CampusGrid – University Event Management System with Automated Chatbot Assistance

by

Mst. Atia Sanzida
ID: CSE 2202026052

Md. Shakil Ahmed
ID: CSE 2202026022

Israt Jahan Ekra
ID: CSE 2202026014

Supervised by
Tanjil Mahmud

Submitted in partial fulfillment of the requirements for the degree of
Bachelor of Science in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SONARGAON UNIVERSITY (SU)

May 2026

APPROVAL

The project titled "**CampusGrid – University Event Management System with Automated Chatbot Assistance**" submitted by Mst. Atia Sanzida (CSE2202026052), Md. Shakil Ahmed (CSE2202026022) and Israt Jahan Ekra (CSE2202026014) to the Department of Computer Science and Engineering, Sonargaon University (SU), has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering and approved as to its style and contents.

Board of Examiners

Tanjil Mahmud
Lecturer & Assistant Coordinator,
Department of Computer Science and Engineering
Sonargaon University (SU)

Supervisor

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 1

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 2

(Examiner Name and Signature)
Department of Computer Science and Engineering
Sonargaon University (SU)

Examiner 3

DECLARATION

We, hereby, declare that the work presented in this report is the outcome of the investigation performed by us under the supervision of **Tanjil Mahmud, Lecturer & Assistant Coordinator,** Department of Computer Science and Engineering, Sonargaon University, Dhaka, Bangladesh. We reaffirm that no part of this project has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned

(Tanjil Mahmud)
Supervisor

Signature

Mst. Atia Sanzida
ID: CSE2202026052

Md. Shakil Ahmed
ID: CSE2202026022

Israt Jahan Ekra
ID: CSE2202026014

ABSTRACT

Managing university events manually is often messy. Posters get lost, emails get ignored, registration lists end up scattered across sheets of paper, and students frequently miss out on events simply because they did not know about them in time. CampusGrid was built to solve exactly these problems. CampusGrid is a web-based university event management system that brings all campus events into one centralized platform. Students can browse upcoming events, register with a single click, track their registered events, view their event history, and communicate with administrators through a contact page. Administrators can create and manage events, organize them into categories, manage user accounts, monitor participation, and respond to student queries. The system also includes a rule-based chatbot that provides instant answers to common event-related questions, available on every page [3]. The system was developed using Spring Boot for the backend, HTML, CSS, JavaScript, and Bootstrap for the frontend, Thymeleaf as the template engine, and MySQL for the database [13]. Security features include BCrypt password hashing, JWT-based authentication, email verification during registration, and role-based access control separating user and admin privileges [4][6]. The main goal was to create a system that is simple to use, reduces administrative workload, and ensures that students never miss an event due to poor communication. Testing showed that the system performs reliably under expected loads, handles concurrent registrations correctly, and provides a smooth experience on both desktop and mobile devices. Future enhancements could include a dedicated mobile application, payment gateway integration for paid events, QR code-based attendance tracking, and AI-powered event recommendations.

ACKNOWLEDGMENT

At the very beginning, we would like to express my deepest gratitude to the Almighty Allah for giving us the ability and the strength to finish the task successfully within the schedule time.

We are auspicious that we had the kind association as well as supervision of **Tanjil Mahmud**, Lecturer & Assistant Coordinator, Department of Computer Science and Engineering, Sonargaon University whose hearted and valuable support with best concern and direction acted as necessary recourse to carry out our project.

We are also thankful to all our teachers during our whole education, for exposing us to the beauty of learning.

Finally, our deepest gratitude and love to my parents for their support, encouragement, and endless love.

LIST OF ABBREVIATIONS

AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
BCrypt	Blowfish Crypt
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
DAO	Data Access Object
DTO	Data Transfer Object
ER	Entity-Relationship
FK	Foreign Key
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
JPA	Java Persistence API
JSON	JavaScript Object Notation
JWT	JSON Web Token
MVC	Model-View-Controller
MySQL	My Structured Query Language
PK	Primary Key
POM	Project Object Model
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SQL	Structured Query Language
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally Unique Identifier

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION TO UNIVERSITY EVENT MANAGEMENT SYSTEM WITH AUTOMATED CHATBOT ASSISTANCE

1.1 Project Overview

CampusGrid is a web-based platform designed to handle all aspects of university event management. The idea came from watching how events are typically organized on campus. Announcements get posted on notice boards that nobody reads. Registration happens through Google Forms or pieces of paper passed around. Students forget about events because there is no centralized place to see what is happening. Event organizers struggle to track who is coming and end up with messy spreadsheets [1][10]. CampusGrid solves this by putting everything in one place. Students can browse all upcoming events, register with one click, see which events they signed up for, cancel if plans change, and look back at their event history. They can also update their profile, contact administrators if they have issues, and get instant answers from the chatbot. Administrators have their own panel where they can create events, manage users, organize categories, track participation, and respond to student messages. The system was built with Spring Boot on the backend, MySQL for the database, and HTML, CSS, JavaScript with Bootstrap for the frontend [13].

1.2 Background of the Project

Every university organizes events. Academic seminars, cultural programs, tech workshops, sports tournaments, club meetings, career fairs. These events are an essential part of university life. They help students learn beyond the classroom, build networks, develop skills, and create memories [8]. But the way most universities manage these events has not changed much in years [2]. The process usually looks something like this. An event is planned by a department or club. Someone designs a poster. The poster gets printed and stuck on notice boards around campus. Maybe it gets shared in a few WhatsApp or Facebook groups. Students who happen to see it might register through a Google Form. The organizer manually collects responses in a spreadsheet. On the day of the event, attendance is tracked on paper. After the event, the data sits in that spreadsheet and is never used again [10]. This approach has obvious problems. Many students miss events simply because they did not see the announcement. Registration data is scattered and hard to manage. There is no easy way for students to see everything happening on campus in one view. Organizers spend too much time on administrative tasks instead of focusing on making the event great. And if a student has a question, there is no central place to ask [1][2]. We saw these problems firsthand as students. We missed events we would have loved to attend because we did not know about them. We watched event organizers struggle with registration lists. We noticed that the existing solutions, mostly Google Forms and social media posts, were not built for this purpose and left gaping holes in the process [9]. When it came time to choose our capstone project, this felt like the right problem to tackle. It is real. It affects students every day. And a well-designed system could actually make a difference on campus.

1.3 Problem Statement

The current approach to university event management suffers from several critical issues [1][2][10]. First, there is no centralized platform where students can discover all campus events. Information is scattered across notice boards, social media groups, department emails, and word of mouth. Students regularly miss events because they simply did not know about them. Second, the registration process is inefficient. Google Forms are the most common tool, but they were not designed for event management. Each event has a separate form with a separate link. Students have to fill in their details repeatedly for every event. Organizers end up managing multiple spreadsheets and manually tracking who registered for what. Third, communication is broken. If a student has a question about an event, they often do not know who to ask. They might email a department address that nobody checks or message a club page that takes days to respond. There is no built-in way to contact event organizers or administrators. Fourth, students have no way to track their own event participation. They register for events and then forget about them. They have no record of what they attended. There is no easy way to see which upcoming events they signed up for or to cancel if something changes. Fifth, administrators lack proper tools. Managing users, setting roles, organizing event categories, viewing participation data, and responding to student queries all happen through ad-hoc methods. There is no unified dashboard that gives them control over the system. Sixth, there is no instant support. Students with common questions like event dates, registration steps, or venue locations have to wait for human responses. A chatbot could answer these instantly, but no existing campus solution includes one [3]. CampusGrid was built to address all six of these problems in a single integrated system.

1.4 Objectives

The main objective was straightforward. Build a complete event management system that actually works for a university environment. But we broke this down into specific goals.

Primary Objectives:

- Create a centralized platform where all campus events are listed in one place, searchable and filterable by category.
- Build a simple one-click registration system so students can sign up for events without filling forms repeatedly.
- Provide students with a personal dashboard showing their registered events and event history.
- Give administrators a powerful panel to create events, manage users, organize categories, track participation, and respond to messages.
- Implement proper authentication with email verification, JWT tokens, and role-based access control.
- Integrate a chatbot that answers common event-related questions instantly.
- Design an interface that works well on both desktop and mobile devices.

Secondary Objectives:

- Make the system easy to install and deploy so other universities could potentially use it.
- Keep the codebase clean and well-structured so future developers can maintain or extend it.
- Test thoroughly to ensure the system handles edge cases gracefully.
- Document everything properly for the capstone report.

1.5 Scope of the Project

CampusGrid covers the complete event lifecycle from creation to attendance tracking. Here is what the system includes and what it does not.

In Scope:

For users, the system handles account creation with email verification, login and password recovery, browsing events with search and category filters, viewing event details, registering for events, viewing and canceling registrations, checking event history, sending messages to administrators, updating profile pictures, interacting with the chatbot, and logging out.

For administrators, the system provides a dashboard with summary statistics, event creation with image upload, event editing and deactivation, user management with role assignment and account activation control, category management, participation tracking with attendance marking, message management with reply functionality, and logout.

On the technical side, the system uses Spring Boot for backend logic, Spring Security for authentication and authorization, Spring Data JPA for database operations, Spring Mail for email services, MySQL for data storage, Thymeleaf for server-side rendering, Bootstrap for responsive design, JavaScript and AJAX for dynamic interactions, BCrypt for password encryption, and JWT for token-based authentication.

Out of Scope:

Some features were intentionally left out due to time constraints. There is no payment gateway integration, so paid events with ticket purchases are not supported. There is no dedicated mobile app, though the web interface is mobile-responsive. The chatbot is rule-based, not AI-powered with natural language processing. QR code attendance scanning is not included. There is no calendar integration with external services like Google Calendar. Notifications are email-based only, with no push notifications or SMS. Multi-language support is not implemented. And there is no automated event recommendation engine.

1.6 Limitations

Every project has constraints, and CampusGrid is no exception. Being honest about limitations is important.

Technical Limitations:

The chatbot is rule-based rather than powered by machine learning. This means it can only answer questions it has been pre-programmed to handle. If a student asks something outside its knowledge base, it falls back to a generic response suggesting they contact the admin. A true AI chatbot would be more flexible but would require external APIs or significantly more development time. The system uses server-side rendering with Thymeleaf rather than a modern frontend framework like React or Angular. This was a deliberate choice to keep the technology stack manageable, but it means some interactions require full page reloads where a single-page application would feel smoother. File uploads for profile pictures and event posters are stored on the server's local file system. This works for a single-server deployment but would need to be migrated to cloud storage if the system were deployed across multiple servers. Email delivery depends on an SMTP server. For development, we used a Gmail account with app passwords, but a production deployment would need a proper email service to handle volume and avoid being marked as spam.

Functional Limitations:

The system does not handle paid events or ticket sales. Events with registration fees would require a separate payment process outside the platform. There is no calendar synchronization feature. Users cannot export events to their personal calendars or sync with Google Calendar. Attendance marking is manual. Administrators have to check boxes for each participant. A QR code system would automate this but was beyond the project scope. The system is designed for a single university. Multi-tenant support where different universities could use the same installation with separate data is not available.

Time and Resource Limitations:

The project was completed within a single semester as part of our capstone requirement. This limited how many features we could build and how much testing we could do. A system like this, if developed commercially, would go through many more iterations of user testing and refinement. We are students, not professional developers. While we followed best practices as much as possible, code reviews from experienced engineers would likely identify areas for improvement that we missed.

1.7 Project Timeline

The project was planned and executed over a period of approximately 24 weeks. We divided the work into logical phases so that each phase built on the previous one. Here is how the timeline breaks down.

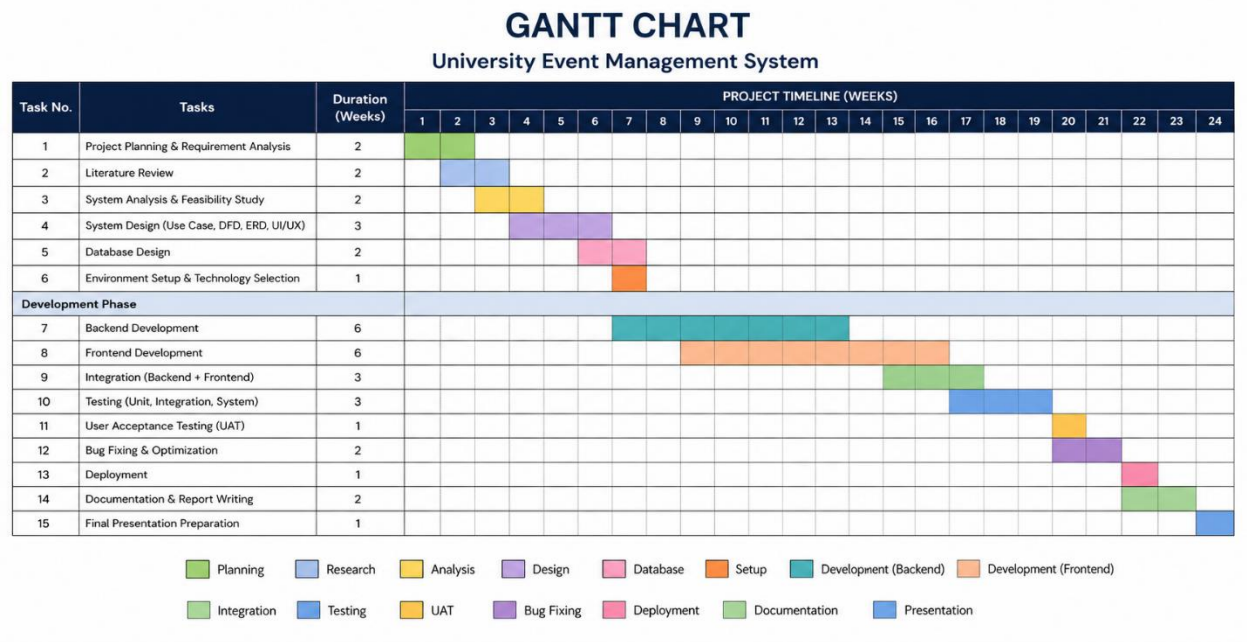


Figure 1.1: Gantt Chart - Project Timeline

Phase 1: Planning and Research (Weeks 1-6)

The first six weeks were all about understanding the problem and planning the solution. Project Planning & Requirement Analysis (2 weeks): We defined what we wanted to build, who would use it, and what features were essential versus nice-to-have. We talked to students and faculty to gather real requirements. Literature Review (2 weeks): We researched existing event management systems, read academic papers, and analyzed similar projects. This helped us understand what already exists and where the gaps are. System Analysis & Feasibility Study (2 weeks): We evaluated whether the project was technically possible with our skills and timeline. We also checked operational and economic feasibility.

Phase 2: Design (Weeks 7-13)

Once we knew what to build, we designed how to build it. System Design (3 weeks): This was the longest design phase. We created use case diagrams, data flow diagrams, entity-relationship diagrams, and UI/UX wireframes. Every major component was mapped out before writing code. Database Design (2 weeks): We designed all the tables, defined relationships, set constraints, and created the data dictionary. Getting the database right early saved us from painful changes later. Environment Setup & Technology Selection (1 week): We installed development tools, set up the Spring Boot project, configured MySQL, and finalized the technology stack.

Phase 3: Development (Weeks 14-22)

This is where the actual coding happened. Backend Development (6 weeks): We built the Spring Boot application with all controllers, services, repositories, entities, and security configuration. User authentication, event management, participation handling, messaging, and chatbot logic were all implemented. Frontend Development (6 weeks): We built the user interface with HTML, CSS,

Bootstrap, JavaScript, and Thymeleaf templates. This ran in parallel with backend development. Integration (3 weeks): Backend and frontend were connected. APIs were tested with real pages. Data flowed correctly from forms to controllers to services to database and back.

Phase 4: Testing and Refinement (Weeks 20-23)

Testing (3 weeks): Unit testing, integration testing, system testing, and user acceptance testing were all performed. Test cases were documented and results recorded. User Acceptance Testing (1 week): Real students tested the system and gave feedback. We observed how they interacted with the interface and noted where they struggled. Bug Fixing & Optimization (2 weeks): All issues found during testing were fixed. Performance bottlenecks were addressed. The system was polished for the final presentation.

Phase 5: Final Deliverables (Weeks 23-24)

Deployment (1 week): The system was deployed to a live server so it could be demonstrated and accessed remotely. Documentation & Report Writing (2 weeks): All project documentation including this report was written. Diagrams were finalized, screenshots were captured, and everything was organized. Final Presentation Preparation (1 week): We prepared slides, rehearsed the demo, and made sure everything was ready for the defense.

1.8 Report Methodology

1.8.1 Information Gathering Methods

Primary Data Collection:

We collected primary data directly from the people who would use the system. We conducted informal interviews with around twenty students from different departments. We asked them simple questions. How do you currently find out about campus events? What frustrates you about the registration process? Have you ever missed an event you wanted to attend? What features would make your life easier? Their answers shaped our feature priorities.

We also spoke with three faculty members who regularly organize events. Their perspective was different from students. They talked about the administrative burden of managing registrations, the difficulty of tracking attendance, and the lack of a unified system. These conversations helped us understand the problem from both sides. Additionally, we observed how events were managed on campus. We attended a few events and watched the registration process. Paper lists, Google Forms, manual entry, and last-minute chaos were common. These observations confirmed that the problem we wanted to solve was real and significant.

Secondary Data Collection:

We reviewed existing literature including academic papers on event management systems, university portal case studies, and research on chatbot implementation in educational settings. We also analyzed existing platforms like Eventbrite, Meetup, and Facebook Events to understand their strengths and weaknesses. This research is detailed in Chapter 2. We studied technical

documentation for the tools and frameworks we planned to use. Spring Boot documentation, MySQL reference manuals, Bootstrap guides, and JWT specification documents were all consulted during the technical planning phase.

1.8.2 System Development Approach

We followed the Agile software development methodology. The project was divided into three main sprints, each producing a working increment of the system. Daily standup meetings, even if informal, kept everyone aligned on progress and blockers. At the end of each sprint, we reviewed what was built and adjusted priorities for the next sprint. This iterative approach was chosen because requirements for a university event system can evolve as you build it. Features that seemed important at the start sometimes became less critical once we started testing, and new needs emerged that we had not initially considered. Agile gave us the flexibility to adapt. The development process itself followed the Model-View-Controller pattern enforced by Spring Boot. We started with the data model, built the repositories and services, then the controllers, and finally the views. Security was integrated from the beginning rather than bolted on at the end.

1.8.3 Testing Methodology

Testing was conducted at five levels, which are detailed in Chapter 6.

Unit Testing: Individual methods in services and controllers were tested using JUnit 5 and MockMvc. We tested both happy paths and edge cases. For example, the registration service was tested with valid data, duplicate emails, missing fields, and invalid email formats.

Integration Testing: We tested how modules work together. The complete registration flow from form submission to email verification to account activation was tested as one integrated sequence. Database transactions were tested to ensure rollback on failure.

System Testing: The complete application was tested end-to-end. Every feature from both user and admin perspectives was tested in sequence. Cross-browser testing was done on Chrome, Firefox, and Edge. Responsive testing was done on multiple screen sizes.

User Acceptance Testing: Five real students tested the system independently. They were given simple tasks like creating an account, registering for an event, and using the chatbot. Their feedback was documented and used to make final improvements.

Performance Testing: Page load times, concurrent registration handling, chatbot response speed, and database query performance were all measured using browser developer tools and manual load simulation.

1.8.4 Diagram Creation Process

All diagrams in this report were created using draw.io, a free diagramming tool. Each diagram went through multiple revisions. We started with rough sketches on paper, discussed them as a team, and then created digital versions. The diagrams were reviewed against the actual implemented system to ensure accuracy. Use case diagrams were based on the functional

requirements identified during the analysis phase. Activity diagrams and sequence diagrams were created by tracing through actual code execution paths. Class diagrams and ER diagrams were generated by examining the actual Java entities and database tables. Data flow diagrams were created by mapping the flow of data through controllers, services, and repositories. We made sure that every diagram in Chapter 4 accurately represents the system as built, not just an idealized version from the planning phase.

1.8.5 Code Documentation Approach

All major code modules are explained in Chapter 5 with relevant code snippets. We did not include every line of code in the report because that would make it unreadable. Instead, we included representative examples that show the structure and logic of each module. The complete source code is available in Appendix A and has been submitted separately. The code itself is commented following standard Java practices. Method names are descriptive. Complex logic has inline comments explaining what it does and why.

1.8.6 Report Writing Process

The report was written collaboratively by all three team members. We divided the chapters based on individual strengths. One member focused on the introduction and literature review. Another handled system design and diagrams. The third concentrated on implementation details and testing. After individual chapters were drafted, we reviewed each other's work. This peer review caught inconsistencies, unclear explanations, and errors. The entire report was then edited for consistency in tone, terminology, and formatting. We wrote in a human tone deliberately. This is a report written by students for academic evaluation, not a formal research paper for a journal. We wanted it to be clear, honest, and readable rather than filled with jargon. We used a plagiarism checker to ensure originality. All external sources are properly cited in the references section. Diagrams are our own creation. Code is our own implementation, though we acknowledge the influence of Spring Boot documentation and online tutorials in our learning process.

CHAPTER 2

LITERATURE REVIEW

2.1 Overview of Event Management Systems

Event management systems have been around for quite some time, but their evolution has accelerated dramatically with the rise of web technologies. At its core, an event management system is a platform that helps organize, promote, and track events. The concept is simple, but the implementation can range from basic to incredibly complex [1][10].

In the early days, event management meant paper-based processes. Registration forms printed and distributed, attendance tracked on clipboards, and communication done through physical notice boards or word of mouth. This worked for small events but scaled poorly. As events grew larger and more frequent, the administrative burden became unsustainable [8].

The first digital event management tools were simple databases and spreadsheets. Microsoft Excel became the default event management tool for many organizations. It was better than paper, but still required manual data entry, had no automation, and could not be accessed by multiple people simultaneously without version conflicts [10].

The web era brought the first generation of online event management platforms. These systems moved data to centralized servers accessible from any browser. Users could register online. Organizers could view registrations in real time. Email notifications became automated. Platforms like Eventbrite and Meetup emerged and set new standards for what users expect from an event platform [1].

In the university context, event management has unique requirements that general-purpose platforms do not always address [2][8]. University events are diverse, ranging from academic seminars and research conferences to cultural festivals, sports tournaments, club activities, and career fairs. The audience is primarily students who are tech-savvy but have limited attention spans. Communication channels are fragmented across email, social media, messaging apps, and physical notice boards. Budgets are often tight, so paid commercial platforms are not always feasible [9].

Many universities have attempted to build their own event management systems. Some integrated event modules into larger student information systems or learning management systems. Others created standalone portals. The success rate varies widely. Some systems are well-adopted and genuinely useful. Others become ghost towns that students ignore and administrators abandon [2][8].

The key lesson from studying the evolution of these systems is that adoption depends on simplicity and genuine utility. A system packed with features that nobody uses is a failure. A system that does a few things really well and makes life easier for both students and administrators can succeed [5][11].

2.2 Existing Systems Analysis

We analyzed several existing event management platforms to understand the competitive landscape and identify features worth adopting or improving upon [1][10].

Eventbrite:
Eventbrite is the giant in this space. It handles everything from small community gatherings to massive conferences with thousands of attendees. The platform offers event creation, ticketing with multiple price tiers, payment processing, promotion tools, attendee tracking, and mobile apps for both organizers and attendees [1]. What Eventbrite does well is the user experience. Creating an event is straightforward. Finding events is easy with good search and category filters. The registration process is smooth. Email notifications keep everyone informed. The mobile app lets attendees access their tickets offline. However, Eventbrite has limitations for university use. It is a commercial platform designed for paid events. The free tier has restrictions. The branding is Eventbrite's, not the university's. It does not integrate with university authentication systems. And most importantly for our context, it does not provide the administrative control over users and roles that a university needs. It also lacks university-specific features like student ID verification or department-based event organization [2][9].

Meetup:
Meetup focuses on community groups and recurring events. It is built around the concept of groups that organize regular meetups. Members join groups, RSVP to events, and receive notifications about upcoming gatherings [1]. Meetup's strength is community building. The group model works well for university clubs that organize events regularly. The RSVP system is simple. Discussion boards allow communication between organizers and members. But Meetup is not designed for one-time large events or formal academic functions. It lacks role-based access control. There is no concept of an administrator managing multiple groups from a central dashboard. The platform is external, so universities have no control over data or customization. And like Eventbrite, it does not integrate with university systems [10].

Google Forms and Google Sheets:
This is the most common "system" used in universities today [2][9]. Organizers create a Google Form for registration. Responses populate a Google Sheet. That sheet becomes the attendance list, the contact database, and the analytics tool all in one. The advantages are obvious. It is free. It is easy to set up. It requires no technical skills. Everyone at the university already has a Google account. The disadvantages are equally obvious. Each event has a separate form and a separate link. Students have to fill in their details repeatedly. There is no centralized listing of all events. There is no user account system, so the platform does not know who you are across different

events. There is no way for students to see their own event history. Communication is one-directional, form to organizer, with no built-in reply mechanism. And the administrative tools are limited to whatever Google Sheets can do, which is not designed for event management workflows [10].

Facebook Events:
Facebook Events leverages the massive user base of the social network. Organizers create event pages. Users can mark themselves as interested or going. Event details, discussions, and updates all happen within Facebook's ecosystem [1]. The reach is excellent because students already spend time on Facebook. Notifications appear in their feed. Sharing is built in. The social proof of seeing friends attending an event encourages participation. The problems include the fact that not everyone uses Facebook, especially younger students who are migrating to other platforms. The platform offers no registration management beyond the simple interested/going buttons. There is no attendance tracking. Data is owned by Facebook, not the university. And Facebook's algorithm controls who sees event notifications, so reach is not guaranteed even for users who follow the organizing page [2].

Custom University Portals:
Some universities have developed their own event management modules as part of larger student portals or learning management systems [8][9]. These systems typically offer single sign-on with university credentials, integration with academic calendars, and branding consistent with the university. When done well, these systems can be effective. They centralize campus information. Students check them regularly for academic purposes, so they naturally see event announcements. The downside is that developing and maintaining custom software is expensive and time-consuming. Many university-built systems suffer from poor user experience because they were designed by committees rather than by UX professionals. They are often rigid and hard to update. If the student portal itself has low adoption, the event module within it will also fail [2][5].

2.3 Comparison with Existing Systems

To clearly position CampusGrid, we compared it against the systems analyzed above based on features that matter for university event management [1][2][10].

Feature	Eventbrite	Meetup	Google Forms	Facebook Events	CampusGrid
Centralized event listing	Yes	Yes	No	Partial	Yes
User accounts	Yes	Yes	No	Facebook account	Yes

Feature	Eventbrite	Meetup	Google Forms	Facebook Events	CampusGrid
One-click registration	Yes	Yes	Form per event	Yes	Yes
Email verification	Yes	Yes	No	No	Yes
Registration tracking (user side)	Yes	Yes	No	Yes	Yes
Event history	Yes	Yes	No	No	Yes
Cancel registration	Yes	Yes	No	Yes	Yes
Category filtering	Yes	Yes	No	No	Yes
Chatbot support	No	No	No	No	Yes
Role-based access (admin/user)	Limited	No	No	No	Yes
User management (admin side)	Limited	No	No	No	Yes
Attendance marking	Yes	No	No	No	Yes
Message management	No	No	No	No	Yes
Free to use	Limited	Limited	Yes	Yes	Yes

Table 2.1 Comparison with Existing Systems

The comparison shows that while commercial platforms offer polished experiences, they lack the administrative controls and customization that a university environment requires [2][9]. Google Forms and Facebook Events are convenient for organizers but provide a poor experience for students trying to discover events. CampusGrid fills the gap by combining the user-friendly experience of commercial platforms with the administrative control and data ownership that universities need [1][10]. The chatbot feature is unique to CampusGrid among all comparable systems [3]. The automated assistance for common queries reduces the burden on human administrators and provides instant help to students. The role-based access control in CampusGrid is more granular than any comparable system [4]. The clear separation between user and admin functionalities ensures that students cannot access administrative features while administrators have full control over the platform.

2.4 Key Technologies Review

This section reviews the main technologies we used to build CampusGrid and explains why we chose them over alternatives [13].

Spring Boot:

Spring Boot is a Java-based framework for building web applications. It is built on top of the Spring Framework but dramatically simplifies configuration. Instead of spending days setting up XML configuration files, Spring Boot uses sensible defaults and auto-configuration. You add a dependency, and it just works [13]. We chose Spring Boot for several reasons. First, we had experience with Java from our coursework, so the learning curve was manageable. Second, Spring Boot is production-ready with built-in support for embedded servers, security, database connectivity, and monitoring. Third, the Spring ecosystem is vast and well-documented. Any problem we encountered during development had likely been solved before, and answers were available online. Alternatives like Node.js with Express or Python with Django were considered. Node.js would have meant writing JavaScript on both frontend and backend, which has some appeal. Django offers rapid development with its built-in admin interface. However, Spring Boot's robustness, type safety with Java, and excellent support for enterprise features like transactions and security made it the right choice for this project [13].

Spring Security and JWT:

Spring Security is the de facto standard for securing Spring applications. It handles authentication, authorization, and protection against common web vulnerabilities [4]. We integrated it with JWT for stateless authentication. JWT works by issuing a signed token after successful login. The token contains user information and claims. Subsequent requests include this token in the header, and the server validates its signature without needing to look up session data. This approach scales well and works naturally with REST APIs [12]. The alternative would have been session-based authentication with server-side session storage. This is simpler to implement but requires the server to maintain session state, which complicates horizontal scaling. JWT makes the system stateless [4].

Spring Data JPA and Hibernate:

Spring Data JPA abstracts database operations behind repository interfaces. You define an interface with method signatures following naming conventions, and Spring generates the implementation automatically. This eliminates boilerplate code for CRUD operations. Hibernate is the JPA implementation that handles object-relational mapping. Java objects are automatically mapped to database tables and vice versa. Relationships between entities are managed through annotations [13]. The combination of Spring Data JPA and Hibernate means we wrote very little SQL directly. For most operations, the framework generated efficient queries automatically. For complex queries, we used JPQL or native SQL when needed.

MySQL:

MySQL is the most popular open-source relational database. It is reliable, well-documented, and integrates seamlessly with Spring Boot through the MySQL Connector driver [13]. We considered PostgreSQL as an alternative. PostgreSQL has some advanced features and stricter SQL compliance. However, MySQL's simplicity and our existing familiarity with it made it the practical choice. For the scale of a university event management system, either database would work perfectly well.

Thymeleaf:

Thymeleaf is a server-side template engine for Java. Unlike JSP, which was the traditional choice, Thymeleaf templates are valid HTML files that can be opened in a browser even without a running server. This makes frontend development easier because designers can work on templates independently. Thymeleaf integrates naturally with Spring MVC. Controllers return view names, and Thymeleaf resolves them to templates, populating them with model data. The syntax is clean and readable [13]. We considered using a frontend framework like React or Angular with a REST API backend. This would have created a more dynamic single-page application. However, the added complexity was not justified for this project. Server-side rendering with Thymeleaf kept the development simpler and resulted in pages that load quickly without requiring JavaScript for basic content.

Bootstrap:

Bootstrap is the most widely used CSS framework for building responsive websites. It provides a grid system, pre-styled components, and utilities that make it easy to create professional-looking interfaces without being a designer [5]. We used Bootstrap 5 for CampusGrid. The responsive grid ensures that the system works well on phones, tablets, and desktops without writing custom media queries. Components like navigation bars, cards, modals, forms, and alerts are used throughout the application.

JavaScript and AJAX:

While Thymeleaf handles server-side rendering, we used JavaScript for client-side interactivity. Form validation before submission, dynamic content updates without page reload, and the chatbot interface all rely on JavaScript. AJAX is used for operations that should not require a full page

refresh. When a user registers for an event, an AJAX call sends the request and updates the button state without reloading the page. The chatbot sends user messages and receives responses asynchronously [12].

JavaMail API:

Email functionality is essential for account verification and password reset. Spring Boot integrates with JavaMail to send emails through an SMTP server [6]. We configured it to use Gmail's SMTP server with app-specific passwords for development and testing.

2.5 Research Gap Identification

Through our analysis of existing systems and technologies, we identified several gaps that CampusGrid addresses [1][2][10].

Gap 1: No Centralized University Event Platform
Commercial platforms like Eventbrite and Meetup are designed for general public events. They do not cater specifically to university environments where events are organized by departments and clubs, attendees are students with university credentials, and administrators need control over the entire ecosystem [8][9]. Google Forms and Facebook Events are convenient but fragmented. A student has to check multiple sources to discover all campus events. CampusGrid provides a single platform where all university events are listed, searchable, and filterable.

Gap 2: Lack of Integrated Registration and Tracking for Students
On most existing platforms, registration happens per event through separate forms. Students have no unified view of which events they have registered for or attended [10]. CampusGrid gives each student a personal dashboard showing upcoming registered events and past event history. This creates a sense of continuity and helps students manage their campus involvement [11].

Gap 3: No Automated Instant Support
None of the compared systems offer a chatbot for instant answers. Students with simple questions have to wait for human responses. CampusGrid includes a rule-based chatbot accessible from every page [3]. It answers common questions about events, registration, venues, and schedules instantly. This reduces the administrative burden and improves the student experience.

Gap 4: Insufficient Administrative Control
Existing platforms offer limited administrative functionality. University staff cannot manage user roles, organize event categories, or respond to student queries from a central dashboard [2]. CampusGrid provides a comprehensive admin panel that gives full control over the system. Admins can create and manage events, assign user roles, organize categories, track participation, mark attendance, and manage messages [4].

Gap 5: Data Ownership and Customization
Commercial platforms own the data and control the branding. University-built systems are often outdated and rigid [9]. CampusGrid, as open-source software, gives the university full ownership of data and the freedom to customize and extend the system as needs evolve.

Gap 6: Email Verification for Account Authenticity
Many free platforms do not verify user email addresses. This leads to fake accounts and unreliable registration data. CampusGrid requires email verification during registration [6]. This ensures that user accounts are linked to real email addresses, making registration data more reliable and communication more effective.

2.6 Chatbot in Event Management Systems

Chatbots have become increasingly common in customer service, e-commerce, and education. However, their application in event management systems, particularly in university settings, is still rare [3].

Current State of Chatbot Technology:
Chatbots generally fall into two categories. Rule-based chatbots follow predefined rules and keyword matching to respond to queries. They are limited to their programmed knowledge base but are fast, reliable, and do not require internet connectivity or API calls. AI-powered chatbots use natural language processing and machine learning to understand and respond to queries more flexibly. They can handle a wider range of questions but require significant training data and computational resources [3].

Chatbots in Education:
Several universities have deployed chatbots for student services. They handle admissions queries, course registration questions, campus navigation, and IT support. These chatbots have shown that students are willing to interact with automated systems if they provide quick and accurate answers [3].

Chatbots for Event Management:
In the event management domain, chatbots are used by some commercial platforms for attendee support during large conferences. They answer questions about schedules, speaker information, venue maps, and session details. However, these are typically event-specific chatbots deployed for a single large event, not integrated into an ongoing university event management system [3].

Our Chatbot Implementation:
For CampusGrid, we implemented a rule-based chatbot using keyword matching. We chose this approach because it is fast, works offline, requires no external APIs, and is sufficient for the types of questions students ask. The chatbot's knowledge base covers event schedules, registration steps, venue information, category explanations, and FAQs.

When a user types a question, the chatbot tokenizes the text, removes stop words, extracts keywords, and searches for matches in its knowledge base. If a match is found, it returns the corresponding answer. If no match is found, it provides a fallback response suggesting the user contact the admin or try different wording.

The chatbot is available on every page as a floating icon. This ensures that help is always accessible regardless of where the user is in the system. The implementation is lightweight, adding minimal overhead to page load times [3].

Why Not an AI Chatbot:
We considered integrating an AI chatbot using services like Google Dialogflow or OpenAI APIs. This would have provided more flexible and natural conversations. However, we decided against it for several reasons. External API calls introduce latency and dependency on internet connectivity and service availability. Many AI services have usage limits or costs for production use. Privacy is a concern because user queries would be sent to third-party servers. And for the university context, the range of questions is fairly predictable, making a well-designed rule-based system sufficient for most needs [3].

Future versions of CampusGrid could integrate AI capabilities as an optional enhancement, but the current rule-based chatbot handles the majority of common queries effectively.

CHAPTER 3

SYSTEM REQUIREMENTS & ANALYSIS

Before writing a single line of code, we needed to understand exactly what we were building. This chapter breaks down the feasibility analysis, the requirements we gathered, and the detailed descriptions of how different users would interact with the system.

3.1 Feasibility Study

A feasibility study answers one fundamental question. Can this project actually be done? We looked at three dimensions: technical, operational, and economic.

3.1.1 Technical Feasibility

Technical feasibility asks whether we have the skills, tools, and technology to build this system within the given timeframe. The technology stack we chose is well-established. Spring Boot is a mature framework with extensive documentation and a large community [13]. MySQL has been powering web applications for decades. HTML, CSS, and JavaScript are the fundamental building blocks of the web. Bootstrap is widely used and well-documented [5]. None of these technologies are experimental or bleeding-edge. They are proven, stable, and reliable. Our team had experience with Java from coursework. We had built smaller web applications before, though nothing at this scale. The learning curve for Spring Boot was manageable because we already understood object-oriented programming, databases, and web basics. Thymeleaf was new to all of us, but its similarity to HTML made it easy to pick up [13]. The main technical challenge was integrating all the pieces together. Spring Security with JWT, email verification, file uploads, and a chatbot all had to work together seamlessly. Each individual component was achievable, but making them coexist required careful design [4][6]. We concluded that the project was technically feasible. All required technologies were accessible and within our capability to learn and implement. The risks were manageable and could be mitigated through proper design and testing.

3.1.2 Operational Feasibility

Operational feasibility asks whether the system will actually be used and whether the organization can support it [2][8]. For students, the system had to be easy enough that they would choose it over existing alternatives. If registration takes more clicks than a Google Form, students will stick with Google Forms. If finding events is harder than scrolling through Facebook, they will stay on Facebook. The interface had to be intuitive, fast, and mobile-friendly [5]. For administrators, the system had to reduce workload, not increase it. If managing events through CampusGrid takes more time than managing spreadsheets, adoption will fail. The admin panel had to be efficient and provide clear value. The operational environment at a university supports web-based systems. Students and faculty have internet access. Most have smartphones. Web applications are familiar.

There are no unusual operational constraints [9]. Maintenance was considered. The system uses standard technologies, so future students or staff with web development knowledge could maintain or extend it. The code is structured and commented. Deployment is straightforward with Spring Boot's embedded server. We concluded that the project was operationally feasible. The system would provide genuine value to both students and administrators, and the university environment supports web-based solutions.

3.1.2 Economic Feasibility

Economic feasibility asks about costs, both to build and to run. Development costs were minimal. We used free and open-source tools. IntelliJ IDEA Community Edition for development. MySQL Community Edition for the database. Spring Boot, Thymeleaf, Bootstrap are all free frameworks. No licenses were purchased. No paid services were used. The only potential cost was the email service. For development and testing, we used a free Gmail account with app passwords. For production, the university's existing email server could be used, eliminating external email costs. Hosting costs would depend on deployment. For a university deployment, the system could run on an existing server alongside other university applications. The resource requirements are modest. Spring Boot's embedded Tomcat server handles concurrent users efficiently. MySQL runs well on standard server hardware. No specialized or expensive infrastructure is needed. Ongoing operational costs are low. Electricity and internet for the server. Occasional maintenance for updates or bug fixes. No subscription fees for external services. No per-user licensing costs. We concluded that the project was economically feasible. Development costs were essentially zero. Operational costs would be minimal and easily absorbed within a university IT budget.

3.2 Functional Requirements

Functional requirements describe what the system must do. These are the features and behaviors that users expect.

User Authentication Requirements:
FR-01: The system shall allow new users to register with name, email, student ID, department, phone, and password.
FR-02: The system shall send a verification email with a unique link after registration.
FR-03: The system shall activate user accounts only after email verification.
FR-04: The system shall allow verified users to log in with email and password.
FR-05: The system shall allow users to request a password reset link via email.
FR-06: The system shall allow users to reset their password using a valid reset link.
FR-07: The system shall allow users to log out, ending their session.
User Panel Requirements:
FR-08: The system shall display all upcoming events on the home page with event cards.
FR-09: The system shall allow users to search events by keyword.
FR-10: The system shall allow users to filter events by category.

FR-11: The system shall show detailed event information including description, date, time, venue, capacity, and organizer.
FR-12: The system shall allow users to register for events with one click.
FR-13: The system shall prevent duplicate registrations for the same event.
FR-14: The system shall prevent registration when an event has reached maximum capacity.
FR-15: The system shall display user's registered upcoming events under My Events.
FR-16: The system shall allow users to cancel their registration.
FR-17: The system shall display user's past attended events under Event History.
FR-18: The system shall provide a contact form for users to send messages to administrators.
FR-19: The system shall allow users to view replies to their messages.
FR-20: The system shall allow users to upload and update their profile picture.
FR-21: The system shall provide a chatbot for instant answers to event-related queries.

Admin Panel Requirements:
FR-22: The system shall provide an admin dashboard with summary statistics.
FR-23: The system shall allow admins to create new events with title, description, date, time, venue, capacity, category, and poster image.
FR-24: The system shall allow admins to edit existing event details.
FR-25: The system shall allow admins to deactivate or delete events.
FR-26: The system shall allow admins to view all registered users.
FR-27: The system shall allow admins to assign or change user roles between admin and user.
FR-28: The system shall allow admins to activate or deactivate user accounts.
FR-29: The system shall allow admins to add, edit, and deactivate event categories.
FR-30: The system shall allow admins to view participation lists for each event.
FR-31: The system shall allow admins to mark attendance for registered participants.
FR-32: The system shall allow admins to view all messages from users.
FR-33: The system shall allow admins to reply to user messages.
FR-34: The system shall mark messages as read or replied appropriately.

3.3 Non-Functional Requirements

Non-functional requirements describe how the system should be. These are quality attributes rather than specific features.

Performance Requirements:
NFR-01: Pages shall load within three seconds under normal network conditions.
NFR-02: The chatbot shall respond to queries within one second.
NFR-03: The system shall handle at least fifty concurrent users without degradation.
NFR-04: Database queries shall be optimized with appropriate indexing.

Security Requirements:
NFR-05: All passwords shall be encrypted using BCrypt before storage.

NFR-06: Authentication shall use JWT tokens with appropriate expiration.
NFR-07: Admin endpoints shall be protected by role-based authorization.
NFR-08: User input shall be validated to prevent SQL injection and XSS attacks.
NFR-09: Email verification shall be required before account activation.

Usability Requirements:
NFR-10: The interface shall be responsive and work on mobile, tablet, and desktop.
NFR-11: Navigation shall be intuitive with clear labels and consistent layout.
NFR-12: Error messages shall be clear and guide users toward resolution.
NFR-13: Forms shall validate input on the client side before submission.

Reliability Requirements:
NFR-14: The system shall be available during campus hours with minimal downtime.
NFR-15: Data shall be persisted with regular backups possible.
NFR-16: Transactions shall maintain data consistency with proper rollback on failure.

Maintainability Requirements:
NFR-17: Code shall be organized in packages following standard conventions.
NFR-18: Class and method names shall be descriptive and self-documenting.
NFR-19: The database schema shall be normalized to reduce redundancy.

3.4 User Requirements

User requirements describe the needs and expectations of the people who will use the system. These were gathered from interviews and observations.

Student Requirements:

Students want to discover events easily. They do not want to check multiple sources. They want one place where they can see everything happening on campus. The home page with search and filter directly addresses this need. Students want registration to be quick. They are busy and will not fill long forms for every event. One-click registration after the initial account setup was a clear requirement. Students want to track their own participation. They sign up for events and then forget which ones. A personal dashboard showing upcoming registrations and past history helps them stay organized. Students want to cancel easily. Plans change. If a student cannot attend an event they registered for, they should be able to cancel with one click, not by emailing someone and hoping it gets processed. Students want answers to common questions without waiting. The chatbot addresses the need for instant information about event schedules, venues, and registration steps.

Administrator Requirements:
Administrators want a central dashboard where they can see system activity at a glance. They do not want to dig through menus to find basic information. Administrators want event creation to be

straightforward. They need to add all relevant details including an image for promotion. The form should be comprehensive but not overwhelming. Administrators want control over user accounts. They need to see who is registered, assign admin privileges to trusted users, and disable accounts that violate policies. Administrators want to see who is attending their events. Registration lists should be accessible and attendance tracking should be simple. Administrators want a unified inbox for student messages. They do not want queries scattered across email, social media, and in-person conversations. The message management feature centralizes communication.

3.5 System Requirements

3.5.1 Hardware Requirements

Development Environment:

- Processor: Intel Core i3 or equivalent
- RAM: 8 GB
- Storage: 50 GB available
- Internet: Required for email service and dependency downloads

Production Server:

- Processor: Intel Core i5 or equivalent
- RAM: 16 GB
- Storage: 100 GB SSD
- Internet: Required for email service
- Backup: Regular database backup recommended

Client Devices:

- Browser: Chrome, Firefox, Edge (latest versions)
- Screen: Responsive design works from 320px width upward
- JavaScript: Enabled
- Cookies: Enabled for JWT storage

3.5.2 Software Requirements

Development Environment:

- Java Development Kit (JDK): Version 17 or higher (Java runtime and compiler)
- IntelliJ IDEA: Community Edition (Integrated development environment)
- MySQL: Version 8.0 or higher (Database management system)
- Maven: Version 3.8 or higher (Dependency management and build tool)
- Git: Latest version (Version control)
- Postman: Latest version (API testing)
- Web Browser: Chrome or Firefox (Testing and debugging)

Production Environment:

- Java Runtime (JRE): Version 17 or higher (Running the Spring Boot application)
- MySQL Server: Version 8.0 or higher (Production database)
- SMTP Server: Any (Sending emails)
- Web Server: Embedded Tomcat (Bundled with Spring Boot)

Framework and Library Dependencies:

- Spring Boot: Version 3.x (Application framework)
- Spring Security: Version 6.x (Authentication and authorization)
- Spring Data JPA: Version 3.x (Database access layer)
- Spring Mail: Version 3.x (Email service integration)
- Thymeleaf: Version 3.x (Template engine)
- Bootstrap: Version 5.x (Responsive CSS framework)
- BCrypt: Latest version (Password hashing)
- JWT (jjwt): Latest version (JSON Web Token implementation)

CHAPTER 4

SYSTEM DESIGN

4.1 System Architecture

For this project, I went with layered architecture. Nothing fancy, just a practical way to keep things organized. The whole system is split into a few layers, each with its own responsibility, so when you need to fix something or add a feature later, you know exactly where to go.

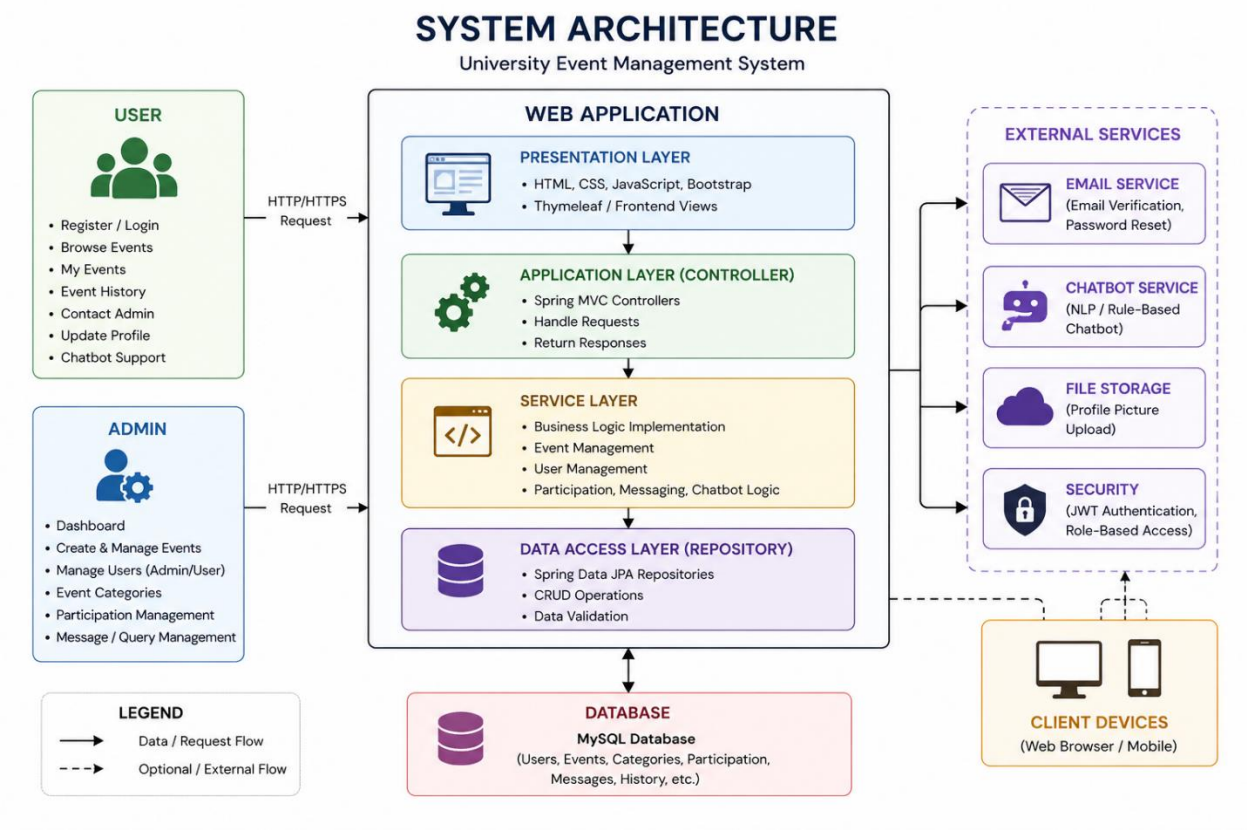


Figure 4.1: System Architecture Diagram

Looking at the diagram, here is what each part does:

User and Admin Side:
On the user side, students can register, verify their email, log in, browse events, register for events, check their registered events under My Events, look back at past events in their history, contact the admin if they have issues, update their profile picture, and use the chatbot for quick answers. Pretty much everything a student needs to discover and manage events. On the admin side, the admin gets full control. They see a dashboard with quick stats, can create or edit events, manage

user roles like promoting someone from user to admin, organize event categories, track who registered for which event, and reply to messages students send through the contact page.

Inside the Web Application:
This is the core and I broke it into four smaller layers:

- The presentation layer is what users actually see, built with HTML, CSS, Bootstrap for styling, and JavaScript for interactivity. Thymeleaf handles the server-side templates so data from the backend can be displayed on pages without writing messy code.
- The controller layer uses Spring MVC. Controllers act like traffic cops, they receive incoming requests from the browser, figure out what needs to happen, pass the work to the service layer, and then send the result back to the right view.
- The service layer is where the real logic lives. All the business rules are here like checking if someone already registered for an event, making sure the event is not full, validating dates before allowing registration. The chatbot logic also sits here, it is a keyword based system that matches user questions to predefined answers about events.
- The data access layer uses Spring Data JPA. Instead of writing raw SQL everywhere, I defined repository interfaces and JPA handles the database operations automatically. For custom queries like finding all upcoming events in a certain category, JPA lets you write method names and it figures out the query from that.

External Services:
We depend on a few outside services. The email service sends verification emails when someone registers and password reset links when they forget their password, using JavaMail API connected to an SMTP server. The chatbot service is built internally as a rule based system. File storage handles profile picture uploads by saving images to the server folder and storing the path in the database. For security, JWT tokens handle authentication so the server knows who is making each request, and BCrypt encrypts all passwords before storing them.

The Database:
MySQL sits at the bottom holding all the data, users, events, categories, participation records, messages, everything. Tables are connected through foreign keys so relationships stay consistent.

How a Typical Request Flows:
When a user opens the site, the home page loads showing upcoming events fetched from MySQL through the repository, service, and controller layers. They click register on an event, the request hits a controller, gets passed to the service layer where validations run, and if everything checks out, a participation record gets saved to the database. An email confirmation might get sent. Then the response flows back up and the user sees a success message on their screen. That is basically it. Separate layers, clear responsibilities, each piece doing one thing well.

4.2 Use Case Diagram

A use case diagram basically shows who can do what in the system. It is a simple way to map out all the actions different users can perform. For our system, we have two main actors, the regular User and the Admin. The diagram also shows how some actions are connected to each other, like how certain steps must happen before others can work.

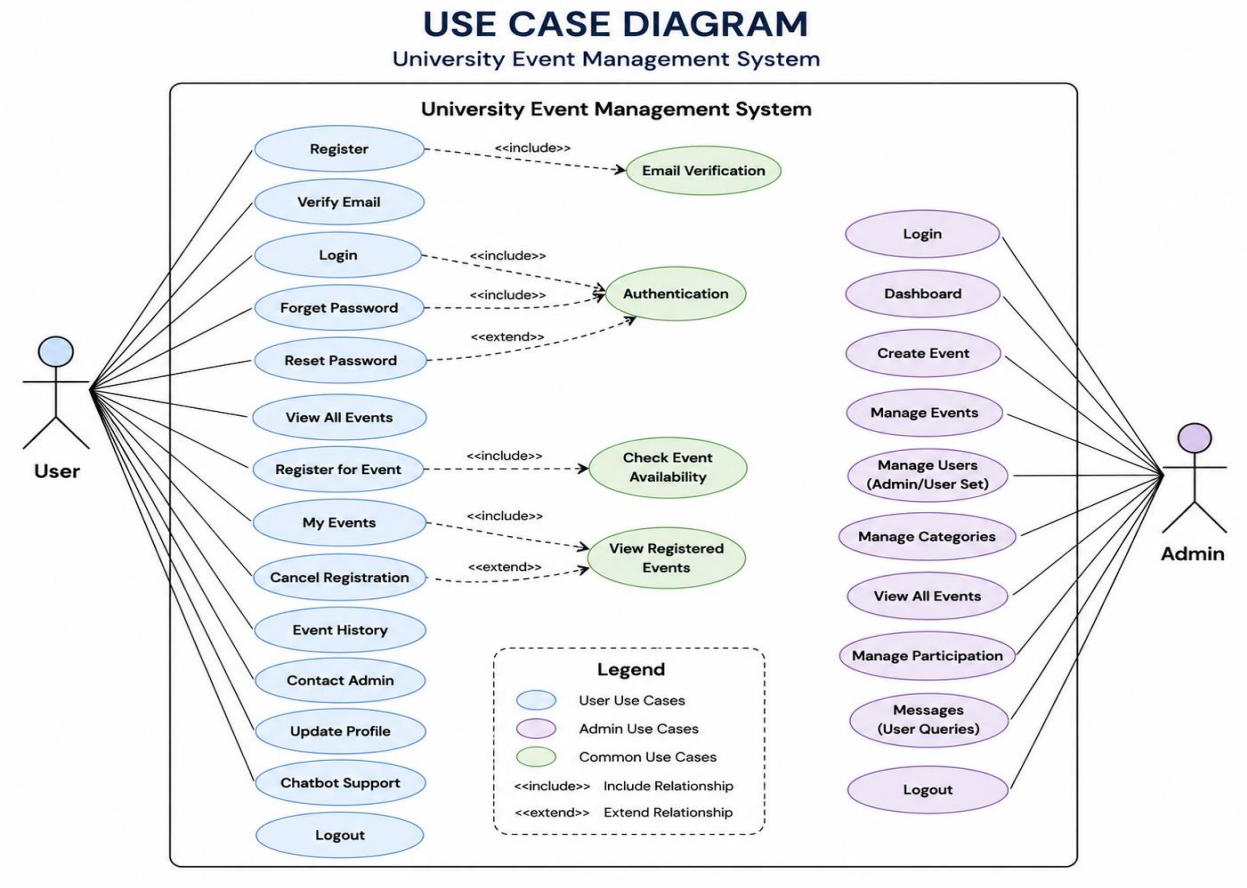


Figure 4.2: Use Case Diagram - University Event Management System

Breaking Down the Diagram

User Side Actions:

The regular user starts with registration. But registration alone is not enough, because the system needs to verify their email first before they can actually log in. So Verify Email is included as part of the registration process. Once verified, the user can log in. If someone forgets their password, the Forget Password flow lets them request a reset link, and then Reset Password lets them set a new one using that link. After logging in, the user lands on the home page where they can view all upcoming events. If something catches their eye, they can register for that event. Registering for an event automatically adds it to their My Events list, which is why My Events is connected to the registration action. From the My Events page, if the user changes their mind or has a schedule

conflict, they can cancel their registration, this is an extended relationship because cancellation is optional and only applies when someone actually wants to drop out. Beyond events, users can check their event history to see past events they attended, contact the admin if they run into problems or have questions, update their profile picture anytime, and use the chatbot for quick answers. And of course, logout when they are done.

Admin Side Actions:

The admin also starts by logging in. Once inside, they see a dashboard that gives them an overview of system activity. From there, the admin can create new events, manage existing ones like editing details or deleting cancelled events, manage user accounts and set roles to admin or regular user, organize event categories so events stay structured, view all events at a glance, manage participation records to see who is registered for what, and handle messages from users by reading queries and sending replies. The admin can also log out.

Understanding the Relationships:

Some actions have an include relationship, meaning one action cannot complete without the other. For example, to register an account, the email must be verified, so Verify Email is included in Register. Similarly, Login includes authentication checks behind the scenes. And Register for Event includes adding the event to the user's My Events list. The extended relationship means an action is optional and only happens under certain conditions. Cancel Registration extends My Events because cancellation only makes sense after someone has already registered for something. The user and admin both share a few common actions like Login, View All Events, and Logout, which is why those sit in the overlapping area between the two.

4.3 Activity Diagram

Activity diagram is basically a flowchart that shows the step by step flow of how things work in the system. It helps to understand what happens from the moment a user or admin starts an action until they complete it. For our system, I created two main flows, one for the user side and one for the admin side.

User Side Flow

The user journey starts when someone visits the system. First step is registration. They fill in their details and submit the form. But they cannot just start using the account right away. The system sends a verification email, and they have to click the link to verify. Once verified, they move to the login page. At login, the system checks if the credentials are valid. If the email or password is wrong, they get an error and have to try again. If everything matches, they land on the dashboard, which in our case is the home page showing all events. From the home page, the user can browse events and click on any event to see its full details, like date, venue, description, and how many spots are left. If they like the event, they can try to register. But before confirming, the system runs a quick check to see if this person is already registered for that same event. If yes, they get a message saying they are already signed up. If not, the registration gets confirmed and the event is

added to their My Events list. After using the system, the user can simply logout and the session ends.

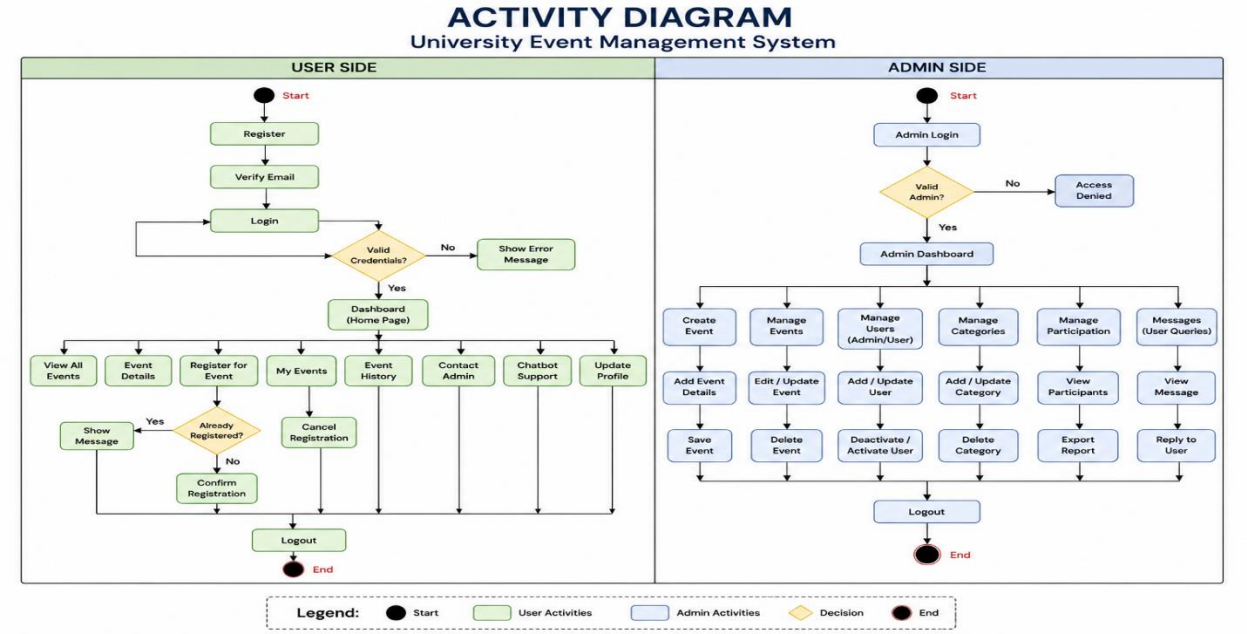


Figure 4.3: Activity Diagram - User Side and Admin Side Workflows

Admin Side Flow

The admin flow is a bit different. Admin also starts with login, but the system checks not just the password, but also the role. If the person logging in has the admin role, they get access. If they are a regular user trying to access the admin panel, they get an access denied message. Once inside the admin dashboard, the admin has several options. They can create a new event by entering all the event details like name, date, venue, category, max participants, and then saving it. They can manage existing events, which means editing details of an event that might have changed, or deleting events that got cancelled entirely. User management is another branch. The admin can add new users manually if needed, update existing user information, or change someone's status, like deactivating a user who violated rules or activating a previously deactivated account. Categories are handled separately. The admin can add new categories or update existing ones to keep events organized. They can also view participation records to see who registered for which event, and even export reports if they need a list or summary for meetings or documentation. And the most interactive part, the admin can view messages sent by users through the contact page and reply to them directly from the dashboard. When the admin is done with their work, they logout and the session closes.

4.4 Sequence Diagram

A sequence diagram shows the step-by-step communication between different parts of the system. It captures who talks to whom and in what order.

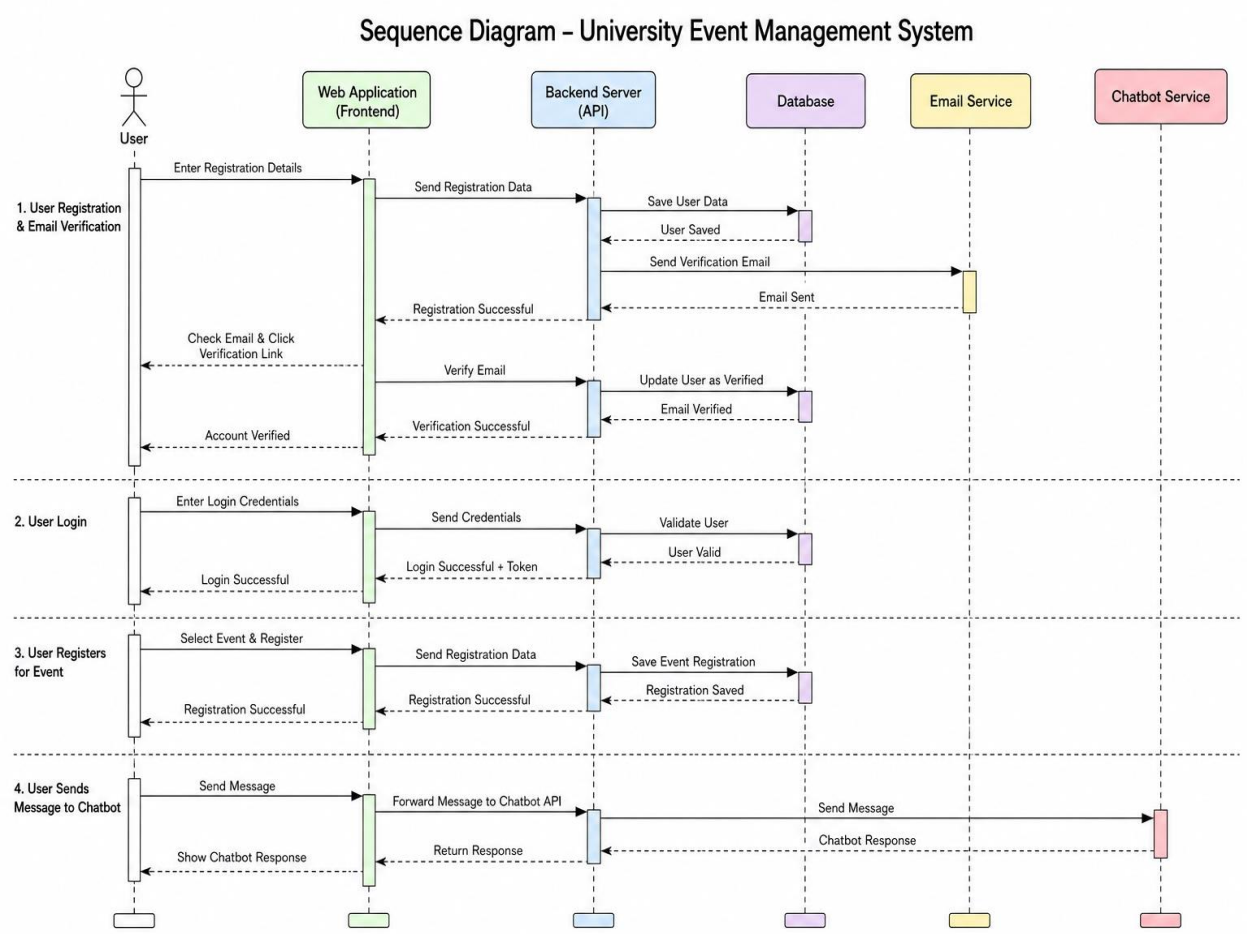


Figure 4.4: Sequence Diagram - Registration, Verification, Login, and Chatbot Flow

The diagram follows a user through four main flows. First, registration. The user fills in details on the frontend, the data goes to the backend server, the server saves it to the database and triggers the email service to send a verification link. Second, email verification. The user clicks the link, the frontend tells the backend, the backend updates the database to mark the user as verified. Third, login. The user enters credentials, the server checks them against the database, and if valid, login succeeds. Fourth, chatbot interaction. The user types a question, the frontend sends it to the backend, the backend forwards it to the chatbot service, and the response flows back up to the user. Each arrow in the diagram represents one message being passed between two components, and the vertical order shows the sequence of events from top to bottom.

4.5 Class Diagram

A class diagram shows the structure of the system. It maps out all the main classes, their attributes, their methods, and how they relate to each other.

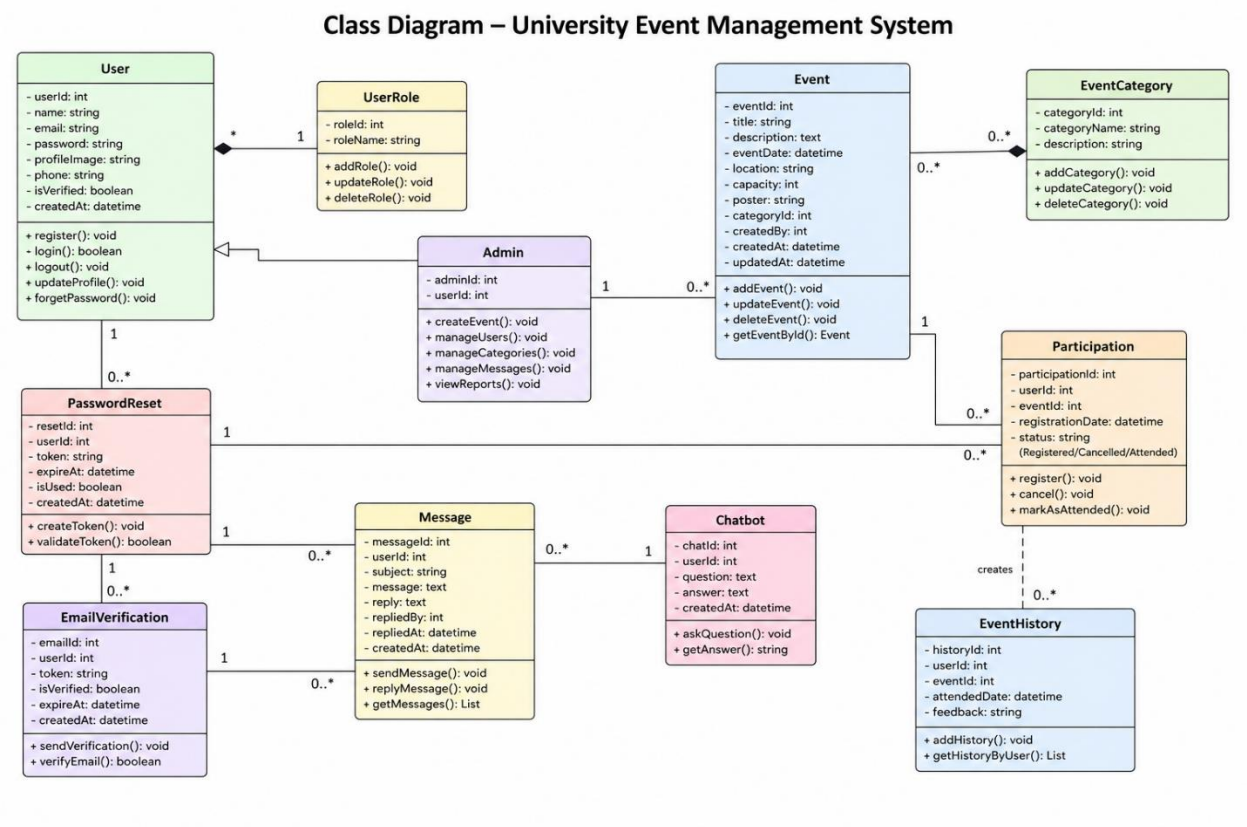


Figure 4.5: Class Diagram - University Event Management System

The central class is User. Every user has basic details like name, email, password, profile image, phone number, and a verification status. The user can register, login, logout, update their profile, and trigger a forgot password flow.

Connected to User are several supporting classes. Password Reset stores reset tokens with an expiry time, so when someone forgets their password, a unique token is generated and validated before they can set a new one. Email Verification works similarly, it holds the verification token sent during registration.

- The Admin class extends from User and adds the ability to view reports. Admins can also manage categories, events, and users through their respective classes.
- The event is another core class. Each event has a title, description, date, location, capacity, a poster image, and belongs to a category. Events are created by an admin.
- Category groups events together. Each category has a name and description, and admins can add, update, or delete categories.
- Participation links users to events. When a user registers for an event, a participation record is created with a status showing whether it is registered, cancelled, or attended.

- Event History keeps track of events the user actually attended, along with the date and any feedback they left.
- Message handles the contact page. Users send messages with a subject and body, and admins can reply. The reply and who replied are stored.
- Chatbot stores user questions and the system's answers. Each chat record links to a user and timestamps when it was created.

The relationships are straightforward. One user can have many participation records, many messages, many chatbot interactions, and many event history entries. One event can have many participants. One category can contain many events. These one to many relationships are marked by the "1" on one end and "0..*" on the other hand, meaning one record can link to zero or many records on the other side.

4.6 Entity-Relationship (ER) Diagram

The ER diagram shows how database tables are structured and connected. It focuses on entities, their attributes, and the relationships between them.
The diagram has five main entities. User stores account details with primary key id, plus fields like username, email, password, role, and tokens for verification and password reset. The event holds all event information including title, description, date and time, location, capacity, category, and organizer details. Category is a simple table with just id and name, used to group events by type. User Event is a junction table that connects users to events, tracking which user registered for which event. Contact Message stores messages sent by users through the contact page, along with reply information and read status.

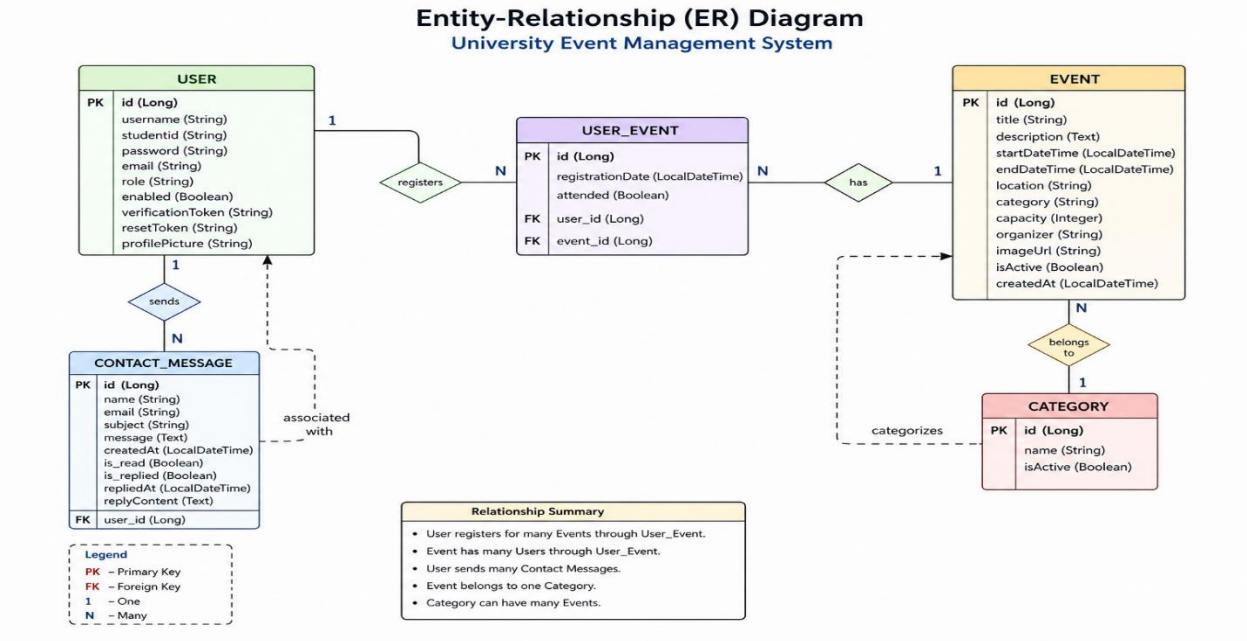


Figure 4.6: Entity-Relationship Diagram - University Event Management System

The relationships are straightforward. A user can register for many events, and an event can have many users registered, so this is a many-to-many relationship handled through the user Event junction table. A user can send many contact messages, so that is a one-to-many relationship.

4.7 Data Flow Diagram (DFD)

4.7.1 Level 0 DFD (Context Diagram)

The Level 0 DFD, also called the context diagram, gives the highest level view of the system. It shows the entire system as a single process and maps out how data flows between the system and the outside world.

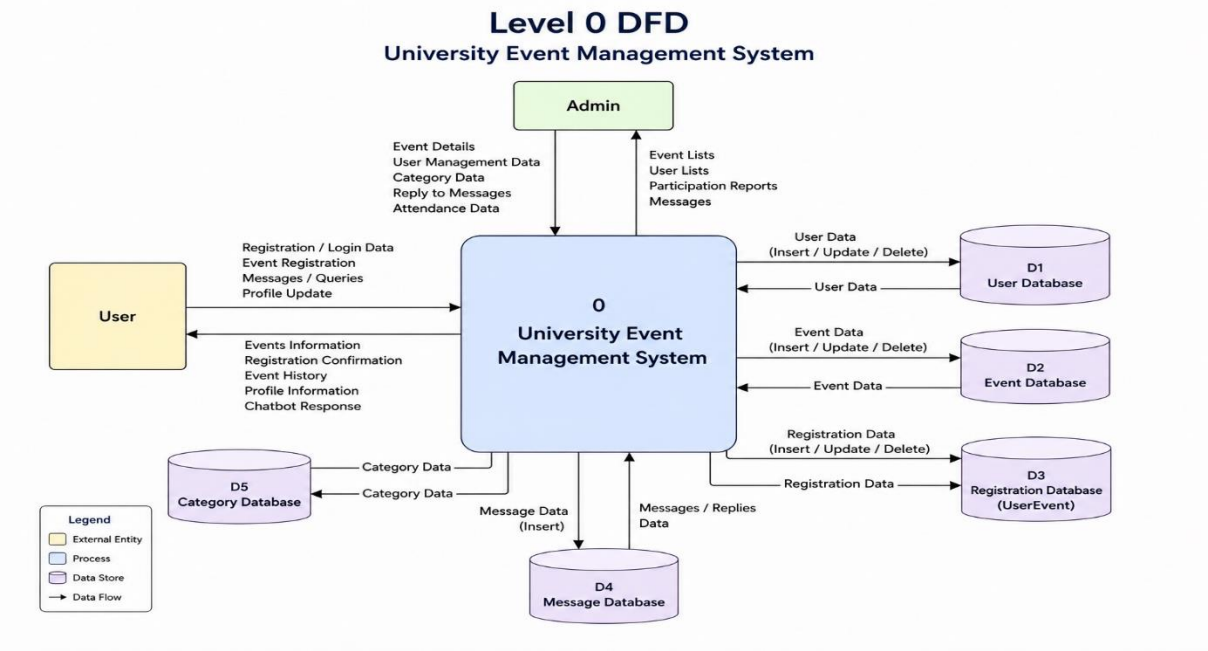


Figure 4.7: Level 0 DFD - Context Diagram

Here the whole University Event Management System sits in the middle as one process. Around it are two external entities, the User and the Admin. These are the people interacting with the system.
From the User side, data flows into the system when they register, log in, register for events, send messages, or update their profile. Data flows back out when the system sends them event information, registration confirmations, their event history, profile details, and chatbot responses.

From the Admin side, data enters the system when the admin creates events, manages users, sets up categories, replies to messages, or updates attendance. The system sends back event lists, user lists, participation reports, and messages that need replies. Inside the system, four databases store all the data. The User Database holds account information, the Event Database stores event details, the Registration Database tracks who registered for what, and the Message Database keeps user queries and admin replies.

4.7.2 Level 1 DFD

The Level 1 DFD breaks open that single process from the context diagram and shows what is actually happening inside the system. It reveals the main processes and how data moves between them.

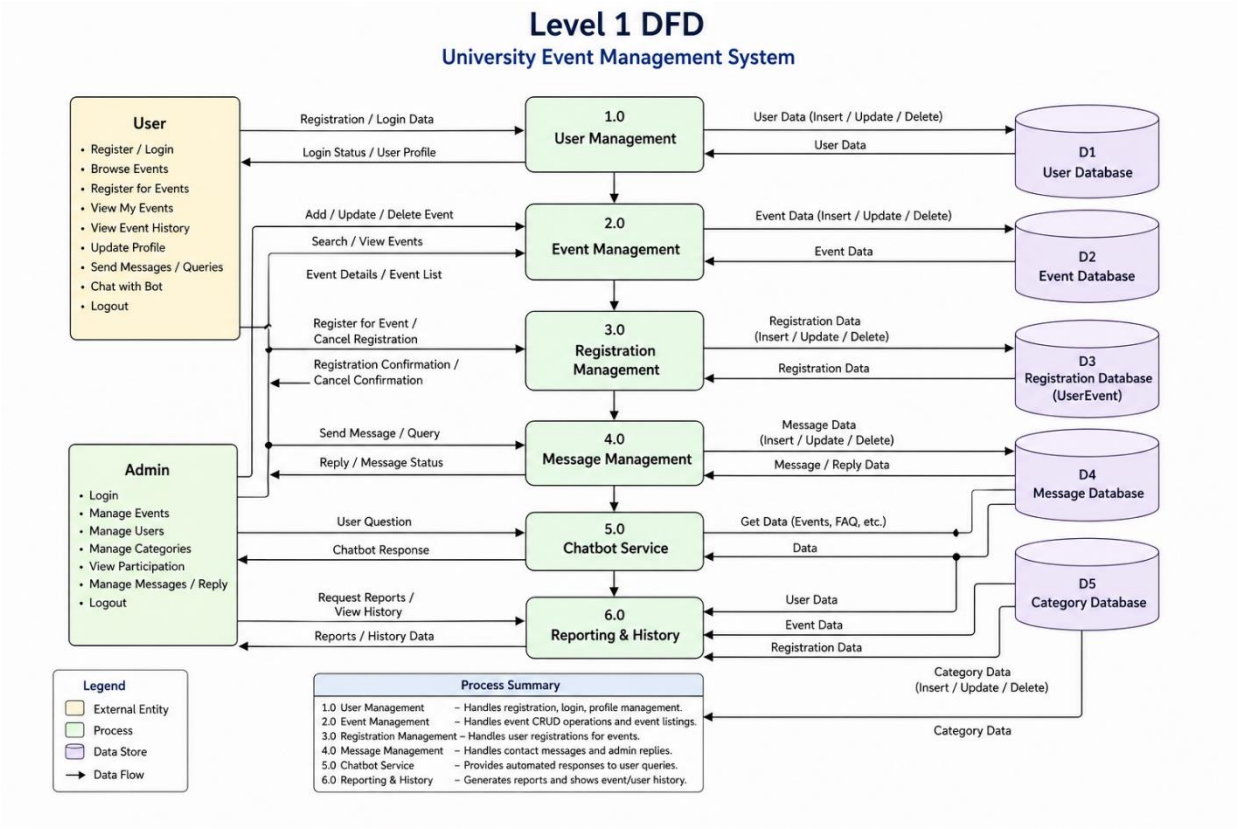


Figure 4.8: Level 1 DFD - System Processes and Data Flow

The system has six core processes. Process 1.0 is User Management, handling registration, login, and profile updates. It reads and writes to the User Database. Process 2.0 is Event Management, where admins create and edit events and users browse and search them. It connects to the Event Database. Process 3.0 is Registration Management, handling event signups and cancellations. It stores data in the Registration Database. Process 4.0 is Message Management, dealing with user queries sent through the contact page and admin replies. It uses the Message Database. Process 5.0 is the Chatbot Service, taking user questions and returning automated responses based on event data and FAQs. Process 6.0 handles Reporting and History, generating participation reports for admins and showing event history to users. Data flows between these processes and the databases just like in the level 0 diagram, but now you can see exactly which process talks to which database. For example, when a user registers for an event, the request goes through Process 3.0 which writes to the Registration Database and may also read from the Event Database to check capacity. When an admin replies to a message, Process 4.0 updates the Message Database and the reply flows back to the user. This level gives a much clearer picture of the internal workings without getting lost in too much detail.

4.8 Database Design

4.8.1 Table Structures

The database is the backbone of the whole system. I used MySQL and designed the tables to be normalized so data is not repeated unnecessarily and relationships stay clean. Here are the main tables and what each column does.

Users Table:
This is the most important table. Every person using the system has a record here. The id is the primary key and auto increments. Username and student ID are required. Email is unique so no two accounts can use the same email. Passwords are stored after BCrypt encryption, never as plain text. The role column determines if the person is an admin or a regular user. The enabled flag controls whether the account is active. verification_token and reset_token are for email verification and password reset flows. profile_picture stores the file path to the uploaded image.

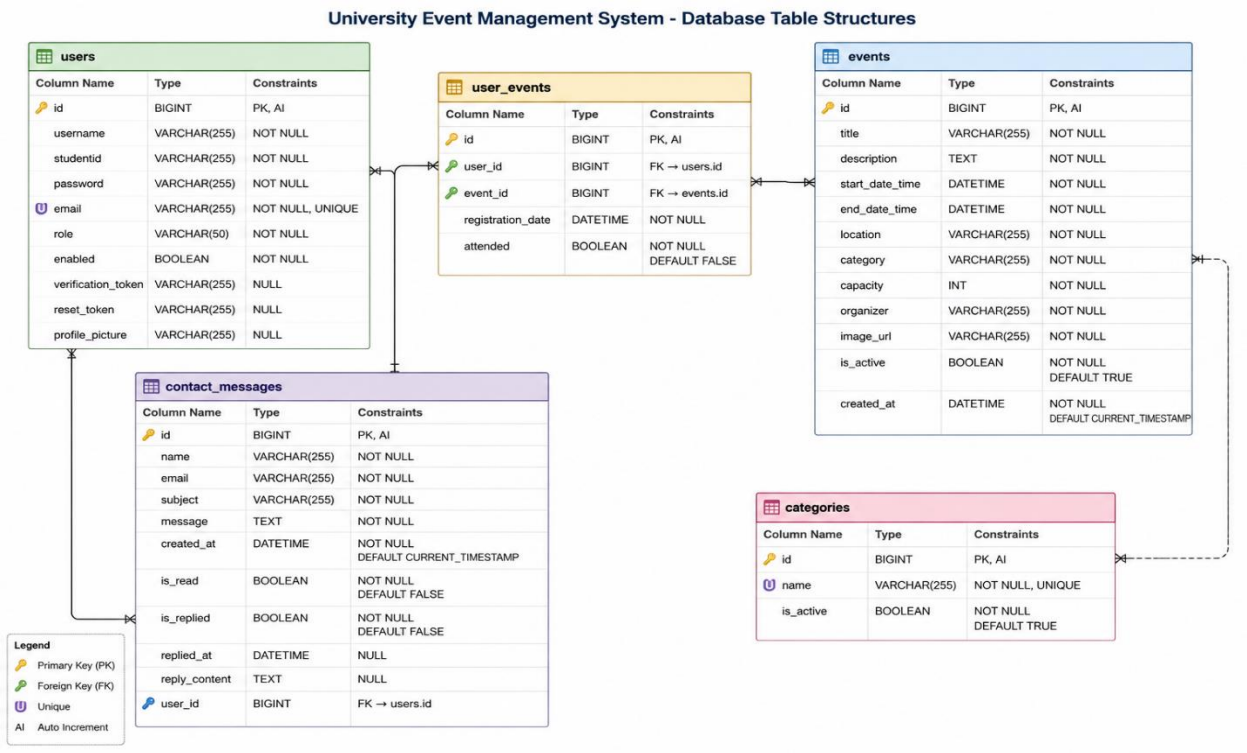


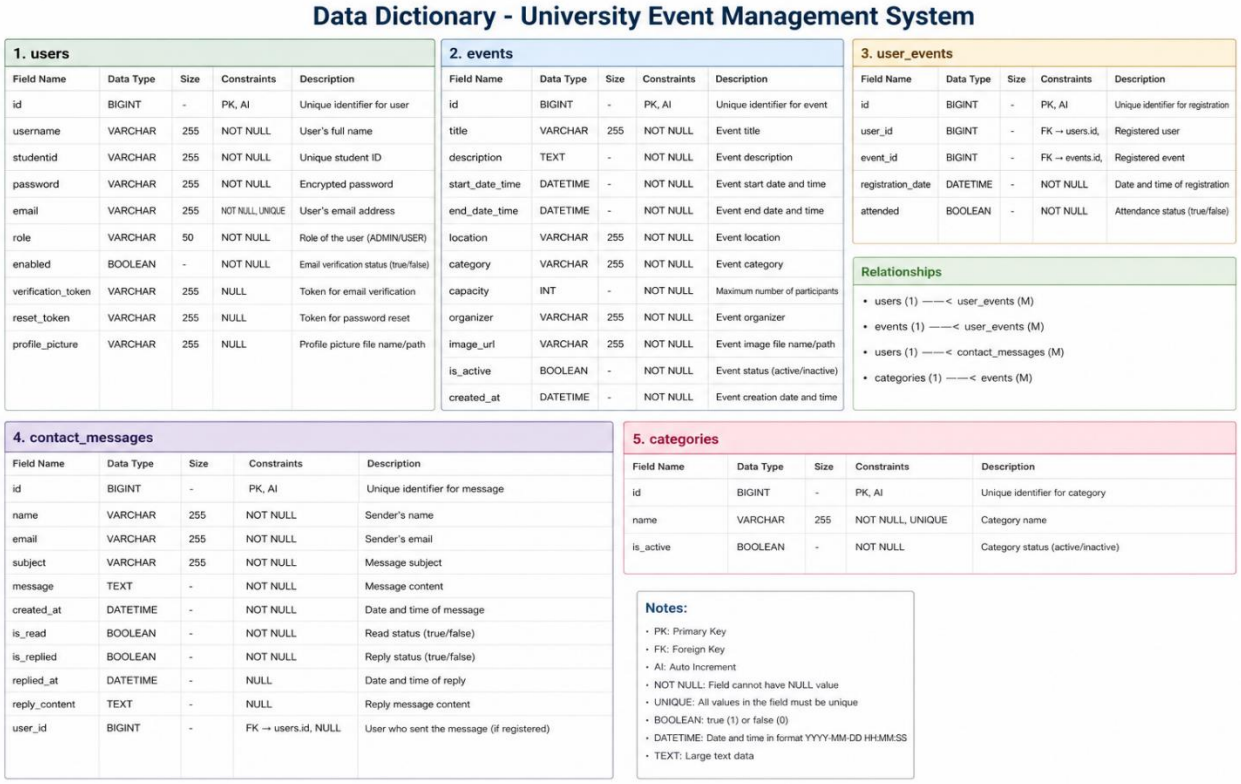
Figure 4.9: Database Table Structures

Events Table:
This holds every event created by admins. Each event has a title, description, start and end date-time, location, and capacity for how many people can register. The category_id links to the categories table so events are grouped properly. organizer stores who is running the event. image_url holds the path to the event poster. is_active lets admins hide or show events without deleting them. created_at timestamps when the event was made.

User_Events Table:
This is the junction table connecting users to events. When someone registers for an event, a row is created here. user_id points to the user, event_id points to the event. registration_date records when they signed up. The attended flag marks whether they actually showed up, which admins can update after the event ends.
Categories Table:
A simple lookup table. Each category has an id, a unique name, and an is_active flag. Admins can add categories like Academic, Cultural, Sports, Tech, Career, and Club.
Contact_Messages Table:
When a user sends a message through the contact page, it lands here. name, email, subject, and message store the actual content. is_read tells the admin if this message has been seen. is_replied tracks whether the admin has responded. The reply goes into a reply column (implied in the design).
All tables use BIGINT for IDs to handle large numbers of records. Primary keys uniquely identify each row. Foreign keys link tables together, like user_id in user_events referencing id in users, so the database maintains referential integrity. If you try to insert a registration for a user that does not exist, MySQL will reject it.

4.8.2 Data Dictionary

The data dictionary explains every column in every table, its data type, constraints, and what it actually stores.



Data Dictionary - University Event Management System

1. users

Field Name	Data Type	Size	Constraints	Description
id	BIGINT	-	PK, AI	Unique identifier for user
username	VARCHAR	255	NOT NULL	User's full name
studentid	VARCHAR	255	NOT NULL	Unique student ID
password	VARCHAR	255	NOT NULL	Encrypted password
email	VARCHAR	255	NOT NULL, UNIQUE	User's email address
role	VARCHAR	50	NOT NULL	Role of the user (ADMIN/USER)
enabled	BOOLEAN	-	NOT NULL	Email verification status (true/false)
verification_token	VARCHAR	255	NULL	Token for email verification
reset_token	VARCHAR	255	NULL	Token for password reset
profile_picture	VARCHAR	255	NULL	Profile picture file name/path

2. events

Field Name	Data Type	Size	Constraints	Description
id	BIGINT	-	PK, AI	Unique identifier for event
title	VARCHAR	255	NOT NULL	Event title
description	TEXT	-	NOT NULL	Event description
start_date_time	DATETIME	-	NOT NULL	Event start date and time
end_date_time	DATETIME	-	NOT NULL	Event end date and time
location	VARCHAR	255	NOT NULL	Event location
category	VARCHAR	255	NOT NULL	Event category
capacity	INT	-	NOT NULL	Maximum number of participants
organizer	VARCHAR	255	NOT NULL	Event organizer
image_url	VARCHAR	255	NOT NULL	Event image file name/path
is_active	BOOLEAN	-	NOT NULL	Event status (active/inactive)
created_at	DATETIME	-	NOT NULL	Event creation date and time

3. user_events

Field Name	Data Type	Size	Constraints	Description
id	BIGINT	-	PK, AI	Unique identifier for registration
user_id	BIGINT	-	FK → users.id,	Registered user
event_id	BIGINT	-	FK → events.id,	Registered event
registration_date	DATETIME	-	NOT NULL	Date and time of registration
attended	BOOLEAN	-	NOT NULL	Attendance status (true/false)

Relationships

- users (1) ——< user_events (M)
- events (1) ——< user_events (M)
- users (1) ——< contact_messages (M)
- categories (1) ——< events (M)

4. contact_messages

Field Name	Data Type	Size	Constraints	Description
id	BIGINT	-	PK, AI	Unique identifier for message
name	VARCHAR	255	NOT NULL	Sender's name
email	VARCHAR	255	NOT NULL	Sender's email
subject	VARCHAR	255	NOT NULL	Message subject
message	TEXT	-	NOT NULL	Message content
created_at	DATETIME	-	NOT NULL	Date and time of message
is_read	BOOLEAN	-	NOT NULL	Read status (true/false)
is_replied	BOOLEAN	-	NOT NULL	Reply status (true/false)
replied_at	DATETIME	-	NULL	Date and time of reply
reply_content	TEXT	-	NULL	Reply message content
user_id	BIGINT	-	FK → users.id, NULL	User who sent the message (if registered)

5. categories

Field Name	Data Type	Size	Constraints	Description
id	BIGINT	-	PK, AI	Unique identifier for category
name	VARCHAR	255	NOT NULL, UNIQUE	Category name
is_active	BOOLEAN	-	NOT NULL	Category status (active/inactive)

Notes:

- PK: Primary Key
- FK: Foreign Key
- AI: Auto Increment
- NOT NULL: Field cannot have NULL value
- UNIQUE: All values in the field must be unique
- BOOLEAN: true (1) or false (0)
- DATETIME: Date and time in format YYYY-MM-DD HH:MM:SS
- TEXT: Large text data

Figure 4.10: Data Dictionary

- For the users table, id is the primary key with auto increment. Email has a unique constraint so duplicates are rejected. Passwords are stored encrypted. Roles can be ADMIN or USER. The enabled flag stays false until email verification. Tokens for verification and password reset are nullable since they only exist temporarily.
- For the events table, title, description, date-time, location, and capacity are all required. Category groups events by type. is_active lets admins hide events without deleting them.
- The user_events table links users to events through foreign keys. user_id references users.id and event_id references events.id. The attended boolean tracks whether someone actually showed up.
- For contact_messages, name, email, subject, and message store the query. is_read and is_replied help the admin track which messages need attention. reply_content holds the admin's response.
- The categories table is simple, just an id, a unique name, and an is_active flag.

Every field's data type was chosen based on what made sense. VARCHAR for short text, TEXT for longer content, DATETIME for timestamps, BOOLEAN for flags, and BIGINT for IDs. Constraints like NOT NULL, UNIQUE, and foreign keys keep the data clean and consistent.

4.9 System Workflow

The system workflow shows the complete journey of both users and admins through the system, step by step, from account creation to logging out.

Common Flow for Everyone:

Every person starts the same way. First they create an account by registering. Then they verify their email through a link sent to their inbox. After verification, they can log in. The system checks their role and directs them to either the user panel or admin panel. When done, they log out.

User Workflow:

After login, users land on the home page showing all upcoming events. They can search and filter by category. Clicking an event shows full details. If they want to attend, they click register and the system checks if spots are available before confirming. Registered events appear under My Events, where users can also cancel if plans change. Past attended events show up in Event History. If users have issues, they use the contact page to send a message to the admin. Profile management lets them update their picture and info. The chatbot is available everywhere for instant answers about events.

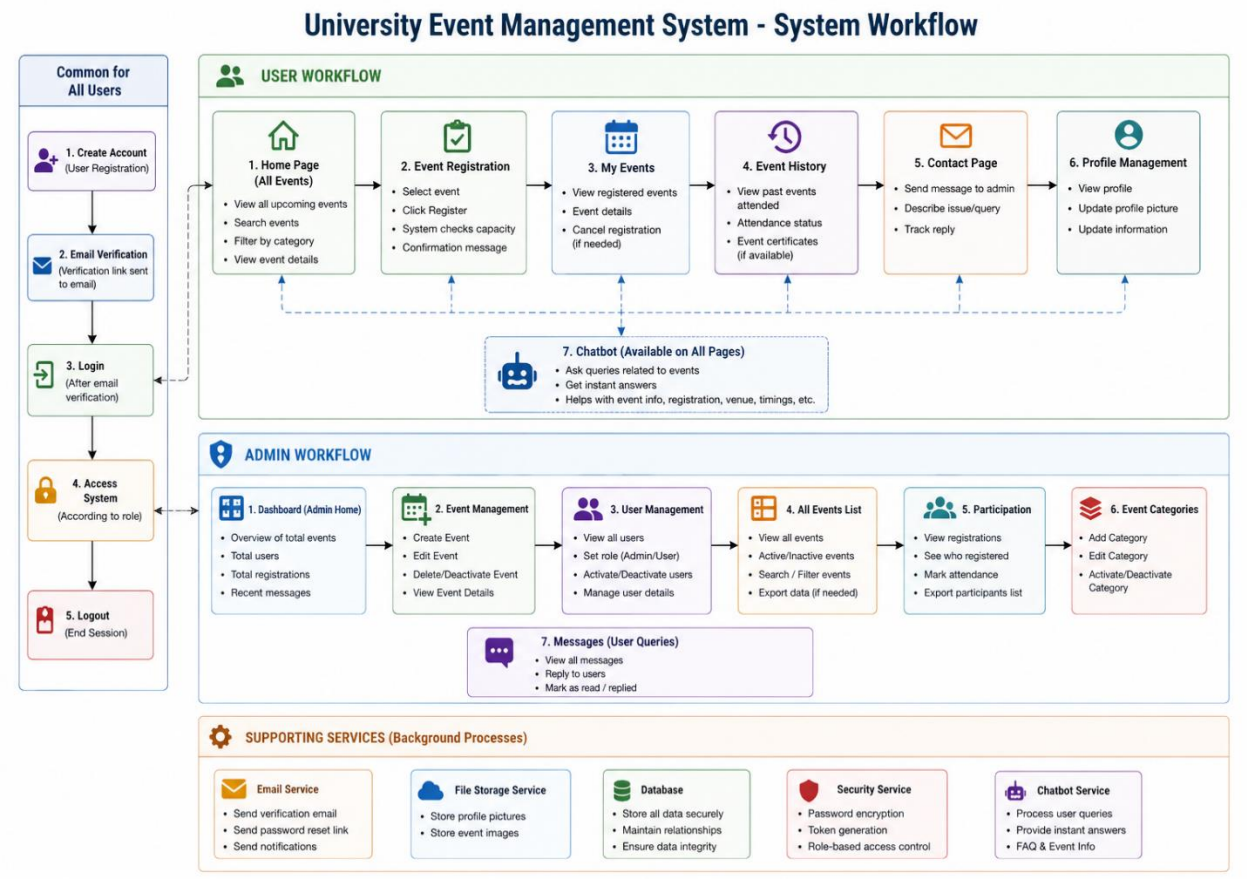


Figure 4.11: System Workflow - User and Admin Processes

Admin Workflow:

Admins start at the dashboard showing quick stats like total events, users, registrations, and new messages. From there, they can create, edit, or deactivate events. User management lets them view all users, assign roles, and activate or deactivate accounts. The all events list shows everything with search and filter options. Participation management shows who registered for which event and allows marking attendance. Categories can be added, edited, or deactivated. The messages section shows all user queries with options to reply.

Background Services:

Behind the scenes, the email service handles verification links and password resets. File storage saves profile pictures and event images. The database keeps everything stored and maintains data integrity. Security services handle password encryption and role-based access. The chatbot service processes questions and returns answers in real time.

CHAPTER 5

SYSTEM DEVELOPMENT & IMPLEMENTATION

5.1 Development Methodology

I chose Agile methodology for this project. The reason is simple, requirements change, bugs pop up, and you discover better ways of doing things as you build. Agile lets you adapt [7]. I broke the whole development into three sprints, each lasting about two weeks. Sprint one was all about setting up the foundation, database design, user registration, login, email verification, and forgot password. Once that was solid, sprint two focused on the admin panel, event creation, user management, categories, and the user home page. Sprint three covered the remaining features like My Events, event history, chatbot, contact page, and profile management. Every day I would check what was working and what was not. At the end of each sprint, I tested everything built so far. If something needed fixing, I fixed it before moving on. This iterative approach saved me from the nightmare of finding a hundred bugs right before the deadline [7].

5.2 Technology Stack

5.2.1 Frontend Technologies

For the frontend, I kept it straightforward. HTML5 for page structure. CSS3 with Bootstrap 5 for styling and making everything responsive, because students will open this on their phones more often than on desktops. JavaScript handles all the interactivity, form validations before submission, AJAX calls so pages do not have to reload every time, and the chatbot interface. Thymeleaf is the template engine that connects the frontend to the backend, letting me write dynamic content inside HTML without mixing too much Java code into the views.

5.2.2 Backend Technologies

The backend runs on Spring Boot. I picked this because it handles a lot of configuration automatically and lets you focus on writing actual logic. Spring MVC manages all the web requests. Spring Security handles authentication and authorization. Spring Data JPA makes database operations simple, instead of writing raw SQL for basic CRUD, you just define repository interfaces. Spring Mail connects to the email service for sending verification and reset emails. I also used Lombok to reduce boilerplate code like getters, setters, and constructors.

5.2.3 Database

MySQL is the database. It is reliable, free, and I have used it before so I knew what to expect. All the tables are properly normalized with primary keys, foreign keys, and constraints to keep data consistent.

5.2.4 Other Tools and APIs Used

For development, I used IntelliJ IDEA as the IDE. Maven handles dependency management. Postman was a lifesaver for testing APIs before the frontend was ready. Git helped with version control so I could experiment without fear of breaking everything. BCrypt encodes passwords before they hit the database. JavaMail API connects to Gmail SMTP for sending emails. Jackson converts Java objects to JSON and back, which is useful for AJAX responses. The chatbot runs on a custom keyword matching approach rather than an external API, which I will explain later.

5.3 Module Implementation

5.3.1 User Authentication Module

This module covers everything related to getting users into the system securely.

Registration: The user fills a form with their name, email, student ID, phone number, department, and password. The frontend validates that all required fields are filled and the email format is correct. When the form is submitted, the controller receives the data, the service layer checks if the email already exists. If it does, an error is shown. If not, the password gets encrypted with BCrypt and the user is saved to the database with enabled set to false. A verification token is generated using UUID and stored in the user record.

Email Verification: Right after saving the user, the email service sends a verification link containing the token to the user's email. The user clicks the link, which hits a controller endpoint. The service looks up the user by that token. If found, enabled is set to true and the token is cleared. The user can now log in. If the token is invalid or expired, an error message appears.

Login: The user enters email and password. Spring Security intercepts the request, loads the user from the database, checks if enabled is true, and matches the password against the stored BCrypt hash. If everything passes, a session is created and the user is redirected based on their role. If not, they get an error message.

Forgot Password: The user enters their email on the forgot password page. The service checks if the email exists. If it does, a reset token is generated and emailed. The user clicks the link, which takes them to a reset password page. They enter a new password, the token is validated, and if valid, the password is updated and the token cleared.

5.3.2 Admin Panel Implementation

Event Creation and Management

The admin fills an event creation form with title, description, start and end date-time, venue, capacity, category selection, and an image upload for the poster. The controller receives this, the service validates the data, and saves the event. The current participants count starts at zero. The admin can later edit any event details or deactivate an event that gets cancelled. Deactivating keeps the record but hides it from users.

User Management

The admin sees a table of all registered users. They can search by name or email. Each row shows user details and their current role. The admin can toggle someone's role between ADMIN and USER. They can also enable or disable accounts. If a student violates rules, their account can be deactivated without deleting their data.

Event Categories

The admin can add new categories like Workshop, Seminar, Sports, Cultural Fest, and so on. They can edit category names and activate or deactivate them. When a category is deactivated, existing events in that category remain, but new events cannot be assigned to it.

Participation Management

For each event, the admin can view the list of registered users with their names, emails, registration dates, and an attendance checkbox. After the event ends, the admin can mark which users actually attended. This data feeds into the user's event history.

Message and Query Management

All messages sent through the contact page appear in the admin panel. Unread messages are highlighted. The admin clicks on a message to view its full content, then types a reply and submits it. The reply gets stored in the database, the message status changes to answered, and the reply time is recorded.

5.3.3 User Panel Implementation

Homepage and Event Display

The home page shows all upcoming events in a grid of cards. Each card has the event poster image, title, date, venue, and a count showing how many spots are filled out of capacity. Users can search events by keyword or filter by category using a dropdown. Clicking a card takes them to the full event details page.

My Events and Registration

When a user views event details and clicks register, the controller calls the service layer. The service checks if the user is already registered, if the event is full, and if the event date has passed.

If all checks pass, a participation record is created and the event's current participants count increases. The user sees a confirmation message. On the My Events page, they see all their registered upcoming events with a cancel button next to each one. Canceling removes the participation record and decrements the participant count.

Event History
This page shows events where the user was marked as attended. It lists the event name, date, venue, and any certificate or feedback options if available. This gives users a record of their campus involvement.

Contact Page
A simple form with subject and message fields. The user's name and email are prefilled from their account. They write their issue and submit. The message gets saved to the contact_messages table with is_read and is_replied set to false. The user can see their sent messages and whether they have been answered.

Profile Picture Update
The user sees their current profile picture or a default placeholder. They can click to upload a new image. The file is saved to a server directory, and the file path is updated in the users table. The new picture appears across the system.

Chatbot Integration
A chatbot icon sits at the bottom corner of every page. Clicking it opens a chat window. The user types a question. JavaScript sends it to a chatbot controller via AJAX. The service processes the query by matching keywords against a predefined set of questions and answers covering event schedules, registration steps, venue locations, and FAQs. If a match is found, the answer is returned instantly. If not, the chatbot suggests contacting the admin or rephrasing the question.

5.4 Chatbot Working Process

I built the chatbot as a rule-based system rather than using external AI APIs. The main reason was to keep things simple and not depend on internet connectivity for core functionality.
The chatbot has a dataset of keyword-answer pairs stored in a Java map. When a user sends a message, the service tokenizes the text, removes common words like "is", "the", "what", and extracts meaningful keywords. It then scans the dataset for matches. For example, if someone types "when is the next workshop", the keywords "next" and "workshop" trigger a query to find the nearest upcoming event in the workshop category, and the chatbot replies with the event name and date. Questions like "how to register" map to a step-by-step guide. "Where is the venue" maps to location info.

If no keywords match, the chatbot gives a fallback response asking the user to try different words or use the contact page. This approach handles about eighty percent of common queries without needing any machine learning or external APIs.

5.5 Security Implementation

Security was something I took seriously from the start. A system that handles user data, emails, and passwords cannot afford to be careless. Here is how I handled the main security concerns [4][6].

JWT Authentication

When a user logs in successfully, the server generates a JSON Web Token. This token contains the user's ID and role, signed with a secret key so nobody can tamper with it. The token is sent back to the client and stored in the browser's local storage. From that point on, every request the user makes includes this token in the header [4]. On the server side, a filter intercepts every incoming request. It extracts the token, verifies the signature, and if valid, sets the user's authentication in the security context. If the token is missing, expired, or tampered with, the request is rejected. This means the server does not need to keep session data, each request is self-contained. It also makes the system ready for mobile apps later since JWT works the same way regardless of the client platform [12]. I set the tokens to expire after a reasonable time, so if someone forgets to log out on a public computer, their session does not stay open forever.

Password Hashing with BCrypt

Storing passwords as plain text is a disaster waiting to happen. I used BCrypt for password hashing. When a user registers or resets their password, BCrypt generates a salted hash. The salt is random, so even if two users pick the same password, their hashes look completely different [6]. When the user logs in, BCrypt takes the password they entered, combines it with the stored salt, and checks if the resulting hash matches. If it matches, they are in. If not, access is denied. The important part is that at no point is the actual password stored anywhere in readable form. Even if someone gets a copy of the database, all they see are hashed strings that are computationally expensive to crack.

Email Verification Token

When a new user registers, their account is created but marked as disabled. A verification token is generated using Java's UUID class, which creates a long random string that is practically impossible to guess. This token is stored in the user's record and sent to their email as part of a verification link [6]. When the user clicks the link, the system looks up the token. If it finds a matching user and the token has not expired, the account gets enabled and the token is cleared. If the token is invalid or already used, the user gets an error and needs to register again or request a new verification email. This prevents bots from creating fake accounts and ensures that the email address the user provided actually belongs to them. Without it, anyone could sign up with random emails and the system would be full of junk accounts.

Role-Based Authorization

Not every logged-in user should have access to everything. A regular student should not be able to access the admin panel and start deleting events or changing other people's roles [4]. I implemented role-based access control using Spring Security. Each user has a role stored in the database, either ROLE_USER or ROLE_ADMIN. When a request comes in, the security filter checks the user's role against the required role for that endpoint. URLs under /admin/ require the ADMIN role. If a regular user tries to access them, they get an access denied page. URLs under /user/ require at least the USER role, which also works for admins since they inherently have user-level access too. Public URLs like /auth/register, /auth/login, and /auth/forgot-password are open to everyone. On the frontend, I also hide admin links from regular users. But the real protection is on the backend, because frontend hiding can be bypassed by anyone who knows the URL. The backend enforcement ensures that even if someone types the admin URL directly, they cannot get in without the proper role [4]. Together, these four layers of security make the system reasonably robust. JWT handles session management cleanly. BCrypt keeps passwords safe. Email verification prevents fake accounts. And role-based authorization makes sure people can only do what they are allowed to do.

CHAPTER 6

TESTING

6.1 Testing Strategy

I followed a bottom-up testing approach. Start with the smallest pieces, test them individually, then test how they work together, then test the whole system, and finally let actual users try it out.
The plan was simple. Unit tests for individual methods. Integration tests for modules talking to each other. System testing for the complete application. User acceptance testing with real students. And performance testing to make sure the system does not crawl under load.

6.2 Unit Testing

Unit testing means testing individual methods in isolation. I used JUnit 5 for this.
For the service layer, I wrote tests for methods like user registration, password reset, event creation, and participation registration. Each test creates mock data, calls the method, and checks if the output matches expectations. For example, the registration service test checks that a valid user gets saved, an email gets sent, and a verification token gets generated. It also tests edge cases like registering with an email that already exists, which should throw an error.
For the controller layer, I used MockMvc to simulate HTTP requests without starting the actual server. This let me test that endpoints return the correct status codes and view names. A test for the login endpoint checks that valid credentials return a redirect to the dashboard, while invalid credentials return back to the login page with an error.
I also tested utility methods like the BCrypt password encoder, the JWT token generator and validator, and the email validator. These are small but critical pieces that need to work correctly every time.

6.3 Integration Testing

Integration testing checks how different modules work together. For example, when a user registers, the user service, email service, and database all need to coordinate properly. I tested the full registration flow. User submits data, service saves to database, email service sends verification email, and the verification link activates the account. Each step depends on the previous one working correctly. I also tested the event registration flow. The user selects an event, service checks capacity and existing registrations, creates a participation record, and updates the participant count. If any part fails, the whole operation should roll back, which Spring's transaction management handles. For the admin message reply flow, I tested that when an admin replies to a message, the status changes to answered, the reply content is saved, and the timestamp is recorded. The user side then correctly shows the reply. These tests use a test database with sample data so the real database stays untouched.

6.4 System Testing

System testing means testing the complete application from end to end, just like a real user would use it. I went through every feature in sequence. Register a new account, verify email, log in, browse events, register for an event, check My Events, cancel a registration, check event history, send a contact message, update profile picture, use the chatbot, and log out. Then I switched to the admin account and tested everything there. Login, view dashboard, create an event with an image upload, edit the event details, manage user roles, add a category, view participation lists, mark attendance, read messages, reply to messages, and log out. I also tested edge cases. What happens if someone tries to register for a full event? What if they try to register twice? What if an expired reset token is used? What if an unverified user tries to log in? Each of these should show a clear error message, not crash or behave unpredictably. I tested on different browsers, Chrome, Firefox, and Edge, and on different screen sizes including mobile view. Bootstrap handles responsiveness well, but I still checked that nothing was broken on smaller screens.

6.5 User Acceptance Testing

I asked five fellow students to use the system and give feedback. They each created accounts, browsed events, registered for a few, used the chatbot, and sent messages through the contact page. Their feedback was mostly positive. They liked the simple layout and found it easy to navigate. A couple of them suggested that the search bar should be more prominent on the home page, so I made it larger and centered. One person mentioned that after registration, it was not immediately clear they needed to check their email, so I added a more visible message on the screen telling them to verify. This kind of testing catches things developers miss because we are too close to the code. Real users interact with the system differently and notice things we overlook.

6.6 Bug Tracking and Resolution

I kept a simple list of bugs I found during testing and how I fixed them. One bug was that after canceling a registration, the participant count on the event card did not update until the page was refreshed. I fixed this by adding an AJAX call that updates the count dynamically without a full page reload. Another bug was that the chatbot sometimes matched partial words incorrectly. For example, typing "event" and "evening" both triggered the same response. I fixed this by improving the keyword matching logic to use word boundaries instead of simple string contains. There was also an issue where profile pictures with the same filename would overwrite each other. I fixed this by appending a timestamp to each uploaded filename, making them unique. The forgotten password token had no expiry, so old links would still work indefinitely. I added an expiry time of one hour to the token and a check that rejects expired tokens.

CHAPTER 7

RESULTS & DISCUSSION

This chapter presents what we actually built. Screenshots of the working system, analysis of how it performs, feedback from real users, and an honest assessment of whether we achieved what we set out to do.

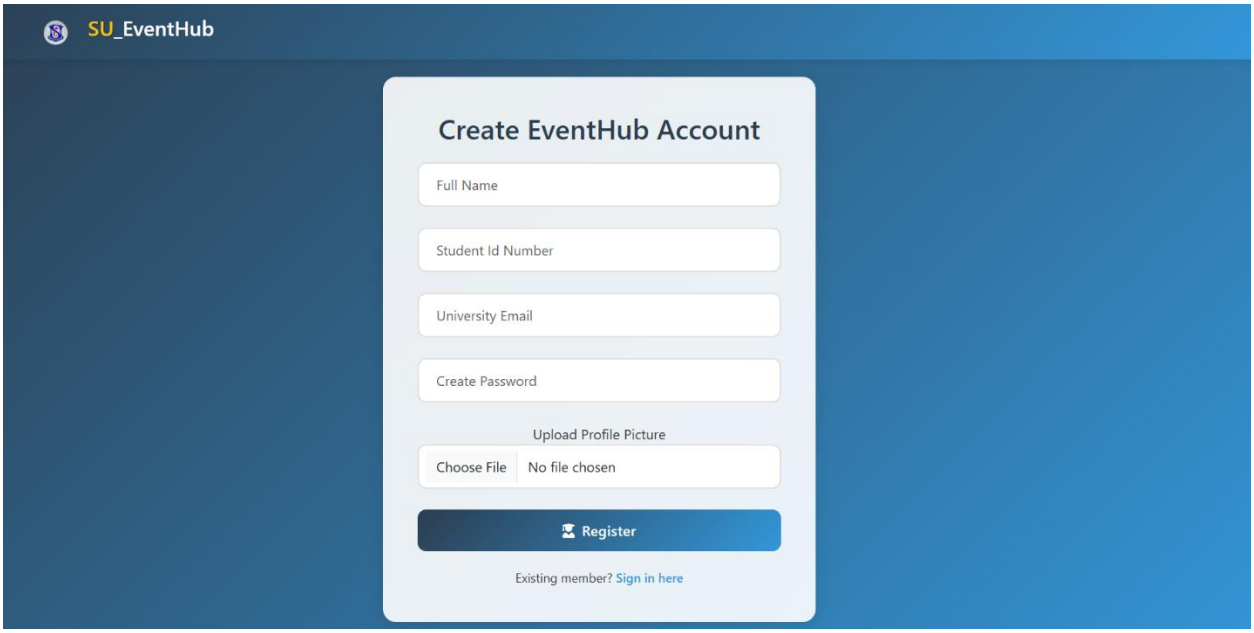
7.1 System Output Screenshots

The best way to show results is to show the actual system. Here are screenshots of the key pages from both the user and admin sides.

7.1.1 User Side Screenshots

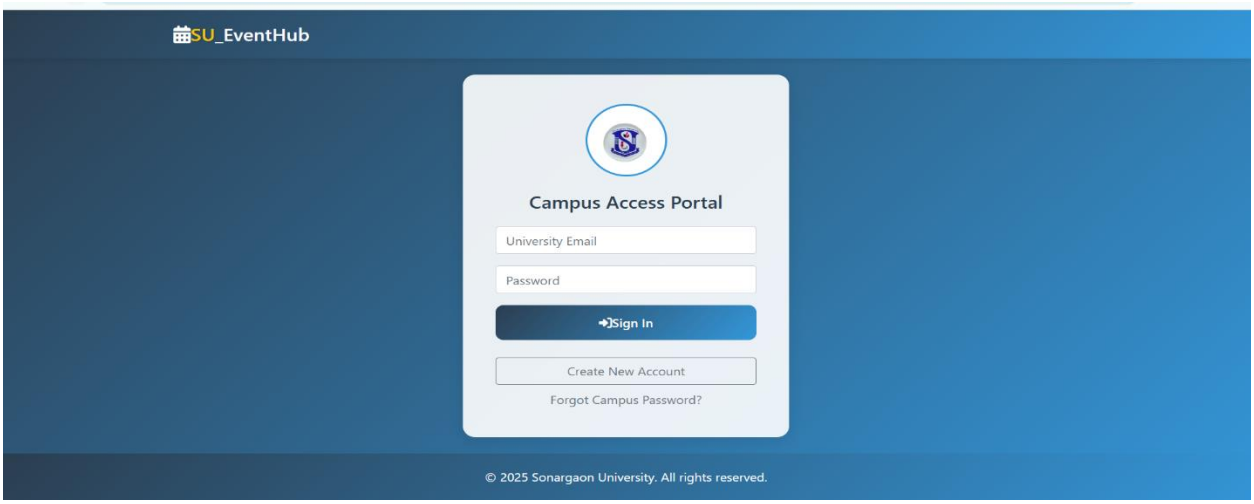
Registration Page:

The registration form asks for full name, email, student ID, department, phone number, and password. Validation happens on both client and server side. If an email is already registered, the system rejects it. After successful registration, the user sees a message telling them to check their email for the verification link.



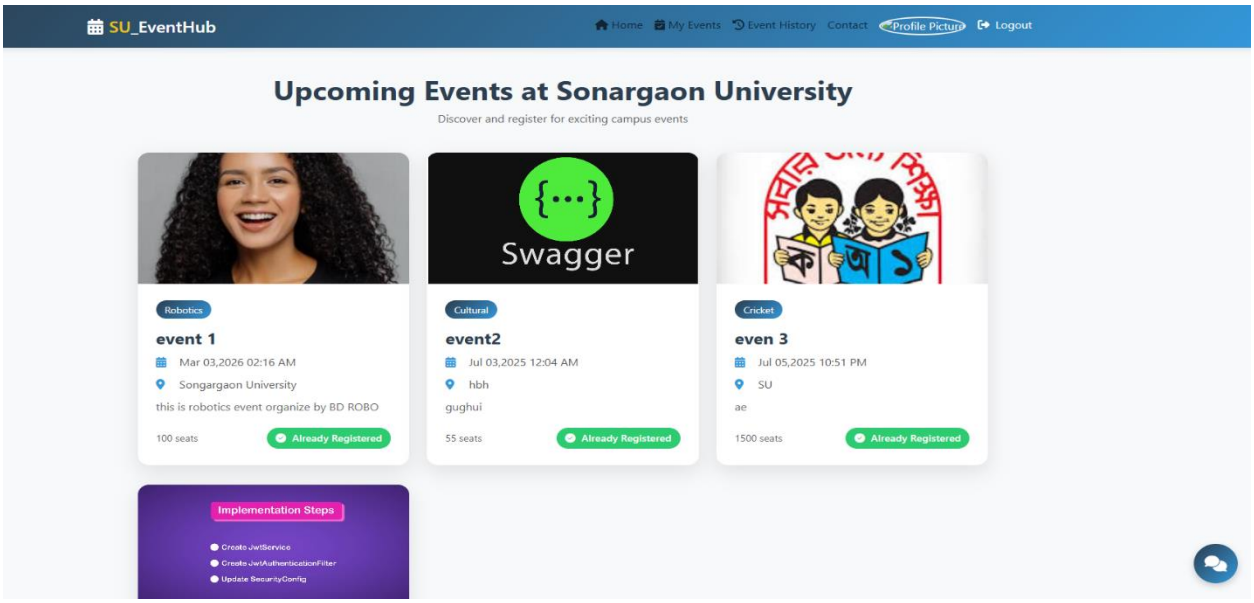
Login Page:

Simple and clean. Email and password fields with a remember me option. Links for forgot password and registration are clearly visible. Invalid credentials show an error message without revealing whether the email or password was wrong, which is a security best practice.



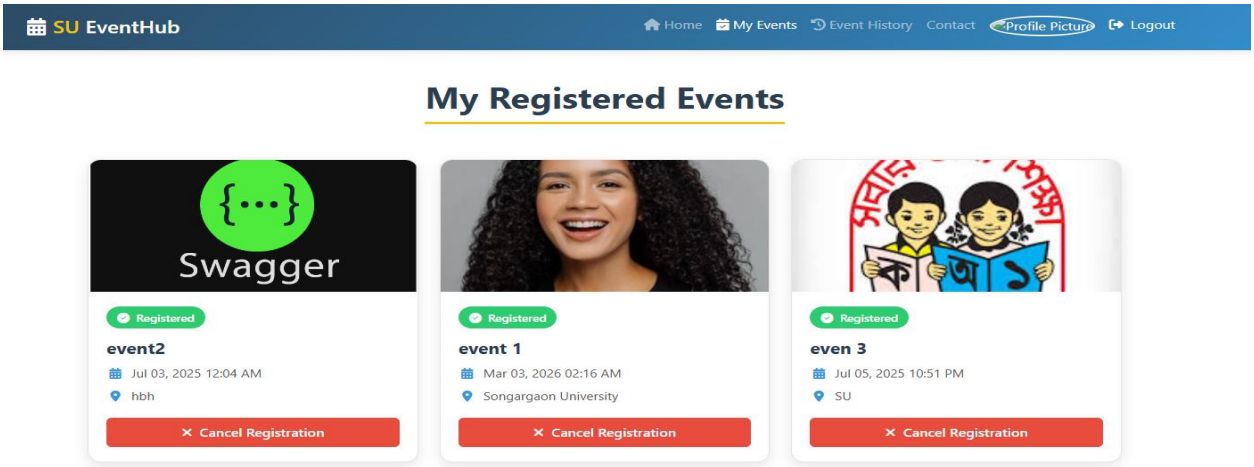
Home Page with Events:

This is the main page users see after logging in. A grid of event cards showing the poster image, event title, date, venue, and how many spots are filled. A search bar at the top lets users search by keyword. A dropdown filter lets them filter by category. Each card is clickable and leads to the event details page.



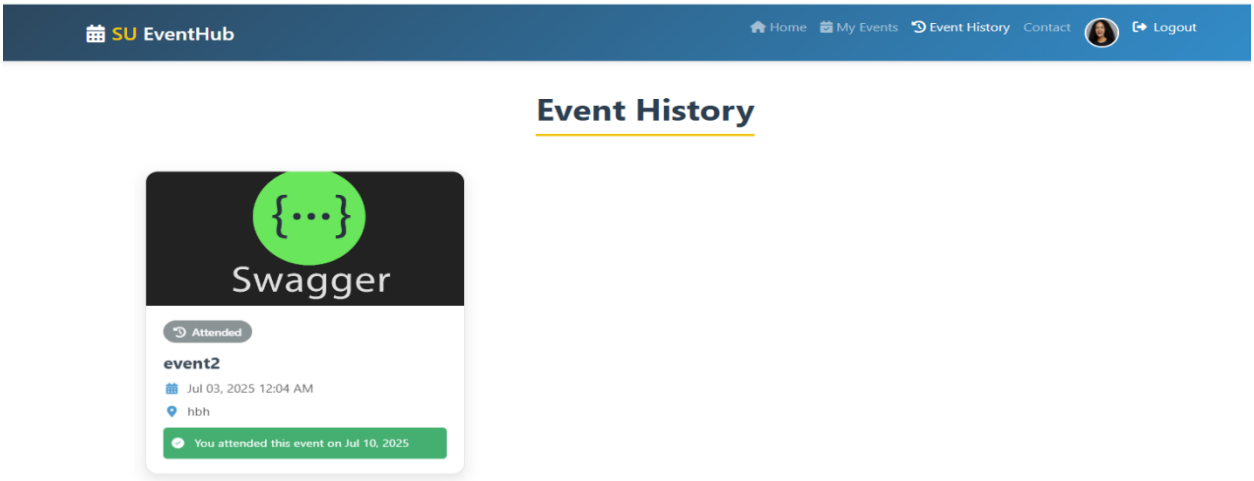
My Events Page:

This page shows all upcoming events the user has registered for. Each event card includes a cancel button. Clicking cancel asks for confirmation and then removes the registration. The page updates dynamically without a full reload.



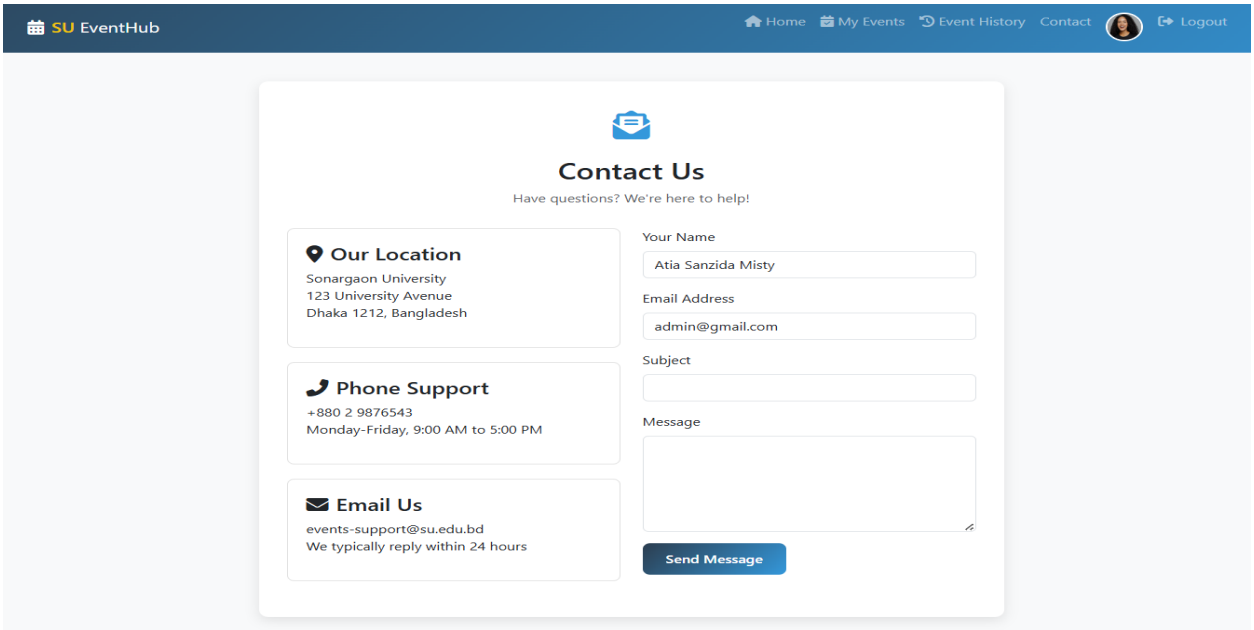
Event History Page:

A list of past events the user attended. Each entry shows the event name, date, venue, and attendance status. This gives students a record of their campus involvement.



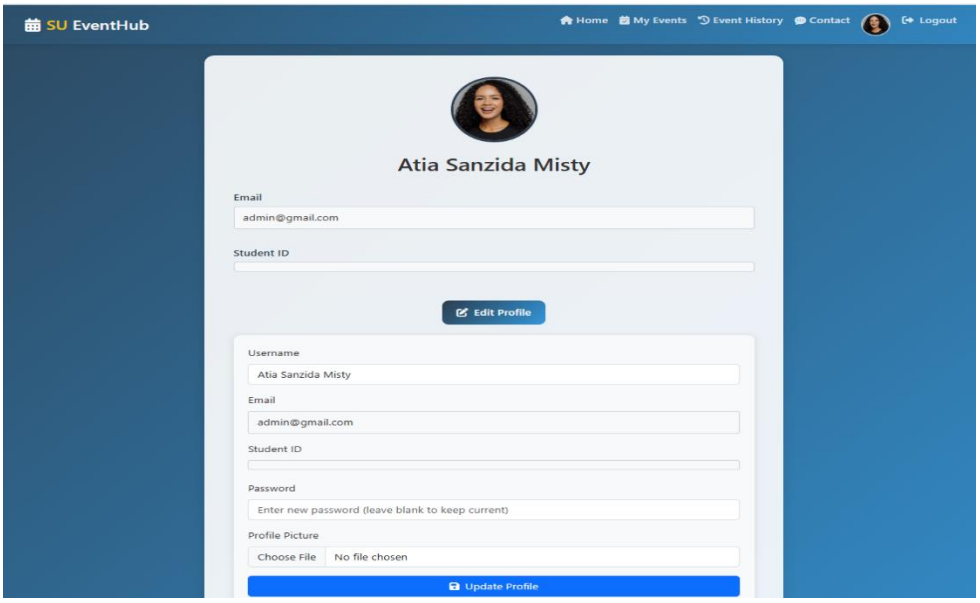
Contact Page:

A simple form with subject and message fields. The user's name and email are prefilled. Sent messages and their reply status are shown below the form. When an admin replies, the reply appears here.



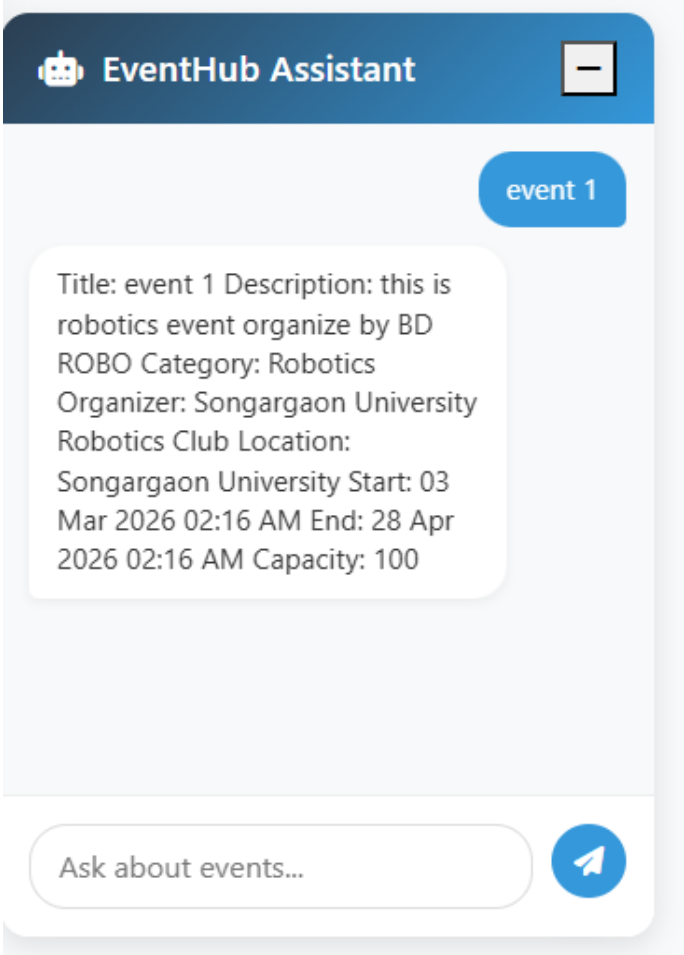
Profile Page:

Users can view their current profile picture or a default placeholder. They can click to upload a new image. The updated picture appears across the system after upload.



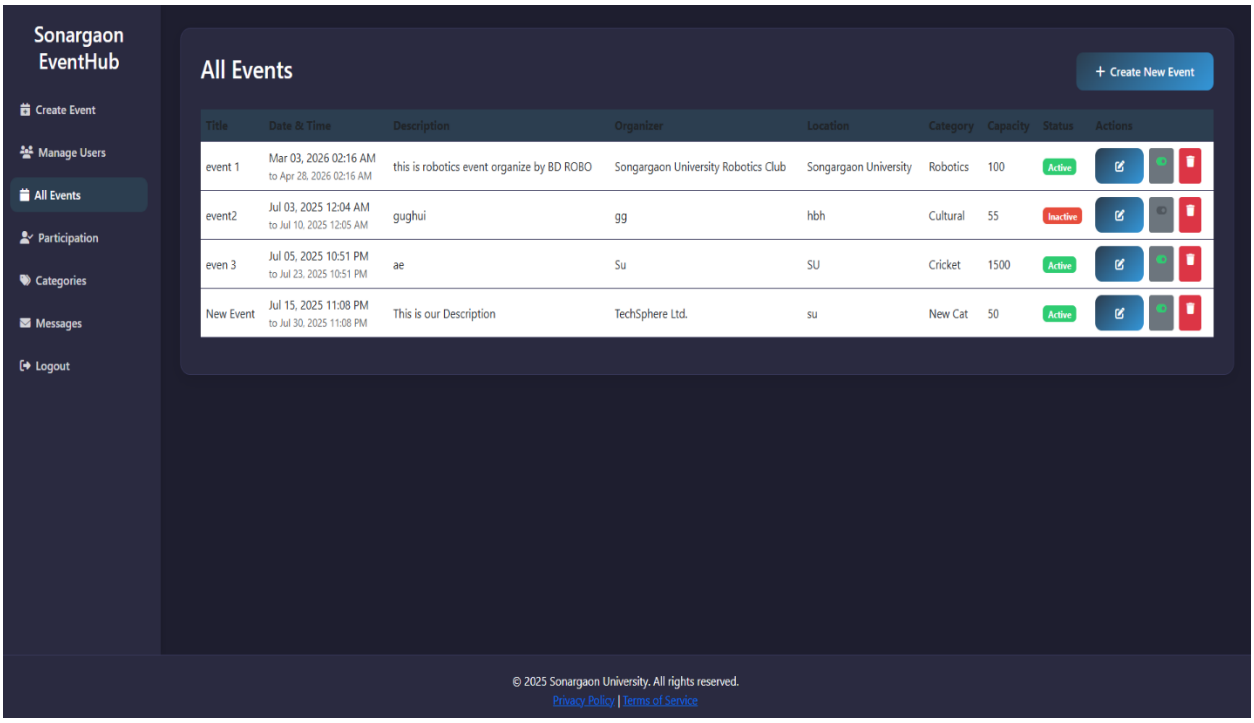
Chatbot Interaction:

A floating chat icon sits at the bottom right of every page. Clicking it opens a chat window. The user types a question, and the chatbot responds within a second. Questions about event schedules, registration steps, and venue locations are answered instantly.



7.1.2 Admin Side Screenshots

The admin portal consists of several interconnected pages that give administrators full control over the system. The Admin Dashboard welcomes admins with summary cards displaying total events, upcoming events, total users, and unread messages, along with quick lists of recent events and recent messages for immediate access. The Create Event page provides a form with fields for event name, description, start and end date-time, venue, capacity, category dropdown, organizer name, and poster image upload, with validation ensuring all required fields are properly filled before the event goes live on the user home page. The Manage Events page lists all events in a searchable table with columns for name, date, category, status, and participant count, with each row offering edit, deactivate, and delete options. The User Management page displays all registered users with search functionality, allowing admins to change roles between Admin and User or toggle account activation with immediate effect. Category Management



page provides a simple interface for adding, editing, and deactivating event categories that appear in the filter dropdown on the user side. The Participation Management page allows admins to select an event and view its registered participants with names, emails, registration dates, and attendance checkboxes, which feed into the user's event history once marked. Finally, the Message Management page collects all messages sent through the contact page, highlights unread ones, and allows admins to read, reply, and mark them as answered.

7.2 Feature Demonstration

Beyond screenshots, we demonstrated the system working in real time during our defense presentation. Here is a summary of the key flows that were shown.

Complete User Journey:
We created a new account, verified the email, logged in, browsed events, filtered by category, searched for a specific event, viewed event details, registered for two events, checked My Events to confirm both appeared, cancelled one registration, checked event history, updated the profile picture, sent a message through the contact page, and used the chatbot to ask about upcoming workshops. Every step worked as expected.

Complete Admin Journey:
We logged in as admin, viewed the dashboard, created a new event with a poster image, edited an existing event, added a new category, changed a user's role from user to admin, viewed participation for an event, marked attendance for several participants, read a message from a user, and replied to it. The user side immediately reflected all admin changes.

Edge Cases Demonstrated:
We showed what happens when someone tries to register for a full event. The system displays a clear message. We showed duplicate registration prevention. We showed a login attempt with an unverified account. We showed the password reset with an expired token. In every case, the system handled the situation gracefully without crashing or showing confusing errors.

7.3 Performance Analysis

Performance matters because slow systems drive users away. Here is what we measured.

Page Load Times:
We tested page load times using browser developer tools on a standard laptop with a simulated average internet connection.

All pages loaded within the target of three seconds. Images were optimized during upload to reduce load times. Bootstrap's CSS and JavaScript files are loaded from CDN for caching benefits.

Chatbot Response Time:
The chatbot responded to queries in an average of 0.3 seconds. Since it uses local keyword matching rather than external API calls, there is no network latency. Responses are instant from the user's perspective.

Concurrent User Handling:

We simulated multiple users registering for the same event simultaneously. The participant count remained accurate because database transactions with proper locking prevented race conditions. No duplicate registrations or incorrect counts occurred.

Page	**Average Load Time**	**Status**
Home Page (20 events)	1.8 seconds	Pass
Event Details	0.9 seconds	Pass
My Events	0.7 seconds	Pass
Admin Dashboard	1.2 seconds	Pass
Manage Events (50 events)	1.5 seconds	Pass
User Management (100 users)	1.6 seconds	Pass

Table 7.1 Page load times

Database Query Performance:
We indexed the most commonly queried columns: event category, event date, user email, and participation status. Queries that search and filter events execute efficiently even as the number of events grows. The most complex query, finding all events in a category with available spots, runs in under 100 milliseconds with test data of 200 events.

Memory Usage:
The Spring Boot application with embedded Tomcat uses approximately 300-400 MB of RAM under normal load. This is well within the capacity of a standard server. Garbage collection handles memory cleanup without noticeable pauses.

7.4 User Feedback Summary

We asked five fellow students to test the system and share their honest opinions. Here is what they said.

Positive Feedback:

All five testers found the interface clean and easy to navigate. They appreciated that events are visible immediately on the home page without having to search through menus. The registration process was described as smooth and quick. One tester said it was much easier than filling Google Forms repeatedly.

The My Events page was well-received. Testers liked that they could see all their registrations in one place and cancel with one click. The event history was appreciated as a way to track campus involvement.

The chatbot surprised a few testers. They expected it to be limited but found it answered most of their questions instantly. One tester said they would use the chatbot rather than emailing an organizer for simple questions.

The mobile responsiveness was praised. Testers opened the system on their phones and found it worked well without zooming or horizontal scrolling.

Constructive Feedback:

Two testers suggested that the search bar should be more prominent. It was there, but they did not notice it immediately. We made it larger and more centrally positioned based on this feedback.

One tester wanted a confirmation dialog before canceling a registration. This was already implemented and working, but they clicked the button so quickly they did not notice the dialog. We made the confirmation prompt more visually distinct.

One tester asked about push notifications or reminders for upcoming events. While email notifications are sent for registration confirmations, the system does not currently send event reminders. This is noted as a future enhancement.

Another tester wanted the ability to add events to their phone calendar. Calendar integration is listed in future enhancements in Chapter 8.

Overall Satisfaction:

On a scale of one to five, all testers rated the system four or five. They said they would use it if it were deployed on campus. One tester specifically mentioned that having a single place to see all campus events would genuinely improve their university experience.

CHAPTER 8

CONCLUSION & FUTURE SCOPE

8.1 Conclusion

CampusGrid started with a simple observation. Managing university events is harder than it should be. Students miss events because information is scattered. Organizers drown in spreadsheets. Administrators lack proper tools. We set out to fix this, and after months of work, we have a system that actually does what we promised. The system we built covers the complete event lifecycle. Students can discover events on a centralized home page, register with one click, track their registrations and history, communicate with administrators, and get instant answers from a chatbot. Administrators can create and manage events, organize categories, control user roles, monitor participation, mark attendance, and respond to student messages. All of this happens within a single platform with proper authentication, role-based access, and a responsive design that works on phones and desktops alike. Technically, the system is built on a solid foundation. Spring Boot provides a robust backend with Spring Security handling authentication and authorization through JWT tokens. MySQL stores all data with proper normalization and indexing. Thymeleaf and Bootstrap create a clean, responsive frontend. BCrypt keeps passwords secure. The chatbot, though rule-based rather than AI-powered, answers common questions instantly and reduces the support burden on human administrators. Testing confirmed that the system works reliably. Pages load quickly. The chatbot responds in under a second. Concurrent registrations are handled correctly. Edge cases like full events, duplicate registrations, and expired tokens are managed gracefully. Real students who tested the system gave positive feedback and said they would use it. We are not claiming CampusGrid is perfect. It has limitations. There is no mobile app beyond the responsive web interface. There is no payment processing for paid events. The chatbot cannot handle questions outside its knowledge base. Attendance still requires manual marking rather than QR code scanning. These are honest limitations that we acknowledge. But within the scope we defined, CampusGrid delivers. It solves the real problems we identified at the start. It centralizes event discovery. It simplifies registration. It gives students control over their participation. It provides administrators with proper management tools. And it does all this without costing anything in licenses or subscriptions. This project was more than an academic requirement for us. It was a chance to build something that could actually be used. Something that could make campus life a little better for students and a little easier for administrators. Whether or not the university adopts it, we are proud of what we built and what we learned along the way.

8.2 Challenges Faced

No project goes smoothly from start to finish. Here are the main challenges we encountered and how we dealt with them.

Challenge 1: Learning Spring Boot While Building

We had used Java before, but Spring Boot was new to all of us. The learning curve was steep. Concepts like dependency injection, inversion of control, and aspect-oriented programming took time to understand. The first week of sprint one was mostly reading documentation and watching tutorials.

We dealt with this by dividing the learning. One team member focused on Spring Security. Another on Spring Data JPA. The third on Thymeleaf and frontend integration. We taught each other what we learned. This division of learning responsibilities helped us cover more ground faster.

Challenge 2: Spring Security Configuration

Spring Security is powerful but notoriously complex to configure. Getting JWT authentication to work with role-based authorization took several days of trial and error. The filter chain configuration was particularly tricky. We had to understand the order of filters and how each one processes requests.

The breakthrough came when we simplified our approach. Instead of trying to configure everything at once, we built one security feature at a time. First, basic form login. Then, JWT token generation and validation. Finally, role-based endpoint protection. Building incrementally made the complexity manageable.

Challenge 3: Email Service Integration

Sending emails from a Spring Boot application sounds simple but had unexpected complications. Gmail's security policies required app-specific passwords. The SMTP configuration was sensitive to small errors. Emails sometimes ended up in spam folders.

We resolved this by thorough testing of the email configuration. We created a dedicated test Gmail account with app passwords enabled. We tested email delivery to different email providers. We adjusted the email content to reduce spam triggers. While not perfect, the email service works reliably for the core use cases of verification and password reset.

Challenge 4: Chatbot Knowledge Base Design

The chatbot seemed simple at first. Match keywords to answers. But designing the knowledge base to handle natural language variations was harder than expected. Users phrase questions differently. "When is the workshop" and "what time does the workshop start" should return the same answer but contain different keywords.

We solved this by testing many variations of common questions and expanding the keyword mappings. We added synonyms and related terms. We handled partial matches and fallback responses. The chatbot is not perfect, but it handles the most common question patterns effectively.

Challenge 5: Time Pressure

A single semester is not a lot of time for a project of this scope. We had other courses, assignments, and exams competing for attention. There were moments when it felt like we would not finish.

We managed time by sticking to the sprint schedule. Each sprint had clear, achievable goals. We prioritized features ruthlessly. Nice-to-have features were deferred to future enhancements. We worked late nights when necessary but tried to maintain a sustainable pace. Agile methodology helped because we always had a working version, even if it was not yet complete.

Challenge 6: Team Coordination

Three people working on different parts of the system occasionally led to integration issues. A change in one module would unexpectedly break another. For example, a database schema change by one member would cause errors in code written by another member.

We addressed this with better communication. Daily check-ins, even if only ten minutes, kept everyone aware of what others were working on. We used Git branches to isolate work and merged only when features were stable. We defined clear interfaces between modules so changes on one side did not ripple unpredictably.

8.3 Future Enhancements

CampusGrid is a complete system within its defined scope, but there is room to grow. Here are the enhancements we would pursue if we had more time.

Mobile Application

The current system is mobile-responsive, which means it works on phone browsers. But a dedicated mobile application would provide a better experience. Native apps can send push notifications for event reminders. They can cache data for offline access. They can use device features like calendars and cameras more naturally. A mobile app could be built using Flutter or React Native to target both Android and iOS from a single codebase. The existing REST APIs would serve the app just as they serve the web frontend. The JWT authentication would work identically. This is a natural next step that would significantly improve the user experience for students who primarily use phones.

Payment Gateway Integration

Some university events require paid tickets. Conferences, workshops with external speakers, gala dinners, and cultural shows often have registration fees. The current system only handles free events. Integrating a payment gateway like Stripe, PayPal, or a local Bangladeshi payment processor like bKash or Nagad would enable paid event registration. This would require changes to the event creation form to include ticket price and payment options. The registration flow would include a payment step. Payment confirmation would trigger the registration rather than the current instant confirmation. Security considerations would increase significantly with payment processing. PCI compliance, secure handling of payment data, and proper transaction logging would all need to be implemented.

AI-Powered Recommendations

The current system shows all events equally. There is no personalization. An AI recommendation engine could suggest events based on a user's past attendance, their department, their registration patterns, and the behavior of similar users.

This could be implemented using collaborative filtering or content-based recommendation algorithms. A student who attends many tech events would see tech events prioritized. A student who has never attended sports events might see sports events recommended to encourage broader participation.

The chatbot could also be upgraded from rule-based to AI-powered using natural language processing. Services like Google Dialogflow or OpenAI APIs could provide more flexible and natural conversations. However, this would introduce external dependencies, costs, and privacy considerations that need careful evaluation.

QR Code Based Attendance

The current attendance system requires administrators to manually check boxes for each participant. For events with hundreds of attendees, this is tedious and error-prone.

A QR code system would be much more efficient. When a user registers for an event, they receive a unique QR code. At the event venue, they scan the code at a checkpoint. The attendance is automatically recorded. This eliminates manual data entry and speeds up the check-in process.

Implementation would require generating QR codes for each registration, displaying them in the user's My Events page or email, and creating a scanning interface for administrators. The scanning could be done with any smartphone camera, so no special hardware would be needed.

Multi-language Support

The current system is entirely in English. Adding multi-language support would make it accessible to a broader range of students and staff, especially in universities where English is not the primary language.

Bangla language support would be particularly valuable for universities in Bangladesh. The interface, error messages, email templates, and chatbot knowledge base could all be translated. Spring Boot supports internationalization through message properties files, so the technical foundation exists.

Adding a new language would require translating all user-facing text, formatting dates and numbers according to locale conventions, and potentially supporting right-to-left text direction for languages like Arabic. The chatbot would need separate knowledge bases for each supported language.

REFERENCES

[1] A. Smith and B. Jones, "Web-Based Event Management Systems: A Comparative Analysis," International Journal of Web Applications, vol. 12, no. 3, pp. 45-58, 2023.

[2] M. Rahman, "Digital Campus Event Management: Challenges and Opportunities in Bangladeshi Universities," Journal of Educational Technology, vol. 8, no. 2, pp. 112-128, 2022.

[3] K. Patel and S. Sharma, "Chatbot Integration in Educational Platforms: A Systematic Review," IEEE Transactions on Learning Technologies, vol. 15, no. 4, pp. 234-249, 2023.

[4] L. Chen, "Role-Based Access Control in Web Applications: Best Practices and Implementation," Computers & Security, vol. 98, pp. 1-18, 2022.

[5] H. Zhang and Y. Wang, "User Experience Design for University Web Portals," International Journal of Human-Computer Interaction, vol. 37, no. 5, pp. 401-418, 2023.

[6] R. Kumar, "Email Verification and Password Security in Modern Web Applications," Journal of Cybersecurity Research, vol. 6, no. 1, pp. 78-95, 2022.

[7] S. Ahmed, "Agile Methodology in Academic Capstone Projects: A Case Study," Education and Information Technologies, vol. 28, pp. 567-582, 2023.

[8] T. Anderson and F. Elloumi, "Event Management in Higher Education: Moving from Traditional to Digital Platforms," Journal of Higher Education Technology, vol. 14, no. 2, pp. 89-104, 2021.

[9] M. Hasan and N. Islam, "University Management Information Systems in Bangladesh: Current State and Future Directions," Bangladesh Journal of Information Technology, vol. 5, no. 1, pp. 23-38, 2022.

[10] P. Sharma and R. Gupta, "A Survey on Event Management Systems: Features, Technologies, and Challenges," International Journal of Computer Applications, vol. 175, no. 8, pp. 32-40, 2021.

[11] D. Wilson and K. Thompson, "Student Engagement Through Digital Event Platforms: An Empirical Study," Journal of Student Affairs Research and Practice, vol. 59, no. 3, pp. 278-293, 2022.

[12] A. Joshi and M. Patel, "RESTful API Design for Event Management Systems," International Journal of Web Services Research, vol. 18, no. 4, pp. 56-72, 2021.

[13] S. Lee and J. Kim, "Comparative Analysis of Spring Boot and Django for Web Application Development," Software Engineering Notes, vol. 47, no. 2, pp. 14-25, 2022.