10/31/2025

CACODEV - Congolese Association for Congo Development.

Software Detailed Design Specification

By

David Katembo

Fabrice Kadima

Pemphyle Nzuzi

Indiana University Southeast

CSCI P445: Capstone Project I Design Fall 2025

Professor Ronald Finkbine, Ph.D.

Table of Contents

1. Data Design

1.1 Overview

The CACODEV management application consists of three main subsystems:

- The User Interface: A web-based client allowing administration and members to interact with the organization's data.
- Server-Side Services: The core business logic handling members, contributions, and events.
- Data Store: A relational database (PostgreSQL/MySQL) where persistent data is stored.

1.2 User Interface Data Objects

The user interface is a web application written in Angular (TypeScript, HTML, CSS). Below are the structures of the key data objects used in the frontend.

1.2.1 Member Data Object

```
{
 "id": "uuid-string-value",
 "memberId": "unique-member-id-string",
 "firstName": "String",
 "lastName": "String",
 "email": "String",
 "phoneNumber": "String",
 "active": true,
 "membershipExpiryDate": "2025-12-31",
 "memberType": {
  "id": "uuid-string-value",
  "name": "GOLD",
  "membershipFee": 100.00,
  "hasVotingRights": true
 }
}
```

1.2.2 Donation Data Object

```
{
 "id": "uuid-string-value",
 "amount": 50.00,
 "currency": "USD",
 "donationDate": "2024-10-15T10:30:00",
 "paymentMethod": "Credit Card",
 "isRecurring": false,
 "recurringFrequency": "ONE_TIME",
 "taxReceiptIssued": true,
 "memberId": "uuid-string-value"
}
```

1.2.3 Event Data Object

```
{
 "id": "uuid-string-value",
 "name": "Annual Gala",
 "description": "Fundraising event for the new community center.",
 "eventDate": "2024-12-20",
 "status": "SCHEDULED",
 "organizer": {
  "firstName": "David",
  "lastName": "Katembo"
 },
 "participantCount": 150
```

}

1.3 Server-Side Services

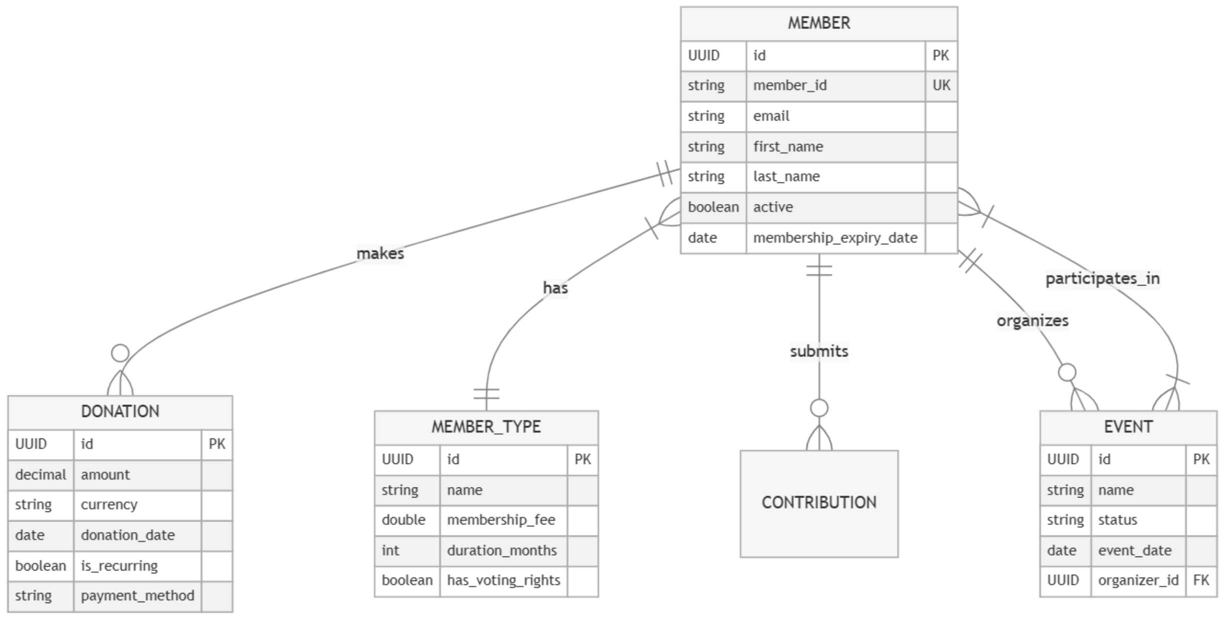
The server-side services subsystem is responsible for:

- Validating incoming JSON requests from the Angular client.
- Executing business rules.
- Interfacing with the Database Repository layer.

1.4 Data Store

The data store used in this application consists of a relational database management system (RDBMS).

1.4.1 Database Schema (ER Diagram)



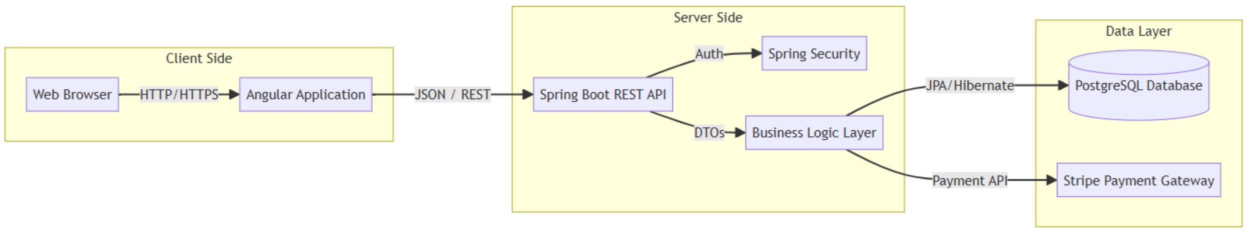
2. Architecture Design

2.1 Overview

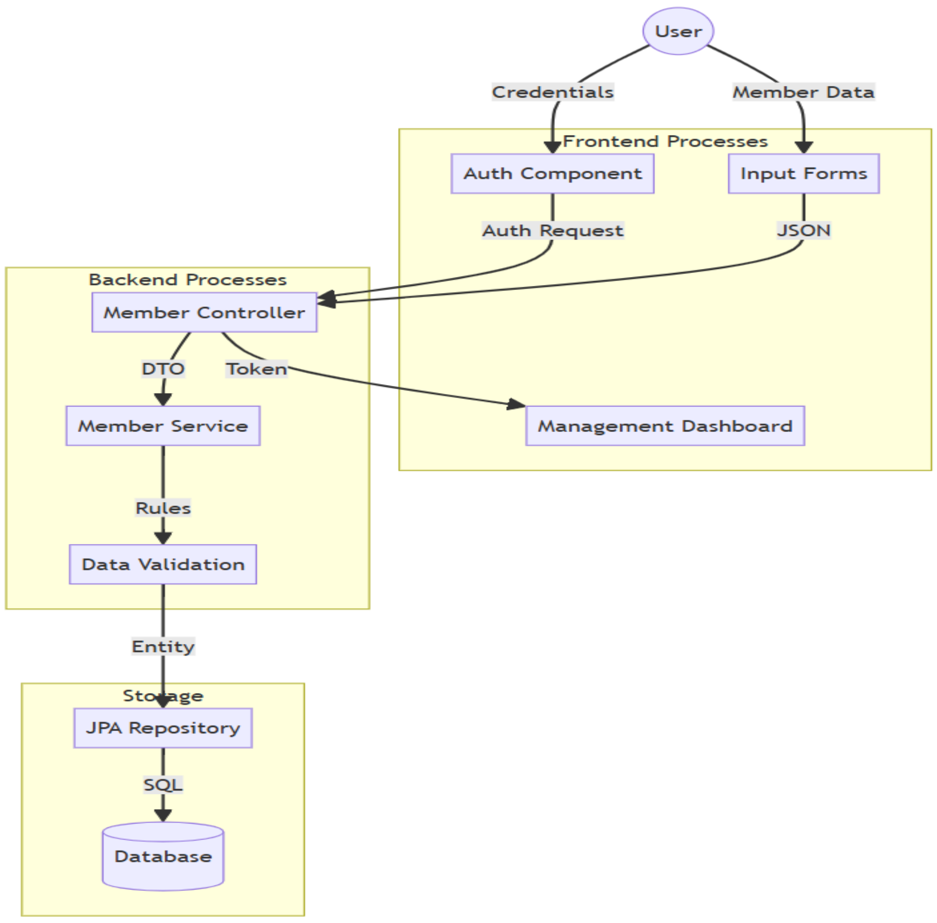
The CACODEV application utilizes a Client-Server pattern. The frontend (Angular) runs on the client's browser and communicates via REST API with the backend (Spring Boot) running on a remote server.

2.2 Architectural Models

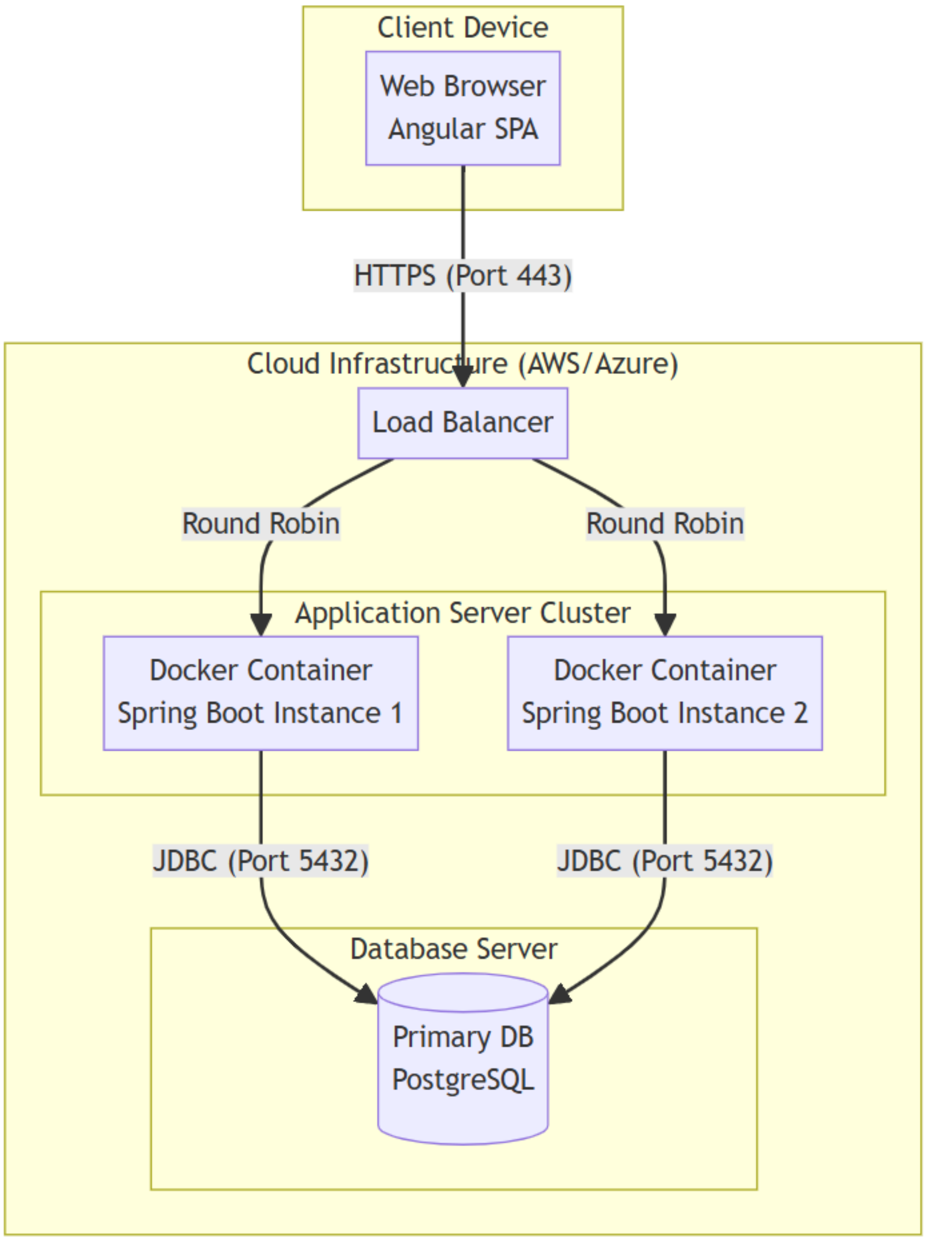
2.2.1 Architecture Block Diagram



2.2.2 Data Flow Diagram



2.2.3 Hardware-Software Mapping - Deployment Diagram

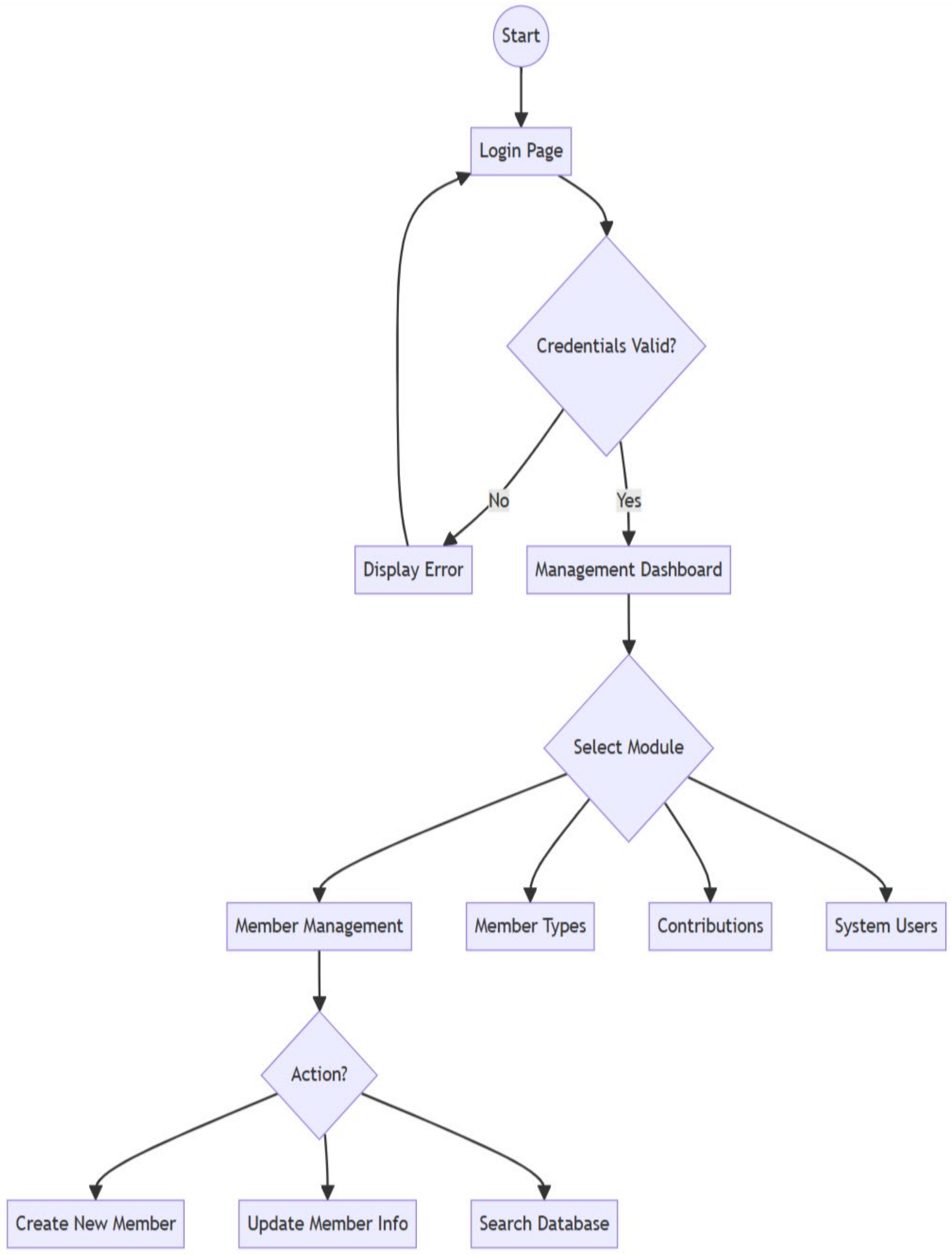


3. Interface Design

3.1 Overview

The User Interface is built with Angular Material. The interface is divided into public-facing pages and a secured administrative area.

3.2 User Interface Process Flow



3.3 User Authentication Page

Description: A clean form centered on the screen.

- Input: Username (Email) and Password.
- Output: JWT Token stored in local storage upon success.

3.4 Management Dashboard

Description: Contains a header "Management" and four primary navigation cards.

- Members Card: Navigates to the Member List.
- Member Types Card: Navigates to configuration for membership levels.
- Contributions Card: Navigates to the donation tracking table.
- Users Card: Navigates to internal system user settings.

3.5 Member Management Page

Description: Search, view, and edit alumni/member records.

- Layout: Top Bar (Search filters), Center (Data grid/table displaying member info).

4. Procedural Design

4.1 Overview

This section describes the procedural logic for the key operations identified in the architecture.

4.2 Component: Member Management

Introduction/Purpose of this Component: This component handles the registration of a new member. It ensures that the member does not already exist and that the assigned member type is valid.

Input for this Component:

- MemberCreateRequest object (Email, First Name, Last Name, Member Type ID).

Output for this Component:

- MemberDTO object (The created member record with generated ID).
- Error Message (If email exists or Type is invalid).

Component Process to Convert Input to Output: The process begins by receiving the creation request. It first queries the database to ensure the email address is unique. If unique, it generates a new internal Member ID. It then validates that the referenced Member Type exists. If all checks pass, it constructs a new Member entity, sets the active flag to false, calculates the expiry date, and persists the entity to the database. Finally, it converts the saved entity into a DTO for the response.

Design Constraints and Performance Requirements:

- Email must be unique in the system.
- Transaction must be atomic.

Process (pseudo-code algorithm):

```
FUNCTION createMember(request):
  // 1. Validation Phase
  IF database.existsByEmail(request.email) THEN
    THROW ResourceAlreadyExistsException("Email in use")
  END IF
  // 2. Generation Phase
  SET newId = generateUniqueMemberId()

  // 3. Dependency Check
  SET memberType = database.findMemberType(request.memberTypeId)
  IF memberType IS NULL THEN
    THROW ResourceNotFoundException("Invalid Member Type")
  END IF

  // 4. Object Creation
  CREATE newMember
  SET newMember.id = newId
  SET newMember.details = request.details
  SET newMember.active = FALSE
  SET newMember.expiry = NOW + memberType.duration

  // 5. Persistence
  database.save(newMember)
  RETURN convertToDTO(newMember)
END FUNCTION
```

4.3 Component: Personal Information Update

Introduction/Purpose of this Component: Updates specific fields of a member's profile. This operation is optimized to only update fields that have actually changed to reduce database overhead.

Input for this Component:

- UUID (Member ID).

- MemberUpdatePersonalInfoRequest (Updated fields).

Output for this Component:

- MemberDTO (The updated member record).

Component Process to Convert Input to Output: The system retrieves the existing member record using the provided ID. It then iterates through each field provided in the request (First Name, Email, etc.). For each field, it compares the new value with the current database value. If a value is different, it updates the entity and flags the record as "dirty." If any changes were detected, the system saves the updated entity to the database and returns the result.

Design Constraints and Performance Requirements:

- Must handle concurrent updates gracefully.
- Cannot update Member ID.

Process (pseudo-code algorithm):

```
FUNCTION updatePersonalInfo(id, request):
  // 1. Retrieval
  SET member = database.findById(id)
  IF member IS NULL THEN THROW NotFoundException

  SET isUpdated = FALSE

  // 2. Selective Update Logic
  IF request.firstName IS NOT NULL AND request.firstName != member.firstName THEN
    member.firstName = request.firstName
    isUpdated = TRUE
  END IF

  IF request.email IS NOT NULL AND request.email != member.email THEN
```

```
        member.email = request.email
        isUpdated = TRUE
    END IF

    // 3. Commit
    IF isUpdated IS TRUE THEN
        database.save(member)
    END IF

    RETURN convertToDTO(member)
END FUNCTION
```

4.4 Component: Event Participation

Introduction/Purpose of this Component: Links an existing Member to an existing Event.

Input for this Component:

- UUID Event ID.
- UUID Participant ID.

Output for this Component:

- EventDTO (Updated event with new participant count).

Component Process to Convert Input to Output: The system receives IDs for both an Event and a Member. It performs two database lookups to verify both entities exist. If either is missing, it returns an error. If both exist, it adds the Member entity to the Event's participant collection (establishing a relationship). The updated Event entity is then saved, and the updated DTO is returned.

Design Constraints and Performance Requirements:

- Transactional integrity required to prevent partial updates.

Process (pseudo-code algorithm):

```
FUNCTION addParticipant(eventId, participantId):
    // 1. Fetch Entities
    SET event = database.findEvent(eventId)
    SET member = database.findMember(participantId)

    // 2. Logic Check
    IF event IS NULL OR member IS NULL THEN
        THROW NotFoundException
    END IF

    // 3. Association
    ADD member TO event.participants

    // 4. Save State
    database.save(event)
    RETURN eventDTO
END FUNCTION
```

5. Breakdown of Individual Contributions

Team Member Name	Role	Specific Contributions
David Katembo	Team Leader / Architect	• System Architecture Design • Database Schema & ERD • Data Structure Definitions
Fabrice Kadima	Backend Developer	• Procedural Design • Service Layer Logic (Java) • API Implementation
Pemphyle Nzuzi	Frontend Developer	• Interface Design (UI) • Angular Component Structure • User Process Flows

6. Key Personnel Information

- David Katembo - Team Leader
- Fabrice Kadima - Development Team
- Pemphyle Nzuzi - Development Team

7. Major Methods & Routines

Class	Method	Description
MemberService	createMember	Validates email uniqueness, assigns a generated ID, calculates membership expiry based on type, and saves the new member as inactive.
MemberService	updatePersonalInfo	Fetches a member and selectively updates fields (Name, Email, Phone) only if the input differs from stored data.
MemberService	enableDisableMember	Toggles the active boolean flag for a specific member ID and saves the change.
MemberService	findExpiredMembership	Queries the repository for all members whose membership expiry date is before the current date.
MemberTypeService	create	Creates a new MemberType (e.g., "Gold"), ensuring the name is stored in uppercase and does not already exist.
EventService	createEvent	Links a valid Organizer (Member) to a new Event entity, sets the initial status to SCHEDULED, and saves it.
EventService	addParticipant	Retrieves an Event and a Member, adds the Member to the Event's participant list, and

		updates the database record.
ContributionService	updateStatus	Retrieves a specific Contribution and updates its status (e.g., from PENDING to COMPLETED).
	dataSource	Configures the HikariCP connection pool with database credentials and sets the maximum pool size to 10.