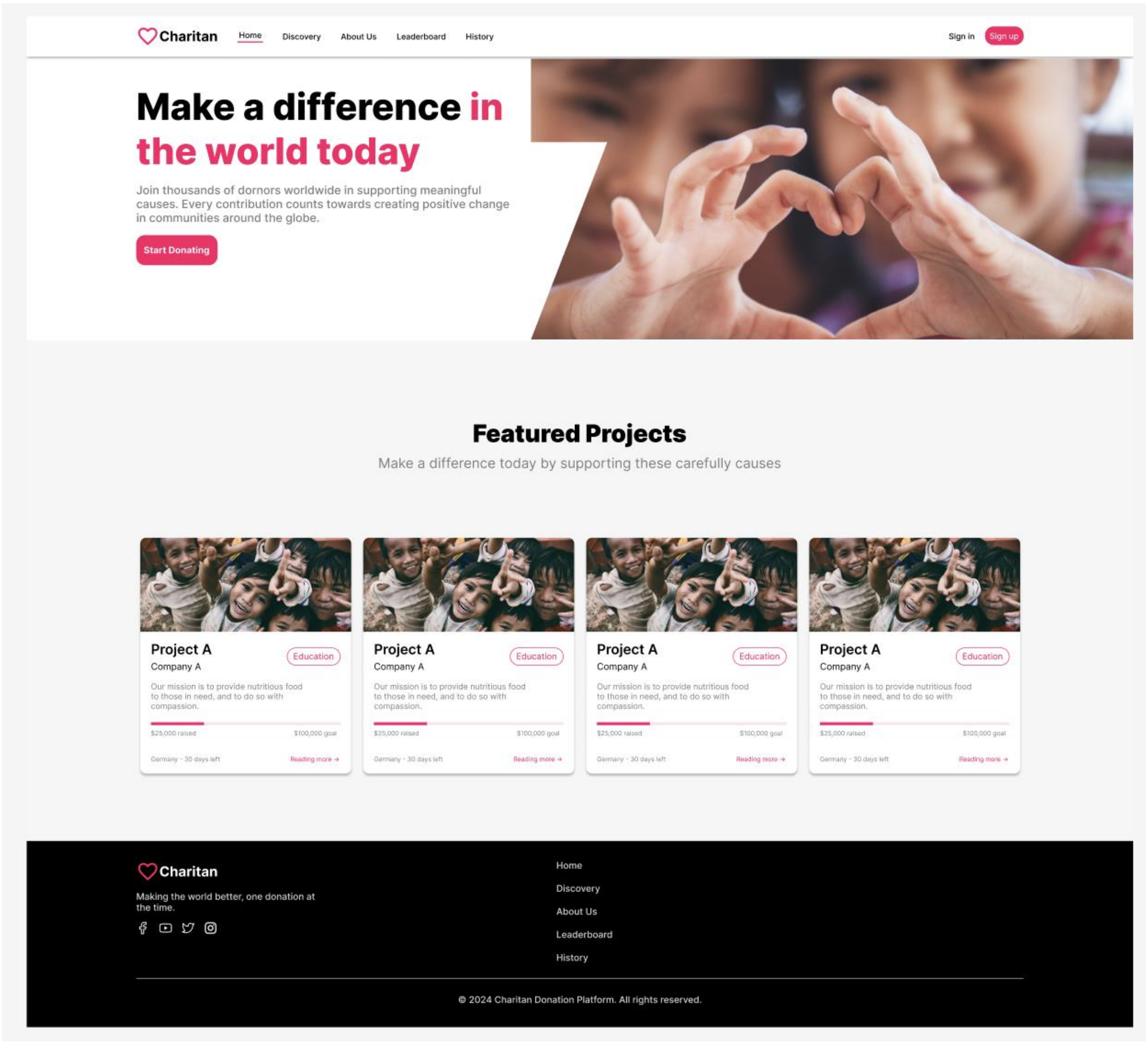
Charitan
A Global Charity Donation Platform
Milestone 2

Student Team:	Tiger – Team A
Student Name & ID:	Pham Thanh Nam - **s3878413**
	Sanghwa Jung - **s3768999**
	Tran Ngoc Minh - **s3911737**
	Nguyen Gia Khanh - **s3927238**
Instructor:	Tri Huynh

Table of Contents

I. Abstract



Make a difference in the world today

Join thousands of dornors worldwide in supporting meaningful causes. Every contribution counts towards creating positive change in communities around the globe.

Featured Projects

Make a difference today by supporting these carefully causes

Project A
Company A
Education
Our mission is to provide nutritious food to those in need, and to do so with compassion.
$25,000 raised $100,000 goal
Germany · 30 days left Reading more →

Project A
Company A
Education
Our mission is to provide nutritious food to those in need, and to do so with compassion.
$25,000 raised $100,000 goal
Germany · 30 days left Reading more →

Project A
Company A
Education
Our mission is to provide nutritious food to those in need, and to do so with compassion.
$25,000 raised $100,000 goal
Germany · 30 days left Reading more →

Project A
Company A
Education
Our mission is to provide nutritious food to those in need, and to do so with compassion.
$25,000 raised $100,000 goal
Germany · 30 days left Reading more →

Charitan
Making the world better, one donation at the time.

© 2024 Charitan Donation Platform. All rights reserved.

Figure 1: Charititan Home Page

This milestone report focuses on the foundational architectural and data modelling design for a comprehensive Charitan Donation Platform, a software solution designed to bridge the gap between donors, volunteers, and charitable organizations globally. For this initial project phase, our team has chosen to implement a Modular Monolith Architecture, a sophisticated approach that provides the benefits of a monolithic system while introducing improved modularity and separation of concerns.

The architectural design addresses six critical aspects: maintainability, extensibility, resilience, scalability, security, and performance. Our approach involves creating a detailed Entity Relationship Model that captures the complex interactions between donors, charities, projects, and administrative functions. The system architecture is meticulously documented using C4 container and component diagrams, clearly delineating the interactions between front-end, back-end, and external systems.

The Modular Monolith design organizes the application into distinct, bounded contexts or modules. This approach allows for a clear separation of concerns while maintaining the deployment simplicity of a monolithic

application. Each module will expose both external and internal APIs, ensuring controlled and intentional communication between different parts of the system.

Key design considerations include creating a flexible data model that can accommodate diverse charitable projects across different categories. The architectural blueprint supports future enhancements and provides a solid foundation for implementing features in the following milestone.

II. Conceptual Data Model for Database

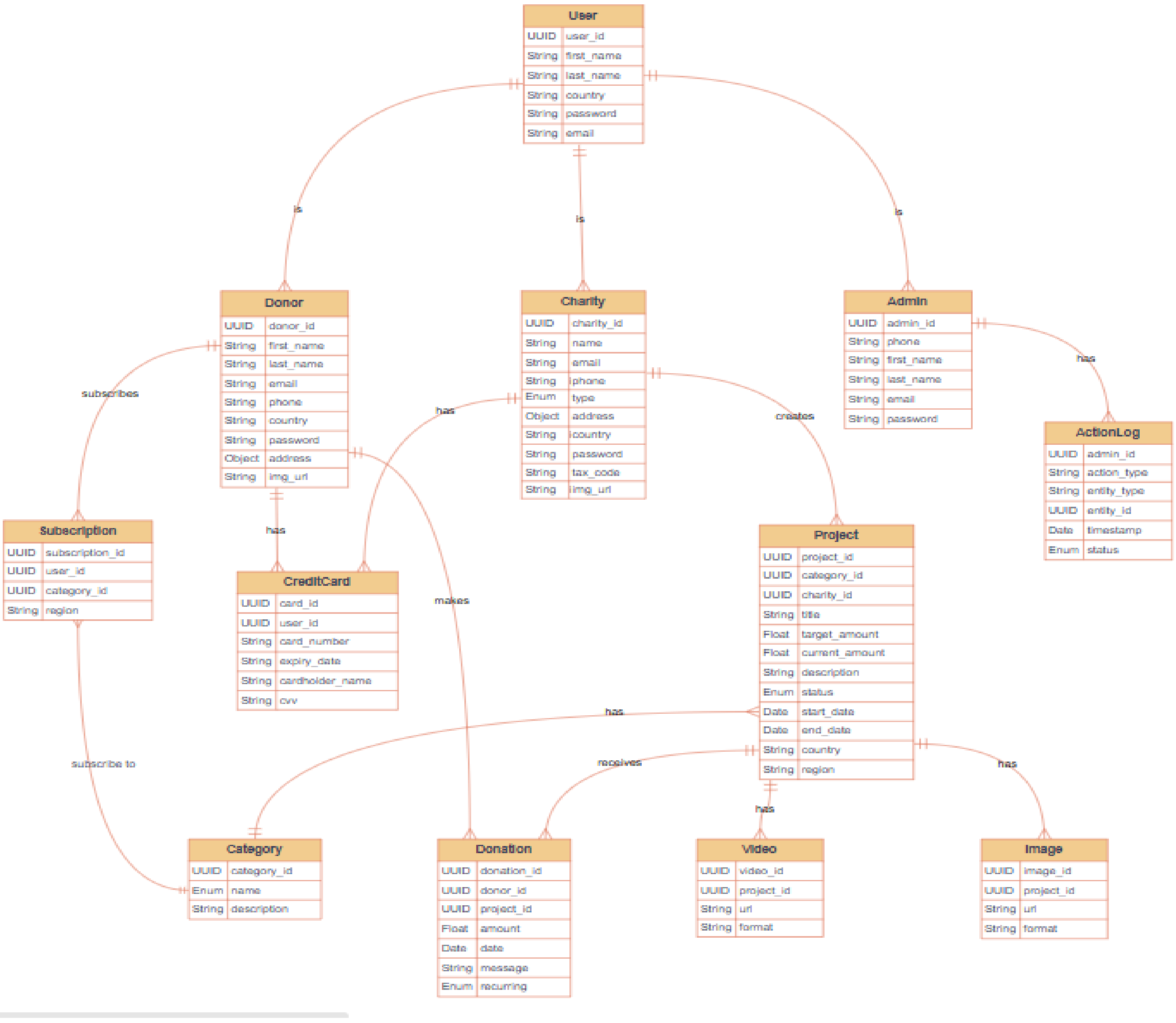


Figure 2: Entity-Relationship Diagram

The ER diagram of the Charitan Donation Platform identifies all the entities, their attributes, and relationships that will help the platform fulfill its functional requirements. Each entity, their attributes, their functions, types, and the relationship between entities are further elaborated in the following:

1. User
    - Attributes:
        - user_id (UUID, PK): A unique identifier for each user.
        - name (String): The name of the user.

        - email (String): The user's email address, used for login and communication.
        - password (String): The user's password, stored securely using encryption.
        - phone (string): The user's password
    - Function: An abstract class is served as a base entity for specialized roles.
    - Relationships:
        - One-to-Many with Donor, Charity, and Admin, indicating a user (abstract class) can specialize in one of these roles.

2. Donor
    - Attributes:
        - donor_id (UUID, PK): Donor's unique identifier.
        - fullName(String): Donor's full name.
        - first_name (String): The donor's first name.
        - last_name (String): The donor's last name.
        - email (String): The donor's email address used for login and communication.
        - password (String): The user's password, stored securely using encryption.
        - phone (String): The donor's phone.
        - country (String): The donor's country.
        - address (String): The donor's physical address.
        - img_url (String): The place to store donor's avatar picture.
    - Function: individuals who contribute to charity projects.
    - Relationships:
        - One-to-Many with Donation: one donor can make many donations.
        - One-to-Many with CardInfo: a donor can only have many card connected to an account.
3. Charity
    - Attributes:
        - charity_id (UUID, PK): A unique identifier for each charity.
        - name (String): The name of the charity organization.
        - email (String): The charity's email address used for login and communication.
        - phone (String): The charity's contact number.
        - type (Enum): The type of charity including individual, company, non-profit.
        - address (String): The charity's physical address.
        - country (String): The charity's physical address.
        - tax_code (String): The charity's tax identification number.
        - img_url (String): The place to store charity's avatar picture.
    - Function: organizations or individuals managing charity projects.
    - Relationships:
        - One-to-Many with Project indicate a charity can create multiple projects.
        - One-to-Many with CardInfo indicate a charity can have many many card number.
4. Admin
    - Attributes:
        - admin_id (UUID, PK): A unique identifier for each admin.
        - fullName(String): admin's full name.
        - first_name (String): The admin's first name.
        - last_name (String): The admin's last name.
        - email (String): The admin email address used for login and communication.
        - password (String): The admin password, stored securely using encryption.
        - phone (String): The admin phone.
    - Function: administrators responsible for Charitan platform oversight.
5. CardInfo
    - Attributes:
        - card_id (UUID, PK): A unique identifier for each card record.

        - user_id (UUID, FK): Foreign key linking to the User entity.
        - card_number (String): The credit card number, stored securely with encryption.
        - expiry_date (String): The expiration date of the card.
        - cardholder_name (String): The name of the cardholder.
        - cvv (String): The card verification value, encrypted for security.
    - Function: Stores credit card information for processing donations.
    - Relationships:
        - Linked to both Donor and Charity entities for storing payment details.

6. Project
    - Attributes:
        - project_id (UUID, PK): A unique identifier for each project.
        - category_id (UUID, FK): Foreign key linking to the Category entity.
        - charity_id (UUID, FK): Foreign key linking to the Charity entity.
        - title (String): The title of the project.
        - description (String): A detailed description of the project.
        - target_amount (Float): The fundraising goal for the project.
        - current_amount (Float): The current amount of the project.
        - status (Enum): The status of the project including active, completed, and halted.
        - start_date (Date): The start date of the project.
        - end_date (Date): The end date of the project.
        - country (String): The country where the project is based.
        - region (String): The continent where the project is based.
    - Function: charity projects created by charities.
    - Relationships:
        - One-to-Many with Donation, as projects can receive multiple donations.
        - One-to-Many with Video and Image indicate that a project can have many videos or images for a campaign.
        - Many-to-One with Category indicates that a project must have a category.
7. Donation
    - Attributes:
        - donation_id (UUID, PK): A unique identifier for each donation.
        - donor_id (UUID, FK): Foreign key linking to the Donor entity.
        - project_id (UUID, FK): Foreign key linking to the Project entity.
        - amount (Float): The amount of money donated.
        - date (Date): The date of the donation.
        - message (String): A personal message from the donor.
        - recurring (Enum): Indicates if the donation is one-time or recurring (pay each month).
    - Function: record the history of donation (bill).
    - Relationships:
        - Many-to-One with Donor indicate that a person can make many donations.
        - Many-to-One with Project indicate that there are many donations for charity projects.
8. Category
    - Attributes:
        - category_id (UUID, PK): A unique identifier for each category.
        - name (Enum): The name of the category.
        - description (String): A description of the category.
    - Function: Defines categories for projects, aiding donors in finding project.
    - Relationships:
        - One-to-Many with Project indicate that each project can only one category but there are many projects or each category.
9. Subscription

- Attributes:
  - subscription_id (UUID, PK): A unique identifier for each subscription.
  - user_id (UUID, FK): Foreign key linking to the User entity.
  - category_id (UUID, FK): Foreign key linking to the Category entity.
  - region (String): The region for which the user subscribes to project notifications.
- Function: Allows donors to subscribe to notifications based on project categories and regions.
- Relationships:
  - Many-to-One with User and Category indicate that user can subscribe to many category.

10. Video
    - Attributes:
      - video_id (UUID, PK): A unique identifier for each video.
      - project_id (UUID, FK): Foreign key linking to the Project entity.
      - url (String): The URL of the video.
      - format (String): The format of the video file.
    - Function: The place to store video for the campaign.
    - Relationships:
      - Many-to-One with Project indicating that a project can have many videos.
11. Image
    - Attributes:
      - image_id (UUID, PK): A unique identifier for each image.
      - project_id (UUID, FK): Foreign key linking to the Project entity.
      - url (String): The URL of the image.
      - format (String): The format of the image file.
    - Function: The place to store images.
    - Relationships:
      - Many-to-One with Project indicating a project can have many pictures to support it.

Advantages and Potential Disadvantages:

Advantages:

1. The relationships between User, Donor, Charity, Project, and Donation clearly and explicitly represent the relationships between classes. The data flow within this system is simplified, making it easier to understand.
2. Standard notations for entity, attribute, and relationship make the ER diagram easy for non-technical users to understand.
3. By normalizing things, the design keeps data from being duplicated. For example, CardInfo is a separate object from User that keeps payment information safe and can be used again.
4. It's easy to add new entities or attributes to the model, like new user role (volunteer) or project categories, without changing the structure that's already there.
5. Private data, like the CVV in CardInfo, can be kept separate from the main object. This means that it can be encrypted and protected separately.
6. Users are more engaged when features like Category and Subscription are available, since it allows contributors to sign up for project updates that are relevant to them.
7. The Video and Image entities linked to the Project provide an organized method for managing multimedia content, hence enhancing project exposure and donor involvement.

Potential Disadvantages:

1. Using multiple relationships may lead to slower query performance with large data sets.
2. Robust validation mechanisms are needed to ensure data consistency across related entities, such as Donation and Project.

III. Frontend

1. Container diagram

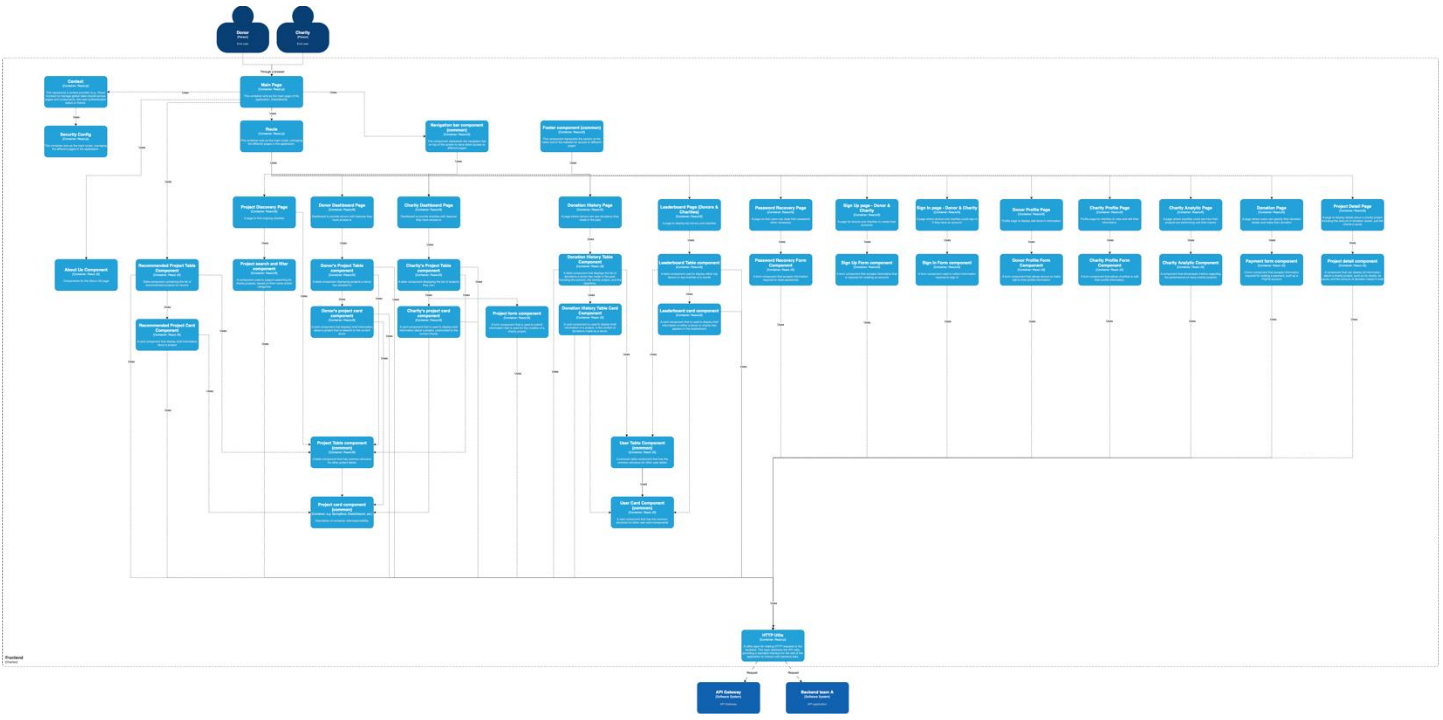

Donor
Charity
Through a browser
Context
Security Config
Main Page
Route
Navigation bar component (common)
Footer component (common)
Project Discovery Page
Donor Dashboard Page
Charity Dashboard Page
Donation History Page
Leaderboard Page (Donors & Charities)
Password Recovery Page
Sign Up page - Donor & Charity
Sign In page - Donor & Charity
Donor Profile Page
Charity Profile Page
Charity Analytic Page
Donation Page
Project Detail Page
About Us Component
Recommended Project Table Component
Recommended Project Card Component
Project search and filter component
Donor's Project Table component
Donor's project card component
Charity's Project Table component
Charity's project card component
Project form component
Donation History Table Component
Donation History Table Card Component
Leaderboard Table component
Leaderboard card component
Password Recovery Form Component
Sign Up Form component
Sign In Form component
Donor Profile Form Component
Charity Profile Form Component
Charity Analytic Component
Payment form component
Project detail component
Project Table component (common)
Project card component (common)
User Table Component (common)
User Card Component (common)
HTTP Utils
API Gateway
Backend team A
Frontend
Uses

Figure 3:Frontend Container Diagram

Link to diagram: Here

This architecture follows a modular, component-based approach, which allows for better organization, reusability, and maintainability of the front-end codebase. Separating concerns between the various containers helps keep the application well-structured and scalable as it grows [1]

The central entry point is the **Homepage/Landing Page**, which serves as the main gateway for users to access the various features and content of the application. From here, users can navigate to several other core pages:

About Us Page: Introduces the organization and people behind the application.

Leaderboard Page: Highlights the top donors for the current month, incentivizing charitable giving.

Charity Discovery Page: Allows users to search for and browse through various charity projects and organizations.

Charity Project Detail Page: Provides an in-depth look at a specific charity, its mission, impact, and donation opportunities.

Once a user is logged in, they gain access to their personalized dashboard, which differs slightly between donors and charity organizers:

Donor Dashboard:

User Profile Page: Allows donors to manage their personal information and settings.

[Removed] Subscription Page: Enables donors to view and manage any recurring donations or charity subscriptions. This page is removed due to the fact our scope could not handle this.

[Removed] Impact Report Page: Provides donors with detailed reporting on how their contributions are making a difference. This page is removed due to the fact our scope could not handle this.

Donation History Page: Allows donors to review their past donations and giving history.

Charity Organizer Dashboard:

[Renamed] Charity Project Creation Page: Empowers charity organizers to set up new fundraising campaigns and initiatives. This page has been turned into project list page, for the sake of simplicity.

Charity Profile Page: Enables organizers to customize and maintain their charity's online presence.

[Removed] Charity Analytics Page: Offers robust data and metrics on the performance and impact of the charity's projects. We have decided to move some aspects of this page to the profile page in order to simplify user access, and enhance their experience.

Additionally, the diagram includes pages and functionality related to the donation process and user account management:

Donation Page: Facilitates the secure processing of monetary contributions from donors.

[Removed] Charity Communication Page: Enables ongoing dialogue and feedback between donors and charity organizers. This page was removed because it is auxiliary in nature, and is out of our scope.

Sign Up/Sign In Pages: Allow new and returning users to access the application.

Password Recovery Page: Provides a mechanism for users to reset forgotten passwords.

Routes: The routing container manages the navigation between the different pages and views within the application. It coordinates the transitions and data flow as users move between the various sections of the app.

Pages: Each page represents a specific part of the application, such as the Main Page, Profile Page, etc. These page components encapsulate the unique content and functionality for that section.

Components: Smaller, focused UI elements that handle specific tasks within a page. Components are designed to be reusable across multiple pages, improving consistency and development efficiency.

Context: This represents a global state management system. It provides a centralized way to share data, states, and functionality that need to be accessed across the entire application.

HTTP Utils: This utility layer abstracts the API calls to the backend, providing a standardized interface for making HTTP requests. It handles concerns like authentication, error handling, and response processing.

Component diagrams

a) Route

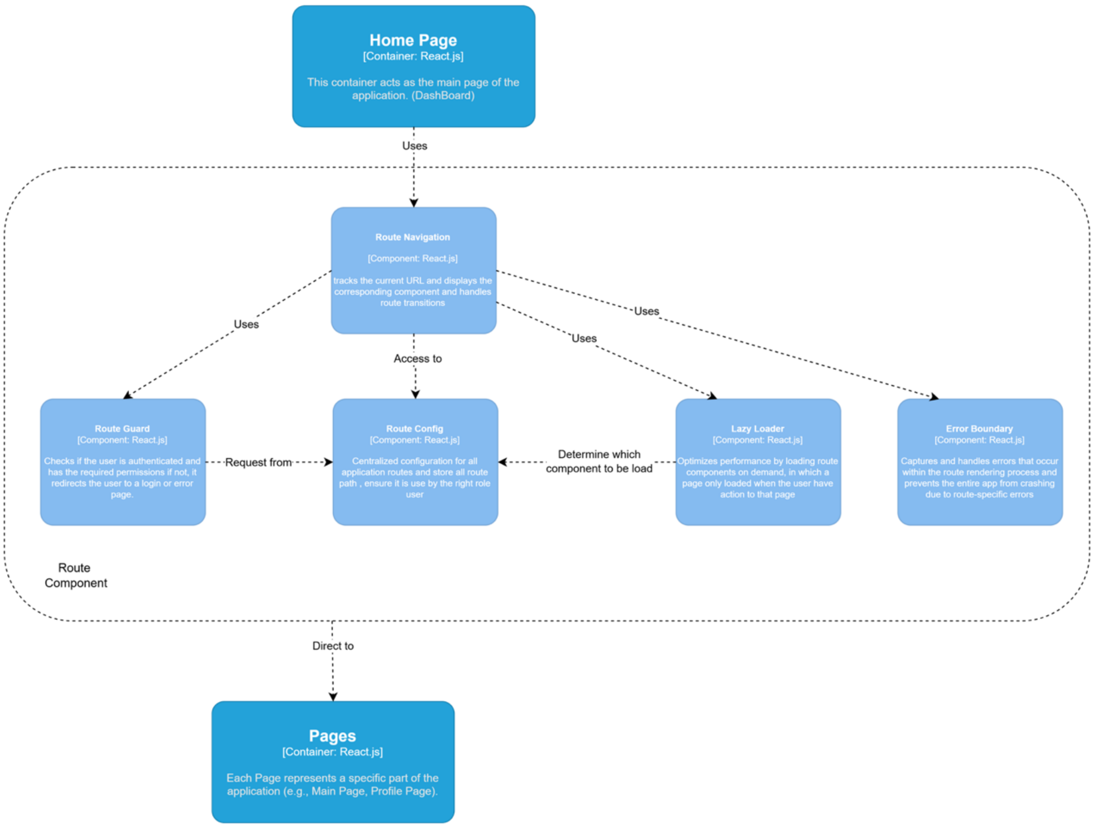

Home Page
[Container: React.js]
This container acts as the main page of the application. (DashBoard)
Uses
Route Navigation
[Component: React.js]
tracks the current URL and displays the corresponding component and handles route transitions
Uses
Access to
Uses
Uses
Route Guard
[Component: React.js]
Checks if the user is authenticated and has the required permissions if not, it redirects the user to a login or error page.
Request from
Route Config
[Component: React.js]
Centralized configuration for all application routes and store all route path , ensure it is use by the right role user
Determine which component to be load
Lazy Loader
[Component: React.js]
Optimizes performance by loading route components on demand, in which a page only loaded when the user have action to that page
Error Boundary
[Component: React.js]
Captures and handles errors that occur within the route rendering process and prevents the entire app from crashing due to route-specific errors
Route Component
Direct to
Pages
[Container: React.js]
Each Page represents a specific part of the application (e.g., Main Page, Profile Page).

Figure 4: Frontend Route

The plan centered around a Route Navigation component that would work in conjunction with several specialized modules: a Route Guard for access control and authentication, a Route Config for centralized route management, a Lazy Loader for dynamic component loading, and an Error Boundary for isolated error handling. Each component was carefully planned to fulfill specific roles in creating a secure, efficient, and user-friendly navigation experience.

However, due to project timeline constraints and evolving requirements, we had to significantly scale back this ambitious design. While the original architecture would have provided sophisticated features like granular access control and optimized performance through lazy loading, we ultimately had to prioritize core functionality over these advanced features. The decision to simplify the routing system, though disappointing from a technical perspective, allowed us to meet our immediate project deadlines and deliver a working solution that satisfied the essential requirements.

b) Components

Link to diagram: Here

The component diagram for Components is organized to promote modularity, allowing for a clean separation of different responsibilities across the system. Each common component encapsulates specific functionality, focusing on reusability and efficiency to extend or update functionalities.

These components are grouped into five sections based on their roles and interactions within the application. Each section highlights the key building blocks that collectively form the foundation of the application's architecture, ensuring a scalable and maintainable design.

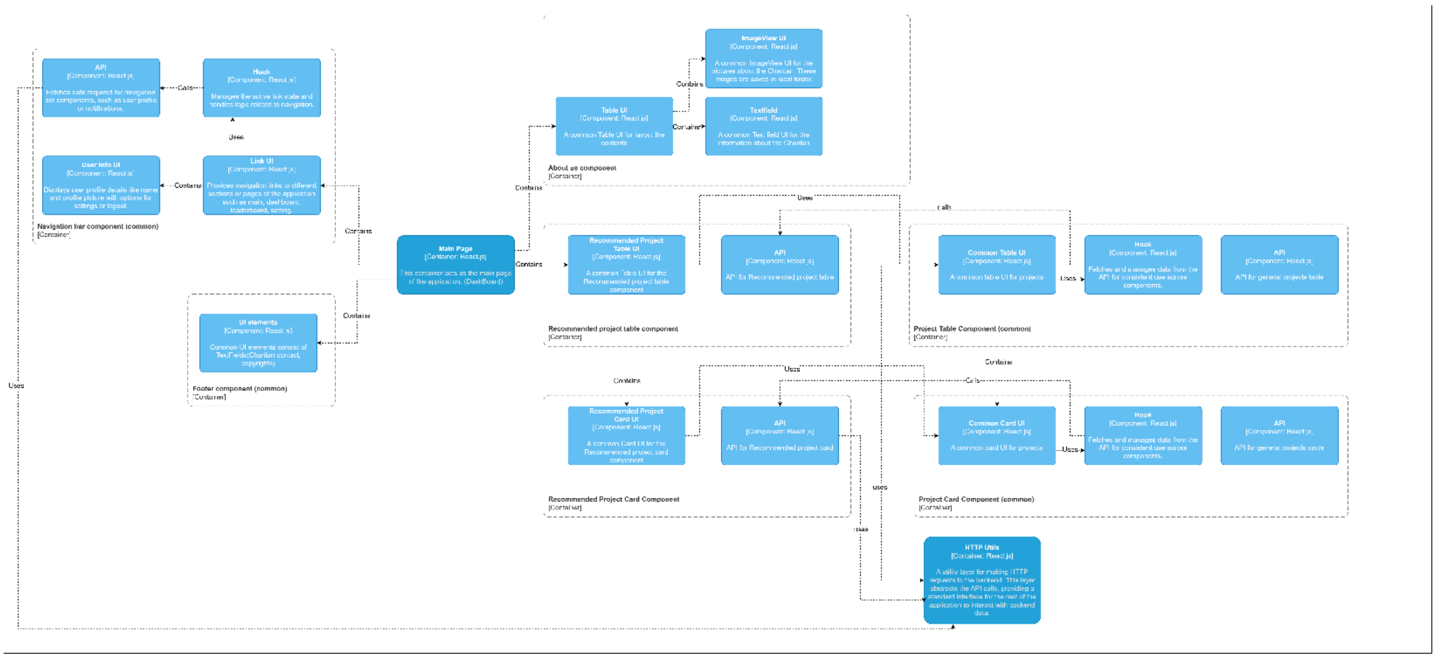

API
[Component: React.js]
Fetches data required for navigation bar components, such as user profile or notifications.
Calls
Hook
[Component: React.js]
Manages the active link state and handles logic related to navigation.
Uses
User Info UI
[Component: React.js]
Displays user profile details like name and profile picture with options for settings or logout
Contains
Link UI
[Component: React.js]
Provides navigation links to different sections or pages of the application such as main, dashboard, leaderboard, setting.
Navigation bar component (common)
[Container]
Contains
Main Page
[Container: React.js]
This container acts as the main page of the application. (DashBoard)
Contains
UI elements
[Component: React.js]
Common UI elements consist of TextFields(Charitan contact, copyrights)
Footer component (common)
[Container]
Uses
ImageView UI
[Component: React.js]
A common ImageView UI for the pictures about the Charitan. These images are saved in local folder.
Contains
Table UI
[Component: React.js]
A common Table UI for layout the contents
Contains
Textfield
[Component: React.js]
A common Text field UI for the information about the Charitan
About us component
[Container]
Contains
Uses
Calls
Recommended Project Table UI
[Component: React.js]
A common Table UI for the Recommended project table component
API
[Component: React.js]
API for Recommended project table
Recommended project table component
[Container]
Common Table UI
[Component: React.js]
A common table UI for projects
Uses
Hook
[Component: React.js]
Fetches and manages data from the API for consistent use across components.
API
[Component: React.js]
API for general projects table
Project Table Component (common)
[Container]
Contains
Uses
Calls
Recommended Project Card UI
[Component: React.js]
A common Card UI for the Recommended project card component
API
[Component: React.js]
API for Recommended project card
Recommended Project Card Component
[Container]
Common Card UI
[Component: React.js]
A common card UI for projects
Uses
Hook
[Component: React.js]
Fetches and manages data from the API for consistent use across components.
API
[Component: React.js]
API for general projects cards
Project Card Component (common)
[Container]
uses
uses
HTTP Utils
[Container: React.js]
A utility layer for making HTTP requests to the backend. This layer abstracts the API calls, providing a standard interface for the rest of the application to interact with backend data.

Figure 5: Main Page Components

This diagram represents the core components used on the **Main Page** of the application.

About Us Component:

- Provides static information about the platform, such as its mission, vision, and team details.
- Encapsulated to ensure the content is modular and easily updatable without affecting other components or pages.

Project Table & Project Card:

- Serve as reusable components for visualizing project-related data.
- The main page utilizes the common table and common card components to maintain consistency and modularity.
- The APIs for the common table and card are not shared with the main page because the main page's data structure differs from other contexts.
- Despite the separation of APIs, these components retain the same visual structure, enabling consistent UI design.

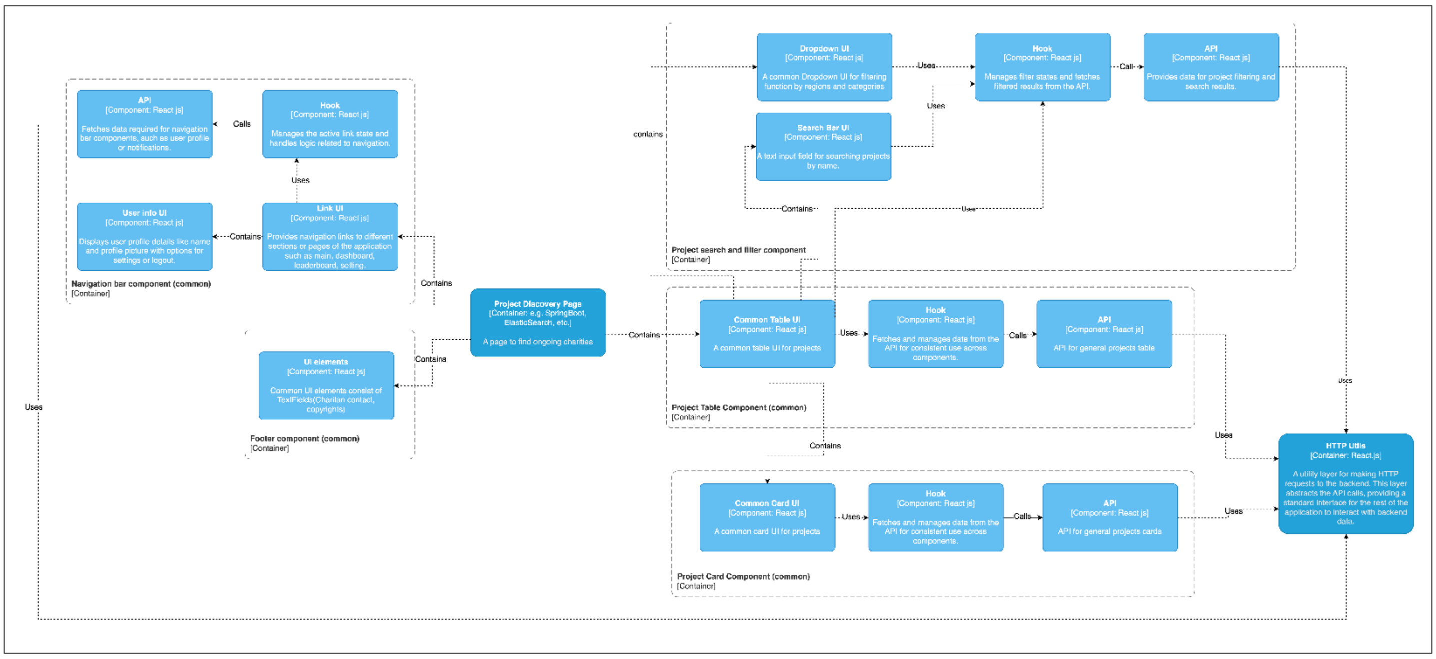

API
[Component: React js]
Fetches data required for navigation bar components, such as user profile or notifications.
Calls
Hook
[Component: React js]
Manages the active link state and handles logic related to navigation.
Uses
User info UI
[Component: React js]
Displays user profile details like name and profile picture with options for settings or logout.
Contains
Link UI
[Component: React js]
Provides navigation links to different sections or pages of the application such as main, dashboard, leaderboard, setting.
Navigation bar component (common)
[Container]
Contains
Project Discovery Page
[Container: e.g. SpringBoot, ElasticSearch, etc.]
A page to find ongoing charities
Contains
UI elements
[Component: React js]
Common UI elements consist of TextFields(Charilan contact, copyrights)
Footer component (common)
[Container]
Uses
contains
Dropdown UI
[Component: React js]
A common Dropdown UI for filtering function by regions and categories
Uses
Hook
[Component: React js]
Manages filter states and fetches filtered results from the API.
Call
API
[Component: React js]
Provides data for project filtering and search results.
Uses
Search Bar UI
[Component: React js]
A text input field for searching projects by name.
Contains
Uses
Project search and filter component
[Container]
Contains
Common Table UI
[Component: React js]
A common table UI for projects
Uses
Hook
[Component: React js]
Fetches and manages data from the API for consistent use across components.
Calls
API
[Component: React js]
API for general projects table
Project Table Component (common)
[Container]
Contains
Uses
Uses
HTTP Utils
[Container: React.js]
A utility layer for making HTTP requests to the backend. This layer abstracts the API calls, providing a standard interface for the rest of the application to interact with backend data.
Common Card UI
[Component: React js]
A common card UI for projects
Uses
Hook
[Component: React js]
Fetches and manages data from the API for consistent use across components.
Calls
API
[Component: React js]
API for general projects cards
Uses
Project Card Component (common)
[Container]

Figure 6: Project Discovery Page Components

This diagram focuses on components involved in the **Project Discovery Page**, emphasizing dynamic filtering and displaying project-related data.

Common Table & Common Card:

- The Common Table and Common Card components are designed to ensure consistency and reusability across multiple pages. However, the APIs used within these components are exclusive to this page, as they are tailored to handle the project-related data dynamically.
- These components also integrate filter functionality, enabling users to refine content based on criteria such as region or category. This combination of internal APIs and filtering makes these components both flexible and highly specific to this page's needs.
- The Table component renders the filtered data by utilizing the filter hook which centralizes state management to ensure consistency and dynamic adaptation to changes in filter criteria.

Filter Components:

- The filter components collect user inputs (e.g., dropdown selections, search queries) and update the filter state using hooks.
- The updated state is sent to the backend via APIs, which returns the filtered data.
- The filtered data is then rendered within the Common Table or Common Card components, ensuring a seamless and interactive user experience.

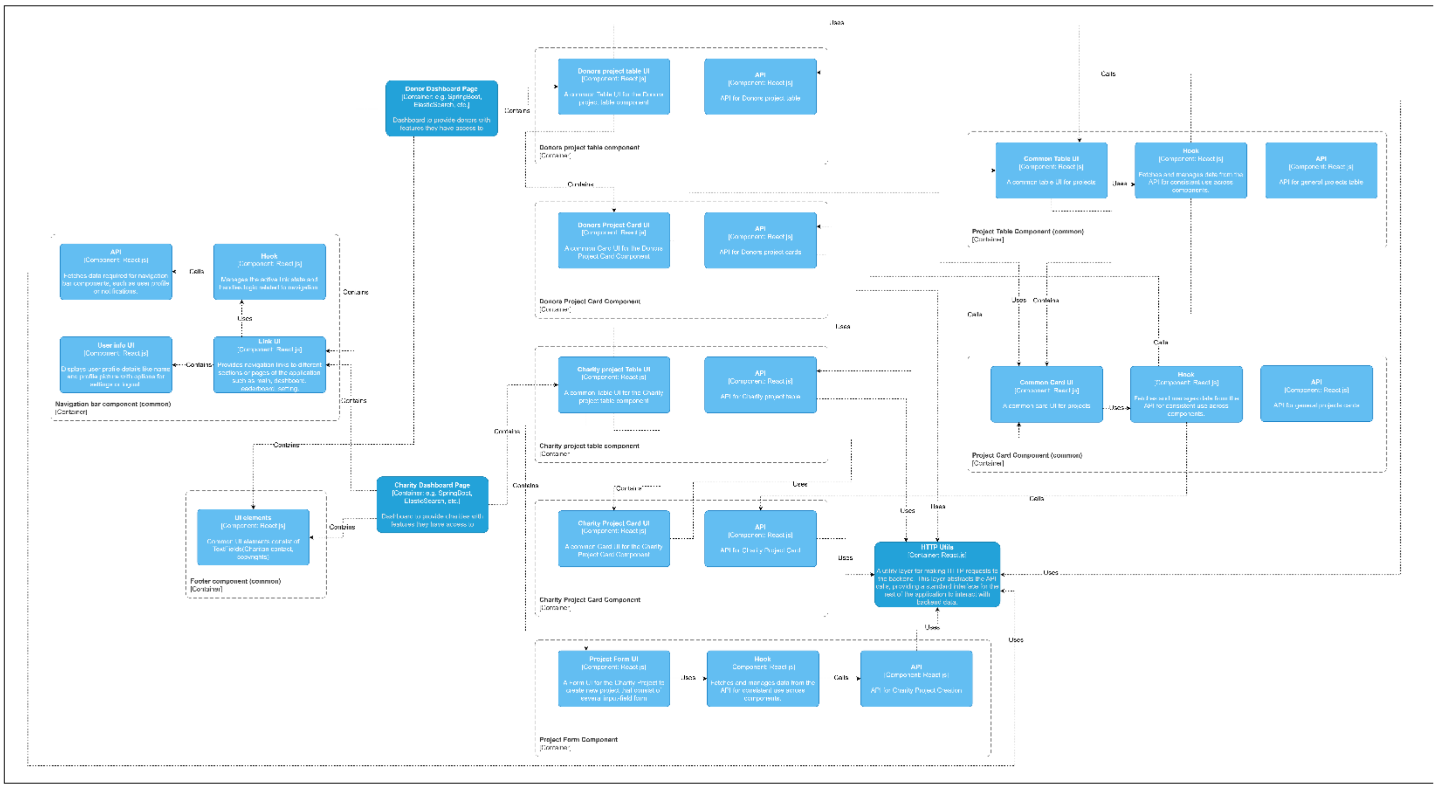

Donor Dashboard Page
[Container: e.g. SpringBoot, ElasticSearch, etc.]
Dashboard to provide donors with features they have access to
Donors project table UI
[Component: React.js]
A common Table UI for the Donors project table component
API
[Component: React.js]
API for Donors project table
Donors project table component
[Container]
Donors Project Card UI
[Component: React.js]
A common Card UI for the Donors Project Card Component
API
[Component: React.js]
API for Donors project cards
Donors Project Card Component
[Container]
Common Table UI
[Component: React.js]
A common table UI for projects
Hook
[Component: React.js]
Fetches and manages data from the API for consistent use across components.
API
[Component: React.js]
API for general projects table
Project Table Component (common)
[Container]
API
[Component: React.js]
Fetches data required for navigation bar components, such as user profile or notifications.
Hook
[Component: React.js]
Manages the active link state and handles logic related to navigation
User info UI
[Component: React.js]
Displays user profile details like name and profile picture with options for settings or logout
Link UI
[Component: React.js]
Provides navigation links to different sections or pages of the application such as main, dashboard, leaderboard, setting.
Navigation bar component (common)
[Container]
Charity project Table UI
[Component: React.js]
A common table UI for the Charity project table component
API
[Component: React.js]
API for Charity project table
Charity project table component
[Container]
Common Card UI
[Component: React.js]
A common card UI for projects
Hook
[Component: React.js]
Fetches and manages data from the API for consistent use across components.
API
[Component: React.js]
API for general projects cards
Project Card Component (common)
[Container]
Charity Dashboard Page
[Container: e.g. SpringBoot, ElasticSearch, etc.]
Dashboard to provide charities with features they have access to
UI elements
[Component: React.js]
Common UI elements consist of TextFields(Charities contact, copyrights)
Footer component (common)
[Container]
Charity Project Card UI
[Component: React.js]
A common Card UI for the Charity Project Card Component
API
[Component: React.js]
API for Charity Project Card
Charity Project Card Component
[Container]
HTTP Utils
[Container: React.js]
A utility layer for making HTTP requests to the backend. This layer abstracts the API calls, providing a standard interface for the rest of the application to interact with backend data.
Project Form UI
[Component: React.js]
A Form UI for the Charity Project to create new project that consist of several input-field form
Hook
[Component: React.js]
Fetches and manages data from the API for consistent use across components.
API
[Component: React.js]
API for Charity Project Creation
Project Form Component
[Container]
Contains
Calls
Uses

Figure 7: Donor Dashboard Page Components

This diagram represents the integration of components used for managing and displaying donor and charity project dashboard information in **Donor Dashboard Page** and **Charity Dashboard Page**.

Donor Dashboard Components:

- The Donor Project Table UI and Donor Project Card UI components are designed to display projects relevant to donors. These components rely on donor-specific APIs to fetch and render project data dynamically.
- These components are modularly connected to ensure that donor-related information is encapsulated and does not interfere with charity-specific functionalities.

Charity Dashboard Components:

- Similarly, the Charity Project Table UI and Charity Project Card UI components focus on displaying projects for charity organizers. They utilize charity-specific APIs to fetch and manage project data.
- These components include a Project Form UI to allow charity organizers to create or manage their projects dynamically.

Common Components:

- Both donor and charity dashboards utilize common table and card components for consistent design and layout. However, these components interact with donor-specific or charity-specific APIs, depending on the context.
- This separation ensures that the API logic is tailored to each dashboard's unique data requirements while retaining visual consistency.

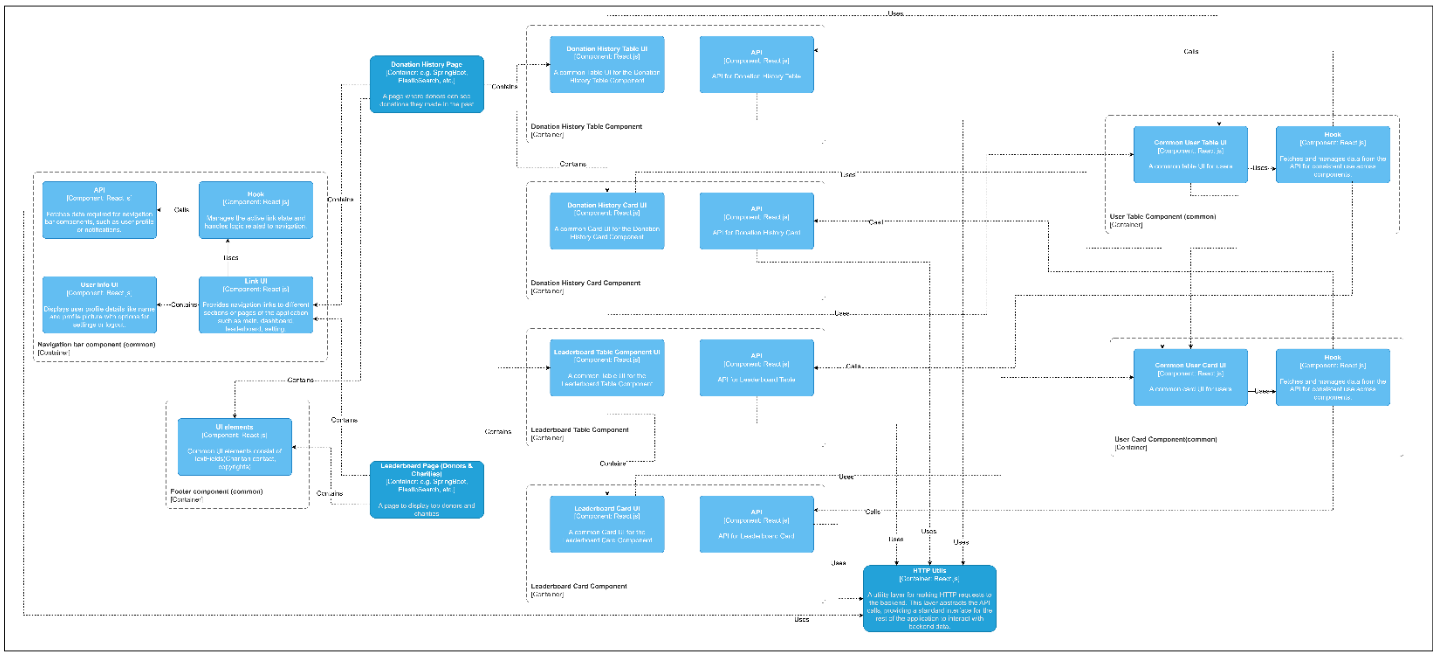

Donation History Page
[Container: e.g. SpringBoot, ElasticSearch, etc.]
A page where donors can see donations they made in the past
Leaderboard Page (Donors & Charities)
[Container: e.g. SpringBoot, ElasticSearch, etc.]
A page to display top donors and charities
Donation History Table UI
[Component: React.js]
A common Table UI for the Donation History Table Component
API
[Component: React.js]
API for Donation History Table
Donation History Table Component
[Container]
Donation History Card UI
[Component: React.js]
A common Card UI for the Donation History Card Component
API
[Component: React.js]
API for Donation History Card
Donation History Card Component
[Container]
Leaderboard Table Component UI
[Component: React.js]
A common Table UI for the Leaderboard Table Component
API
[Component: React.js]
API for Leaderboard Table
Leaderboard Table Component
[Container]
Leaderboard Card UI
[Component: React.js]
A common Card UI for the Leaderboard Card Component
API
[Component: React.js]
API for Leaderboard Card
Leaderboard Card Component
[Container]
API
[Component: React.js]
Fetches data required for navigation bar components, such as user profile or notifications.
Hook
[Component: React.js]
Manages the active link state and handles logic related to navigation.
User Info UI
[Component: React.js]
Displays user profile details like name and profile picture with options for settings or logout.
Link UI
[Component: React.js]
Provides navigation links to different sections or pages of the application such as main, dashboard, leaderboard, setting.
Navigation bar component (common)
[Container]
UI elements
[Component: React.js]
Common UI elements consist of textFields(Chart on contact, copyrights)
Footer component (common)
[Container]
Common User Table UI
[Component: React.js]
A common table UI for users
Hook
[Component: React.js]
Fetches and manages data from the API for consistent use across components.
User Table Component (common)
[Container]
Common User Card UI
[Component: React.js]
A common card UI for users
Hook
[Component: React.js]
Fetches and manages data from the API for consistent use across components.
User Card Component(common)
[Container]
HTTP Utils
[Container: React.js]
A utility layer for making HTTP requests to the backend. This layer abstracts the API calls, providing a standard interface for the rest of the application to interact with backend data.
Contains
Calls
Uses

Figure 8: Donation History Page Components

This diagram illustrates components for the **Donation History Page** and **Leaderboard Page**, emphasizing reusable structures and dynamic data retrieval through specific APIs.

Donation History Page Components:

- Donation History Table UI and Donation History Card UI are the primary components used to display detailed donation records for users.
- These components use Donation History APIs to fetch data dynamically and render it into their respective structures.
- The structure leverages Common User Table UI and Common User Card UI, ensuring consistency across different pages while tailoring them to donation-specific needs.

Leaderboard Page Components:

- The Leaderboard Table Component UI and Leaderboard Card UI focus on showcasing user rankings based on donation contributions.
- These components interact with the Leaderboard API for real-time ranking data and present it using reusable card and table structures.
- The modularity of these components allows the leaderboard to adapt to potential ranking metrics without requiring structural changes.

Common Components:

- Both pages make extensive use of Common User Table UI and Common User Card UI, providing a consistent layout while allowing for page-specific adaptations.
- These components integrate hooks and APIs dynamically, enabling tailored content rendering while maintaining a unified design language.

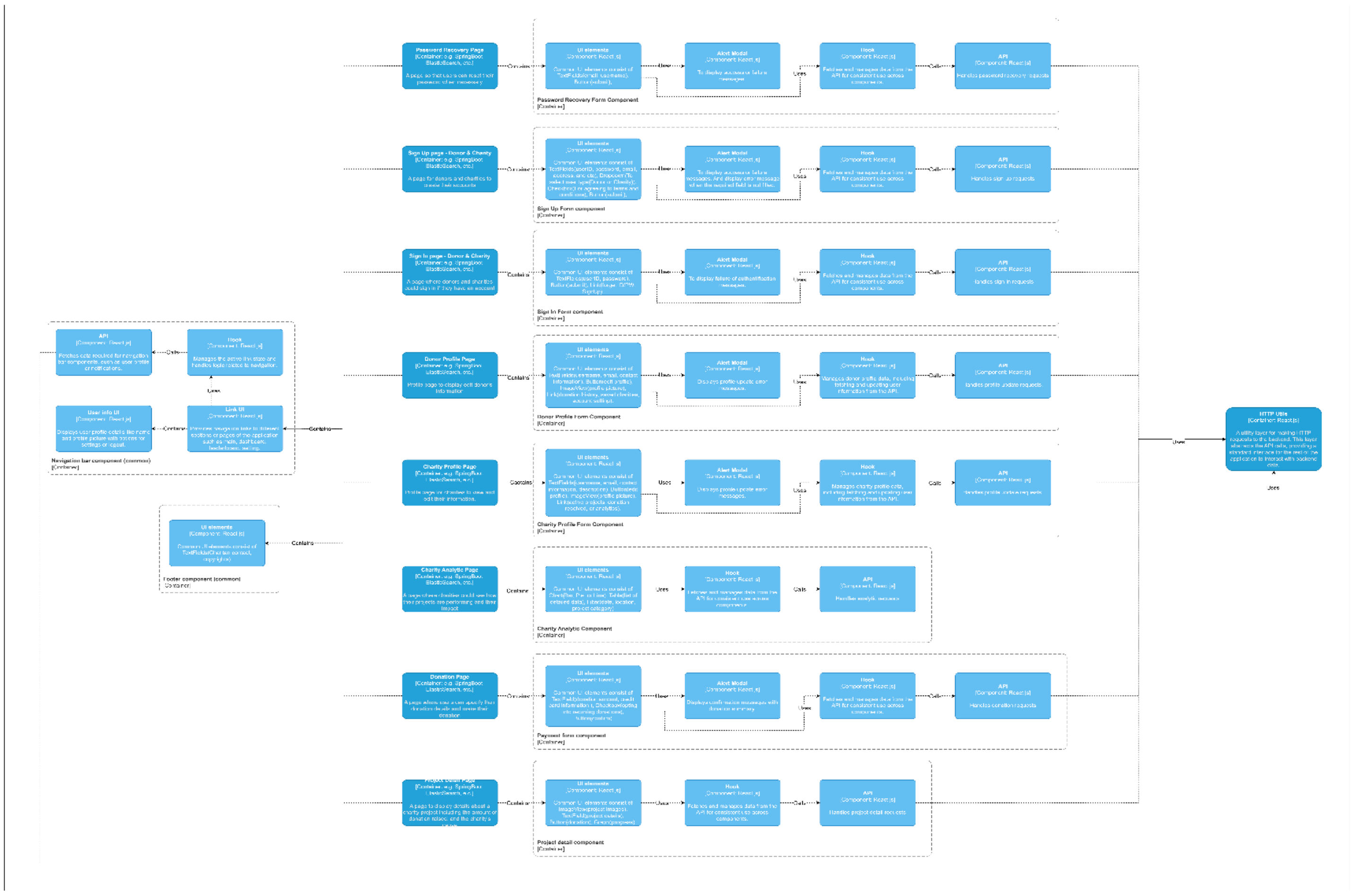


Figure 9. Pages dataflow

The pages in this diagram share a consistent structure and data flow, emphasizing modularity and reusability.

UI Elements:

- Input fields, forms, and modals serve as the primary user interaction layer.
- These components handle data collection and immediate user feedback (e.g., validation errors, success messages).

Hooks:

- Manage state updates, form validation, and API requests.
- Ensure seamless communication between UI and backend, including loading states and error handling.

APIs:

- Handle specific backend calls to retrieve, update, or process data.
- APIs are categorized based on the purpose of each page (e.g., authentication, profile management, analytics).

Alert Modals:

- Provide standardized feedback for actions, such as success or error notifications, ensuring a consistent user experience.

In addition to the unique components and flows discussed earlier, the Navbar, Footer, and HTTP Utils serve as foundational elements that are shared across all pages. These components not only ensure consistency in navigation and user experience throughout the application but also streamline the communication between the frontend and backend by providing a centralized utility for API calls. Below is an explanation of their role and implementation.

Navigation Bar & Footer:

- Both are encapsulated components included on every page to ensure a consistent layout and user experience.
- Provide shared functionality, such as navigation links, branding (via the logo), and access to external links or policies.
- Encapsulation avoids redundancy and makes these components easy to maintain and update globally.

HTTP Utils Integration:

- The HTTP Utils layer abstracts API calls for the different dashboards and ensures a standardized way of interacting with the backend.
- This centralized management reduces redundancies and provides flexibility for future API changes.

Architecture rationale:

- **Maintainability:** Each component encapsulated with the specific functionality allows isolated development and debugging. In addition, modular hooks and APIs minimize code duplication and simplify changes to core logic across components. The modular structure ensures that updates in one component do not unintentionally impact others and reduce regression risks.
- **Extensibility:** Reusable components such as Common Project Table UI or Common project Card UI and hooks allow new pages or features to integrate seamlessly with existing components, minimizing development overhead. The architecture supports the addition of unique components like Leaderboard Table UI without altering the base structure.
- **Resilience:** Hooks manage API calls, handle error states, and validate data to ensure consistent and robust user experiences. Components like Alert Modal provide immediate feedback for errors or successful operations, ensuring clear communication with the user.
- **Scalability:** API-specific components allow backend services to scale independently of frontend components. Reusable UI elements support efficient development and integration as the application grows, maintaining performance across complex workflows.
- **Security:** Encapsulation of APIs within their respective components prevents unauthorized access or unintended data exposure.

Lana

c) Context

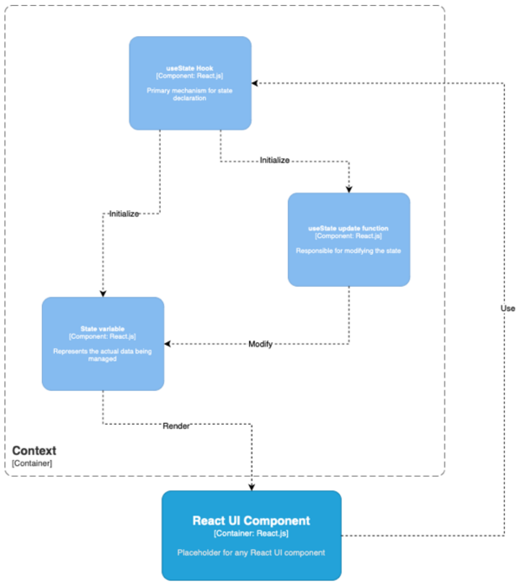


Figure 10. Context container diagram

The diagram depicts a React-based system with a primary state management mechanism centered around the "useState" hook. This hook is part of the React library and provides a way to manage the state within functional components. [2]

The main components shown are:

- **UseState Hook**: This is the core state management mechanism, responsible for managing the state of the application.
- **State Variable**: This represents the actual data being managed and stored by the useState hook.
- **UseState Update Function**: This is the function provided by the useState hook to update the state variable.
- **React UI Component:** This is a placeholder for any React UI component that might utilize the state managed by the useState hook.

The useState hook in React provides a straightforward and lightweight way to manage state within a component. It keeps the state local to the component, making it easy to reason about and manage. This simplicity and performance benefit are key advantages, especially for smaller, self-contained components.

However, as an application grows in complexity, using useState alone can become challenging. Larger applications may require a more centralized state management solution like Redux [3] or Zustand [4] which offer better mechanisms for global state sharing, improved testability, and handling of advanced state management tasks. The right choice depends on factors like application size, state complexity, and the need for global state management and testability.

IV. Backend

2. Container diagram

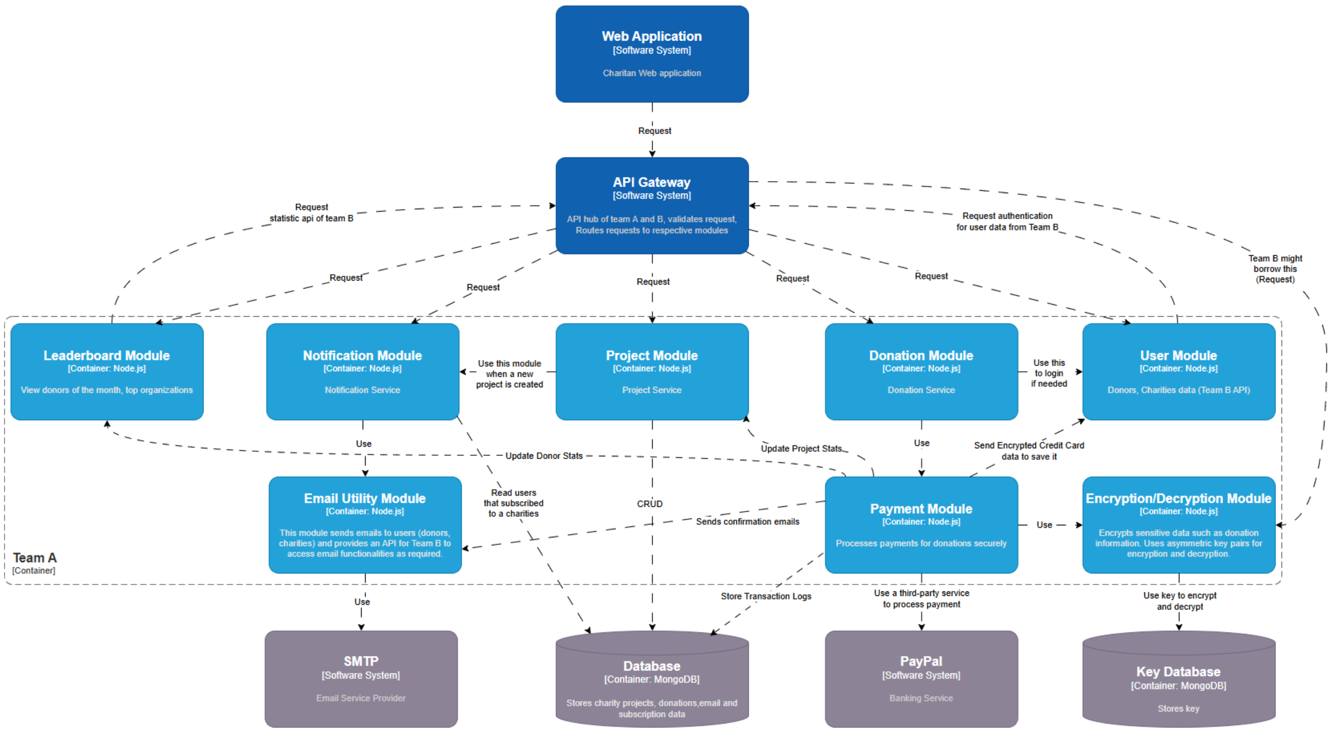

Web Application
[Software System]
Charitan Web application
Request
API Gateway
[Software System]
API hub of team A and B, validates request, Routes requests to respective modules
Request statistic api of team B
Request authentication for user data from Team B
Team B might borrow this (Request)
Request
Request
Request
Request
Request
Leaderboard Module
[Container: Node.js]
View donors of the month, top organizations
Notification Module
[Container: Node.js]
Notification Service
Use this module when a new project is created
Project Module
[Container: Node.js]
Project Service
Donation Module
[Container: Node.js]
Donation Service
Use this to login if needed
User Module
[Container: Node.js]
Donors, Charities data (Team B API)
Use
Update Donor Stats
Update Project Stats
Use
Send Encrypted Credit Card data to save it
Email Utility Module
[Container: Node.js]
This module sends emails to users (donors, charities) and provides an API for Team B to access email functionalities as required.
Read users that subscribed to a charities
CRUD
Sends confirmation emails
Payment Module
[Container: Node.js]
Processes payments for donations securely
Use
Encryption/Decryption Module
[Container: Node.js]
Encrypts sensitive data such as donation information. Uses asymmetric key pairs for encryption and decryption.
Team A
[Container]
Use
Store Transaction Logs
Use a third-party service to process payment
Use key to encrypt and decrypt
SMTP
[Software System]
Email Service Provider
Database
[Container: MongoDB]
Stores charity projects, donations,email and subscription data
PayPal
[Software System]
Banking Service
Key Database
[Container: MongoDB]
Stores key

Figure 11: Team A's Milestone 1 Backend Container

This container describes the modules that will be used to interact with Team A's requirements, which caters to Donors and Charities. Following a Modular Monolith design, at the highest level, Web Application is the software which refers to the interface which the users view and interact. Through this interaction, each request regarding dynamic information viewing and adjusting will be requested through the API gateway, containing APIs for both Team A and B; depending on the request, it will send to individual modules to handle the action done by the user. However, this container (figure 11) illustrates the previous iteration. During the implementation, the team decided to reconfigure the diagram to match the requirement, changing the workflow and databases. Thus, the team came up with a new container diagram that strictly adheres to the requirement.

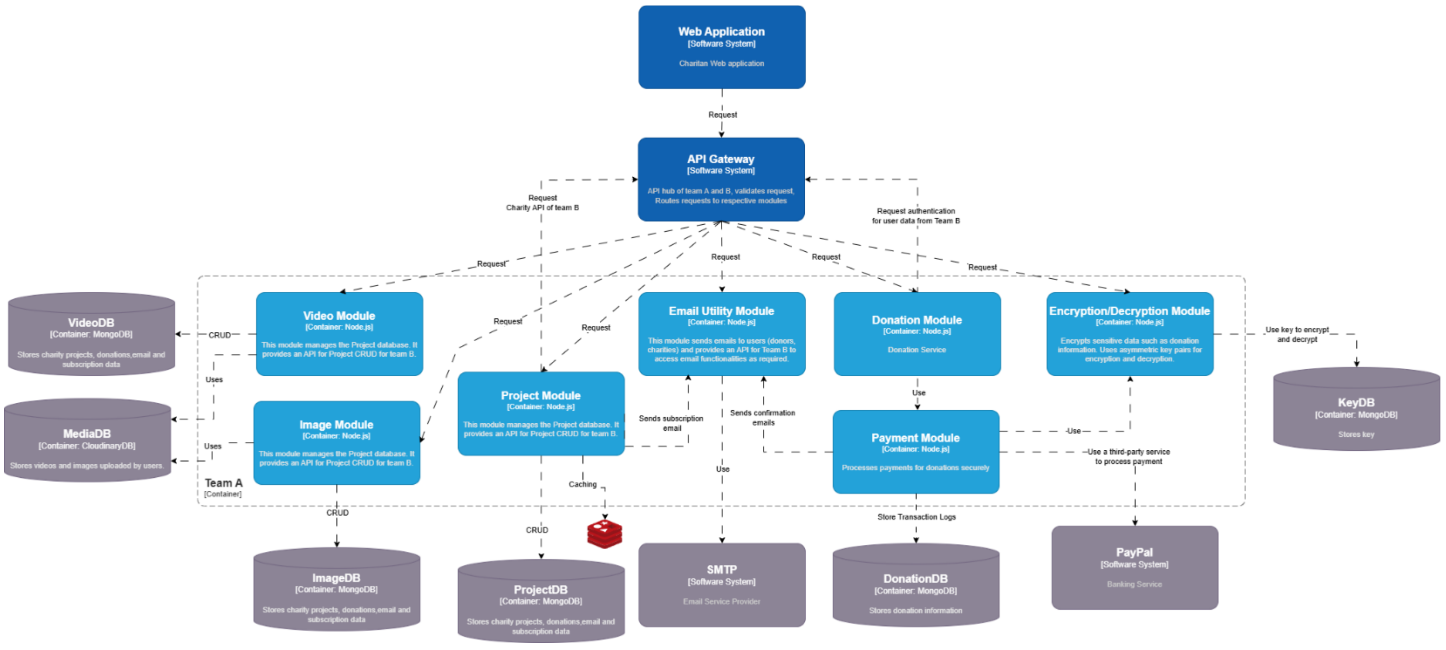

Web Application
[Software System]
Charitan Web application
Request
API Gateway
[Software System]
API hub of team A and B, validates request, Routes requests to respective modules
Request Charity API of team B
Request authentication for user data from Team B
Request
Request
Request
Request
Request
Request
VideoDB
[Container: MongoDB]
Stores charity projects, donations,email and subscription data
CRUD
Video Module
[Container: Node.js]
This module manages the Project database. It provides an API for Project CRUD for team B.
Uses
MediaDB
[Container: CloudinaryDB]
Stores videos and images uploaded by users.
Uses
Image Module
[Container: Node.js]
This module manages the Project database. It provides an API for Project CRUD for team B.
Project Module
[Container: Node.js]
This module manages the Project database. It provides an API for Project CRUD for team B.
Email Utility Module
[Container: Node.js]
This module sends emails to users (donors, charities) and provides an API for Team B to access email functionalities as required.
Sends subscription email
Sends confirmation emails
Donation Module
[Container: Node.js]
Donation Service
Use
Payment Module
[Container: Node.js]
Processes payments for donations securely
Use
Use a third-party service to process payment
Encryption/Decryption Module
[Container: Node.js]
Encrypts sensitive data such as donation information. Uses asymmetric key pairs for encryption and decryption.
Use key to encrypt and decrypt
KeyDB
[Container: MongoDB]
Stores key
Team A
[Container]
CRUD
Caching
CRUD
Use
Store Transaction Logs
ImageDB
[Container: MongoDB]
Stores charity projects, donations,email and subscription data
ProjectDB
[Container: MongoDB]
Stores charity projects, donations,email and subscription data
SMTP
[Software System]
Email Service Provider
DonationDB
[Container: MongoDB]
Stores donation information
PayPal
[Software System]
Banking Service

Figure 12. Team A's Milestone 2 container diagram

[Removed] Leaderboard module: Responsible for storing information regarding users who are the top donors of the month for that charity, as well as displaying the top charitable organizations in Charitan. The module achieves this by requesting the statistical API developed by team B, using the API gateway. Once completed, this information is then displayed onto the Viewing layer of the Web application.

[New] Video and Image Module: Responsible for storing videos and images uploaded by each project from the Charities. The videos and images are stored in an External Service called Cloudinary, which serves as a media database. After uploading an image, Cloudinary will provide a link for Frontend to render the specified image; where the link will be stored by respectively in its MongoDB with a link to the project that uploaded the image.

[Removed] Notification module: works to alert the donors and charities, for donors, they will receive a notification when they have created an account, a regional, category or charity-specific subscription regarding its activities.

Email Utility Module: Responsible for sending out email to users (donors, and charities), which is used for notification purposes. Furthermore, this module provides an API for Team B to use email functionality corresponding to the requirement.

Project Module: Used to create, read, update or delete any related information regarding each Charity's Project. This module will be able to use the Email Module to send notification upon project creation or completion based on the donors subscription to the charity.

Donation Module: Handles each donation from donors to charities as it stores the transaction history and confirms the transaction completion to the web application interface.

Payment Module: A module used by Donation Module to process the transactions. Moreover, depending on the donor, if an account log in is required, the system will use the User Module which stores and handles Donors and Charities data, requesting an API for Authentication from Team B to complete its action.

Encryption/Decryption Module: Functions to secure the transactional data by using asymmetric key pairs for encryption and decryption. This module is required to secure users banking information, preventing access from third parties.

Simple Mail Transfer Protocol (SMTP): A protocol that exchanges email between servers, precisely used to send and receive emails. This protocol will be used by the Email Utility Module to send out emails as notification, other than device notifications maintained by the Notification Module.

[Removed] User Module: A module to store user (donors and charities) information, which uses APIs from team B for authentication when necessary. This module serves to display user information to the current logged in user (donors, and charities), and the charities that are behind each project.

For the external services to support email, data storage, transaction and security functionality, four services were chosen.

Database: Using MongoDB as the main database, a NoSQL database is used to store charity projects, donations, email and subscription data for a dynamic range of data, without requiring strict schema constraints.

Redis: A service to cache project related information to lessen the Project database's data access.

Paypal: A service used to authenticate banking information, and process transactions. Donors will use this service to submit donations to charities.

3. Component diagrams

a) Project Module

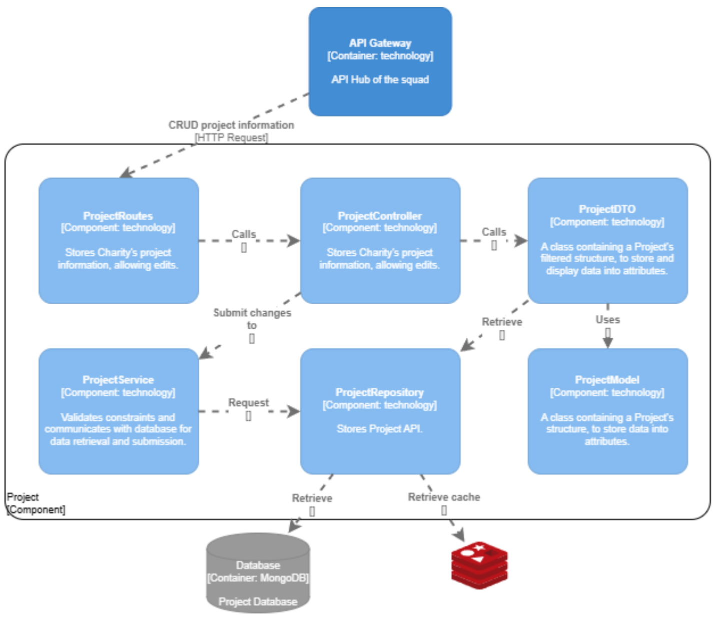
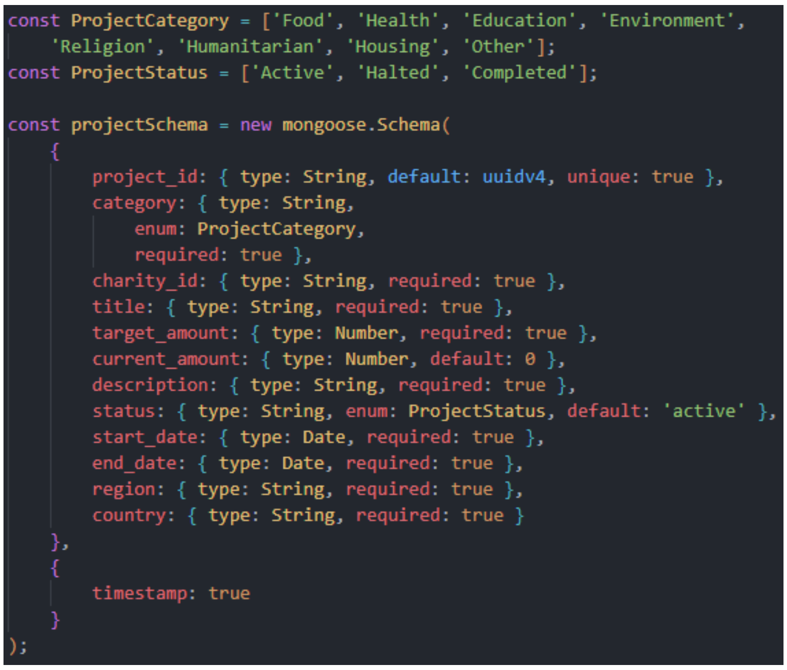


Figure 13. Project Module component diagram

The Project Module handles all project-related functionalities, that is, retrieving, storing, creating, updating and deleting Project information. In consideration of Charitan's Projects functionality, the design contains components which individually has responsibility to allow Donors to view, Charities to create, edit and update, and Admins to delete projects. Therefore, the composition of this module has the following components:

ProjectModel: A model which stores the Project attributes (Figure 14). This component is responsible for structuring the data received from the database or Redis and store it in proper attributes to be filtered by the ProjectDTO.



```
const ProjectCategory = ['Food', 'Health', 'Education', 'Environment',
    'Religion', 'Humanitarian', 'Housing', 'Other'];
const ProjectStatus = ['Active', 'Halted', 'Completed'];

const projectSchema = new mongoose.Schema(
    {
        project_id: { type: String, default: uuidv4, unique: true },
        category: { type: String,
            enum: ProjectCategory,
            required: true },
        charity_id: { type: String, required: true },
        title: { type: String, required: true },
        target_amount: { type: Number, required: true },
        current_amount: { type: Number, default: 0 },
        description: { type: String, required: true },
        status: { type: String, enum: ProjectStatus, default: 'active' },
        start_date: { type: Date, required: true },
        end_date: { type: Date, required: true },
        region: { type: String, required: true },
        country: { type: String, required: true }
    },
    {
        timestamp: true
    }
);
```

Figure 14. Project Model

[New] ProjectDTO: A data transfer object that filters excess data from retrieval requests such as _id and __v from MongoDB.

ProjectRepository: A component storing all related Project APIs to communicate with MongoDB. Its main purpose is to have reusable APIs that can retrieve common clusters of information to be used by the Controller or Service components in the module.

ProjectService: A component responsible for processing the information regarding projects. Specifically, it will validate the data that is being collected from the Web Application during Create and Update procedures on whether the data field fulfills the predefined constraints. The result will be the validation of the submitted data to be rejected or accepted before publishing it onto the database using ProjectRepo.

ProjectController: A component responsible for storing a charity's project information, edited project information submission. In addition, ProjectController communicates with the Web Application to handle the Frontend's retrieval of clusters or all information about the Charitan's current projects.

[New] ProjectRoutes: A component responsible for storing HTTP routes to call functions using the API Gateway.

Architecture rationale:

- **Maintainability:** This design allows developers to create, update, improve and remove Project-related features in its respective component, such as Project Services to handle logic, and Project Controller to handle displayed data on the web application. During this implementation, changes in one component is isolated from other components' functionality, reducing impact on other features.
- **Extensibility:** The modular design that makes each module have one specific responsibility results in an easier navigation to implement new features, such as implementing a new validation rule will be localized to ProjectServices or adding new API being within ProjectRepo. This way, it allows a simple debugging process to troubleshoot the error within the component.
- **Resilience:** Due to the validation process of ProjectService, it reduces the chance of faulty, corrupt data to be sent to the database; reducing the chance system failure in data retrievals.
- **Scalability:** By using ProjectRepo with MongoDB and Redis, the module can handle read-heavy loads, during the retrieval process, as data can be cache in Redis to lessen the load on reaching the database.
- **Security:** Project functions are encapsulated within its own module, from the system, where necessary it provides APIs from the ProjectRepo for other module to strictly retrieve Project data.

b) [New] Image and Video Module

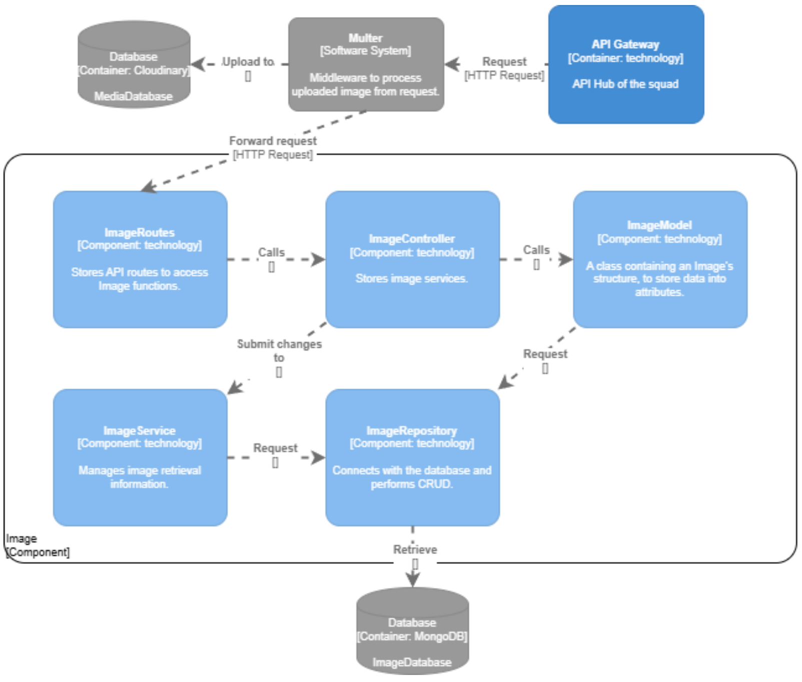
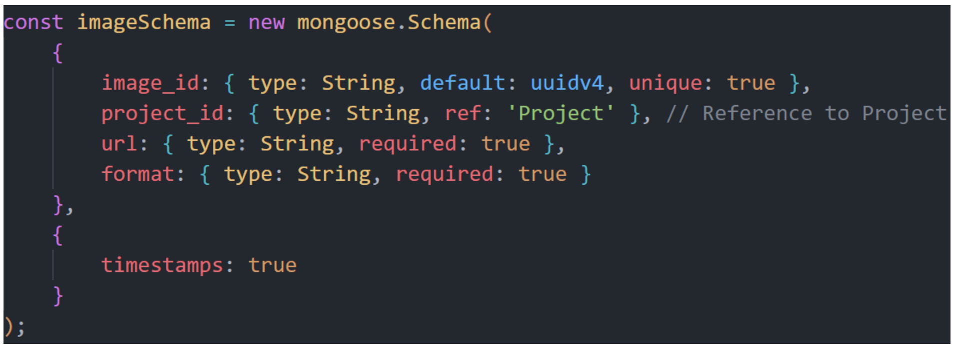


Figure 15. Image module component diagram

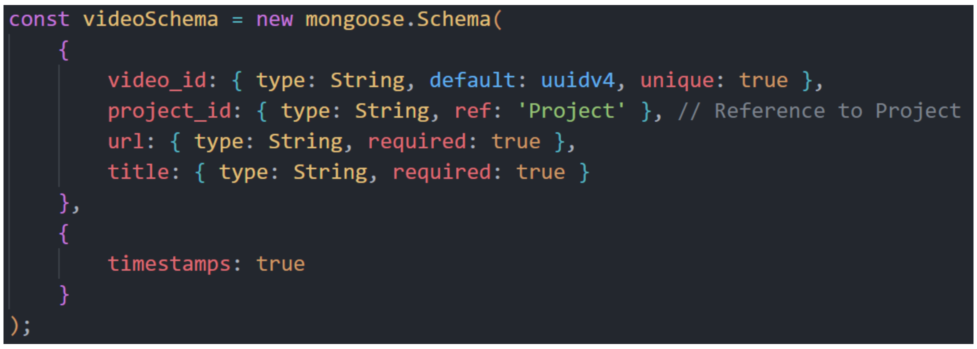
Image and Video Module function similarly as it both receives respective media files, uploads the media onto Cloudinary and stores the uploaded image's Cloudinary URL to MongoDB's database. This method of storing images allows the Frontend to restore the necessary images stored in the database according to the uploaded images by the Charities onto their Projects.

ImageModel and VideoModel: These are the schemas that data from Images and Videos are stored in MongoDB.



```
const imageSchema = new mongoose.Schema(
    {
        image_id: { type: String, default: uuidv4, unique: true },
        project_id: { type: String, ref: 'Project' }, // Reference to Project
        url: { type: String, required: true },
        format: { type: String, required: true }
    },
    {
        timestamps: true
    }
);
```

Figure 16. Image Schema



```
const videoSchema = new mongoose.Schema(
    {
        video_id: { type: String, default: uuidv4, unique: true },
        project_id: { type: String, ref: 'Project' }, // Reference to Project
        url: { type: String, required: true },
        title: { type: String, required: true }
    },
    {
        timestamps: true
    }
);
```

Figure 17. Video Schema

ImageRoutes and VideoRoutes: Routes component stores the API address for the request to access functionality of both media.

ImageController and VideoController: The Controller component is responsible for receiving a request and responding accordingly upon successful completion of the function.

ImageService and VideoService: The Service component is responsible filtering and validating the data before it is submitted to be stored in the database.

ImageRepository and VideoRepository: The Repository component is the communication component that contacts the Image and Video database respectively to perform CRUD activities.

Architecture rationale:

- **Maintainability:** This design allows developers to create, update, improve and remove Media-related features in its respective component, such as Image and Video Services to handle logic or filtering and validation, while Image Controller, and Video Controller to handle displayed media on the web application. During this implementation, changes in one component is isolated from other components' functionality, reducing impact on other features.
- **Extensibility:** The modular design that makes each module have one specific responsibility results in an easier navigation to implement new features, such as implementing a new validation rule will be localized to the Services component or adding new API being within repository. This way, it allows a simple debugging process to troubleshoot the error within the component.
- **Resilience:** In future implementations, thorough validation processes can be implemented in the Service components to ensure that data is consistent and has no error. To which, this data can be inserted into the database and guarantee that no data mismatch will occur.
- **Scalability:** By using Repository with MongoDB and Cloudinary, the component can handle read-heavy loads, during the retrieval process, as the data storing process is split between MongoDB and Cloudinary for media information and media content respectively.
- **Security:** Each functions are encapsulated within its own module, from the system, where necessary it provides APIs from the Repository for other module to strictly use methods that does not interfere with other modules.

c) [Adjusted] Encryption/ Decryption module

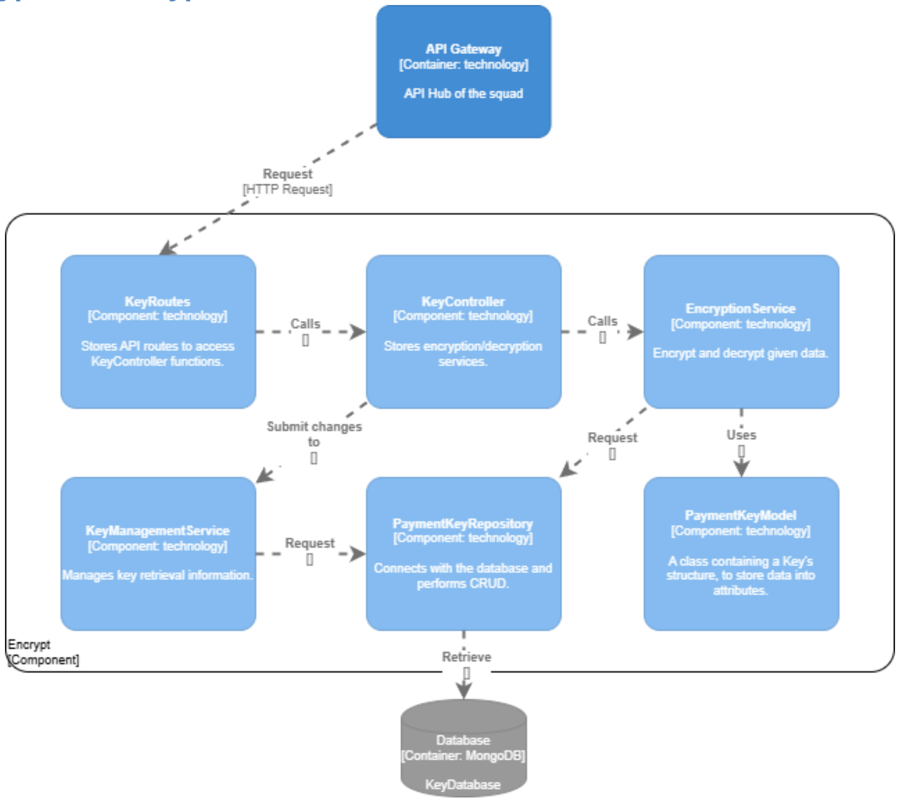


Figure 18. Encryption/Decryption module component diagram

The Encryption/Decryption module is an important part of Charitan's backend system, ensuring the security and privacy of sensitive data such as payment details. Its main job is to encrypt data while it is being stored or transmitted and decrypt it when needed. This ensures the data stays secure and cannot be accessed by unauthorized users. The module uses RSA encryption, which protects data by encrypting it with a public key and decrypting it with a private key. Its design is flexible, supporting both anonymous donors, whose transactions are secured using temporary one-time keys, and registered donors, who use securely stored encrypted credit card information saved Payment Database.

The module includes two main services: the EncryptionService, and the Key Management Service. The Encryption Services handle the core tasks of securing and decoding data, while the Key Management Service manages encryption keys by storing and retrieving them through the paymentKeyRepository, which connects to a MongoDB-based Key Database. This centralized storage system avoids the need to generate a new key for every transaction, which reduces system load and improves scalability.

However, storing encryption keys in a database does come with some risks. If the Key Database is breached, attackers could access the stored keys, potentially compromising all encrypted data. There is also the risk of someone intercepting a key during retrieval or use, particularly during operations like database migrations. To reduce these risks, the system uses strict access controls so that only authorized services can access the keys. Additionally, key exchanges are encrypted to protect them during transfer, and the cryptographic operations are kept separate from other parts of the system. However, securing keys over the long term is still a challenge. For instance, if the database is moved or compromised, it can be difficult to locate and secure the keys.

The Encryption/Decryption module is designed to balance performance and security. By managing keys centrally, the system avoids the need to generate millions of keys for individual transactions, which improves efficiency, especially in high-transaction environments. However, this approach requires careful monitoring of the database and strict policies to ensure key security. Overall, this design allows the system to process transactions safely while remaining scalable and reliable.

d) Email utility module

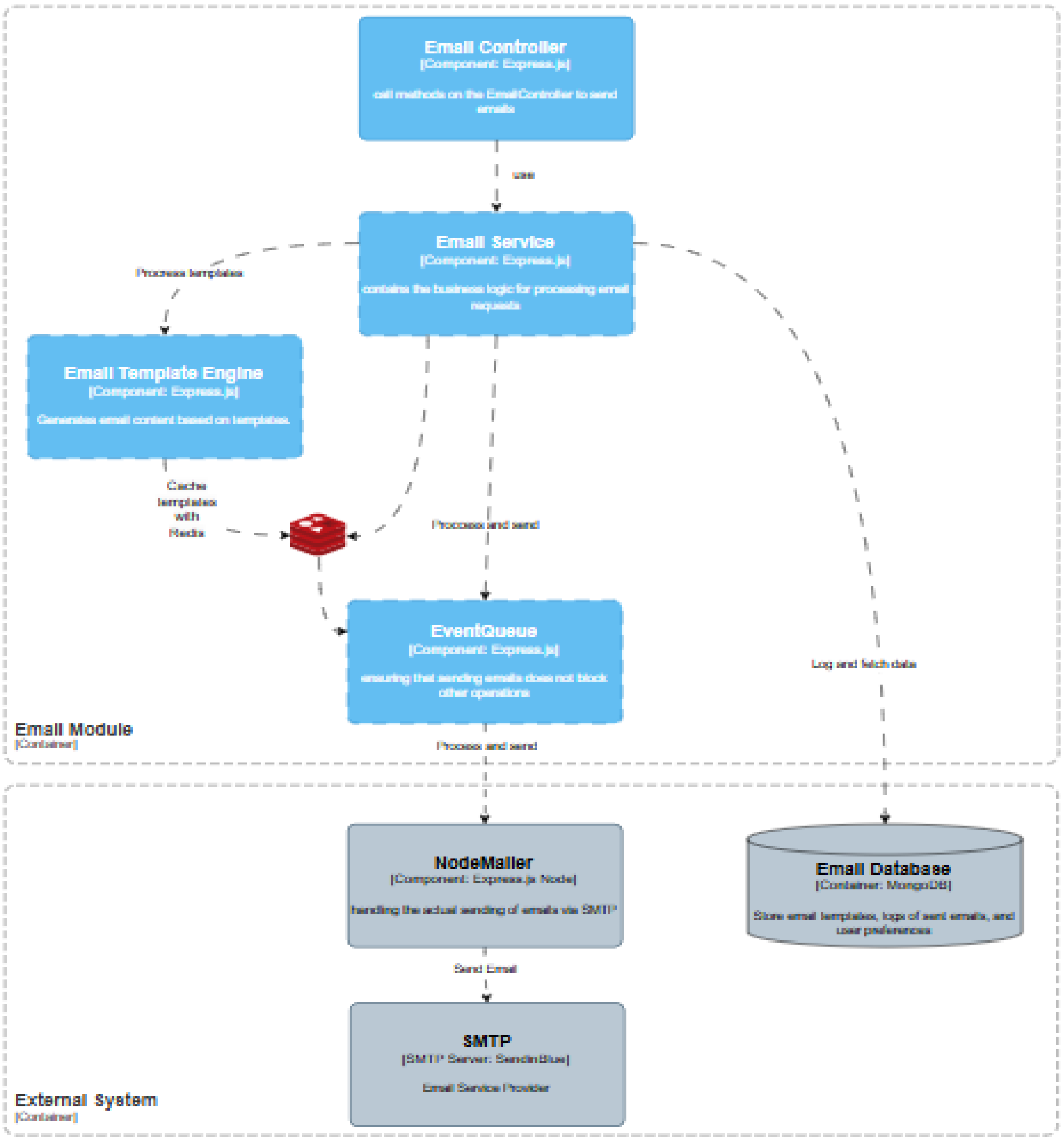

Email Controller
[Component: Express.js]
call methods on the EmailController to send emails
use
Email Service
[Component: Express.js]
contains the business logic for processing email requests
Process templates
Email Template Engine
[Component: Express.js]
Generates email content based on templates
Cache templates with Redis
Process and send
EventQueue
[Component: Express.js]
ensuring that sending emails does not block other operations
Log and fetch data
Email Module
[Container]
Process and send
NodeMailer
[Component: Express.js Node]
handling the actual sending of emails via SMTP
Email Database
[Container: MongoDB]
Store email templates, logs of sent emails, and user preferences
Send Email
SMTP
[SMTP Server: SendinBlue]
Email Service Provider
External System
[Container]

Figure 14 19:Email Module

The Email Module is responsible for email sending tasks. It ensures seamless communication between the platform and app users. It is designed to fit the needs of the Charitan platform: scalable, maintainable, and efficient. [5, 6, 7]

Key Features of Email Module:

1. This module send email to user users via email for purposes such as donation confirmations, notifications for new projects, account creation confirmations, project create confirmations.
2. The Email Template Engine generates dynamic emails based on given, reusable templates.
3. EventQueue ensures that email sending operations do not block other processes and further enhances system responsiveness
4. Redis is used to cache frequently used email templates, reducing latency, and improving performance.
5. All sent emails are saved to the Email Database for supporting, auditing, and debugging purposes.

Email Module Components:

- Email Controller:
    a. Entry point for email module to call methods in Email Service to process and send emails.
- Email Service:
    a. Provides the business logic to process email requests.
    b. Interacts with the Email Template Engine, EventQueue, and NodeMailer.
- Email Template Engine:
    a. Provides the generation of email content by a predefined template
    b. Supports dynamic content insertion to personalize the sending of emails.
    c. Uses Redis to cache templates for quick access.
- EventQueue:
    a. Ensures all emails are sent at various times without being blocked.
    b. Avoids blocking operations in sending emails.
- NodeMailer:
    a. Performs the actual sending of emails via the SMTP server.
- Database:
    a. Stores email templates, sent emails log, and user preferences.
- SMTP Server:
    a. External service SendinBlue for delivering e-mails.

Architecture Rationale:

- **Maintainability**
    - The module is designed in a way that separation of concerns is clear, hence making it easy to update and test.
    - Reusable templates ensure consistency and reduce duplication.
- **Extensibility:**
    - New email templates and functionalities can be added without affecting the existing ones.
    - The design allows integration with other services, for example, the Notification Module
- **Resilience:**
    - EventQueue ensures that failures in sending emails do not impact other operations.
    - Logs in the Email Database provide a fallback for retrying failed emails.
- **Scalability:**
    - For maintaining a huge number of e-mails, this module relies on Redis caching and also performs asynchronously.
- **Performance:**
    - Caching with Redis reduces latency in generating email templates.

e) Payment module

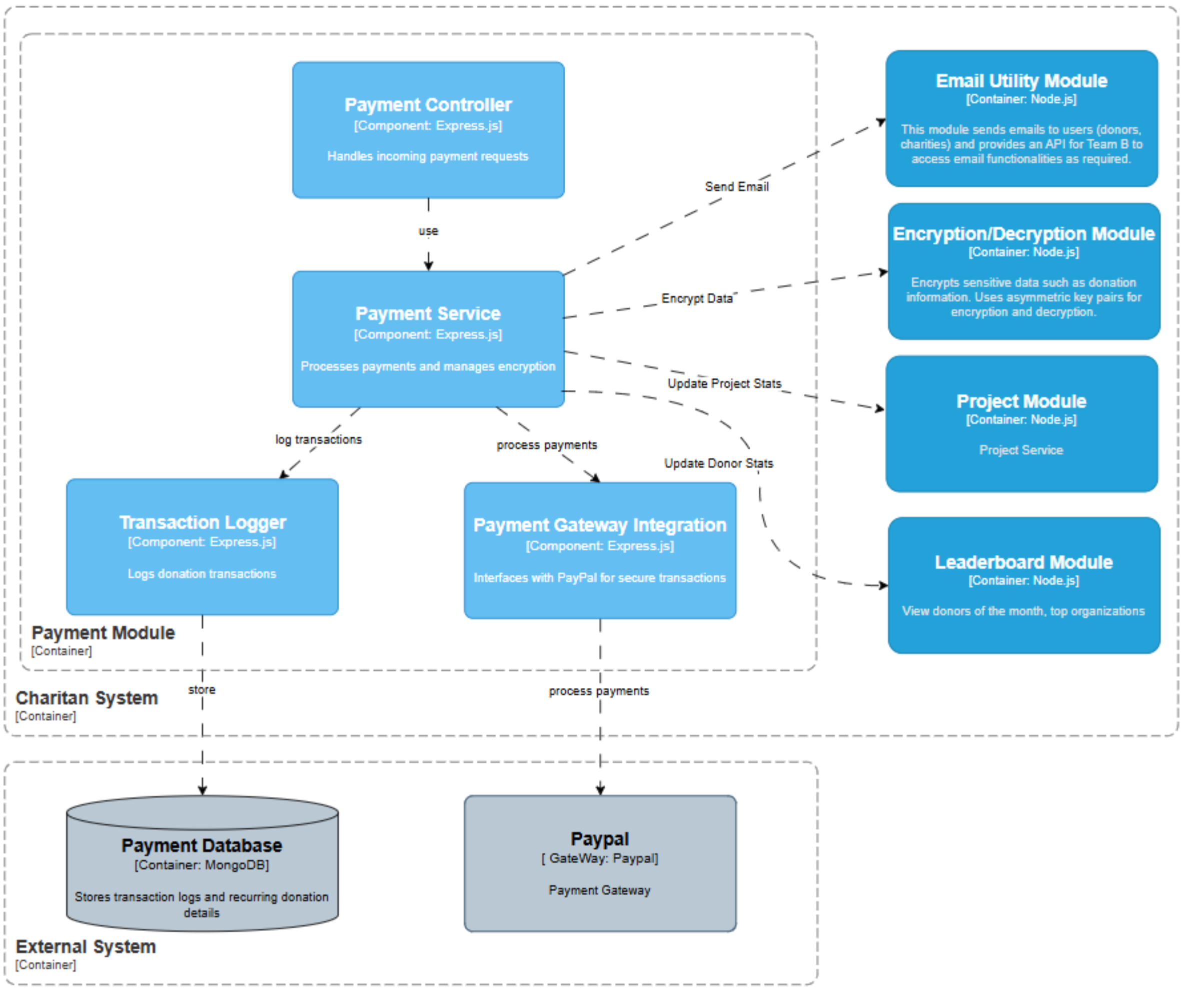


Figure 15 20:Payment Module

The Payment Module is responsible for secure donation processing, recurring payments handling, and interaction with the external payment gateway. [8, 9, 10]

Key Features of Payment Module:

1. Encrypts sensitive payment data such as card number, cvv, and dates using Encryption Module using RSA encryption.
2. Automates monthly donations, processed on the 15th of each month.
3. Allows donors to donate without creating an account.
4. Logs all transactions in the payment database for auditing and reporting.
5. Sends donation confirmation emails via the Email Module.
6. Connect to the Project Module for updating project details.
7. Update donor statistics in the Leaderboard Module.

Payment Module Components:

- Payment Controller:
    a. Entry point for Payment module to call methods in Payment Service.
- Payment Service:
    a. Provides the business logic for processing payments.
    b. Interacts with the Project Module, LeaderBoard Module, Email Module, and Encryption Module.
- Email Template Engine:
    a. Provides the generation of email content by a predefined template

   b. Supports dynamic content insertion to personalize the sending of emails.
   c. Uses Redis to cache templates for quick access.
- Payment GateWay Intergration:
   a. Manages external payment processing via PayPal's sandbox.
- Transaction Logger:
   a. Logs donation transactions in the Payment Database for auditing and reporting.

Related Module:

- Encryption/Decryption Module:
   a. Secures sensitive payment data using RSA encryption.
   b. Decrypts data when necessary for processing.
- Email Module:
   a. Send donation confirmations and failure notifications
- Email Template Engine:
   a. Provides the generation of email content by a predefined template
   b. Supports dynamic content insertion to personalize the sending of emails.
   c. Uses Redis to cache templates for quick access.
- Project Module:
   a. Updates the current amount of the project.

Architecture Rationale:

- **Maintainability**
   - The module is designed in a way that separation of concerns is clear, hence making it easy to update and test.
   - Reusable templates ensure consistency and reduce duplication.
- **Extensibility:**
   - Additional payment gateways can be added without disrupting existing functionality.
   - The design allows integration with other services, for example, the Notification Module
- **Resilience:**
   - All transactions are logged for auditing and recovery.
   - Handles payment failures gracefully with retry mechanisms and email notifications.
- **Scalability:**
   - Asynchronous processing allows the module to handle large volumes of donations.
- **Security:**
   - Sensitive information is encrypted by RSA encryption
   - Follows PCI DSS standards for payment processing.

f) Donation module

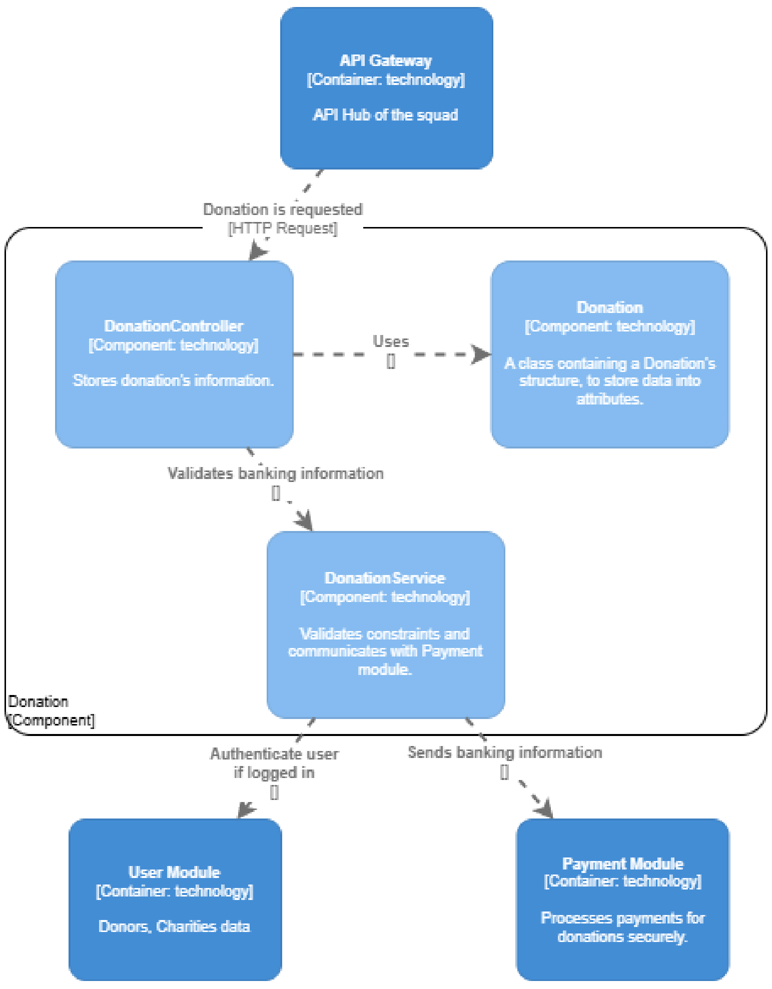


Figure 16 21. Donation module component diagram

The Donation Module acts as a broker between the frontend and complicated banking process. Specifically, it stores user submitted information about the donation and validates its authenticity before proceeding with the payment process in the backend. This module is necessary to prevent information fraud from the start of the donation process. In conjunction with this purpose, three components exist in the Donation Module.

DonationController: A component that stores the donor's donation information from the web application. This information consists of their name (if they are donating as a guest), donated value, charity name, and project.

DonationService: A component that verifies the validity of the charity, its project, the donated value and uses the Payment Module to proceed with the donation's banking process.

User Module: The User Module will be needed and contacted by the DonationService if the donor decides to log in to their account to donate towards the project; this module will authenticate the log using Team B's authentication API, and store the user's donated value in their profile.

Architecture rationale:

- **Maintainability:** Each component has its clear responsibilities, simplifying the debugging and updating process to those respective components. For instance, all related Donation features, from storing Donation history and processing it are implemented in one module, allowing the ease of updates, and fixes to remain within this module.

- **Extensibility:** New features implementation to the Donation Module are isolated from other modules, therefore reduces impact on the overall system aside from its dependencies.
- **Resilience:** Donation related information is processed by the DonationService to ensure the data traveling to other dependencies will not cause invalidity. In situations where there are issues, the Donation Module will not affect other functionality and remains unaffected by external modules due to its closed environment.
- **Scalability:** Due to using other components to handle complicated processes such as authentication in User Module and payment processing to Payment Module, it reduces module computational load and enables scaling.
- **Security:** Donation Module only handles the storage of Donation-related information, leaving the important security features to be accessed by the Payment Module if requested, and remains isolated from other modules. Therefore, limiting vulnerabilities to its own module, simplifying the process to develop countermeasures against exploits.

g) Database

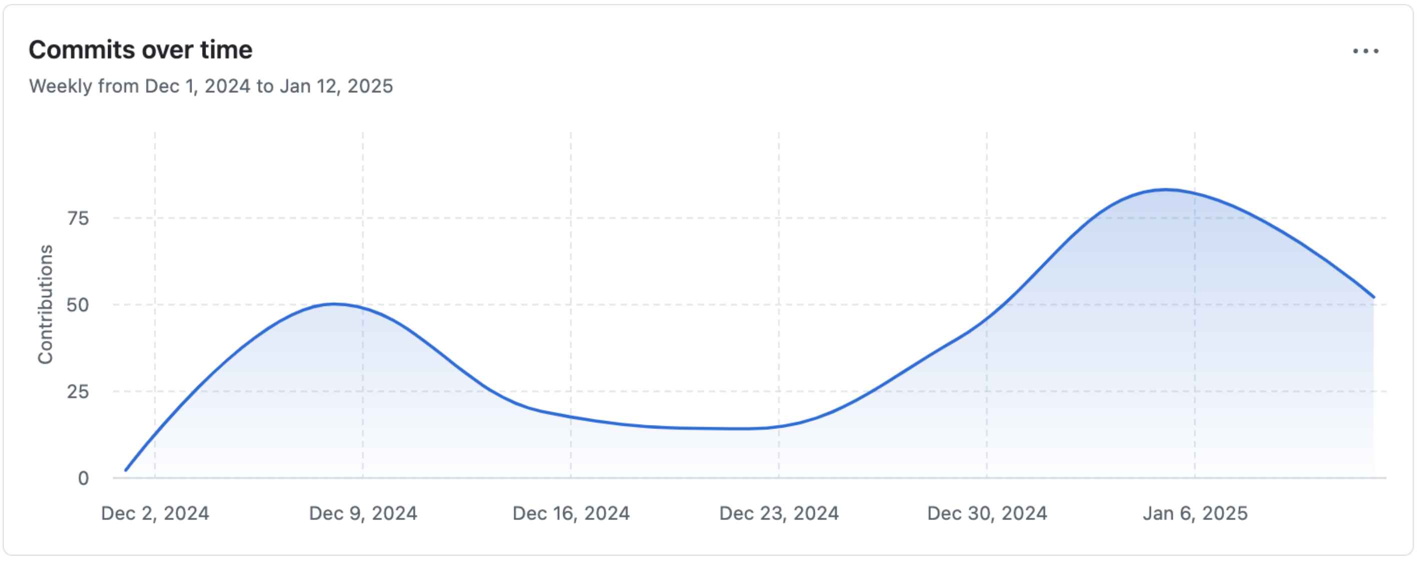
The main data storage technology the squad has decided to use is MongoDB, a NoSQL database. With the use of Javascript, consistently used throughout the MEN stack, this choice allows ease of maintenance without using multiple languages across systems. Moreover, using NoSQL allows the use of dynamic structured data, it is beneficial in cases where certain data fields do not need to exist; for instance, guest user information in the transaction documents, which does not need to be included in the database. In addition, a supporting database for image storage called Cloudinary was used. The extra Cloudinary database takes off the load for MongoDB to encode and store media files, and provides the team with a URL to render the image based of what users have uploaded.

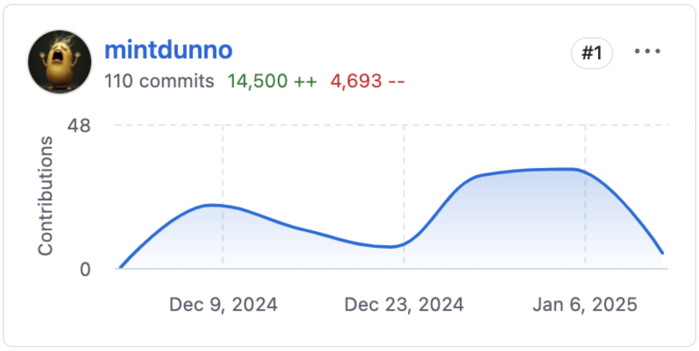
For maintenance and ease of use, Charitan implements a database on each module, where each database stores its encoded schema. This implementation ensures Charitan banking information stored by users from compromises and exploits in cases of data breaches in other databases. Furthermore, this allows each module’s development independence from affecting other modules, providing extensibility and resilience to security features.

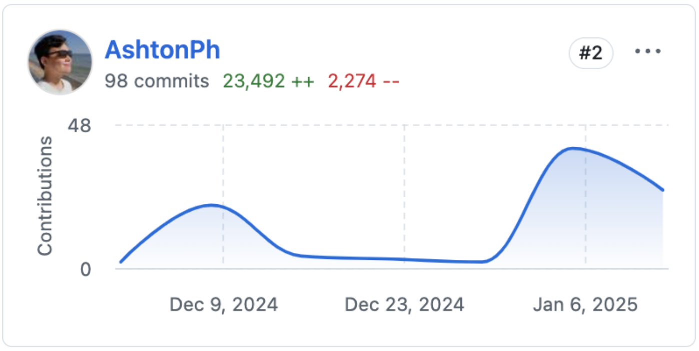
Redis is also implemented to cache frequent accessed Projects for resilience and user experience purposes. The service provides information access through cache data, where it reduces the reads of unchanging data onto MongoDB. Despite Redis usefulness, this service requires a setup to ensure that it does not cause replication or communication issues with the database.

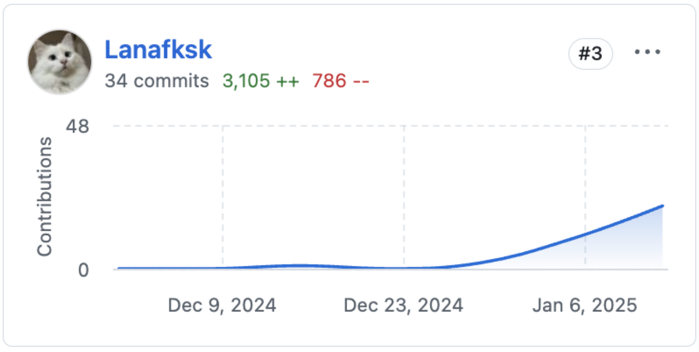
With two separate databases, and Redis to cache data, the Charitan data storage system can be flexible to store data while maintaining security from data breaches and resilience to faulty downtimes. Meanwhile, Redis’ implementations improve the user experience by ensuring that data is stored in the local cache to reduce load times by the system.

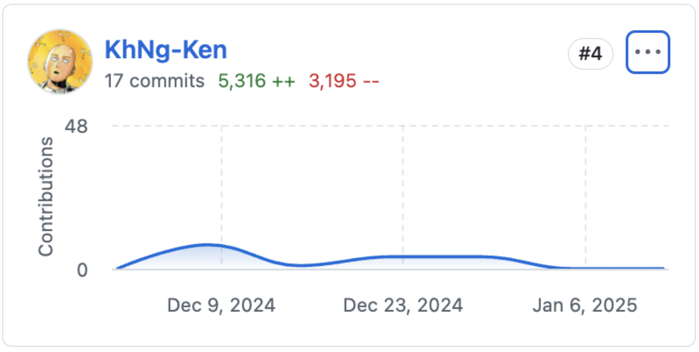
IV. Github Contributions


Commits over time
Weekly from Dec 1, 2024 to Jan 12, 2025
Contributions
75
50
25
0
Dec 2, 2024
Dec 9, 2024
Dec 16, 2024
Dec 23, 2024
Dec 30, 2024
Jan 6, 2025

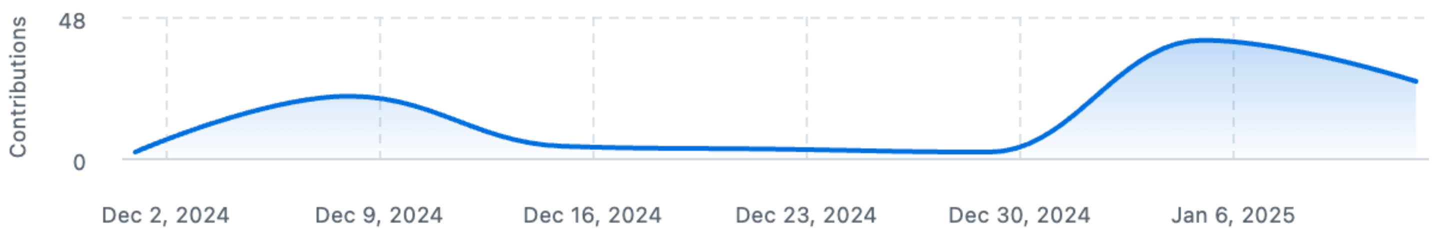

mintdunno
110 commits 14,500 ++ 4,693 --
#1
Contributions
48
0
Dec 9, 2024
Dec 23, 2024
Jan 6, 2025


AshtonPh
98 commits 23,492 ++ 2,274 --
#2
Contributions
48
0
Dec 9, 2024
Dec 23, 2024
Jan 6, 2025

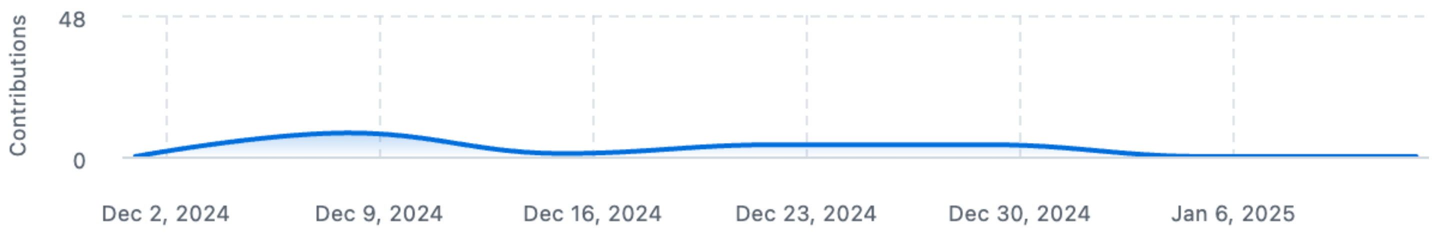

Lanafksk
34 commits 3,105 ++ 786 --
#3
Contributions
48
0
Dec 9, 2024
Dec 23, 2024
Jan 6, 2025


KhNg-Ken
17 commits 5,316 ++ 3,195 --
#4
Contributions
48
0
Dec 9, 2024
Dec 23, 2024
Jan 6, 2025

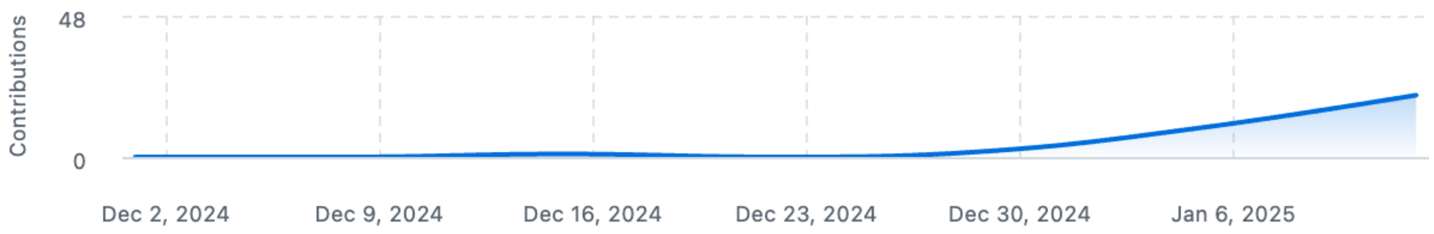
AshtonPh's Commits



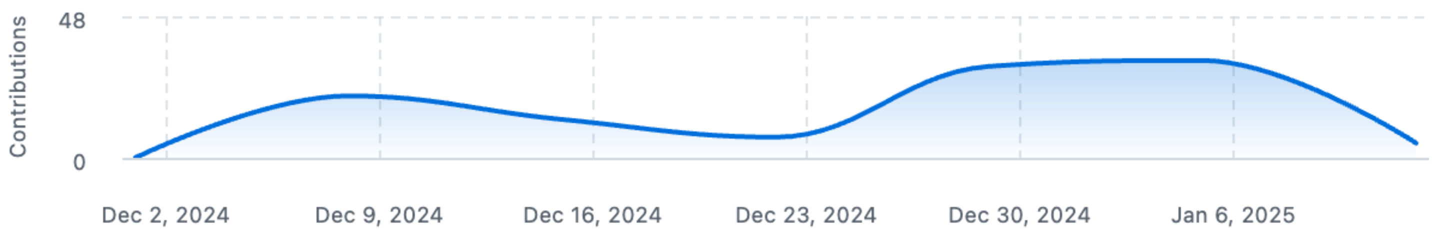
KhNg-Ken's Commits



Lanafksk's Commits



mintdunno's Commits



V. Architecture Rationale

1/ Maintainability: Enables the system to be easily understood, updated, and tested independently by current and future Development Teams.

The architecture of Charitan optimizes maintainability through a modular monolith approach by encapsulating major functions such as Donations, Project Dashboards, Payments, User Management, and Leaderboards within their own modules. This architecture ensures that each module remains independent, allowing issues to be addressed without impacting unrelated parts of the system. For example, issues or bugs in the Payment Module will not affect the Donation Module. This separation also simplifies debugging and testing, thereby enhancing the platform’s reliability.

Reusable components play a significant role in improving the maintainability of the system by reducing code duplication and making the system more consistent, which also makes it easier for developers to understand. The APIs within the modules are structured to ensure they provide clear interfaces for data exchange. This reduces the complexity associated with implementing changes, thereby allowing updates to the internal logic of one component without influencing the interactions between other modules. Furthermore, the modular monolith architecture employed ensures that new features can be added with minimal refactoring of existing code, thanks to its specialized modules. These clear boundaries between modules support developers and teammates in collaborating on different components concurrently without interfering with each other's work.

2/ Extensibility: Enables new features to be added with minimal impact on existing components. This approach allows for future enhancements and adaptations to changing user needs or market demands.

The extensibility of Charitan is demonstrated in a Modular Monolith design ensuring each module, and component has a definitive responsibility, allowing the development of new functions to be localized respectively. For instance, in situations where a SMS notification function is required, developers can determine this implementation to produce a new module of use to the Notification Module.

Furthermore, these new functions can be added with minimal effects towards components in the system as each component is independent from other components. For example, adding a function to Donation will have no effects on Project, Email or Notification modules since these modules encapsulated, and isolated from others. In similar cases, this allows Charitan to integrate new features, where debugging can be simplified to each module instead of the whole system.

In addition, services can be added or replaced without changing the core logic of handling its interactions, such as switching PayPal with Stripe to handle payment during donations. In turn, reusing the templates provided in each module, new features that extend off existing features can reduce code duplication, and reduce development time.

However, issues can arise for modules with extensive reliability to its dependencies, such as User Module relating to Donation and Payment Module. In these cases, updates can be prone to data flow issues in the data transmission between modules. To handle this, developers will have to track changes and ensure the updates do not compromise the data flow of their function updates' related component.

Overall, extensibility is supported through a modular monolith design through the presence of modules that simplifies the development process to be isolated within the module and its related dependencies.

3/ Resilience: The system's ability to continue functioning correctly in the face of unexpected conditions, such as component failures or high loads.

The system employs a blended architectural approach, incorporating both monolithic and n-tier elements. This hybrid design presents a mix of advantages and challenges when it comes to system resilience.

The monolithic structure provides the benefits of a tightly integrated, unified system, which can simplify development and deployment, especially for a platform focused on a specific domain like charitable fundraising. The modular, n-tier components, on the other hand, allow for a more scalable and maintainable design, as individual parts of the system can be updated or replaced without affecting the whole application. [11]

The centralized API Gateway serves as a single-entry point for all backend requests, streamlining management and monitoring of API traffic. This centralized approach can enhance security, rate limiting, and load balancing. Additionally, the separation of concerns between the front-end and back-end components, along with the specialized modules like Notification, Project, and Payment, promotes maintainability and evolvability.

However, the monolithic nature of the system also introduces potential weaknesses. A single point of failure within the monolith could bring down the entire application, impacting all users and functionality. The tight coupling between components may make it more challenging to isolate and replace individual parts of the system, hindering the ability to quickly address issues or scale specific components. [12]

As the system grows in complexity and user base, scaling the monolithic application may become more difficult, as certain components may require more resources than others, making efficient scaling a challenge. The reliance on external services, such as SMTP and PayPal, introduces dependencies that could impact the overall resilience of the system, requiring careful management and monitoring of these third-party integrations.

4/ Scalability: The capacity of a system to handle increased load or user demand without sacrificing performance

The scalability of the Charitan system is achieved by a modular monolith architecture combined with thoughtfully designed components and external integrations. Each module, for example, the Notification Module, Email Module, Payment Module, and Donation Module, has clear boundaries within which it operates, thus allowing independent scaling (Micro Service). The API Gateway is at the centre, ensuring load balancing and efficient routing to the backend modules, hence managing a high volume of traffic. MongoDB supports horizontal sharding for database scalability in case of increasing data loads, while Redis caching optimizes performance by reducing latency for frequently accessed data such as email templates and project details.

5/ Security: The measures taken to protect a system and its users from unauthorized access, data breaches, and other threats.

The security system of the Charitan platform is designed to protect sensitive user data and ensure safe operations across all components. The platform uses RSA encryption, a reliable asymmetric encryption method, to secure critical information such as payment credentials. This method relies on a public-private key pair, ensuring that only authorized users can access or decrypt sensitive data [13]. To store and manage these keys, a centralized Key Store works alongside the Key Management Service, which applies strict policies for retrieving and storing cryptographic keys. Additionally, the key database is encrypted, and regular audits and monitoring processes are used to identify any suspicious activity [14].

For managing state on the frontend, the platform uses React's useState hook, a straightforward and efficient method for handling localized updates without requiring complex libraries like Redux. While this simplifies state management, it's essential to follow secure coding practices to prevent unauthorized changes or data manipulation during user interactions.

This design aligns with the Open-Closed Principle (OCP), which ensures that the system can be extended to include new security measures or encryption methods without requiring significant changes to its existing components. For instance, additional encryption layers or advanced security techniques can be added to the Key Store or Key Management Service as needed, supporting the system's ability to adapt to future requirements [15].

However, using a centralized key database comes with some risks. While it simplifies the process of managing encryption keys and avoids the complexity of creating a unique key for every transaction, it introduces vulnerabilities. If the database is compromised, attackers could gain access to all encryption keys, exposing sensitive information. There is also a risk of "man-in-the-middle" attacks during key retrieval, where an unauthorized entity could intercept or alter the key being transmitted. Such risks are especially concerning during database migrations or system updates, where security may be more difficult to enforce. This centralized approach makes the system easier to scale and maintain because it avoids the burden of managing separate keys for each transaction.

By following security best practices and principles like OCP, the platform ensures that it can protect sensitive data while also remaining flexible to future improvements. This approach balances security, scalability, and maintainability, creating a solid foundation for the platform's growth and reliability.

6/ Performance: Performance relates to how quickly and efficiently a system responds to user requests and processes data.

The system's ability to respond quickly and efficiently to user requests is crucial for providing a seamless and satisfactory user experience.
One of the core performance advantages of the platform's architectural approach is the separation of concerns between the front-end and back-end components. By encapsulating the user interface, navigation, and state management in the front-end modules, the system can optimize the responsiveness of these critical user-facing elements.

The modular nature of the n-tier architecture also contributes to the platform's performance characteristics. By dividing the backend into specialized modules like Notification, Project, and Payment, the system can scale these components independently based on their specific resource requirements. This allows the platform to allocate resources more efficiently, ensuring that high-traffic or computationally intensive modules receive the necessary processing power and resources to maintain optimal performance [12].

Furthermore, the inclusion of caching mechanisms, such as in-memory caching or content delivery networks, can significantly improve the platform's performance by reducing the number of requests that need to be processed by the backend. By storing frequently accessed data or pre-rendered content closer to the end-users, the system can provide faster responses and reduce the load on the core backend services.
However, the platform's performance profile is not without its challenges. The integration with external services, such as SMTP and PayPal, introduces potential performance bottlenecks that are outside the direct control of the platform's architecture. Ensuring the reliability and responsiveness of these third-party integrations is crucial for maintaining overall system performance [12].

Additionally, as the platform's user base and data volumes grow, the monolithic nature of the backend may become a limiting factor. The tight coupling between components within the monolith could make it more difficult to scale specific parts of the system independently, leading to potential performance degradation.
To address these performance challenges, the platform's architecture incorporates strategies such as caching, load balancing, asynchronous processing, and the use of content delivery networks. Continuous monitoring and optimization of the system's performance metrics, as well as proactive planning for future scaling requirements, will be essential for maintaining the platform's responsiveness and efficiency as it evolves and grows.

I. Conclusion

The first milestone of the Charitan Donation Platform project establishes a robust architectural foundation through our architecture design. By carefully balancing the six critical architectural aspects—maintainability, extensibility, resilience, scalability, security, and performance—we have designed a system that provides clear module boundaries, isolated business domains, and the flexibility to adapt to evolving platform requirements.

As we transition to Milestone 2, our architectural design will be translated into a functional system, implementing core modules, developing foundational APIs, and creating initial features for donors, charities, and administrators. While our design reflects deliberate trade-offs, it provides a sophisticated approach that bridges the complexity of microservices with the simplicity of monolithic deployment, positioning the Charitan Donation Platform for successful development and future growth.

References

[1] D. Shrestha, “Navigating software architecture: evaluating monolithic architectures in modern development,” *Theseus*, 2024. https://www.theseus.fi/handle/10024/858838

[2] “useState – React.” https://react.dev/reference/react/useState

[3] “React Redux | React Redux.” https://react-redux.js.org/

[4] “Poimandres documentation.” https://zustand.docs.pmnd.rs/

[5] R. Sypchenko, “How to Send Email Using Node.js - Roman Sypchenko - Medium,” *Medium*, Dec. 25, 2023. [Online]. Available: https://medium.com/@r.sipchenko/how-to-send-email-using-node-js-40fe5708607c

[6] Morikun, “Gửi mail với NodeMailer trong NodeJS,” *Viblo*, Nov. 03, 2023. https://viblo.asia/p/gui-mail-voi-nodemailer-trong-nodejs-rQOvPNqjeYj

[7] Šarūnas, “Send Emails with Node.js: API and Nodemailer (SMTP),” *MailerSend*, Nov. 22, 2024. https://www.mailersend.com/blog/send-email-nodejs

[8] N. Prince, “Add payments feature in your application using Node.js/Express and Stripe,” *DEV Community*, May 09, 2023. https://dev.to/nishimweprince/add-payments-feature-in-your-application-using-nodejsexpress-and-stripe-305a#create-a-stripe-account

[9] GeeksforGeeks, “How to Integrate Paypal in Node?,” *GeeksforGeeks*, Mar. 28, 2024. https://www.geeksforgeeks.org/how-to-integrate-paypal-in-node/

[10] Tucker and A. Tucker, “What is RSA Asymmetric Encryption? How Does it Work?,” *SecureW2*, Oct. 01, 2024. https://www.securew2.com/blog/what-is-rsa-asymmetric-encryption

[11] D. Shrestha, “Navigating software architecture: evaluating monolithic architectures in modern development,” *Theseus*, 2024. https://www.theseus.fi/handle/10024/858838

[12] M. Kaloudis, “Evolving Software Architectures from Monolithic Systems to Resilient Microservices: Best Practices, Challenges and Future Trends,” *International Journal of Advanced Computer Science and Applications*, vol. 15, no. 9, Jan. 2024, doi: 10.14569/ijacsa.2024.0150901.

[13] Tucker, “What is RSA Asymmetric Encryption? How Does it Work?” *SecureW2*, Oct. 01, 2024. https://www.securew2.com/blog/what-is-rsa-asymmetric-encryption

[14] J. J, “Types Of Database Encryption: Best Practices For Securing Your Data,” *RedSwitches*, Nov. 06, 2024. https://www.redswitches.com/blog/encrypted-database/

[15] D. Azevedo, “Understanding the Open/Closed Principle (OCP) from SOLID: Keep Code Flexible Yet Stable,” *DEV Community*, Oct. 18, 2024. https://dev.to/dazevedo/understanding-the-openclosed-principle-ocp-from-solid-keep-code-flexible-yet-stable-jo7#:~:text=The%20Open%2FClosed%20Principle%20states%20that%3A%20%E2%80%9CSoftware%20entities%20%28classes%2C,functionality%20to%20be%20added%20wi