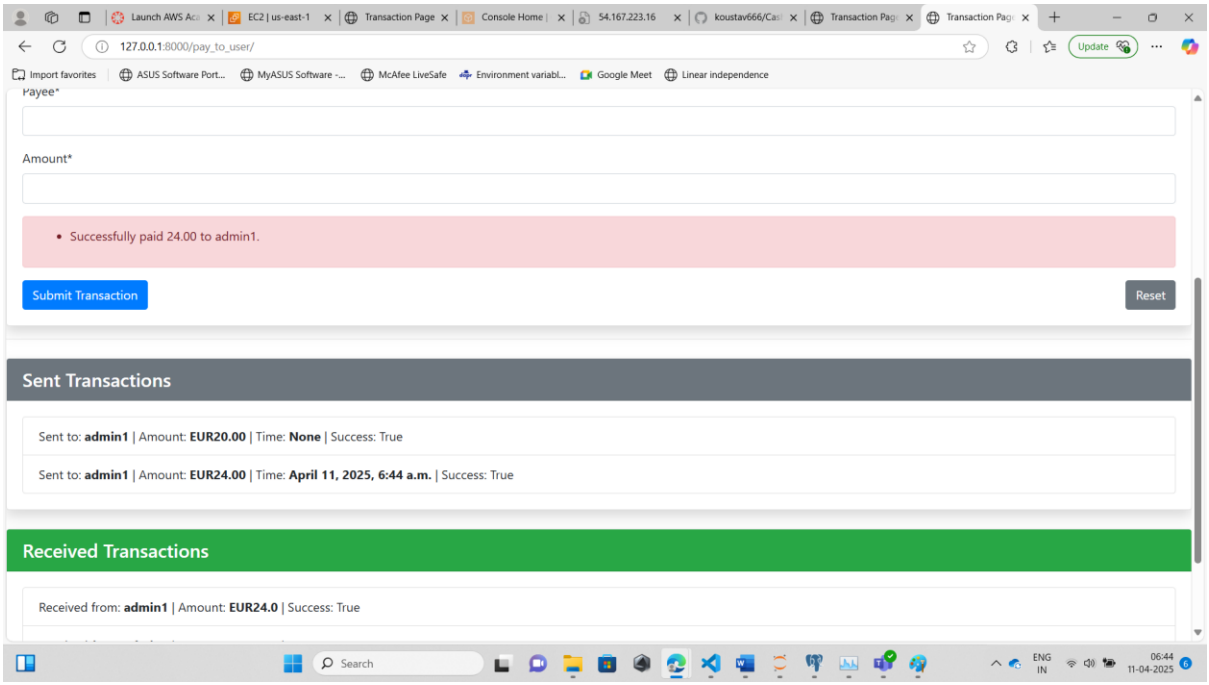




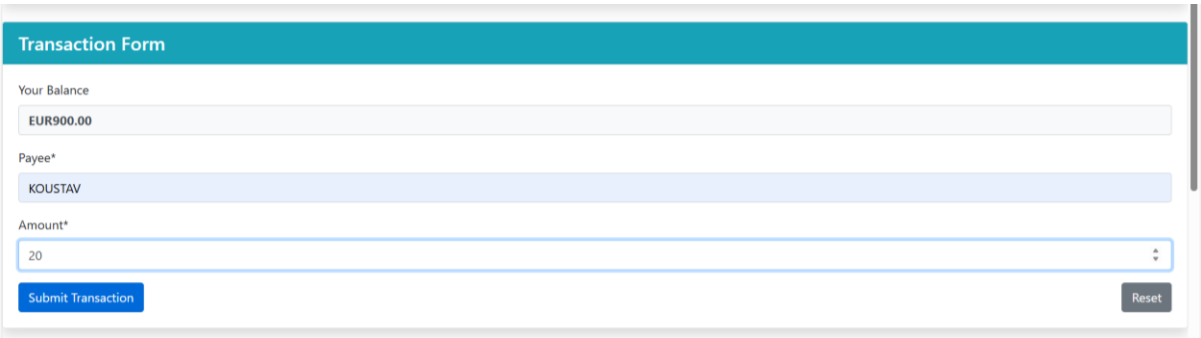
4. Respond to payment requests.

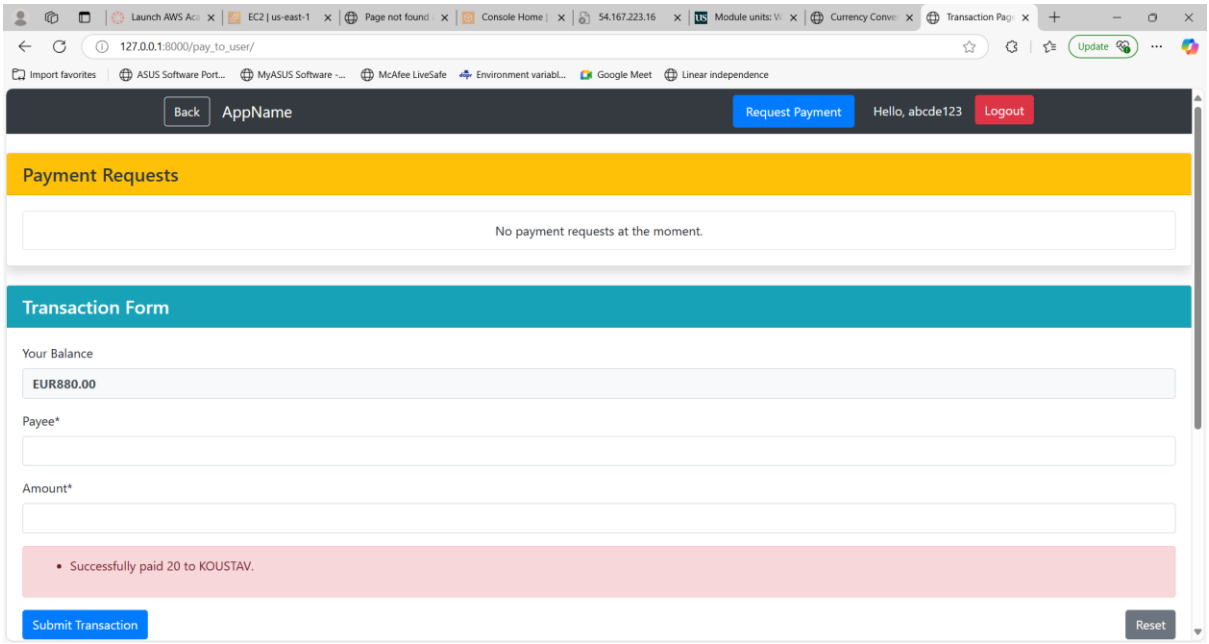


5. Transaction History

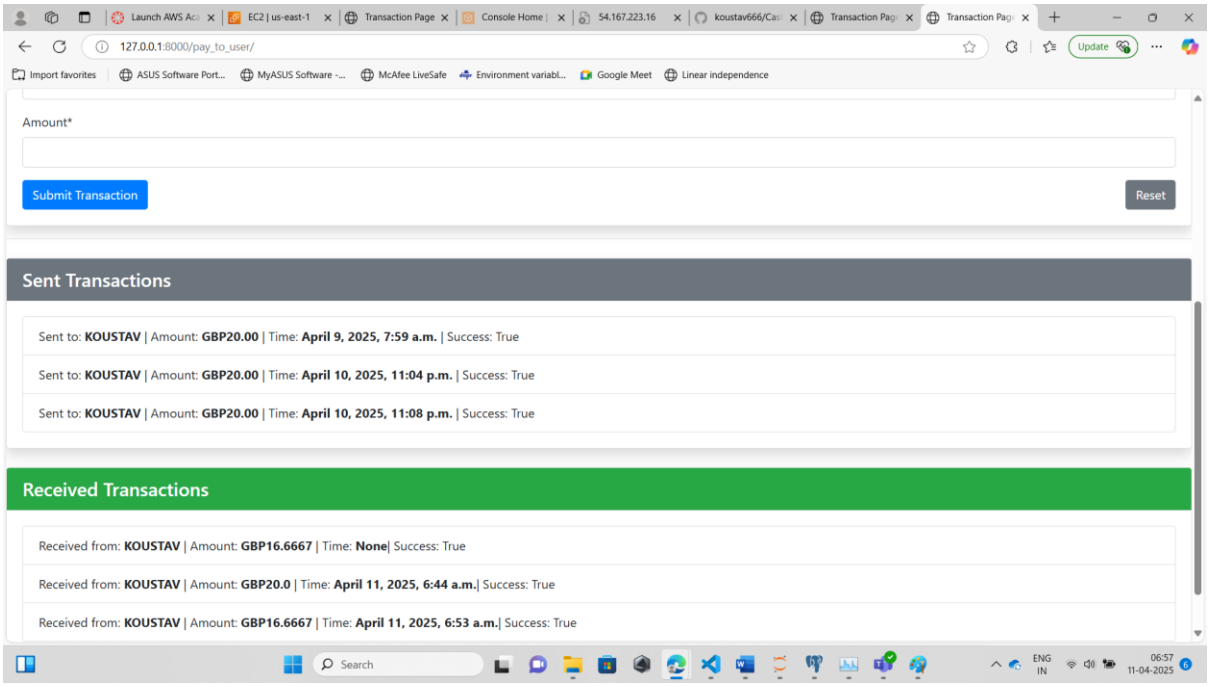


6. Send payment



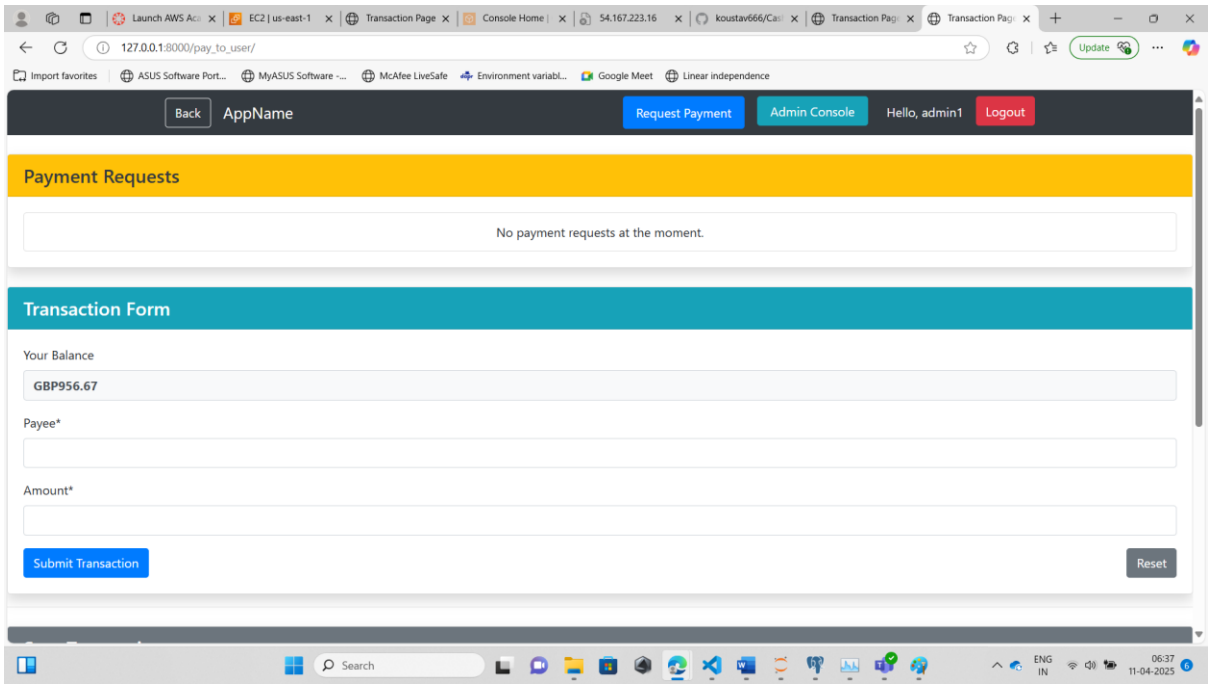


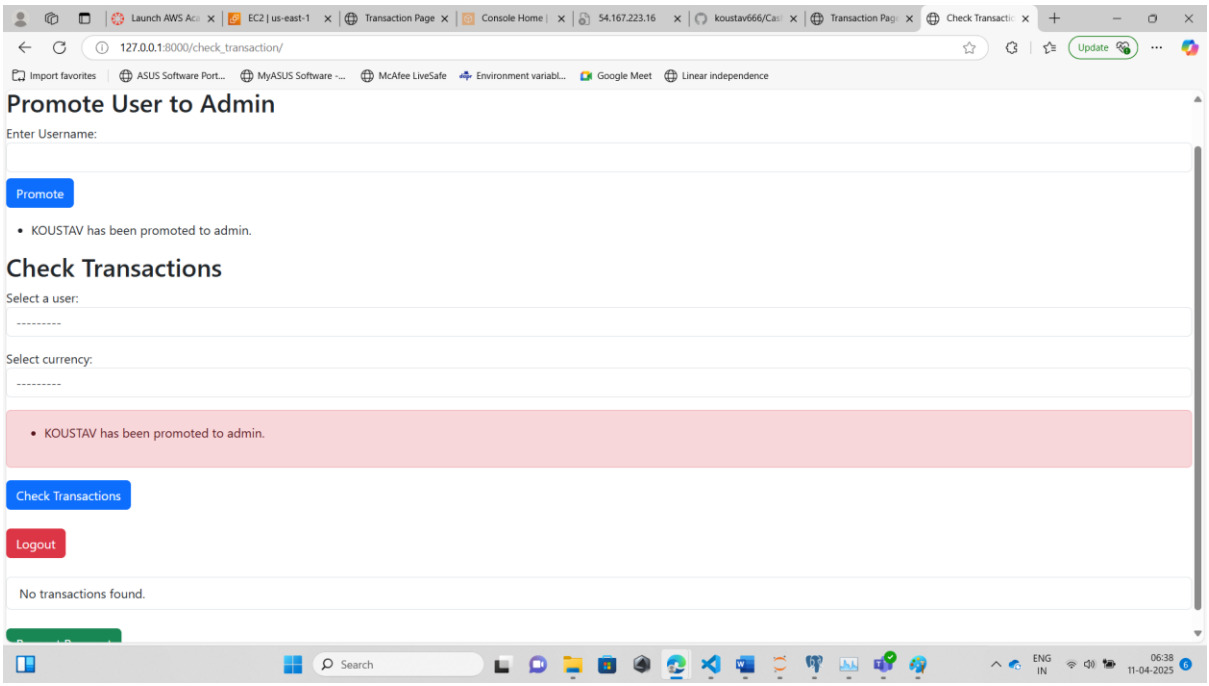
7. Appropriate conversions are made using the currency conversion REST API



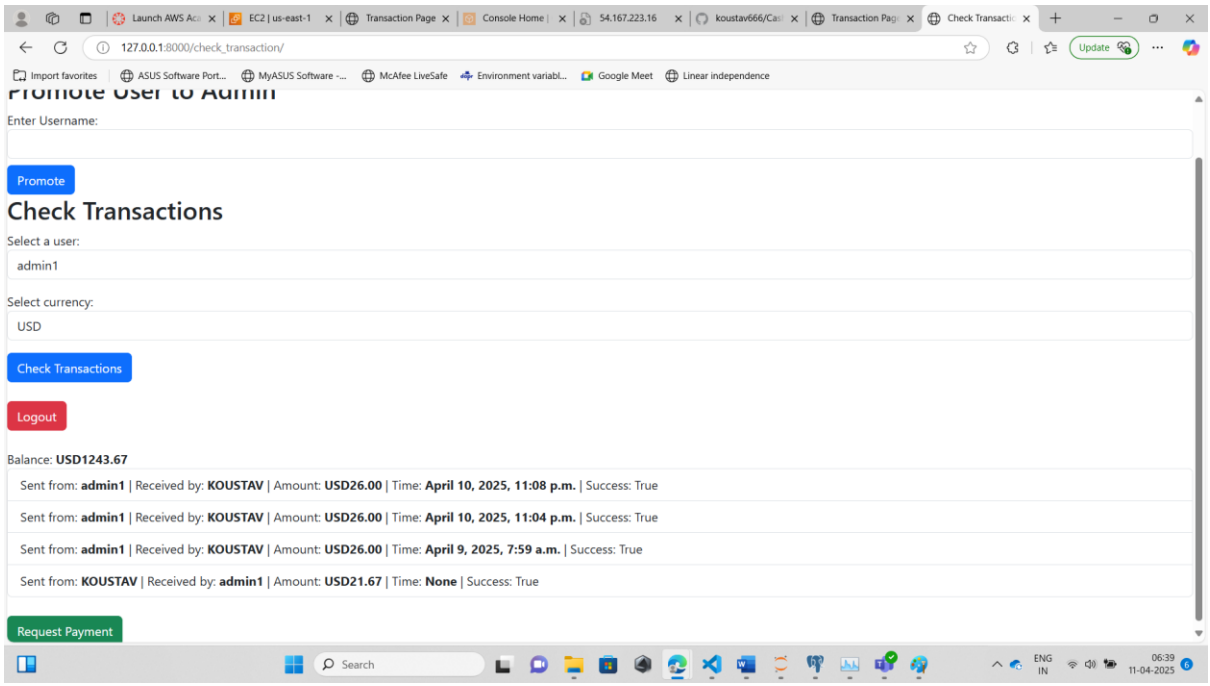
For Admins:

1. Log in with admin1/admin1. It has an additional button Admin console



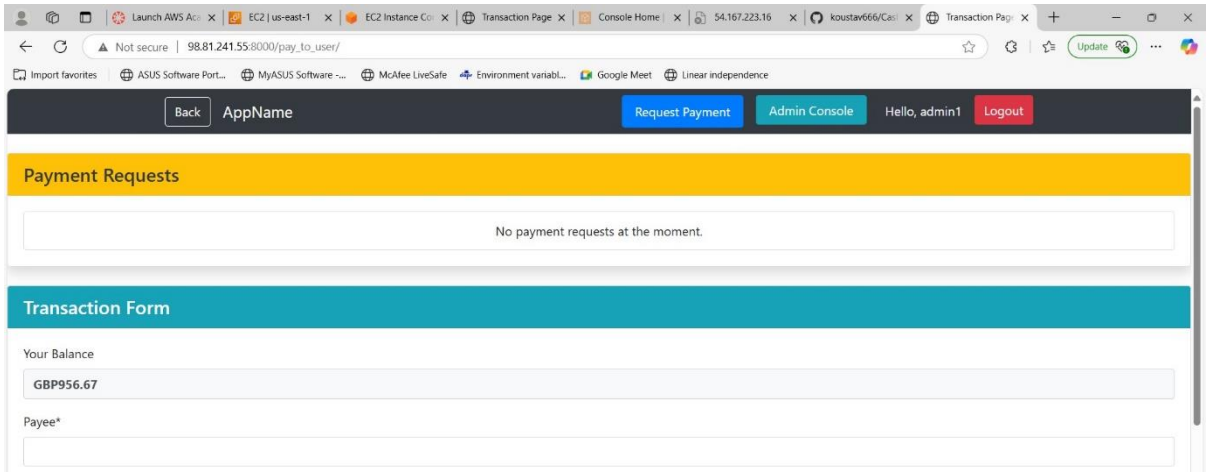


2. View user data and promote users.

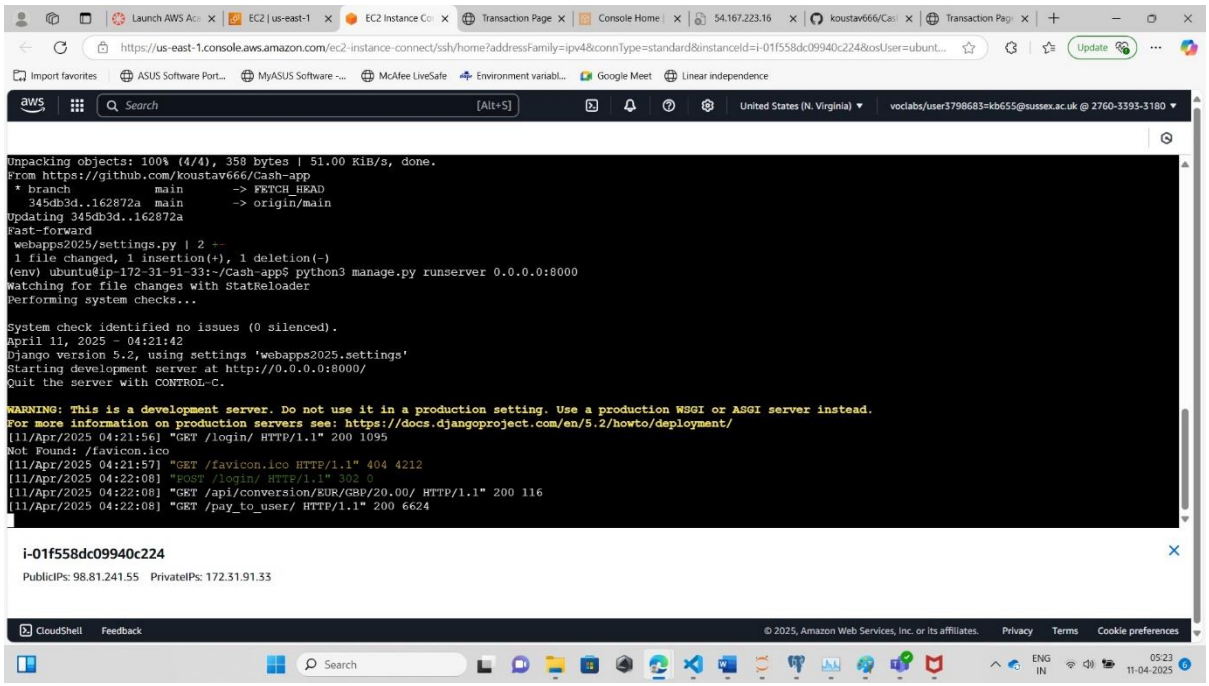


AWS screenshots:

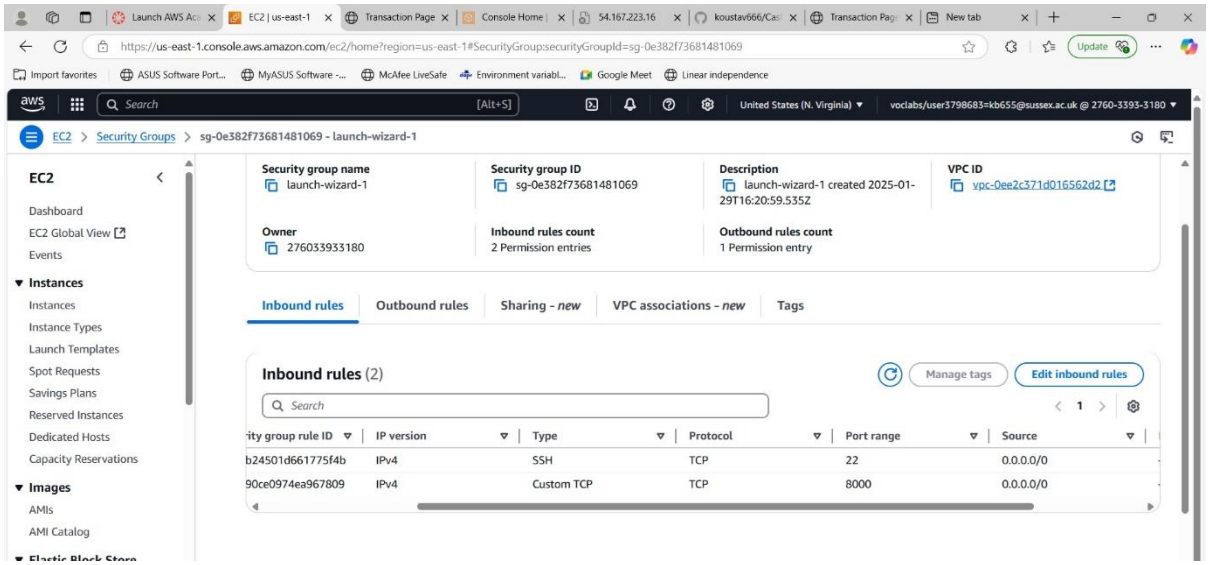
1. Running on 98.81.241.55 IP



2. EC2 logs



3. Security group inbound rules



---

11. Conclusion

This project implements a secure, modular, and user-friendly **online payment system** using Django.

Key Achievements:

- Secure and atomic transaction handling
- Admin and access control
- REST API (separate app for currency conversion)
- RPC (timestamping via Apache Thrift)
- Deployment on AWS EC2 (IP: 98.81.241.55)

The application demonstrates **layered architecture**, **modular design**, and **full-stack development** principles, making it scalable for real-world use.

Potential Fixes:

- The messages appearing are a bit buggy and needs to be improved.
- The Pay model needs to be more detailed as we need to call the conversion API lots of times to convert amounts into the user's currency. This increases the response time.
- The request payment layout could be improved
- Overall all functionalities are ok but better message labelling like '20EUR has been paid to user' needs to be done.
- Certificates don't have an **issued to** and **issued by** section as it was left blank.