EEET2582
Software Engineering: Architecture and Design

Charitan, A Global Charity Donation Platform
Milestone 1

Course Coordinator	Tri Huynh Minh	tri.huynh5@rmit.edu.vn
Group	Trung Le Tran Minh	s3927071@rmit.edu.vn
	Pavel Potemkin	s3963284@rmit.edu.vn
	Saurabh Padmakumar	s3878804@rmit.edu.vn
	Nguyen Do Khoa	s3978796@rmit.edu.vn
	Kien Vo Nguyen	s3938016@rmit.edu.vn

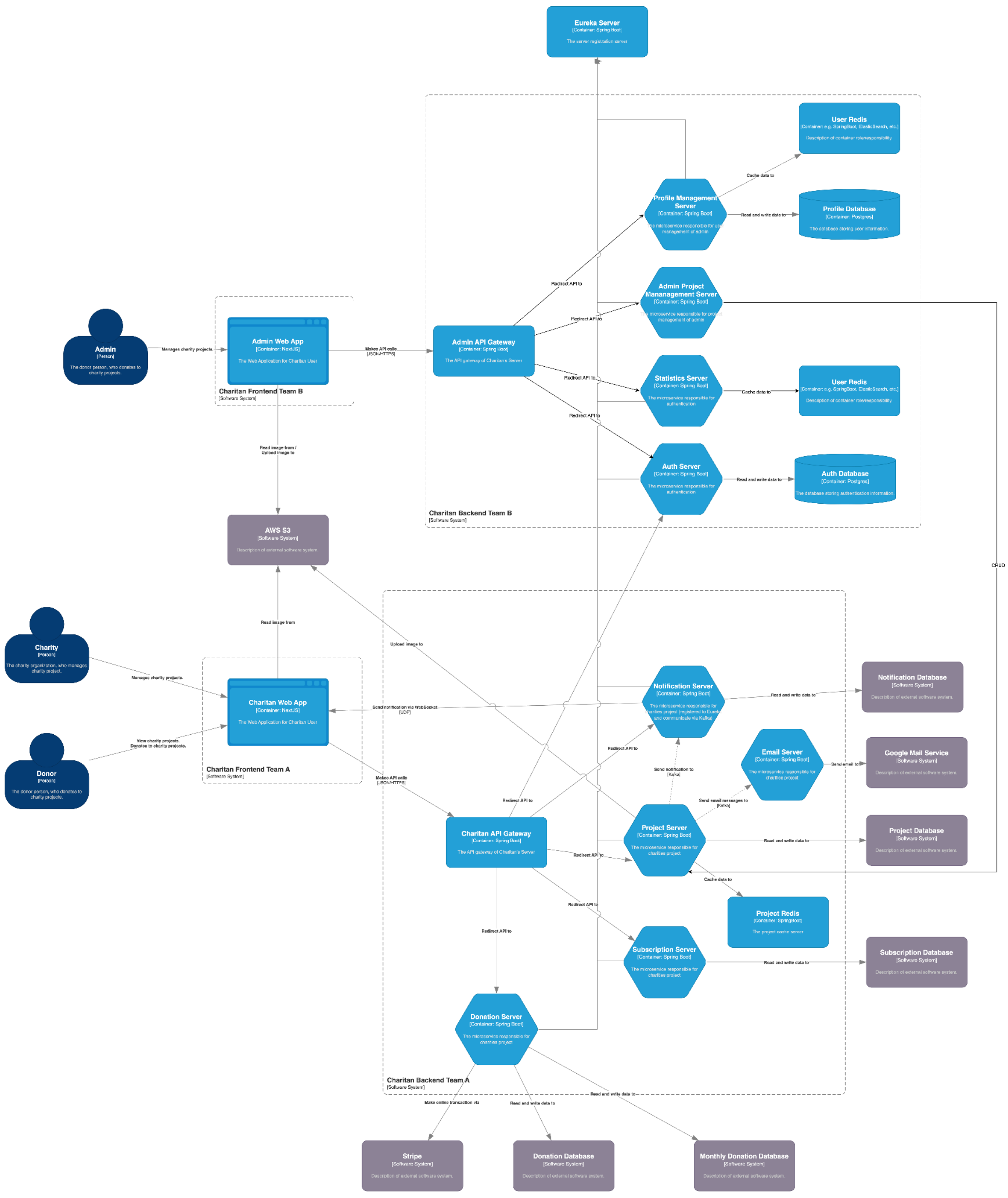
Submission Date Dec 4th, 2024

Table of Contents

Table of Contents 1
Context Diagram 3
I. Data Model 4
I.I Design 4
I.II Ideas and purpose 4
I.II.I User Centric Design 4
I.II.II Role-Based Access Control (RBAC) 5
I.II.III Project Management 5
I.II.IV Advantages 6
I.II.V Drawbacks 7
II. Backend Architectural Justification 8
II.I Introduction 8
II.II Containers 9
II.III API Gateway and Authentication Component 10
II.III.I Background 10
II.III.II Caveats 12
II.III.III Implementation 12
II.IV Statistics Component 16
II.V Profile Management Component 17
II.VI Project Management Component 20
II.VII Design Justification 21
II.VII.I Maintainability 21
II.VII.II Extensibility 21
II.VII.III Resilience 22
II.VII.IV Scalability 22
II.VII.V Security 22
II.VII.VI Performance 23
III. Frontend Architectural Justification 24
III.I Introduction 24
III.II Containers 24
III.II.I Tools and Libraries Used 24
III.II.II API Client 25
III.II.III State Management 26
III.V Form Component 26
III.VI. Table Component 27
III.VII. Chart Component 28
III.VIII. Modal Form Component 29
III.IX. Design Justification 29

III.IX.I. Maintainability 29
III.IX.II. Extensibility 29
III.IX.III. Resilience 29
III.IX.IV. Scalability 30
III.IX.V. Security 30
III.IX.VI. Performance 30
IV. References 31
V. Appendix 33

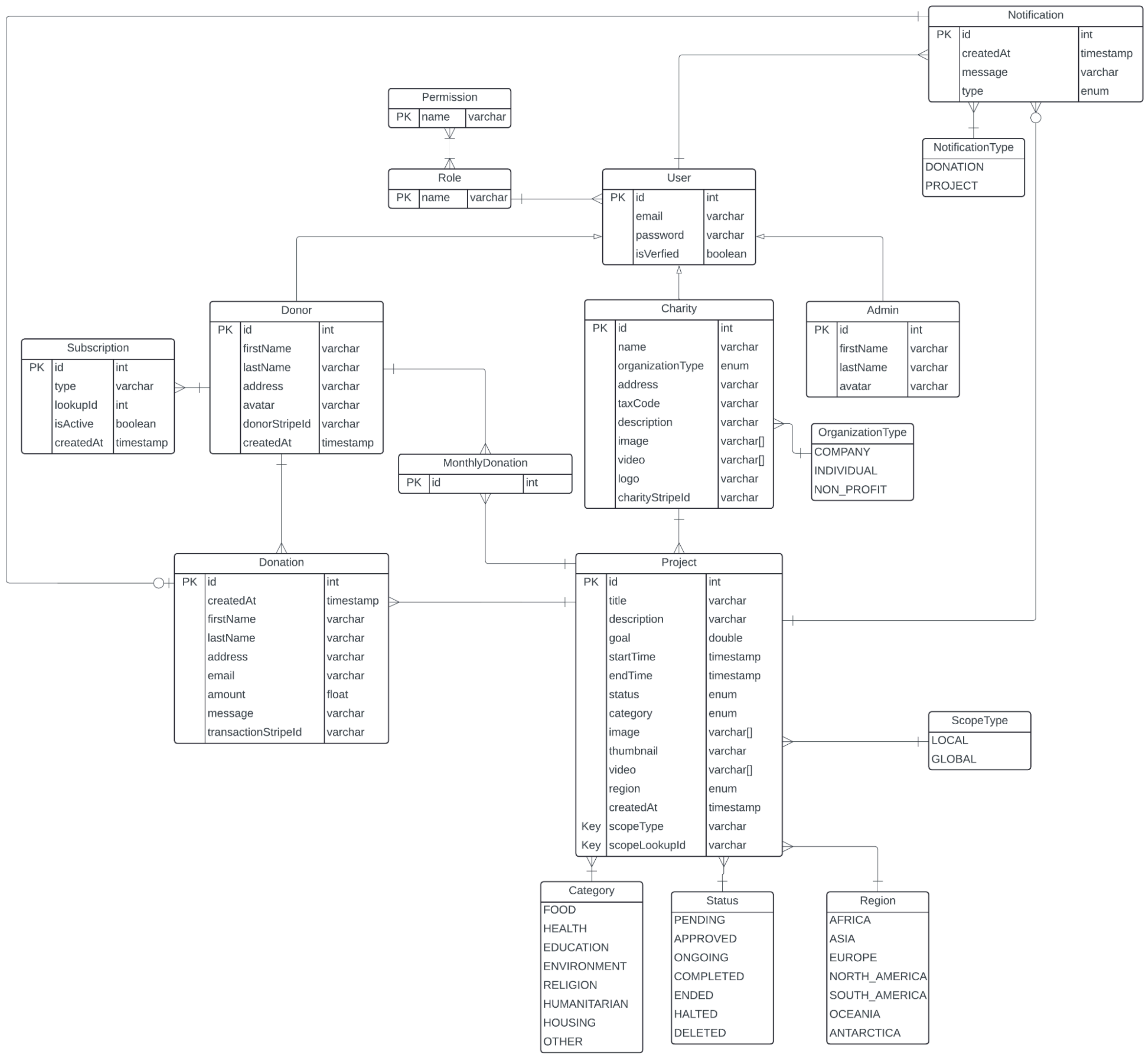
Context Diagram



Eureka Server
[Container: Spring Boot]
The server registration server
User Redis
[Container: e.g. SpringBoot, ElasticSearch, etc.]
Description of container role/responsibility.
Cache data to
Profile Management Server
[Container: Spring Boot]
The microservice responsible for user management of admin
Read and write data to
Profile Database
[Container: Postgres]
The database storing user information.
Redirect API to
Admin Project Mananagement Server
[Container: Spring Boot]
The microservice responsible for project management of admin
Redirect API to
Admin API Gateway
[Container: Spring Boot]
The API gateway of Charitan's Server
Redirect API to
Statistics Server
[Container: Spring Boot]
The microservice responsible for authentication
Cache data to
User Redis
[Container: e.g. SpringBoot, ElasticSearch, etc.]
Description of container role/responsibility.
Redirect API to
Auth Server
[Container: Spring Boot]
The microservice responsible for authentication
Read and write data to
Auth Database
[Container: Postgres]
The database storing authentication information.
Charitan Backend Team B
[Software System]
CRUD
Admin
[Person]
The donor person, who donates to charity projects.
Manages charity projects.
Admin Web App
[Container: NextJS]
The Web Application for Charitan User
Charitan Frontend Team B
[Software System]
Makes API calls
[JSON/HTTPS]
Read image from / Upload image to
AWS S3
[Software System]
Description of external software system.
Read image from
Upload image to
Charity
[Person]
The charity organization, who manages charity project.
Manages charity projects.
Charitan Web App
[Container: NextJS]
The Web Application for Charitan User
Donor
[Person]
The donor person, who donates to charity projects.
View charity projects.
Donates to charity projects.
Charitan Frontend Team A
[Software System]
Send notification via WebSocket
[UDP]
Makes API calls
[JSON/HTTPS]
Notification Server
[Container: Spring Boot]
The microservice responsible for charities project (registered to Eureka and communicate via Kafka)
Read and write data to
Notification Database
[Software System]
Description of external software system.
Redirect API to
Email Server
[Container: Spring Boot]
The microservice responsible for charities project
Send email to
Google Mail Service
[Software System]
Description of external software system.
Send notification to
[Kafka]
Send email messages to
[Kafka]
Redirect API to
Charitan API Gateway
[Container: Spring Boot]
The API gateway of Charitan's Server
Redirect API to
Project Server
[Container: Spring Boot]
The microservice responsible for charities project
Read and write data to
Project Database
[Software System]
Description of external software system.
Cache data to
Project Redis
[Container: SpringBoot]
The project cache server
Redirect API to
Redirect API to
Subscription Server
[Container: Spring Boot]
The microservice responsible for charities project
Read and write data to
Subscription Database
[Software System]
Description of external software system.
Donation Server
[Container: Spring Boot]
The microservice responsible for charities project
Charitan Backend Team A
[Software System]
Make online transaction via
Read and write data to
Read and write data to
Stripe
[Software System]
Description of external software system.
Donation Database
[Software System]
Description of external software system.
Monthly Donation Database
[Software System]
Description of external software system.

I Data Model

I.I Design



Data model for the Charitan system.

To explore the data model interactively, see Appendix A.

I.II Ideas and purpose

I.II.I User Centric Design

The User entity functions as a core base for all types of users which includes Donor, Charity, Admin. Hence, the common feature such as email, password can be controlled from one location, which also ensures consistency and reduces redundancy. The user entity will also include an attribute called isVerified to check if the user has verified their email. Therefore, we use a generalization relationship from donor, charity and admin to user.

The Donor as a child of User has its own attributes such as first name, last name, address, avatar, Stripe ID with address and avatar as two optional fields. The idea of Stripe ID is that when the user successfully creates their account, the user will be assigned to a Stripe customer on our Stripe's system account. Therefore, when a user makes a donation, Stripe will remember the cards owned by that user and let the user reuse them for next donations, and those cards will also be stored securely by Stripe.

Verified organizations or individuals can create and manage their accounts on the platform using Charity entities. The charity needs to provide their name, description, address, tax code, images, videos, logo as three optional fields. The charity will also be given a Stripe ID allowing the charity to receive donations directly and securely through Stripe. Besides, each charity will have a type for their organization which are company, individual, and non profit.

I.II.II Role-Based Access Control (RBAC)

RBAC is used to link certain to enhance permissions to the system's roles in order to offer fine-grained access. Therefore, this enhances the security of the application by making sure that users with specific roles and permissions can access resources at the proper level [1].

There are three main roles in our system according to the requirements, which are admin, donor, and charity. A role can be assigned to many users, but a user can only have one role.

On the other hand, permissions are available within the system, such as `CREATE_USER`, `CREATE_PROJECT`, `GET_DONATION`, etc. A permission can be assigned to many roles, and one role can have a list of different permissions.

I.II.III Project Management

The project management section is essential since it lets the charity plan, coordinate, and monitor their fundraising progress. The charity needs to provide relevant information when creating the project including title, description, start time, end time. Media fields like images, thumbnail, videos promote donor engagement by improving visibility and storytelling.

Each project will be assigned with one category enumeration value, and one category enumeration value can be assigned by many projects.

Similarly, we implement a region enum class to categorize global projects based on geographical location (continent). For a local project (country-level), we will use an external country API to get the country's code and store it into the project object. For example:

- If a project is hosted in Asia, its `scopeType` will be Global, and its `scopeLookUpID` will be ASIA.
- if a project's base is in Vietnam, its `scopeType` will be Local, and its `scopeLookUpID` will be VNM.

There is an enum class called status to address the status of the project, which will be managed by the system when the project goes through different phases. At first, the project's status will be pending on creation by the charity; then, when the admin approves the project, the status will be approved. When the project opens for receiving donations based on the start time, the project will switch to ongoing, and once the goal is reached, the project will be considered to be completed, but still open for extra donation. The project truly ends once it passes the end time. For projects that are deleted or halted, it will also be set to relevant statuses.

I.II.IV Advantages

Maintainability

Having User as the parent class, all shared information such as email, password, and isVerified are defined and managed in one location, minimizing duplicated data and guarantee consistency across user types. This shows hierarchical relationships between classes more clearly, and provides an easy way to add extra details to defined classes later in development cycles. For instance, if users' phone numbers are required to be stored later on, we only need to add a new field to the parent class User, and this would save deployment time and reduce errors.

Moreover, utilizing enum for Status, Category, Region, the class could ensure data consistency and reduce errors.

Stripe Integration

The system has utilized Stripe for secure and reusable payment management by having Stripe ID for both Donor and Charity entities:

- Donors can safely use their card to make donations as the card information is stored securely by Stripe, which lets them have a better future donation experience.

- Charities can also set their card for money receiving, and keep track of donations.

Regional and Country Context

Users can easily filter projects based on geographical location with the help of Region and Country entities.

Status Tracking

A project's lifetime can be clearly defined with the help of the Status object.

Transparency and accountability are promoted by donors' capacity to monitor the status of the initiatives they fund.

Security

RBAC provides fine-grained access by connecting specific enhanced permissions to the roles in the system. By ensuring that users with particular roles and permissions may access resources at the appropriate level, this improves the application's security.

Performance

Enums for regions or statuses can help reduce storage overhead and enhance query efficiency.

I.II.V Drawbacks

Extensibility

Categories as enum can reduce the flexibility and maintainability of the system. For example, adding or deleting a category, we need to modify the inner code of the system.

Mitigation: consider switching from enum to a class so that the admin can directly manage categories via a dashboard.

External dependency / Resilience

The system heavily depends on the external country API for obtaining the country codes, and from the code back to the country or even checking the region that country belongs to. Therefore, if the API is down or slow, the system’s functionality relies on it will fail or be delayed as well.

Mitigation: cache country codes and its region to our system’s database or memory, or we can have a backup external country API.

II Backend Architectural Justification

II.I Introduction

For the design of the Charitan Donation Platform’s backend, the decision to use a microservice architecture is brought forward by both technical and organizational considerations. As the microservices provide a modular approach to application development, the justification aligns with Conway's Law which states that “organizations design systems that mirror their communication structure”.

Team B have divided the platform into five core services - API Gateway, Statistics, Profile Management, Authentication, and Admin Project Management. The justification for this division is that in the case of Charitan, the platform’s distinct roles and functionalities can be directly mapped to an independent service. This alignment allows the development teams to work autonomously on their respective services while enabling focused development within the team.

In summary, there are five main services that the team will work on:

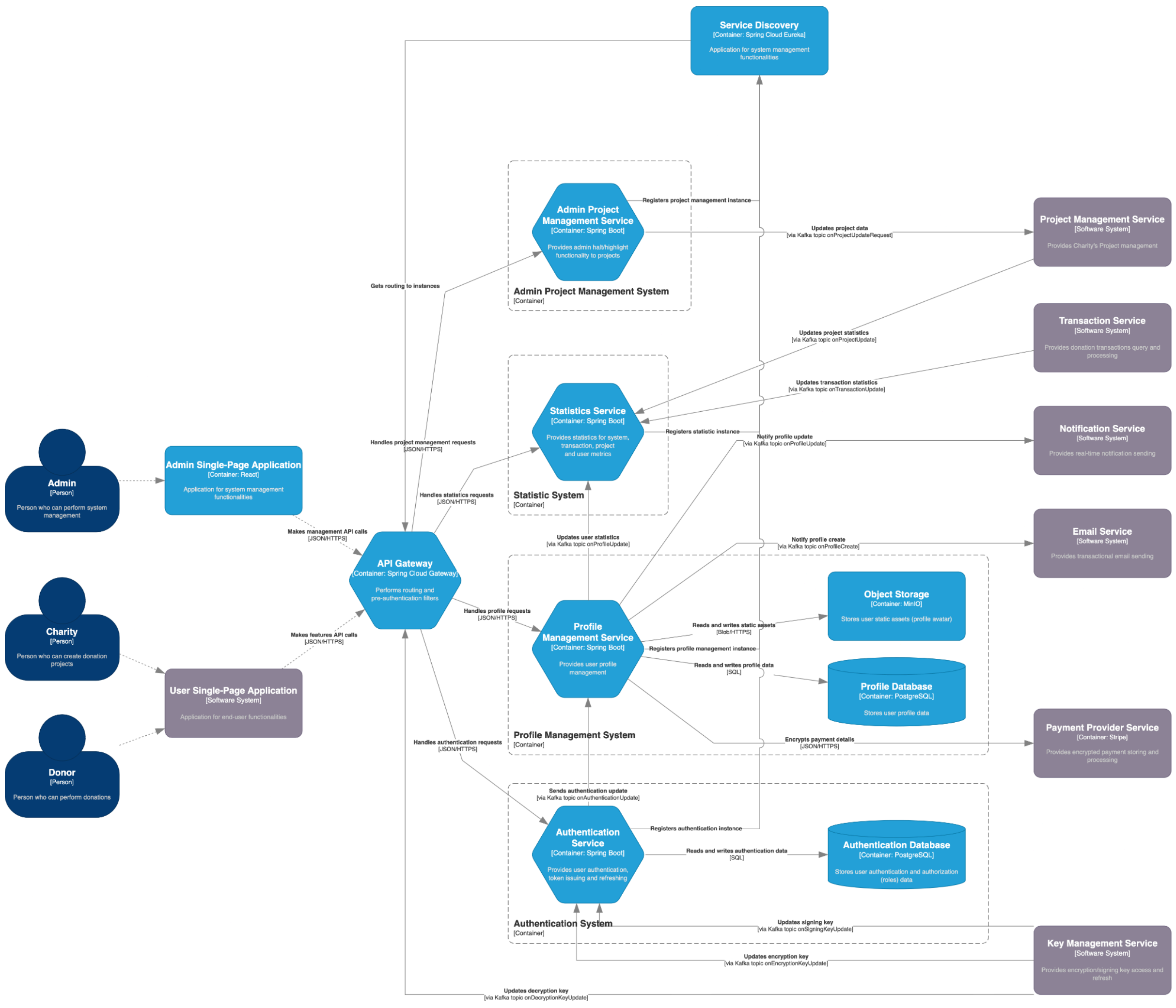
1. API Gateway
2. Authentication Service
3. Profile Management Service
4. Statistics Service
5. Admin Project Management Service

In implementation, data of communications between internal services and external interfaces are designed to be separated using data transfer objects (DTO) and interfaces to prevent users and services access to unwanted data and reduce exposure of internal implementation details, that includes internal and external DTOs.

Internal DTOs are used for communication within the boundaries of a service which are not exposed to external systems or clients. This will encapsulate data used within the backend or between internal layers of the system and contain all necessary fields required for backend operations, including sensitive data not meant for external users.

External DTOs are used for data transfer between the system and external clients, such as frontend applications or third-party services. Using external DTO will enhance the security of the system by exposing only the data that is safe and necessary for external users to see or manipulate.

II.II Containers



Service Discovery
[Container: Spring Cloud Eureka]
Application for system management functionalities
Admin Project Management Service
[Container: Spring Boot]
Provides admin halt/highlight functionality to projects
Admin Project Management System
[Container]
Registers project management instance
Updates project data
[via Kafka topic onProjectUpdateRequest]
Project Management Service
[Software System]
Provides Charity's Project management
Gets routing to instances
Transaction Service
[Software System]
Provides donation transactions query and processing
Updates project statistics
[via Kafka topic onProjectUpdate]
Updates transaction statistics
[via Kafka topic onTransactionUpdate]
Statistics Service
[Container: Spring Boot]
Provides statistics for system, transaction, project and user metrics
Registers statistic instance
Notify profile update
[via Kafka topic onProfileUpdate]
Notification Service
[Software System]
Provides real-time notification sending
Statistic System
[Container]
Handles project management requests
[JSON/HTTPS]
Handles statistics requests
[JSON/HTTPS]
Admin
[Person]
Person who can perform system management
Admin Single-Page Application
[Container: React]
Application for system management functionalities
Makes management API calls
[JSON/HTTPS]
Updates user statistics
[via Kafka topic onProfileUpdate]
Notify profile create
[via Kafka topic onProfileCreate]
Email Service
[Software System]
Provides transactional email sending
API Gateway
[Container: Spring Cloud Gateway]
Performs routing and pre-authentication filters
Object Storage
[Container: MinIO]
Stores user static assets (profile avatar)
Handles profile requests
[JSON/HTTPS]
Profile Management Service
[Container: Spring Boot]
Provides user profile management
Reads and writes static assets
[Blob/HTTPS]
Registers profile management instance
Reads and writes profile data
[SQL]
Charity
[Person]
Person who can create donation projects
Makes features API calls
[JSON/HTTPS]
Profile Database
[Container: PostgreSQL]
Stores user profile data
User Single-Page Application
[Software System]
Application for end-user functionalities
Profile Management System
[Container]
Handles authentication requests
[JSON/HTTPS]
Encrypts payment details
[JSON/HTTPS]
Payment Provider Service
[Container: Stripe]
Provides encrypted payment storing and processing
Donor
[Person]
Person who can perform donations
Sends authentication update
[via Kafka topic onAuthenticationUpdate]
Authentication Service
[Container: Spring Boot]
Provides user authentication, token issuing and refreshing
Registers authentication instance
Reads and writes authentication data
[SQL]
Authentication Database
[Container: PostgreSQL]
Stores user authentication and authorization (roles) data
Authentication System
[Container]
Updates signing key
[via Kafka topic onSigningKeyUpdate]
Key Management Service
[Software System]
Provides encryption/signing key access and refresh
Updates encryption key
[via Kafka topic onEncryptionKeyUpdate]
Updates decryption key
[via Kafka topic onDecryptionKeyUpdate]

Back-end container diagram. Containers in blue background are implemented.

Services are developed using microservices architecture with Spring Boot to provide clear separation of concerns, forcing each service to only serve its related functions (single-purpose) and decrease unnecessary cross-service data access. Moreover, service databases are properly scoped to perform data sharding and replication to ensure high availability while minimizing data loss, and are deployed with PostgreSQL. For storing static assets and objects, MinIO is used for its S3-compatible API and fully self-hosted structure, providing a convenient interface for scaling or changes in provider later in project life-span.

With many microservices being deployed and replicated throughout multiple regions in production, communication between microservices will be handled more efficiently using an event streaming service to ensure data is properly sent and handled in real-time. Using Apache Kafka, services could produce (send) or consume (receive) messages asynchronously via a defined topic without knowing the sender or receiver address, while messages are guaranteed to be kept and delivered on arrival at a Kafka's message broker node [2]. Since messages are being produced and consumed asynchronously, services are not bottlenecked by others execution time, overall improves system efficiency.

Using this principle, content of latest messages could be stored in memory at each microservice level, while replication of services could continue to easily access the same data using Kafka message offset query, resulting in infrequent communication requests being minimized. In case of data updates, all downstream consumer services could listen to the same Kafka topic and update accordingly, without polling to request data from upstream producer services.

By applying event-driven architecture design, microservices can be horizontally scaled by deploying more instances and load balancing their traffic, including Kafka instances with KRaft for distributed scaling.

To explore the back-end's container and component diagrams interactively, see Appendix B.

II.III API Gateway and Authentication Component

Being mission-critical to the whole system by providing a secure way to handle authenticated requests between clients and services, end-to-end encryption, properly routing requests and restricting unauthorized access, the API gateway and authentication component are designed to be highly available through self-contained service scaling and distribution, applying stateless authentication methods.

Therefore, API gateway will provide payload decryption and routing capability, with authentication component handle validation and token issuing on login and signup.

II.III.I Background

Payload encryption

Ensuring user credentials and details are securely handled from every client to corresponding services, authentication data such as login and signup details are designed to be end-to-end encrypted using JSON Web Encryption (JWE) in the team authentication system, taking advantage of asymmetric encryption similar to HTTPS/TLS [3].

In detail, end-to-end payload encryption is made possible using a keypair that includes a public key and a private key. Using that private key, anyone could encrypt the message that only the one holding the corresponding private key would be able to decrypt and “read” that message. In this context, every client of the system (front-end or client-side) could encrypt the authentication payload so that only some of our trusted systems (back-end) could decrypt the message [4].

Using that principle, encrypting payloads from the client would prove to be useful in client-server communication, ensuring any sent data from the client is not being tampered or exposed to third-parties.

Claims signature

To further enhance the security and authenticity of requests to, from, and between our services, authentication claims tokens issued to the client are signed and serialized using JSON Web Signature (JWS), moreover implemented with asymmetric encryption instead of common symmetric encryption.

While asymmetric encryption may be similar to encrypting payload as JWE in keypair usage, the role of private and public keys in the signing scenario is different in both system roles and implementation. Instead of using a private key to decrypt in the encryption process, that key is only used by some trusted systems to sign the payload (claims in this context) to ensure its content will not be not changed in transit. After that when tokens are delivered to the controlled services, the request authorization could be validated to ensure it is authentic and is not tampered in data exchanges [4].

In service to service communications such as routing between gateway and microservices, signing claims would prove to better enhance the system security to internal factors like token tampering in cross-service exchanges that could occur even in better-controlled internal networks routes.

Authentication strategy

Combining claims signature and payload encryption, the authentication component delivers authenticated data as a JWE token, with its payload being an embedded JWS token, resulting in a Nested JWT that would be delivered to the client user. This is similar to industry-standard authentication protocols such as OpenID Connect [5].

Applying this strategy, two of the needed keypairs would then be distributed throughout different services to decrease attack surface such as key breaches, while providing both JWE and JWS benefits such as confidentiality of the data in transport and integrity of the data in processing [6].

In case of attacks, keys could be refreshed or rotated to another generated pair by internal key service and re-distribute to ensure integrity of the system, denying bad actors of using known or generated tokens using breached keys. Moreover, this also provides a strategy to further tighten the system security by periodically refreshing keypairs, using concurrently two keypairs for both encryption and signing, combined with refresh tokens on client-side to ensure users are always authenticated with latest tokens.

II.III.II Caveats

While there are multiple strengths in security with the team approach, it poses some minor inconvenience in service bootstrapping.

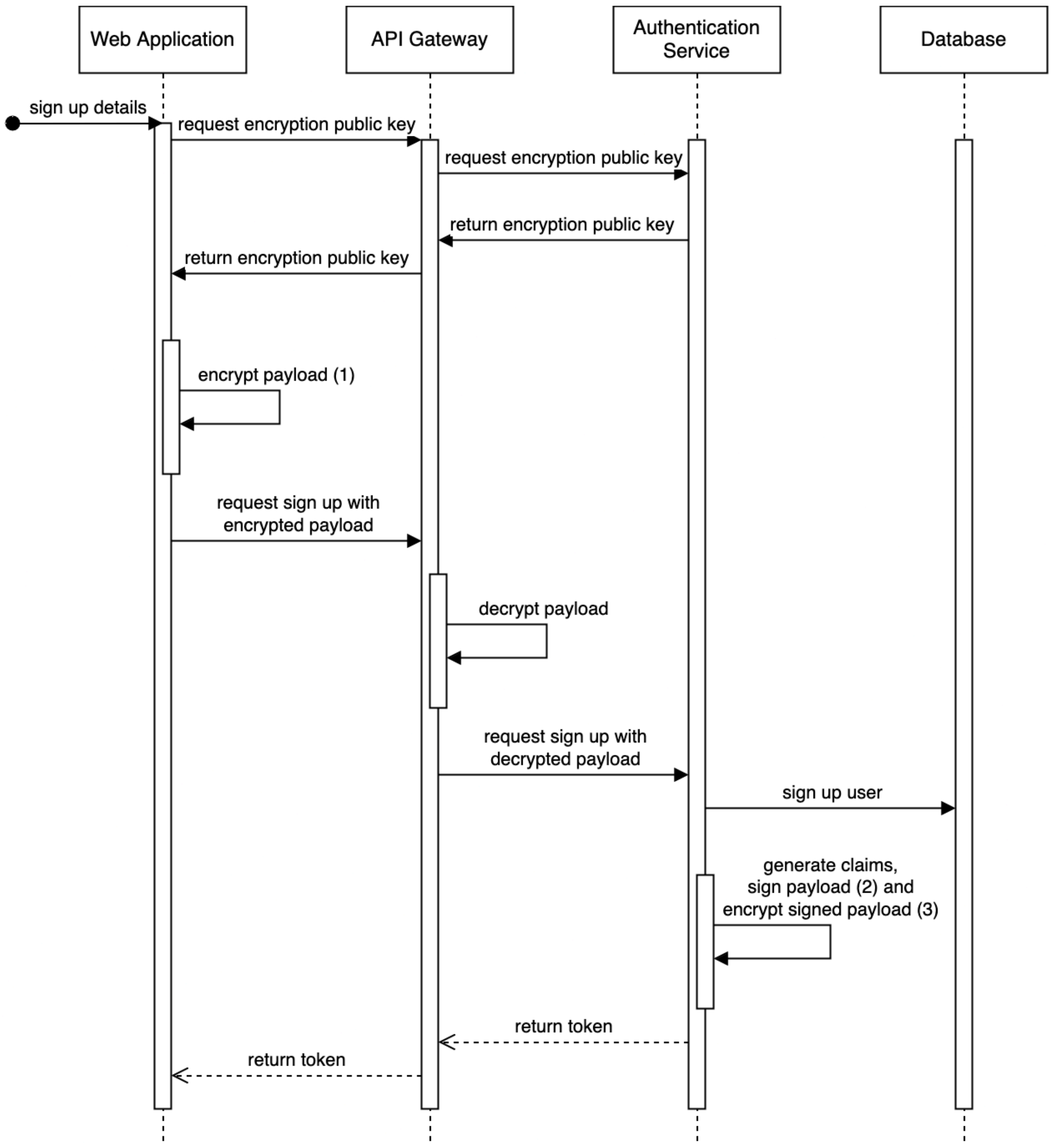
Since JWS tokens are expected to be validated in each service before processing requests to ensure authenticity of that request, implementation of each service may be prone to increase in application dependencies and support modules such as a "common" JWS validation module and a local key handling module.

Moreover, as JWS are stateless, compromised tokens are difficult to revoke and provide a way for bad actors to execute requests as if they are the token user. In implementation, this could be partially solved by strengthening token storage on the front-end (by using secure HTTP-only cookie) and decreasing token expiry time on back-end, requiring refresh calls to be called more frequently.

II.III.III Implementation

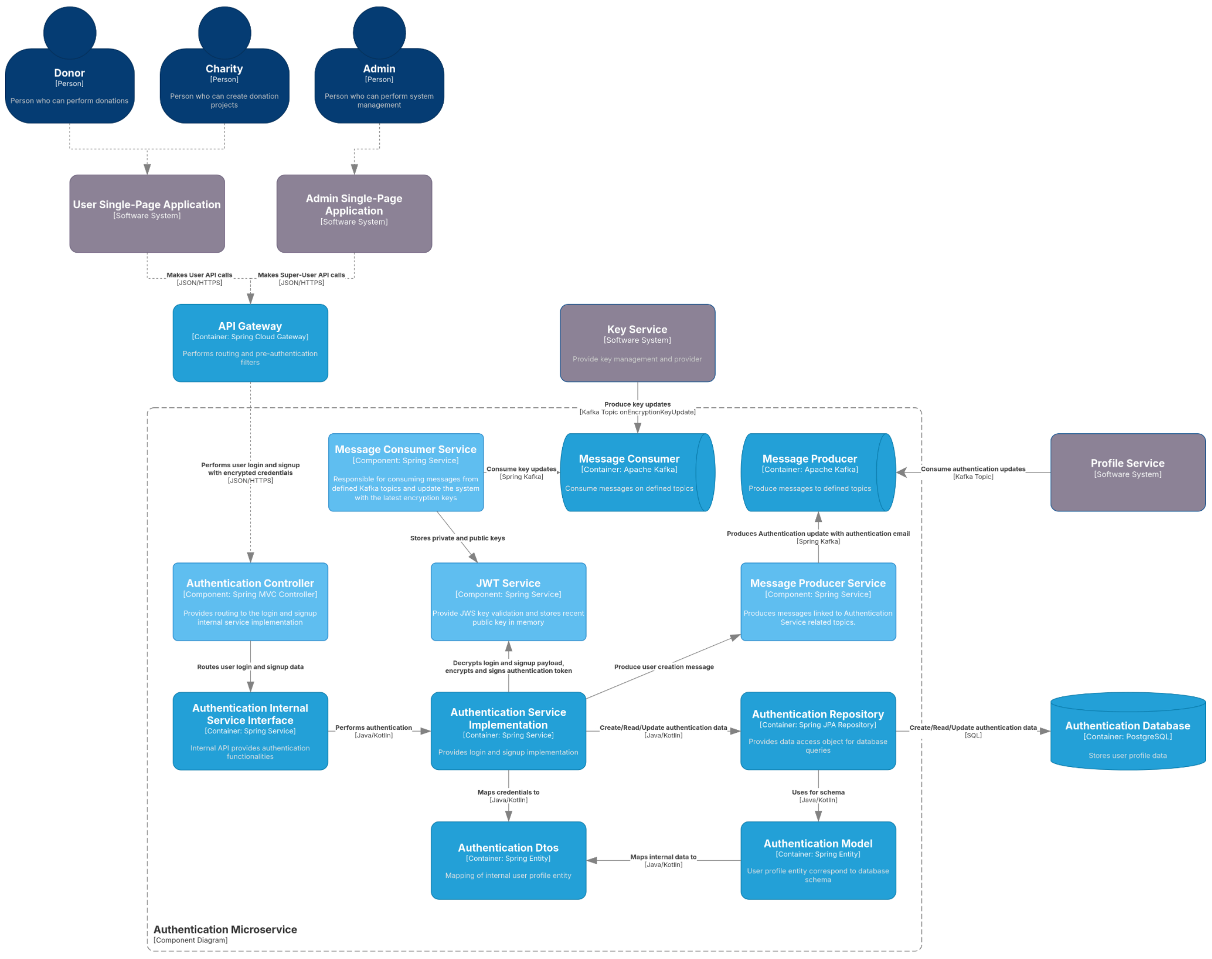
In the Charitan application, there are four main components that are essential in providing authentication and authorization to the user securely, namely API gateway, authentication service and key service, with last being every other protected microservices that requires authentication. For the key algorithm, encryption keypair will use RSA with SHA-512 for wide support on clients (needed for client-side encryption), and signing keypair will use more recent EdDSA.

First, a JWE decrypt security filter will be implemented in API Gateway to decrypt data sent to the system, that includes login and signup details (for authentication), and authenticated tokens (for authorization). The gateway service will require an encryption private key to perform decryption of JWE payload.


Web Application
API Gateway
Authentication Service
Database
sign up details
request encryption public key
request encryption public key
return encryption public key
return encryption public key
encrypt payload (1)
request sign up with encrypted payload
decrypt payload
request sign up with decrypted payload
sign up user
generate claims, sign payload (2) and encrypt signed payload (3)
return token
return token

Simplified sign up sequence diagram.
(1) JWE-encrypted data, (2) JWS-signed data, (3) nested JWT.

Since decrypted, tokens from API Gateway to Microservices will only include serialized registration and login data for authentication, or JWS token (using Nested JWT) for others to validate. Using this strategy could simplify the validation logic on microservice-side providing cross-service network calls are often controlled and private (using cloud-native networking such as AWS VPC or mesh VPNs/tunnels such as Tailscale), while providing more secure authentication on client-side using encrypted payload to gateway servers.

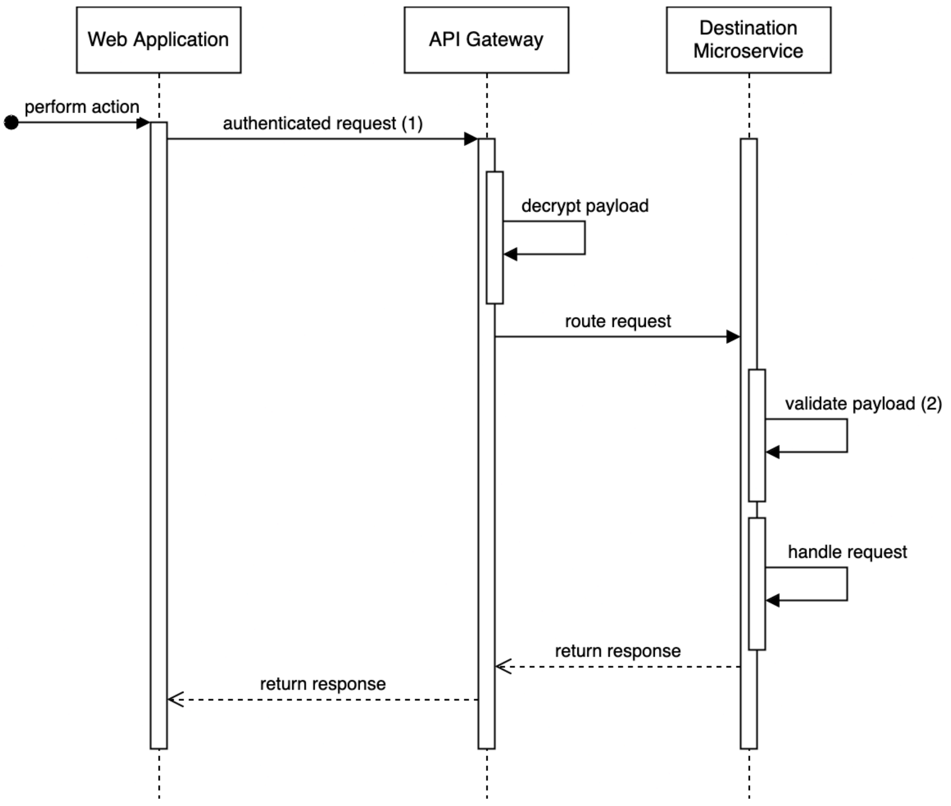

Donor
[Person]
Person who can perform donations
Charity
[Person]
Person who can create donation projects
Admin
[Person]
Person who can perform system management
User Single-Page Application
[Software System]
Admin Single-Page Application
[Software System]
Makes User API calls
[JSON/HTTPS]
Makes Super-User API calls
[JSON/HTTPS]
API Gateway
[Container: Spring Cloud Gateway]
Performs routing and pre-authentication filters
Key Service
[Software System]
Provide key management and provider
Produce key updates
[Kafka Topic onEncryptionKeyUpdate]
Performs user login and signup with encrypted credentials
[JSON/HTTPS]
Message Consumer Service
[Component: Spring Service]
Responsible for consuming messages from defined Kafka topics and update the system with the latest encryption keys
Consume key updates
[Spring Kafka]
Message Consumer
[Container: Apache Kafka]
Consume messages on defined topics
Message Producer
[Container: Apache Kafka]
Produce messages to defined topics
Consume authentication updates
[Kafka Topic]
Profile Service
[Software System]
Stores private and public keys
Produces Authentication update with authentication email
[Spring Kafka]
Authentication Controller
[Component: Spring MVC Controller]
Provides routing to the login and signup internal service implementation
JWT Service
[Component: Spring Service]
Provide JWS key validation and stores recent public key in memory
Message Producer Service
[Component: Spring Service]
Produces messages linked to Authentication Service related topics.
Routes user login and signup data
Decrypts login and signup payload, encrypts and signs authentication token
Produce user creation message
Authentication Internal Service Interface
[Container: Spring Service]
Internal API provides authentication functionalities
Performs authentication
[Java/Kotlin]
Authentication Service Implementation
[Container: Spring Service]
Provides login and signup implementation
Create/Read/Update authentication data
[Java/Kotlin]
Authentication Repository
[Container: Spring JPA Repository]
Provides data access object for database queries
Create/Read/Update authentication data
[SQL]
Authentication Database
[Container: PostgreSQL]
Stores user profile data
Maps credentials to
[Java/Kotlin]
Uses for schema
[Java/Kotlin]
Authentication Dtos
[Container: Spring Entity]
Mapping of internal user profile entity
Maps internal data to
[Java/Kotlin]
Authentication Model
[Container: Spring Entity]
User profile entity correspond to database schema
Authentication Microservice
[Component Diagram]

Authentication microservice component diagram.

To provide authentication tokens for users, the authentication service will be implemented using Spring Boot to handle all login and signup requests, then produce side-effects on success: issue token and registration message.

On both login and signup operation, the system will perform data transfer object (DTO) mapping internally for each individual request to reduce chances of access to non-public data, which also validates the input using Spring Validation. Issuing authentication token will be performed after each successful login and signup, where the authentication service will first create user claims, sign the claim with a signing private key and encrypt it with an encryption public key to create a Nested JWT that will be attached to the response cookie header. All signing and encrypting operations will be handled separately using an internal JWT service bean that is implemented using Java JJWT library [7].

On successful registration, the service will produce a corresponding message via Kafka with the user ID for later handling in user profile creation for storing user data that is not authentication-related. As needed by the client to encrypt authorization data, authentication service will also be implemented to respond to clients' encryption public key requests.



Simplified authenticated requests sequence diagram.
(1) nested JWT, (2) JWS-signed data.

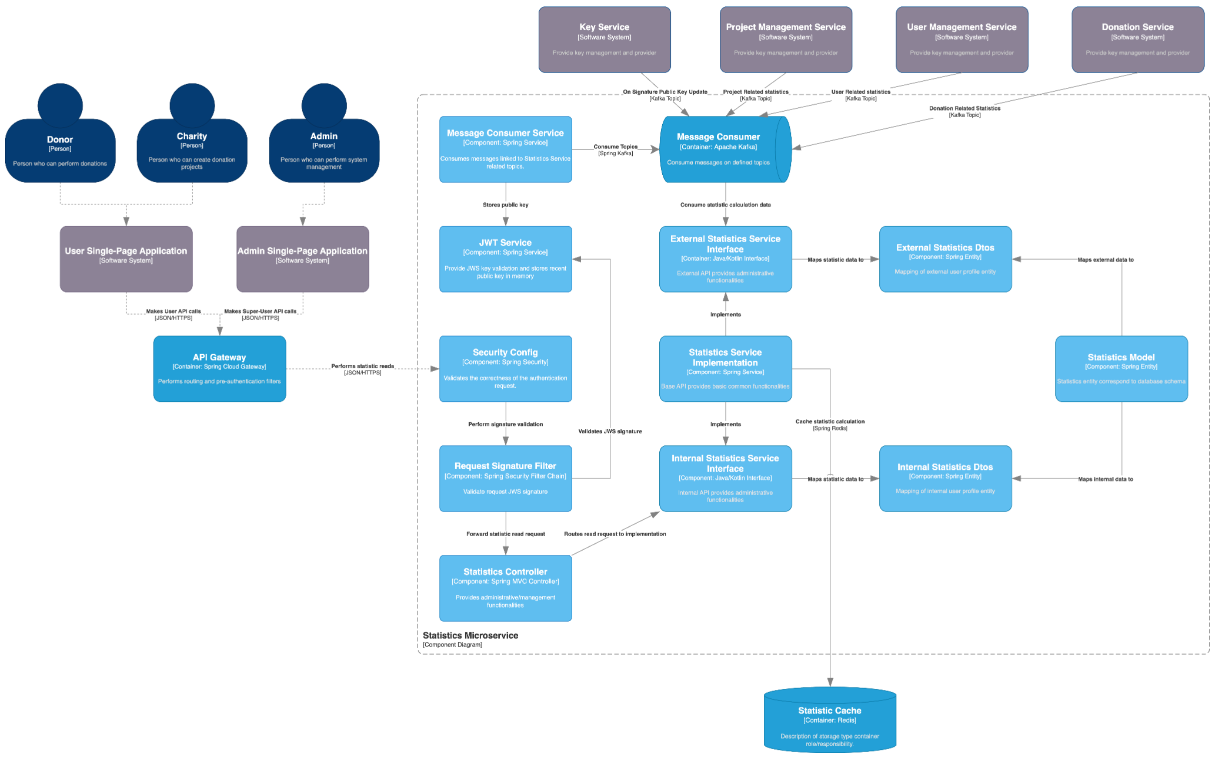
On every protected microservices, a security filter will be implemented to validate user requests by checking token signature, which requires a signing public key to perform validation.

For key delivery, each service will listen and consume its corresponding key topic via Apache Kafka, requiring the key to be stored safely to memory and used on demand. By using a stateless authentication method that is JWT and distributed key services, the authentication could be scaled horizontally according to demand by starting more instances and load balance between them. Moreover, distributed key locations could significantly decrease the chance of keypair being leaked, further increasing the security of the whole system.

In summary, the Key Service is required to provide the following keys by Kafka topics to corresponding services.

	Private key	**Public key**
Encryption	API Gateway	Authentication Service
Signature	Authentication Service	Protected Microservices

II.IV Statistics Component


Key Service
[Software System]
Provide key management and provider
Project Management Service
[Software System]
Provide key management and provider
User Management Service
[Software System]
Provide key management and provider
Donation Service
[Software System]
Provide key management and provider
On Signature Public Key Update
[Kafka Topic]
Project Related statistics
[Kafka Topic]
User Related statistics
[Kafka Topic]
Donation Related Statistics
[Kafka Topic]
Donor
[Person]
Person who can perform donations
Charity
[Person]
Person who can create donation projects
Admin
[Person]
Person who can perform system management
Message Consumer Service
[Component: Spring Service]
Consumes messages linked to Statistics Service related topics.
Consume Topics
[Spring Kafka]
Message Consumer
[Container: Apache Kafka]
Consume messages on defined topics
Stores public key
Consume statistic calculation data
User Single-Page Application
[Software System]
Admin Single-Page Application
[Software System]
JWT Service
[Component: Spring Service]
Provide JWS key validation and stores recent public key in memory
External Statistics Service Interface
[Container: Java/Kotlin Interface]
External API provides administrative functionalities
Maps statistic data to
External Statistics Dtos
[Component: Spring Entity]
Mapping of external user profile entity
Maps external data to
Makes User API calls
[JSON/HTTPS]
Makes Super-User API calls
[JSON/HTTPS]
Implements
API Gateway
[Container: Spring Cloud Gateway]
Performs routing and pre-authentication filters
Performs statistic reads
[JSON/HTTPS]
Security Config
[Component: Spring Security]
Validates the correctness of the authentication request.
Statistics Service Implementation
[Component: Spring Service]
Base API provides basic common functionalities
Statistics Model
[Component: Spring Entity]
Statistics entity correspond to database schema
Perform signature validation
Validates JWS signature
Implements
Cache statistic calculation
[Spring Redis]
Request Signature Filter
[Component: Spring Security Filter Chain]
Validate request JWS signature
Internal Statistics Service Interface
[Component: Java/Kotlin Interface]
Internal API provides administrative functionalities
Maps statistic data to
Internal Statistics Dtos
[Component: Spring Entity]
Mapping of internal user profile entity
Maps internal data to
Forward statistic read request
Routes read request to implementation
Statistics Controller
[Component: Spring MVC Controller]
Provides administrative/management functionalities
Statistics Microservice
[Component Diagram]
Statistic Cache
[Container: Redis]
Description of storage type container role/responsibility.

Statistics microservice component diagram.

The **Statistics Component** in the Charitan Donation Platform is responsible for aggregating, processing, and delivering critical statistical insights to administrators. It integrates data from various external services, such as user management, project management, and donation services, to provide real-time updates on system activity.

Upon successful authentication and validation, the statistical data request is routed to the **Statistics Controller**. This controller serves as the intermediary between the incoming API requests and the internal processing services. It communicates with the **Internal Statistics Service Interface**, which encapsulates the core business logic, ensuring that the system's internal processes remain well-organized and maintainable.

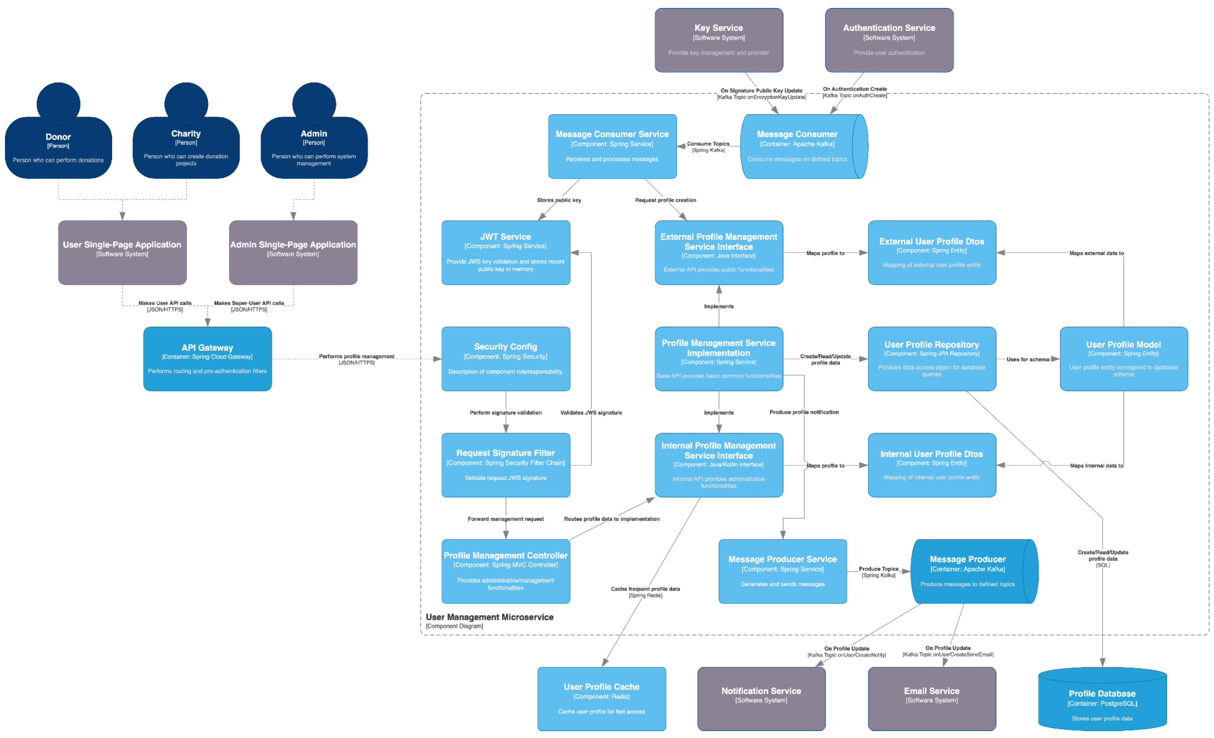
To enhance data retrieval times and optimize performance, the **Internal Statistics Service Interface** leverages **Redis** as a caching layer. By caching the latest calculated statistical values, the system can efficiently retrieve and update statistics without redundant recalculations, thereby improving overall system performance and responsiveness.

The architecture further integrates with various external services, including the **Key Service**, **Project Management Service**, **User Management Service**, and

Donation Service, to aggregate comprehensive statistical data. These services publish relevant data to specific **Kafka** topics, which are then consumed by the **Message Consumer Service**. This adoption of **Apache Kafka** for event-driven message brokering ensures real-time data processing and timely statistical calculations in response to changes within these microservices.

The **Message Consumer Service** interfaces with the **JWT Service** to manage public key updates and maintain the integrity of JWS signatures. This ensures that as public keys are rotated and updated, the authentication mechanisms remain robust and secure. Additionally, integrating the **JWT Service** with Kafka consumption guarantees that security remains uncompromised even as external services evolve, providing a resilient and scalable security framework.

II.V Profile Management Component



Key Service
[Software System]
Provide key management and provider
Authentication Service
[Software System]
Provide user authentication
On Signature Public Key Update
[Kafka Topic onEncryptionKeyUpdate]
On Authentication Create
[Kafka Topic onAuthCreate]
Donor
[Person]
Person who can perform donations
Charity
[Person]
Person who can create donation projects
Admin
[Person]
Person who can perform system management
Message Consumer Service
[Component: Spring Service]
Receives and processes messages
Consume Topics
[Spring Kafka]
Message Consumer
[Container: Apache Kafka]
Consume messages on defined topics
Stores public key
Request profile creation
User Single-Page Application
[Software System]
Admin Single-Page Application
[Software System]
JWT Service
[Component: Spring Service]
Provide JWS key validation and stores recent public key in memory
External Profile Management Service Interface
[Component: Java Interface]
External API provides public functionalities
Maps profile to
External User Profile Dtos
[Component: Spring Entity]
Mapping of external user profile entity
Maps external data to
Makes User API calls
[JSON/HTTPS]
Makes Super-User API calls
[JSON/HTTPS]
Implements
API Gateway
[Container: Spring Cloud Gateway]
Performs routing and pre-authentication filters
Performs profile management
[JSON/HTTPS]
Security Config
[Component: Spring Security]
Description of component role/responsibility.
Profile Management Service Implementation
[Component: Spring Service]
Base API provides basic common functionalities
Create/Read/Update profile data
User Profile Repository
[Component: Spring JPA Repository]
Provides data access object for database queries
Uses for schema
User Profile Model
[Component: Spring Entity]
User profile entity correspond to database schema
Perform signature validation
Validates JWS signature
Implements
Produce profile notification
Request Signature Filter
[Component: Spring Security Filter Chain]
Validate request JWS signature
Internal Profile Management Service Interface
[Component: Java/Kotlin Interface]
Internal API provides administrative functionalities
Maps profile to
Internal User Profile Dtos
[Component: Spring Entity]
Mapping of internal user profile entity
Maps internal data to
Forward management request
Routes profile data to implementation
Profile Management Controller
[Component: Spring MVC Controller]
Provides administrative/management functionalities
Cache frequent profile data
[Spring Redis]
Message Producer Service
[Component: Spring Service]
Generates and sends messages
Produce Topics
[Spring Kafka]
Message Producer
[Container: Apache Kafka]
Produce messages to defined topics
Create/Read/Update profile data
[SQL]
User Management Microservice
[Component Diagram]
On Profile Update
[Kafka Topic onUserCreateNotify]
On Profile Update
[Kafka Topic onUserCreateSendEmail]
User Profile Cache
[Component: Redis]
Cache user profile for fast access
Notification Service
[Software System]
Email Service
[Software System]
Profile Database
[Container: PostgreSQL]
Stores user profile data

Profile Management microservice component diagram.

The **Profile Management Component** plays a critical role that handles all the interactions with the user profile data. It integrates with different external services to ensure the efficient and secure management of user profiles across the system.

The Profile Management Controller plays an important role in this microservice. The Profile Management Controller is a key component in the backend and handles all profile-related requests. It serves both Admin-specific and general user operations, routing them to the appropriate internal or external services. By separating the routing and request processing from the business logic, the controller ensures a clear separation of concerns, making the system easier to maintain and extend, as well as enforcing authentication and authorization before delegating actions to services.

Profile Management Services

The system architecture distributes clearly between Internal Services and External Services to achieve modularity, security, and scalability. This separation ensures that different responsibilities are handled by distinct services, improving maintainability and reducing risks.

Internal User Management Service

The internal profile management service is used to handle the Admin functions such as: creating, updating and retrieving user profiles as well as integrations with the database and the message queues for Admin actions. The internal service is private and designed to handle tasks that are restricted to specific roles or functionalities within the system, which is not exposed to external users or services. This implementation will enhance the encapsulation of the business logic as well as restrict access to sensitive operations, ensuring that they are only available to authenticated and authorized users.

External Profile Management Service

The external service is the publicly accessible API layer which provides necessary functions that can be used by various roles such as Donors, Charities and Admins. It is designed to handle shared functionalities in order to ensure a clear distribution between public logic and internal system operations. Implementing the external service will ensure that safe data is sent to external clients which will improve the security and performance of the application. The external profile management service is used to process requests coming from the frontend or external systems via the API Gateway. These requests are typically read-heavy operations that involve fetching or displaying data.

Data Transfer Object (DTO)

In the Profile Management Service, the system uses the Internal DTO and External DTO in order to transfer the data between different layers without exposing the internal implementation details. The Internal DTO is used for internal communication which is the backend operation, while the External DTO is used to communicate with external systems by transferring only the safe and necessary data. This implementation will enhance the encapsulation and ensure the data security of the service.

User Profile Repository and Database

The repository connects the application to the PostgreSQL database, which stores all user profile information. This implementation provides a structured way to manage database operations, ensuring consistency between the application and the data while PostgreSQL was chosen for its reliability, scalability, and support for complex queries.

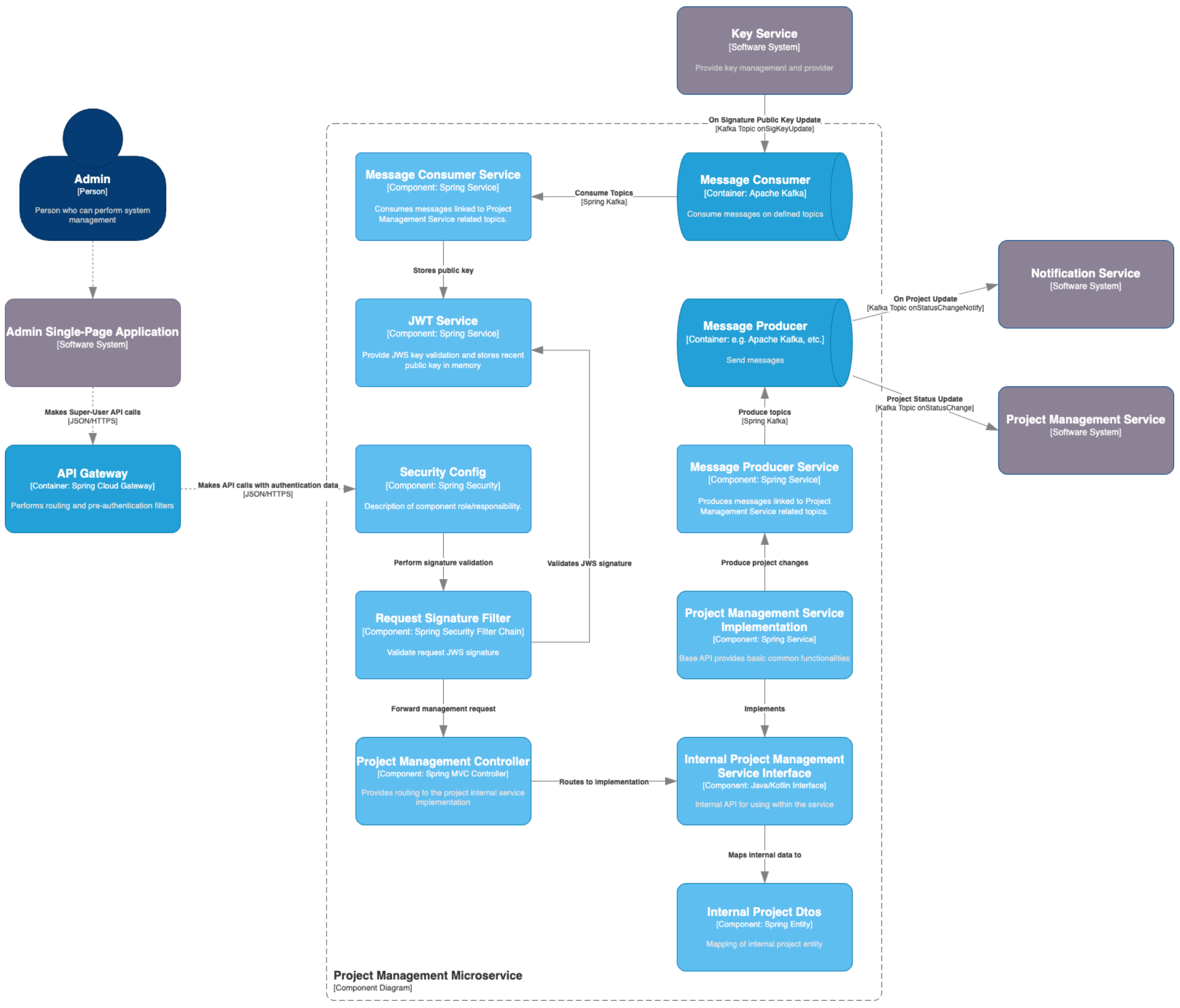
User Profile Cache

Frequently accessed user data is stored in Redis, an in-memory data store, for faster retrieval which will significantly reduce the load on the database and speeds up operations, making it ideal for read-heavy tasks like retrieving user profiles.

System Flow

After successful authentication and validation which is handled by the **API Gateway**, it will route the request to the appropriate backend service. If the request is the client-facing operations, the **External Profile Management Service** will handle the request using **External DTO** to transfer only safe and necessary data. On the other hand, sensitive or role-specific operations will be handled by the **Internal Profile Management Service** which uses the **Internal DTO** to encapsulate the necessary fields. After completing the operation, the service interacts with the database for storage and updates the **Redis** cache to improve the performance of the service, while also producing an event to a Kafka topic (such as to communicate with other services and these external services such as the Notification Service will consume these events to send emails. Meanwhile, the backend service will immediately send a confirmation response back to the user via the **API Gateway** which ensures the efficiency and security of the service.

II.VI Project Management Component



Key Service
[Software System]
Provide key management and provider
Admin
[Person]
Person who can perform system management
On Signature Public Key Update
[Kafka Topic onSigKeyUpdate]
Message Consumer Service
[Component: Spring Service]
Consumes messages linked to Project Management Service related topics.
Consume Topics
[Spring Kafka]
Message Consumer
[Container: Apache Kafka]
Consume messages on defined topics
Stores public key
Admin Single-Page Application
[Software System]
JWT Service
[Component: Spring Service]
Provide JWS key validation and stores recent public key in memory
Message Producer
[Container: e.g. Apache Kafka, etc.]
Send messages
On Project Update
[Kafka Topic onStatusChangeNotify]
Notification Service
[Software System]
Project Status Update
[Kafka Topic onStatusChange]
Project Management Service
[Software System]
Makes Super-User API calls
[JSON/HTTPS]
Produce topics
[Spring Kafka]
API Gateway
[Container: Spring Cloud Gateway]
Performs routing and pre-authentication filters
Makes API calls with authentication data
[JSON/HTTPS]
Security Config
[Component: Spring Security]
Description of component role/responsibility.
Message Producer Service
[Component: Spring Service]
Produces messages linked to Project Management Service related topics.
Perform signature validation
Validates JWS signature
Produce project changes
Request Signature Filter
[Component: Spring Security Filter Chain]
Validate request JWS signature
Project Management Service Implementation
[Component: Spring Service]
Base API provides basic common functionalities
Forward management request
Implements
Project Management Controller
[Component: Spring MVC Controller]
Provides routing to the project internal service implementation
Routes to implementation
Internal Project Management Service Interface
[Component: Java/Kotlin Interface]
Internal API for using within the service
Maps internal data to
Internal Project Dtos
[Component: Spring Entity]
Mapping of internal project entity
Project Management Microservice
[Component Diagram]

Project Management microservice component diagram.

The component serves as a middleman API between the frontend web application (Admin client) and the Project Management CRUD service developed by Team A. This intermediary is necessary for the following reasons:

1. Security: The backend developed by Team A uses Kafka for communication, and exposing its topics and communication channels directly to the client is a security risk.
2. Data Filtering and Validation: Incoming data from the client must be filtered and validated before being consumed by Kafka.

Note about inner DTO: since the project management component is not exposed to anyone except for the admin, there is no need for the **External Interface.**

When a request is received from the client:

- The API endpoint forwards the request data, along with the JWT and public auth key received through [onSigKeyUpdate] Kafka topic, to the **Project Management Controller**.
- The controller interacts with the **Project Management Service**, which is an implementation of Spring Boot's **Internal Project Management Service Interface**.

Data Processing and Kafka Communication

- The **Project Management Service** processes the data and forwards it to a **Message Producer Service**.
- The **Message Producer Service**, in turn, uses Kafka's generic **Message Producer** to publish the data to the appropriate Kafka topics ([onStatusChangeNotify] and [onStatusChange]).
- The component maintains an active connection with both Team A's **Project Management Service** and a **Notification Service**.
- Relevant data is sent to these services via the designated Kafka topics.

II.VII Design Justification

II.VII.I Maintainability

The backend is divided into distinct microservices, each encapsulating specific functionalities and as a result it makes it easier for developers to understand the logic of a single service without needing to understand the entire system.

Each service can be tested independently, enabling targeted unit, integration and performance testing.

Each microservice follows the layered design which separates responsibilities into distinct layers such as controllers, business logic and data access. This separation of layers makes it easier to understand and update logics. Furthermore, rules modified in the service layer do not impact the API endpoints.

II.VII.II Extensibility

The microservice design makes sure that each service represents a specific domain, making it easier to locate and address bugs or make changes. This architecture also ensures that an update to one service does not require changes to other services, reducing unintended side effects caused by maintaining a specific service.

The system uses internal and external DTOs to control the flow of data between services. The use of DTO enables backward compatibility by allowing new versions of data models (after maintenance) to coexist with existing ones.

The centralized API Gateway ensures that any changes to routing or authentication logic can be made centrally without modifying each service.

The system uses RBAC to enforce access control, therefore new roles or permissions can be added without major changes to the codebase.

II.VII.III Resilience

The system utilizes Redis for caching the countries and regions from the external countries API to avoid inaccessibility to that API due to rate-limit or API server down.

We also enhance backend's availability with PostgreSQL Replication by having multiple copies of the primary database. In case the primary database fails, one of the copies can be promoted to be used by the system [8].

II.VII.IV Scalability

The system uses Apache Kafka for event driven communication between services. Kafka can handle large volumes of data with low latency which makes it ideal to handle increased load.

Redis is used to cache the frequently accessed data, this minimizes direct database queries, allowing the system to handle more users simultaneously.

The micro-service architecture enables each service to be containerized using docker and additional instances of the service can be deployed using Kubernetes.

II.VII.V Security

RBAC not only restricted users by roles but also by the permissions attached to them which make it even more secure on which resources they can access.

The authentication system uses JSON Web Encryption (JWE) for secure data transmission, taking advantage of asymmetric encryption similar to HTTPS/TLS.

The system uses the key service to distribute encryption and signature keys securely through Kafka. This reduces the attack surface for potential breaches.

The use of data transfer objects along with internal and external service interfaces ensure that only necessary data is exposed during communication.

II.VII.VI Performance

For frequently accessed data such as statistics and users, we implement Redis as a caching layer to reduce database load and avoid repetitive lookups in the database.

Not only caching statistics and users, we also cache the countries and regions from the external countries API to minimize the latency due to network delay.

III. Frontend Architectural Justification

III.I Introduction

According to the requirements, our team must develop a web application for admin to monitor user, project, and view statistics. Therefore, there are four main pages that we will work on:

1. Authentication page
2. User Management page
3. Project Management page
4. Statistics page

As we investigate, there are many duplicated components throughout the application. For example, with the table component, we have user management table, project management table, and for form component, we have login form, user creation form, project create form, etc.

We decided to move on with headless UI design choices. Creating multiple reusable components will render their UI into headless components' UI based on their states.

III.II Containers

To explore the front-end's container and component diagrams interactively, see Appendix C.

III.II.I Tools and Libraries Used

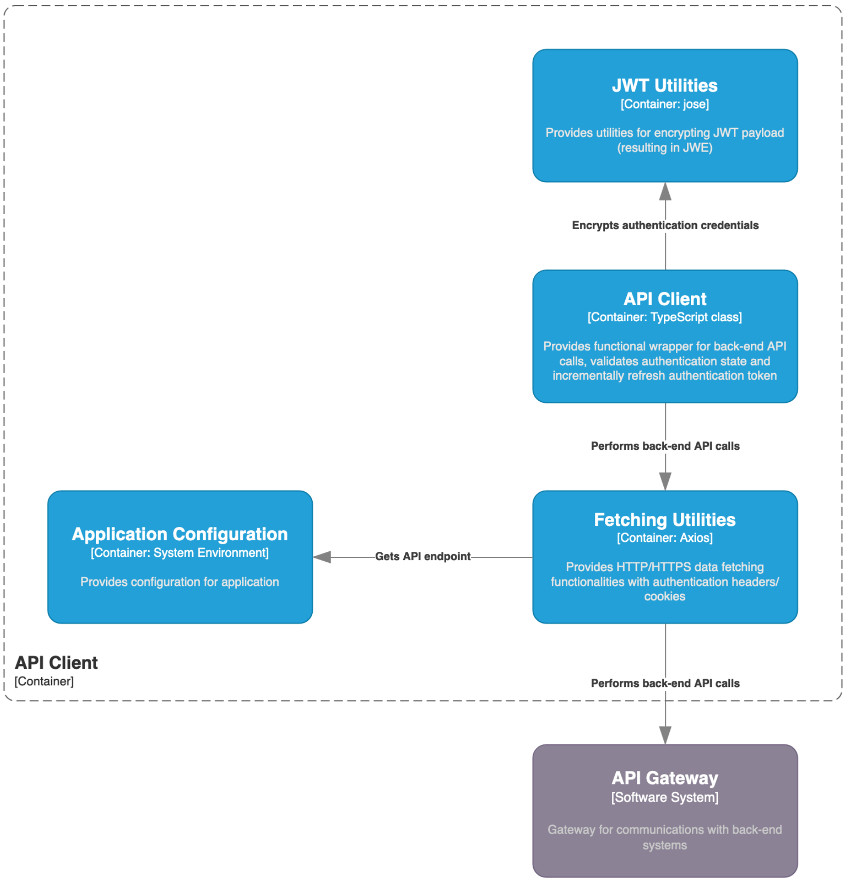
For the ready to use UI components, we are utilizing shadcn/ui components [9]. This will enable us to not reimplement the logic of complex components like Table or Modal by ourselves.

For local URL routing between the components, we are using the React Router v7 [10].

To validate the form data more easily (e.g. applying email / phone number masks), we are using the Zod library for validation utilities [11].

Lastly, we are using Axios for automatic JSON, error, timeout handling and interceptor support [12].

III.II.II API Client


JWT Utilities
[Container: jose]
Provides utilities for encrypting JWT payload (resulting in JWE)
Encrypts authentication credentials
API Client
[Container: TypeScript class]
Provides functional wrapper for back-end API calls, validates authentication state and incrementally refresh authentication token
Performs back-end API calls
Application Configuration
[Container: System Environment]
Provides configuration for application
Gets API endpoint
Fetching Utilities
[Container: Axios]
Provides HTTP/HTTPS data fetching functionalities with authentication headers/ cookies
API Client
[Container]
Performs back-end API calls
API Gateway
[Software System]
Gateway for communications with back-end systems

API Client container diagram

Providing utilities for communication with back-end systems and conveniently managing client authentication state, an API client was implemented with encryption handling, fetching utilities wrappers, functions for defined API calls and optimal token refresh execution, similar to some well-known third-party JavaScript client libraries such as Supabase [13] and Google Firebase [14].

Moreover, by implementing a separate API client, the project could enhance development time by reducing boilerplate in common tasks logic, making it reusable in future projects and providing type inference to request and response object types for TypeScript projects.

Using JWE for client-side encryption on both login and signup payload, and securely storing authentication tokens issued by the system, a JWT utility is needed to encrypt payload data and request token refreshes on expiry. In the Charitan application, JWT utility is implemented using panva/jose JavaScript library [15].

For usage in each component or page, a hook can be further implemented to handle response states, data, validation and errors.

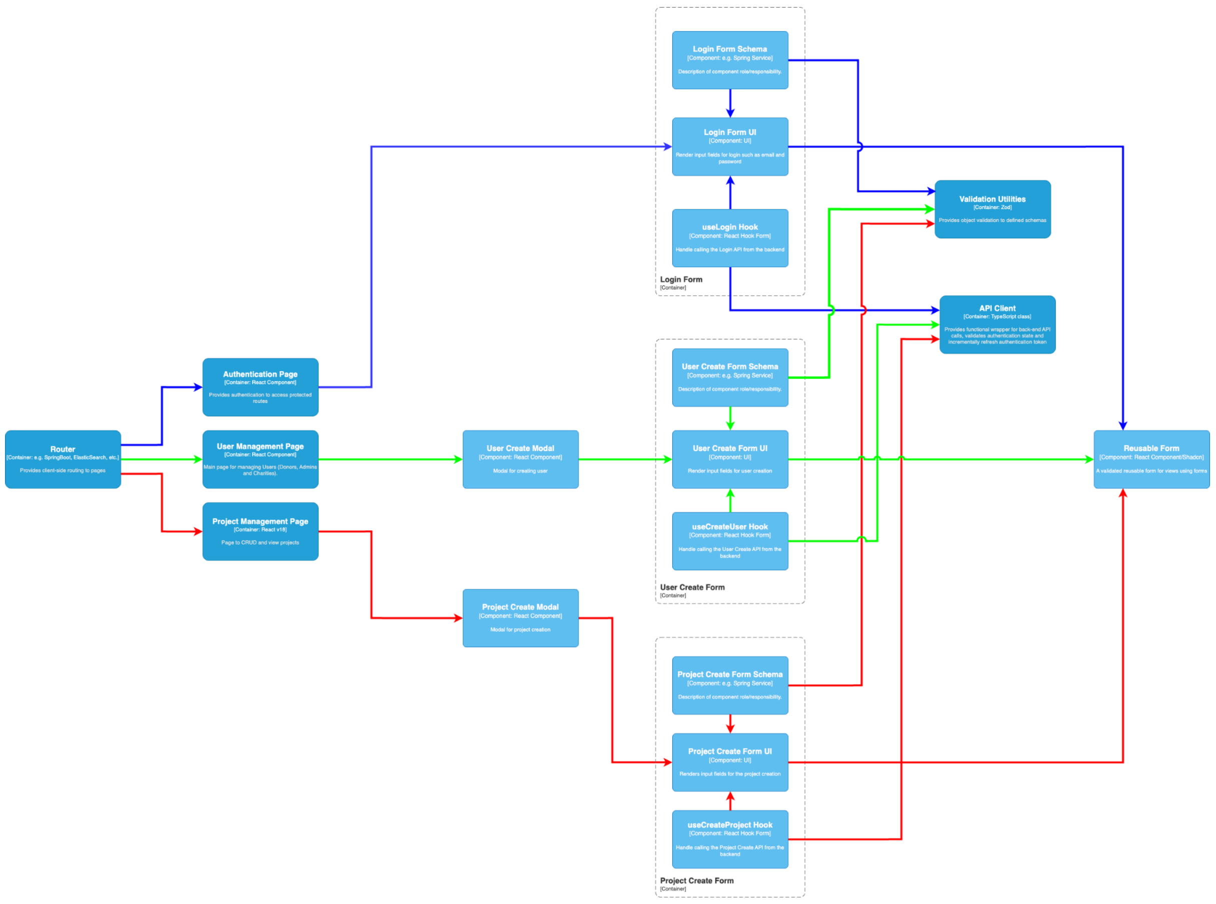
III.II.III State Management

We implement the Zustand library [16] for our system's state management including project management, user management, and statistics to ensure consistency and avoid unnecessary API calls.

Zustand was chosen due to its simplicity, flexibility, and ability to handle both global and local state without the necessity of prop drilling. It is also significantly easier to handle async actions with it.

Our team believes that using Zustand will reduce code boilerplate and let the developer team focus on expressing app logic in a more concise way, which, again, will increase code maintainability.

III.V Form Component



Login Form Schema
[Component: e.g. Spring Service]
Description of component role/responsibility.
Login Form UI
[Component: UI]
Render input fields for login such as email and password
useLogin Hook
[Component: React Hook Form]
Handle calling the Login API from the backend
Login Form
[Container]
Validation Utilities
[Container: Zod]
Provides object validation to defined schemas
API Client
[Container: TypeScript class]
Provides functional wrapper for back-end API calls, validates authentication state and incrementally refresh authentication token
Authentication Page
[Container: React Component]
Provides authentication to access protected routes
Router
[Container: e.g. SpringBoot, ElasticSearch, etc.]
Provides client-side routing to pages
User Management Page
[Container: React Component]
Main page for managing Users (Donors, Admins and Charities).
Project Management Page
[Container: React v18]
Page to CRUD and view projects
User Create Modal
[Component: React Component]
Modal for creating user
Project Create Modal
[Component: React Component]
Modal for project creation
User Create Form Schema
[Component: e.g. Spring Service]
Description of component role/responsibility.
User Create Form UI
[Component: UI]
Render input fields for user creation
useCreateUser Hook
[Component: React Hook Form]
Handle calling the User Create API from the backend
User Create Form
[Container]
Reusable Form
[Component: React Component/Shadcn]
A validated reusable form for views using forms
Project Create Form Schema
[Component: e.g. Spring Service]
Description of component role/responsibility.
Project Create Form UI
[Component: UI]
Renders input fields for the project creation
useCreateProject Hook
[Component: React Hook Form]
Handle calling the Project Create API from the backend
Project Create Form
[Container]

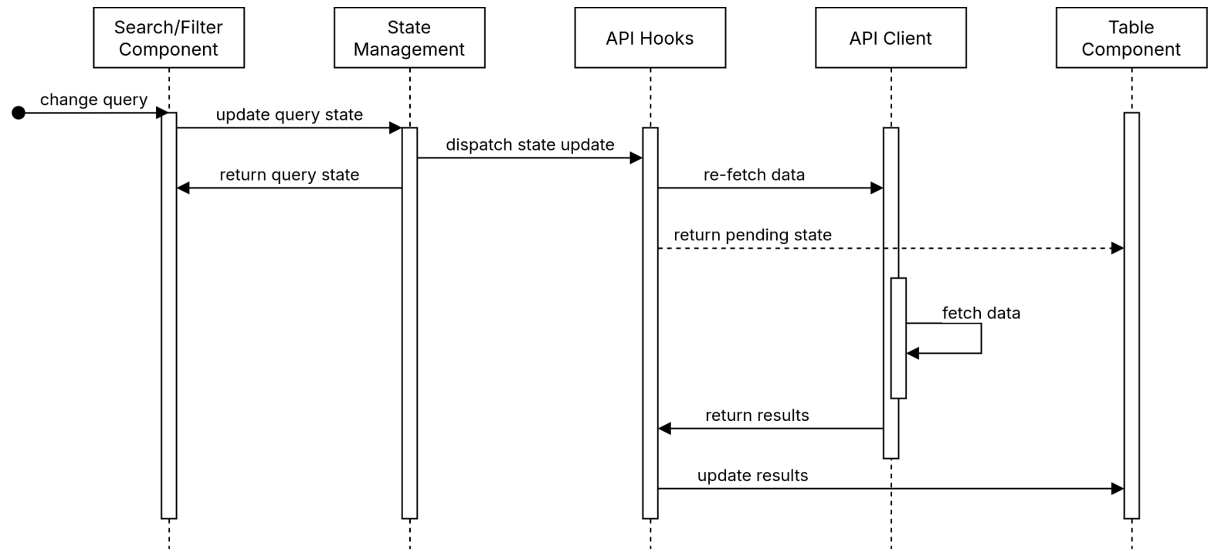
Form component diagram.

The architecture uses a reusable form component alongside validation utilities to handle form-related functionalities across different pages. This is utilized by the Authentication, User Management and Project Management Pages to ensure consistency and promote a "Don't repeat yourself" (DRY) codebase while creating new projects, users or during authentication [17] [18].

The data flow begins when the user input is captured through the form components with the structure defined by the Form Schema. This input data is immediately validated using the centralized validation utilities. Once validated, the data is passed to the logic layers through custom hooks such as `userLogin()`, `useCreateUser()` and `useCreateProject()`, which handles the backend interaction via the API Client. The return results are then returned back to the respective hooks.

III.VI Table Component

There are several tables across the application including user management table, project management table, project statistics table, and donation statistics table. Therefore, a reusable table component is implemented to avoid logic duplication and guarantee consistency.



Inter-component state changes and table data refetch sequence diagram.

Reusable Filter Component, Search Bar Component: Filter and search also affect the table content based on filter and search condition; hence, these two components are included inside the table component diagram. Similar to table components, different pages will have different functional filters and search bars. For that reason, it is necessary to implement a reusable filter component and a reusable search bar component.

Data retrieval, state management, and user interaction are handled by the headless logic layer. The table hook only controls the logic.

The data from the logic layer will be passed to the reusable table user interface component to render the table and show it to the user.

For the search bar, the view layer will be the same as it is just an input field for capturing user’s keywords. Then, the logic layer will get the updated list based on the user's input and update the user/project/donation state management and re-render the table.

Filter is the same as search bar, and the difference is that project filter, user filter, and donation filter will have some uncommon categories.

III.VII Chart Component

Reusable charts component: There are two primary charts—one for projects and one for donations. These charts visually display data, like trends or summaries, to help users understand the information quickly. The logic for fetching and structuring the data is handled in the background, allowing developers to customize how the charts look.

Reusable Search bar component: These allow users to search for specific projects or donations. They connect to the backend of the system to find relevant results while keeping the user interface simple and clean.

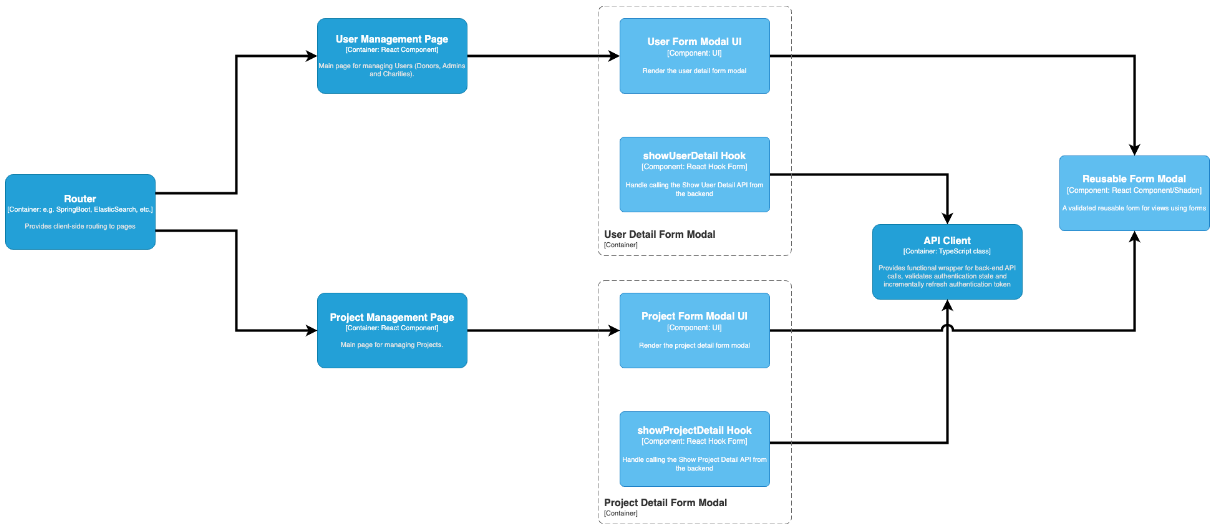
Reusable Filter component: These components are used to filter out the specific projects or donations based on some specified criteria.

For the flow of the charts, the data of the project charts and donation charts are fetched using hooks such as useProject hook and useDonation hook. These hooks pull necessary data through the API client and provide it to the chart which is visualized dynamically.

For the flow of the search bar, the UI takes the keyword from the users which enables users to locate the specified items based on that keyword. It uses the searchProject hook or the searchDonation hook to interact with the backend via the API client to retrieve the matching results to display.

For the flow of the filter, the UI provides various criteria for users to choose which allows users to narrow down the results they want to view based on their options. It uses the filterProject hook and filterDonation hook to fetch filtered data from the backend via the API client and update the respective charts.

III.VIII Form Modal Component



Router
[Container: e.g. SpringBoot, ElasticSearch, etc.]
Provides client-side routing to pages
User Management Page
[Container: React Component]
Main page for managing Users (Donors, Admins and Charities).
Project Management Page
[Container: React Component]
Main page for managing Projects.
User Form Modal UI
[Component: UI]
Render the user detail form modal
showUserDetail Hook
[Component: React Hook Form]
Handle calling the Show User Detail API from the backend
User Detail Form Modal
[Container]
Project Form Modal UI
[Component: UI]
Render the project detail form modal
showProjectDetail Hook
[Component: React Hook Form]
Handle calling the Show Project Detail API from the backend
Project Detail Form Modal
[Container]
API Client
[Container: TypeScript class]
Provides functional wrapper for back-end API calls, validates authentication state and incrementally refresh authentication token
Reusable Form Modal
[Component: React Component/Shadcn]
A validated reusable form for views using forms

Form Modal component diagram.

Form Modals are used for the CRUD-like actions in User Management and Project Management containers of the application. It makes sense to designate them into their own components since they already have multiple usages in the application scope and will be useful if the application grows and incorporates more CRUD-like actions.

State management and inner form data is managed by React Hook Form [19] in the headless layer. It lets us get the form data from each input just by providing each input with a name and then wrapping the form's submit handler with an interceptor provided by their hook in a JSON format.

The hook only controls the logic and styling is applied in the implementations (Modal Form in User Management Component and in Project Management Component).

III.IX Design Justification

III.IX.I Maintainability

Base components such as tables, modals, forms can be reused throughout the application; hence, it promotes reusability.

The UI of components can easily be updated or modified without impacting the logic behind it.

The application uses a centralized Router to manage navigation, making it easy to understand and update page transitions.

III.IX.II Extensibility

Since the headless components are easy to customize, new features can be added seamlessly.

Validation utilities and reusable form logic ensures consistency throughout the application, therefore new features can use these utilities without duplicating code.

III.IX.III Resilience

Isolated logic focused on being good at a single thing ensures that updates on some of the components won’t break functionality of the app [20].

III.IX.IV. Scalability

The headless components focus only on the logic and functionality layer while leaving the presentation layer to the reusable components, which separates two points of concern, giving developers two “interfaces” to further extend app’s functionality.

Furthermore, implementing strategies like lazy loading to reduce initial load times and pagination to handle data heavy pages increases the capacity of the system to handle increased load.

III.IX.V Security

User credentials and details are designed to be end-to-end encrypted using JSON Web Encryption (JWE) for secure data tranmission from every client to corresponding services.

Headless UI approach reduces reliance on third-party libraries and components, thus lowering the potential attack surface.

Client-side validation ensures that form input follows the required formats before being submitted to the backend, reducing the risk of injection attacks.

III.IX.VI Performance

Lightweight components minimize overhead, ensuring that only necessary parts are updated, thus improving rendering performance.

IV References

[1] J. S. Park, R. Sandhu, and G.-J. Ahn, “Role-based access control on the web,” ACM Transactions on Information and System Security, vol. 4, no. 1, pp. 37–71, Feb. 2001, doi: https://doi.org/10.1145/383775.383777 (accessed Dec. 1, 2024)

[2] J. Kreps, L. Corp, N. Narkhede, and J. Rao, “Kafka: a Distributed Messaging System for Log Processing.” Available: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf

[3] T. Dierks and C. Allen, “The TLS Protocol Version 1.0,” Jan. 1999, doi: https://doi.org/10.17487/rfc2246 (accessed Dec. 1, 2024)

[4] M. Jones, J. Bradley, and N. Sakimura, “JSON Web Token (JWT),” May 2015, doi: https://doi.org/10.17487/rfc7519 (accessed Dec. 1, 2024)

[5] Y. Wilson and Abhishek Hingnikar, “OpenID Connect,” Apress eBooks, pp. 103–126, Nov. 2022, doi: https://doi.org/10.1007/978-1-4842-8261-8_6 (accessed Dec. 1, 2024)

[6] S. Brady, “Understanding JSON Web Encryption (JWE),” Scott Brady, Aug. 17, 2022. https://www.scottbrady91.com/jose/json-web-encryption (accessed Dec. 03, 2024)

[7] “jwtk/jjwt,” *GitHub*, May 19, 2022. https://github.com/jwtk/jjwt

[8] “Chapter 27. High Availability, Load Balancing, and Replication,” PostgreSQL Documentation, Feb. 09, 2023. https://www.postgresql.org/docs/current/high-availability.html

[9] shadcn, “shadcn/ui,” shadcn/ui. https://ui.shadcn.com

[10] “React Router: Declarative Routing for React,” ReactRouterWebsite. https://reactrouter.com/

[11] "TypeScript-first schema validation with static type inference," GitHub. https://zod.dev/

[12] "Axios," axios-http.com. https://axios-http.com

[13] "supabase-js," *GitHub*, Apr. 30, 2023. https://github.com/supabase/supabase-js

[14] "Firebase Javascript SDK," *GitHub*, Dec. 07, 2022. https://github.com/firebase/firebase-js-sdk

[15] F. Skokan, "jose," *GitHub*, May 31, 2023. https://github.com/panva/jose

[16] "Zustand," *GitHub*, May 23, 2022. https://github.com/pmndrs/zustand

[17] A. Hunt, D. Thomas, and A. Wesley, "Pragmatic Programmer, The: From Journeyman to Master," 1999. Accessed: Dec. 03, 2024. [Online]. Available: https://www.cin.ufpe.br/~cavmj/104The%20Pragmatic%20Programmer%2C%20From%20Journeyman%20To%20Master%20-%20Andrew%20Hunt%2C%20David%20Thomas%20-%20Addison%20Wesley%20-%201999.pdf

[18] "artima - Orthogonality and the DRY Principle," Artima.com, 2024. https://www.artima.com/articles/orthogonality-and-the-dry-principle (accessed Dec. 03, 2024).

[19] "React Hook Form," *React-hook-form.com*, 2020. https://www.react-hook-form.com/ (accessed Dec. 03, 2024).

[20] Y. Prajwal, J. Parekh, and Rajashree Shettar, "A Brief Review of Micro-frontends." Available: https://uijrt.com/articles/v2/i8/UIJRTV2I80017.pdf

V Appendix

Appendix A: Data Model

The diagram is visually represented in Lucidchart, which can be access through this link: Chartian - Data Model

Appendix B: Container and Component Diagram for Back-end of Admin Application

The diagram is visually represented in draw.io, which can be access through this link: Charitan - Back End For Admin Application

Appendix C: Container and Component Diagram for Front-end of Admin Application

The diagram is visually represented in draw.io, which can be access through this link: Charitan - Front End For Admin Application

Appendix D: Github page for Team B Milestone 1, which can be accessed via this link: https://github.com/Charitan-Ace/milestone1-teamB