Community Support & Financial Empowerment Platform in Egypt

A transformative initiative designed by students of the German International University to bridge the gap between charitable support and sustainable economic independence in Egypt’s developing economy.

Team Members:
Sarah El-Feel
Danya Danish
Alaa Ashraf
Abdelrahman Elnagar
Abdelrahman Samir
Omar Sherif
Mohamed Hossam
Hussein Mansour

Table of Contents

Milestone 1

Introduction

Vision and Mission Statement

- The platform’s vision is to create a connected, transparent, and supportive ecosystem where financial assistance and developmental resources empower individuals and communities to achieve long-term economic stability. Its mission is to harness innovation to build trust, streamline continuous aid distribution, and promote growth, self-reliance and cohesive community among Egyptians.

Key Objectives

- **Empowerment**: Enable low-income individuals to access financial support etc..
- **Transparency**: Build donor confidence by using blockchain to track and report on donations avoiding any corruption inside the organization.
- **Community Engagement**: Foster active participation from community members and local organizations through crowdfunding, feedback systems, and event participation.
- **Scalability and Sustainability**: Use partnerships with businesses and NGOs to expand services and build a self-sustaining model.

Core Values

- **Integrity and Transparency**: Ensuring that all operations are traceable and transparent.
- **Collaboration**: Strengthening ties between stakeholders
- **Sustainability**: Promoting financial independence over repeated dependence on aid.
- **Inclusivity**: support accessible to underserved and remote areas by digital solutions.
- **Inclusivity and Neutrality**: The organization operates without affiliation to any political parties or certain religion, ensuring unbiased support for all community members.

Stakeholders

- **Primary Stakeholders**: Donors, NGOs, local community leaders, low-income individuals and families.
- **Secondary Stakeholders**: Local businesses, financial institutions, government agencies, and policymakers.

Importance and impact

- This platform is significant in the context of Egypt's economic and social landscape, as it addresses the gaps in existing charitable efforts by promoting a long-term vision for economic empowerment. The platform’s ability to leverage modern technologies ensures scalable solutions that adapt to the dynamic needs of communities
- **Economic Growth**: By supporting micro-investments, the platform contributes to workforce development and economic stimulation.
- **Increased Trust**: Transparent donation tracking builds trust among donors, encouraging more contributions.
- **Education and Financial Literacy**: Equips individuals with essential skills for managing finances and building sustainable livelihoods.

Scenario for the System

Background

Ahmed, a successful business owner in Cairo, wants to give back to his local community by making charitable donations. Fatima, a single mother in the rural town of Beni Suef, struggles to cover her children’s medical expenses. The NGO "Life Makers" is planning a medical camp to provide check-ups and treatment in Upper Egypt. Layla, a university student in Alexandria, is eager to volunteer for such initiatives. This platform connects these stakeholders seamlessly, tailored to Egypt’s unique cultural and logistical context.

Interaction Scenario:

1. Donor Interaction (Ahmed):

- **Registration and Verification**:
    - Ahmed visits the platform's website and registers as a donor, filling out his profile with personal details and business credentials. His identity is verified by cross-checking with Egypt’s national ID database.
    - The platform approves his account within 48 hours, notifying him via SMS in Arabic, the preferred communication method.
- **Browsing and Donating**:
    - Ahmed logs in and browses a list of requests from beneficiaries and projects like Fatima’s plea for medical support and "Life Makers" upcoming medical camp in Minya.
    - He donates to Fatima’s request through an easy-to-use payment portal supporting local Egyptian banks and mobile payment options (e.g., Fawry, Vodafone Cash, telda and instapay).
- **Tracking and Notifications**:
    - Ahmed receives a notification in Arabic via SMS and email, confirming his donation. He also gets a message of gratitude from Fatima and "Life Makers" via the platform.
    - His donor dashboard, customized with icons and terms familiar in Egypt, displays a real-time progress tracker showing how the funds are used, including updates on Fatima’s treatment and the medical camp’s preparation.

2. Beneficiary Interaction (Fatima):

- **Registration and Aid Request**:
    - Fatima registers through the platform’s mobile app, available in Arabic for ease of use. She uploads proof of her income and medical documents, verified by the platform’s team in collaboration with local government offices in Beni Suef.
    - Once approved, her request is made public, complete with details like her children’s medical needs and expected treatment costs.
- **Receiving Aid**:
    - Fatima receives an SMS alert notifying her that Ahmed has funded her request. The platform transfers the money to her digital wallet linked to an Egyptian mobile payment service or postal order.
    - She uses the wallet to pay directly at a local medical center, a partner of the platform, or transfers funds to her bank account at Egyption national bank.
- **Updates and Engagement**:
    - Fatima posts a heartfelt thank-you video in Arabic, showing her children receiving their treatment and thanking Ahmed and the community for their support.

3. NGO Interaction (Life Makers):

- **Event Planning and Volunteer Recruitment**:
    - "Life Makers" logs into their account and posts an event for the upcoming medical camp in Minya, specifying volunteer roles needed, such as nurses, registration staff, and on-site assistants.
    - Verified volunteers in the region, including Layla, receive notifications in Arabic about the opportunity.
- **Donation Requests**:
    - The NGO lists specific items and equipment needed for the camp (e.g., medical supplies and food for volunteers). The platform sends targeted messages to donors like Ahmed.
- **Volunteer Management**:
    - "Life Makers" reviews applications from interested volunteers and schedules a briefing session in Cairo and online for those from different governorates.

4. Volunteer Interaction (Layla):

- **Registration and Profile Setup**:
    - Layla creates a volunteer profile, filling in her details in Arabic and uploading certificates to showcase her previous volunteer work.
    - She is verified by the platform and approved by "Life Makers" after an online interview conducted via Zoom.
- **Notifications and Event Participation**:
    - Layla receives reminders via SMS and app notifications about her upcoming volunteer shift, with details like transportation to Minya and accommodation.
    - On the day of the event, Layla checks in using a QR code through the app, updating the attendance system for "Life Makers".
- **Feedback and Recognition**:
    - After the medical camp, Layla receives a thank-you note in Arabic and is encouraged to share her experience. Her volunteer hours are logged, increasing her credibility on the platform.

5. Administrative Oversight:

- **Platform Monitoring**:
    - The platform's admin team monitors activities to ensure transparency and authenticity, using local partnerships with government offices to verify beneficiaries and NGOs.
    - The team reviews Fatima's and Layla's profiles and validates donation and expenditure records.
- **Donation Tracking and Reporting**:
    - The admin uses a specialized dashboard tailored for the Egyptian context to track donations, oversee fund allocation, and compile detailed reports. Donors like Ahmed receive monthly impact reports, fostering trust and a sense of contribution.
- **Community Engagement**:
    - The admin team sends out memos and success stories via SMS and social media platforms like WhatsApp and Facebook, popular among Egyptians, to encourage further participation.

Outcome

Ahmed gains satisfaction from directly supporting his community, witnessing the real impact of his contributions. Fatima receives essential medical aid for her children, enhancing their quality of life. "Life Makers" successfully runs a medical camp, strengthening its reputation and extending healthcare services. Layla builds experience and grows her volunteer network, deepening her commitment to social work. The platform thrives as a reliable tool, creating a sense of unity and shared responsibility across Egypt.

Functional Requirements

Donor Management

- Donors can register to create a profile, enter all the legal requirements and get verified within 24 hours and then manage their accounts.
- Donors can send monetary donations cash or by online payment, track their donations through dashboards, and receive notifications on how their funds are being utilized with a thanks message.
- Donors can browse beneficiaries' posts and pay them.

Volunteer Management

- Volunteers can register and manage their profiles.
- Volunteers can browse events organized by NGOs and apply for participation and receive the timing and location to be interviewed for the first time with the NGO or event details if already verified from the organization.
- Volunteers receive notifications for upcoming events and shifts they are scheduled for.

NGO Management

- NGOs (including local businesses and food banks) can create profiles, organize events (workshops,recurring events, health camps), and specify the types of volunteers needed and wait 24 hours to be verified by the board.
- NGOs can request donations for specific projects or events and send notifications to registered donors.
- NGOs receive the details of the Volunteers applied with their projects to interview them.

Beneficiary Management

- Beneficiaries can register, create profiles .
- Beneficiaries can submit detailed requests for assistance providing proof and can be connected to either an NGO or a direct donor within a week.
- Beneficiaries can post funding requests for specific needs, such as medical bills or community projects.
- Beneficiaries receive aid or payments into a secure digital wallet or transferred to a bank account or postal order in most cases.

Administrative Access and Control (Board)

- The board has access to browse all NGOs in the system and aid requests.
- The board can segment the Volunteers and NGOs based on the data collected.
- Receive the beneficiaries' requests and verify them then send them to the NGOs
- The board has a centralized management system for tracking and distributing donated money, food, clothing, and essentials.
- The board verifies the NGO accounts and activates them.
- The high Donor segment is assigned to deal with the NGOs directly to manage their charitable contribution deductions from taxes.

Why are they considered functional requirements?

These requirements are functional for software design and architecture as they define distinct roles (Donors, Volunteers, NGOs, Beneficiaries, and the Board) with clear functionalities, ensuring a comprehensive ecosystem for effective charity management. They address user registration, verification, account management, and tailored services like tracking donations, browsing events, or requesting aid. Notifications, dashboards, and payment handling enhance user experience and transparency. Administrative controls provide centralized oversight for segmentation, verification, and resource distribution, ensuring accountability. The modular nature facilitates scalability and system coherence, aligning with real-world needs while supporting user engagement and operational efficiency.

Non-functional Requirements

1. **Scalability**
    - The system must be capable of scaling to support an increasing number of users (donors, NGOs, volunteers, and beneficiaries) up to 30 Million active users.
    - The infrastructure should accommodate expanding operations such as increase in posts and active users.
    - Partner with financial institutions and mobile network providers to extend wallet functionalities and reach.
2. **Performance**
    - The platform should support high concurrent user activity without noticeable delays or degradation in response time to be within 0.1 second.
    - The system must handle a large volume of data up to 500 GB, ensuring quick load times for dashboards and seamless reporting functionalities.
3. **Security**
    - Implement multi-layered security protocols to protect user data, including encryption, secure authentication, and role-based access control.
    - Ensure all financial transactions and personal data handling adhere to best security practices to safeguard against breaches and unauthorized access.
    - Blockchain technology is used for secure and transparent tracking of donations and distributions to build trust.
4. **Transparency**
    - Ensure all financial and aid distribution activities are logged and verifiable by relevant stakeholders.
5. **Reliability**
    - Use reliable cloud to enhance uptime and consistent service.
6. **User Experience (UX)**
    - The system interface should be intuitive and easy to navigate with **Arabic Language and many icons**, ensuring a seamless experience for all user types.
    - The platform should offer user-friendly onboarding, tutorials, and support documentation for first-time users.
    - Text to speech functionality.
7. **Maintainability**
    - The system architecture should be **modular and well-documented** to enable efficient maintenance, updates, and troubleshooting.
8. **Compliance and Data Privacy**
    - Ensure compliance with local and international data protection laws (e.g., **GDPR, Egyptian Data protection Law 151)** and financial regulations to protect user privacy and manage financial transactions responsibly.
    - Ensure that personal and financial data are only accessible to authorized personnel through strict access control measures.

Conceptual Design

Design Patterns

Architectural

Microservices Architecture Pattern:

- **Justification**: This architectural style breaks down the platform into independently deployable services, each dedicated to a core functionality—such as donation tracking, volunteer coordination or community feedback. By modularizing the system, each service can operate autonomously, enhancing the flexibility to adapt to the evolving needs of donors, NGOs, volunteers, and community members. This separation also enables rapid deployment and modification of individual services, promoting long-term sustainability.
- Benefits:
    - **Scalability:** Each service can scale independently based on its specific needs. For example, If there's an increase in donations around a specific time, only the donation management service would need to scale up to handle the load. Other services, such as volunteer coordination modules, would remain unaffected..
    - **Maintainability:** Smaller codebases for each service mean less complexity during maintenance, reducing the likelihood of unforeseen bugs affecting other functionalities.
    - **Flexibility for Growth:** Future modules or features (e.g., advanced analytics for donor feedback or financial support tracking) can be incorporated with minimal conflicts, maintaining a cleaner, more flexible codebase as the platform grows.

Event-Driven Pattern:

- **Justification:** This pattern is ideal for facilitating communication between microservices by triggering and responding to events, such as user donations or volunteer registrations. It promotes a decoupled architecture where services interact asynchronously, enhancing the system's responsiveness and scalability.
- Benefits:
    - **Decoupling:** Reduces dependencies between services, improving modularity.
    - **Responsiveness:** Services can react to events in real time, enhancing user experience.
    - **Scalability:** Enables more dynamic resource allocation and handling of high loads efficiently.

Structural

Facade Pattern:

- **Justification:** With the facade pattern, each user role interacts with a specific interface designed for their needs, which abstracts the complexity of the backend. For instance, while the donation management, volunteer assignments, services may involve several subsystems working together, the facade pattern hides these intricate connections, presenting only what each user needs.
- Benefits:
    - **User-Centric Interface:** Each user role accesses only relevant functionalities, making the platform easier to use and enhancing the user experience.
    - **Loose Coupling:** The facade isolates user-facing components from backend services, allowing modular development and easier maintenance.
    - **Enhanced Security and Access Control:** Facades enforce role-based access control, simplifying permissions and protecting sensitive operations and data.
    - **Ease of Maintenance:** Changes to backend services don't directly impact the user interface, allowing smooth updates and scalable growth.

Creational

Factory Method Pattern:

- **Justification:** The Factory Method Pattern streamlines the process of creating user instances (e.g., donors, volunteers, NGO representatives, board members) by centralizing the instantiation logic. This abstraction is useful for managing different types of users with potentially varying initialization processes and attributes.
- Benefits:
    - **Loose Coupling:** By abstracting object creation, the system doesn't need to directly depend on specific classes, which adheres to the principle of separation of concerns.
    - **Open/Closed Principle:** The pattern makes it easy to extend the system with new types of users or roles without altering existing code, making the design more adaptable and maintainable.
    - **Simplified Code Maintenance:** Reduces repetitive code related to user creation and management, resulting in cleaner and more maintainable codebases.

Behavioral

Observer Pattern:

- **Justification:** This pattern suits the need for real-time notifications and updates within the system. For instance, when a donor makes a donation, or when an event is scheduled by an NGO, the system can push notifications to the respective users (e.g., donors, volunteers, beneficiaries) without tight coupling between components.
- Benefits:
    - **Decoupled System:** The observer pattern enables a loosely coupled system where different modules can subscribe to events without knowing each other's internal workings.
    - **Extensibility:** Adding or modifying notification types becomes straightforward, as the observers simply need to implement the necessary interfaces or subscribe to the event without changing the core logic.
    - **User-Centric Features:** This pattern supports timely updates and communication with users, which is essential for engagement and trust-building.

Software Decomposition

Understanding the Problem Domain

Identifying the Problem Domain and its Key Components:

- **Problem Domain:** Community Support & Financial Empowerment Platform in Egypt.
- **Key Components:**
  - **Donors:** Individuals or organizations providing financial support.
  - **NGOs:** Non-governmental organizations managing aid distribution and organizing community engagement programs.
  - **Volunteers:** Individuals participating in fieldwork, and assisting NGOs and beneficiaries.
  - **Beneficiaries:** Low-income individuals and families seeking financial aid, or support for stable income.
  - **Administrators:** Board members overseeing the platform's operations, verifying requests, and managing donations and users.

Analyzing Requirements and Determining the Scope:

- **Primary Objective:** Create a platform that connects donors, NGOs, volunteers, and beneficiaries, focusing on transparency, empowerment, and sustainability.
- **Key Features:**
  - Donation management and tracking.
  - Volunteer application for fieldwork.
  - Aid requests for beneficiaries.
  - Blockchain-based transparency for donation flow.
  - Financial services like digital wallets for beneficiaries.
  - Scalability for up to 30 million active users.

Recognizing Potential Areas for Decomposition:

- **Donor Management:** Profile creation, donation tracking.
- **Volunteer Management:** Registration, Event search, application.
- **NGO Management:** Event creation, donation requests, volunteer assignments.
- **Beneficiary Management:** Requesting aid, and accessing digital wallets.
- **Admin Management:** Oversight of platform operations, including donation flow, volunteer segments, and NGO activities.

Functional Decomposition

Justification for Functional Decomposition:

Functional decomposition is appropriate for this platform because the problem domain involves multiple distinct functionalities that can be logically separated into smaller components. By breaking down the platform's functionality into smaller, manageable pieces, we can ensure that each part of the system can be developed, tested, and maintained independently.

Functional decomposition will help manage the complexity of different components such as donor management, volunteer management, and financial tracking, while ensuring that each component contributes to the overall system's objectives.

Detailed System Functional decomposition

Identify System Functions:

Based on the analysis of the requirements, we identify the following main functions for the platform:

1. **Donor Management:**
   - Register donors and verify their accounts.
   - Enable donors to make donations and track their contributions.
   - Provide donors with notifications and updates on how their donations are being used.
2. **Volunteer Management:**
   - Register volunteers and manage their profiles.
   - Enable volunteers to browse and apply to events.
   - Schedule and notify volunteers of their shifts or events.
3. **NGO Management:**
   - Allow NGOs to create profiles and organize events.
   - Enable NGOs to request donations for specific causes.
   - Allow NGOs to receive details of the Volunteers applied for the event.
4. **Beneficiary Management:**
   - Register beneficiaries and manage their requests for aid.
   - post funding requests for specific needs.
   - Provide digital wallets for aid distribution and transaction management.
5. **Admin Management:**
   - Allow the admin to manage platform-wide settings and oversee user activities.
   - Verify accounts for different users.
   - Verify and approve beneficiary requests for aid.
   - Track and manage donations and financial transactions.

    ○ Segment Volunteers and donors for analysis and reporting.

Break Down Functions:

Define Interfaces:

- **Data Interfaces:** Data will flow between modules such as donor profiles, transaction data, volunteer details, and beneficiary requests.
- **User Interfaces:** Each component will have its own user interface (UI) for donors, volunteers, NGOs, and beneficiaries.

Assign Responsibilities:

- **Donor Management:** Handle donor registrations, donations, notifications.
- **Volunteer Management:** Handle volunteer registrations, event scheduling and notifications.
- **NGO Management:** Handle NGO registrations, Handle event organization, donation requests, and volunteer coordination.
- **Beneficiary Management:** Handle Beneficiary Registration, aid requests, share aid posts, and payments receiving.
- **Admin Management:** Oversee the platform's operation, ensuring proper verification, tracking, and reporting.

Define Component Dependencies:

- **Donor Management** and **NGO Management:** Donors interact directly with NGOs through donation requests.
- **Volunteer Management** and **NGO Management:** Volunteers apply for events organized by NGOs.
- **Beneficiary Management** and **Admin Management:** Admins verify beneficiary requests for aid.
- **Donor Management** and **Beneficiary Management**: Donor interacts directly with the Beneficiary through donation posts.

Establish Hierarchy:

1. **Top-Level Components:** Donor Management, Volunteer Management, NGO Management, Beneficiary Management, Admin Management.
2. **Subcomponents:** Each main component is broken down into smaller sub-functions (e.g., registration, donation processing, event scheduling).
3. **Dependencies:** Functional components depend on each other to facilitate data exchange and communication (e.g., donors donate to beneficiaries via NGOs).

Refine and Iterate:

As development progresses, the system decomposition may be refined based on new requirements or insights. For example, sub-functions may evolve to support new features, or interfaces may be optimized for better communication between components.

Object-Oriented decomposition

Justification for Object-Oriented Decomposition

To break down our complex software system into smaller, more manageable and cohesive functional components or modules enabling us to modify and test each on its own.

Introduction to Decomposition

This section outlines the object-oriented decomposition for a platform aimed at bridging charitable support and economic empowerment in Egypt. The platform incorporates core functionalities such as donor management, volunteer coordination, NGO management, beneficiary management, and administrative control to ensure transparency, engagement, and sustainability so let’s break them into an Objects-Oriented System.

List of Identified Objects within each functional module:

- User: abstract base class for all types of users (Donors, Volunteers, Beneficiaries, NGO).
- Profile: A detailed profile attached to each user.
- Dashboard: Provides interfaces for donors and board members to track activity.

1. Donor Management

- **Donor(id, name, email, legalDocuments, verifiedStatus, dashboardData, walletBalance)**: Represents a user who contributes financial support to beneficiaries and NGOs.
- **PaymentMethod(lastDigits, type)**: Defines the various ways a donor can make payments (e.g., credit card, bank transfer).
- **Transaction(transactionId, sender, receiver, amount, type,timestamp, PaymentMethod)**: Logs financial movements and interactions on the platform between all the entities across the platform.

2. Volunteer Management

- **Volunteer(id, name, appliedEvents, pastEvents, segment)**: Represents an individual who offers their time and services to assist in NGO activities and events.
- **VolunteerApplication(applicationId, date, details, type, RequesterId)**: Manages the process of volunteers applying for specific events.

3. NGO Management

- **NGO(id, organizationName, NGOtype, eventList, donationRequests):** Represents non-profit organizations that create and manage events, request donations, and organize volunteer work.
- **Event(EventId, eventName, date, location, description, organizerId, volunteerIdsList)**: Captures the details of activities hosted by NGOs, including information about time, place, and volunteer needs.
- **DonationRequest(RequestId, date, details, type)**: Represents requests by NGOs for funds from donors.

4. Beneficiary Management

- **Beneficiary(id, name, aidRequests, walletId,postalNumber, legalDocuments)**: Represents individuals or groups who receive aid from donors and NGOs.
- **AidRequest(SubmissionDate, details, type, RequesterId, proofDocuments, approvalStatus)**: Formalizes requests for aid submitted by beneficiaries, tracking the status and approval process.
- **DigitalWallet(walletId, balance, transactionHistory)**: Provides a financial account for beneficiaries to receive, store, and use donations.
- **Posts(PostId, message, senderId, timestamp, status)**: shared by the beneficiary to request aid and help.

5. Admin Management

- **Admin(id, boardMemberName, accessLevel, verificationQueue)**: Represents platform administrators responsible for overseeing platform operations, ensuring all activities are running smoothly.

DDD/BPMN

1. Ubiquitous Language:

The following terms from the shared language used by all stakeholders to ensure a consistent understanding of the domain:

- **Donor**: A person or organization providing financial aid or resources.
- **Beneficiary**: An individual or family seeking or receiving financial or material support.
- **NGO**: A non-governmental organization that coordinates aid distribution and events.
- **Volunteer**: A person who offers their time to assist with events, aid distribution, or other activities.
- **Board,Admin,Administrator**: The administrative entity overseeing platform operations, verifying profiles, and managing high-level activities.
- **Profile Verification**: The process of confirming the legitimacy of users (donors, NGOs, volunteers, beneficiaries) on the platform.
- **Digital Wallet:** A secure, digital account where beneficiaries receive and manage their aid.
- **Event,Project:** An organized activity (workshop, health camp) facilitated by an NGO for community benefit.
- **Aid Request:** A request made by a beneficiary outlining specific needs for which support is required forwarded to the NGO by the Admin.
- **Request,DonationRequest,VolunteerRequest,EventRequest:** all reflect the requests submitted by the NGO that need admin approval to pass.
- **Post:** beneficiary request that doesn;t need to get approved and can be accessed by the donors to help the beneficiary directly.
- **Notification System:** A feature that informs users about updates, donations, approvals, or event details.
- **Blockchain:** A decentralized system used for tracking and verifying the transparent flow of donations, ensuring accountability.
- **Financial Institution:** Partners involved in handling digital transactions and providing wallet services to beneficiaries.

2. Bounded Contexts:

The platform is divided into distinct subsystems, each with its own domain model and associated responsibilities:

1. Donor Context:
    - **Focus**: Managing donors, their profiles, and their donations
    - **Key Components**: Donor Registration, Donation Processing, Donation Tracking
    - **Interactions**: This context interacts with the NGO/beneficiary context to donate funds and track donation usage.
2. NGO Context:
    - **Focus**: Managing NGO activities such as event organization, donation requests, and volunteer coordination.
    - **Key Components**: NGO Registration, Event Creation, Donation Requesting, Volunteer Management.
    - **Interactions:** It interacts with the donor/beneficiary context to receive/make donations and with the volunteer context for managing volunteers.
3. Volunteer Context:
    - **Focus:** Managing volunteers, their registration, activities.
    - **Key Components:** Volunteer Registration, Event Participation
    - **Interactions:** This context interacts with the NGO context for event scheduling and volunteer assignment.
4. Beneficiary Context:
    - **Focus:** Managing beneficiary requests, aid distribution.
    - **Key Components:** Beneficiary Registration, Aid Requests, Digital Wallet Management.
    - **Interactions:** Interacts with the NGO context for aid distribution, with the financial institution context for wallet transactions, and the admin context for verifying requests.
5. Administration Context:
    - **Focus:** Platform-wide management, overseeing donations, aid distribution, user activities, and verification processes.
    - **Key Components:** User Management, Request Verification, Donation and Transaction Tracking, Report Generation.
    - **Interactions:** This context interacts with all other contexts to ensure platform integrity and oversight.
6. Financial Institutions Context:
    - **Focus:** Managing and tracking transactions/donations
    - **Key Components:** Blockchain Ledger, Donation Logging, Transaction Verification, Digital Wallet, Bank servers
    - **Interactions:** Integrates with donor, NGO, and beneficiary contexts to verify and record donations and aid distributions.

3. Entities, Value Objects, and Aggregates:

Entities:

- Donor:
  - **Attributes**: `id, name, email, legalDocuments, verifiedStatus, dashboardData, walletBalance`
  - **Behavior**: `Register, donate, track donations, browse Posts.`
- Beneficiary:
  - **Attributes**: `id, name, legalDocuments, aidRequests, walletBalance,postalNumber.`
  - **Behavior**: `Register, request aid, create Posts, manage received funds.`
- NGO:
  - **Attributes**: `id, organizationName, NGOtype, eventList, donationRequests`
  - **Behavior**: `Register, organize events, request donations, manage volunteers.`
- Volunteer:
  - **Attributes**: `id, name, appliedEvents, pastEvents, segment`
  - **Behavior:** `Register, browse events, apply for participation, receive notifications.`
- Board:
  - **Attributes**: `id, boardMemberName, accessLevel, verificationQueue`
  - **Behavior:** `Verify user and NGOs profiles, approve funding requests, monitor donations, create segments.`
- EventDetails:
  - **Attributes**: `EventId, eventName, date, location, description, organizerId, volunteerIds`
  - Provides information about an event without an identity.
- Digital Wallet:
  - **Attributes**: `walletId, balance, transactionHistory.`
  - Provides functionality like add funds, withdraw funds, transfer funds to beneficiaries.
- Transaction:
  - Attributes: `sender, receiver, amount, transactionId, type, timestamp, PaymentMethod`
  - Records any financial movements like donations and spendings by beneficiaries.
- AidRequest:
  - Attributes: `AidId, SubmissionDate, details, type, RequesterId, proofDocuments, approvalStatus.`
  - Represents the aid requests submitted by the beneficiary and approved by the admin then linked to the NGOs.

- Post:
    - Attributes: `PostId, interactionCount, message, senderId, timestamp, status.`
    - shared by the beneficiary to request aid and help in the form of posts.

Value Objects:

- Notification:
    - **Attributes**: `message, senderId, recipientId, timestamp, status`
    - Represents a communication object that can be sent to any entity.
- Request:
    - **Attributes**: `date, details, type(eventRequest, volunteerRequest, DonationRequest), RequesterId`
    - It represents any request object, such as NGOs creating events, volunteers requesting to participate in such events, and donation requests applying for financial aid.
- PaymentMethod:
    - Attributes: `lastDigits, type(bankTransfer, creditCard, instantPayments)`
    - Immutable and defines the details of the payment method for a donation.

Aggregates:

- Donor Aggregate:
    - **Root Entity:** `Donor`
    - **Includes**: `Transaction objects, notification list.`
- Beneficiary Aggregate:
    - **Root Entity:** `Beneficiary`
    - **Includes:** `Aid requests, digital wallet details, Posts`
- NGO Aggregate:
    - **Root Entity:** `NGO`
    - **Includes**: `Event details, volunteer applications, and donation requests.`
- Volunteer Aggregate:
    - **Root Entity:** `Volunteer`
    - **Includes:** `List of applied events, participation history and event volunteering request`
- Board Aggregate:
    - **Root Entity:** `Board`
    - **Includes:** `Verification of accounts and aid requests, NGO interactions/requests, donations, events, link Aid Requests with the NGO`

4. Domain Services:

Domain services encapsulate operations that do not fit within a single entity:

- Verification Service:
  - Verifies the authenticity of users and NGOs by processing legal documents and managing verification queues.
- Donation Processing Service:
  - Handles secure financial transactions, updates donor and beneficiary accounts, and issues receipts.
- Event Management Service:
  - Coordinates event creation, volunteer assignment, and sends reminders/notifications to participants.
- Notification Service:
  - Manages the sending and tracking of notifications across different entities.
- Transaction Management Service
  - Admin can manage transactions and track them

5. Domain Events:

These represent key occurrences that signal state changes within the system:

- UserRegisteredEvent:
  - Triggered when a new user (donor, beneficiary, NGO, volunteer) registers on the platform.
- ProfileVerifiedEvent:
  - Triggered when a user's or NGOs' profile is successfully verified by the board.
- DonationMadeEvent:
  - Captured when a donor completes a contribution to a beneficiary or NGO.
- FundingRequestSubmittedEvent:
  - Indicates a new funding request has been posted by a beneficiary.
- FundingResponseEvent:
  - Indicates acceptance or rejection of the funding request
- AidDisbursedEvent:
  - Marks the successful transfer of funds to a beneficiary's digital wallet.
- EventAppliedEvent:
  - Occurs when a volunteer applies for an NGO event.
- EventOrganizedEvent:
  - Indicates that an NGO has successfully scheduled and published an event.

6. Adapting Architectural Patterns

The integration of the domain model into the software design influences the choice of architectural patterns that best suit the system's needs.

Microservices Architecture: Given the large scope of the domain, microservices could be used to implement the distinct bounded contexts (Donor, NGO, Volunteer, Beneficiary). Each microservice would encapsulate a bounded context and expose APIs for other services to interact with. Each microservice would be independent, ensuring scalability, fault tolerance, and flexibility.

Event-Driven Architecture: Since domain events play a crucial role in the system, an an event-driven approach is suitable. Here, events trigger asynchronous communication between different components, ensuring loose coupling and scalability. Technologies like Kafka, RabbitMQ, or Redis Pub/Sub can be used for event handling.

7. Integrating the Domain Model into Software Design & Architecture

1. Designing Aggregates and Entities:

Aggregates and entities are translated into classes or objects within the software. They must reflect attributes, operations, and relationships, ensuring consistency and alignment with the problem domain.

- **Donor Class Design:**

```
public class Donor {

   private final int id;

   private String name;

   private String email;

   private List<Transaction> donations;

   private File legalDocuments[];

   private bool verifiedStatus;

   private Dashboard dashboardData;

   private double walletBalance;

Methods: register(), makeDonation(), trackDonations(), browsePosts()

}
```

- **Beneficiary Class Design:**

```
public class Beneficiary {

   private final int id;

   private String name;

   private DigitalWallet wallet;

   private List<AidRequest> aidRequests;

   private double walletBalance;

   private File legalDocuments[];
   private int postalNumber;

Methods: register(), verify(), submitFundingRequest(), createPost(),
manageReceivedFunds()

}
```

- **NGO Class Design:**

```
public class NGO {

   private final int id;

   Private NGOType type;

   private String organizationName;

   private List<EventDetails> eventList;

   private List<Transaction> donationRequests;

 Methods: register(), organizeEvent(), requestDonation(), makeDonation()

}
```

- **Volunteer Class Design:**

```
public class Volunteer {

   private final int id;

   private String name;

   private int segment;

   private List<EventDetails> appliedEvents;

   private List<EventDetails> pastEvents;

   Methods: register(), verify(), applyForEvent(), receiveNotification()

}
```

- **Admin Class Design:**

```
public class Admin {

   private final int id;

   private final String boardMemberName;

   private String accesslevel;

   private Queue<Request> verificationQueue;

Methods: verify(), approveRequest(), MonitorDonation(), CreateSegments(),
   linkAidRequest()

   }
```

- **EventDetails Class Design:**

```
public class EventDetails {

   private final int EventId;

   private final String eventName;

   private LocalDate date;

   private String location;

   private final String description;

   private final NGO organizer;

   private Volunteer volunteers[];

}
```

- **Digital Wallet:**

```
public class Wallet {

   private final int walletId;

   private double balance;

   private String transationHistory[];

}
```

- **AidRequest:**

```
public class AidRequest {

  private int aidId;

  private LocalDate submissionDate;

  private String details;

  private String type;

  private int requesterId;

  private String[] proofDocuments;

  private String approvalStatus;

}
```

- **Transaction:**

```
public class Transaction{

   private final int sender;

   private final int receiver;

   private final double amount;

   private final int transactionId;

   private final String type;

   private final Localdate timestamp;

   Private final PaymentMethod PaymentMethod;

}
```

- **Post:**

```
public class Post {

  private int postId;

  private int interactionCount;

  private String message;

  private int senderId;

  private LocalDate timestamp;

  private String status;

}
```

3. Incorporating Value Objects:

Value objects are integrated into the design as immutable classes that encapsulate attributes and operations relevant to their role in the domain.

- **Notification Class Design:**

```
public class Notification {

   private final String message;

   private final LocalDate timestamp;

   private final String senderId;

   private final String recipientId;

   private boolean status;

}
```

- **Request:**

```
public class Request {

   private LocalDate timestamp;

   private String details;

   private final String type;

   private final String RequesterId;

}
```

- **PaymentMethod:**

```
public class PaymentMethod {

   private final String lastDigits;

   private final String type;

}
```

4. Implementing Domain Services:

Domain services encapsulate operations that do not belong to a single entity or aggregate. These services are implemented as stateless modules or classes.

- **VerificationService Implementation**:

```
public class VerificationService {

   public boolean verifyProfile(User user) {   }

}
```

- **DonationProcessingService Implementation:**

```
public class DonationProcessingService {

public Receipt processDonation(Sender sender, Receiver receiver, double
   amount) {    }

}
```

Note: Sender and Receiver are two interfaces. Sender will be implemented by Donor and NGO, while Receiver will be implemented by NGO and Beneficiary

- **EventManagementService Implementation:**

```
public class EventManagementService {

   public void organizeEvent(NGO ngo, EventDetails event) {

   }

}
```

- **TransactionManagementService Implementation:**

```
public class TransactionManagementService {

   public void manageTransactions() {

   }

}
```

5. Handling Domain Events:

Domain events facilitate communication and interaction between components, maintaining system flexibility and responsiveness.

- **UserRegisteredEvent Implementation:**

```
public class UserRegisteredEvent {

   private final String userId;

   private final LocalDateTime timestamp;

   private final String userType;

   //Constructor and methods to trigger event handling logic

}
```

- **ProfileVerifiedEvent Handling:**

```
   public class ProfileVerifiedEvent {

   private final String userId;

   private final LocalDateTime verificationTime;

   //Constructor and event logic

}
```

- **DonationMadeEvent Integration:**

```
public class DonationMadeEvent {

   private final Sender senderId;

   private final Receiver receiverId;

   private final double amount;

   //Constructor and event dispatch logic

}
```

- **FundingRequestSubmittedEvent**

```
public class FundingRequestSubmittedEvent {

    private final String requestId;

    private final String beneficiaryId;

    private final double requestedAmount;

    private final LocalDateTime requestTime;

}
```

- **FundingResponseEvent**

```
public class FundingResponseEvent {

    private final String requestId;

    private final String responderId;

    private final boolean status;

    private final LocalDateTime responseTime;

    private final String reason;

}
```

- **AidDisbursedEvent**

```
public class AidDisbursedEvent {

    private final String transactionId;

    private final String beneficiaryId;

    private final double disbursedAmount;

    private final LocalDateTime disbursementTime;

}
```

- **EventAppliedEvent**

```
public class EventAppliedEvent {

    private final String eventId;

    private final String volunteerId;

    private final LocalDateTime applicationTime;

}
```

- **EventOrganizedEvent**

```
public class EventOrganizedEvent {

    private final String eventId;

    private final String ngoId;

    private final String eventDetails;

    private final LocalDateTime organizationTime;

}
```

8. Domain Model Interaction with Software Architecture

The domain model interacts with other components of the software architecture, ensuring seamless communication and functionality across the platform.

1. User Interface (UI)

The UI Layer acts as the entry point for users to interact with the domain model. It sends user inputs to the domain model and displays results from domain services.

- Donor Context:
    - User actions like registration, donation, and tracking trigger domain events such as `UserRegisteredEvent` and `DonationMadeEvent`.
    - UI fetches updated donation statuses and notifications via the Notification Service.
- NGO Context:
    - UI facilitates event creation, donation requests, and volunteer management using commands routed to domain services like the Event Management Service.
- Beneficiary Context:
    - UI integrates with Digital Wallet and Aid Requests for managing and viewing transactions.

2. Persistence Layer

The domain aggregates are mapped to database entities for consistent storage and retrieval.

- Donor Aggregate: Stores donor data, transaction history, and notifications.
- NGO Aggregate: Persists event details, volunteer applications, and donation requests.
- Beneficiary Aggregate: Manages aid requests, wallet balances, and posts.
- Transaction Management Service ensures accurate logging and querying of donation records via the Financial Institutions Context.

3. External Systems

- Financial Institutions Context:
    - Integrates blockchain ledgers and bank APIs for secure and verified transactions, triggered by events like DonationMadeEvent and AidDisbursedEvent.
    - Domain services such as the Transaction Management Service ensure compliance with financial regulations.
- Notification Service:
    - Sends email, SMS, or app notifications using external messaging systems when events like FundingRequestSubmittedEvent occur.

9. Benefits and Challenges

Challenges	Benefits
Understanding Boundaries and Aggregates: One of the primary challenges in DDD was identifying clear boundaries for each domain and properly structuring aggregates. This requires a deep understanding of the business logic and domain intricacies to avoid overly complex or tightly coupled aggregates, which can impact performance and scalability.	**Enhanced Communication and Clarity:** DDD encourages a shared language across technical and business teams. By modeling the domain closely to real-world terms, communication improved among team members, fostering a deeper understanding of requirements and goals. All team members now understand the complex terminologies and how they contribute to our system. The services, events, objects were named in a way that matches their functionality and real-world scenario.
Communication Across Teams: DDD requires research about the domain to accurately capture the domain language and logic. This process can be challenging, especially when trying to bridge the gap between technical language and business language to ensure alignment across the development team. For example, we had to define what donor actually is, who a beneficiary represents, what the role of NGO is in our system and how volunteers are represented in our system. These were challenging as in each domain these terms might represent different things.	**Better Organization of Code and Responsibilities:** DDD's structure encouraged the separation of concerns, reducing dependency issues and simplifying the codebase. This structure allowed each team member to focus on specific areas without worrying about unintended side effects in unrelated parts of the application. By creating objects, values and incorporating them in aggregates we were able to separate the classes and structure them well.

Milestone 2

Architectural Styles and Justification

Styles and Justification of Architecture

Chosen Style: Microservices with Event-Driven Architecture

To implement all these functional and non-functional requirements of the platform, we combined a hybrid approach between Microservices and Event-Driven Architecture. With this in place, the system can be designed to be modular and scalable with the responsiveness that any user expects while performance and reliability remain uncompromised.

Alignment with Functional Requirements (FRs)

1. **Donor Management:**
   Microservices enable a donor management service independently for user sign-up, legal verification, donation tracking, and dashboards. Event-driven architecture provides the ability for real-time notifications, such as when a donor is informed about his contribution usage or automatically gets thank-you messages the moment any transaction happens.
2. **Volunteer Management:**
   It enables volunteers to sign up, explore events, and be notified about their schedules. The volunteer management is integrated in the event management service. This also means that updates and maintenance related to events and volunteers can be done without affecting other services. The event-driven model facilitates the easy sending of notifications in real time, such as reminders of upcoming events or volunteer assignments.
3. **Management of NGOs:**
   Similarly, that same service for events will handle all NGO related functionality to be able to handle event organization, requests for donation, and volunteer management for the NGOs. In the same way, event-driven triggers should intimate the concerned stakeholder—for example, NGOs at the time donors make their donations, and similarly volunteers at the time of approval of events.
4. **Beneficiary Management:**
   Donation Processing Service ensures the Beneficiaries can send in their requests and get in touch with the donor or NGOs within the stipulated time as well as the management of their financial accounts handled . Events ease this process by sending notifications when there is a match for donation or aid requests.

5. **Administrative Access and Control:**
   The board of the platform will be able to view all the data of the system and manage the operations through the centralized services. Event-driven updates allow for real-time tracking of donated resources and requests for aid, thus efficient decision-making.

Conformance to Non-Functional Requirements (NFRs)

1. **Scalability:**
   - **Contribution of Microservices:** Each service will scale independently to handle fluctuating demands. For example, if there are a lot of donations, the donation service will scale, but others will not be affected.
   - **Event-Driven Contribution:** Such is the architecture of asynchronous communication, which allows events—including those regarding notifications of donations—not to burden the system and hence scale up to 30 million active users.
2. **Performance:**
   - **Contribution of Microservices:** Isolated services guarantee very fast responses for high-priority functionalities, like loading a dashboard or processing an aid request.
   - **Event-driven contribution:** The updates are in real time, demanding a very responsive system. To clarify this, a donation made will change the dashboards in less than a second, and notifications appear to NGOs in less than 0.1 seconds from the moment of the event.
3. **Reliability:**
   - **Microservices Contribution:** In this, service isolation ensures that bugs in one area, say Donation management, do not affect the event or verification functionalities.
   - **Event-driven contribution:** Events are buffered and processed, even in cases of temporary services unavailability, thus ensuring consistent operations without any loss of data.
4. **Safety:**
   - **Contribution by Microservices:** The sensitive services, like donation management, are based on secure protocols of encryption and role-based access control.
   - **Event-driven contribution:** It assures total transparency and tamper-proof tracking in donations and distributions due to the integration with blockchain through events.

5. **Transparency:**
    - **Microservices Contribution:** Modular design allows precise tracking and verification of financial transactions and aid distributions.
    - **Event-Driven Contribution:** Events and donations log every financial and operational activity, ensuring verifiable transparency for all stakeholders.
6. **User Experience:**
    - **Contribution of Microservices:** Independent deployment updating single functionalities, such as the Arabic interface or dashboards, promotes better usability.
    - **Event-driven Contribution:** It enhances the experience of each user in real time, whether a volunteer when details about an event are sent or a donor does when updates about the contribution are available.

Other Architectures Considered and Their Trade-offs

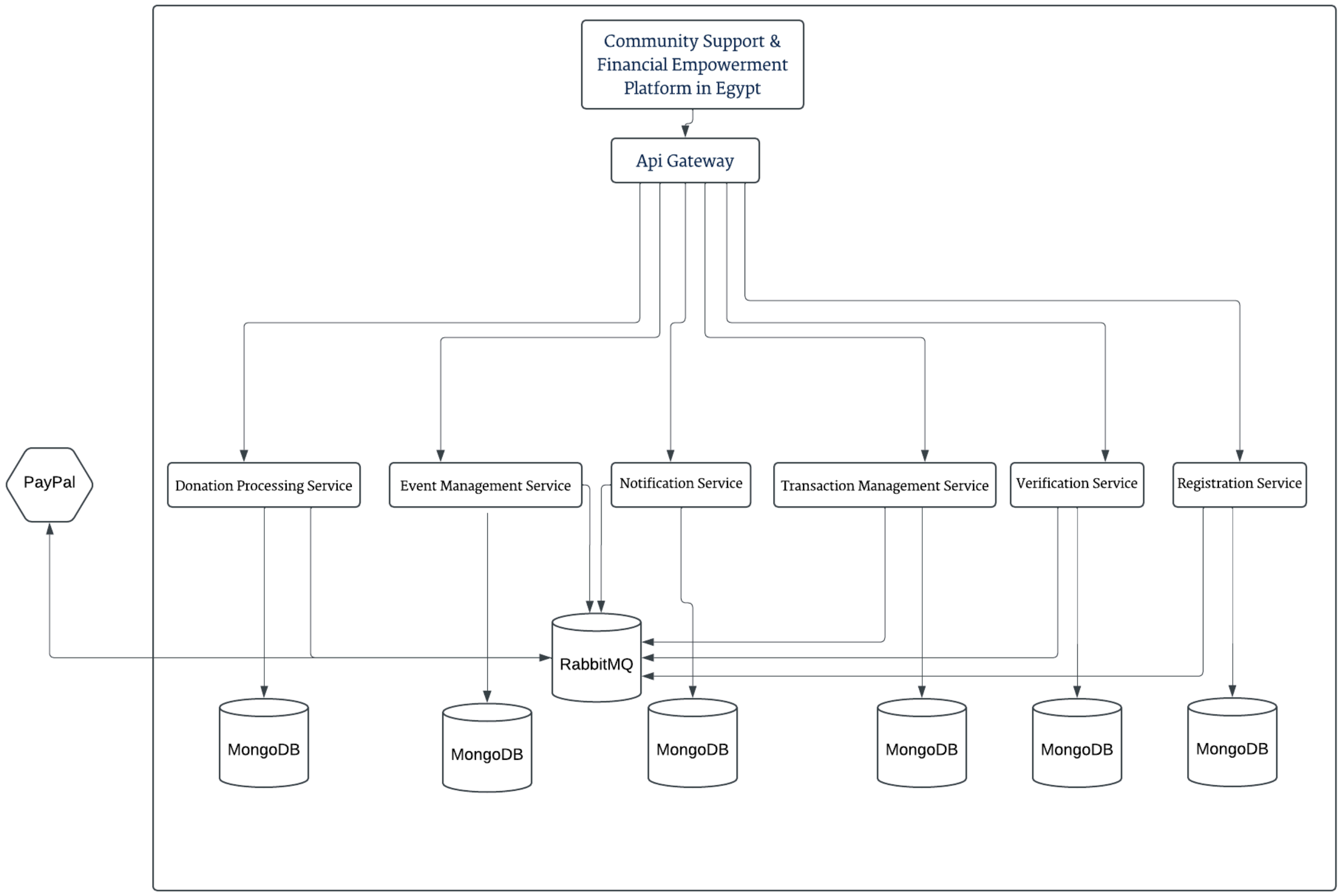
1. **Layered Architecture:**
    - **Pros:** Easy to implement and maintain, as it clearly separates the concerns (UI, business logic, and data layers).
    - **Trade-offs:** Less flexible scaling because the entire layer stack has to be scaled for even localized performance demands. As an example, this means that scaling up donor tracking would require scaling other unrelated functionalities, which would lead to inefficiencies.
2. **Monolithic Architecture:**
    - **Pros:** Easier to construct and deploy at first, especially for smaller systems.
    - **Trade-offs:** A tightly coupled monolithic system does not suit the scale and complexity of our platform. Any update requires redeployment of the system, which means downtime and slows down the pace of development. For example, an update of event management stops donation tracking and beneficiary aid processing.
3. **Pure Event-Driven Architecture:**
    - **Advantages:** Great for handling both real-time and asynchronous jobs, such as sending notifications or managing volunteer schedules.
    - **Trade-offs:** Purely event-driven systems are more difficult to debug and manage due to their decentralized nature. Further, tasks requiring strict workflows, such as donor verification, are better supported with structured interactions afforded by microservices.

4. **Service-Oriented Architecture (SOA):**
    - **Advantages:** It enables reutilization and integration, especially in applications whose systems need shared services.
    - **Trade-offs:** This dependency on a single centralized service bus can be a bottleneck and a single point of failure under heavy loads. In other words, when there are too many donation events through the service bus, the delays will affect other functionalities such as NGO notifications. SOA also does not provide the modularity and autonomy that microservices can give, and hence it does not suit our scalability requirements. Also, a single database wouldn't be feasible in a system of that scale.

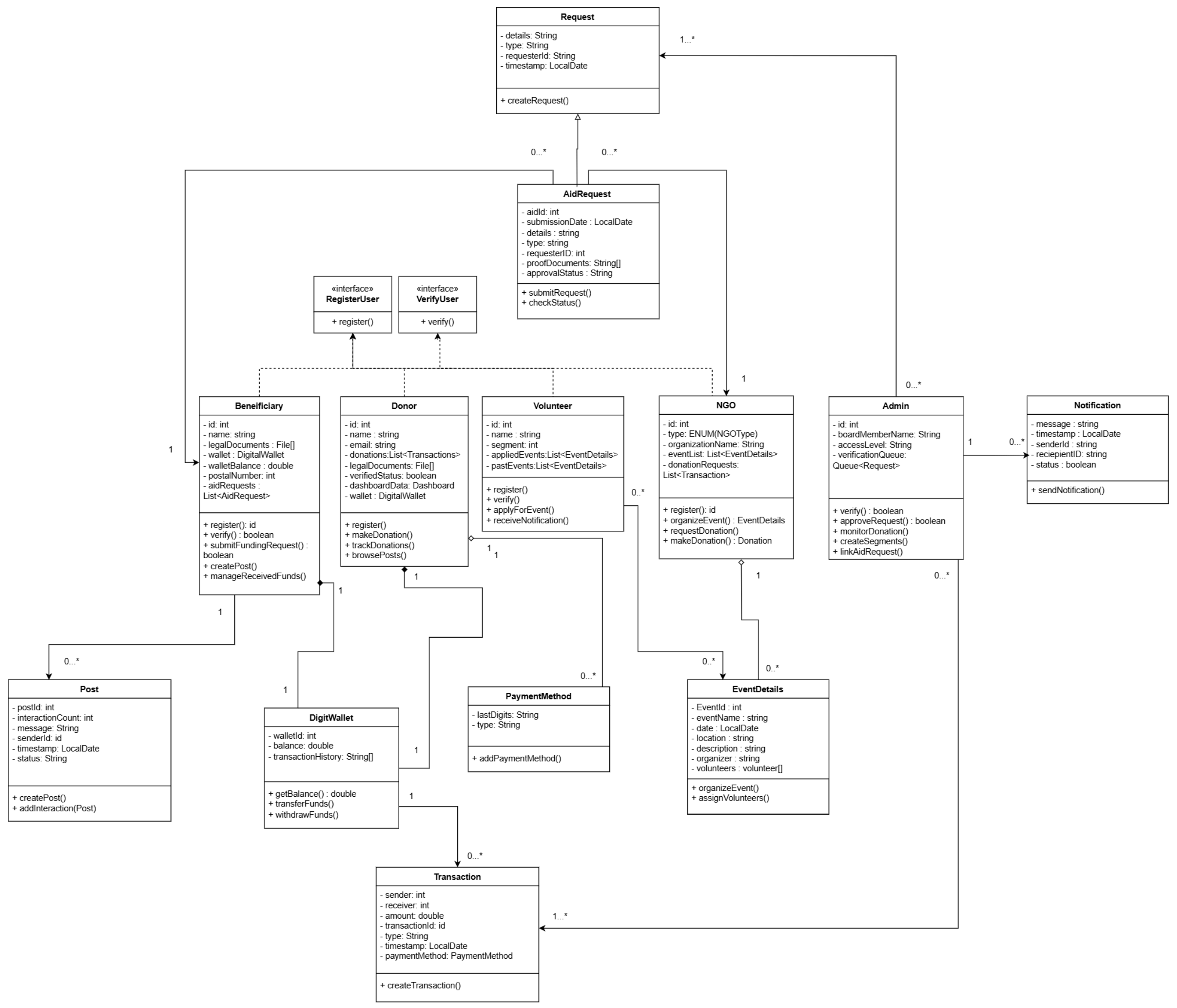
Why Microservices with Event-Driven Architecture Works Best for Us

This hybrid approach balances modularity, scalability, and responsiveness. Microservices ensure each functionality is independently deployable and scalable, while event-driven design facilitates seamless communication and real-time interactions across services. Together, they will create a robust, adaptable, user-friendly platform that can grow with our users' evolving needs. Whether it's keeping donors informed on how their funds are being used, or managing volunteer activities, or matching beneficiaries with NGOs, this architecture provides the performance, reliability, and flexibility to help us achieve our mission.

High-Level Architecture Diagram

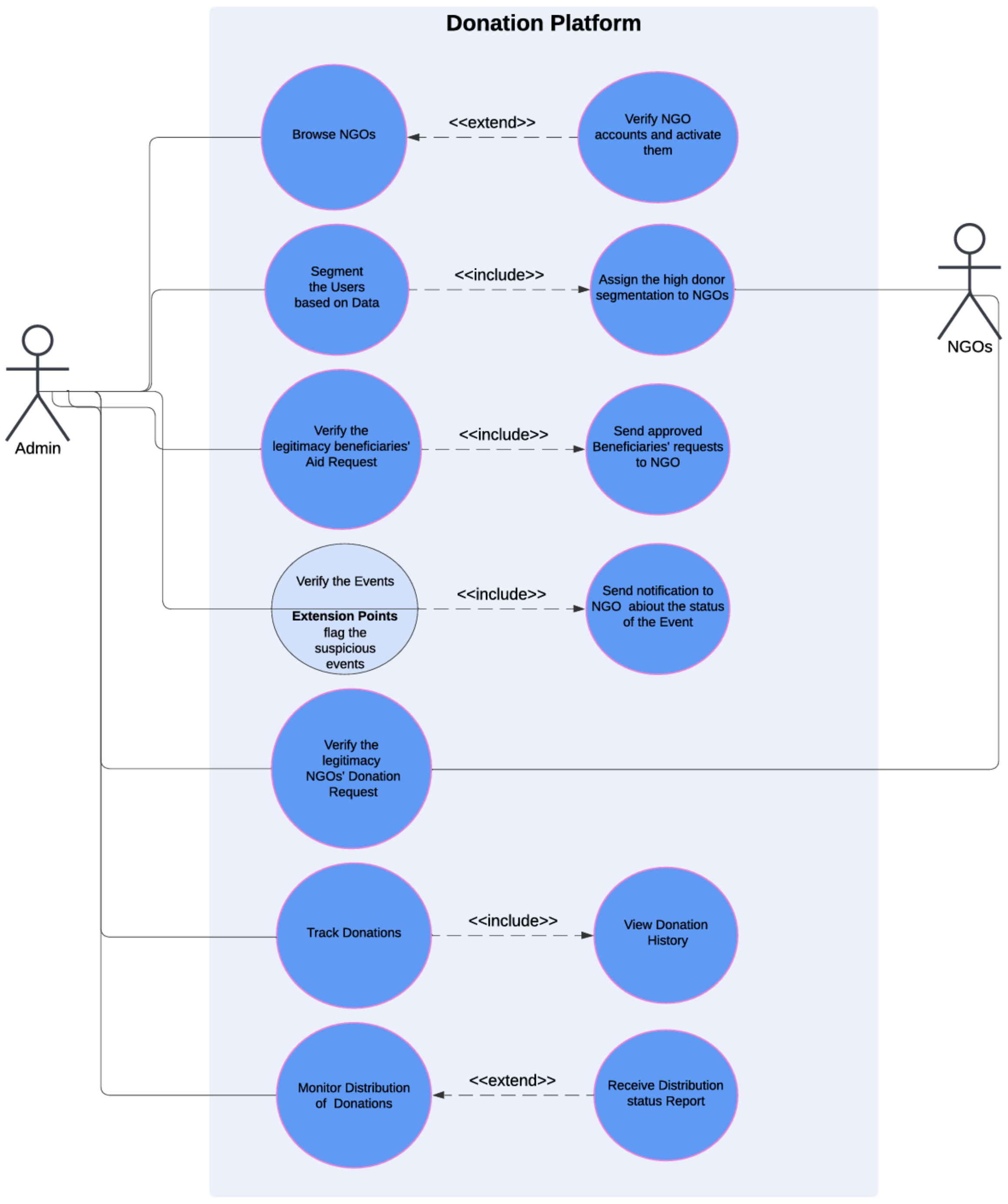


Class Diagram

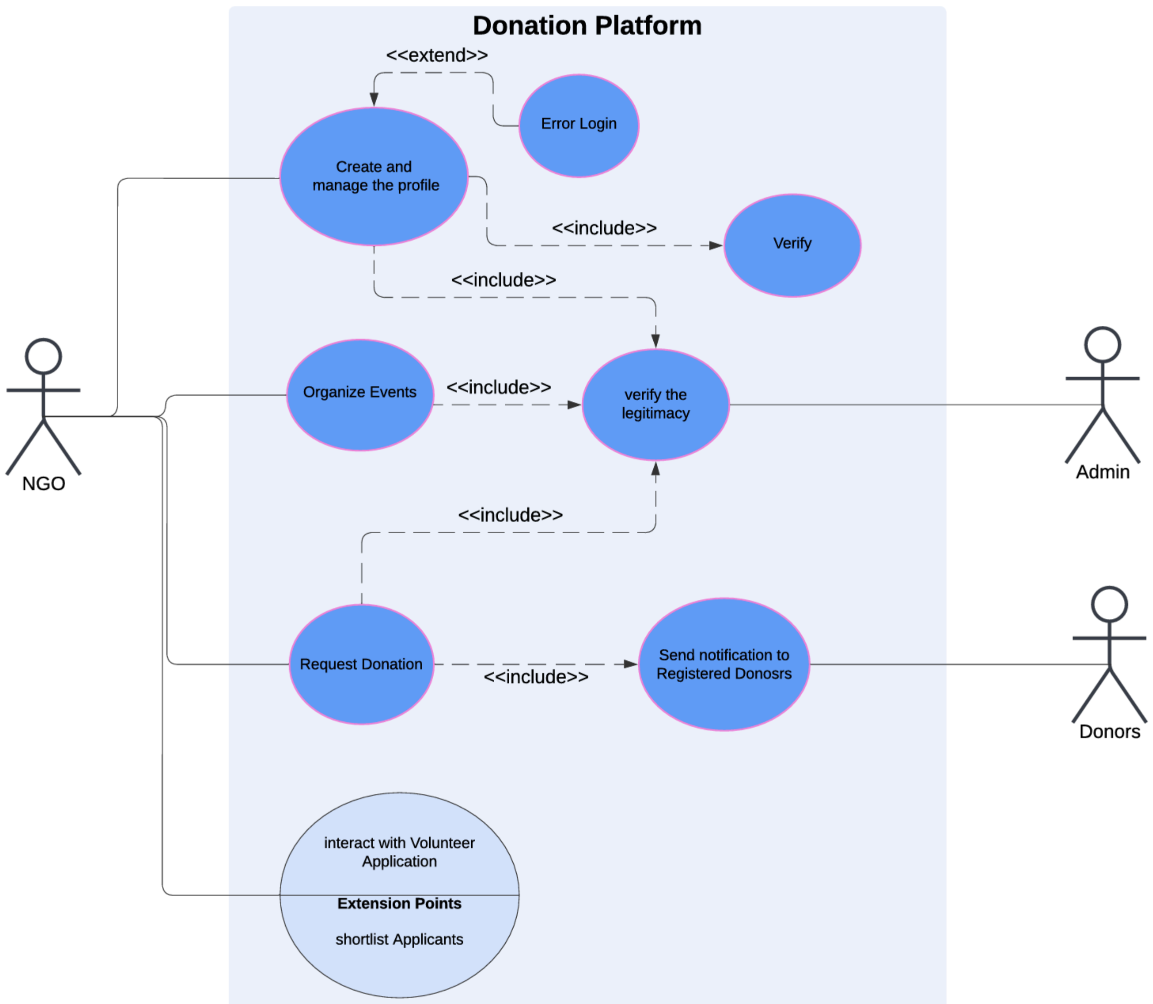


Use Case Diagram

Admin Use Case

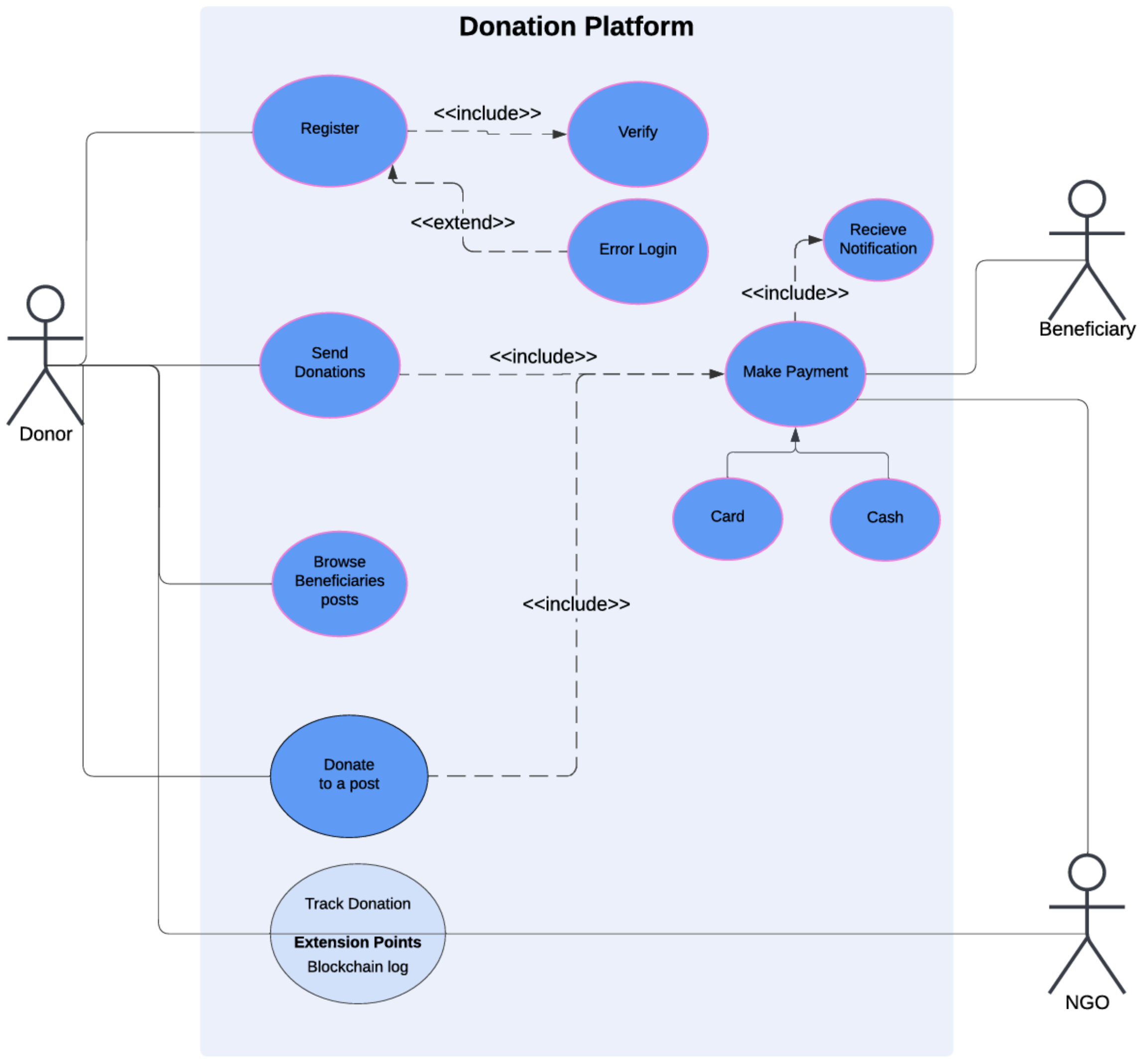

Donation Platform
Browse NGOs
<<extend>>
Verify NGO accounts and activate them
Segment the Users based on Data
<<include>>
Assign the high donor segmentation to NGOs
NGOs
Admin
Verify the legitimacy beneficiaries' Aid Request
<<include>>
Send approved Beneficiaries' requests to NGO
Verify the Events
Extension Points
flag the suspicious events
<<include>>
Send notification to NGO abiout the status of the Event
Verify the legitimacy NGOs' Donation Request
Track Donations
<<include>>
View Donation History
Monitor Distribution of Donations
<<extend>>
Receive Distribution status Report

NGO Use Case

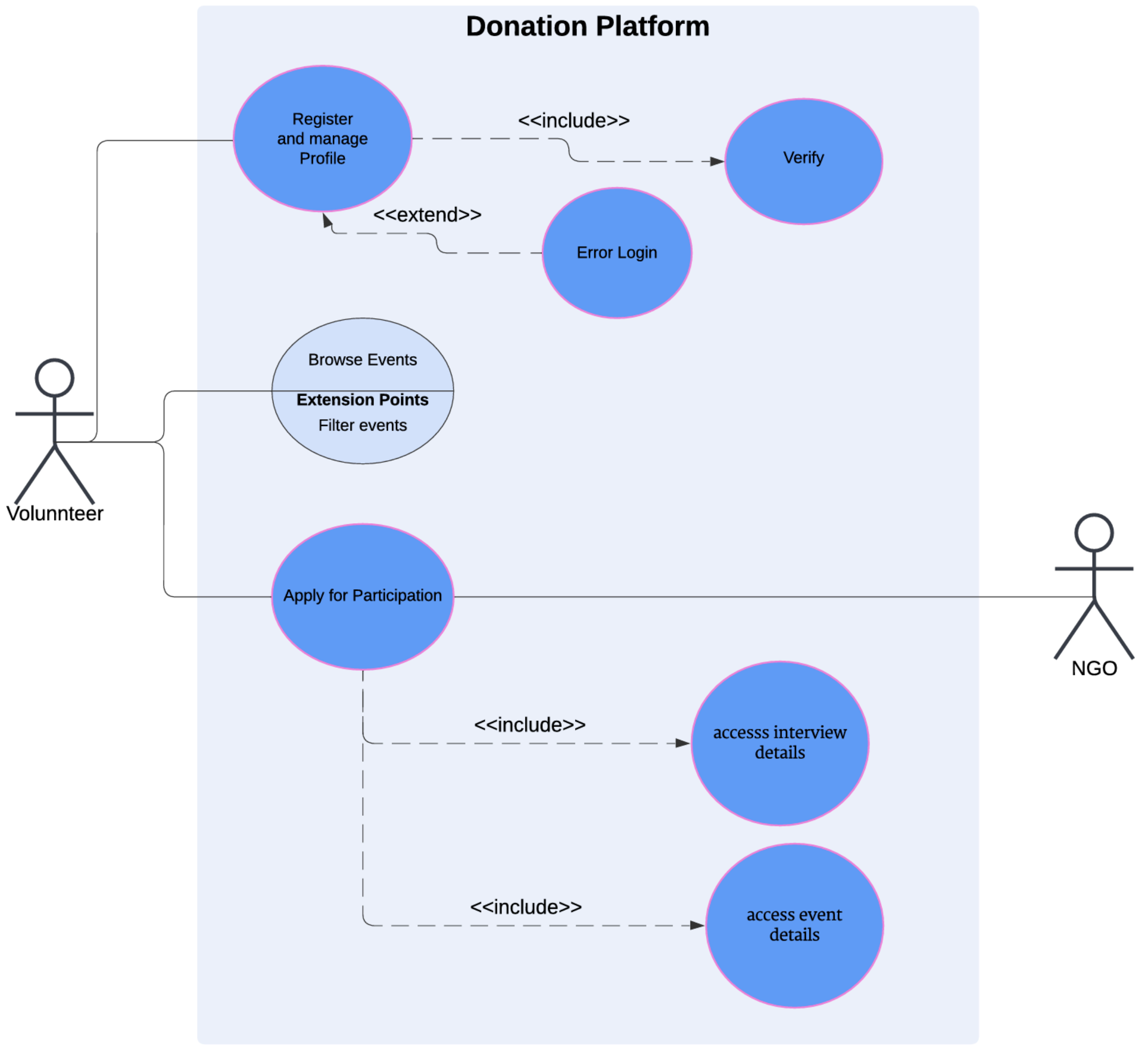


Donation Platform
<<extend>>
Error Login
Create and manage the profile
<<include>>
Verify
<<include>>
NGO
Organize Events
<<include>>
verify the legitimacy
Admin
<<include>>
Request Donation
<<include>>
Send notification to Registered Donosrs
Donors
interact with Volunteer Application
Extension Points
shortlist Applicants

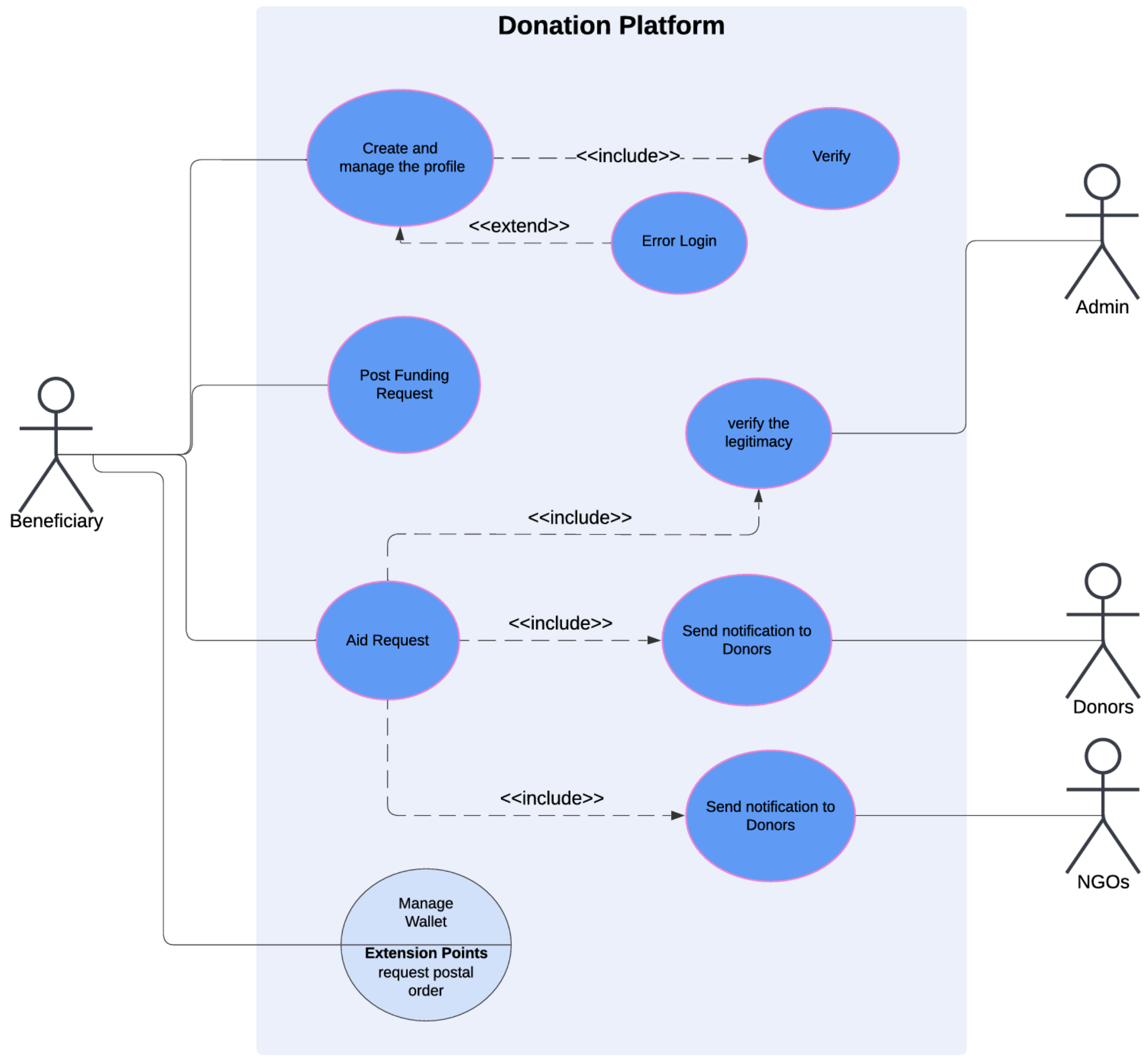
Donor Use Case



Volunteer Use Case



Beneficiary Use Case



Integration With Milestone 1

Conceptual Design

In the previous system, an event-driven architecture was not taken into account. However, when building the architecture and diagrams; it was evident that a message broker needed to be implemented. The reason being that the system is full of concurrent events that need to happen asynchronously. Notifications needed to be implemented asynchronously to ensure a fast and reliable experience. Moreover, logs of all transactions were needed to be stored in the Transaction Management Service to ensure history availability for Admin Tracking, Security, and for extension features (Think of an AI Model). Moreover, to ensure a smooth and fast Verification Process, the Registration Service needed a way of forwarding data to the Verification Service since such a process is costly and can not be done synchronously. For that reason, we integrated event-driven architecture into our existing microservice architectural design.

All the points discussed above ensure a highly scalable, fast, reliable and loosely coupled Microservice Event-Driven Architecture. Since all service messaging is handled using a message broker (RabbitMQ -Kafka is an alternative-), all services can function and scale independently of other services.

DDD

After diving into the details of the system and drawing a class diagram to show the relations and functions of the system; we saw that some adjustments were needed. Firstly, some methods were added in some of the classes like the Digital Wallet and the Transaction Classes. The reason being that in the previous DDD only the general picture was represented without diving deep into the details, and when implementing the DDD we realized that some relationships are not modeled correctly in our DDD. Class Diagram helps translate the DDD components that we did into a static overview of the system classes and objects, and when diving deep into the relationships, we realized some methods needed to be added to have a fully functional system. Moreover, drawing a class diagram made us define relationships, for example we needed to include a Wallet in the Donor to have a composition relationship with the Digital Wallet Class such that a Donor is able to use a Wallet.

Appendix

Below are the shareable links to our diagrams:

Architecture diagram:

https://lucid.app/lucidchart/7697bc20-d91b-461c-b61a-4b6498a96b04/edit?viewport_loc=-1833%2C-200%2C1761%2C1200%2C0_0&invitationId=inv_58580897-7c5b-4248-8a61-3b476d39eb3c

Class Diagram:

https://drive.google.com/file/d/18xSCode3DjBXkPwAqxHUxTsqf4eRa-KO/view?usp=sharing

Use Case Diagrams:

Admin Use Case
https://lucid.app/lucidchart/518a13d5-da81-4bb2-bfd3-81d3c51a8586/edit?invitationId=inv_d4a2bf7f-6ace-4f1c-89ba-10b249600bf9

NGO Use Case
https://lucid.app/lucidchart/36666089-b969-45f0-83d7-aad4916736c9/edit?invitationId=inv_aa6d549b-9782-4636-9cf1-0227f99fdb7b

Donor Use Case
https://lucid.app/lucidchart/a55d6f3a-8d11-4f94-a123-1edffa3e5ed1/edit?invitationId=inv_9a6a8e98-dbae-4b13-94dc-8dcc06f0f038

Volunteer Use Case
https://lucid.app/lucidchart/84dfc5e3-66ac-4c43-92ea-92e7a1bdd657/edit?invitationId=inv_230716ec-fd97-4d37-b577-9a452f4dbff3

Beneficiary Use Case
https://lucid.app/lucidchart/fdf18e5f-283d-41b0-9b86-84cb59effc72/edit?invitationId=inv_5135a164-7c76-49dc-b354-4b448255128a