Togetherly

A Cross-Platform Volunteer Management Mobile App

Student

2366076

Project Dissertation

Swansea University
Prifysgol Abertawe

Department of Computer Science
Adran Gyfrifidureg

24th Apil 2026

Declaration

Statement 1

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed  ..... Farzod Badriddinov (2366076)

Date 24.04.2026.....

Statement 2

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by citations giving explicit references. A bibliography is appended.

Signed  ..... Farzod Badriddinov (2366076)

Date 24.04.2026.....

Statement 3

The University's ethical procedures have been followed and, where appropriate, ethical approval has been granted.

Signed  ..... Farzod Badriddinov (2366076)

Date 24.04.2026.....

Abstract

Many organizations still struggle with volunteer recruitment and coordination, mostly due to fragmented digital practices and limited integration between existing tools. Current platforms often focus on either volunteer discovery or organization management tools but rarely provide a unified solution that supports both. This forces volunteers to monitor multiple sources and organizations to use separate systems for communication and management.

This project presents Togetherly, a cross-platform mobile application designed to solve this problem, by integrating event discovery, application management and communication within a single system. The application allows organizations to create and manage volunteering opportunities, while volunteers can search for events, apply to participate and contact hosts within the platform. Developed as a full-stack system, the project focuses on delivering core functionality that supports both roles in a simple and accessible way.

The system demonstrated that integrating key features such as communication and location-based discovery into a single app can improve usability and reduce fragmentation in volunteer coordination. Developing this project also highlights the challenges of building a complete software system independently within limited timeframe, including design trade-offs, time constraints, and evolving requirements. Generally, this work shows how relatively simple, well-integrated solution can provide meaningful improvements over more complex but fragmented systems.

Acknowledgements

I would like to express my deepest thanks to Dr. Neal Harman. His expertise, direction and mentorship were the driving forces behind the development of this project, and I am profoundly grateful for the guidance he provided at every critical stage; this work would not have reached its current state without his invaluable support.

I also want to thank all my peers, whose insightful discussions and suggestions regarding the expansion and potential applications of this work were incredibly helpful to my progress.

Most importantly, I want to thank my friends and family for their unwavering belief in me. A special thanks goes to my friend, Temur, whose suggestion provided this project its name.

Finally, to my parents and my sister, your encouragement has been foundation of my academic career. I am especially grateful to my sister, whose care and late-night meals sustained me through the most demanding parts of this work. This achievement belongs to all of you.

Table of Contents

1. Introduction

Organizations looking for volunteers continue to face recruitment and retention challenges, and current digital practice is often fragmented. A literature review prepared for the NSW Government reports that lack of information about opportunities remains a meaningful barrier to participation, with 56% of surveyed young people stating that they did not know how to find out about volunteer opportunities [26]. Another research shows that many small nonprofit organizations still rely heavily on digital communication practices. In practice, this means that opportunities are frequently promoted across websites, newsletters, social media posts, and messaging groups, which can make discovery and coordination unnecessarily difficult.

1.1. Motivation

Togetherly is motivated by the idea that this process should be simpler and more cohesive. Instead of asking volunteers to monitor multiple platforms and asking organizations to repeat the same information across different platforms, the project proposes a single cross-platform mobile application in which organizations can post and manage events, while volunteers can discover opportunities, apply, communicate, and track participation in one place.

On a personal level, I wanted to build something genuinely useful that addresses a real-world problem while also allowing me to develop and demonstrate my technical skills. This project provides an opportunity to design and implement a complete full-stack system, covering areas such as mobile development, backend engineering, security, and data modelling. It also challenges me to manage a project independently within a limited timeframe, which helps develop both technical and project management skills.

Another important motivation emerged during the development of the project. The increasing relevance of artificial intelligence (AI) tools in software development encouraged me to explore their capabilities. During this process, I recognized that AI-assisted coding is becoming more common in professional practice. This created an opportunity to understand how such tools can be used effectively in a controlled and reflective way, rather than relying on them completely.

From a problem perspective, although several volunteering platforms already exist, many of them have their limitations. Some focus mainly on matching volunteers without supporting full event management, while others lack effective communication features. These gaps highlight the need for a more integrated solution. Therefore, this project aims to develop a platform that

improves both usability and coordination by bringing together event discovery, application, and communication into a single system.

1.2.Aims and Objectives

The overall aim of Togetherly is to demonstrate the ability to design, engineer, and evaluate a full stack project independently within a limited timeframe.

The main objectives of the project are:

1. To create a platform that helps people help people by enabling nonprofits and volunteers to connect easily, organize events, and communicate effectively, making volunteering more accessible and efficient.
2. To demonstrate software engineering and project management skills by building and delivering a complete full-stack application within a limited timeframe, using good development practices throughout the project.
3. To explore and apply mobile capabilities such as biometric authentication and location services to improve both technical understanding and application functionality.
4. To explore the practical capabilities and limitations of AI generation tools in software development and coding, while using them in a guided way where all key design decisions, validation, and code quality remain my responsibility.

1.3.Overview

This document begins with background research into volunteer recruitment, retention, engagement, and existing digital platforms. It then presents the system specification and design direction for Togetherly, followed by the main implementation decisions across backend, frontend, data management, security, and supporting tools. Later sections evaluate the system through testing and reflection, including what was learned from the development process and the use of AI-assisted tooling.

2. Background Research

2.1. Volunteer Recruitment and Engagement

The literature suggests that the problem is not simply a lack of digital tools, but a mismatch between how opportunities are advertised, how volunteers discover them, and how organizations sustain engagement after recruitment. The NSW review emphasizes that volunteering offers should align with the key motivations and benefits of potential volunteers, and it notes that people respond more positively to volunteering advertisements when the message matches their

personal motivations [26]. Volunteering Australia makes a similar argument, stating that identifying and enabling volunteers to fulfill their primary motivations plays a pivotal role in satisfaction and retention, and that recruitment messages aligned with those motivations are more effective and persuasive [31]. These findings imply that a useful digital platform should move beyond generic listings and instead support richer categorization, interest-based browsing, and more personalized matching between users and opportunities [26] [31].

Recent research in volunteer assignment strengthens this position. For example, Kaur, Pazour, and Ausseil propose an optimization framework that creates personalized task recommendation menus for volunteers, balancing organizational needs with volunteer preferences, skills, and motivations [19]. Their work is important because it shows that better matching is not only a usability improvement, but it can also improve assignment quality and volunteer satisfaction. In practical terms, this supports the inclusion of structured volunteer profiles, category-based event discovery, and recommendation logic within Togetherly. While the system does not implement automatic recommendation logic, it provides flexible filtering and search capabilities that help users efficiently find relevant opportunities instead of browsing large amounts of unrelated information [19].

Mobile delivery is particularly relevant in this context. Chen et al. describe a mobile volunteering matching system that allows organizations to promote ongoing volunteer activities on an interactive map, enabling users to browse nearby opportunities that match their interests [9]. This is significant because it shows how mobile devices can reduce the friction between intention and action. Instead of relying on generic posts, users can access location-based opportunities through a dedicated interface. For Togetherly, this supports the decision to implement a cross-platform mobile application with location-based browsing features. While the final version of the system does not include notifications or fully context-aware recommendation mechanisms, it allows users to filter and discover nearby events more efficiently compared to static platforms [9].

2.2.Existing Platforms

2.2.1. VolunteerMatch

VolunteerMatch is one of the largest platforms designed to connect volunteers with organizations [17]. It focuses mainly on helping users discover volunteering opportunities at scale. A key feature is its integration with LinkedIn, where opportunities can be shared through the LinkedIn Volunteer Marketplace, allowing users to showcase their volunteering experience as part of their CV [24].

One user reported finding five suitable candidates within one week, suggesting that VolunteerMatch is effective in recruitment. It allows organizations to reach a large audience and quickly identify suitable candidates. In addition, the platform is free to use, making it accessible for small nonprofit organizations [8, 29].

However, the platform mainly focuses on matching rather than full event management. It does not provide strong support for tracking participation or managing volunteer behavior after application. Reviews also highlight issues such as unreliable attendance and low accountability.

To address these issues, in my application I implemented an attendance tracking feature, where organizations can record whether a volunteer attended an event. This helps improve coordination and accountability.

2.2.2. Volunteero

Volunteero focuses more on organizational workflows rather than large-scale discovery. It provides tools for managing volunteers, including registration, scheduling, and coordination [33]. The platform replaces traditional methods such as spreadsheets and email communication with a centralized system [35].

Reviews suggest that Volunteero performs well in managing events and reducing administrative workload. Features such as self-registration and automatic capacity control improve efficiency and prevent overbooking. The platform is also considered easy to use, even for users with limited technical experience [33].

However, the platform operates on a paid model, which makes it more suitable for larger organizations. Smaller groups or informal communities may find it less accessible.

In contrast, I designed Togetherly as a free mobile application to improve accessibility. The system includes similar core features, such as application management and in-app messaging, but focuses on delivering them in a simpler and more accessible way. Due to time constraints, advanced administrative tools such as analytics dashboards and complex reporting were not implemented, and priority was given to core interaction features [9].

2.2.3. Golden Volunteer

Golden Volunteer is another platform focused on improving efficiency through automation. It includes features such as email scheduling, push notifications, and QR-code attendance tracking, as well as reporting dashboards for organizations [22].

Reviews highlight that Golden is strong in automation and data reporting, making it particularly useful for larger organizations that need to manage a high volume of volunteers.
However, some feedback points to limitations in the accuracy of its matching system. In particular, recommended opportunities are not always relevant to the user's location, which reduces usability and engagement. This limitation lowers the overall relevance of suggestions and can lead to decreased user involvement over time [22].

In Togetherly, I focused more on improving basic usability and relevance rather than advanced automation. For example, I implemented location-based filtering using geographic queries in the backend, allowing users to find events within a specific radius. I also implemented a simpler version of their attendance tracking feature. These features improve coordination and the relevance of search results without introducing complex systems [9].

2.2.4. Identified Gap and System Positioning

From analysing these platforms, I realized that they focus on different parts of the volunteering process. VolunteerMatch focuses on discovery and recruitment, while Volunteero and Golden focus more on management, coordination, and organizational workflows. This separation highlights a gap in existing solutions.

Similar platforms either help users find opportunities or help organizations manage them, but they do not fully integrate both aspects into a single system.
Togetherly is designed to address this gap by combining key features into a single mobile application, within the scope of the project. However, it is important to note that the system focuses on core functionality rather than advanced enterprise features. For example, it does not include automated recommendation algorithms, custom analytics dashboards, or automated notification systems, which were initially considered but not implemented due to time constraints.

3. System Specification

The system is designed as a cross-platform mobile application that provides a unified platform for interaction between organizations and volunteers. Organizations can create and manage volunteering events, while volunteers can search for opportunities based on their interests and location, and apply to participate.

I designed the system in a way that removes the need for external communication tools such as messaging applications. Users can interact through event-based comments and direct messaging, allowing coordination and discussion to take place within the app (see Appendix A).

3.1.System Entities and Roles

The system includes two main user roles: **volunteers and organizations**, each with specific responsibilities and permissions. Initially, I planned to include an administrator role for moderation, but due to time constraints, I prioritized implementing advanced features instead. These roles interact with core entities such as events and comments, which support the creation and management of volunteering opportunities.

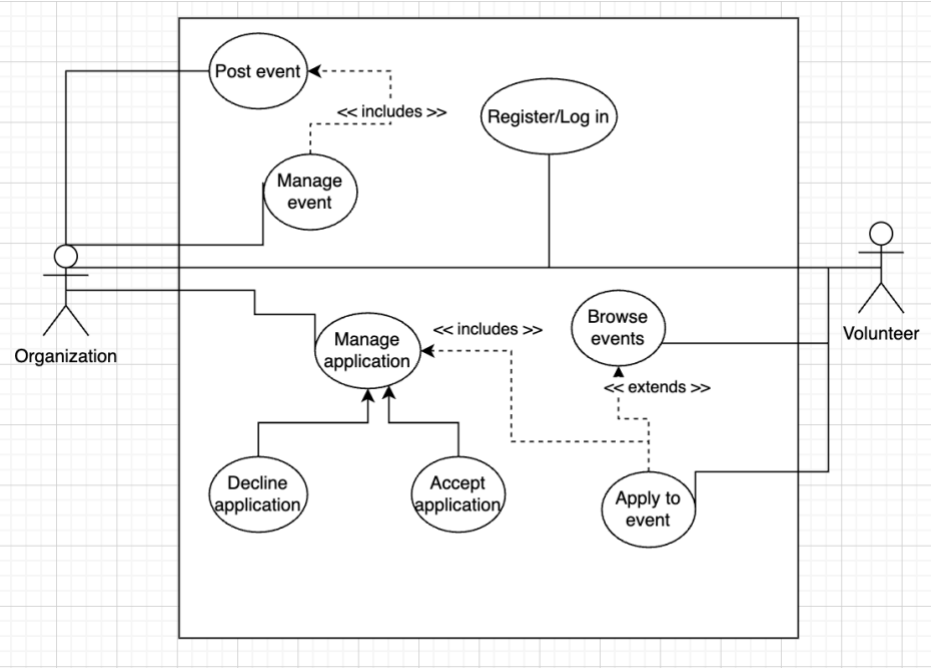


Figure 1. Use case diagram.

User Roles:

1. **Volunteers**

Volunteers represent individuals who are looking for volunteering opportunities. They can create and manage personal profiles, including information such as skills, interests, and preferred location range.
Their main interaction with the system involves discovering and applying to events. Volunteers can browse available opportunities, filter them based location or categories, and submit applications to participate. In addition, they can communicate with organizations through comments and direct messaging and track their participation history through attended events.

2. **Organizations**

Organizations represent entities that create and manage volunteering opportunities. They are responsible for publishing events and defining relevant details such as descriptions, requirements, and location.
Organizations also manage volunteer participation. This includes reviewing applications submitted by volunteers and either accepting or declining them. Furthermore, they also can communicate with volunteers through comments and messaging features.

3.2.Core Features

The following features represent the essential functionality I built that were required for the system to achieve its main goal of connecting organizations with volunteers.

- User Registration and Authentication

I implemented a secure registration and login system. During registration, users choose their role (volunteer or organization), which determines the functionality available to them (see Appendix A). Authentication is handled using standard email and password mechanisms, following common security practices.

- Role-Based Functionality

Each user role has access to different features within the system. Organizations can create and manage events, while volunteers can search for events and apply to participate. This separation ensures that users only interact with features relevant to their role (see Appendix A).

- Commenting System

I implemented a commenting feature that allows volunteers and organizations to communicate directly on event pages. This enables discussions, questions, and updates to remain linked to specific events (see Appendix A figure A.14).

- Attended Events Tracker

Volunteers can track the events they have attended through their profile. This includes viewing the total number of events completed within a specific period. This also could be extended to be a dashboard of the best volunteers, to increase user engagement and encourage participation, but due to limited timeframe was not (see Appendix A figure A.9).

- Location Awareness

The application includes location-based filtering. Users can search for events within a specified radius from their current location. This feature improves usability by helping volunteers discover nearby opportunities in real time (see Appendix A figure A.11).

- Search and Filtering System

I implemented a search and filtering system that allows users to find events based on criteria such as interests, skills, and distance. This improves the matching process between volunteers and suitable opportunities (see Appendix A figure A.10).

3.3.Additional Features

After completing the MVP, I added several additional features to enhance user experience. These features are not essential for the system to function but improve usability and convenience.

- Calendar Integration

I implemented a feature that allows users to add events directly to their device calendar. When a user selects this option, the system opens the calendar application with pre-filled event details such as title, date, and time. The user can then edit or confirm the event before saving it. This reduces the need for users to manually track event schedules within the app and allows users easily add events into their schedule (see Appendix A figure A.16).

- Biometric Authentication

The system supports biometric authentication, such as facial recognition. This improves both security and user convenience by allowing faster login without entering credentials manually (see Appendix A figure A.15).

- Real-Time Chat System

I implemented a real-time chat feature that allows direct communication between organizations and volunteers within the platform. This ensures that all communication remains centralized. Due to time constraints, group chat functionality was not implemented, but it could be added in future versions of the system (See Appendix A figures A.12, A.13).

- Attendance Checker for Organizations

This feature was added after analyzing existing platforms, where a common issue was volunteers registering but not attending events. To address this, I implemented an attendance tracking feature that allows organizations to mark whether volunteers attended the event. In future, this

could be extended into a rating system or used to restrict users who repeatedly fail to attend events (See Appendix A figure A.7).

4. Implementation

4.1. Backend: Spring Boot

The backend was implemented using Spring Boot [30]. One of the main reasons for this choice was my prior experience with the framework, which allowed for faster and more efficient development. Alternative backend frameworks were also considered, including Django (Python), Express.js (JavaScript), and Laravel (PHP) [10, 11, 20]. A more detailed comparison and of these technologies is provided in the initial project document, where different options were evaluated in greater depth [6].

4.1.1. Layered Architecture

The application follows a layered architecture based on the Model-View-Controller (MVC) pattern, which separates the system into controller, service, and repository layers (see Figure 2). In this project, the controller layer is responsible for handling incoming HTTP requests and returning responses, the service layer contains the business logic, and the repository layer manages interaction with the database through Spring Data JPA and Hibernate [2]. This separation was important because it prevented controllers from becoming overloaded with application logic or direct database access. Instead, each layer had a clear responsibility, which made the backend easier to maintain, test, and extend
(see Figure 2).

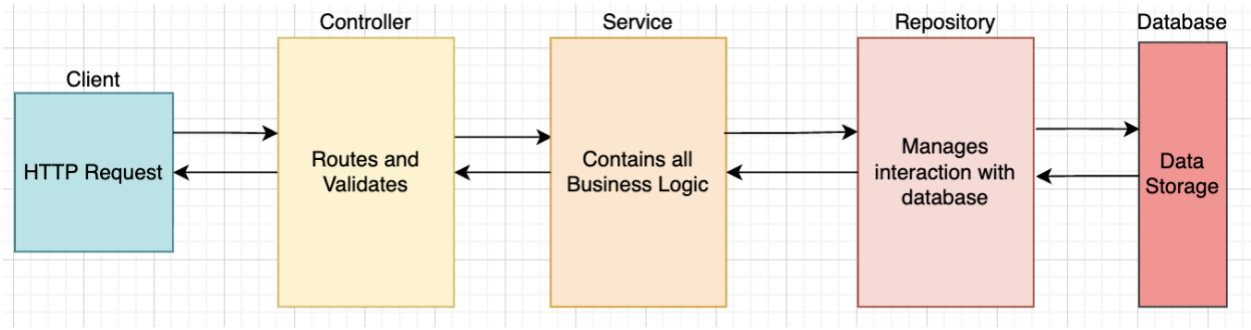


Figure 2. Backend request flow through controller, service, and repository layers.

For example, controllers such as `EventController`, `CommentController`, and `ChatController` define the available endpoints and accept request data from the client. However, they do not perform database operations directly. Instead, they delegate work to

service classes such as `EventService`, `CommentService`, and `ChatService`, where the main business rules are implemented. These service classes then use repository interfaces such as `EventRepository`, `CommentRepository`, `ConversationRepository`, and `MessageRepository` to retrieve or store data.

4.1.2. API Design

The API was designed around resource hierarchy. Instead of mixing unrelated endpoints together, related functionality was grouped under common paths. For example, event-related operations are grouped under `/api/v1/events`, comments are nested under `/api/v1/events/eventId/comments`, and chat functionality is grouped under `/api/v1/chat`. This made the API structure more logical and easier to understand from the frontend side, because endpoints reflected the relationships between resources in the application [36].

This resource-based design also helped make the backend more consistent. For example, creating an event, retrieving an event, browsing events, applying to an event, and viewing event applications are all handled in the event-related controller. Similarly, comments are treated as part of a specific event, so their routes are nested under the corresponding event identifier. This structure reflects the actual domain model rather than exposing disconnected endpoints.

The backend follows REST-style design principles in the way it uses HTTP methods. `POST` is used when creating records, `GET` is used for retrieval, `PUT` and `PATCH` are used when updating state, and `DELETE` is used for removal. This made the API predictable and easier to integrate with the frontend (see Appendix B).

4.1.3. DTOs and Mappers

To transfer data between the frontend and backend, the system uses Data Transfer Objects (DTOs). A Data Transfer Object is a class used specifically to carry data between layers without exposing the internal entity model directly. This was important because the database entities should not always be returned to the client as they are stored internally. DTOs allowed the API to control exactly what data is sent and received, while also creating a more stable contract for the frontend [7].

Request DTOs are used to validate and receive incoming client data. Examples include `EventCreateDto`, `CommentCreationDto`, and `SendChatMessageRequestDto`. Lombok annotations such as `@Getter` and `@Setter` were used to reduce boilerplate code by automatically generating accessor methods [28]. Validation is also applied at this stage to ensure that incoming data meets required constraints (see Appendix C).

Response DTOs were used to return clean and structured data to the frontend. I implemented a generic `ResponseDto<T>` wrapper, which gives responses a consistent structure containing fields such as ‘success’, ‘data’, ‘message’, and optional ‘metadata’. This allowed frontend to rely on a common response format across different endpoints. Examples of these dynamic response structures can be found in Appendix D.

Mappers were used to convert between entities and DTOs. In this project, MapStruct was used for this purpose [5]. For example, `EventMapper`, `CommentMapper`, and `ChatMapper` map internal entity objects into response DTOs and, where necessary, convert request DTOs into entities. This reduced repetitive boilerplate code and kept transformation logic outside of the controllers and services (see Appendix E).

4.1.4. Business Logic and Validation

I implemented a two-level validation in the system to ensure data accuracy. At the API boundary, DTO classes such as `EventCreateDto` use validation annotations to ensure that incoming data meets required constraints. In addition to this, the service layer applies rule-based validation that depends on system behavior. This approach effectively separates concerns, DTO validation ensures that input data is structurally correct, while service-level validation ensures that actions are logically valid within the context of the system. I applied the same pattern across different services, including `EventService`, `CommentService`, and `ChatService`.

A clear example of this service-layer logic can be seen in the event application. I added a validation in `EventService` which verifies whether a volunteer has already applied to the same event. This is done using `EventApplicationRepository` through the method `existsByEventIdAndVolunteerId(...)`. If a duplicate is detected, an `EntityExistsException` is thrown with an appropriate error message (see Figure 3). This approach is considered good practice because the database ensures data integrity, while the service layer allows the system to return clearer and more user-friendly feedback [27].

```java
@Transactional  1 usage   Fbdrv
public ResponseDto<UUID> applyToEvent(UUID id) {
    var currentUser = currentUserService.getCurrentUser();
    var event = eventRepository.findById(id)
            .orElseThrow(() -> new EntityNotFoundException("Event not found"));
    var volunteer = volunteerRepository.findById(currentUser.getId())
            .orElseThrow(() -> new EntityNotFoundException("Volunteer not found"));

    if (eventApplicationRepository.existsByEventIdAndVolunteerId(event.getId(), currentUser.getId())) {
        throw new EntityExistsException("You already have an application for this event");
    }
```

Figure 3. Example of service-level checks.

4.1.5. Error Handling

Error handling in the backend is centralized using a global exception handler. Instead of writing error-handling logic separately in every controller, exceptions thrown in the application are intercepted by a `GlobalExceptionHandler`, which converts them into consistent HTTP responses. This ensures that the frontend receives meaningful status codes and messages when something goes wrong.

The handler covers several common cases. For example, missing entities return '404 Not Found', access violations return '403 Forbidden', validation failures return '400 Bad Request', duplicate-record situations return '409 Conflict', and invalid request payloads are also handled as bad requests. This approach is considered a common practice in backend development, as it improves consistency of API responses and reduces duplication by centralizing error handling [27]. An example of this implementation can be found in Appendix F.

4.1.6. Real-Time Chat Implementation

One of the more advanced backend features in this project is the real-time chat system. It is built with Spring WebSocket support and STOMP (Simple Text Oriented Messaging Protocol), which enables bidirectional communication between the client and server [21]. The WebSocket endpoint is configured at `/ws/chat`, the application destination prefix is `/app`, the user destination prefix is `/user`. This allows the backend to support live message delivery in addition to standard HTTP requests.

The chat functionality is split between REST and WebSocket responsibilities. REST endpoints in `ChatController` are used to retrieve conversations and paginated message history, while `WebSocketController` handles real-time message sending and delivery. Using both of

them was useful because chat history and new messages have different requirements, this allowed to handle chat history through HTTP methods and live messages through WebSocket.

Authentication in WebSocket is the same JWT-based security model used elsewhere in the backend. During STOMP connection setup, the backend uses a channel interceptor to extract and validate the bearer token and associate the authenticated user with the WebSocket session. This is necessary because WebSocket communication does not automatically reuse the standard HTTP request authentication flow in the same way as REST endpoints.

Within the chat service, the backend either retrieves an existing conversation between two users or creates a new one if none exists. To avoid duplicate conversations, the database enforces uniqueness for the participant pair. When a message is sent, its content is validated, persisted in the database, and then delivered in real time through `SimpMessagingTemplate` to the sender's and recipient's user-specific queues. Additional checks ensure that users cannot send messages to themselves and that only participants in a conversation are allowed to view its messages. This keeps the direct messages feature both functional and consistent with the application's service layer security.

4.1.7. Filtering and Location-Based Event Search

Another important backend feature is event filtering, including location-based filtering. This functionality was implemented in the event service and repository layers. The `/api/v1/events/browse` endpoint allows events to be filtered by category, search query, and geographic parameters such as latitude, longitude, and radius (see Figure 4).

```
@PreAuthorize("hasRole('VOLUNTEER')")  Fbdrv
@GetMapping("/browse")
public ResponseDto<List<EventDto>> browseEvents(
        @RequestParam(required = false) Set<EventCategory> categories,
        @RequestParam(required = false) String q,
        @RequestParam(required = false) Double latitude,
        @RequestParam(required = false) Double longitude,
        @RequestParam(required = false) Double radiusKm
) {
    return eventService.browseEvents(categories, q, latitude, longitude, radiusKm);
}
```

Figure 4. Event browsing endpoint with category, search, and location-based filtering.

The interesting part of this implementation is that location filtering is not done purely in application code. Instead, the backend delegates spatial searching to the database using a repository query with 'ST_DWithin'. This allows PostgreSQL to return events that fall within a specified radius from a given point. If explicit coordinates are not provided in the request, the service falls back to the current volunteer's stored home location and commutable distance. This makes the feature more user-friendly because volunteers can browse nearby events without manually entering their location every time. The implementation of this query is shown in the Figure below.

```
    @Query(value = """  1 usage   Fbdrv
            SELECT * FROM events e
            WHERE e.location IS NOT NULL
              AND ST_DWithin(
                  e.location::geography,
                  ST_SetSRID(ST_MakePoint(:lng, :lat), 4326)::geography,
                  :radiusMeters
              )
            """, nativeQuery = true)
    List<EventEntity> findNearby(
            @Param("lat") double lat,
            @Param("lng") double lng,
            @Param("radiusMeters") double radiusMeters
    );
}
```

Figure 5. PostgreSQL ST_DWithin query used for location-based event filtering.

4.1.8. Security and Access Control

The backend implements security at both request and method levels using Spring Security. I used JSON Web Tokens (JWT), which are digitally signed tokens used to securely transmit information between parties to manage authentication. To make sure my backend somehow identifies the JWT created by an external provider I implemented `KeycloakRealmRoleConverter`. This class converts the roles provided by Keycloak into Spring Security authorities [16]. This allows endpoints to be protected using annotations such as `@PreAuthorize` (see Figure 6). A more detailed discussion of the authentication and Keycloak integration is provided in Section 4.4.

This was important because different roles in the application need different permissions. For example, only organizations can create events or manage attendance, while only volunteers can apply to events. Although these rules are enforced through method-level annotations, I also implemented additional checks inside the service layer. These checks are necessary when ownership or participant-specific access must be verified more precisely (see Figure 3).

This combination of declarative security and service-level checks made the access control more robust. Because the backend does not rely solely on frontend restrictions, unauthorized operations are blocked even if a client attempts to call the endpoint directly.

```
@PreAuthorize("hasRole('ORGANIZATION')")  Fbdrv
@PostMapping
public ResponseEntity<ResponseDto<UUID>> createEvent(@Valid @RequestBody EventCreateDto eventCreateDto) {
    return ResponseEntity.status(HttpStatus.CREATED)
            .body(eventService.createEvent(eventCreateDto));
}
```

Figure 6. Example of role-based access control using @PreAuthorize in a controller.

4.1.9. Time Management and Work Consistency

One of the challenges during development was time management. Initially, I followed a Kanban-based workflow as outlined in the initial project document, with weekly tasks and regular progress tracking [6]. In theory this seemed like a disciplined approach, but in practice it became unnecessarily complex for a solo project. There were periods where I would do large amount of work in one week, followed by very little progress in the next.

This inconsistency affected the pace of backend development. Working on a system that includes multiple layers, like security, persistence, API design, and testing requires steady progress. Due to time constraints and the need to balance this project with other modules, longer gaps between development sessions made it more difficult to resume work efficiently and maintain progress.

To address this, I simplified the workflow. Instead of following a strict weekly task structure, progress was reviewed roughly every two weeks. The focus shifted away from strict planning towards maintaining steady development. This made the process more manageable and reduced the complexity of trying to maintain a workflow that was too formal for a solo project like this.

I believe this issue was mainly due to limited experience in managing a long-term independent project. A more effective approach would have been to prioritize consistency from the beginning by doing smaller amounts of work regularly, rather than large amounts occasionally. That would likely have improved productivity, reduced stress, and helped me finish the MVP earlier.

4.2. Frontend: Flutter

4.2.1. Application Structure and Frontend Architecture

The frontend of the system was developed as a Flutter mobile application. The entry point is defined in `main.dart`, where the root `MaterialApp` is configured and the initial screen is launched. From there, the application is structured into three main layers: screens, services, and

models (see Figure 7). This structure is widely recommended for Flutter applications, and I considered it as the most suitable approach for this project [12].

Screens are organized inside the pages directory, where each file represents a specific part of the user interface, such as login, event browsing, event details, profile management, and direct messages. Backend communication and device-related functionality are handled within the services layer. This includes HTTP requests, authentication token management, location access and calendar integration. Data mapping is handled in the models layer, where classes such as `Event`, `Comment`, `ChatMessage`, and `ResponseDto` are used to transform backend JSON into strongly typed Dart objects.

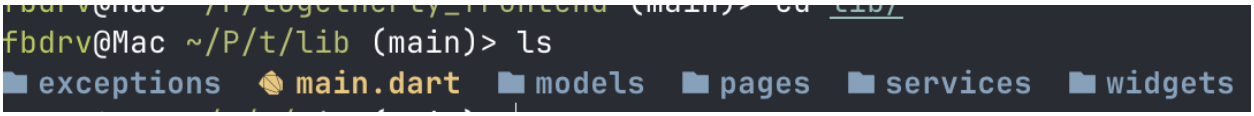


Figure 7. Flutter project structure.

4.2.2. Navigation and Main Screen Flow

Navigation in the application is designed around user roles, as the system supports both volunteers and organizations. The first screen, `ChooseRolePage` in `choose_role_page.dart`, allows the user to select a role or log into an existing account. After authentication, the user is redirected to either `MainPageVol` or `MainPageOrg`, depending on the role extracted from the access token in `login_page.dart` (see Appendix A for examples of these pages).

The main interface uses tab-based navigation. In both versions of `main_page.dart`, an `IndexedStack` is combined with a `BottomNavigationBar`. I chose `IndexedStack` because it preserves the state of each tab instead of rebuilding screens when switching between them. This improves the user experience, as lists, search inputs, and chat views remain stable during navigation, and is more efficient since tabs are not destroyed and rebuilt when switching between them [3].

Navigation between pages is handled using `Navigator.push` and `MaterialPageRoute`. For example, in the volunteer flow, users move from `login_page.dart` to `events_feed_page.dart`, then to `event_info_page.dart`, where they can apply to events (see Figure 8). A similar structure is used for organizations, but with a focus on event management and communication features.

```
53        Widget destination;
54        if (roles.contains(element: "volunteer")) {
55          destination = const MainPageVol();
56        } else if (roles.contains(element: "organization")) {
57          destination = const MainPageOrg();
58        } else {
59          throw Exception(message: "Unknown role: $roles");
60        }
61
62        Navigator.pushAndRemoveUntil<dynamic>(
63          context: context,
64          newRoute: MaterialPageRoute<dynamic>(builder: (BuildContext _) => destination),
65          predicate: (Route<dynamic> route) => false,
66        );
```

Figure 8. Role-based redirection to pages based on user roles.

4.2.3. User Interface Design and User Experience

The user interface is built using standard Flutter widgets such as `Scaffold`, `SafeArea`, `Column`, `Row`, `ListView`, `GridView`, `FutureBuilder`, `TabBar`, and `BottomNavigationBar`. I aimed to maintain a consistent visual design across the application by reusing common UI elements such as rounded cards, neutral background colors, and consistent button styles.

Forms and input validation are implemented across multiple screens. In `login_page.dart`, a structured login form provides feedback for loading and validation states. Separate registration flows are implemented in `register_form_page_vol.dart` and `register_form_page_org.dart`, as volunteers and organizations require different input fields. Validation is performed before sending data to the backend, which helps prevent invalid submissions and improves user feedback (see Appendix A).

Asynchronous data is handled using FutureBuilder, which connects backend responses to the UI state. This allows the interface to display loading indicators, error messages, and content dynamically without blocking the application.

To improve user experience, I also implemented feedback mechanisms such as `SnackBar` messages for actions including login, registration, event applications, attendance updates, and messaging (see Appendix A).

4.2.4. Authentication and Keycloak Integration

Authentication in the frontend is managed through Keycloak. When a user attempts to log in using their email or username and password, a request is sent to the Keycloak token endpoint to obtain an access token and a refresh token (see Figure 9). This process is implemented in `auth_service.dart`, where the returned response is mapped into the `AuthResult` class

defined in `auth_result.dart`. After a successful login, the token payload is decoded using the 'extractKeycloakRoles()' method in `auth_service.dart` (see Appendix I Figure I.2 for an implementation of this method). The extracted role information is then used to determine whether the user should be redirected to the volunteer or organization interface (see Figure 8).

To maintain authentication across sessions, I used the `flutter_secure_storage` package to store both the access token and refresh token in `token_service.dart` I chose this approach because secure storage is more appropriate for sensitive data than standard local storage. When the application later makes API requests, the required token is retrieved from secure storage and attached to the request as a Bearer token. This keeps the authentication logic centralized and avoids repeating token-handling code across multiple screens. Logout is implemented by clearing the stored tokens and redirecting the user back to the initial screen.

4.2.5. API Integration with the Spring Boot Backend

The frontend communicates with the backend using the http package. I tested this connection early in development to ensure that the mobile application could successfully interact with backend endpoints before starting to implement the app.

API interaction is primarily handled in `events_service.dart`. This service includes methods for retrieving events, viewing event details, applying to events, managing comments, and updating attendance. Each method constructs an HTTP request, sends it to the backend, processes the response, and converts the result into strongly typed model objects (see Appendix J for an implementation of this).

Since backend returns a consistent `ResponseDto<T>`, similar one was implemented in frontend for easier mapping and decoding of the results. This simplifies error handling and ensures consistent behavior across different features.

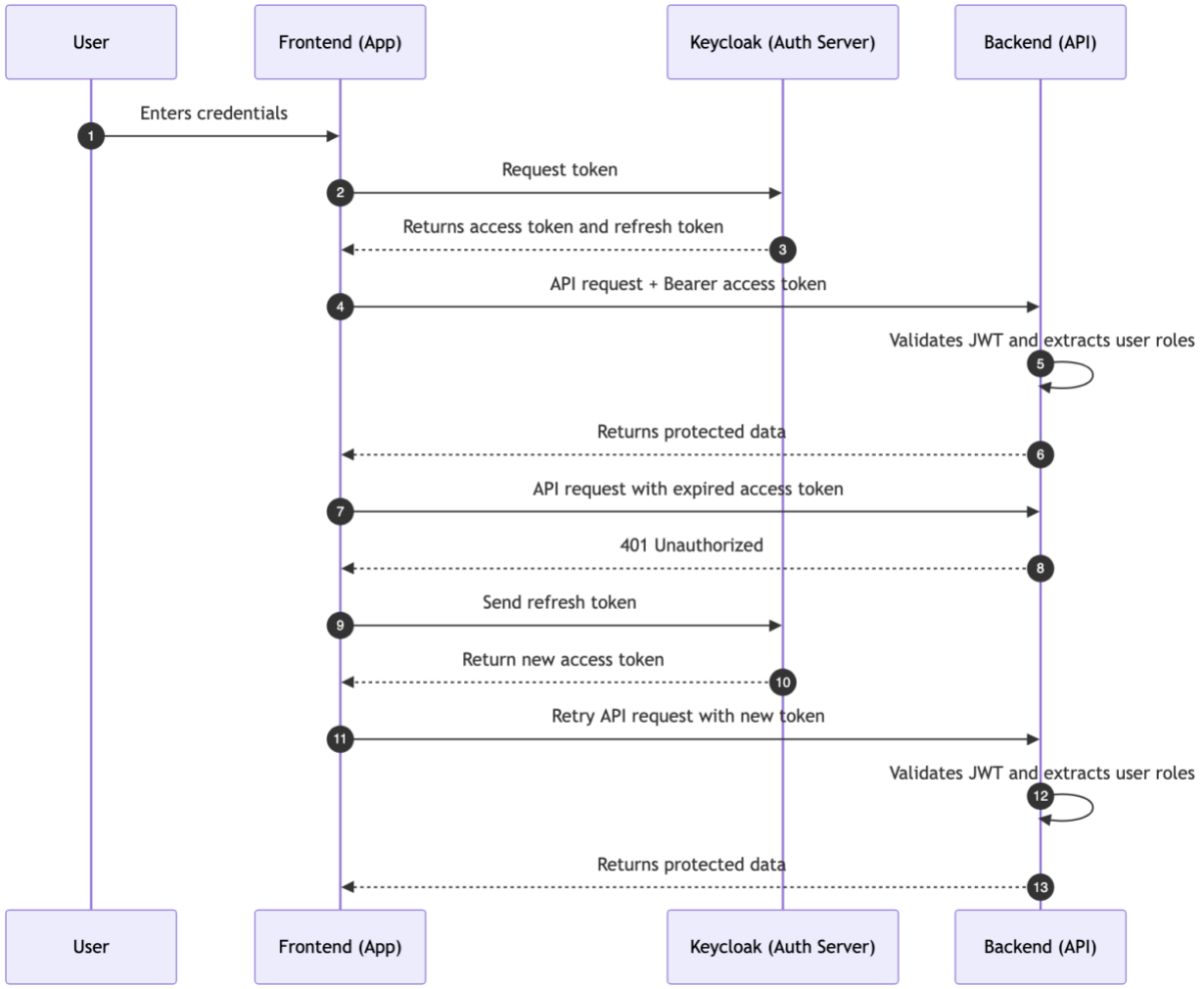


Figure 9. Shows the complete authentication lifecycle using Keycloak and JWT.

4.2.6. Event Browsing, Filtering, and Application Flow

Event browsing is implemented in `events_feed_page.dart`, where event data is requested from the backend and stored as a Future. The FutureBuilder widget is then used to connect this data to the UI, allowing the page to react dynamically to loading, success, and error states.

To improve usability, several filtering mechanisms are implemented. A text search field allows users to search events by name. Category filtering is implemented using a modal bottom sheet, where categories are loaded from the backend and selected by the user. Selected categories are then sent as query parameters in backend requests.

Location-based filtering is implemented using `LocationService` in `location_service.dart`. This service handles permission checks, retrieves the user's

current coordinates, and optionally converts them into a readable address. These coordinates are passed to the backend, which performs distance-based filtering. I chose this approach so that the frontend remains responsible for collecting input, while the backend handles more complex filtering logic.

Event details and application functionality are implemented in `event_info_page.dart`. When an event is opened, updated data is requested from the backend instead of relying only on previously loaded information. This ensures that dynamic data, such as participant counts or comments, remains accurate. When a user applies to an event, the request is sent through `events_service.dart`, and feedback is displayed using a `SnackBar`.

4.2.7. Comments

The commenting system is implemented as a reusable component in `comments_bottom_sheet.dart`. I designed it this way so that both volunteer and organization views could share the same functionality. When opened, the component loads comments from the backend and displays them in a scrollable list. The Comment model from `comment.dart` is used to map and format the data.

When a comment is submitted, it is sent to the backend, and the interface is updated immediately. I clear the input field, remove focus from the keyboard, and refresh the list so the new comment appears without requiring additional actions from the user.

4.2.8. Real-Time Chat Implementation

On the frontend, I implemented the chat feature within the Flutter application with a focus on managing user interaction and maintaining a responsive interface. The main responsibility of the frontend is to load conversation data, establish a connection for receiving updates, and ensure that the user interface reflects changes in real time.

The main logic is implemented in `chat_service.dart`. This service is responsible for retrieving conversation lists and message history from the backend, and for maintaining the live connection used for incoming messages. To support multiple Flutter platforms, I introduced an abstraction layer using `chat_socket.dart`, `chat_socket_io.dart`, `chat_socket_web.dart`, and `chat_socket_stub.dart`. This allows platform-specific connection details to be handled separately, while keeping the overall chat logic consistent across the application.

When new messages are received, they are processed within the service layer and converted into `ChatMessage` objects. This keeps data handling separate from the user interface and ensures that the UI only works with structured data. Similarly, when a user sends a message, the request

is passed through the service layer, which forwards it to the backend. This separation follows the structure used in other parts of my application.

The user interface is implemented in `chat_conversations_page.dart` and `chat_conversation_page.dart`. In the conversations page, I load the list of existing conversations and update it when new messages appear. In the individual conversation view, I display the message history and dynamically append new messages as they arrive. I also implemented automatic scrolling so that the most recent message becomes visible. This improves usability and makes the chat experience feel immediate and natural.

4.3. Database: PostgreSQL

The database layer of the system was implemented using PostgreSQL [15]. This choice was made because the application relies on structured and highly related data, including users, organizations, volunteers, events, applications, comments, conversations, and messages. A relational database was therefore more suitable than a non-relational alternative, as the system depends on clearly defined relationships, referential integrity, and efficient querying across multiple tables [6].

The implemented schema is structured around several core entities that reflect the main features of the application (see Appendix H). The central entity is `UserEntity`, which represents the base user model. Role-specific data is separated into `VolunteerEntity` and `OrganizationEntity`, while other entities such as `EventEntity`, `EventApplicationEntity`, `CommentEntity`, `ConversationEntity`, and `MessageEntity` support event management, applications, user interaction, and messaging.

In the final implementation, PostgreSQL was the only database used. Although offline support with SQLite had been considered earlier in the project, it was not implemented in the final version due to time constraints.

Description of main system entities:

- **User**
  Represents a system user and serves as the base entity for both volunteers and organizations. It contains shared attributes such as authentication-related data and is extended to support role-specific information (see Appendix G Figure G.2 for implementation of this class).
- **Event**
  Represents a volunteering opportunity created by an organization. Each event contains details such as title, description, date, time, location, and required skills. Events act as the central element of the system, linking volunteers and organizations.

- **EventApplication**
  Represents a volunteer's application to participate in a specific event. This entity captures the relationship between volunteers and events, including application status and attendance tracking (see Appendix G Figure G.4 for implementation of this class).
- **Comment**
  Represents user-generated content associated with an event. Comments allow volunteers and organizations to communicate directly within the context of a specific event, supporting discussion and clarification.
- **Conversation**
  Represents a communication channel between two users in the messaging system. It groups related messages and defines the participants involved in a conversation.
- **Message**
  Represents individual messages exchanged between users within a conversation.

4.3.1. Database Technologies and Mapping

The backend uses Java Persistence API (JPA) annotations together with Hibernate to map Java classes to PostgreSQL tables. Each major entity in the application is represented by a dedicated class annotated with `@Entity`, while relationships are defined using annotations such as `@ManyToOne`, `@OneToMany`, and `@ManyToMany`. Hibernate is then responsible for handling the object-relational mapping between the application and the database, allowing database operations to be performed through Java objects rather than raw SQL queries (see Appendix G).

UUIDs were used as primary keys across the entity model. This can be seen in the entity classes, where identifiers are generated using `GenerationType.UUID`. Using UUIDs ensures uniqueness across records and avoids reliance on sequential numeric identifiers (see Appendix G).

4.3.2. Inheritance for Users, Volunteers, and Organizations

Since authentication in the system is handled using Keycloak, one of the main challenges during database design was modelling user roles in a way that integrates well with Keycloak while also avoiding data duplication and supporting role-specific attributes.

Initially, I implemented this using a one-to-one relationship between `UserEntity` and both `VolunteerEntity` and `OrganizationEntity`. While this approach worked structurally, it introduced unnecessary complexity when querying data. In many cases, it was required to first retrieve the base user using its identifier and then perform additional joins to access role-specific data. This made queries more complicated and reduced overall efficiency.

After further research, I identified JPA inheritance as a more suitable approach and refactored the design accordingly. In the final implementation, `UserEntity` is defined as an abstract base class, and both `VolunteerEntity` and `OrganizationEntity` extend it. The inheritance strategy used is `InheritanceType.JOINED`, where shared fields are stored in the base `users` table, and role-specific attributes are stored in separate tables linked by the same primary key (see Appendix G Figure G.2).

This approach ensures that common user data, such as username, email, Keycloak identifier, role, and timestamps, is not duplicated across multiple tables. At the same time, it allows each role to maintain its own specific attributes. For example, volunteers store personal details such as name, date of birth, interests, and location, while organizations store fields such as organization name, description, and website.

4.3.3. Relationships Between Entities

The implemented schema uses one-to-many, many-to-one, and many-to-many relationships. These are represented explicitly using foreign keys and JPA relationship annotations.

A one-to-many relationship exists between organizations and events. One organization can own many events, while each event belongs to one organization. This reflects the application logic in which organizations create and manage volunteering opportunities.

Another one-to-many structure exists between events and comments. One event can have many comments, while each comment belongs to one event. In addition, each comment belongs to one user, meaning that a user can create many comments, but each comment has only one author.

The messaging system also uses one-to-many relationships. A conversation can contain many messages, while each message belongs to one conversation. Each message also references one sender, which is a user.

A many-to-many relationship is implemented between events and volunteers through the `event_volunteer` join table. In the code, this is mapped using `@ManyToMany` in `EventEntity` and `VolunteerEntity`. This relationship represents volunteers associated with events.

4.3.4. Event Applications

In addition to the direct many-to-many volunteer to event association, the implementation also includes a separate `EventApplicationEntity`. This entity stores a volunteer, an event, an application status, attendance status, attendance metadata, and timestamps. It is therefore not just a simple join table, but a separate entity used to model the application process.

This was an important part of the implementation because applying to an event requires more information than a simple relationship. The system needs to record whether an application is pending, accepted, or canceled, and it also tracks attendance-related information through fields such as `attendanceStatus`, `attendanceMarkedAt`, and `attendanceMarkedBy` (see Appendix G Figure G.4).

A unique constraint is defined on the combination of `event_id` and `volunteer_id`, ensuring that the same volunteer cannot create multiple application records for the same event. This is an example of enforcing a business rule directly at database level rather than relying only on application logic (see Appendix G Figure G.4).

4.3.5. Constraints and Data Integrity

In addition to the application-level validation, I implemented several database level constraints to enforce strict data integrity. First, foreign key relationships ensure referential integrity between records. For example, an event application cannot exist without a valid volunteer and a valid event, and a message cannot exist without both a valid conversation and a valid sender. Similarly, comments must reference both a valid user and a valid event (see Figure 10).

```java
@Table(name = "messages")
public class MessageEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "conversation_id", nullable = false)
    private ConversationEntity conversation;
```

Figure 10. Example of a foreign key relationship enforcing referential integrity in `MessageEntity`.

Second, unique constraints are used to prevent the creation of duplicate or conflicting records. In the base user table, both `username` and `email` are marked as unique, to duplicate identities from being stored in the system. In `EventApplicationEntity`, the combination of event and volunteer is also unique, which prevents duplicate applications for the same event (see Appendix G Figure G.4). By implementing these constraints, I ensured that the database itself is responsible for enforcing the system's core correctness rules, which provides a final layer of protection against data corruption.

4.3.6. Geolocation Support

To support the spatial requirements of the application, I integrated geospatial capabilities using PostgreSQL spatial types and Hibernate Spatial. I configured the database to store location data using the `Point` type with the `geometry(Point, 4326)` column definition. I applied this structure to both the `location` field for events and the `homeLocation` field for volunteers. By representing coordinates as geographic points rather than plain text, the system is able to handle complex spatial queries. I made this architectural choice to improve the efficiency, because implementing it in service layer would make it very inefficient. By using the `ST_DWithin` function at the database level, the system can accurately identify all events within a specified radius of a point. Using PostgreSQL was particularly useful for this project because it provides native, high-performance support for advanced spatial operations, which would be significantly more complex and less efficient to implement entirely within the application code.

4.3.7. Querying and Retrieval

The repository layer shows that the application performs several structured queries beyond simple record retrieval. Examples include searching events by title or organization name, filtering by event categories, retrieving events linked to a specific organization, and finding events near a volunteer's location (see Appendix A Figures A.10 and A.11).

The application also queries event applications by volunteer, by event, and by status. In some cases, related entities are loaded together using 'join fetch', which helps retrieve related event, organization, or volunteer data in a single query.

4.4. Authentication and Security

I implemented authentication and authorization using Keycloak, which serves as the external Identity Provider. In this design the backend is configured as an OAuth2 (OpenAuthorization) resource server. By using this architecture, I avoided storing and managing passwords directly in my application database. Instead, the backend validated access tokens issued by Keycloak. This setup is configured within the `application.properties` and `SecurityConfig.`

I chose this approach early in development because I wanted to use a more secure and realistic model from the start. At first, I considered using basic Spring Security and then moving to Keycloak later. However, after testing Keycloak and reading its documentation, I found out that the integration was simpler than I expected. Because of this, I decided to integrate Keycloak from the beginning, allowing me to avoid extra refactoring later and saved me time in development.

4.4.1. User Identity Mapping

In practice, the authentication process relies on the data contained within JWT. The most important part of the token for my implementation is the `subject` field, which represents the unique Keycloak user ID. I stored this value in the `keycloakId` field of `UserEntity` so that the user stored in Keycloak could be connected to our local application user. I then implemented `CurrentUserService`, which reads the authentication object from the security context, extracts the Keycloak ID from the JWT and queries the database for the matching user record. This ensures that even though the authentication happens externally, the application can still associate the authenticated user with their specific profile and permissions within the local system.

4.4.2. Keycloak Administration Integration

The implementation also includes integration with the Keycloak administration API. This was important because the system does not only validate tokens but also needs to create and manage users in Keycloak. I implemented this functionality in `KeycloakService`. This class is responsible for getting an administrative access token, sending requests to create users, assigning roles, and deleting users when required (see Figure 11). I created and configured a dedicated realm and a corresponding client in Keycloak to support this integration.

A detailed guide on how to set up Keycloak with Spring Boot is available in the official documentation [16].

```
public void deleteUser(String keycloakId) {  2 usages   Fbdrv
    getBackendAccessToken().flatMap( String token -> keycloakWebClient.delete()  RequestHeadersUriSpec<capture of ?>
        .uri( uri: userUrl + "/" + keycloakId)  capture of ?
        .header( headerName: "Authorization",  ...headerValues: "Bearer " + token)
        .header( headerName: "Content-Type",  ...headerValues: "application/json")
        .accept(MediaType.APPLICATION_JSON)
        .retrieve()  ResponseSpec
        .onStatus(HttpStatusCode::isError,  ClientResponse response ->
                response.bodyToMono(String.class)
                        .flatMap( String errorBody -> Mono.error(
                                new KeycloakException(
                                        "An error occurred while deleting user with keycloakId '%s':".formatted(keycloakId)
                                                + errorBody))))
        .bodyToMono(Void.class)).block();
}
```

Figure 11. Example of the method that uses Keycloak API.

To support user creation, `UserService` builds a `UserRepresentation` object before sending it to Keycloak. This object contains the user data expected by Keycloak, such as username, email, and assigned role. This approach allows sensitive user information to be

managed securely within Keycloak, while additional role specific data is stored in the apps database.

4.4.3. User Synchronization Difficulties

One of the main challenges I faced was keeping Keycloak users synchronized with the backend database. During development, I used Hibernate's 'create-drop' mode in `application.properties` because it made schema changes faster to test. However, this also meant that backend user records were deleted whenever the application restarted, while Keycloak users remained unchanged.

This created a bug, where a user could still authenticate successfully through Keycloak and receive a valid JWT, but the backend would fail to find the corresponding internal user record through `keycloakId`. As a result, protected features could not work correctly even though authentication itself had succeeded.

I first attempted to solve this by introducing Flyway for database migrations. However, this created a conflict because both Flyway and Hibernate were trying to manage the database schema. Since the project was still in active development and the schema was changing frequently, and I did not have any sensitive data I wanted to save, I realized that this approach added unnecessary complexity.

To solve the problem in a simpler way, I continued using Hibernate's schema generation and implemented a development-focused seeding solution. I created `DevDataSeeder`, which implements `CommandLineRunner` so that it runs automatically when the application starts (see Figure 12). This class recreates predefined backend user records after a reset, helping keep the local database aligned with Keycloak accounts. Although this is not a complete synchronization solution, it was a practical solution during development and reduced repeated manual setup.

```java
public class DevDataSeeder implements CommandLineRunner {
    private final UserRepository userRepository;

    @Override  Fbdrv *
    public void run(String... args) throws Exception {
        if (!userRepository.existsByUsername("org")) {
            OrganizationEntity orgEntity = OrganizationEntity.builder()
                    .username("org") capture of ?
                    .email("org@mail.com")
                    .role(UserRole.ORGANIZATION)
                    .address("Tashkent")
                    .description("IDK")
                    .websiteUrl("LOL")
                    .keycloakId("fea6550e-8035-4580-bb0e-a461e85540dd")
                    .organizationName("DummyOrganization")
                    .build();

            userRepository.save(orgEntity);
        }
```

Figure 12. Uses `DevDataSeeder` to automatically recreate predefined user records when the application starts.

Overall, this implementation resulted in a secure and structured authentication system, where Keycloak handles identity management, Spring Security validates JWT's, and the backend enforces role-based access control. During development, I realized that integrating an external authentication provider can introduce unexpected challenges, like keeping Keycloak users and backend records in sync when the database is reset.

5. Testing

I implemented an automated test suite focused on unit tests for the service layer. In practice, this means each test checks a service method while mocking external dependencies such as repositories, mappers, and helper services. This keeps the tests fast, consistent, and easy to run often. This follows common good practice, where most tests are written at a lower level rather than relying heavily on UI tests [34].

In addition to automated testing, I also performed manual testing of the application to ensure correct behavior across platforms. Since the application is cross-platform, I ran both iOS and Android emulators during development. This allowed me to verify that the user interface and functionality behave consistently across different operating systems (see Appendix A Figure A.2, Figure A.3).

For backend testing, I used Postman to manually test API endpoints. I created a dedicated Postman collection containing backend endpoints, which made it easier to test different scenarios such as authentication, event creation, and application workflows. This helped ensure that the API behaves correctly independently of the frontend (see Figure 13) [25].

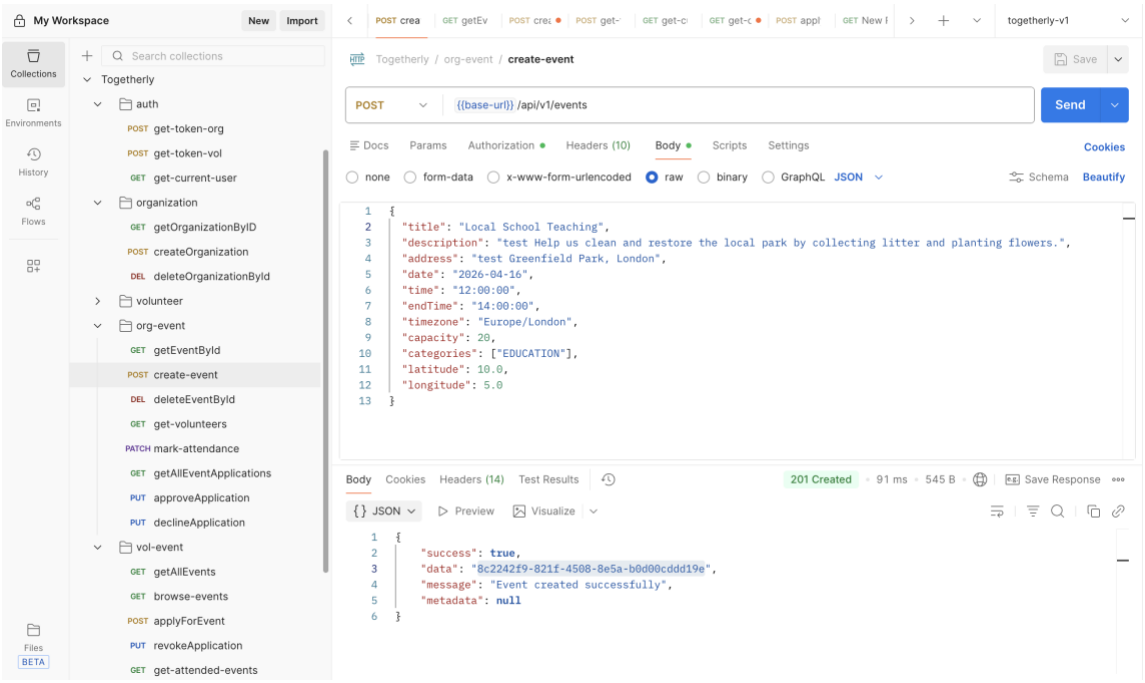


Figure 13. Postman collection used for testing backend API endpoints.

5.1. Why service layer testing

I focused on testing the service layer because this is where the main business logic is implemented. Services define how the system behaves, including rules, validations, and interactions between components. Because of this, testing services gives the most useful confidence that the system works correctly.

Service layer tests are also easier to isolate using mocks, which means they run quickly and give fast feedback. Compared to controller or UI tests, they do not require setting up the full web framework or dealing with request/response conversion. This allows me to focus directly on business logic such as permissions, state changes, and validation rules. This approach also follows the idea of the test pyramid, where lower-level tests are faster, cheaper, and easier to maintain than UI tests [34].

5.2. Implementation using JUnit and Mockito

I organized all tests under the `src/test/` directory using a consistent naming style such as `ClassNameTest` and `method_expectedResult`. This makes it easier to understand what each test does and quickly identify where a failure comes from. I wrote the tests using JUnit 5 (Jupiter) and Mockito. JUnit provides the structure for writing and running tests, while Mockito allows me to mock dependencies [1, 32].

Mocks are initialized using `@ExtendWith(MockitoExtension.class)`, and I used annotations such as `@Mock` and `@InjectMocks`. Across the tests, I followed the Arrange, Act, Assert (AAA) pattern to keep them clear and readable, where **Arrange** stands for prepare input data and define mock behavior, **Act** for call the method being tested **Assert** for verify the result and behavior [14].

In addition to checking return values, I also verified interactions using `verify()` (and sometimes `times(n)`) to make sure the service calls the correct repository or helper methods. I also used matchers such as `any()` and `eq()` where needed.

For example, I tested `EventService.applyToEvent()` by mocking the current user and repository calls. The test checks that a new application is saved only when all conditions are valid (see Appendix K Figure K.2).

5.3. UI testing

I could also implement UI tests using tools such as Selenide to automate full user flows. However, I decided not to include this in the project. UI tests are generally more complex to build and maintain, as they depend on UI structure, selectors, and timing. Given the limited timeframe of this project, implementing UI tests was considered out of scope.

Instead, I focused on developing the application itself and validating the core logic through unit tests. For the frontend and overall system behavior, I relied on manual testing to ensure that features work correctly when integrated.

6. Tools

6.1. Engineering Tools

During development, I used GitHub as the main version control system [13]. It allowed me to track changes, manage different features using branches, and safely experiment without affecting the main codebase. This was particularly useful when working on different parts of the system in parallel. GitHub also helped maintain a clear history of the project and made it easier to revert changes when needed.

Docker was used to simplify the setup of external services, specifically PostgreSQL and Keycloak [18]. Instead of installing and configuring these tools manually, I used Docker containers to run them in isolated environments. This made the development setup more consistent and easier to manage, as both services could be started quickly with predefined configurations.

Postman was used as an API testing tool during development. It allowed me to send HTTP requests to backend endpoints and inspect responses without relying on the frontend. This was useful for verifying endpoint behavior, testing different scenarios, and debugging issues during development (see Figure 12).

6.2. AI Usage

At the beginning of this project, the use of artificial intelligence (AI) tools was not part of my original plan. My initial goal was to build a strong foundation in Flutter by developing the project independently. However, by the time I completed the MVP, AI tools had advanced significantly, and I realized that they would likely be an important tool in my future career.

Therefore, I decided to start using tools such as Codex and Claude Code, mainly to support frontend development [4, 23]. These tools were used to assist with tasks such as UI structure and repetitive widget patterns, rather than generating full solutions. I followed a structured workflow, where I first defined the feature, its design, and the backend endpoints it interacts with, then used AI to generate code, and finally reviewed and adjusted it to ensure it matched the project architecture and best practices.

AI was also used to extend backend test coverage. After writing the initial tests myself, I used AI to generate additional tests for uncovered methods, ensuring that they followed existing patterns such as Mockito usage, and the AAA pattern. All generated code was manually reviewed to ensure correctness.

Overall, AI tools acted as a support mechanism that improved development efficiency and speed, while I remained responsible for all key design and architectural decisions.

7. Reflection

One of the main challenges I faced at the beginning of the project was a lack of experience in developing a full system independently. The overall scope of the project felt overwhelming, and I initially struggled to decide where to start. Although I had a general idea of how the system should work, I did not spend enough time developing a complete database design beforehand. Instead, I began implementing features early and expanded the system as I progressed. This led to multiple changes in the data model, which increased development time. Although I saved time on security by integrating Keycloak early, the overall development still took longer than expected due to these modifications. Looking back, I should have spent more time planning the system architecture and database schema before starting implementation. However, it is common for designs to change during development as requirements become clearer. I believe the most effective approach is to create an initial design while allowing room for adjustments.

I also overcomplicated certain aspects of the system. One example was attempting to follow a Kanban-based workflow, which added unnecessary complexity for a solo project. Another example was attempting to introduce Flyway for database migrations during development. Since the schema was still changing frequently and there was no need to preserve production data, this ended up being unnecessary and resulted in additional time spent debugging conflicts between Flyway and Hibernate. As a result, Flyway was removed from the final implementation. This issue occurred mainly due to insufficient research before introducing new tools into the project.

Time management was another significant challenge. My workflow was inconsistent, with periods of high productivity followed by little or no progress. This made it difficult to maintain steady progress in development. A more consistent approach, with smaller and regular development sessions, would have improved both productivity and overall progress.

Due to the need to balance this project with other university work, several planned features were not implemented or had to be simplified. For example, push notifications were not included in the final version, despite being an important feature for mobile applications. Other features, such as attendance tracking and location-based functionality, were implemented in a simplified form. Additional features, including a custom analytics dashboard, volunteer ranking system, and automation tools, were also not implemented. These decisions were necessary to ensure that the core functionality of the system remained stable and complete, which was my main focus during development.

If more time were available, I would focus on improving both functionality and user experience. I would especially improve the user interface and implement features that were cut off, such as dark mode, multilingual support, and group messaging. These improvements would make the system more practical for real-world use.

The use of AI tools also influenced the development process. Initially, I did not plan to use them, but I started using them later in the project to support frontend development and testing. These tools helped improve efficiency and saved time. However, I limited their use to ensure that I developed my own understanding of the technologies. Introducing them earlier could have helped me complete more features, but it might have reduced the learning value of the project. For this reason, they were used mainly towards the end and in a controlled manner.

Overall, the development process highlighted the importance of proper planning, realistic scope definition, and consistent work habits. Given the limited timeframe and the fact that the project was completed individually, the final result represents a reasonable balance between functionality and complexity.

8. Summary

Charities play crucial role in society and rely heavily on volunteers to achieve their goals. However, many organizations still depend on fragmented tools such as messaging applications to promote opportunities and hire volunteers. While existing platforms attempt to address this issue, they often focus only one part of the problem and do not fully provide event discovery, management and communication within one app.

Togetherly is a cross-platform mobile application designed to address this problem. The project was developed as a full-stack system, combining a Flutter frontend, Spring Boot backend, and PostgreSQL database.

The application supports two main user roles, volunteers and organizations, each with clearly defined responsibilities. This structure enables both event discovery and management within a single platform, addressing the gap identified in existing solutions.

Overall, the project achieved its main aims and objectives. It demonstrates the ability to design and implement a complete software system using best development practices. While some advanced features were not implemented due to time constraints, the final system provides a stable and functional solution. These features could be added in future work to further extend the system and improve its real-world use.

9. Bibliography

[1] "Tasty mocking framework for unit tests in Java." https://site.mockito.org/ (accessed 30 March).

[2] "Hibernate ORM." https://hibernate.org/orm/ (accessed 7th April).

[3] A. Anthony. "Efficient State Management in Flutter Using IndexedStack." https://www.freecodecamp.org/news/efficient-state-management-in-flutter-using-indexedstack/ (accessed 14 April).

[4] Anthropic. "Claude Code Docs." https://code.claude.com/docs/en/overview (accessed 21 April).

[5] M. authors. "MapStruct." https://mapstruct.org/ (accessed 7th April).

[6] F. Badriddinov, "Togetherly : Project Specification and Planning Document," 27th October 2025.

[7] Baeldung. "The DTO Pattern (Data Transfer Object)." https://www.baeldung.com/java-dto-pattern (accessed 7th April).

[8] Capterra. "VolunteerMatch Reviews 2025: Verified Reviews, Pros & Cons." Capterra. https://www.capterra.com/p/168325/YourMatch/reviews/ (accessed 21 October 2025).

[9] W.-C. Chen, Y.-M. Cheng, F. E. Sandnes, and C.-L. Lee, "Finding suitable candidates: the design of a mobile volunteering matching system," in *International Conference on Human-Computer Interaction*, 2011: Springer, pp. 21–29.

[10] D. S. Foundation. "Django Documentation – The Web Framework for Perfectionists with Deadlines." https://docs.djangoproject.com/en/stable/ (accessed 18 October 2025).

[11] O. Foundation. "Express.js – Fast, Unopinionated, Minimalist Web Framework for Node.js." https://expressjs.com/ (accessed 18 October 2025).

[12] geeksforgeeks. "Flutter - File Structure." https://www.geeksforgeeks.org/flutter/flutter-file-structure/ (accessed 2026).

[13] GitHub. "About Version Control – GitHub Docs." https://docs.github.com/en/get-started/start-your-journey/about-github-and-git (accessed 14 October 2025).

[14] P. Gomes. "nit Testing and the Arrange, Act and Assert (AAA) Pattern." https://medium.com/@pjbgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80 (accessed 17 April).

[15] P. G. D. Group. "PostgreSQL 16 Documentation." https://www.postgresql.org/docs/ (accessed 18 October 2025).

[16] R. Hat. "Keycloak Documentation – Identity and Access Management for Modern Applications." https://www.keycloak.org/documentation (accessed 18 October 2025).

[17] Idealist. "VolunteerMatch." https://www.idealist.org/volunteermatch (accessed 20 October 2025).

[18] D. Inc. "Docker Overview Documentation." https://docs.docker.com/get-started/overview/ (accessed 18 October 2025).

[19] M. P. Kaur, J. A. Pazour, and R. Ausseil, "An optimization framework to provide volunteers with task selection autonomy and group opportunities," *Socio-Economic Planning Sciences,* vol. 96, p. 102095, 2024.

[20] L. LLC. "Laravel Documentation – The PHP Framework for Web Artisans." https://laravel.com/docs (accessed 18 October 2025).
[21] S. Nema. "Real-Time Chat application using Spring Web Sockets and StompJs." https://medium.com/@satviknema/real-time-chat-application-using-spring-web-sockets-and-stompjs-0d286696e2b0 (accessed 7 April).
[22] NPTechNews. "Review: Golden – Automation Tools and Volunteer Engagement." https://www.nptechnews.com/index.php/reviews/item/4423-review-golden (accessed 20 October 2025).
[23] OpenAI. "Codex." https://openai.com/codex/ (accessed 21 April).
[24] A. Petru. "LinkedIn, VolunteerMatch Team Up to Connect Nonprofits With Volunteers." TriplePundit. https://www.triplepundit.com/2014/linkedin-volunteermatch-team-connect-nonprofits-volunteers/ (accessed 21 October 2025).
[25] I. Postman. "Postman." https://www.postman.com/ (accessed 19 April).
[26] M. R. Randle, Samantha, "Recruitment and Retention of Volunteers: A Rapid Literature Review," NSW Government, New South Wales, Australia, 2023. Accessed: 16 April 2026. [Online]. Available: https://www.nsw.gov.au/sites/default/files/2023-08/recruitment_and_retention_of_volunteers__a_literature_review.pdf
[27] Riskiana. "Where Should You Put Your Validations in a Spring Boot Microservice?" https://medium.com/@kiana.proudmoore/where-should-you-put-your-validations-in-a-spring-boot-microservice-b8dc1219c0b7 (accessed 7th April).
[28] Shanakaprince. "What is Lombok and Why Should We Use It?" https://medium.com/@shanakaprince/what-is-lombok-and-why-should-we-use-it-4b1357248d62 (accessed.
[29] Sitejabber. "VolunteerMatch Reviews." https://www.sitejabber.com/reviews/volunteermatch.com (accessed 21 October 2025).
[30] Spring. "Spring Boot." https://spring.io/projects/spring-boot (accessed 17 October 2025).
[31] A. Stukas and S. Wilson, "Understanding motivations to volunteer," *Volunteering Australia,* 2022.
[32] T. J. Team. "The programmer-friendly testing framework for Java and the JVM." https://junit.org/ (accessed 30 March).
[33] Trustpilot. "Volunteero Reviews 2025: Read Customer Feedback and Ratings." https://www.trustpilot.com/review/volunteero.org (accessed 18 October 2025).
[34] H. Vocke. "The Practical Test Pyramid." https://martinfowler.com/articles/practical-test-pyramid.html (accessed 30 March).
[35] Volunteero. "Racing Welfare Case Study: Streamlining Volunteer Onboarding and Management." Volunteero. https://volunteero.org/case-studies/racing-welfare (accessed 18 October 2025).
[36] R. Wu. "From Resource Hierarchy to RESTful API." https://medium.com/random-life-journal/from-resource-hierarchy-to-restful-api-dabc1866821e (accessed 7th April).

10. Appendices

10.1. Appendix A - UI screens of the app

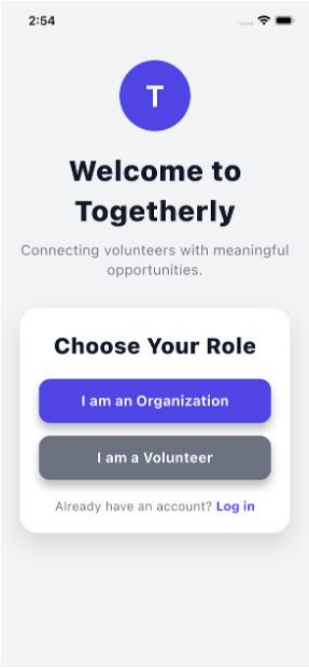


Figure A.1. Login page where users can choose their role.

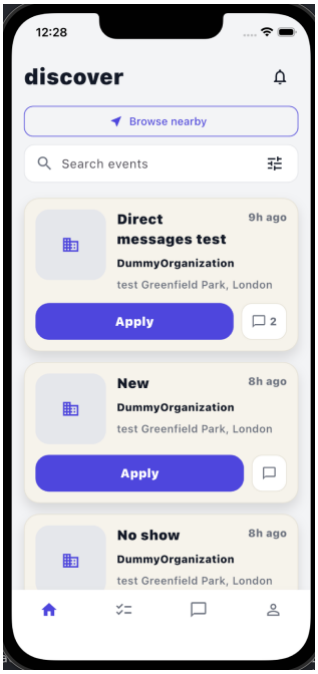


Figure A.2. Volunteers main page (IOS).

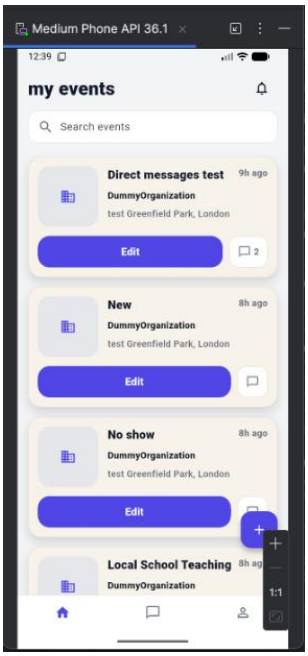


Figure A.3.Organizations main page (Android).

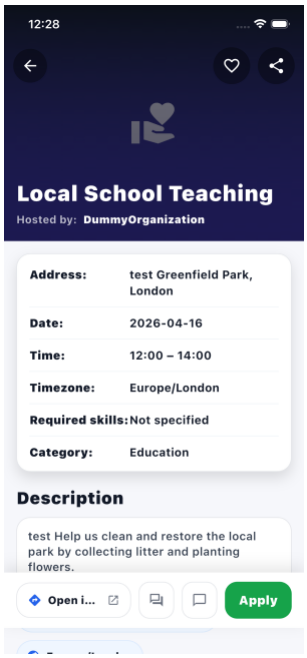


Figure A.4. Event info page for volunteer role.

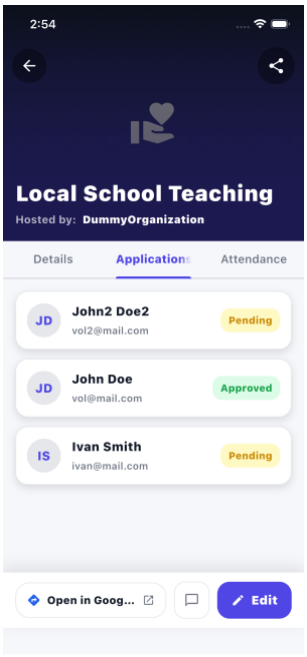


Figure A.5. Applications page for organizations.

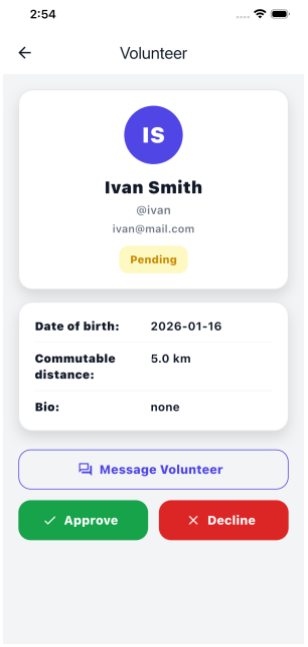


Figure A.6. Organization application management page.

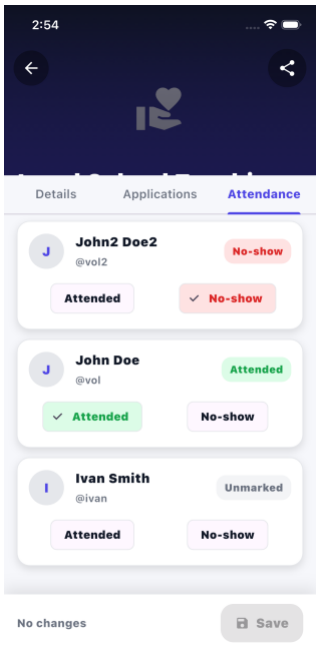


Figure A.7. Organizations attendance management page.

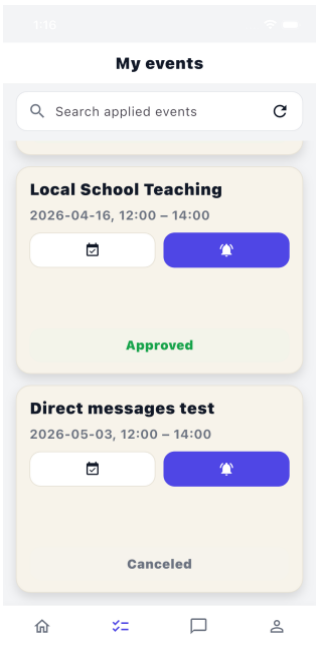


Figure A.8. Volunteers applied events page.

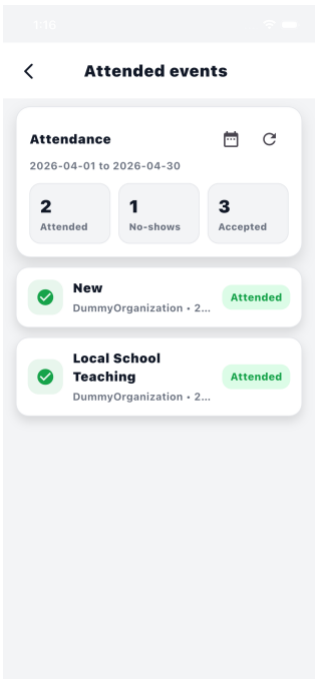


Figure A.9. Volunteers attended events metrics page.

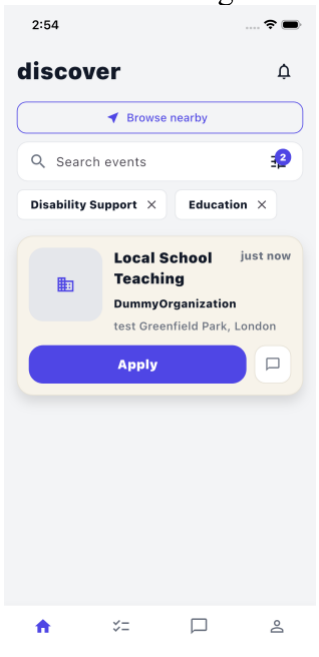


Figure A.10. Category filtering page.

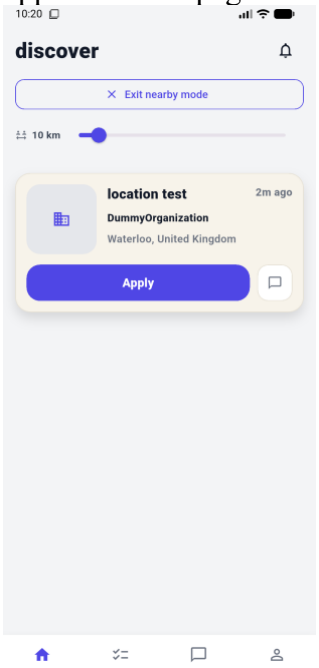


Figure A.11. Location filtering page.

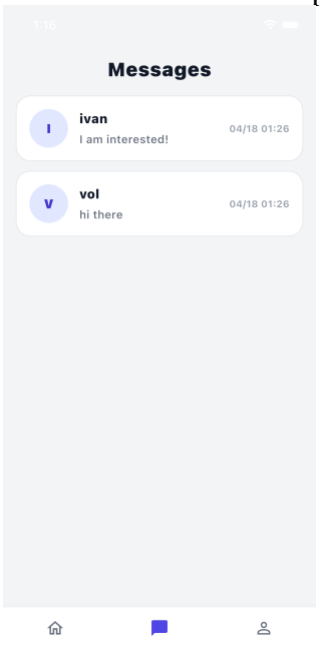


Figure A.12. Conversation page.

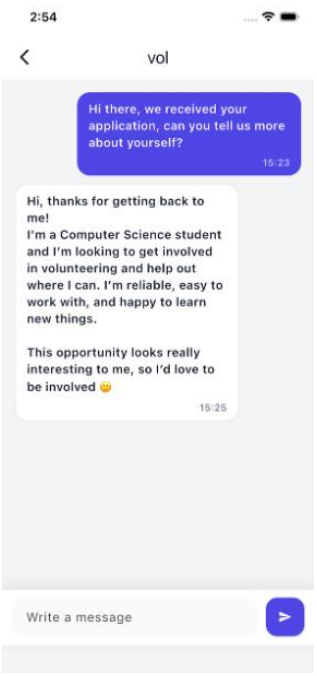


Figure A.13. Direct messages page.

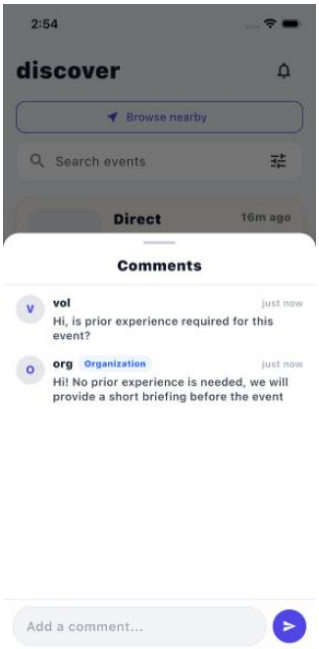


Figure A.14. Commenting system page.

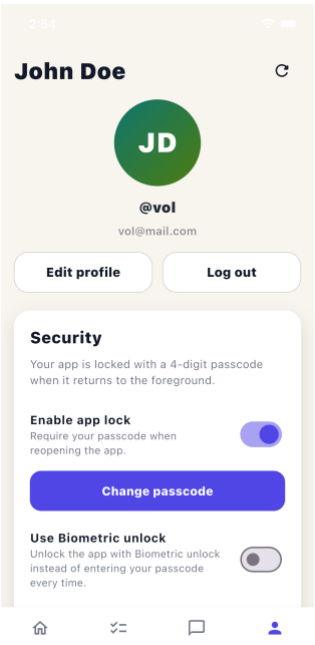


Figure A.15. Password and biometric lock page.

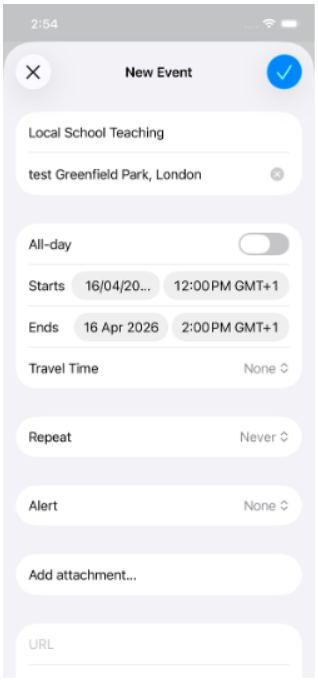


Figure A.16. Calendar integration page.

Validation

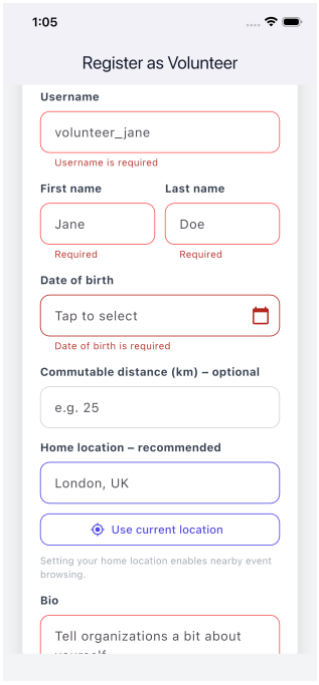


Figure A.17. Volunteer registration page.

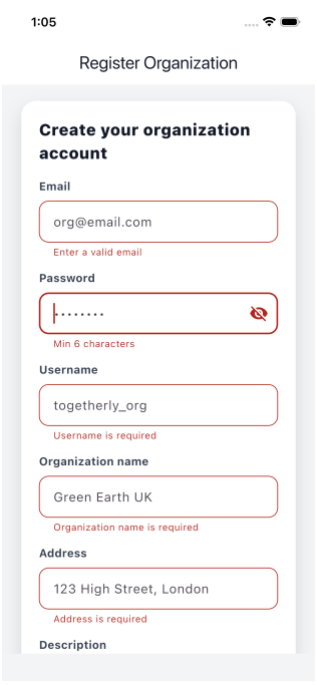


Figure A.18. Organization registration page.

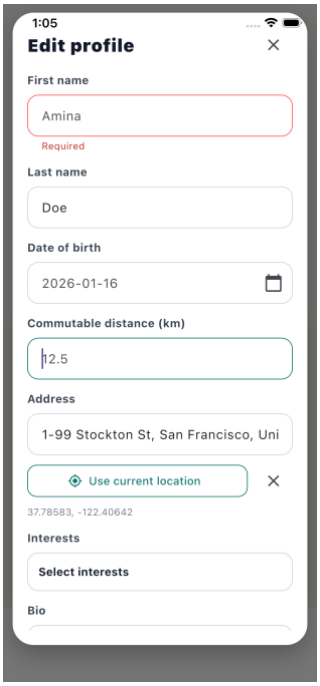


Figure A.19. Profile edit.

User Feedback

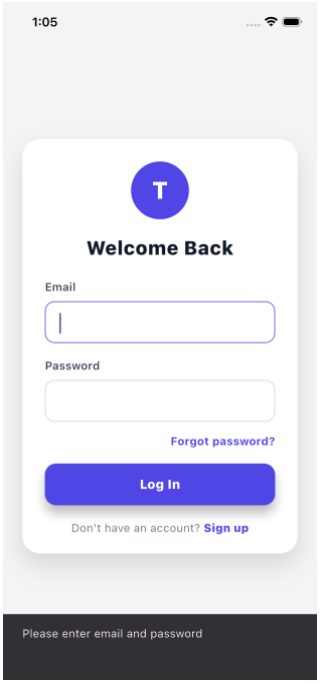


Figure A.20. SnackBar feedback for validation.

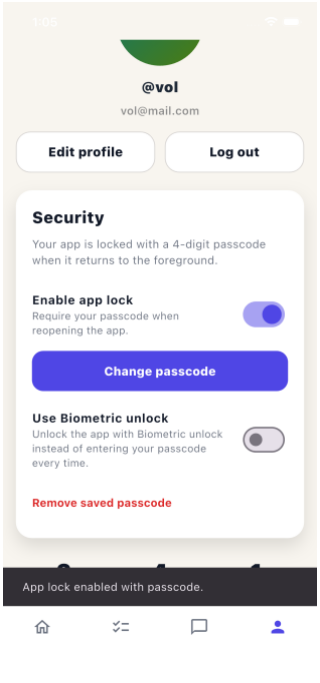


Figure A.21. SnackBar feedback on success.

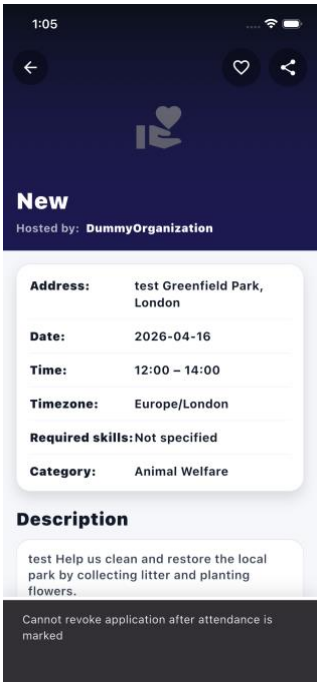
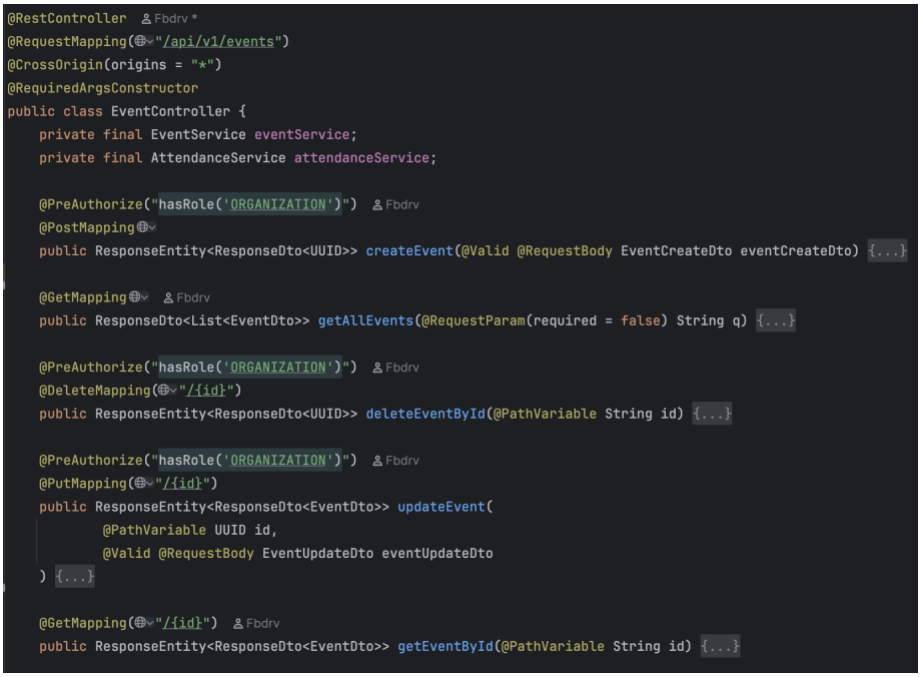


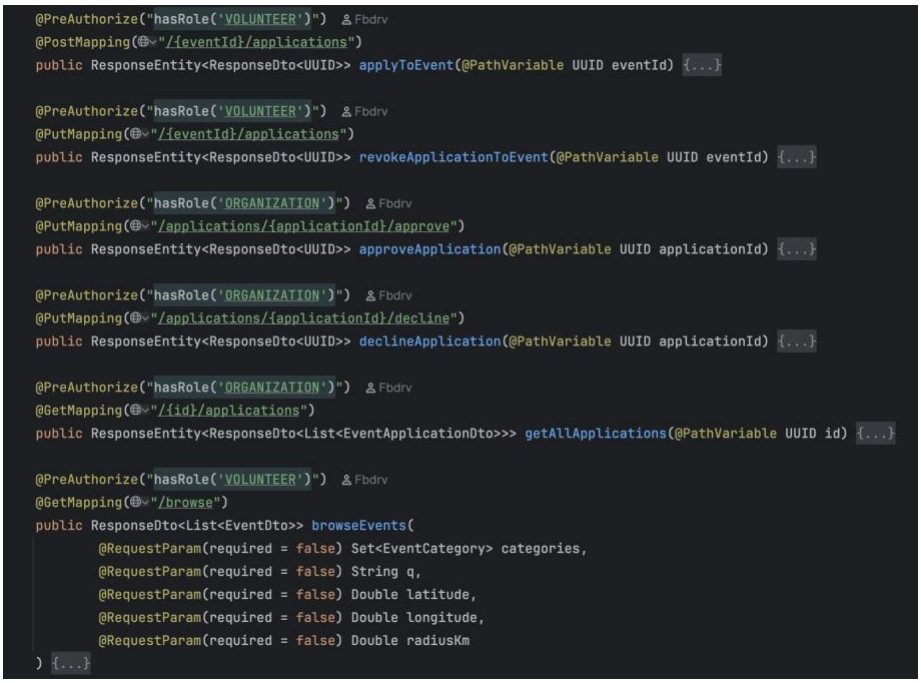
Figure A.22. SnackBar feedback on the wrong state.

10.2. Appendix B - Example of REST API Endpoints in EventController



```java
@RestController  Fbdrv *
@RequestMapping("/api/v1/events")
@CrossOrigin(origins = "*")
@RequiredArgsConstructor
public class EventController {
    private final EventService eventService;
    private final AttendanceService attendanceService;

    @PreAuthorize("hasRole('ORGANIZATION')")  Fbdrv
    @PostMapping
    public ResponseEntity<ResponseDto<UUID>> createEvent(@Valid @RequestBody EventCreateDto eventCreateDto) {...}

    @GetMapping  Fbdrv
    public ResponseDto<List<EventDto>> getAllEvents(@RequestParam(required = false) String q) {...}

    @PreAuthorize("hasRole('ORGANIZATION')")  Fbdrv
    @DeleteMapping("/{id}")
    public ResponseEntity<ResponseDto<UUID>> deleteEventById(@PathVariable String id) {...}

    @PreAuthorize("hasRole('ORGANIZATION')")  Fbdrv
    @PutMapping("/{id}")
    public ResponseEntity<ResponseDto<EventDto>> updateEvent(
            @PathVariable UUID id,
            @Valid @RequestBody EventUpdateDto eventUpdateDto
    ) {...}

    @GetMapping("/{id}")  Fbdrv
    public ResponseEntity<ResponseDto<EventDto>> getEventById(@PathVariable String id) {...}
```

Figure B.1. This figure shows REST endpoints in the EventController, including POST, GET, PUT, and DELETE operations. The endpoints are structured under /api/v1/events.



```java
    @PreAuthorize("hasRole('VOLUNTEER')")  Fbdrv
    @PostMapping("/{eventId}/applications")
    public ResponseEntity<ResponseDto<UUID>> applyToEvent(@PathVariable UUID eventId) {...}

    @PreAuthorize("hasRole('VOLUNTEER')")  Fbdrv
    @PutMapping("/{eventId}/applications")
    public ResponseEntity<ResponseDto<UUID>> revokeApplicationToEvent(@PathVariable UUID eventId) {...}

    @PreAuthorize("hasRole('ORGANIZATION')")  Fbdrv
    @PutMapping("/applications/{applicationId}/approve")
    public ResponseEntity<ResponseDto<UUID>> approveApplication(@PathVariable UUID applicationId) {...}

    @PreAuthorize("hasRole('ORGANIZATION')")  Fbdrv
    @PutMapping("/applications/{applicationId}/decline")
    public ResponseEntity<ResponseDto<UUID>> declineApplication(@PathVariable UUID applicationId) {...}

    @PreAuthorize("hasRole('ORGANIZATION')")  Fbdrv
    @GetMapping("/{id}/applications")
    public ResponseEntity<ResponseDto<List<EventApplicationDto>>> getAllApplications(@PathVariable UUID id) {...}

    @PreAuthorize("hasRole('VOLUNTEER')")  Fbdrv
    @GetMapping("/browse")
    public ResponseDto<List<EventDto>> browseEvents(
            @RequestParam(required = false) Set<EventCategory> categories,
            @RequestParam(required = false) String q,
            @RequestParam(required = false) Double latitude,
            @RequestParam(required = false) Double longitude,
            @RequestParam(required = false) Double radiusKm
    ) {...}
```

Figure B.2. This figure shows endpoints for event applications and browsing, including applying, approving, and retrieving applications. It also includes a filtering endpoint (/browse) with query parameters.

10.3. Appendix C - DTO Validation Annotations

```java
@Getter 16 usages  Fbdrv
public class EventCreateDto {

    @NotBlank
    private String title;

    @NotBlank
    private String description;

    @NotBlank
    private String address;

    @NotNull
    private LocalDate date;

    @NotNull
    private LocalTime time;

    @NotNull
    private LocalTime endTime;

    @NotBlank
    private String timezone;

    @Min(1)
    private int capacity;

    @NotEmpty
    private Set<EventCategory> categories;
```

```java
@Getter 12 usages  Fbdrv
public class VolunteerCreateDto {
    @Email
    @NotBlank
    private String email;
    @NotBlank
    private String password;
    @NotBlank
    private String username;
    @NotBlank
    private String firstName;
    @NotBlank
    private String lastName;
    @NotNull
    private LocalDate dateOfBirth;
    @NotNull
    private Double commutableDistance;
    @NotBlank
    private String bio;

    private Set<EventCategory> interests;

    private String address;

    private Double latitude;

    private Double longitude;
}
```

```java
@Getter 9 usages  Fbdrv
public class CommentCreationDto {

  @NotBlank(message = "Comment body cannot be empty")
  @Size(max = 1000, message = "The text is too big.")
  private String body;
}
```

```java
@Getter 11 usages  Farzod
public class OrganizationCreationDto {

    @Email
    @NotBlank
    private String email;

    @NotBlank
    private String password;

    @NotBlank
    private String username;

    @NotBlank
    private String organizationName;

    @NotBlank
    private String address;

    @NotBlank
    private String description;

    @NotBlank
    private String websiteUrl;
}
```

Figure C.1. Validation annotations in request DTOs.

10.4. Appendix D - Examples of API Responses Using ResponseDto<T>

These examples demonstrate the consistent structure of the ResponseDto<T> wrapper across different endpoints.

```
{
    "success": false,
    "data": null,
    "message": "User not found",
    "metadata": null
}
```

Figure D.1. Example error response.

```
{
    "success": true,
    "data": "1b7ac523-6343-42f0-956e-034f6fa63628",
    "message": "Organization created successfully",
    "metadata": null
}
```

Figure D.2. Example successful account creation response.

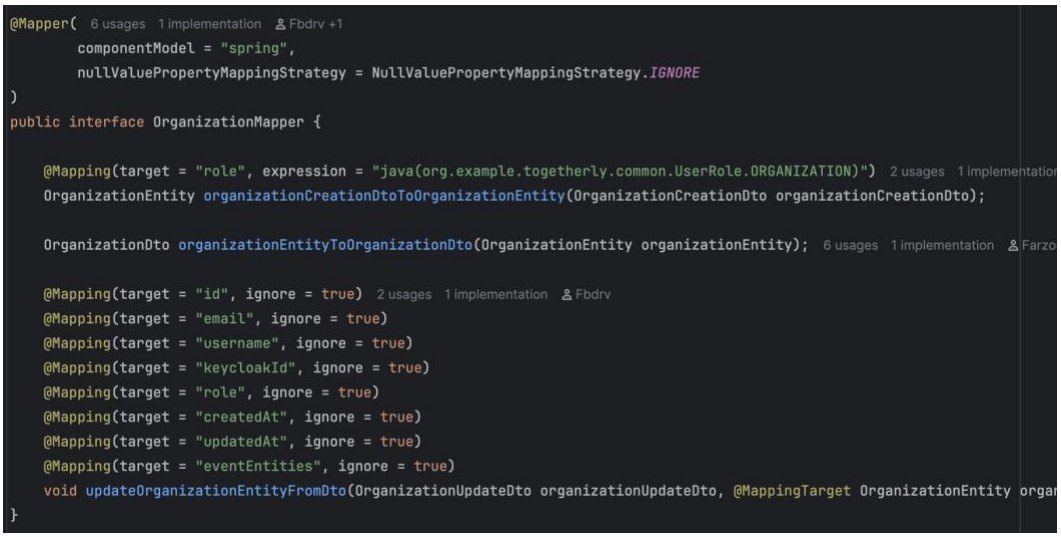
```
{
    "success": true,
    "data": {
        "id": "61594f8d-e013-481c-834a-7e7093caa896",
        "title": "ANIMAL_WELFARE",
        "description": "test Help us clean and restore the local
        "address": "test Greenfield Park, London",
        "organization": {
            "id": "1b7ac523-6343-42f0-956e-034f6fa63628",
            "organizationName": "DummyOrganization1"
        },
        "date": "2026-03-03",
        "time": "12:00:00",
        "endTime": "14:00:00",
        "timezone": "Europe/London",
        "createdAt": "2026-04-07T16:52:58",
        "capacity": 20,
        "categories": [
            "ANIMAL_WELFARE"
        ],
        "volunteersCount": 0,
        "commentsCount": 0,
        "latitude": 10.0,
        "longitude": 5.0
    },
    "message": "Event successfully retrieved",
    "metadata": null
}
```

Figure D.3. Example response for event retrieval.

10.5. Appendix E - Entity–DTO Mapping Using MapStruct

```
@Mapper(componentModel = "spring")  4 usages  1 implementation  Fbdrv
public interface CommentMapper {

    @Mapping(target = "author", source = "user")  1 usage  1 implementat
    CommentDto entityToDto(CommentEntity entity);

    CommentAuthorDto userToAuthorDto(UserEntity user);  1 usage  1 impl

    List<CommentDto> entitiesToDtos(List<CommentEntity> entities);
}
```

Figure E.1. Maps CommentEntity objects to CommentDto, including conversion of the associated user into an author representation.



```
@Mapper(  6 usages  1 implementation  Fbdrv +1
        componentModel = "spring",
        nullValuePropertyMappingStrategy = NullValuePropertyMappingStrategy.IGNORE
)
public interface OrganizationMapper {

    @Mapping(target = "role", expression = "java(org.example.togetherly.common.UserRole.ORGANIZATION)")  2 usages  1 implementation
    OrganizationEntity organizationCreationDtoToOrganizationEntity(OrganizationCreationDto organizationCreationDto);

    OrganizationDto organizationEntityToOrganizationDto(OrganizationEntity organizationEntity);  6 usages  1 implementation  Farzo

    @Mapping(target = "id", ignore = true)  2 usages  1 implementation  Fbdrv
    @Mapping(target = "email", ignore = true)
    @Mapping(target = "username", ignore = true)
    @Mapping(target = "keycloakId", ignore = true)
    @Mapping(target = "role", ignore = true)
    @Mapping(target = "createdAt", ignore = true)
    @Mapping(target = "updatedAt", ignore = true)
    @Mapping(target = "eventEntities", ignore = true)
    void updateOrganizationEntityFromDto(OrganizationUpdateDto organizationUpdateDto, @MappingTarget OrganizationEntity orga
}
```

Figure E.2. Handles conversion between organization DTOs and OrganizationEntity, including creation, retrieval, and partial updates while preserving existing data.

Figure E.3. Maps event DTOs and entities, including conversion between geographic coordinates and location objects for storing and retrieving spatial data.

10.6. Appendix F - Centralized Exception Handling

```java
@RestControllerAdvice  Fbdrv *
public class GlobalExceptionHandler {

    @ExceptionHandler(jakarta.persistence.EntityNotFoundException.class)  Fbdrv
    public ResponseEntity<ResponseDto<Void>> handleNotFound(jakarta.persistence.EntityNotFoundException ex) {
        return ResponseEntity.status(404).body(
                ResponseDto.<~>builder()
                        .success(false)
                        .message(ex.getMessage())
                        .build()
        );
    }

    @ExceptionHandler(AccessDeniedException.class)  Fbdrv
    public ResponseEntity<ResponseDto<Void>> handleAccessDenied(AccessDeniedException ex) {
        return ResponseEntity.status(403).body(
                ResponseDto.<~>builder()
                        .success(false)
                        .message(ex.getMessage())
                        .build()
        );
    }


    @ExceptionHandler(EntityExistsException.class)  Fbdrv *
    public ResponseEntity<ResponseDto<Void>> handleEntityExists(EntityExistsException ex) {
        return ResponseEntity.status(409).body(
                ResponseDto.<~>builder()
                        .success(false)
                        .message(ex.getMessage())
                        .build()
        );
    }
```

Figure F.1. Example of centralized exception handling in GlobalExceptionHandler.

10.7. Appendix G - Entity Class Implementations

```java
@SuperBuilder
@Getter
@Setter
@Entity
@Table(name = "volunteers")
public class VolunteerEntity extends UserEntity {
    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "date_of_birth")
    private LocalDate dateOfBirth;

    @Column(name = "commutable_distance")
    private Double commutableDistance;

    @Column
    private String bio;

    @ElementCollection(fetch = FetchType.LAZY)
    @CollectionTable(
            name = "volunteer_interests",
            joinColumns = @JoinColumn(name = "volunteer_id")
    )
    @Column(name = "interest", nullable = false)
    @Enumerated(EnumType.STRING)
    private Set<EventCategory> interests;

    @ManyToMany(mappedBy = "volunteers", fetch = FetchType.LAZY)
    private List<EventEntity> events;

    @Column(name = "home_location", columnDefinition = "geometry(Point, 4326)")
    private Point homeLocation;
```

Figure G.1. Extends UserEntity to include volunteer-specific data such as personal details, interests, and location.

```java
@NoArgsConstructor  Farzod +1
@AllArgsConstructor
@SuperBuilder
@Getter
@Setter
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
@Table(name = "users")
public abstract class UserEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String keycloakId;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING)
    private UserRole  role;

    @Column(name = "created_at", updatable = false)
    @CreationTimestamp
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    @UpdateTimestamp
    private LocalDateTime updatedAt;
}
```

Figure G.2. Defines the base user model with shared fields such as username, email, role, and Keycloak identifier.

```java
@NoArgsConstructor  Farzod
@AllArgsConstructor
@SuperBuilder
@Getter
@Setter
@Entity
@Table(name = "organizations")
public class OrganizationEntity extends UserEntity {
    @Column(name = "organization_name")
    private String organizationName;

    @Column
    private String address;

    @Column
    private String description;

    @Column(name = "website_url")
    private String websiteUrl;

    @OneToMany(mappedBy = "organization", cascade = CascadeType.ALL, orphanRemoval = true)
    private List<EventEntity> eventEntities;
}
```

Figure G.3. Extends UserEntity to include organization-specific fields such as organization name, description, and related events.

```java
@Table(name = "event_applications",
        uniqueConstraints = {
                @UniqueConstraint(
                        name = "event_applications_volunteer",
                        columnNames = {"event_id", "volunteer_id"}
                )
        })

public class EventApplicationEntity {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    @Column(nullable = false)
    @Builder.Default
    @Enumerated(EnumType.STRING)
    private ApplicationStatus status = ApplicationStatus.PENDING;

    @Column(name = "attendance_status", nullable = false)
    @Builder.Default
    @Enumerated(EnumType.STRING)
    private AttendanceStatus attendanceStatus = AttendanceStatus.NOT_MARKED;

    @Column(name = "attendance_marked_at")
    private LocalDateTime attendanceMarkedAt;

    @Column(name = "attendance_marked_by")
    private UUID attendanceMarkedBy;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "volunteer_id", nullable = false)
    private VolunteerEntity volunteer;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "event_id", nullable = false)
    private EventEntity event;
```

Figure G.4. Defines the relationship between volunteers and events, including application status and attendance tracking.

10.8. Appendix H - Database Schema Design

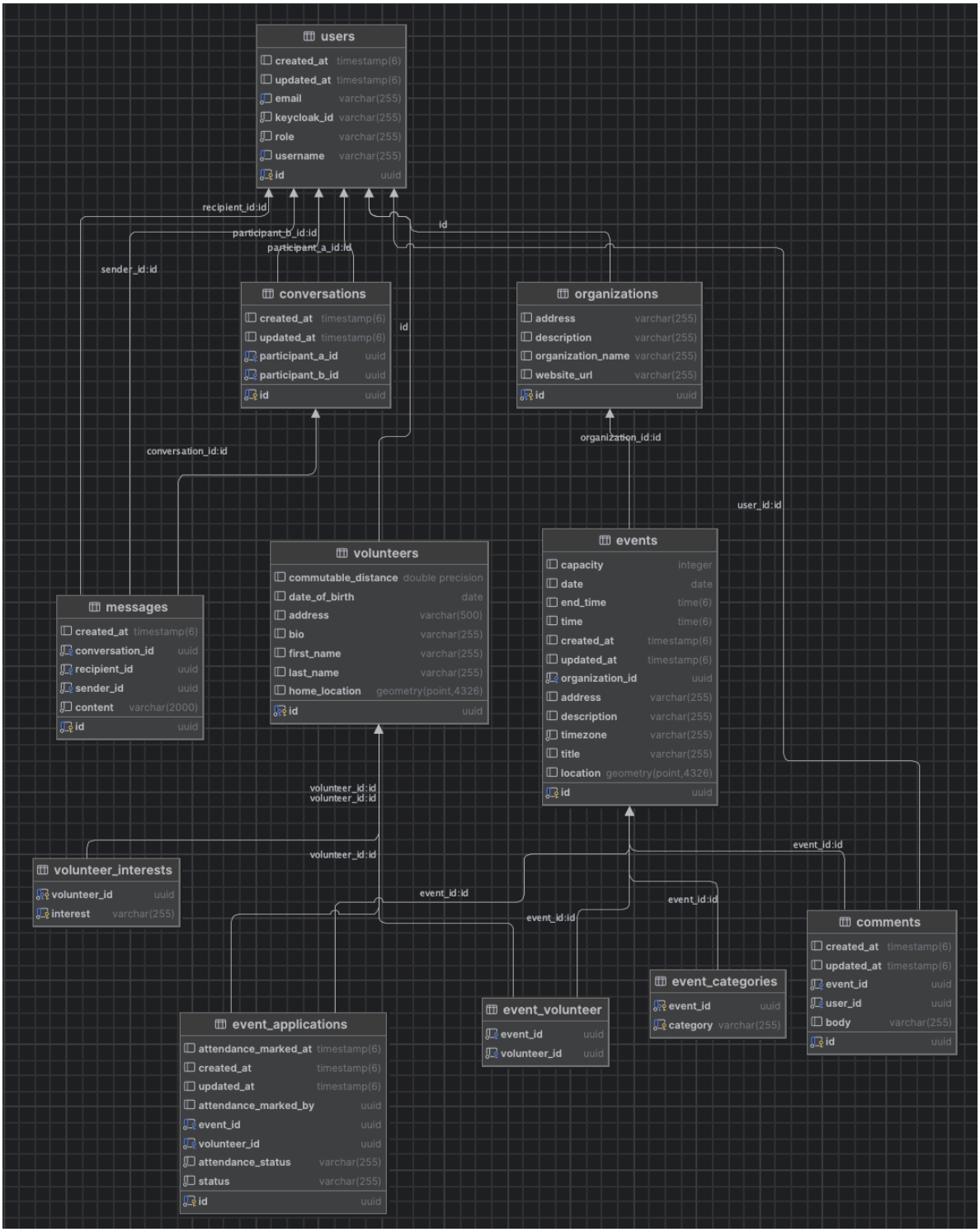


Figure H.1. Database schema showing core entities and relationships, including users, volunteers, organizations, events, applications, messaging, and interactions.

10.9. Appendix I - JWT Role Extraction and Parsing

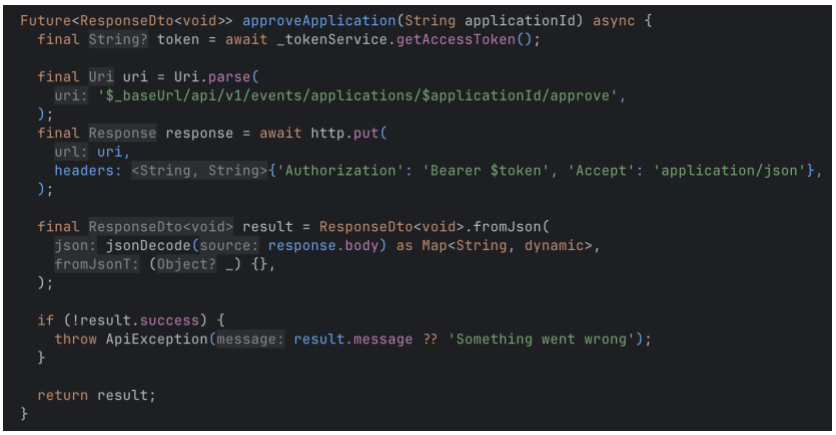
```java
public class KeycloakRealmRoleConverter implements Converter<Jwt, Collection<GrantedAuthority>> {

    @Override  Fbdrv
    public Collection<GrantedAuthority> convert(Jwt jwt) {
        Map<String, Object> realmAccess = jwt.getClaim("realm_access");
        if (realmAccess == null) {
            return Collections.emptySet();
        }

        Object rolesObj = realmAccess.get("roles");
        if (!(rolesObj instanceof Collection<?> rolesRaw)) {
            return Collections.emptySet();
        }

        Set<String> roles = rolesRaw.stream() Stream<capture of ?>
                .filter(Objects::nonNull)
                .map(Object::toString) Stream<String>
                .collect(Collectors.toSet());

        return roles.stream() Stream<String>
                .map( String role -> role.startsWith("ROLE_") ? role : "ROLE_" + role)
                .map(SimpleGrantedAuthority::new) Stream<SimpleGrantedAuthority>
                .collect(Collectors.toSet());
    }
}
```

Figure I.1 Extracts user roles from the JWT in the backend by reading the realm_access claim and converting roles into granted authorities.

```dart
List<String> extractKeycloakRoles(String accessToken) {
  final List<String> parts = accessToken.split(pattern: '.');
  if (parts.length != 3) {
    throw Exception(message: 'Invalid JWT token');
  }

  String payload = parts[1];
  payload = payload.replaceAll(from: '-', replace: '+').replaceAll(from: '_', replace: '/');
  switch (payload.length % 4) {
    case 2:
      payload += '==';
      break;
    case 3:
      payload += '=';
      break;
  }

  final String decoded = utf8.decode(codeUnits: base64.decode(encoded: payload));
  final Map<String, dynamic> claims = jsonDecode(source: decoded);

  final Map<String, dynamic>? realmAccess = claims['realm_access'] as Map<String, dynamic>?;
  if (realmAccess == null || realmAccess['roles'] == null) {
    return <String>[];
  }

  return (realmAccess['roles'] as List<dynamic>)
      .map<String>(toElement: (dynamic r) => r.toString())
      .toList();
}
```

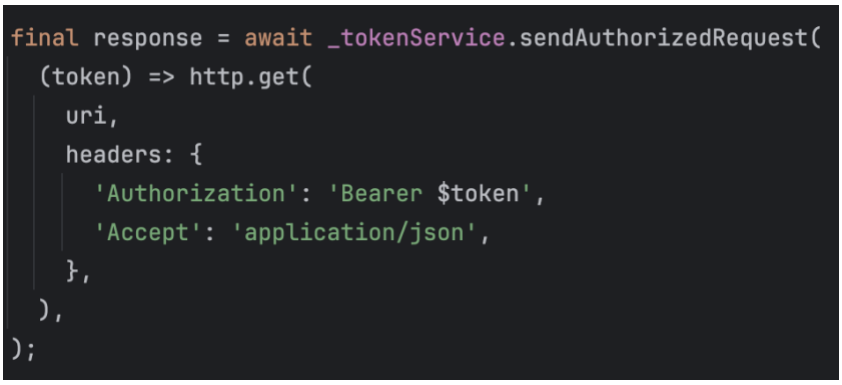
Figure I.2 Extracts user roles from the JWT on the frontend by decoding the token payload and reading roles from the realm_access claim.

10.10. Appendix J - Frontend API Integration Example



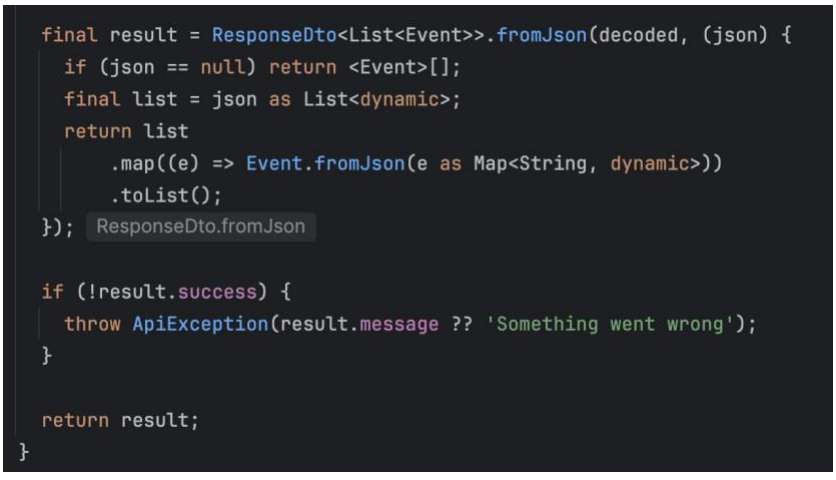
```
Future<ResponseDto<void>> approveApplication(String applicationId) async {
  final String? token = await _tokenService.getAccessToken();

  final Uri uri = Uri.parse(
    uri: '$_baseUrl/api/v1/events/applications/$applicationId/approve',
  );
  final Response response = await http.put(
    url: uri,
    headers: <String, String>{'Authorization': 'Bearer $token', 'Accept': 'application/json'},
  );

  final ResponseDto<void> result = ResponseDto<void>.fromJson(
    json: jsonDecode(source: response.body) as Map<String, dynamic>,
    fromJsonT: (Object? _) {},
  );

  if (!result.success) {
    throw ApiException(message: result.message ?? 'Something went wrong');
  }

  return result;
}
```

Figure J.1. Sends an authorized PUT request to approve an event application and parses the backend response.



```
final response = await _tokenService.sendAuthorizedRequest(
  (token) => http.get(
    uri,
    headers: {
      'Authorization': 'Bearer $token',
      'Accept': 'application/json',
    },
  ),
);
```

Figure J.2. Sends an authorized HTTP request from the frontend to a backend endpoint using a Bearer token.



```
  final result = ResponseDto<List<Event>>.fromJson(decoded, (json) {
    if (json == null) return <Event>[];
    final list = json as List<dynamic>;
    return list
        .map((e) => Event.fromJson(e as Map<String, dynamic>))
        .toList();
  }); ResponseDto.fromJson

  if (!result.success) {
    throw ApiException(result.message ?? 'Something went wrong');
  }

  return result;
}
```

Figure J.3. Parses the API response by converting JSON data into application models using a generic response wrapper.

10.11. Appendix K - Example of Service Layer Unit Test Implementation

```java
@ExtendWith(MockitoExtension.class)  Fbdrv *
class EventServiceTest {

    @Mock  9 usages
    private EventMapper eventMapper;

    @Mock  22 usages
    private EventRepository eventRepository;

    @Mock  1 usage
    private OrganizationRepository organizationRepository;

    @Mock  16 usages
    private CurrentUserService currentUserService;

    @Mock  15 usages
    private EventApplicationRepository eventApplicationRepository;

    @Mock  4 usages
    private VolunteerRepository volunteerRepository;

    @Mock  1 usage
    private EventApplicationMapper eventApplicationMapper;

    @Mock  1 usage
    private EventCalendarService eventCalendarService;

    @InjectMocks  21 usages
    private EventService eventService;
```

Figure K.1. Initializes mocks and injects dependencies in EventServiceTest using Mockito annotations.

```java
@Test  Fbdrv
void applyToEventCreatesNewApplication() {
    UUID eventId = UUID.fromString( name: "0a0a0a0a-0a0a-0a0a-0a0a-0a0a0a0a0a0a");
    VolunteerEntity volunteer = volunteer( id: "0b0b0b0b-0b0b-0b0b-0b0b-0b0b0b0b0b0b", firstName: "Amina");
    EventEntity event = event(eventId, organization( id: "0c0c0c0c-0c0c-0c0c-0c0c-0c0c0c0c0c0c", name: "Helping Hands"
    EventApplicationEntity savedApplication = application( id: "0d0d0d0d-0d0d-0d0d-0d0d-0d0d0d0d0d0d", event, volunteer

    when(currentUserService.getCurrentUser()).thenReturn(volunteer);
    when(eventRepository.findById(eventId)).thenReturn(Optional.of(event));
    when(volunteerRepository.findById(volunteer.getId())).thenReturn(Optional.of(volunteer));
    when(eventApplicationRepository.existsByEventIdAndVolunteerId(eventId, volunteer.getId())).thenReturn( t: false);
    when(eventApplicationRepository.save(any(EventApplicationEntity.class))).thenReturn(savedApplication);

    ResponseDto<UUID> response = eventService.applyToEvent(eventId);

    assertThat(response.getData()).isEqualTo(savedApplication.getId());
    ArgumentCaptor<EventApplicationEntity> captor = ArgumentCaptor.forClass(EventApplicationEntity.class);
    verify(eventApplicationRepository).save(captor.capture());
    assertThat(captor.getValue().getEvent()).isEqualTo(event);
    assertThat(captor.getValue().getVolunteer()).isEqualTo(volunteer);
}
```

Figure K.2. Example unit test method using the Arrange-Act-Assert pattern to verify service-layer behavior.