



# Greek Life Member Management System

11.30.2018

Software Engineering 1 - Group 9 - Report 3

Jason Pulis, Christopher Whetsel, Mike Winkelmann

File Repository -

<https://github.com/mawinkelmann/databaseUpgradeSEGroup9>

Development Server - <http://glmms.online>

Client Server - <http://spdmizzou.com>

Development Blog - <https://glmms.home.blog>



**All team members  
contributed equally.**

---

# Table of Contents

<b>Summary of Changes</b>	<b>5</b>
<b>1 Project Description</b>	<b>6</b>
1.1 Problem Diagnosis	6
1.2 Solution	7
1.3 Glossary of Terms	8
<b>2 System Requirements</b>	<b>10</b>
2.1 Enumerated Functional Requirements	10
2.2 Enumerated Nonfunctional Requirements	11
2.3 On-Screen Appearance Requirements	11
<b>3 Functional Requirements Specification</b>	<b>13</b>
3.1 Stakeholders	13
3.2 Actors and Goals	13
3.3 Use Cases	15
3.3.1 Casual Description	15
3.3.2 Use Case Diagram	16
3.3.3 Traceability Matrix	18
3.3.4 Fully-Dressed Description	19
3.4 System Sequence Diagrams	28
<b>4 Effort Estimation Using Use Case Points</b>	<b>36</b>
4.1 UUCP: Unadjusted Use Case Points	36
4.1.1 Unadjusted Use Case Points (UUCPs)	36
4.1.2 Unadjusted Use Case Weight (UUCW)	37
4.2 TCF: Technical Complexity Factor	38
4.3 ECF: Environmental Complexity Factor	39
4.4 Deriving Project Duration from Use Case Points	40
<b>5 Domain Analysis</b>	<b>41</b>
5.1 Domain Model	41
5.1.1 Concept Definitions	42
5.1.2 Association Definitions	43
5.1.3 Attribute Definitions	44
5.1.4 Traceability Matrix	45
5.2 System Operations Contracts	47
<b>6 Interaction Diagrams</b>	<b>50</b>
<b>7 Class Diagrams and Interface Specifications</b>	<b>64</b>

7.1 Class Diagram	64
7.2 Data Types and Operation Signatures	64
7.3 Traceability Matrix	69
<b>8 System Architecture and System Design</b>	<b>73</b>
8.1 Architecture Style	73
8.2 Subsystems	73
8.3 Subsystems to Hardware Map	74
8.4 Persistent Data Storage	75
8.5 Network Protocol	76
8.6 Global Control Flow	76
8.7 Hardware Requirements	77
<b>9 Algorithms and Data Structures</b>	<b>78</b>
9.1 Algorithms	78
9.2 Data Structures	78
<b>10 User Interface Design and Specification</b>	<b>78</b>
10.1 Interface Design	78
10.1.1 Use Case UC-1: Searching Member Information	79
10.1.2 Use Case UC-2: Using Administrator Powers	81
10.1.2.1 Admin homepage	81
10.1.2.2 Add/Edit Members	81
10.1.3 Use Case UC-3: Creating a Text Announcement	82
10.1.4 Use Case UC-4: Updating Member Information	82
10.1.5 Use Case UC-5: Recovering Password	84
10.1.7 Use Case UC-7: Creating and Viewing Announcements	85
10.1.10 Use Case UC-10: Managing Events	87
10.1.11 Use Case UC-11: Logging into the System	88
10.2 User Effort Estimation	89
10.2.1 Scenario 1	89
10.2.2 Scenario 2	89
10.2.3 Scenario 3	90
10.2.4 Scenario 4	90
<b>11 Design of Tests</b>	<b>91</b>
11.1 Test Cases	91
11.2 Test Coverage	96
11.3 Integration Testing Strategy	97
<b>12 History of Work</b>	<b>98</b>
12.1 History of Work	98
12.2 Accomplishments	100
12.3 Future Work	100

---

12.4 Breakdown of Final Responsibilities	100
<b>13 References</b>	<b>102</b>

## Summary of Changes

1. We removed some use cases from the Demo-2 objective because of time constraints. Specifically, [Use Cases](#) 6, 8, 9, 12, 13, and 14, as they were determined to be of lesser importance.
2. Since Use Case 6 was deferred for development, the member\_picker method was removed from the [MemberViews Class](#)
3. Password reset use case was changed to use an email link to reset the password instead of security questions. This caused changes to the database, removing some tables, and the description of the [Use Case 5](#).
4. Because the password reset use case was changed, the [AuthViews](#) class had its display\_security\_qs method removed and the methods display\_reset\_email, handle\_reset\_email, and display\_reset\_form added. The handle\_reset method was also changed.
5. References to the [actor](#) 'Google Calendar' were changed to 'Calendar Application.'
6. Total PW weight and Max Priority Weight were added to the [Priority Weight table](#).
7. Fully dressed Use Cases, Sequence Diagrams, and Interaction diagrams were added for UC 4, 5, 7, 10.

# 1 Project Description

## 1.1 Problem Diagnosis

One of the problems that the engineering fraternity Sigma Phi Delta is facing is the inability to effectively communicate events, knowledge and other pertinent information to the members of the group. This is brought about by the extensive list of applications that are designed to bring groups to a common line of communication. Sigma Phi Delta is currently using a hybrid of communication networks such as Slack, GroupMe, email and OrgSync. As a result, the members and officers in our organization are frustrated by the constant application hopping, inconsistent notifications for events and the lack of carry-over from each application. The Executive Board of the fraternity has also found out that many of the members have turned off the notifications from the chat because of the spam that comes from the communication overload. Thus, the Executive Board of Sigma Phi Delta has decided to seek a centralized location for member information and communication.

Another issue that the fraternity is facing is a jumbled mess of documents, by-laws and bills that stream through emails and get put into a cluttered Google Drive. Unless a member of the fraternity plays a large role in the upkeep and development of the documents that are used every week, there is no reason for them to look at them. That being said, when a member does need to reference a by-law or lookup formatting for a bill, the Google Drive becomes a maze of folders - leading to dead-ends, out of date files and duplicates. Because of this, some variety of search function and organization needs to be implemented to help the less tech-savvy people in the fraternity.

For the executive board and other decision makers in the fraternity to be successful they need to be able to access information about the members they are leading. Currently, when we need information, the member seeking it needs to search through the fraternity Google Drive for it, if it even exists. This process is time consuming and often fruitless. If that information cannot be found, they resort to polling the group message and hoping all necessary members will respond with their information. This is inefficient and often inaccurate, costing time and delaying meaningful work that requires these answers.

The Secretary of Sigma Phi Delta has also raised the issue that weekly officer reports are time consuming to compile and collect from each position holder. It is a menial task he must perform every week. They are submitted by each officer via email to the secretary who is incharge of compiling them. The officers often forget to send one in or do not indicate to the Secretary that they will not be submitting one. The Fraternity also is seeking a way to speed up its officer elections. Currently voting is done by hand and counted manually by the Secretary. This takes time and is possible to introduce error.

Last year, an attempt was made by members of the Sigma Phi Delta fraternity to implement a member information system. While the design of the database itself was good, the

---

implementation of the web interface and the small set of features offered made sure that it quickly failed. It has since fallen out of use and never fulfilled its goals. Its interface was confusing and difficult to navigate, providing little instruction on how to use the system. The database also never held information for all members, so using it did not provide valuable information. We need a way to provide the functionality this system tried to achieve, but in a way that will be used by the fraternity for years to come.

Clearly, Sigma Phi Delta is facing many issues regarding management of information, documents, and communication that frustrates our members, costs fraternity leaders time, and could lead to inaccurate action to be taken. These issues need to be addressed.

## 1.2 Solution

To solve the problems described in the above section, we propose a Greek Life Member Management System, which will build upon the existing member information database that Sigma Phi Delta previously attempted to implement. Lack of interest because of unfinished features and an unpolished user experience were major drawbacks of the previous solution. So much so, that it has been abandoned as a solution to our problems. The new solution must be more user friendly and provide new features which will make it more appealing for general members and decision makers in our organization alike. The main benefit of the system to users will be a centralized and searchable location accessible to all members for information regarding organization personnel, announcements, events, and documents. We also want this solution to be useable and maintainable for many years, so that we have a continuity of member information.

Our desired solution has several goals to improve fraternity communication and information management. One goal in creating this system is to provide valuable and accurate information for the leaders of the fraternity as they make decisions impacting all members. The solution must also be easy to use and should save members time and hassle when they are trying to compile information. The information must be gatherable by the members without having to reach out to each member individually or through a group message. The system should also provide a way to coordinate members for events and will provide an organized archive for past chapter documents. Events should also be synchronized with the fraternity Google Calendar as most members already have it on their phones. Our solution will also make it simple to find important announcements for the chapter and will allow urgent messages to be sent out by text to all members. Another desirable feature would be to randomly pick members to clean up after events and to keep track of who has already been chosen in the past. A feature to allow officers to submit their reports online and compile them would be desirable. The system should also send reminders to officers that have not submitted a report. There should be a way to create a voting poll for each position when we hold elections that allows each member to vote once. Finally, the solution must be secure and only accessible to members of the hosting organization.

This project hopes to satisfy the requirements for what a Greek Life chapter needs to function as one cohesive unit. If the implemented features are easy to learn and maintain, the better the organization will survive in the long run because we will be able to manage and learn



from our information. Each member must be able to pick-up the programs that are created and intuitively solve whatever they seek to accomplish. It should also be expandable so that in the future, the fraternity can add features as new needs arise. As an end goal, this could be used for any Greek Life organization. One of the things that we realize this relies on is member participation in the system, because if we do not consistently use this as our form of communication and keep the information up to date, we will not be able to benefit from this system.

## 1.3 Glossary of Terms

**Database** - a usually large collection of data organized especially for rapid search and retrieval (as by a computer).

**Fraternity** - a student organization for scholastic, professional, or extracurricular activities.

**Greek Life** - Fraternities and sororities, or Greek letter organizations (GLOs) (collectively referred to as "Greek life") are social organizations at colleges and universities.

**Fraternity Father/Son** - In fraternities, during the new member education process, a new member to the organisation is given a "father" to help mentor them in the fraternity.

**Chapter** - a local branch of an organization (fraternity) or a weekly meeting held by a fraternity or sorority which all members are required to attend.

**Greek Life Member** - a member of a Greek Life organization.

**Fraternity Member** - an initiated member of a fraternity.

**Member Information** - information about a member of an organization that could be useful for the organisation to keep track off. E.g. major, phone number, graduation date, email etc.

**Sigma Phi Delta** - an international professional-social fraternity of engineers.

**Executive Board** - the group of elected individuals who oversee the activities of the organization. In Sigma Phi Delta, the Executive Board consists of the President, Vice President, Internal Operations Engineer, Secretary, Treasurer, Social Chair, Recruitment Chair, Philanthropy Chair, and New Member Educator.

**Slack** - a cloud-based set of proprietary team collaboration tools and services. The name is an acronym for "Searchable Log of All Conversation and Knowledge".

**GroupMe** - a mobile group messaging app.


**OrgSync** - an online community management system to higher education institutions in the United States and Canada. OrgSync enables colleges and universities to communicate with students and staff, track student involvement, and manage campus organizations and programs.

**Google Drive** - a file storage and synchronization service developed by Google.

**Decision Makers** - members of the organisation who have to make decisions that will impact the entire organisation.

**Officer Reports** - something that an certain member of the fraternity sends in to have read during chapter, usually important announcements

**Email Server** - a computer running special software which enables the exchange of emails using Simple Mail Transfer Protocol (SMTP).



**SMS Gateway** - it allows a computer to send or receive Short Message Service (SMS) transmissions to or from a telecommunications network, mostly routing to mobile phone networks.

**Adobe Photoshop** - a computer program that allows users to create and edit images interactively on the computer screen developed and published by Adobe Systems for macOS and Windows.

**draw.io** - draw.io is an open source technology for creating diagrams

**Github** - is a web-based hosting service for version control using Git which is mostly used for computer code.

**Calendar Application**- Online Calendar Application used by the Fraternity to keep track of organization events.

**SQL** - Stands for Structured Query Language. SQL lets you access and manipulate databases.

## 2 System Requirements

Based upon the needs of Sigma Phi Delta, we developed a list of requirements for the system to possess. For each requirement, we assign an identifier in the form of REQ-(1,2,3...x), as well as a priority weight from 1 to 5. A lower priority weight indicates that the corresponding requirement is more essential to the success of the project, and more critical to fulfilling the customer's needs.

### 2.1 Enumerated Functional Requirements

ID	Priority	Description
REQ-1	1	System shall have searchable member information database with the ability to generate spreadsheets from a query
REQ-2	1	System shall have secure login over https
REQ-3	1	System shall allow administrators to change member information and add new members
REQ-4	1	System shall send urgent announcements via text message to all active members
REQ-5	2	System shall have different levels of access for certain users (i.e. admin or executive member)
REQ-6	2	System shall have the ability for each member to update and view their information on the system
REQ-7	2	Users shall be able to display member and alumni employment information and position history
REQ-8	2	The system shall be able to send initial and periodic reminders for users to update their information
REQ-9	2	System shall have filterable announcements feed with topics
REQ-10	3	Members who have a position should be able to access transition material and the description of their position responsibilities
REQ-11	3	Users should have the ability to recover account after forgotten password
REQ-12	3	Users should have the ability to randomly choose members from the database for extra duties

<b>REQ-13</b>	<b>3</b>	System should have the ability to link events from database to fraternity Calendar Application
<b>REQ-14</b>	<b>4</b>	Users should have the ability to create an event that users can RSVP to and the event creator can send email reminders to the RSVP-ed users
<b>REQ-15</b>	<b>4</b>	System should have an organised file archive
<b>REQ-16</b>	<b>5</b>	System should have the ability to generate a graphical fraternity family tree
<b>REQ-17</b>	<b>5</b>	System should have the ability to host fraternity election voting polls
<b>REQ-18</b>	<b>3</b>	System should have the ability to create/submit officer report forms

## 2.2 Enumerated Nonfunctional Requirements

<b>ID</b>	<b>Priority</b>	<b>Description</b>
<b>REQ-19</b>	<b>1</b>	UI shall be intuitive.
<b>REQ-20</b>	<b>1</b>	UI shall be mobile friendly.
<b>REQ-21</b>	<b>2</b>	System response times to User actions should be under 3 seconds.

## 2.3 On-Screen Appearance Requirements

Our website will serve as a starting point for the members of the fraternity. Important links should be easy to find, toolbars should be used in the header and/or footer. Our website must be intuitive and functional while remaining aesthetically pleasing. Many of the design elements will be taken from the central colors and themes of the fraternity. These images are mock-ups, and might change in the future based on needs/requirements.



fig. 1 (Crest of Sigma Phi Delta) Will be included in the design of the UI.

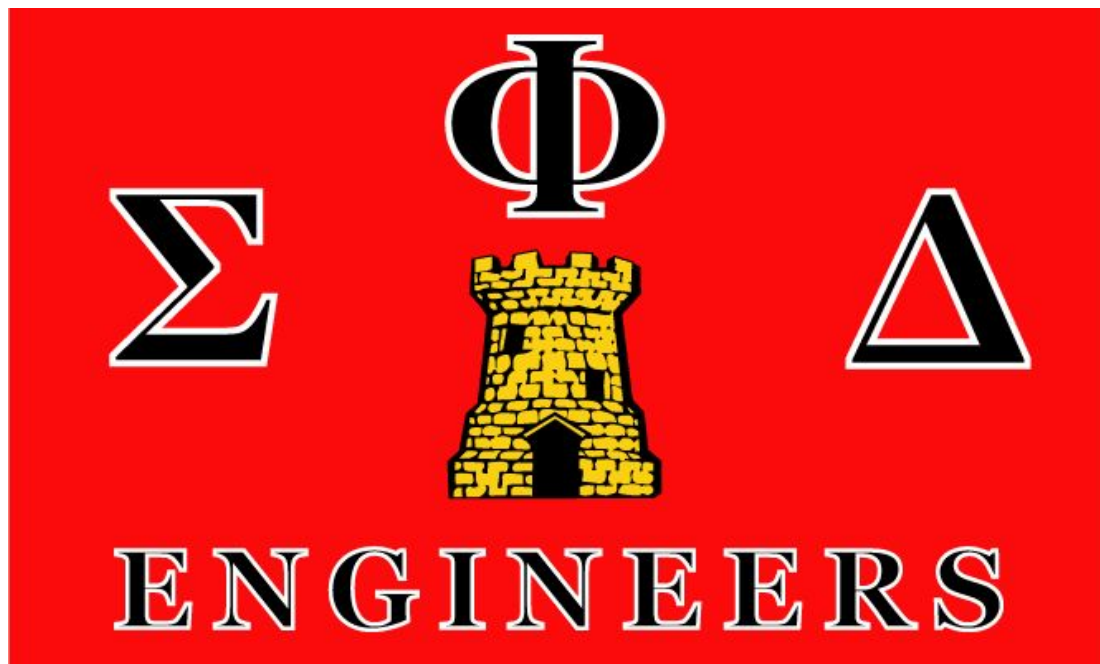


fig. 2 (Flag of Sigma Phi Delta) Will be included in the design of the UI.

## 3 Functional Requirements Specification

### 3.1 Stakeholders

Stakeholders include individuals and organizations which are interested in the completion and use of a given product. The amount of stakeholders and different types of stakeholders relies on the versatility and ease-of-use of the product in question. Stakeholders for this project include all current and future members and alumni of the Sigma Phi Delta fraternity. Other Greek Life organizations at Mizzou or around the country may also be interested in this project to help organize their information and communication inside their organization.

Specifically, from Sigma Phi Delta, the Executive Board has sanctioned the creation of this product and will be the primary product owners for the project. Members of the Executive Board include:

- A. President: Jake Shulman
- B. Vice President: Dameron Taylor
- C. Internal Operations Engineer: Thomas Schuly
- D. Secretary: Mike Winkelmann
- E. Treasurer: Josh Westbrook
- F. Social Chair: Aaron Henry
- G. Recruitment Chair: Marcos Ehinger
- H. Philanthropy Chair: Austin Kimes
- I. New Member Educator: Zack Becker

### 3.2 Actors and Goals

Actors can be defined as people or devices that will directly interact with the product, and can also be loosely labeled as either "initiators" or "participators". These actors will have a specific goal with the given product, which is what the actors are attempting to achieve by interacting with the system. Actors and their respective goals are: Human actors will be interacting with the system, through a created account. These actions could range from sending in Officer Reports to finding the list of actives and new members in the website database. Some actors will have specific functions at their disposal, exclusive to the executive board.

<i>Actor</i>	<i>Actor's Goal</i>	<i>Use Case</i>
User(initiator)	Create and edit all accounts (as administrator)	UC-2

User(initiator)	Manage users own account	UC-4
User(initiator)	Query database for member information and view results	UC-1
User(initiator)	Send announcements via text (SMS)	UC-3
User(initiator)	Recover lost password	UC-5
User(initiator)	Randomly choose active members for extra duties	UC-6
User(initiator)	Access fraternity files materials	UC-8, UC-12
User(initiator)	Create events with an RSVP that send reminders to those that plan to attend	UC-10
User(initiator)	Create and vote in fraternity elections	UC-14
User(Initiator)	View/create announcements	UC-7
User(initiator)	Type and submit officer reports every week using the system.	UC-9
User(Initiator)	User logs into the System using their Username and Password	UC-11
User(Initiator)	User wants to generate a graphical representation of member's Fraternity lineage.	UC-13
Database(participating)	The Database will store and retrieve information based on requests made by the User.	UC-1, UC-2, UC-3, UC-4, UC-5, UC-6, UC-7, UC-9, UC-10, UC-11, UC-13, UC-14
Email Server(participating)	Web email server will send an email to the a Mobile Carriers' SMS gateway or to a User.	UC-2, UC-3, UC-6, UC-10
SMS Gateway(participating)	Mobile Carrier email to SMS gateway will take in an email from the web server email server and send out a text message to the desired phone	UC-3

	number.	
Calendar Application(participating)	The Fraternity Calendar will be updated by the System when a new Event is created.	UC-10

## 3.3 Use Cases

### 3.3.1 Casual Description

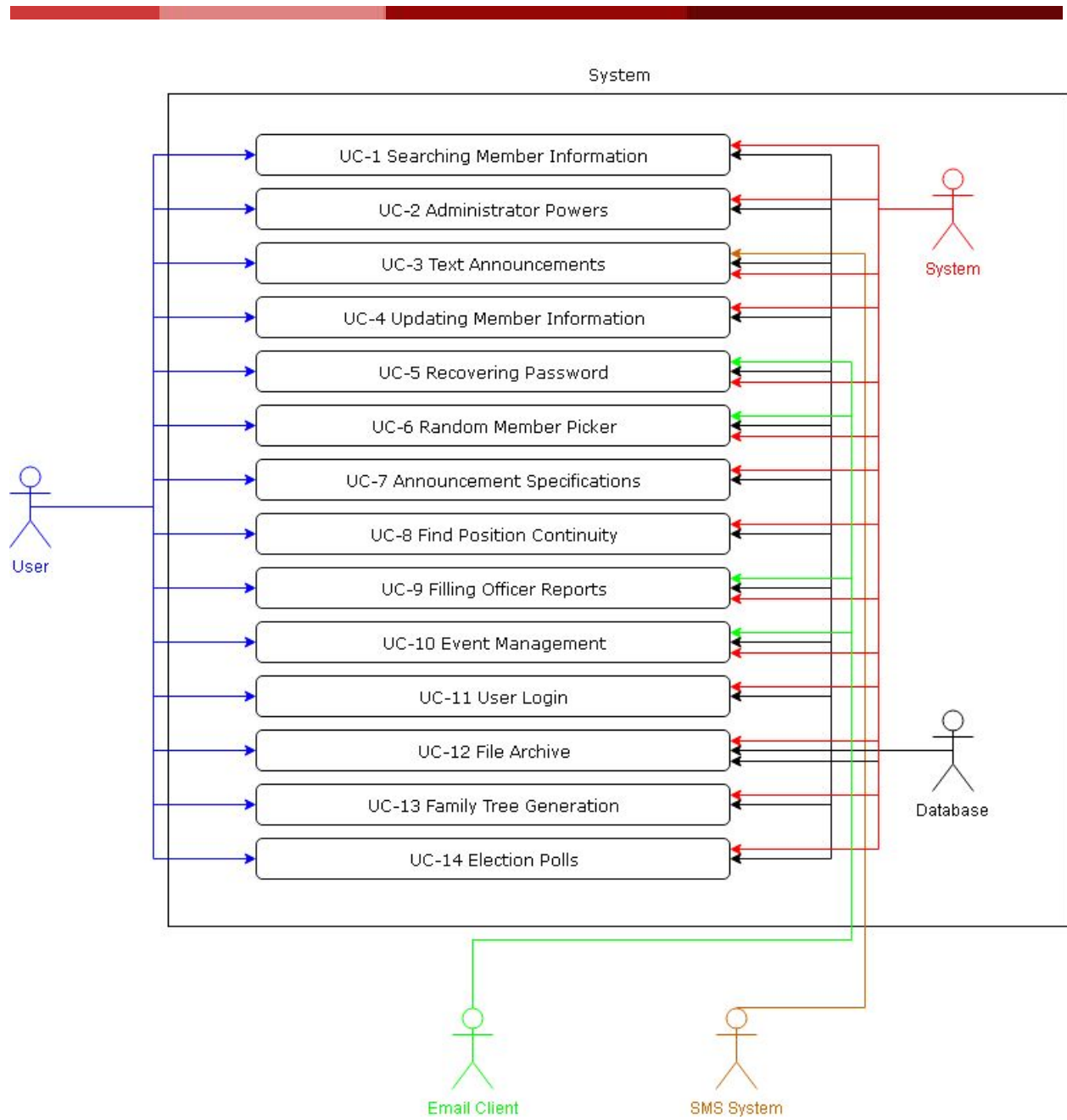
The following table shows a casual text description of each Use Case. Use Cases marked with a (\*) will not be implemented by the final demo. They could still be considered for future work.

Use Case	Name	Description
UC-1	Searching Member Information	Users can query a database, based on the parameters set by the user. Any member will have read access to all information in the database. The user can choose to generate a spreadsheet from the results of this query. The searchable information will include: Employment history, fraternity position history, campus involvement, phone numbers, and other personal information about members.
UC-2	Using Administrator Powers	Administrative users can change member information and add new members.
UC-3	Creating a Text Announcement	Users can send urgent announcements via text message from the System to all active members
UC-4	Updating Member Information	Users can update their information in the Database on the system
UC-5	Recovering Password	Users can recover their account after forgotten password.
UC-6*	Picking Random Members	Users can randomly choose members from the Database for extra duties.
UC-7	Creating and Viewing Announcements	Users can access a filterable announcements feed with topics and create an announcement.



UC-8*	Finding Position Information	Users who have a position will be able to access transition material and the description of their position responsibilities.
UC-9*	Submitting Officer Reports	Users can send in weekly officer reports using the System.
UC-10	Managing Events	Users can create an event that users can RSVP to and the event creator can send email reminders to the RSVP-ed users. Created events will be linked to the fraternity Calendar Application.
UC-11	Logging into the System	Users can login to the site with their unique Username and Password.
UC-12*	Accessing File Archive	Users will have access to a organized fraternity file archive.
UC-13*	Generation Family Tree	Users can generate a graphical fraternity family lineage by choosing a member.
UC-14*	Voting in Elections	Users can vote in position elections on the System.

### 3.3.2 Use Case Diagram



### 3.3.3 Traceability Matrix

The Traceability Matrix allows the reader to cross the functional and non-functional requirements described earlier with the use cases. The importance of each Use Case is determined by the number of requirements associated with it and the average priority of its associated requirements. Lower priority weights are given to requirements that are more essential to the project. Non-functional requirements (REQ-19,20,21) are not taken into account when calculating these values. The tie-breaker for Use Cases will be if either is the only Use Case associated with a specific requirement e.g. UC-11 is the only Use Case associated with REQ-2 so it is given precedence over UC-9.

*Note: Priority is represented as (!)*

*Note: Use Cases marked with a (\*) will not be implemented by the final demo.*

	!	UC - 1	UC - 2	UC - 3	UC - 4	UC - 5	UC - 6*	UC - 7	UC - 8*	UC - 9*	UC - 10	UC -11	UC -12 *	UC -13 *	UC -14 *
REQ-1	1	X								X	X			X	
REQ-2	1											X			
REQ-3	1		X												
REQ-4	1			X											
REQ-5	2		X							X		X			
REQ-6	2				X										
REQ-7	2	X													
REQ-8	2			X							X				
REQ-9	2							X							
REQ-10	3	X							X			X			
REQ-11	3					X									
REQ-12	3						X								

<i>REQ-1</i> <i>3</i>	3										X				
<i>REQ-1</i> <i>4</i>	4										X				
<i>REQ-1</i> <i>5</i>	4												X		
<i>REQ-1</i> <i>6</i>	5													X	
<i>REQ-1</i> <i>7</i>	5														X
<i>REQ-1</i> <i>8</i>	3									X					
<i>REQ-1</i> <i>9</i>	1	X			X		X		X		X	X			
<i>REQ-2</i> <i>0</i>	1			X	X	X	X	X	X	X	X	X			X
<i>REQ-2</i> <i>1</i>	2	X	X	X	X	X	X	X	X	X	X	X	X	X	X
<i>Min</i> <i>PW</i>		1	1	1	2	3	3	2	3	1	1	1	4	1	5
<i>Max</i> <i>PW</i>		3	2	2	2	3	3	2	3	3	4	3	4	5	5
<i>Total</i> <i>PW</i>		6	3	3	2	3	3	2	3	6	10	6	4	6	5
<i>Total</i> <i>REQs</i>		3	2	2	1	1	1	1	1	3	4	3	1	2	1
<i>Avg.</i> <i>PW</i>		2	1.5	1.5	2	3	3	2	3	2	2.5	2	4	3	5

### 3.3.4 Fully-Dressed Description

Use Case UC-1: Searching Member Information
<b>Related Requirements:</b> REQ-1, REQ-7, REQ-10, REQ-19, REQ-21, REQ-22  <b>Initiating Actor:</b> User  <b>Participating Actors:</b> System, Database

**Actor's Goal:** Display information from the Database

**Preconditions:**

- User must be logged into the Greek Life Member Management System.
- The relevant information must be recorded in the Database.

**Post-Conditions:**

- The information searched for by the User will be displayed in tabular form.
- The User has the option to download the results as a Microsoft Excel spreadsheet.

**Flow of Events for Main Success Scenario:**

→ User selects the “Member Information” page from the System navigation bar.  
← The system displays the “Member Information” page to the User.  
→ On the “Member Information” page, the user selects the option to “Search Information.”  
← The system provides the user with a form containing options on what to search for and what filters to apply..  
→ User makes their selections and presses the “Search” button.  
← The System generates an SQL query from the User inputs and sends it to the Database. The results are displayed to the User in tabular form.

**Flow of Events for Alternate Success Scenario (Spreadsheet Generation):**

→ User selects the “Member Information” page from the System navigation bar.  
← The system displays the “Member Information” page to the User.  
→ On the “Member Information” page, the user selects the option to “Search Information.”  
← The system provides the user with a form containing options on what to search for and what filters to apply..  
→ User makes their selections and presses the “Search” button.  
← The System generates an SQL query from the User inputs and sends it to the Database. The results are displayed to the User in tabular form.  
→ User then presses the “Generate Spreadsheet” button.  
← The System creates a Microsoft Excel spreadsheet and offers it for download for the user.

**Use Case UC-2: Using Administrator Powers**

**Related Requirements:** REQ-3, REQ-5, REQ-21, REQ-22

**Initiating Actor:** User (Administrator)

**Participating Actors:** System, Database, Email Server

**Actor's Goal:** To add new Users to the System or update Users' information.

**Preconditions:**

- The User must be logged into the Greek Life Member Management System.
- The User must have Administrative access.

### **Post-Conditions:**

- The initiating User will be displayed a message indicating successful completion of their action and a description of what was changed.
  - A new User will be recorded in the Database.
  - The new User will receive an email containing their Username and Password and instructions on how to access the System.
- OR
- The chosen User's information will be updated in the Database.

### **Flow of Events for Main Success Scenario:**

- User selects the "Admin" page from the System navigation bar.
- ← The system displays the "Admin" page to the User.
- On the "Admin" page, the user selects the option to "Create a New User."
- ← The system provides the user with a form to enter the new user's pawprint and email.
- User enters the pawprint and email of the new user.
- ← The System displays the information to the user for review.
- User approves the creation or edits it.
- ← Once approved, the System creates an email going to the new user containing their system-generated password and instructions on how to access the System.
- ← The System records the new member in the Database.
- ← The email server sends the email to the new user.
- ← The system displays a success message to the User.

### **Flow of Events for Alternate Success Scenario (Update Member Information):**

- User selects the "Admin" page from the System navigation bar.
- ← The system displays the "Admin" page to the User.
- On the "Admin" page, the user selects the option to "Update Member Info."
- ← The system provides the user with a form to edit the member's information.
- User edits the relevant information.
- ← The System displays the information to the user for review.
- User approves the change or edits it.
- ← The System records the new information in the Database.
- ← The system displays a success message to the User.

### **Flow of Events for Alternate Success Scenario (Cancel Member Creation):**

- User selects the "Admin" page from the System navigation bar.
- ← The system displays the "Admin" page to the User.
- On the "Admin" page, the user selects the option to "Create a New User."
- ← The system provides the user with a form to enter the new user's pawprint and email.
- User enters the pawprint and email of the new user.
- ← The System displays the information to the user for review.
- User presses the "Cancel" button.
- ← The system exits the form and displays a cancel message to the User.

### Use Case UC-3: Creating a Text Announcement

**Related Requirements:** REQ-4, REQ-8, REQ-20, REQ-21, REQ-22

**Initiating Actor:** User

**Participating Actors:** System, Email Server, SMS Gateway

**Actor's Goal:** To send a text message announcement to all active Fraternity members.

**Preconditions:**

- User must be logged into the System.
- Active members must have provided their phone numbers and mobile carriers to the Database.

**Post-Conditions:**

- The initiating User will be displayed a message indicating successful completion of their action.
- Each current member of the fraternity will receive a SMS message containing the announcement.

**Flow of Events for Main Success Scenario:**

- User selects the “Announcements” page from the System navigation bar.
- ← The system displays the “Announcements” page to the User.
- On the “Announcements” page, the user selects the option to “Create a New Announcement.”
- ← The system provides the user with a form to enter their message.
- User enters their message, chooses its topic, and selects the “Also Send as Text” option.
- ← The System displays the message to the user for review.
- User approves the message or edits it.
- ← Once approved, the System creates an email addressed to the SMS Gateway of each active member.
- ← The System records the announcement in the Database.
- ← The email server sends the email to the SMS Gateways.
- ← The SMS Gateways send the announcement as SMS messages to the active members.
- ← The System displays a success message to the User.

**Flow of Events for Alternate Success Scenario (User Cancels message):**

- User selects the “Announcements” page from the System navigation bar
- ← The system displays the “Announcements” page to the User.
- On the “Announcements” page, the user selects the option to “Create a New Announcement.”
- ← The system provides the user with a a form to enter their message.
- User enters their message, chooses its topic, and selects the “Also Send as Text” option.
- ← The System displays the message to the user for review.
- User decides not to send the message and selects the “Cancel” button.
- ← The System deletes the message and displays a cancel message to the User.

### Use Case UC-4: Updating Member Info

**Related Requirements:** REQ-6, REQ-19, REQ-20, REQ-21

**Initiating Actor:** User

**Participating Actors:** System, Database

**Actor's Goal:** Update their information stored in the Database

**Preconditions:**

- User must be logged into the Greek Life Member Management System.
- The User must have a profile.

**Post-Conditions:**

- The updated information will be stored in the Database.
- The User will be displayed a success or failure message.

**Flow of Events for Main Success Scenario:**

- User selects the “Your Profile” page from the System navigation bar.
- ← The system displays the “Profile” page to the User.
- User selects the “Edit Info” button on the profile page.
- ← The system displays the “Edit Profile” page to the User with their current info prepopulated into the edit form.
- On the “Edit Profile” page, the user enters the new information into the fields they wish to change.
- ← The System validates and cleans the entered information, then updates the user’s Profile in the Database.
- ← The System redirects the user to the “Your Profile” page and displays a success message.

**Flow of Events for Alternate Success Scenario (Invalid input):**

- User selects the “Your Profile” page from the System navigation bar.
- ← The system displays the “Profile” page to the User.
- User selects the “Edit Info” button on the profile page.
- ← The system displays the “Edit Profile” page to the User with their current info prepopulated into the edit form.
- On the “Edit Profile” page, the user enters the new information into the fields they wish to change.
- ← The System attempts to validate the entered information, but it is invalid.
- ← The System re-displays the “Edit Profile” page to the user with a message describing the correct format for the failed input.

### Use Case UC-5: Recovering Password



**Related Requirements:** REQ-11, REQ-20, REQ-21

**Initiating Actor:** User

**Participating Actors:** System, Database, Email Server

**Actor's Goal:** Recover a forgotten password for their account.

**Preconditions:**

- User must have a registered email address with the System.
- The User must not be logged into the System.

**Post-Conditions:**

- The User's password will be changed to a new password of their choosing.

**Flow of Events for Main Success Scenario:**

→ User selects the "Login" option from the System navigation bar.  
← The system displays the "Login" page to the User.  
→ On the "Login" page, the user selects the "Lost Password?" option.  
← The system displays the "Password Reset Email" page.  
→ User enters their registered email and clicks the "Password Reset" button.  
← The System verifies the email is associated with an account and uses the Email Server to send an email with a password reset link. Then displays a message to the user indicating these actions.  
→ User follows the link in their email.  
← The System displays the "Password Reset" page with a form to enter a new password.  
→ User enters their new password into the "New Password" and "Confirm Password" inputs and clicks the "Save" button.  
← The System validates the User's password and saves it into the Database. The system redirects the User to the "Home" page and displays a success message.

**Flow of Events for Alternate Success Scenario (Incorrect Email):**

→ User selects the "Login" option from the System navigation bar.  
← The system displays the "Login" page to the User.  
→ On the "Login" page, the user selects the Lost Password?" option.  
← The system displays the "Password Reset Email" page.  
→ User enters their registered email and clicks the "Password Reset" button.  
← The System sees the email is not associated with an account and displays an error message to the User.  
→ User retries to enter their account's email address until successful.

**Use Case UC-7: Creating and Viewing Announcements**

**Related Requirements:** REQ-9, REQ-20, REQ-21

**Initiating Actor:** User

**Participating Actors:** System, Database

**Actor's Goal:** View or create Announcements in the System.

**Preconditions:**

- User must be logged into the Greek Life Member Management System.
- The User must be a member of the Executive Board or an admin.

**Post-Conditions:**

- The new announcement will be recorded in the Database and members will have received a text message with the contents of the Announcement if it was marked Urgent.
- The announcements recorded in the Database will be displayed for the User to view.

**Flow of Events for Main Success Scenario:**

→ User selects the “Announcement” page from the System navigation bar.  
← The system displays the “Announcement” page to the User with a list current announcements.  
→ On the “Announcements” page, the user selects the option to filter by the “Social” topic.  
← The System updates the list to contain only announcements of the type “Social” and displays the “Announcements” page.  
→ On the “Announcements” page, the user selects an announcement.  
← The System displays the “Announcement Detail” page for the selected announcement.

**Flow of Events for Alternate Success Scenario (Create Announcement Normal):**

→ User selects the “Announcement” page from the System navigation bar.  
← The system displays the “Announcement” page to the User with a list current announcements.  
→ On the “Announcements” page, the user selects the “Create Announcement” button.  
← The System displays the “Create Announcement” form.  
→ The User enters the title, topic, message, and chooses the “Normal” type. Then, the User presses the “Create” button.  
← The System validates and cleans the input, then creates the new announcement in the Database.  
← The System displays the “Announcement Detail” page for the announcement that was just created with a success message.

**Use Case UC-10: Managing Events**

**Related Requirements:** REQ-1, REQ-8, REQ-13, REQ-14, REQ-19, REQ-20, REQ-21

**Initiating Actor:** User

**Participating Actors:** System, Database, Email Server, Calendar App

**Actor's Goal:** Display information from the Database

**Preconditions:**

- User must be logged into the Greek Life Member Management System.

**Post-Conditions:**

- The event created by the User is recorded in the Database, the Calendar Application
- The User has RSVP-ed to the event and it is recorded in the Database.
- The RSVP-ed Users receive an email reminder the day before the event.

**Flow of Events for Main Success Scenario:**

→ User selects the “Events” page from the System navigation bar.  
← The system displays the “Events” page to the User which shows upcoming events.  
→ On the “Events” page, the user selects the option to “Create an Event”  
← The system provides the user with the create event form.  
→ User enters the required inputs into the form and clicks the “Create” button.  
← The System validates and cleans the input and creates the event in the Database.  
← The System sends a command to the Calendar Application to add the event to the Fraternity Calendar.  
← The System displays the “Event Detail” page for the event that was just created with a success message.

**Flow of Events for Alternate Success Scenario (User RSVP):**

→ User selects the “Events” page from the System navigation bar.  
← The system displays the “Events” page to the User which shows upcoming events.  
→ On the “Member Information” page, the user selects an event from the list.  
← The system displays the “Event Detail” page for the chosen event.  
→ User presses the “RSVP” button.  
← The System records the user as RSVP-ed in the Database and displays a success message.  
← The Email Server sends the RSVP-ed users a reminder email the day before the event.

**Use Case UC-11: Logging into the System**

**Related Requirements:** REQ-2, REQ-5, REQ-10, REQ-19, REQ-20, REQ-21, REQ-22

**Initiating Actor:** All Users

**Participating Actors:** System, Database

**Actor's Goal:** Gain access to the Greek Life Member Management System.

**Preconditions:**

- The user must have had an account created for them by an administrator who provided the user with a Username and Password.

- The user must know their Username and Password
- The user must be able to access the Login Page over the Internet.

**Post-Conditions:**

- The user has access to the Greek Life Member Management System until they log out.
- The user's position and name are retrieved from the database .
- If the user is a privileged user (position holder, administrator), they have access to additional features.

**Flow of Events for Main Success Scenario:**

- User connects to the Login page of the Greek Life Member Management System over https.
- User enters their Username and Password into the fields and presses the login button.
- ← The system uses a hash function on the password and queries the Database for the user.
- ← The Username and hashed Password are found in the Database; the user is granted access and redirected to the homepage.
- ← The System records the Login in the Database.

**Flow of Events for Alternate Success Scenario (Unknown User Denied Access):**

- User connects to the Login page of the Greek Life Member Management System over https.
- User enters their Username and Password into the fields and presses the login button.
- ← The system uses a hash function on the password and queries the Database for the user.
- ← The Username and hashed Password are not found in the Database, the user is displayed an error message saying their Username or Password is incorrect. They are not granted access to the System.

## 3.4 System Sequence Diagrams

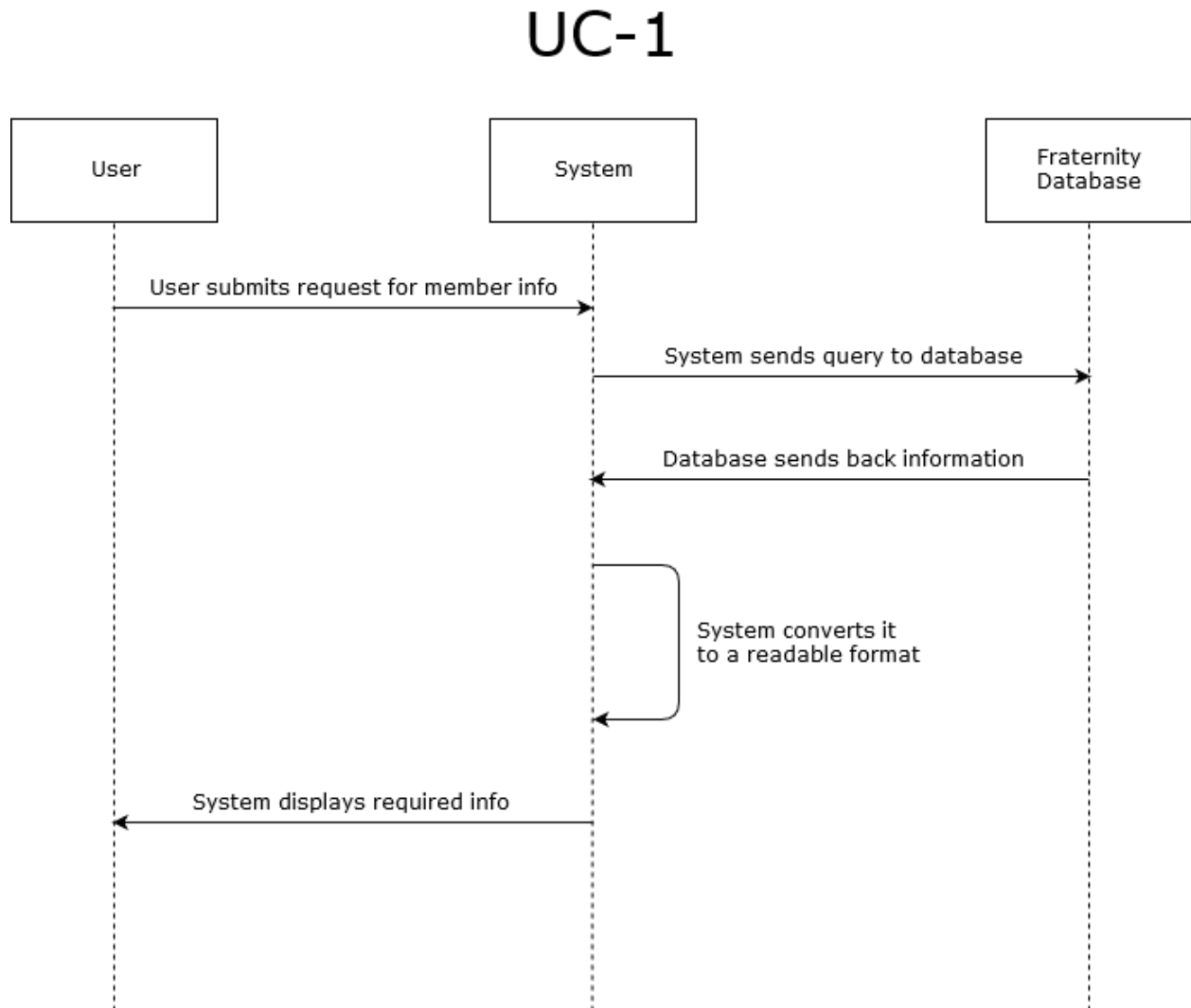


Figure 3.4.1 - Sequence Diagram of Use Case 1

Figure 3.4.1 shows the sequence diagram of Use Case 1 which is “Searching Member Information”. The objective in this diagram is to easily view data that is dynamically retrieved by the system, working hand in hand with the fraternity database. It is important because of the integral role that certain groups of members play within internal operations. The diagram corresponds to the Main Success Scenario.

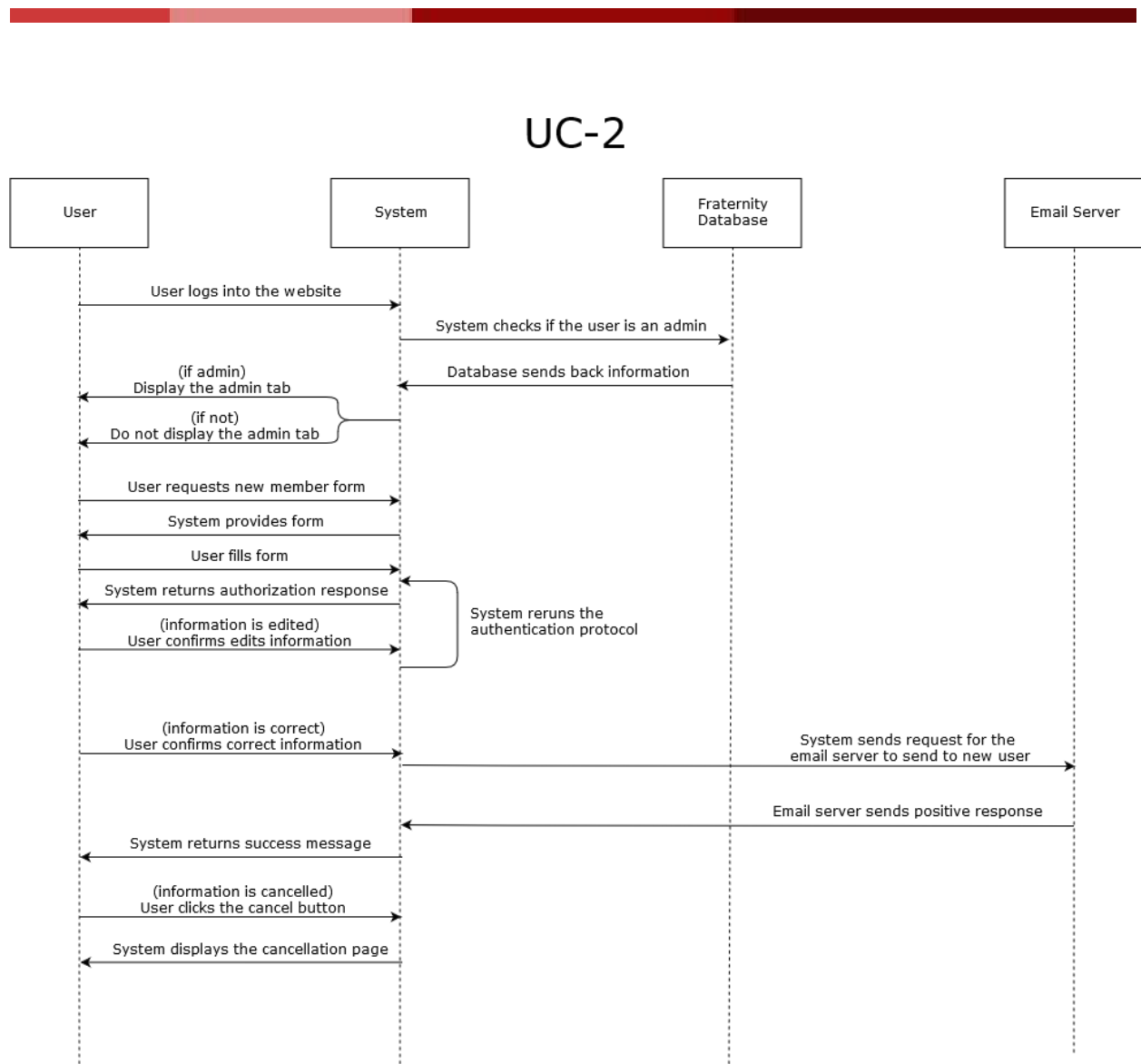


Figure 3.4.2 - Sequence Diagram of Use Case 2

Figure 3.4.2 shows the sequence diagram of Use Case 2 which is “Using Administrator Powers”. The objective of this diagram is to layout many possible outcomes of an admin only form. This use case relies on not only the fraternity database but also the email server to send the final confirmation message that the user has been sent the email. The diagram corresponds to both Main and Alternate Success Scenarios.

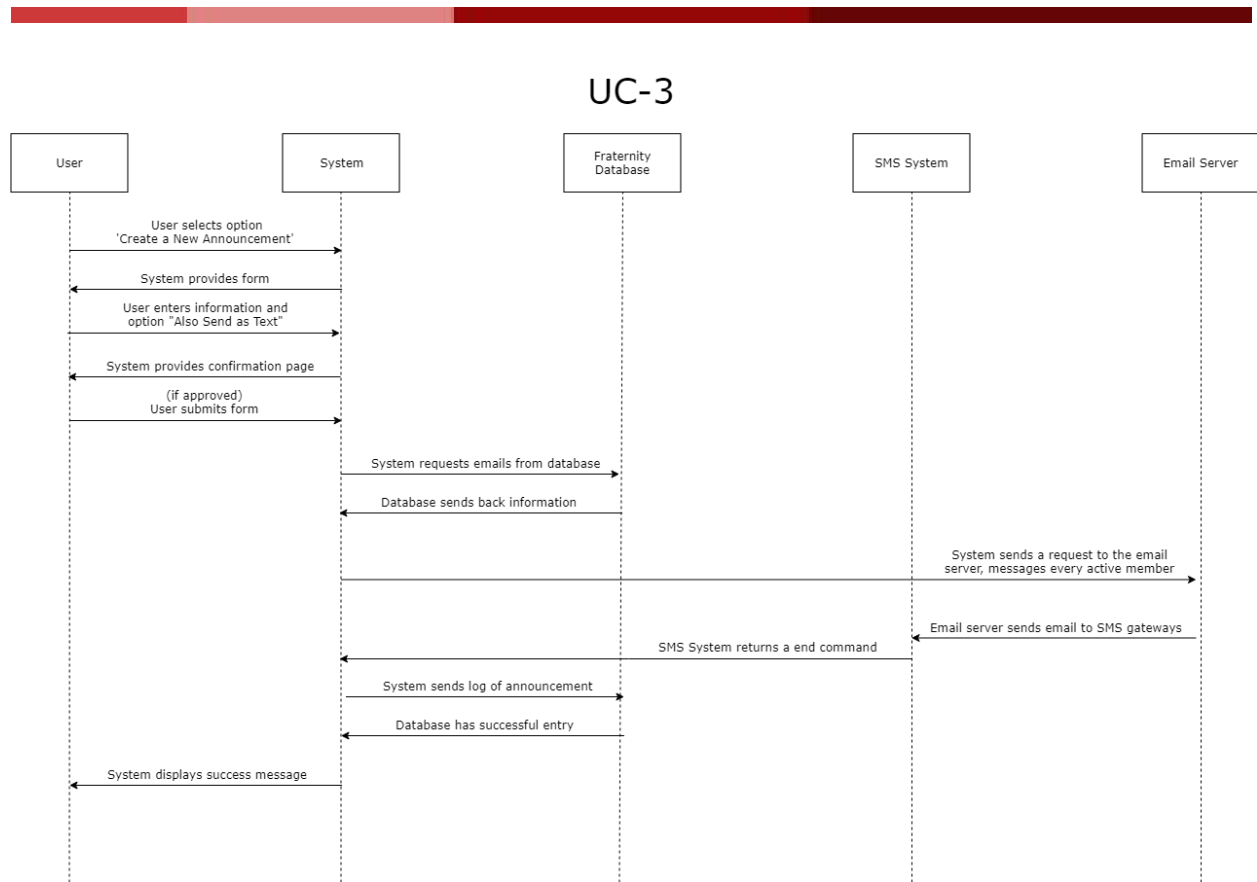


Figure 3.4.3 - Sequence Diagram of Use Case 3

Figure 3.4.3 shows the sequence diagram of Use Case 3 which is “Creating a Text Announcements”. The objective of this diagram is to create a trail of how the group plans on accomplishing the task of system input to mass text feature. This feature would be used mainly to send very important announcements and would be an executive board exclusive feature. The diagram corresponds to both Main and Alternate Success Scenarios.

## UC-4

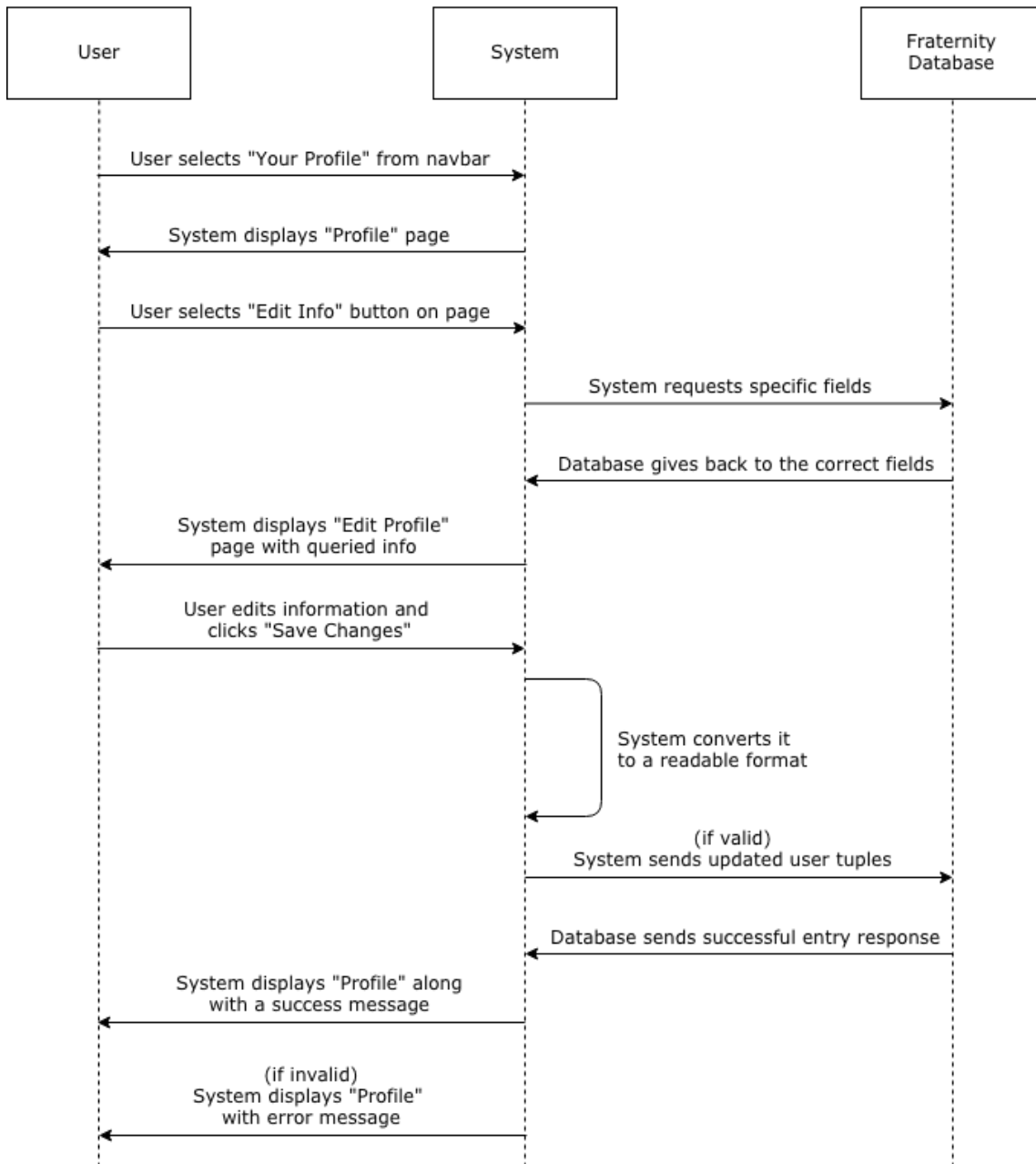


Figure 3.4.4 - Sequence Diagram of Use Case 4

Figure 3.4.4 shows the sequence diagram of Use Case 4 which is “Updating Member Info”. The objective of this diagram is to layout the possibilities of editing a user’s profile. This use case relies on the system to handle the queries that the user asks for. The diagram corresponds to both Main and Alternate Success Scenarios.



## UC-5

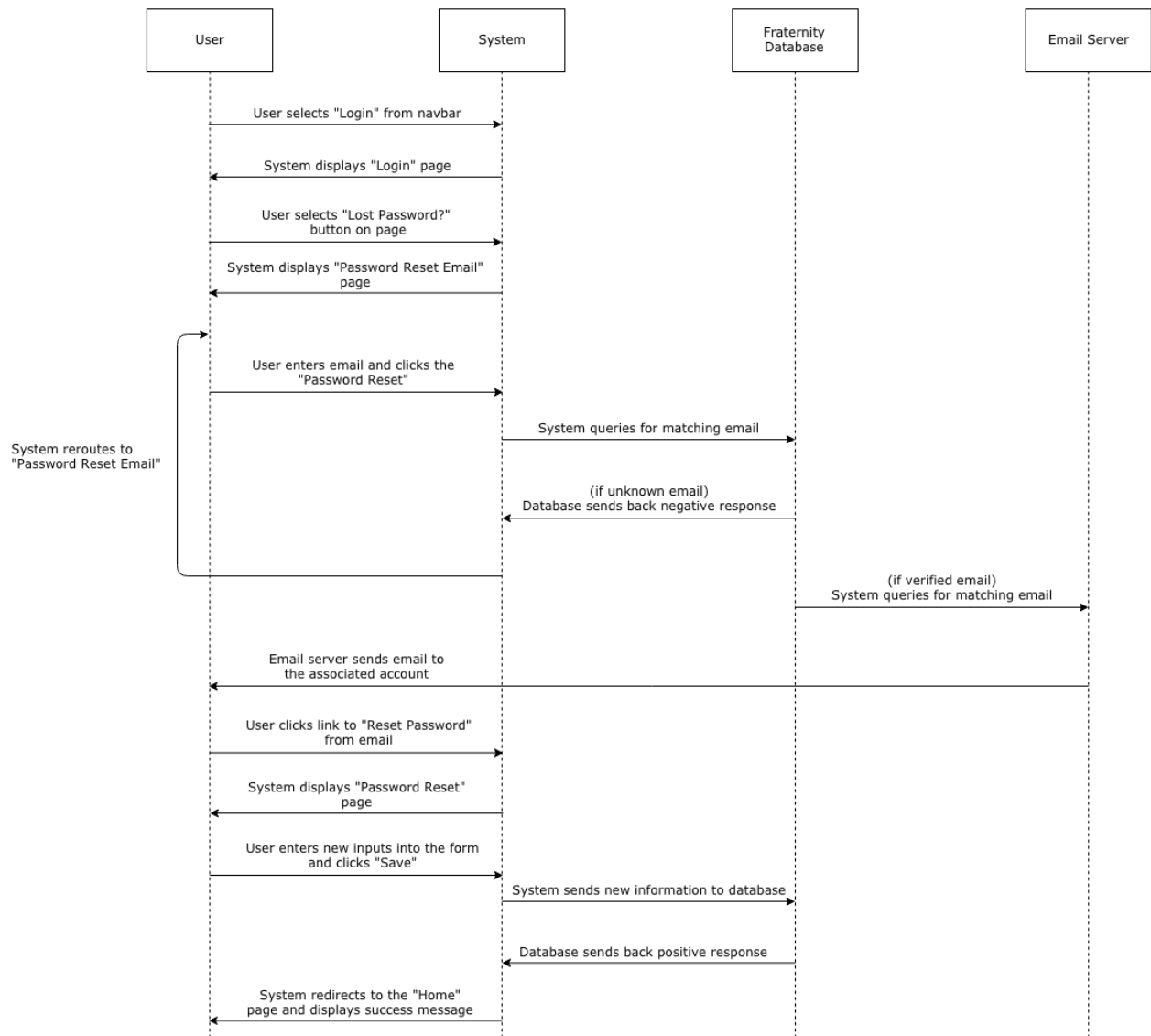


Figure 3.4.5 - Sequence Diagram of Use Case 5

Figure 3.4.5 shows the sequence diagram of Use Case 5 which is “Recovering Password”. The objective of this diagram is to display how the User uses the application to recover their password. This Use Case manipulates the database to change user information and the email server to send out the recovery email. The diagram corresponds to both Main and Alternate Success Scenarios.

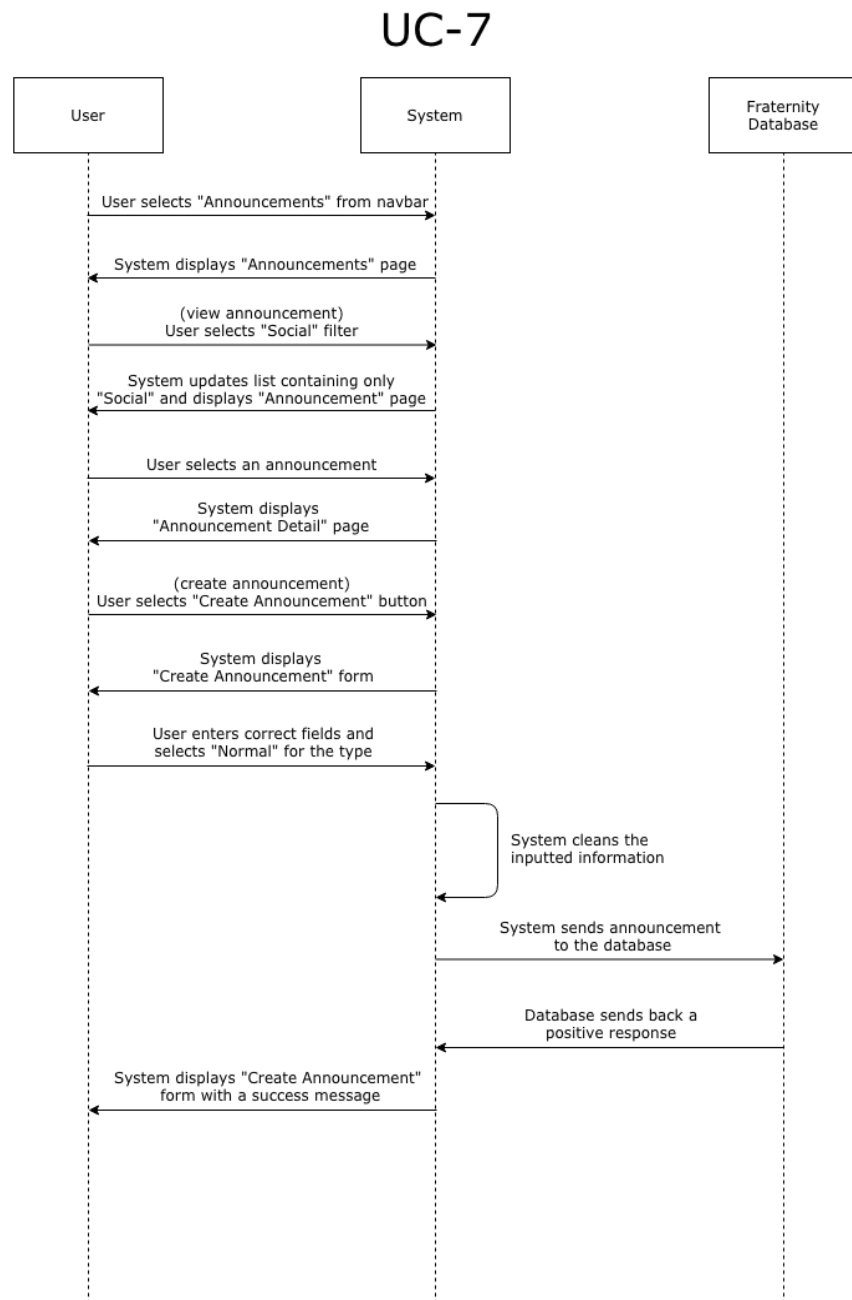


Figure 3.4.6 - Sequence Diagram of Use Case 7

Figure 3.4.6 shows the sequence diagram of Use Case 7 which is “Creating and Viewing Announcements”. The objective of this diagram is to detail the process of creating and viewing announcements, with two separate paths for viewing and creating. This use case relies on the database to properly store the data so that it is manipulated correctly. The diagram corresponds to both Main and Alternate Success Scenarios.

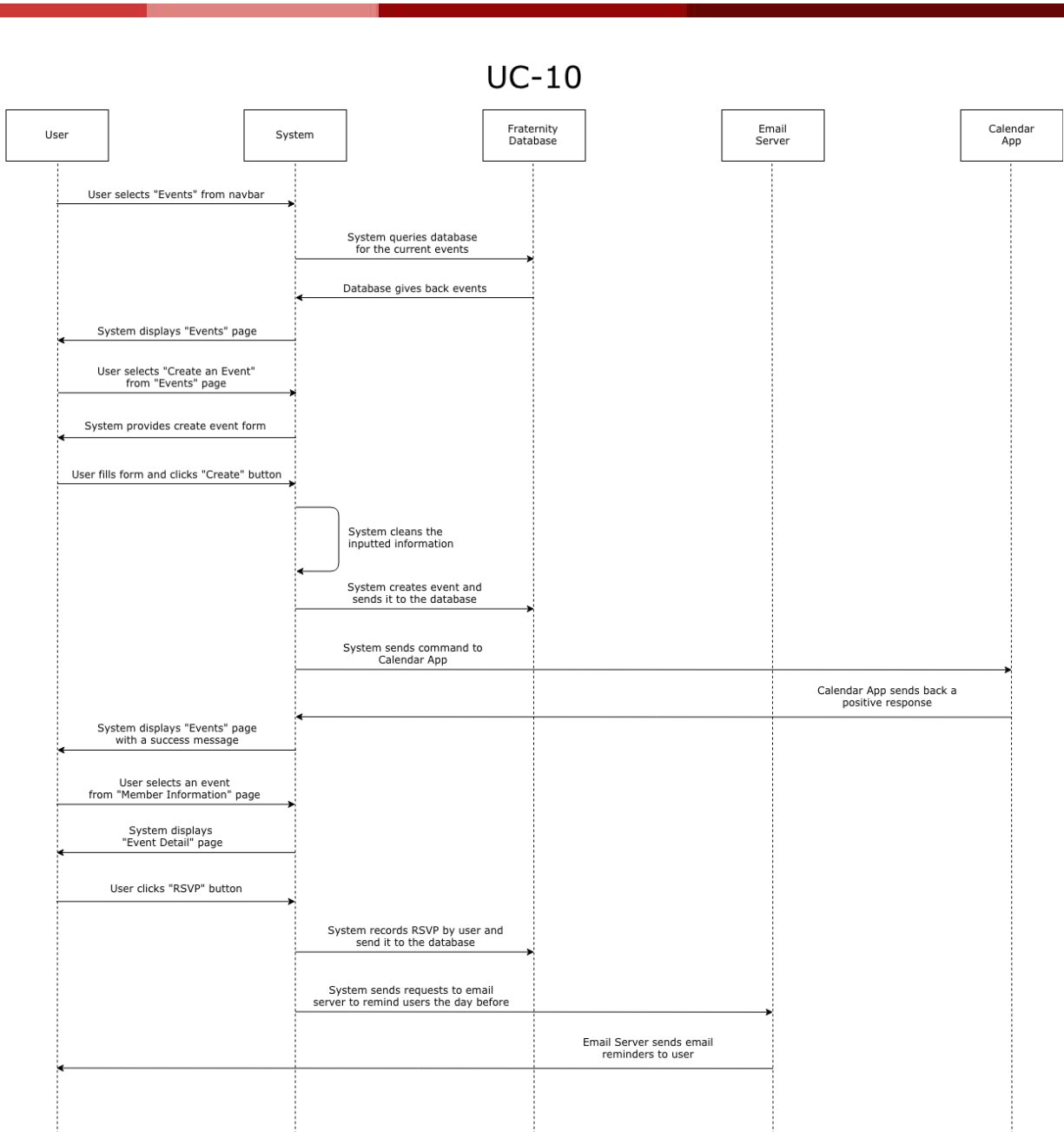


Figure 3.4.7 - Sequence Diagram of Use Case 10

Figure 3.4.7 shows the sequence diagram of Use Case 10 which is “Managing Events”. The objective of this diagram is to detail the various pathways a user will take to create and manipulate the event structure. This Use Case relies on the most systems of any Use Case diagram, with the System manipulating three separate services to move data. The diagram corresponds to both Main and Alternate Success Scenarios.

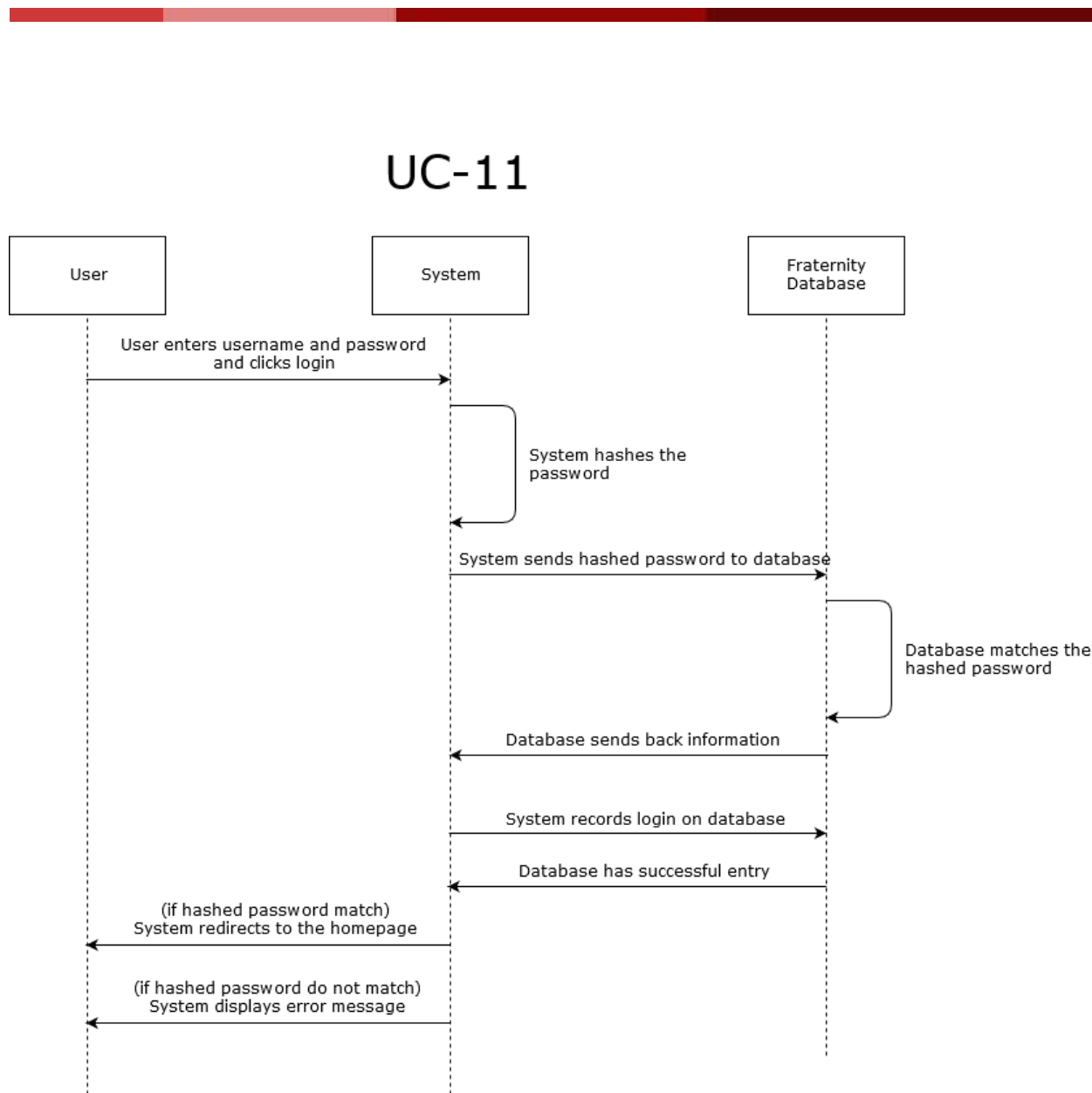


Figure 3.4.8 - Sequence Diagram of Use Case 11

Figure 3.4.8 shows the sequence diagram of Use Case 11 which is “Logging into the System”. The objective of this diagram is to explain the importance of a secure login for the project. In order to access many of the features that are explained in the system requirements, the database needs to be password protected. This ensures secure entries and the safeguarding of data. The diagram corresponds to both Main and Alternate Success Scenarios.

## 4 Effort Estimation Using Use Case Points

$$UCP = UUCP \times TCF \times ECF$$

### 4.1 UUCP: Unadjusted Use Case Points

$$UUCP = UAW + UUCW$$

Unadjusted Actor Weight (UAW): An actor in a use case can be a person, a software program or a hardware device. The weight for an actor depends on the sophistication of the interface between the actor and the system. We present the UAW for the Greek Life Member Management System:

#### 4.1.1 Unadjusted Use Case Points (UUCPs)

Unadjusted Use Case Points (UUCPs) are computed as a sum of these two components:

1. The Unadjusted Actor Weight (UAW), based on the combined complexity of all the actors in all the use cases.
2. The Unadjusted Use Case Weight (UUCW), based on the total number of activities (or steps) contained in all the use case scenarios.

Actor	Description	Complexity	Weight
Database	A place where information is stored	Average	2
User (unregistered)	Any individual who does not have an account yet	Simple	1
User (registered)	Any individual who has a basic account	Average	2
System Administrator	An individual who interacts with the system through an admin account and is responsible for maintaining and managing the system.	Complex	3
Email Server	A machine responsible for sending messages to	Average	2

	investors via E-mail and SMS.		
--	-------------------------------	--	--

Table 4-1 UAW

In accordance with the data above, the UAW is calculated as follows:

$$\text{UAW(GLMMS)} = \sum [(\text{Complexity Weight}) \times (\# \text{ of Actors associated with Complexity})]$$

$$\text{UAW(GLMMS)} = [(1) \times (1)] + [(2) \times (3)] + [(3) \times (1)] = 10$$

### 4.1.2 Unadjusted Use Case Weight (UUCW)

The complexity level of the use cases is primarily derived from the number of steps in the main success scenario. Nonetheless, the number of participating actors, and the number of steps in the alternate scenario play a considerable role as well. Below we present the UUCW for our project:

Use Case	Description	Complexity	Weight
UC-1: Searching Member Information	Simple user interface. Three participating actors. Five steps for success	Average	10
UC-2: Using Administrator Powers	Complex user interface. Four participating actors. Fifteen steps for success	Complex	15
UC-3: Creating a Text Announcement	Moderate user interface. Five participating actors. Thirteen steps for success	Average	10
UC-4: Updating Member Information	Complex user interface.	Moderate	15
UC-5: Recovering Password	Simple user interface.	Simple	5
UC-7: Creating and Viewing Announcements	Moderate user interface.	Moderate	10
UC-10: Managing	Average user	Moderate	10

Events	interface.		
UC-11: Logging into the System	Simple user interface. Three participating actors. Nine steps for success	Simple	5

Table 4-2 UUCW

Based on the data above, the UUCW is as follows:

$$\text{UUCW}(\text{GLMMS}) = \sum (\text{Complexity Weight}) \times (\# \text{ of Use Cases in Complexity category})$$

$$\text{UUCW}(\text{GLMMS}) = [(5) \times (2)] + [(10) \times (4)] + [(15) \times (2)] = 80$$

$$\text{UUCP} = \text{UUCW} + \text{UAW} = 80 + 10 = 90$$

## 4.2 TCF: Technical Complexity Factor

$$\text{TCF} = \text{Constant-1} + \text{Constant-2} \times \text{Technical Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^{13} W_i \cdot F_i$$

where,

$$\text{Constant-1 } (C_1) = 0.6$$

$$\text{Constant-2 } (C_2) = 0.01$$

Technical factors identify and estimate the impact on productivity of the overall project due to the involvement of non-functional requirements. There are thirteen standard technical factors. We present the TCF of our project below:

Technical Factor	Description	Weight	Perceived Complexity	Calculated Factor
TF-1	Distributed system	2	5	10
TF-2	Performance objectives	1	1	1
TF-3	End-user efficiency	1	4	4
TF-4	Complex internal processing	1	1	1
TF-5	Reusable design	1	3	3

	or code			
TF-6	Easy to install	0.5	0	0
TF-7	Easy to use	0.5	5	2.5
TF-8	Portable	2	2	4
TF-9	Easy to change	1	3	3
TF-10	Concurrent use	1	0	0
TF-11	Special security features	1	0	0
TF-12	Provides direct access for third parties	1	2	2
TF-13	User training	1	1	1
			Total	31.5

Based on the data above, the TCF is as follows:

$$TCF(GLMMS) = 0.6 + [(0.01) \times 31.5] = 0.915$$

### 4.3 ECF: Environmental Complexity Factor

$$ECF = \text{Constant-1} + \text{Constant-2} \times \text{Environmental Factor Total} = C_1 + C_2 \cdot \sum_{i=1}^8 W_i \cdot F_i$$

where,

$$\text{Constant-1 } (C_1) = 1.4$$

$$\text{Constant-2 } (C_2) = -0.03$$

Environmental factors identify and estimate the impact on productivity of the overall project due to the experience of the development team. There are eight standard technical factors. We present the ECF of our project below:

Environmental Factor	Description	Weight	Perceived Impact	Calculated Factor
----------------------	-------------	--------	------------------	-------------------



E1	Familiar with the development process	1.5	1	1.5
E2	Application experience	0.5	3	1.5
E3	Object-oriented experience	1	3	3
E4	Lead analyst capability	0.5	3	1.5
E5	Motivation	1	4	4
E6	Stable requirements	2	2	4
E7	Part-time staff	-1	5	-5
E8	Difficult programming language	-1	4	-4
			Total	6.5

Based on the data above, the TCF is as follows:

$$ECF(GLMMS) = 1.4 + [(-0.03) \times 6.5] = 1.205$$

Calculating the Use Case Points (UCP)

$$\text{Total UCP} = \text{UUCP} \times \text{TCF} \times \text{ECF}$$

$$= 90 \times 0.915 \times 1.205 = 99.23 = 99 \text{ UCPs}$$

## 4.4 Deriving Project Duration from Use Case Points

$$\text{Duration} = \text{UCP} \times \text{PF}$$

PF is the productivity factor. Although this is another factor that needs to be estimated, we were told to assume the Productivity Factor as 28 hours per point.

Therefore,

$$\begin{aligned} \text{Total Duration of The Greek Life Member Management System} &= 99 \times 28 = 2772 \\ \text{hours} &= 115.5 \text{ days} = 3.85 \text{ months} = 3 \text{ Months } 25.5 \text{ days (30 day months)} \end{aligned}$$

## 5 Domain Analysis

The analysis of the domain model began by extracting the domain model concepts and their responsibilities, attributes, and associations from the Use Cases defined above. The domain model diagram and tables with text descriptions are below.

### 5.1 Domain Model

This domain model diagram uses a 'smiley face' to represent a worker, a 'document' to represent an object, a 'stick figure' represents an actor, and '--' denotes an attribute.

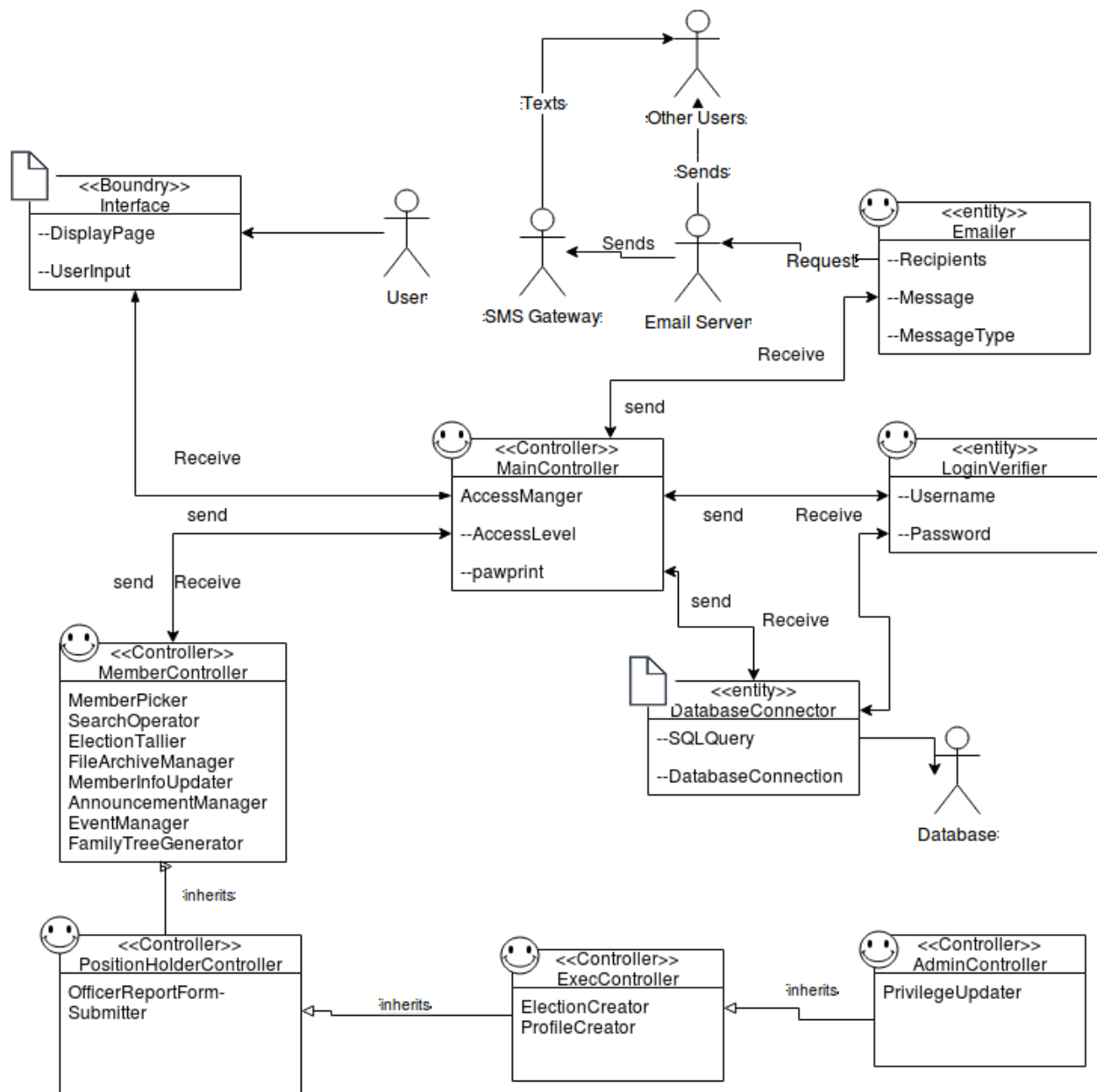


Figure 5-1: Domain Model

### 5.1.1 Concept Definitions

In the table types “D” and “K” denote doing and knowing responsibilities respectively. Also, concepts marked with a (\*) correspond to a Use Case that will not be implemented by the final demo.

Responsibility	Type	Concept
R1: Interface with the Database to store and retrieve information needed for the operation of the System.	K	DatabaseConnector
R2: Receive and pass on user input and display web pages for the user to interface with or to view results of an action.	K	Interface
R3: Coordinates all System actions. Delegates actions to appropriate controllers or concepts.	D	MainController
R4: Guards access to certain System functions based on a User’s access level.	D	AccessManager
R5: Handles all emailing required by the controllers to Users	D	Emailer
R6: Checks that username and password entered during the login process match in the Database.	D	LoginVerifier
R7: Lock system after failed login attempts	D	MemberController
R8: Randomly chooses a number of current members of the Fraternity.	D	MemberPicker*
R9: Creates an SQL query based on the desired search criteria from the user.	D	SearchOperator
R10: Handles the submission of a user’s ballot when voting in an election	D	ElectionTallier*
R11: Handles the uploading and accessing of the file archive.	D	FileArchiveManager*
R12: Handles a user’s request to update the information of a specific member in the database.	D	MemberInfoUpdater
R13: Handles all actions on the Announcement feed including, creation, searching, and displaying.	D	AnnouncementManager
R14: Handles all actions on Events feed including, creation, display, and reminders	D	EventManager
R15: Delegates actions specific to position holder	D	PositionHolderController

access level to appropriate concepts.		
R16: Handles the submission of user Officer Reports.	D	OfficerReportFormSubmitter*
R17: Delegates actions specific to the Exec access level to the appropriate concepts.	D	ExecController
R18: Creates an election based on the user's desired candidates and positions.	D	ElectionCreator*
R19: Creates a profile with the user specified username and a random password.	D	ProfileCreator
R20: Delegates actions specific to the Admin access level to the appropriate concepts.	D	AdminController
R21: Changes a user's access level.	D	PriveledgeUpdater
R22: Creates a graphical fraternity family tree.	D	FamilyTreeGenerator*

### 5.1.2 Association Definitions

Concept Pair	Association Description	Association Name
User $\leftrightarrow$ Interface	Used to receive input from the user and display info from the system.	InfoDisplay
Interface $\leftrightarrow$ MainController	Transmit data from the interface to the main controller.	SendData
MainController $\leftrightarrow$ MemberController	Grants access to basic member functions.	FunctionalityLevel1
MemberController $\leftrightarrow$ PositionHolderController	Inherits functionality from the level above it and allows access to more functionality.	FunctionalityLevel2
PositionHolderController $\leftrightarrow$ ExecController	Inherits functionality from the level above it and allows access to more functionality.	FunctionalityLevel3
ExecController $\leftrightarrow$ AdminController	Inherits functionality from the level above it and allows access to more functionality.	FunctionalityLevel4

MainController<=>Emailer	Send information to be emailed or texted to other users.	SendMessage
Emailer<=>EmailServer	Transmits info from the server to be emailed to other users.	EmailTransmitter
EmailServer<=>SMSTGateway	Sends email to phone carrier to be converted to text message.	TextTransmitter
EmailServer<=>OtherUsers	Sends email to other users.	Emailer
SMSTGateway<=>OtherUsers	Sends text message to other users.	Texter
MainController<=>LoginVerifier	Transmit data to be verified for login attempts.	VerifyInfo
MainController<=>DatabaseConnector	Sends requests and gets data from SQL queries.	QueryTransfer
LoginVerifier<=>DatabaseConnector	Sends requests for login attempt verification.	LoginTransfer
DatabaseConnector<=>Database	Sends and receives information from the database.	DataRetrieval

### 5.1.3 Attribute Definitions

Concept	Attribute	Attribute Description
Emailer	Recipients	A list of people that are drawn from either the form or list, queried to a database.
	Message	A string text that contains information submitted via form.
	MessageType	Selected from a list, it will determine if the message will also be send via SMS.
LoginVerifier	Username	A value pulled from a database corresponding to a

		password.
	Password	A value pulled from a database corresponding to a username.
MainController	AccessLevel	A field determining the interface of the user.
	Pawprint	A string of numbers and letters, referencing the Mizzou email system.
DatabaseConnector	SQLQuery	The list of commands sent to the database based on the input of the user.
	DatabaseConnection	A string to validate the connection to the database.
Interface	DisplayPage	The current Interface page that needs to be displayed to the user
	UserInput	The Input that the User entered into the forms on the displayed page.

### 5.1.4 Traceability Matrix

The traceability matrix describes the relationship between the domain concepts and the Use Cases implemented in the project. It demonstrates how the Use Cases may contain many domain concepts as well as the fact that a domain concept can be used by more than one Use Case.

*Note: Use Cases marked with a (\*) will not be implemented by the final demo.*

Domain Model	UC - 1	UC - 2	UC - 3	UC - 4	UC - 5	UC - 6*	UC - 7	UC - 8*	UC - 9*	UC - 10	UC -11	UC -12 *	UC -13 *	UC -14 *
DatabaseConnector	X	X	X	X	X	X	X	X	X	X	X		X	X

Interface	X	X	X	X	X	X	X	X	X	X	X	X	X	X
MainController	X	X	X	X	X	X	X	X	X	X	X	X	X	X
AccessManager	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Emailer		X			X	X	X		X	X				
LoginVerifier					X						X			
MemberController	X			X		X	X			X		X	X	X
MemberPicker						X								
SearchOperator	X													
ElectionTaller														X
FileArchiveManager								X				X		
MemberInfoUpdater		X		X										
AnnouncementManager		X	X				X							
EventManager										X				

FamilyTreeGenerator													X	
PositionHolderController								X	X					
OfficerReportFormSubmitter									X					
ExecController		X	X											
ElectionCreator														X
ProfileCreator		X												
AdminController		X												
PrivilegeUpdater		X												

## 5.2 System Operations Contracts

### DatabaseConnector

PRECONDITION: DatabaseConnection has been established with the Database, and SQLQuery has been provided by the MainController.

POSTCONDITION: SQLQuery has been executed and results from the Database have been passed to the MainController.

### Interface

PRECONDITION: DisplayPage is being shown to the user.

POSTCONDITION: UserInput is collected from the inputs on the DisplayPage.





## **AccessManger**

PRECONDITION: AccessLevel and Pawprint needs to be set from the results of the LoginVerifier for each logged in user.

POSTCONDITION: AccessLevel and Pawprint entries assigned to the user are cleared when the user logs out of the system.

## **SearchOperator**

PRECONDITION: The interface must be displaying the “Member Information” page.

POSTCONDITION: The SearchOperator generates an SQL statement from the user inputs to be given to the DatabaseConnector and the results will be displayed to the user by the Interface.

## **Emailer**

PRECONDITION: The emailer is given the information for Recipients of the email, the Message of the email, and the MessageType which is text or email.

POSTCONDITION: The Emailer has sent the correct type of message to the Email Server and the Other Users will receive either a text or an email.

## **MemberInfoUpdater**

PRECONDITION: The interface must be displaying the “Update Member Information” page.

POSTCONDITION: The MemberInfoUpdater creates the correct SQL statement to give to the DatabaseConnector to update the desired information in the Database.

## **ProfileCreator**

PRECONDITION: The interface must be displaying the “Create New User” page and the entered pawprint must not already exist as a profile.

POSTCONDITION: A new profile is recorded in the Database and the new user receives an email.

## **PriviledgeUpdater**

PRECONDITION: The interface must be displaying the “Update Member Information” page and the user must have Admin access.

POSTCONDITION: The MemberInfoUpdater creates the correct SQL statement to give to the DatabaseConnector to update the desired user’s access level in the Database.



## **AnnouncementManger**

PRECONDITION: The interface must be displaying the “Announcements” page and the user must have Exec access level.

POSTCONDITION: A new Event is created and added to the Database.

## **LoginVerifier**

PRECONDITION: Username and Password must be set from the user input on the “Login” page displayed by the Interface.

POSTCONDITION: The user is granted access to the system if their username and password match that of an entry in the Database and their pawprint and access level are given to the AccessManager. If they do not match, then the user is not granted access beyond the “Login” page and an error message is displayed by the Interface.

## 6 Interaction Diagrams

Note: items in **Bold** are Objects and items in *italics* are methods.

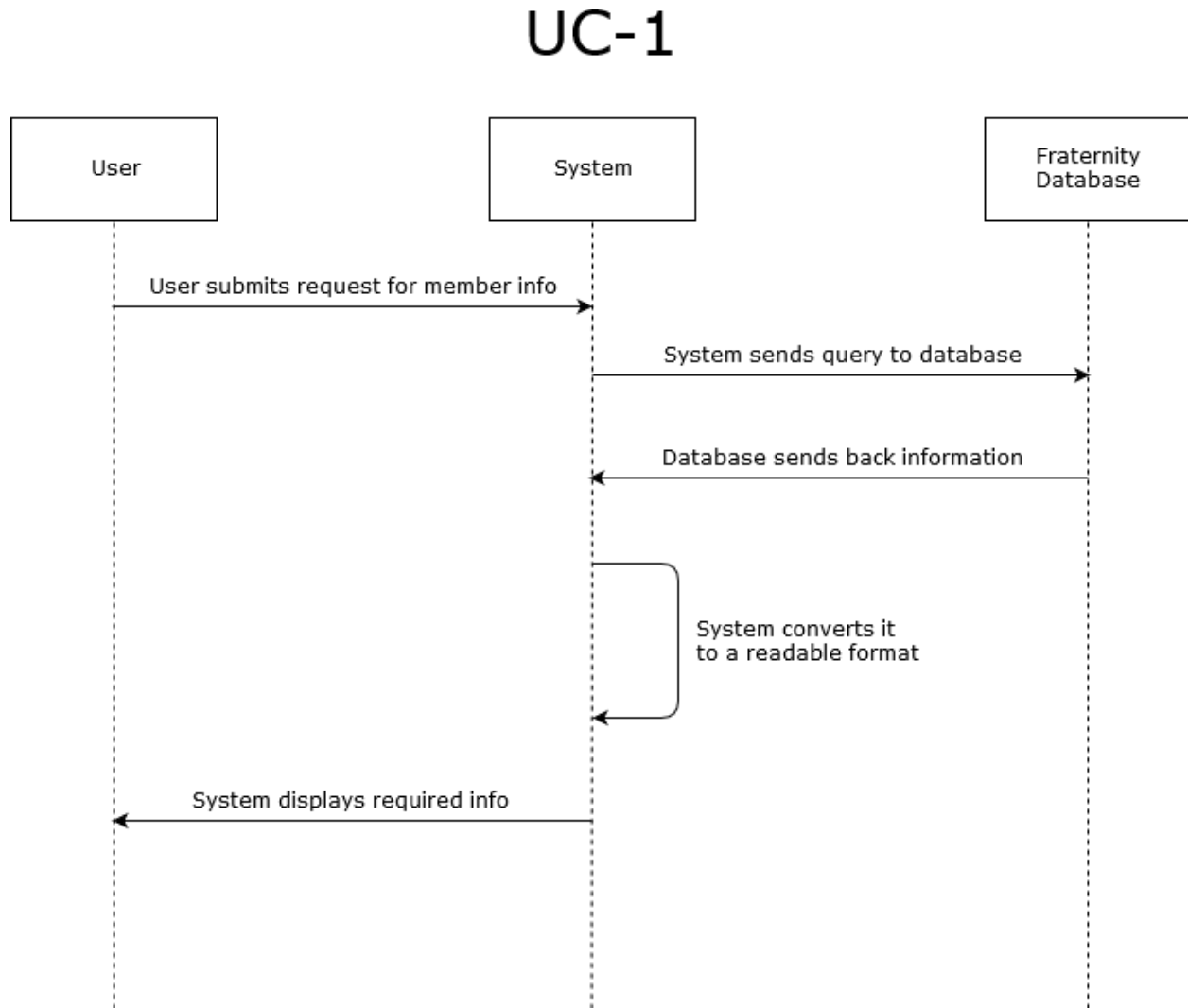


Figure 6.1 System Sequence Diagram for Use Case 1

Figure 6.1 shows the sequence diagram of Use Case 1 which is “Searching Member Information”. The goal of the Use Case is to allow the User to search for information about one or more members and display it to the user. They also have to option to download the results as a spreadsheet. The diagram shows the main success scenario, but the main and alternate are discussed here. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

When the User is on the “Member Information Search” interface page, they enter their criteria into the form and click “Submit.” The **Interface** collects this input and invokes

*serve\_request(request, url)* on the **Controller** (High Cohesion Principle) . The **Controller** invokes *respond\_to\_request(request)* on the **MemberViews** (High Cohesion Principle) . The **MemberViews** executes its *member\_search(request)* which then invokes the *select(from, fields, options)* method of the persistent **Model** (Knower) for the requested information (Low Coupling Principle). Upon receipt of the information, the **MemberViews** sends a template and the information to the **Renderer** which executes *render(template, context)* which generates and returns a web page to the **MemberViews** (High Cohesion Principle). The **MemberViews** then returns the web page to the **Interface** where it is displayed to the User (High Cohesion Principle). The User can then decide to click “Generate Spreadsheet.” The **Interface** (hiCoPri) then communicates this request to the **Controller**. The **Controller** then communicates the request to the **MemberViews** (High Cohesion Principle). The **MemberViews** then executes *generate\_spreadsheet(request)* which converts the requested member information into a spreadsheet and returns it to the **Interface** where its given to the User.

## UC-2

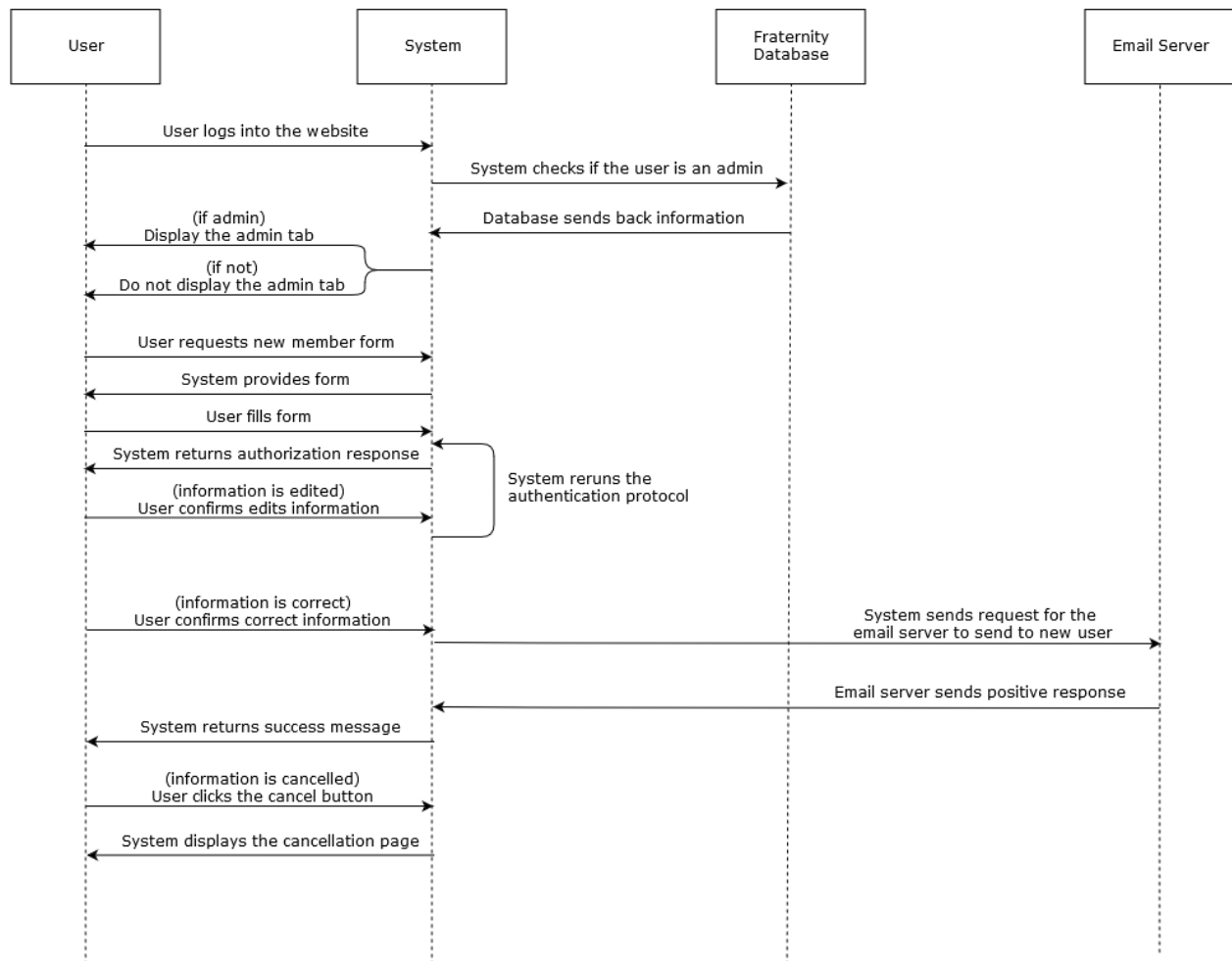


Figure 6.2 System Sequence Diagram for Use Case 2

Figure 6.2 shows the sequence diagram of Use Case 2 which is “Using Administrator Powers”. The objective of this diagram is to layout many possible outcomes of an admin only form. The diagram shows the main and alternate success scenarios and they are discussed below. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

When the Admin is on the “Add New Member” interface page, they enter the required information into the form and click “Submit.” The **Interface** shows the Admin a confirmation page and the Admin either clicks “Confirm” or they are returned to the first step. The **Interface** collects this input and invokes *serve\_request(request)* on the **Controller** (High Cohesion Principle) . The **Controller** invokes *respond\_to\_request(request)* on the **AdminViews** (High Cohesion Principle). The **AdminViews** executes its *member\_create(request)* which then performs a *insert(in, fields)* into the persistent **Model** (Knower) adding the member to the Database (Low Coupling Principle). The method *member\_create(request)* also invokes the *send\_email(recipients, title, message)* on the **Emailer** which sends an email to the desired recipients (High Cohesion Principle). Upon completion of the action, the **AdminViews** sends a template and the success or failure to the **Renderer** which executes *render(template, context)* which generates and returns a web page to the **AdminViews** (High Cohesion Principle). The **AdminViews** then returns the web page to the **Interface** completing its request where it is displayed to the Admin (High Cohesion Principle).

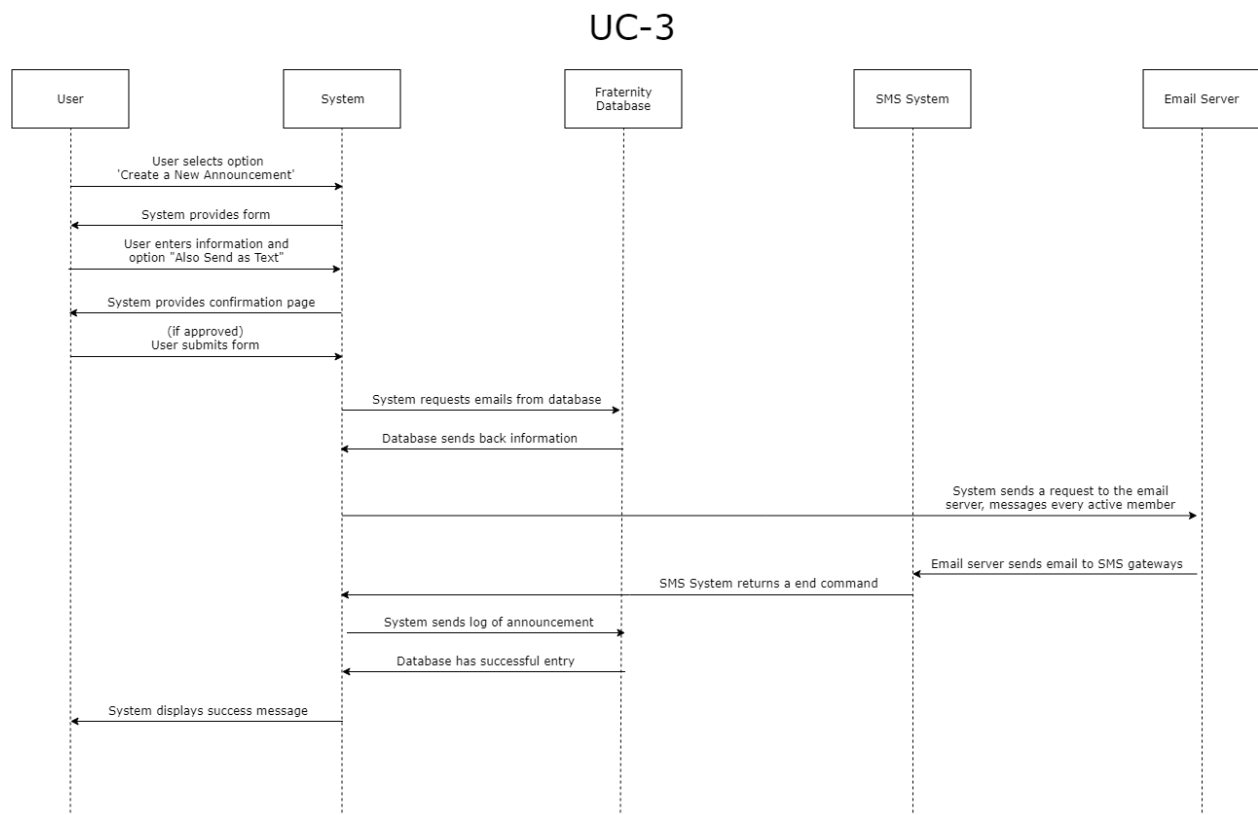


Figure 6.3 - Sequence Diagram of Use Case 3

---

Figure 6.3 shows the sequence diagram of Use Case 3 which is “Creating a Text Announcements”. The objective of this diagram is to create a trail of how the group plans on accomplishing the task of system input to mass text feature. This feature would be used mainly to send very important announcements and would be an executive board exclusive feature. The diagram corresponds to both Main and Alternate Success Scenarios which are discussed below. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

When the User is on the “Announcements” interface page, they click the “Create New Announcement” button and enter the required information into the form and click “Submit.” The **Interface** shows the User a confirmation page and the User either clicks “Confirm” or they are returned to the first step. The **Interface** collects this input and invokes *serve\_request(request)* on the **Controller** (High Cohesion Principle). The **Controller** invokes *respond\_to\_request(request)* on the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** executes its *create\_announcement(request)* which then performs a *insert(in, fields)* into the persistent **Model** (Knower) adding the announcement to the Database (Low Coupling Principle). It also requests the phone numbers and SMS gateway addresses for each recipient using the *select(from, fields, options)* method on the **Model** (High Cohesion Principle). The **Model** responds and the method *announcement\_create(request)* also invokes the *send\_text(recipients, title, message)* on the **Emailer** which sends an email to the SMS gateway address for each recipient (High Cohesion Principle). Upon completion of the action, the **AnnouncementViews** sends a template and the success or failure to the **Renderer** which executes *render(template, context)* which generates and returns a web page to the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** then returns the web page to the **Interface** completing its request where it is displayed to the User (High Cohesion Principle).

## UC-4

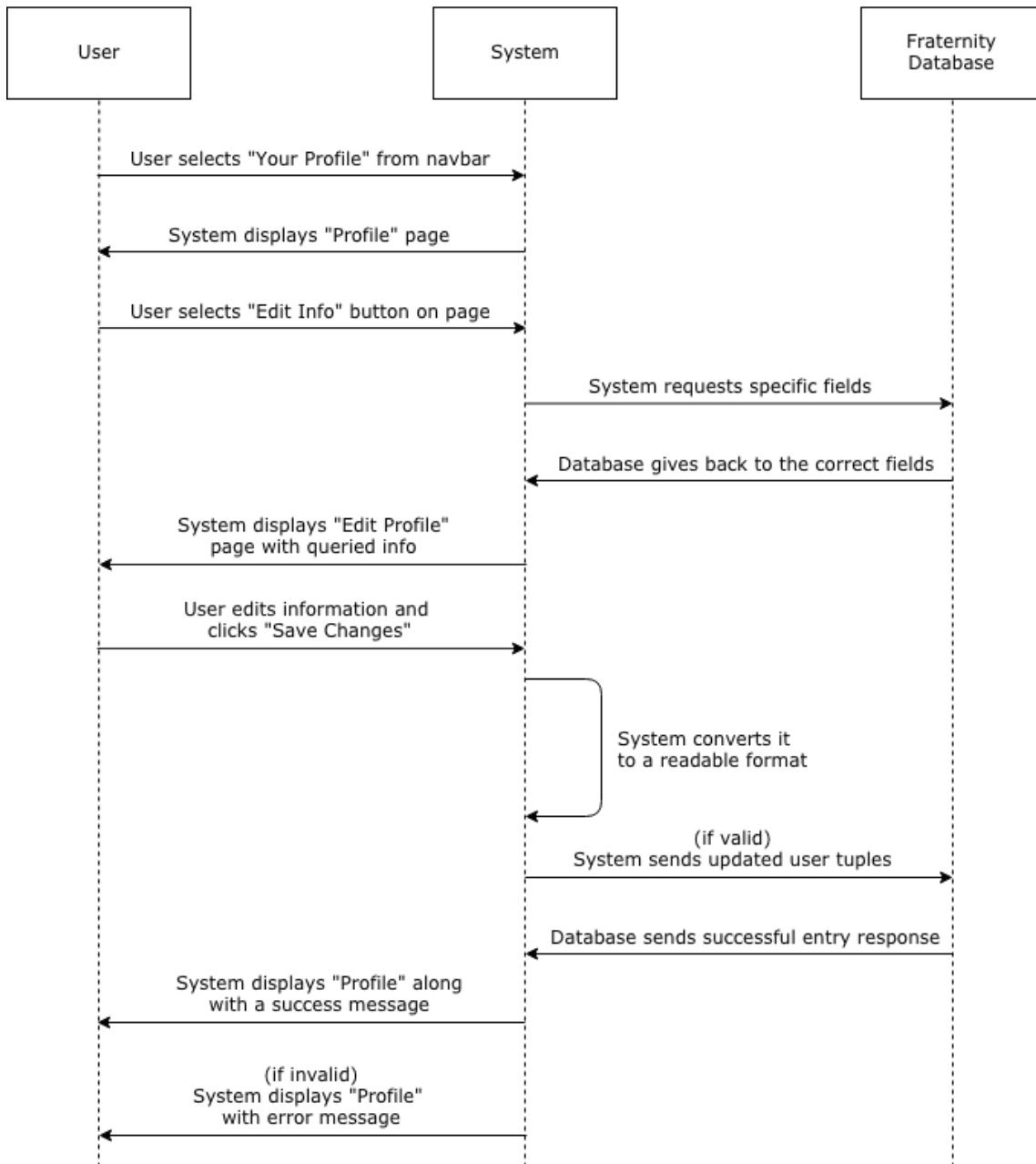


Figure 6.4 - Sequence Diagram of Use Case 4

Figure 6.4 shows the sequence diagram of Use Case 4 which is “Updating Member Info”. The objective of this diagram is to layout the possibilities of editing a user’s profile. This use case relies on the system to handle the queries that the user asks for. The diagram corresponds to both Main and Alternate Success Scenarios which are discussed below. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

While on the “Your Profile” page, a logged-in User chooses the “Edit Information” button on the page. The **Controller** invokes *respond\_to\_request(request)* on the **MemberViews**

---

(High Cohesion Principle). The **MemberViews** executes its *display\_edit*(request) which then performs a *select*(from, fields) from the **Model** (Knower) to populate the edit form fields with pre-existing data (Low Coupling Principle). The **MemberViews** then sends the template with the profile information to the **Renderer** which executes *render*(template, context) which generated and returns a web page to the **MemberViews** (High Cohesion Principle). The **MemberViews** returns the web page to the **Interface** where it is displayed to the User (High Cohesion Principle). The User then enters the new information into their desired fields and clicks “Update.” The **Interface** collects this input and invokes *serve\_request*(request) on the **Controller** (High Cohesion Principle) . The **Controller** invokes *respond\_to\_request*(request) on the **MemberViews** (High Cohesion Principle). The **MemberViews** executes its *hadle\_edit*(request) which validates the User input. If it is invalid, the **MemberViews** returns the same edit form with an error message to the **Interface** (High Cohesion Principle). If it is valid, **MemberViews** then performs an *update*(in, fields, values) into the persistent **Model** (Knower) changing the information in the Database (Low Coupling Principle). The **Model** responds and upon completion of the action, the **MemberViews** sends a profile template and the success or failure to the **Renderer** which executes *render*(template, context) which generates and returns a web page to the **MemberViews** (High Cohesion Principle). The **MemberViews** then returns the web page to the **Interface** completing its request where it is displayed to the User (High Cohesion Principle).



## UC-5

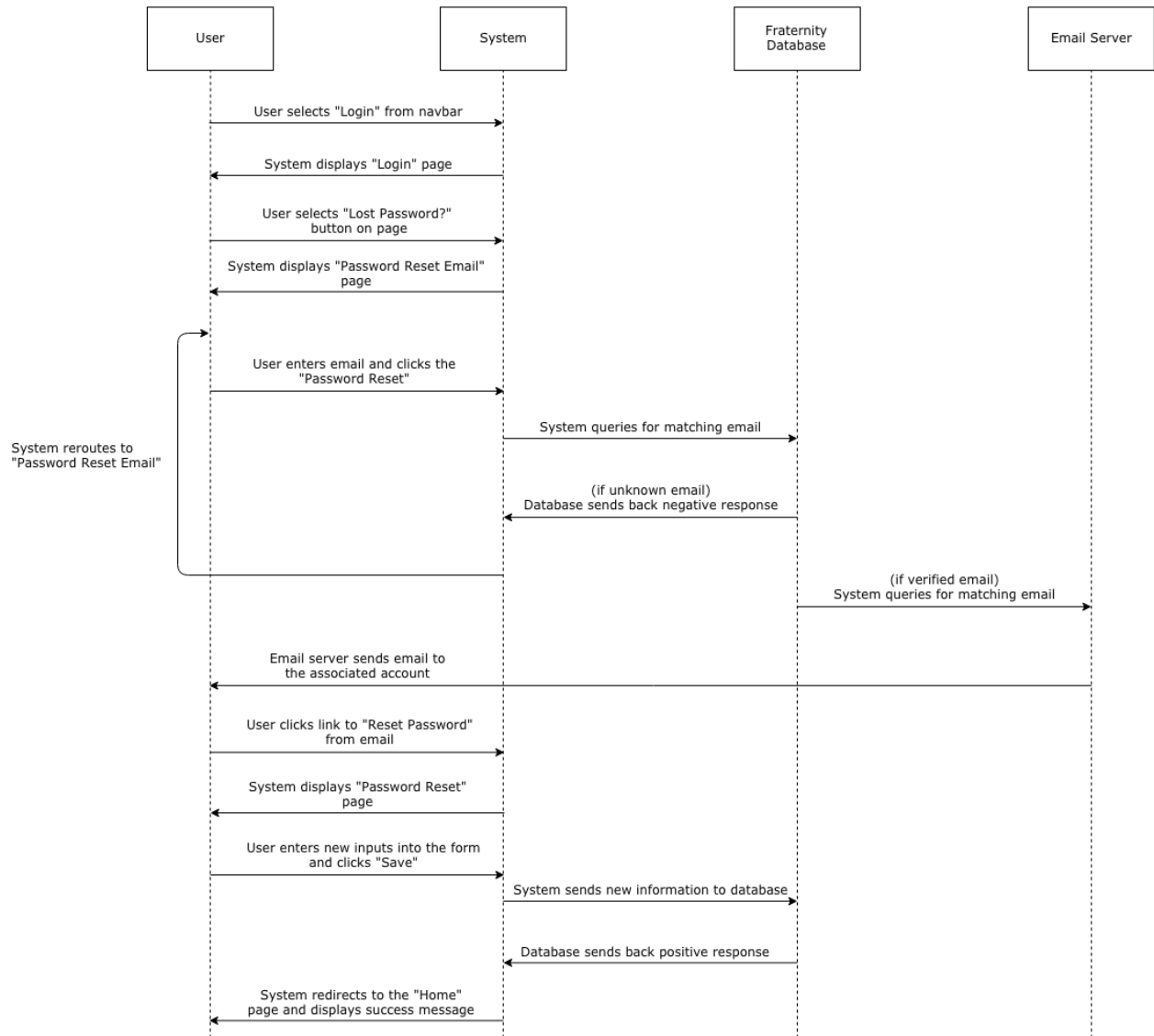


Figure 6.5 - Sequence Diagram of Use Case 5

Figure 6.5 shows the sequence diagram of Use Case 5 which is “Recovering Password”. The objective of this diagram is to display the user uses the application to recover their password. This use case manipulates the database to change user information and the email server to send out the recovery email. The diagram corresponds to both Main and Alternate Success Scenarios which are discussed below. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

While on the “Password Reset” page, a logged-in User enters their email into the form field then presses the “Password Reset” button on the page. The **Interface** collects this input and invokes *serve\_request(request)* on the **Controller** (High Cohesion Principle). The **Controller** invokes *respond\_to\_request(request)* on the **AuthViews** (High Cohesion Principle). The **AuthViews** executes its *handle\_reset\_email(request)* which then performs a

---

*select*(from, fields) from the **Model** (Knower) to check if the email address exists in the system (Low Coupling Principle). If it does not exist, the **AuthViews** returns the same “Password Reset” with an error message to the **Interface** (High Cohesion Principle). If it is valid, the method *handle\_reset*(request) also invokes the *send\_email*(recipients, title, message) method on the **Mailer** which sends an email to the entered email address with a link (High Cohesion Principle). Upon completion of the action, the **MemberViews** sends a template and the success or failure to the **Renderer** which executes *render*(template, context) which generates and returns a web page to the **AuthViews** (High Cohesion Principle). The **AuthViews** then returns the web page to the **Interface** with further instructions for the User. When the User follows the link sent to them, they are taken to the “Password Reset Form” where they enter their new password and click the “Save” button. The **Interface** collects this input and invokes *serve\_request*(request) on the **Controller** (High Cohesion Principle). The **Controller** invokes *respond\_to\_request*(request) on the **AuthViews** (High Cohesion Principle). The **AuthViews** executes its *handle\_reset*(request) which then performs an *update*(in, fields, values) on the **Model** (Knower) to save the new password (Low Coupling Principle). Upon completion of the action, the **MemberViews** sends a template and the success or failure to the **Renderer** which executes *render*(template, context) which generates and returns a web page to the **AuthViews** (High Cohesion Principle). The **AuthViews** then returns the web page to the **Interface** completing its request where it is displayed to the User (High Cohesion Principle).

## UC-7

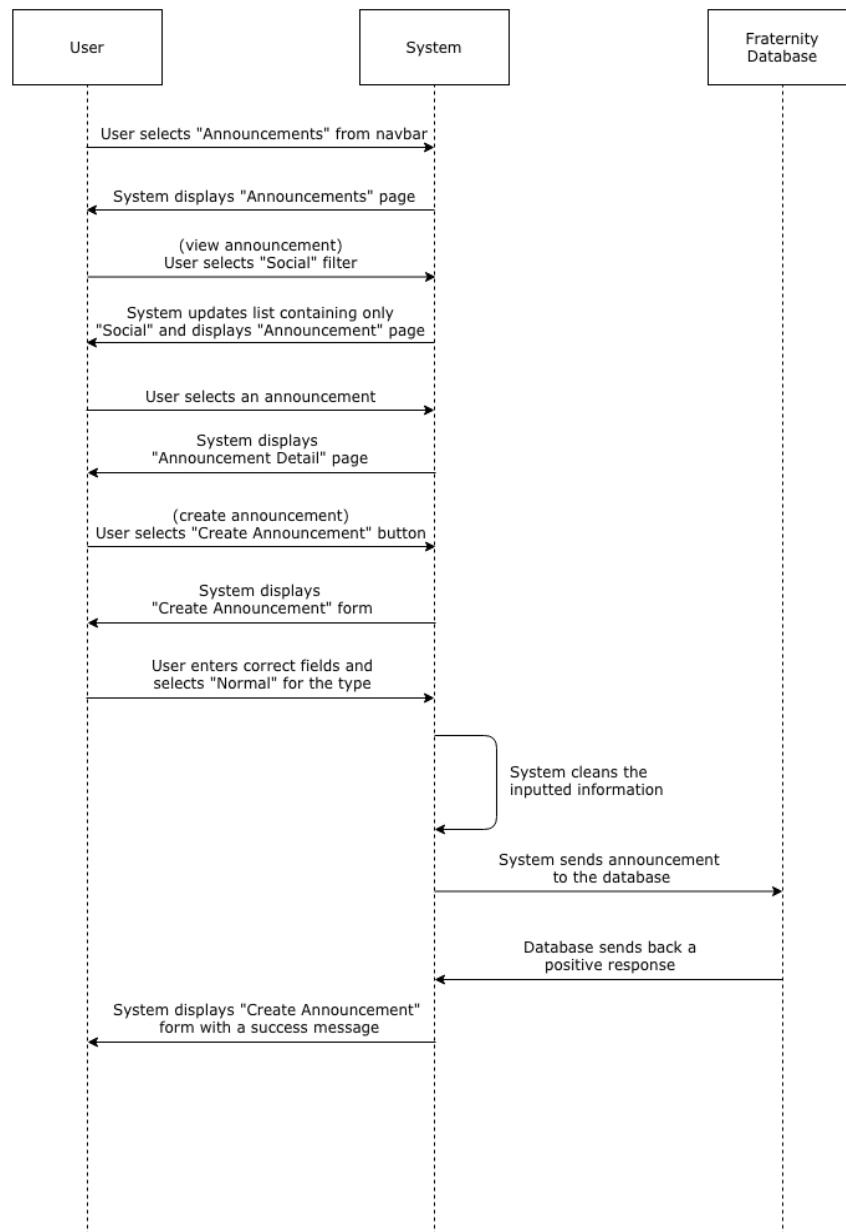


Figure 6.6 - Sequence Diagram of Use Case 7

Figure 6.6 shows the sequence diagram of Use Case 7 which is “Creating and Viewing Announcements”. The objective of this diagram is to detail the process of creating and viewing announcements, with two separate paths for viewing and creating. This use case relies on the database to properly store the data so that it is manipulated correctly. The diagram corresponds to both Main and Alternate Success Scenarios which are discussed below. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

For the Main Scenario, when the User is on the “Announcements” interface page which displays all the current announcements, they select the “Social” filter option. The **Interface** collects this input and invokes `serve_request(request)` on the **Controller** (High Cohesion

---

Principle) . The **Controller** invokes *respond\_to\_request(request)* on the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** executes its *display\_index(request)* which then performs a *select(from, fields)* from the persistent **Model** (Knower) retrieving only announcements with the type “social” from the Database (Low Coupling Principle). The **Model** responds and upon completion of the action, the **AnnouncementViews** sends a template and the list of announcements to the **Renderer** which executes *render(template, context)* which generates and returns a web page to the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** then returns the web page to the **Interface** which only contains “social” type announcements (High Cohesion Principle). The User then selects an announcement from that list and the **Interface** invokes *serve\_request(request)* on the **Controller** (High Cohesion Principle) . The **Controller** invokes *respond\_to\_request(request)* on the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** executes its *display\_detail(request)* which then performs a *select(from, fields)* from the persistent **Model** (Knower) retrieving only the announcement chosen by the User from the Database (Low Coupling Principle). The **AnnouncementViews** sends a template and the announcement to the **Renderer** which executes *render(template, context)* which generates and returns a web page to the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** then returns the web page showing the details of the chosen announcement to the **Interface** where it is displayed to the User (High Cohesion Principle).

For the alternate scenario, when the User is on the “Announcements” interface page, they click the “Create New Announcement” button and enter the required information into the form and click “Submit.” The **Interface** shows the User a confirmation page and the User either clicks “Confirm” or they are returned to the first step. The **Interface** collects this input and invokes *serve\_request(request)* on the **Controller** (High Cohesion Principle) . The **Controller** invokes *respond\_to\_request(request)* on the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** executes its *create\_announcement(request)* which then performs an *insert(in, fields)* into the persistent **Model** (Knower) adding the announcement to the Database (Low Coupling Principle). The **AnnouncementViews** sends a template and the success or failure to the **Renderer** which executes *render(template, context)* which generates and returns a web page to the **AnnouncementViews** (High Cohesion Principle). The **AnnouncementViews** then returns the web page to the **Interface** where it is displayed to the User (High Cohesion Principle).

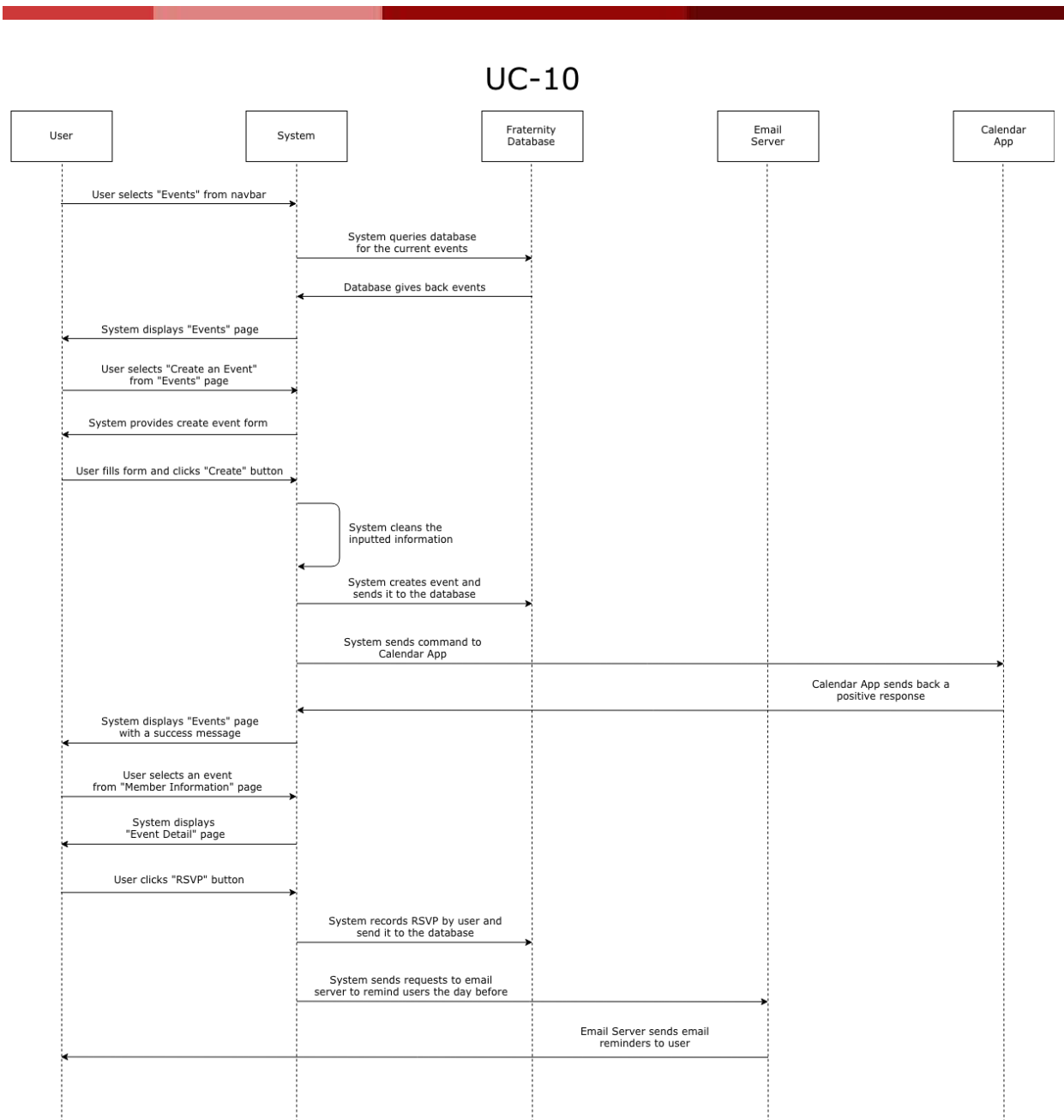


Figure 6.7 - Sequence Diagram of Use Case 10

Figure 6.7 shows the sequence diagram of Use Case 10 which is “Managing Events”. The objective of this diagram is to detail the various pathways a user will take to create and manipulate the event structure. This use case relies on the most systems of any use case diagram, with the system manipulating three separate services to move data. The diagram corresponds to both Main and Alternate Success Scenarios which are discussed below. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

For the Main scenario, when the User is on the “Events” interface page, they click the “Create New Event” button and enter the required information into the form and click “Submit.” The **Interface** collects this input and invokes *serve\_request(request)* on the **Controller** (High

---

Cohesion Principle). The **Controller** invokes *respond\_to\_request*(request) on the **EventViews** (High Cohesion Principle). The **EventViews** executes its *create\_event*(request) which then performs an *insert*(in, fields) into the persistent **Model** (Knower) adding the event to the Database (Low Coupling Principle). The **EventViews** sends a template and the success or failure to the **Renderer** which executes *render*(template, context) which generates and returns a web page to the **EventViews** (High Cohesion Principle). The **EventViews** then returns the web page to the **Interface** where it is displayed to the User (High Cohesion Principle).

For the alternate scenario, the User then selects an event from the list on the “Events” interface page and the **Interface** invokes *serve\_request*(request) on the **Controller** (High Cohesion Principle). The **Controller** invokes *respond\_to\_request*(request) on the **EventViews** (High Cohesion Principle). The **EventViews** executes its *display\_detail*(request) which then performs a *select*(from, fields) from the persistent **Model** (Knower) retrieving only the event chosen by the User from the Database (Low Coupling Principle). The **EventViews** sends a template and the event to the **Renderer** which executes *render*(template, context) which generates and returns a web page to the **EventViews** (High Cohesion Principle). The **EventViews** then returns the web page showing the details of the chosen event to the **Interface** where it is displayed to the User (High Cohesion Principle). The User then clicks the “RSVP” button on the detail page and the **Interface** invokes *serve\_request*(request) on the **Controller** (High Cohesion Principle). The **Controller** invokes *respond\_to\_request*(request) on the **EventViews** (High Cohesion Principle). The **EventViews** executes its *rsvp\_event*(request) which then performs an *insert*(in, fields) into the persistent **Model** (Knower) adding the User to the list of RSVP-ed Users for this event in the Database (Low Coupling Principle). The **EventViews** sends a template and success or failure to the **Renderer** which executes *render*(template, context) which generates and returns a web page to the **EventViews** (High Cohesion Principle). The **EventViews** then returns the web page showing the details of the chosen event to the **Interface** where it is displayed to the User (High Cohesion Principle). Before the event, the Controller will invoke the *remind\_event*(request) method on the **EventViews** (High Cohesion Principle). The method performs a *select*(from, fields) from the persistent **Model** (Knower) to retrieve the RSVP-ed Users for the event and their email addresses from the Database (Low Coupling Principle). The method then invokes the *send\_email*(recipients, title, message) method on the **Emailer** which sends an email to the email address for each RSVP-ed User (High Cohesion Principle).

## UC-11

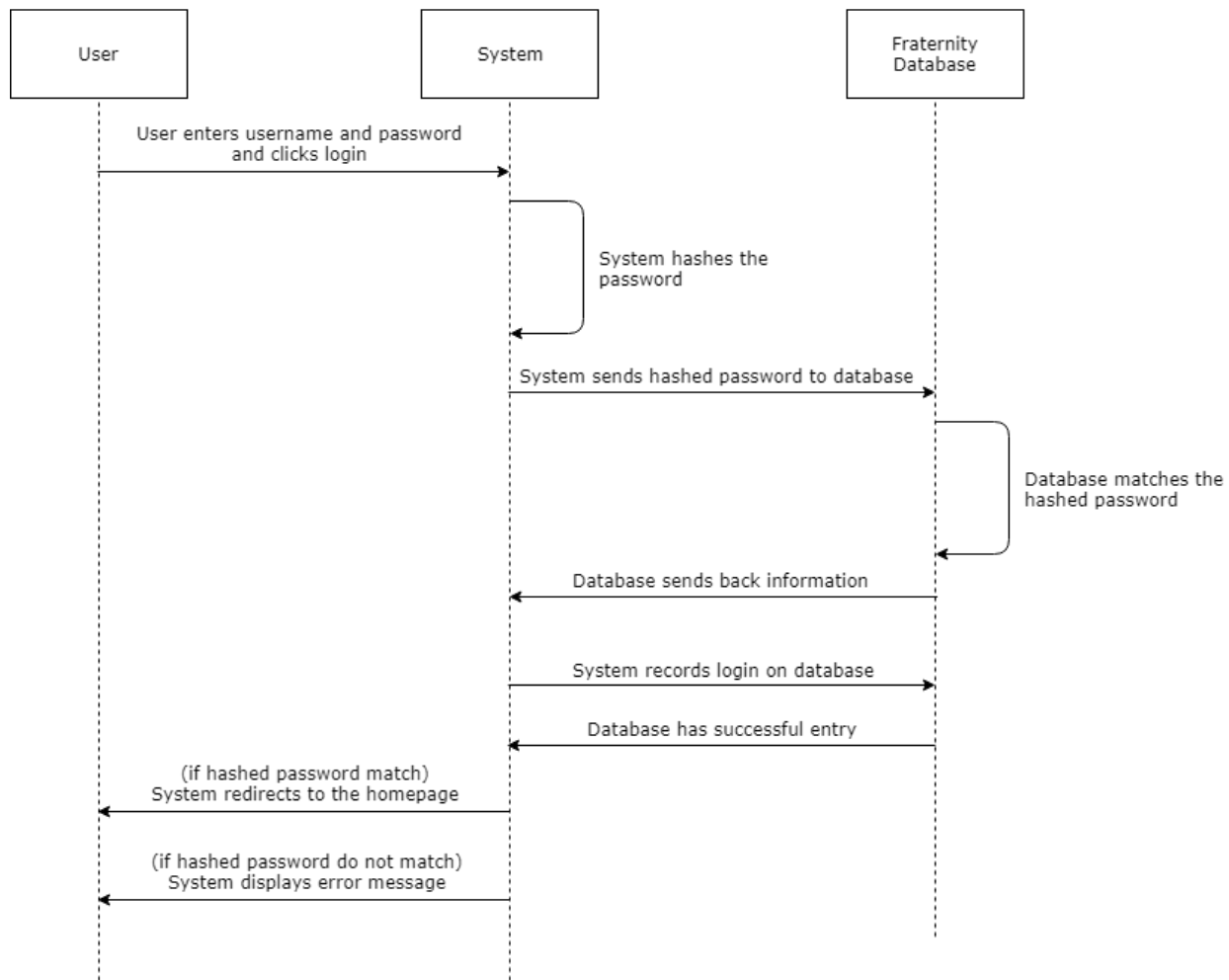


Figure 6.8 - Sequence Diagram of Use Case 11

Figure 6.8 shows the sequence diagram of Use Case 11 which is “Logging into the System”. The objective of this diagram is to explain the importance of a secure login for the project. In order to access many of the features that are explained in the system requirements, the database needs to be password protected. This ensures secure entries and the safeguarding of data. The diagram corresponds to both Main and Alternate Success Scenarios which are discussed below. The discussion below goes into detail about what interactions happen inside the System in the above diagram.

When the User is on the “Login” interface page, they enter their criteria into the form and click “Login.” The **Interface** collects this input and invokes *serve\_request(request)* on the **Controller** (High Cohesion Principle) . The **Controller** invokes *respond\_to\_request(request)* on the **AuthViews** (High Cohesion Principle) . The **AuthViews** executes its *login\_check(request)* which then performs a *select(from, fields)* method call on the **Model** (Knower) for a username and hashed password match (Low Coupling Principle). Upon receipt of the response, if a match was found, the **AuthViews** invokes the *update(in, fields)* method on

---

the **Model** to record the login. Next, *login\_check* sends a template and the success/failure to the **Renderer** which executes *render*(template, context) which generates and returns a web page to the **AuthViews** (High Cohesion Principle). The **AuthViews** then returns the web page to the **Interface** where it is displayed to the User (High Cohesion Principle).

## 6.1 Design Patterns

The System makes use of several design patterns in its interactions to solve common problems throughout our design. Using a given design pattern gives the solution a structured approach that has been proven effective in many other applications.

A design pattern that we employ that is common across all interactions is the **Front Controller pattern**. Our Controller actor handles all requests coming into the System by the User. The Controller can then delegate the request to the correct object to handle the request. This centralized entry point to the System gives us the flexibility to change how a request is handled easily and quickly. It also allows us to separate the Interface from the Views that will handle its request promoting loose coupling by keeping objects from referring to each other explicitly, and it allows their function to vary independently.

Another pattern widely used in our interactions is the **Command pattern**. This allows a request to be encapsulated as an object and passed between objects. This is seen when a request is made by the User using the Interface. The Interface packages the request and gives it to the Controller who passes it to the correct object which then services the request. This command object encapsulates all the information needed to perform the request. This pattern makes it easy for us to delegate the responsibility of each request to the appropriate object. It also helps with tracking what actions are being done in the system and by what Users.



# 7 Class Diagrams and Interface Specifications

## 7.1 Class Diagram

The class diagram for the GLMMS is shown below in figure 2.1.1.

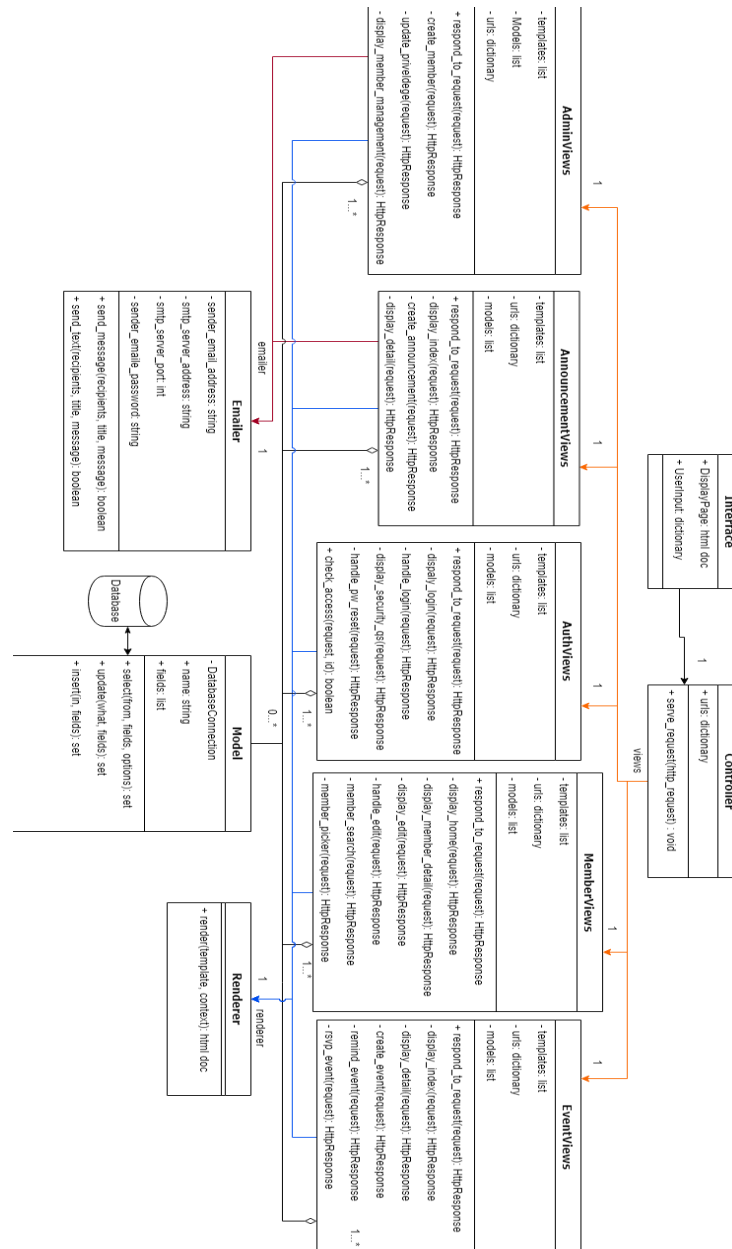


Figure 7.1.1 - Class Diagram. Zoom in for better view.

## 7.2 Data Types and Operation Signatures

The **Interface** class describes the boundary of the system where the the user is displayed information and enters their input.

Interface
<b>Attributes:</b> <ul style="list-style-type: none"><li>• displayPage: html doc; The displayPage is the html document currently displayed to the user.</li><li>• userInput: dictionary; The userInput attribute is a dictionary mapping the values entered into the Interface by the user. The names of these inputs are mapped to where they were entered. They are part of the HttpRequest sent to the controller.</li></ul>
<b>Methods</b> <ul style="list-style-type: none"><li>• N/A</li></ul>

The **Controller** class is in charge of routing user requests to the correct View class in the System.

Controller
<b>Attributes:</b> <ul style="list-style-type: none"><li>• urls: dictionary; The urls attribute is a dictionary, mapping each url that the user can request to the correct View to handle the request.</li></ul>
<b>Methods</b> <ul style="list-style-type: none"><li>• serve_request(http_request): void; This method is invoked by the Interface to ask the System to respond to a user request. The parameter http_request is a HttpRequest which contains the user input, if any, and the requested resource.</li></ul>

The **AdminViews** class is the View responsible for responding to actions that Admin-level Users can execute on the System.

AdminViews
<b>Attributes:</b> <ul style="list-style-type: none"><li>• templates: list; The templates attribute is a list of templates for how to display data for each webpage this view is responsible for creating.</li><li>• models: list; The models attribute is a list of Models representing the database tables that this View interacts with.</li><li>• urls: dictionary; The urls attribute is a dictionary that maps a url to the appropriate method to respond to that url.</li></ul>
<b>Methods</b> <ul style="list-style-type: none"><li>• respond_to_request(request): HttpResponse; This method is invoked by the Controller to tell this View to generate a response. It uses the urls attribute to invoke the correct method on itself to generate the response, sending it to the Interface.</li><li>• update_privelgede(request): HttpResponse; This method is called when a Admin User requests to update the privileges of a member. This updates the privileges using the Member Model or it will return an error response.</li></ul>

- `create_member(request)`: `HttpResponse`; This method is called when a Admin User requests to create a new member. Creates the member using the Member Model or returns an error response.
- `display_member_management(request)`: `HttpResponse`; This method responds with the member management page which allows Admin Users to create members or update privileges.

The **AnnouncementViews** class is the View responsible for responding to actions regarding announcements that can be executed on the System.

<b>AnnouncementViews</b>
<b>Attributes:</b> <ul style="list-style-type: none"> <li>• <code>templates</code>: list; same as <code>AdminViews.templates</code>.</li> <li>• <code>models</code>: list; same as <code>AdminViews.models</code>.</li> <li>• <code>urls</code>: dictionary; Same as <code>AdminViews.urls</code>.</li> </ul>
<b>Methods</b> <ul style="list-style-type: none"> <li>• <code>respond_to_request(request)</code>: <code>HttpResponse</code>; same as <code>AdminViews.respond_to_request(request)</code></li> <li>• <code>display_index(request)</code>: <code>HttpResponse</code>; This method generates the index page for announcements.</li> <li>• <code>create_announcement(request)</code>: <code>HttpResponse</code>; This method handles the request to create a new announcement using the Announcements Model. Responds with either a success or failure page.</li> <li>• <code>display_detail(request)</code>: <code>HttpResponse</code>; This method responds with a page showing the full details for a requested announcement.</li> </ul>

The **AuthViews** class is the View responsible for authorizing and authenticating users.

<b>AuthViews</b>
<b>Attributes:</b> <ul style="list-style-type: none"> <li>• <code>templates</code>: list; same as <code>AdminViews.templates</code>.</li> <li>• <code>models</code>: list; same as <code>AdminViews.models</code>.</li> <li>• <code>urls</code>: dictionary; Same as <code>AdminViews.urls</code>.</li> </ul>
<b>Methods</b> <ul style="list-style-type: none"> <li>• <code>respond_to_request(request)</code>: <code>HttpResponse</code>; same as <code>AdminViews.respond_to_request(request)</code></li> <li>• <code>display_login(request)</code>: <code>HttpResponse</code>; This method responds with the User login page for the System.</li> <li>• <code>handle_login(request)</code>: <code>HttpResponse</code>; This method is in charge of handling the login logic for the System. It uses the Auth Model to match the username and password and generates either the home page or a failure.</li> <li>• <code>display_security_qs(request)</code>: <code>HttpResponse</code>; This method generates the password reset page with the security questions and the new password fields.</li> <li>• <code>display_reset_email(request)</code>: <code>HttpResponse</code>; This method displays the form to enter the email for the account that needs its password reset.</li> </ul>

- `handle_reset_email(request)`: `HttpResponse`; This method handles the submission of an email address associated with an account that needs its password reset.
- `display_reset_form(request)`: `HttpResponse`; This method responds with the form in which Users enter their new password.
- `handle_reset(request)`: `HttpResponse`; This method checks handles the reset of a user's password.
- `check_access(request, id)`: `boolean`; This method is used by the Controller to decide whether or not a logged-in User can access a certain resource or functionality using the Auth Model. The request is the `HttpRequest` describing the requested resource and the id is that of the logged-in User.

The **MemberViews** class is the View responsible for responding to actions that all levels of Users can execute on the System.

<b>MemberViews</b>
<b>Attributes:</b> <ul style="list-style-type: none"> <li>• <code>templates</code>: list; same as <code>AdminViews.templates</code>.</li> <li>• <code>models</code>: list; same as <code>AdminViews.models</code>.</li> <li>• <code>urls</code>: dictionary; Same as <code>AdminViews.urls</code>.</li> </ul>
<b>Methods</b> <ul style="list-style-type: none"> <li>• <code>respond_to_request(request)</code>: <code>HttpResponse</code>; same functionality as <code>AdminViews.respond_to_request(request)</code></li> <li>• <code>display_home(request)</code>: <code>HttpResponse</code>; This method responds with the homepage of the application.</li> <li>• <code>display_member(request)</code>: <code>HttpResponse</code>; This method responds with a page displaying the full details of a member using the Member Model.</li> <li>• <code>display_edit(request)</code>: <code>HttpResponse</code>; This page generates the form to edit a member's info based on the Member Model.</li> <li>• <code>handle_edit(request)</code>: <code>HttpResponse</code>; This method is used to handle a request to edit a member's information. It uses the Member Model to edit and save the new information.</li> <li>• <code>member_search(request)</code>: <code>HttpResponse</code>; This method generates a page displaying a list of members and dynamic information based on the user's inputs from the User using the Member Model.</li> </ul>

The **EventViews** class is the View responsible for responding to actions regarding events that can be executed on the System.

<b>EventViews</b>
<b>Attributes:</b> <ul style="list-style-type: none"> <li>• <code>templates</code>: list; same as <code>AdminViews.templates</code>.</li> <li>• <code>models</code>: list; same as <code>AdminViews.models</code>.</li> <li>• <code>urls</code>: dictionary; Same as <code>AdminViews.urls</code>.</li> </ul>
<b>Methods</b>

- `respond_to_request(request)`: `HttpResponse`; same as `AdminViews.respond_to_request(request)`
- `display_index(request)`: `HttpResponse`; This method responds with the index page for the events in the database.
- `display_detail(request)`: `HttpResponse`; This method generates a page showing the full details of an event using the Event Model.
- `create_event(request)`: `HttpResponse`; This method handles the creation of an event into the database using the Event Model. It responds with a success or failure page.
- `remind_event(request)`: `HttpResponse`; This method handles the request to send an email reminder to the RSVPed Users to an event. It responds with a success or failure page.
- `rsvp_event(request)`: `HttpResponse`; This method handles a request from the User to RSVP to an event. It creates a record using the RSVP Model and responds with a success or failure.

The **Mailer** class is responsible for encapsulating the functionality to send emails and SMS messages to members.

<b>Mailer</b>
<b>Attributes:</b> <ul style="list-style-type: none"> <li>• <code>sender_email_address</code>: string; This is the email address to use to send the emails</li> <li>• <code>smtp_server_address</code>: string; This is the address of the SMTP server to use to send the email.</li> <li>• <code>smtp_server_port</code>: int; This is the port number on the SMTP server to send the message to.</li> <li>• <code>sender_email_password</code>: string; This is the password to the email address being used to send the emails.</li> </ul>
<b>Methods</b> <ul style="list-style-type: none"> <li>• <code>send_message(recipients, title, message)</code>: boolean; This method is invoked to send an email to the list of recipients with the provided title and message.</li> <li>• <code>send_text(recipients, title, message)</code>: boolean; This method is invoked to send a text message to the provided recipients with the title and message through an SMS gateway. It uses the Cell Carrier Model to find the correct SMS gateway for each member.</li> </ul>

The **Model** class is the class representation of a table in the database. Each Model is connected to the database and allows searching, saving, and updating of its entries in the database.

<b>Model</b>
<b>Attributes:</b> <ul style="list-style-type: none"> <li>• <code>databaseConnection</code>: A connection to the database.</li> <li>• <code>name</code>: string; The name attribute is the name of the database table.</li> <li>• <code>fields</code>: list; The fields attribute is a list of tuples of the columns' names and data types in the database.</li> </ul>
<b>Methods</b>

- select(from, fields, options): set; This method performs a select query on the from parameter with the provided fields and options.
- update(what, fields): set; This method updates a record in the database.
- insert(in, fields): set; This method inserts a new record in the database.

The **Render** class is responsible for rendering templates into html pages based on a given context of variables.

<b>Renderer</b>
<b>Attributes:</b> <ul style="list-style-type: none"> <li>• N/A</li> </ul>
<b>Methods</b> <ul style="list-style-type: none"> <li>• render(template, context): html doc; This method is invoked by the View classes to render a template into a html page based on the variables and values defined in the given context dictionary.</li> </ul>

## 7.3 Traceability Matrix

	<b>Class</b>									
<b>Domain Concepts</b>	Interface	Controller	AdminViews	AnnouncementViews	AuthViews	MemberViews	EventViews	Emailer	Model	Renderer
DatabaseConnector									X	
Interface	X									
MainController		X								
AccessManager					X					
Emailer								X		
LoginVerifier					X					
MemberController		X								
MemberPicker										
SearchOperator						X				
ElectionTallier										

FileArchiveManager										
MemberInfoUpdater						X				
AnnouncementManager				X						
EventManager							X			
PositionHolderController		X								
OfficerReportFormSubmitter										
ExecController		X								
ElectionCreator										
ProfileCreator			X							
AdminController		X								
PrivilegeUpdater			X							
FamilyTreeGenerator										

Table 7.3.1 - Traceability Matrix from concepts to classes.

We have chosen to use the Model View Controller (MVC) architecture style, so we combined all the Controller concepts from Report 1 Section 5. Domain Analysis into one Controller which will map requests to Views based on their url. The subconcepts found under the Controllers were moved into separate View classes based on their function: the MemberController, ExecController, PositionHolderController concepts -> MemberView; AdminController concepts -> AdminView; ExecController -> MemberView; The DatabaseConnector concept was also converted to the Models class, but retains its functionality to fit with the MVC architecture. We have also added a Renderer concept that decides how data is displayed on a webpage based on a template. We added this concept to keep the presentation of data away from the application logic. Another change from the Report-1 domain model was to combine the LoginVerifier and the AccessManager concepts into the AuthView class because their responsibilities are very similar. Finally, the Interface concept refers to the client's browser since our implementation is using a web architectural style. The System Architecture is discussed in more detail in Section 8. More information about MVC pattern can be found at the links in Section 13. References of this report.

Some concepts will not be made into classes for Demo-2. Some Use Cases were determined to be of lesser value to the project and have been deferred for development until a later time. These Use Cases are UC-6, UC-8 , UC-9, UC-12, UC-13, UC-14. The concepts corresponding to these use cases are FamilyTreeGenerator, ElectionCreator, OfficerReportFormSubmitter, FileArchiveManager, and ElectionTallier.

## 7.4 Design Patterns

The System makes use of several design patterns in its classes to solve common problems throughout our design. Using a given design pattern gives the solution a structured approach that has been proven effective in many other applications.

For the Views classes, we use the **Module pattern**. This allows us to collect all the functionality and classes for related uses into one conceptual entity. These modules can then be referenced by the Controller to service requests. The Controller only needs to know what Module to pass the request to, not the exact method to call. The pattern also, helps us to organize our code.

We also use the **Servant pattern** when designing the EMailer and Renderer classes. These classes provide functionality to all the Views classes without defining that functionality in each of them. We use this pattern to avoid defining this functionality in all these classes promoting code reuse and avoiding repeating the code.

## 7.5 Object Constraint Language

Class	Invariant	Precondition	Postcondition
<i>Interface</i>	The user has entered correct credentials into the login.	Correct pages are loaded in variables to display on click.	Webpages are shown based on the request.
<i>Controller</i>	The link/submission is correctly routed.	Each of the routes have corresponding links/methods to handle the request.	The server request has been correctly directed to the interface.
<i>AdminViews</i>	The user has entered credentials with the correct authority to view the class.	Credentials are ready to be sent to the database for validation.	Database pumps out a successful response for access.
<i>AnnouncementViews</i>	Only pages that are within the announcement class are requested.	Link/submission is correctly routed.	Displayed page is in the announcement class.
<i>AuthViews</i>	The forms pages will displayed to the user	Authorization credentials are ready	Success/failure code sent from the



	to manipulate.	to be sent to the database.	database.
<i>MemberViews</i>	The user has entered credentials with the correct authority to view the class.	User is logged into an account.	User retains the status of logged in.
<i>EventViews</i>	Only pages that are within the event class are requested.	Link/submission is correctly routed.	Displayed page is in the event class.
<i>Mailer</i>	The database will hold a certain column of data, which this class has access to.	The corresponding addresses have been queue to send to the API.	Either a positive/negative response code received.
<i>Model</i>	A connection is established between server and database.	Data has been correctly inputted to send to the database.	Database has received the appropriate code for outcome.
<i>Renderer</i>	Variables are given to render HTML.	Correct variables have been given to the class.	Success code from the render of said page.

## 8 System Architecture and System Design

### 8.1 Architecture Style

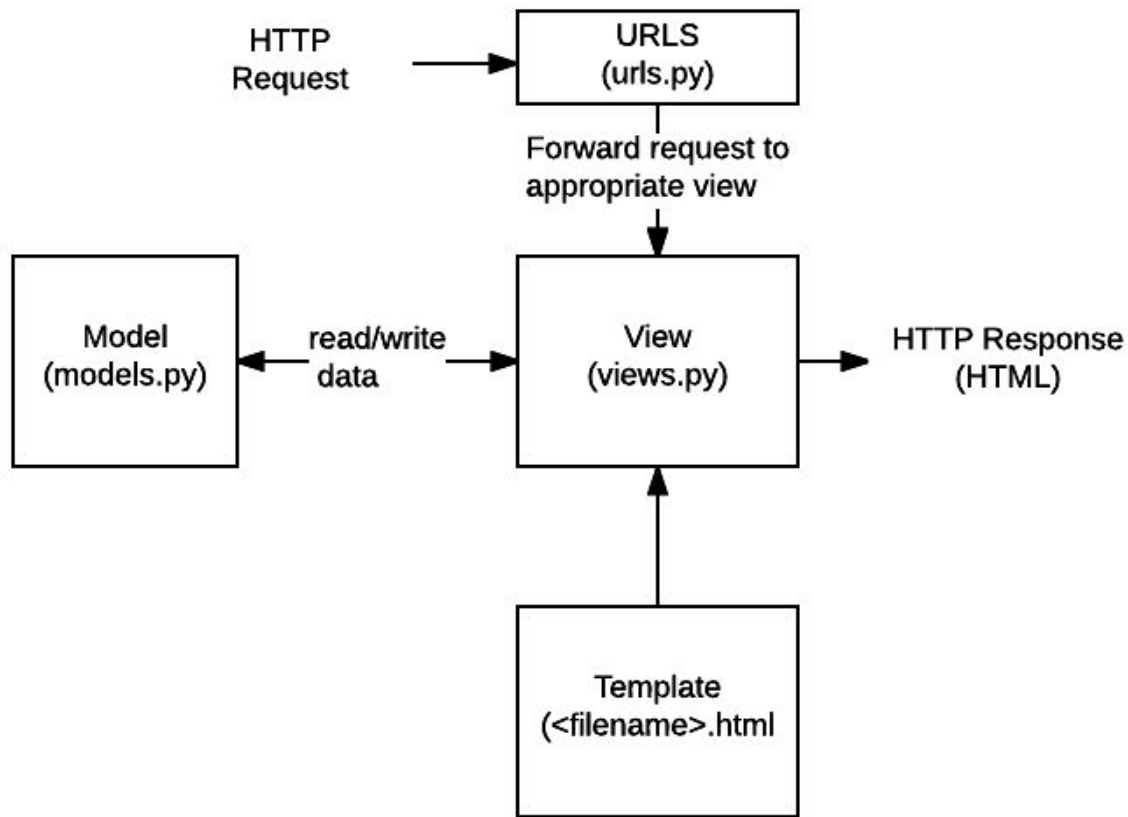


Figure 8.1.1 - Architecture Style

Our architectural style is a variation of Model-View-Controller. Model-View-Controller is an architectural pattern commonly used for developing user interfaces that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.<sup>12</sup> Django, the framework we are using, utilizes a modified version of this called Model-Template-View. We chose this design pattern for our application because it is very user interface centric since it is a web application. This pattern also helps enforce good, modular software design.

### 8.2 Subsystems

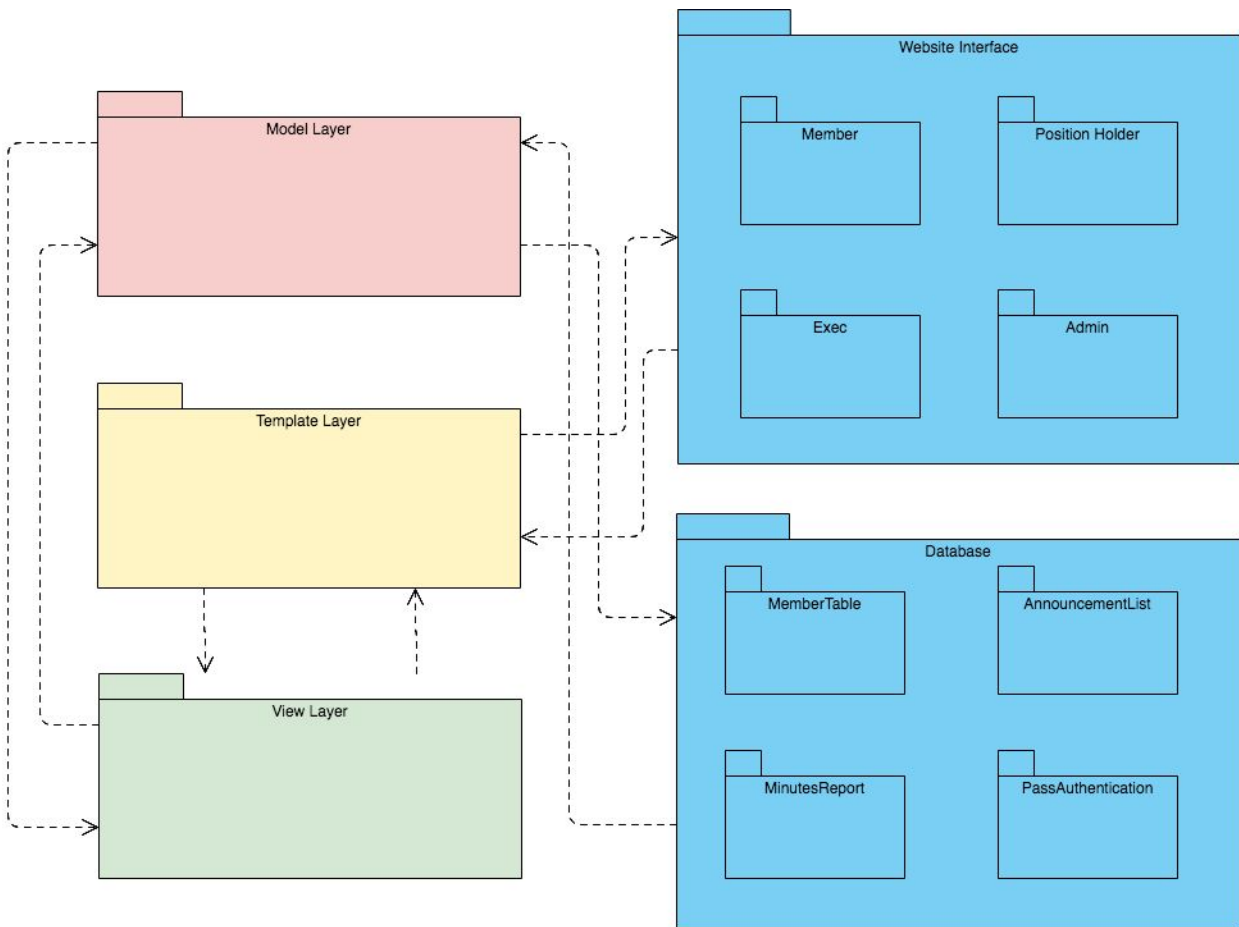



Figure 8.2.1 - Subsystems Used in the System

The package diagram is composed of all the subsystems of the System. Firstly, the website interface contains all the types of UI that the user could be looking at depending on if you are a base member, position holder, executive member or administrator of the site. Next, the database holds many of the tables that are central to a function of the website, most importantly: the member list, announcement tables, and the authentication of the site. The model handles flow of information transfer between the database and other packages which request or send anything through the model. The model consists of an instance for each table in the database. This evolved from our domain model concept of the DatabaseConnector. Instead of one DatabaseConnector, we have a model for each table in the database so that they can be treated as a normal class. The template package is in charge of describing how to display the data generated by the View package on an HTML page. Finally, the view package works with the model to retrieve, update, and create data and the template to display that data as a web page to the user.

### 8.3 Subsystems to Hardware Map



Where there is web access, our project will be accessible. Our system is based on a client server relationship as it is web application. A web browser is a web client, and can request resources from the server using HTTP. This web client/server model will need to be run across multiple computers. A web browser provides the interface for our application and will be run on a client machine. It will be used to request the various data and forms from our server. The database will be saved on hard-drives on a server, which will be owned and operated by the private server provider. Our system stops at the web pages displayed by the web browser on the client machine and on the server it stops at our files and software running on the computers provided by the private server provider.

## 8.4 Persistent Data Storage

Persistent data storage is a very important part of our project. We chose to use a relational database for our storage needs because helps to enforce relationships between objects.

The user interfaces is highly dependent on SQL queries to make sure that the content is dynamic. Member information, employer information and event information all are sent to the interface of the website. This data is then sometimes manipulated and needs to be sent back.

The database diagram is shown below showing the structure and relationship of the tables.

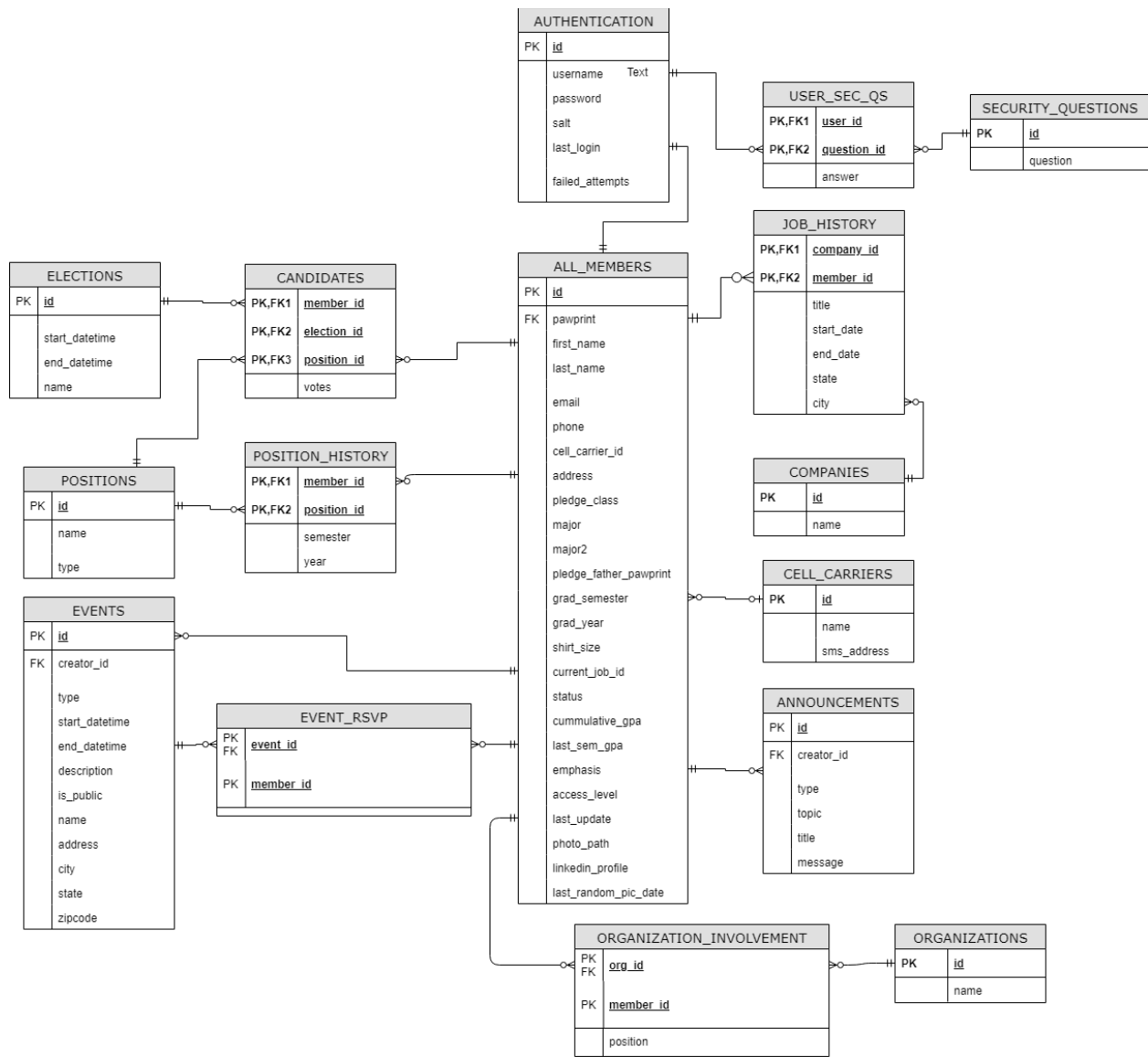



Figure 8.4.1 - Database Schema. Zoom for a better view.

## 8.5 Network Protocol

Network protocols come in many different flavors such as FTP (File Transfer Protocol), HTTP (HyperText Transfer Protocol) and SSH (Secure Shell). In this system, a user uses the website to log into his/her personal account and access the software user interface. Because of the limitations that are apparent if we do NOT use HTTP, we have to use this protocol because other users need to be able to access the website.

## 8.6 Global Control Flow

This project is very procedure driven, as evident by the many forms that are available to the various users. None of the forms are dynamic enough to warrant a loop to wait for the user to



decide on a course of action. A button is pressed to request a certain form, the form is filled out to the proper specifications, and then sent to the database.

Our project is event-response. Events are usually generated by User input such as clicking links or submitting forms.

The project will not use multi-thread programming.

## 8.7 Hardware Requirements

End-user: The user will only require an internet enabled device and an up to date web browser. The majority of the information will be stored on the server so Users' systems should not be taxed in a way that they would need specialized equipment.

Server-side: The server will be hosting the database as well as the Django environment. It will need to store user data, announcements, emails, etc. It will need to be connected to the Internet. The server will need 1GB of hard disk space to store all images and to allow the database to grow over time. The server should have a bandwidth of at least 100 Mbps to handle all requests.

## 9 Algorithms and Data Structures

### 9.1 Algorithms

There are no mathematical models to implement in this project.

### 9.2 Data Structures

The most complex data structures in the project are the models that are used to generate the tables of the database. Data was chosen to be stored in database tables because of the flexibility and performance they provide. They can grow indefinitely as long as there is disk space and they provide efficient searching. We also use dictionaries to store some data such as mapping urls to functions. They were chosen for ease of use and the efficient mapping of two items.

## 10 User Interface Design and Specification

Our preliminary designs included in Report 1 for the user interface design have been replaced in this section with screenshots of the actual UI implementation as of this Report. The design is continually improving and may change in the future. Only examples of the UI for the Use Cases that have been implemented are shown below (UC 1, 2, 3, 4, 5, 7, 10, 11).

### 10.1 Interface Design

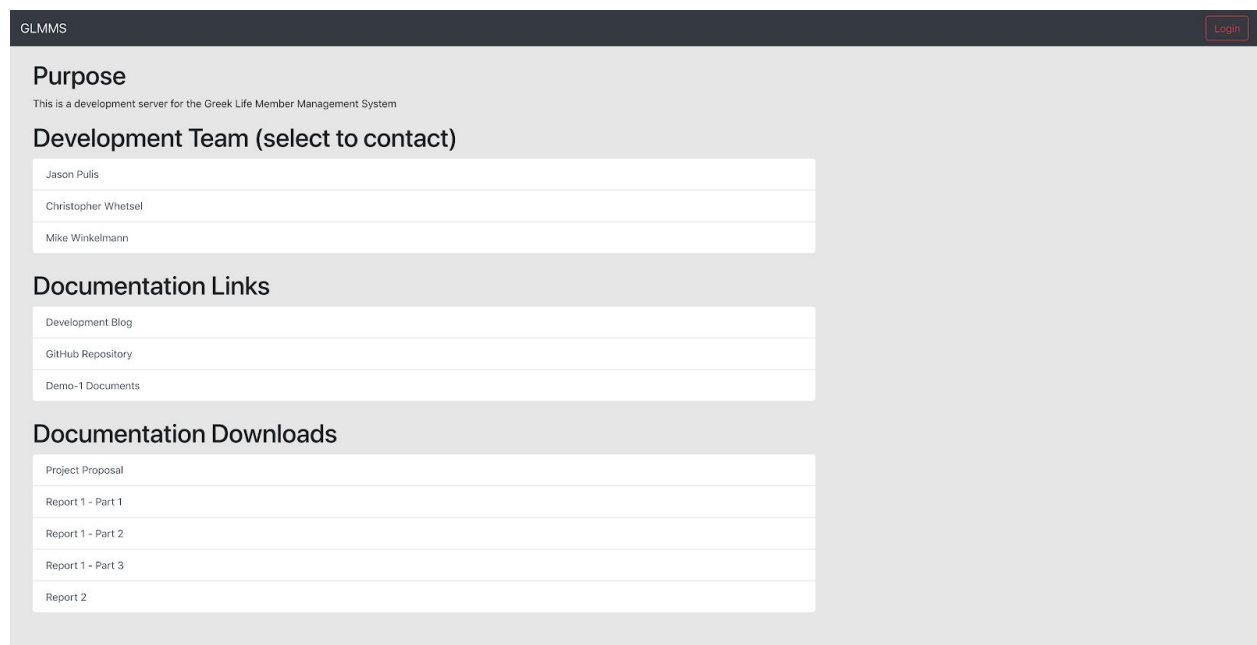


Figure 10.1.1 member homepage (not logged in)

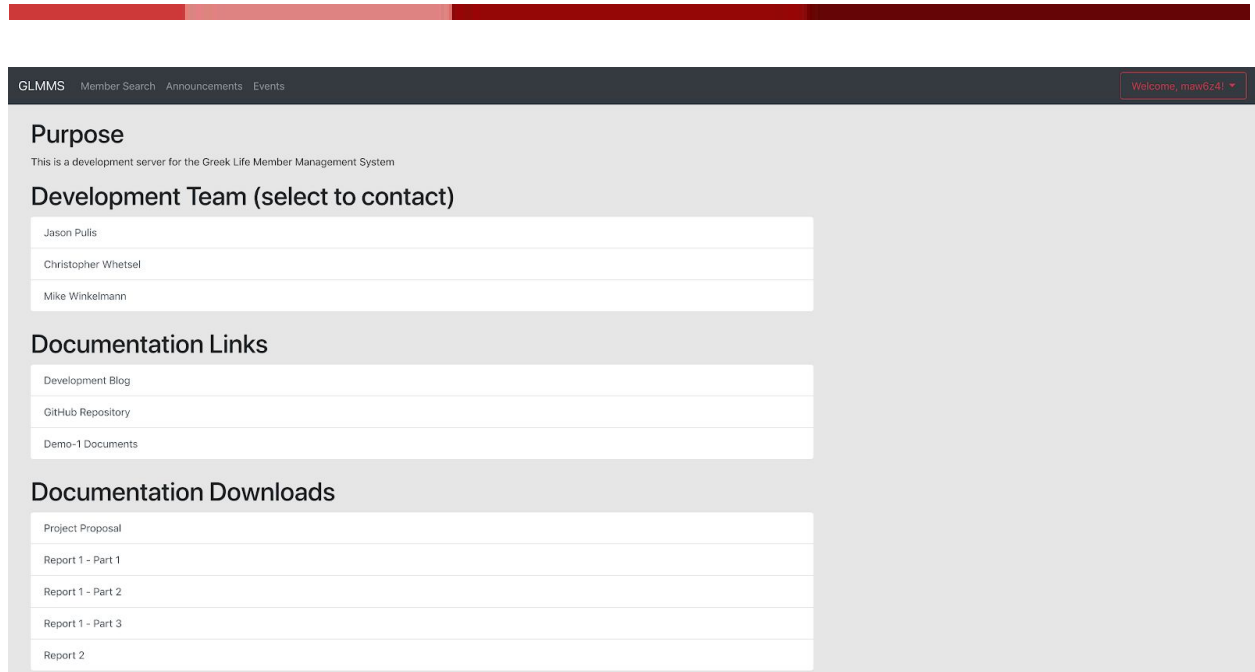


Figure 10.1.2 member homepage (logged in)

### 10.1.1 Use Case UC-1: Searching Member Information

This is the member search page. There will be multiple fields for narrowing search results from the database. The search will be initiated by a submit button at the bottom of the page. The member's information will be displayed below the search options.

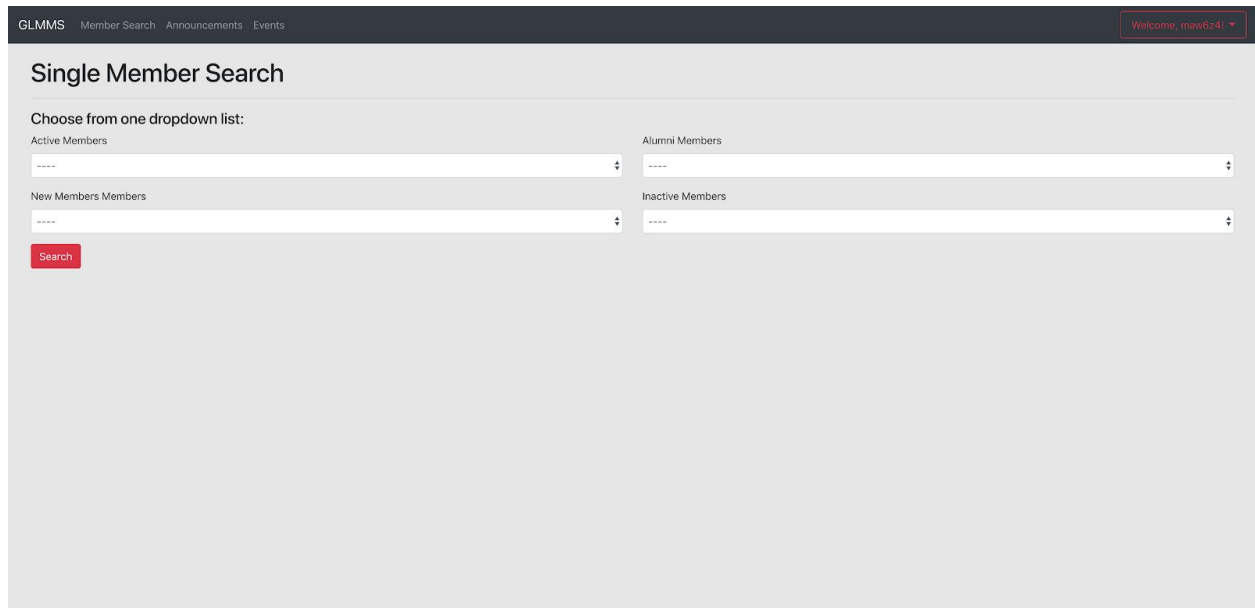


Figure 10.1.3 single member search (pre-search)



## Single Member Search

Choose from one dropdown list:

Active Members


Alumni Members

New Members Members

Inactive Members

Search

### Jason Pulis's Profile



**Biography**

**Email**  
jbp2pd@mail.missouri.edu

**LinkedIn Profile**  
[No profile](#)

phone number	address	pledge class	declared major	second major	emphasis	semester graduating	year graduating	shirt size	member status	cumulative GPA	last semester GPA	last update	birth date	last random pic date
3167377878	Columbia	N/A	Information Technology	N/A	N/A	F	2018	L	A	None	None	2018-10-30	N/A	N/A

Figure 10.1.4 single member search (post-search)

GLMMS Member Search Announcements Events

Welcome, max624!

## Multi-Member Search

Choose your search criteria from the dropdown lists below:

Member Group to Search?\*

All Members

What information to display?\*

Everything

phone number

address

pledge class

How to sort the results?\*

Username

Last Name

First Name

phone number

Search

Figure 10.1.5 multi member search (pre-search)

GLMMS
Member Search
Announcements
Events
Welcome, maw6z4

## Multi-Member Search

Choose your search criteria from the dropdown lists below:

Member Group to Search?\*

All Members

What information to display?\*

Everything  
phone number  
address  
pledge class

How to sort the results?\*

Username  
Last Name  
First Name  
phone number

Search

Number of Results: 5

Username	Name	Email	phone	address	pledge_class
user	User Loser	cjwhetsel@gmail.com	N/A	N/A	N/A
admin	Admin McAdminFace	a@e.com	N/A	N/A	N/A
jbp2pd	Jason Pulis	jbp2pd@mail.missouri.edu	3167377878	Columbia	N/A
cjwgr5	Christopher Whetsel	cjwgr5@mail.missouri.edu	3146816755	N/A	N/A
maw6z4	Mike Winkelmann	maw6z4@mail.missouri.edu	N/A	N/A	N/A

Figure 10.1.6 multi member search (post-search)

## 10.1.2 Use Case UC-2: Using Administrator Powers

### 10.1.2.1 Admin homepage

These pages demonstrates the extra abilities that members with administrative privileges will have. The extra abilities will take the administrator to the desired page. They can edit anything about any object in the database.

GLMMS administrations
WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Site administration

#### ANNOUNCEMENT

Announcements [+ Add](#) [Change](#)

#### AUTHENTICATION AND AUTHORIZATION

Groups [+ Add](#) [Change](#)

Users [+ Add](#) [Change](#)

#### EVENT

Events [+ Add](#) [Change](#)

#### MEMBER

Cell\_carriers [+ Add](#) [Change](#)

Companies [+ Add](#) [Change](#)

Job\_historys [+ Add](#) [Change](#)

Organization\_involvements [+ Add](#) [Change](#)

#### Recent actions

**My actions**

- [cjwgr5: Christopher Whetsel](#)  
Profile
- [maw6z4](#)  
User
- [jbp2pd](#)  
User
- [+ Something else](#)  
Announcement
- [+ Checking the datetime](#)  
Announcement
- [+ Moment of Truth](#)  
Announcement
- [+ Announcement object \(2\)](#)  
Announcement
- [+ Announcement object \(1\)](#)  
Announcement

Figure 10.1.7 Admin index page.

### 10.1.2.2 Add/Edit Members

The admin will also be able to create and edit users.

GLMMS administrations

WELCOME, ADMIN / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Authentication and Authorization > Users > Add user

### Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only.

Password:

Your password can't be too similar to your other personal information.  
Your password must contain at least 8 characters.  
Your password can't be a commonly used password.  
Your password can't be entirely numeric.

Password confirmation:

Enter the same password as before, for verification.

Save and add another Save and continue editing SAVE

Figure 10.1.8 Admin member creation.

The add member page gives the User many fields to enter including username, password, and email.

GLMMS administrations

WELCOME, ADMIN / VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Authentication and Authorization > Users > admin

### Change user

HISTORY

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only.

Password:

Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.

Personal info

First name:

Last name:

Email address:

Permissions

☒ Active

Figure 10.1.9 Admin member edit page.

The edit member page gives the user the ability to edit any User's information.

### 10.1.3 Use Case UC-3: Creating a Text Announcement

This case uses the same announcement system as that used in UC-7.

### 10.1.4 Use Case UC-4: Updating Member Information

This page is the profile page that a user will be able to see after logging in to the system. Their ability to edit their information will be accessed by clicking the edit link below their information.

A new page will be displayed to the user with a pre-populated form on it. When editing is complete the user will select the submit button to update the database and return to this updated screen.

GLMMS
Member Search
Announcements
Events
Welcome, maw6z4!

### Mike Winkelmann's Profile

**Biography**

**Email**  
maw6z4@mail.missouri.edu

**LinkedIn Profile**  
[No profile](#)

phone number	address	pledge class	declared major	second major	emphasis	semester graduating	year graduating	shirt size	member status	cumulative GPA	last semester GPA	last update	birth date	last random pic date
1234567890	N/A	N/A	Computer Science	N/A	N/A	F	2020	M	A	None	None	N/A	N/A	N/A

Edit Info

10.1.10 Member profile page.

GLMMS
Member Search
Announcements
Events
Welcome, maw6z4!

### Member Edit Info

Edit your information and click save:

First name: Mike
Last name: Winkelmann
Email address: maw6z4@mail.missouri.edu

Bio

Phone number: 1234567890
Address:

Pledge class:
Declared major: Computer Science
Second major:
Emphasis:

Shirt size: Medium
Member status: Active
Semester graduating: Fall
Year graduating: 2020

LinkedIn profile

Cumulative GPA:
Last semester GPA:
Last update:
Birth date:

Pledge father:
Cell carrier id\*: MetroPCS

Photo  
Currently: profile\_pics/maw6z4\_JetFuel\_v\_SteelBeams.png
Change: Choose File No file chosen

Save

10.1.11 Member profile edit page.

## 10.1.5 Use Case UC-5: Recovering Password

The following images are the pages that members will see when they wish to recover their account after a password is lost. Users will first enter the email for the account that they wish to recover. They will be sent an email with a link to a page to create a new password. After following the link they will be given access to the password reset page where they will enter their new password. Once complete, they will be shown a success page.

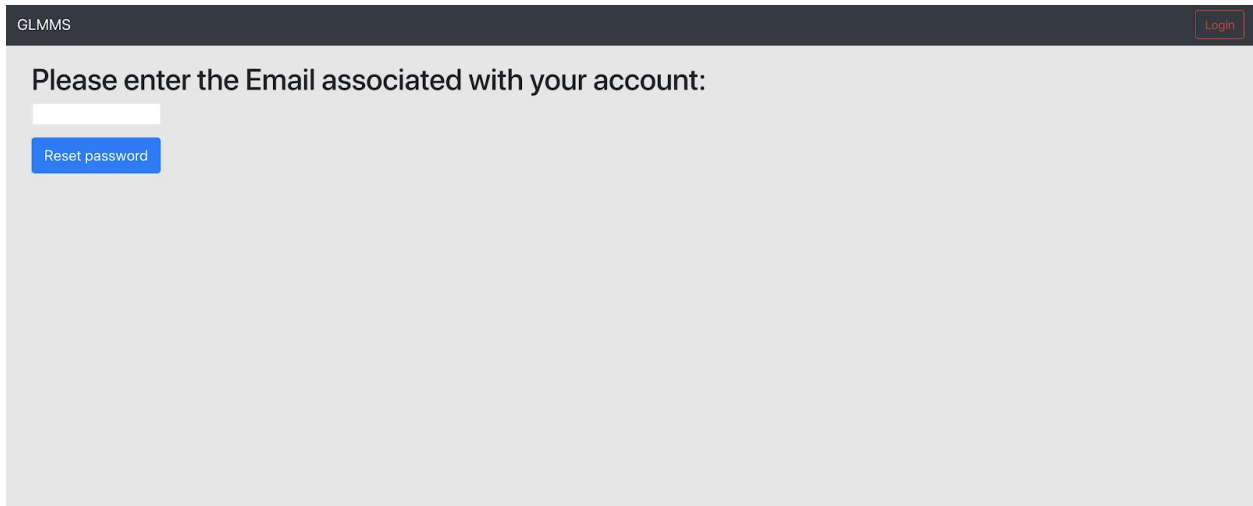
A screenshot of a web application interface for password recovery. At the top, a dark header bar contains the text 'GLMMS' on the left and a 'Login' button on the right. The main content area has a light gray background. It starts with the text 'Please enter the Email associated with your account:' followed by a white text input field. Below the input field is a blue button labeled 'Reset password'.

Figure 10.1.12 Password reset request page

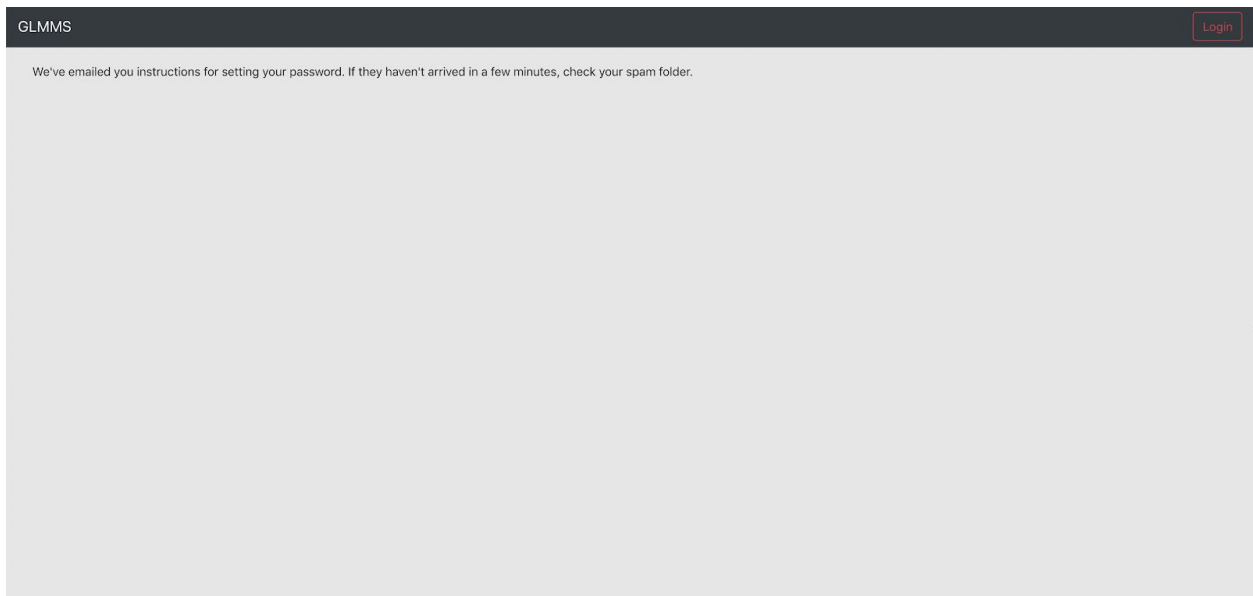
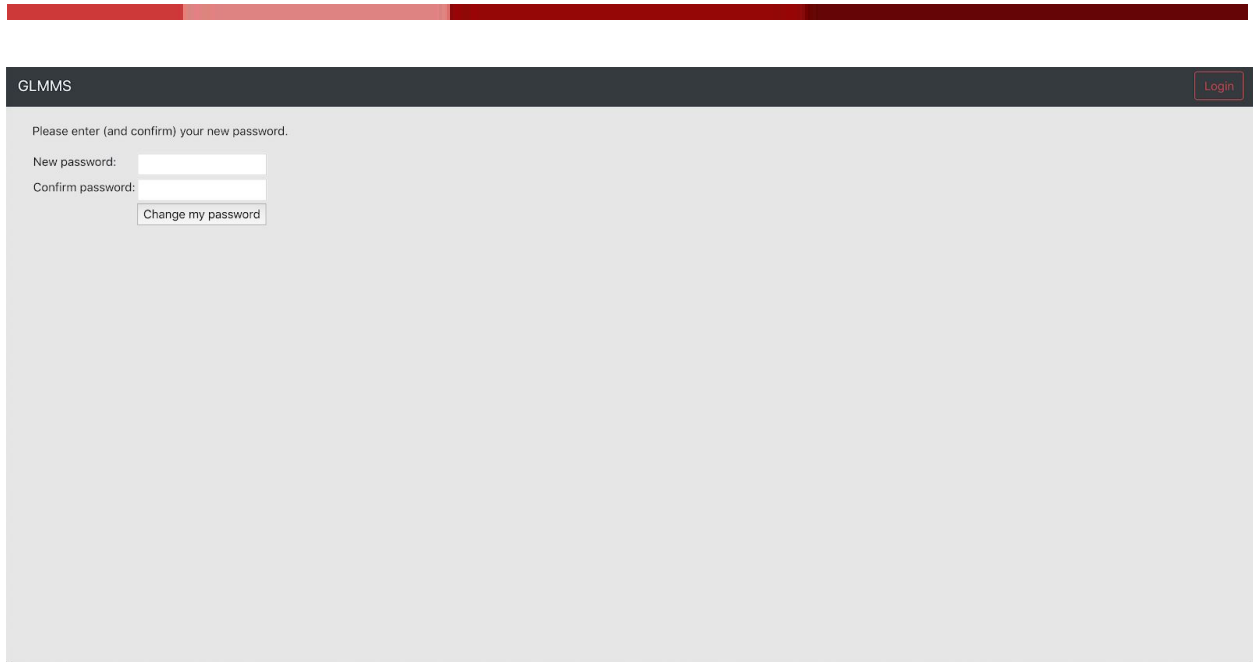
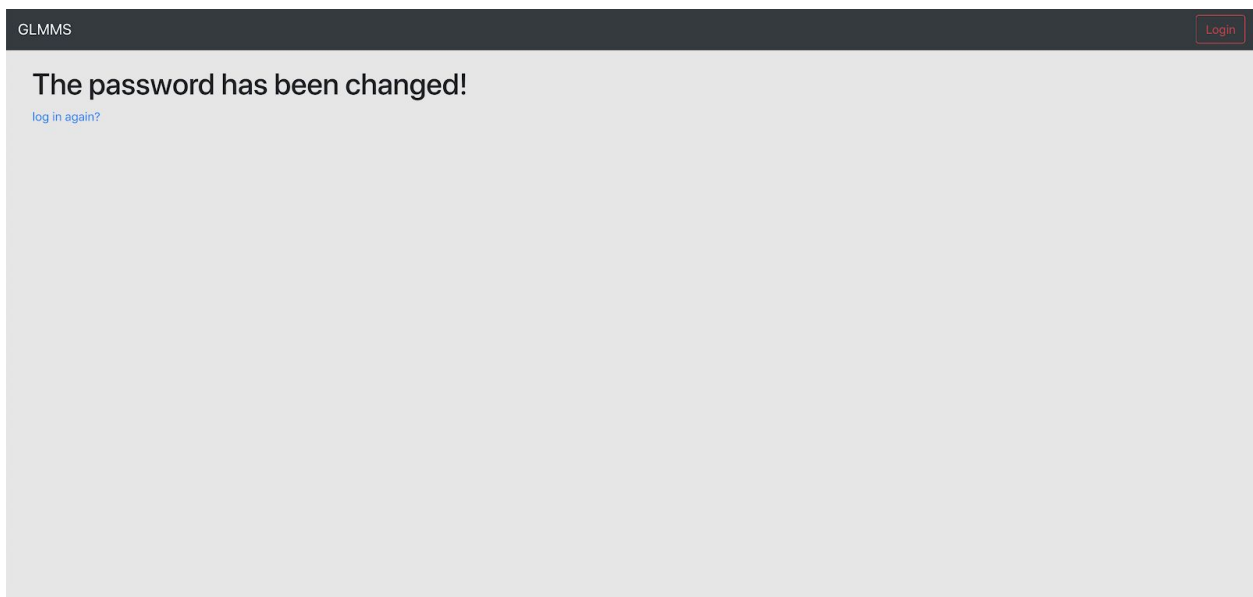
A screenshot of a web application interface showing a confirmation message. The dark header bar at the top contains 'GLMMS' on the left and a 'Login' button on the right. The main content area has a light gray background and displays the text: 'We've emailed you instructions for setting your password. If they haven't arrived in a few minutes, check your spam folder.'

Figure 10.1.13 Password reset email sent page.



The screenshot shows a web application interface for GLMMS. At the top, there is a dark header bar with the text "GLMMS" on the left and a "Login" button on the right. Below the header, the main content area has a light gray background. It contains the instruction "Please enter (and confirm) your new password." followed by two input fields: "New password:" and "Confirm password:". Below these fields is a button labeled "Change my password".

Figure 10.1.14 Password reset new password page.



The screenshot shows the same GLMMS web application interface, but now displaying a success message. The header bar remains the same. The main content area displays the text "The password has been changed!" in a large, bold font. Below this message is a smaller, blue link that says "log in again?".

Figure 10.1.15 Password reset success page.

### 10.1.7 Use Case UC-7: Creating and Viewing Announcements

The announcements page will have a fillable text area for creating a new announcement. The announcement will be posted to the Announcements index page upon selection of the submit button. In order to post the announcement and send a message to other members phones the `urgent` type option must be selected.

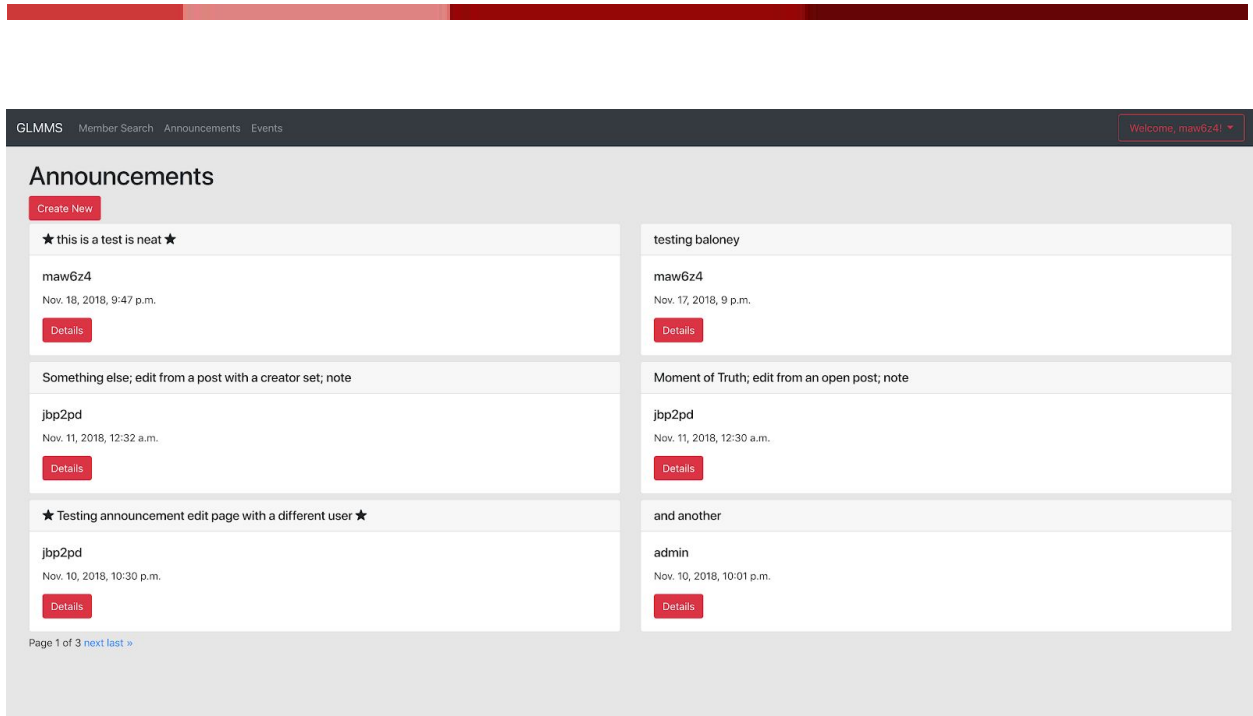


Figure 10.1.16 Announcement index page

The screenshot shows the 'New Announcement' form in the GLMMS website. The navigation bar at the top is identical to the previous page. The form title 'New Announcement' is followed by two dropdown menus for 'Type\*' (set to 'Normal') and 'Topic\*' (set to 'General'). Below these are input fields for 'Title\*' and a large text area for 'Message\*'. A red 'Save' button is positioned at the bottom left of the form area.

Figure 10.1.17 Announcement creation page

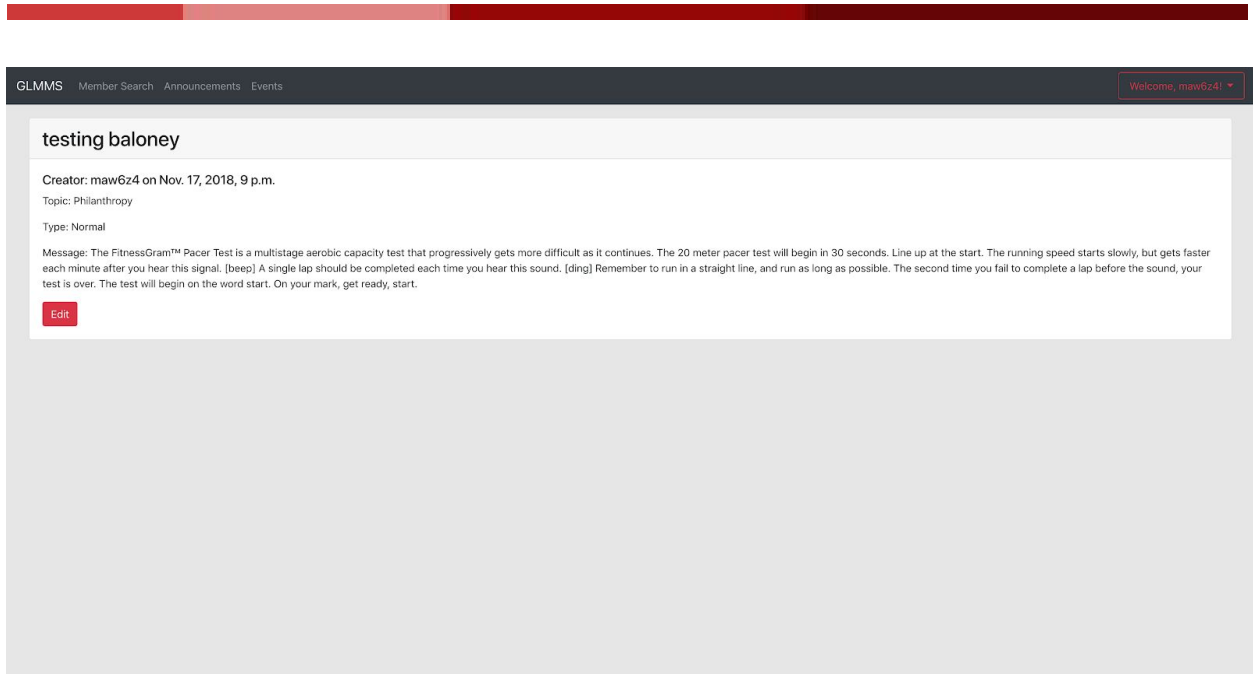


Figure 10.1.18 Announcement detail page

### 10.1.10 Use Case UC-10: Managing Events

On the Event index page, events will be shown in a list with the soonest events at the top. Press the 'Create New' button to create a new event. Users can use this page to create an event. There are options for title, subject, date, and location. Once completed, the User is shown the event detail page. The 'Show RSVP' button will show the names of members who have RSVP-ed. Users can also RSVP on this page.

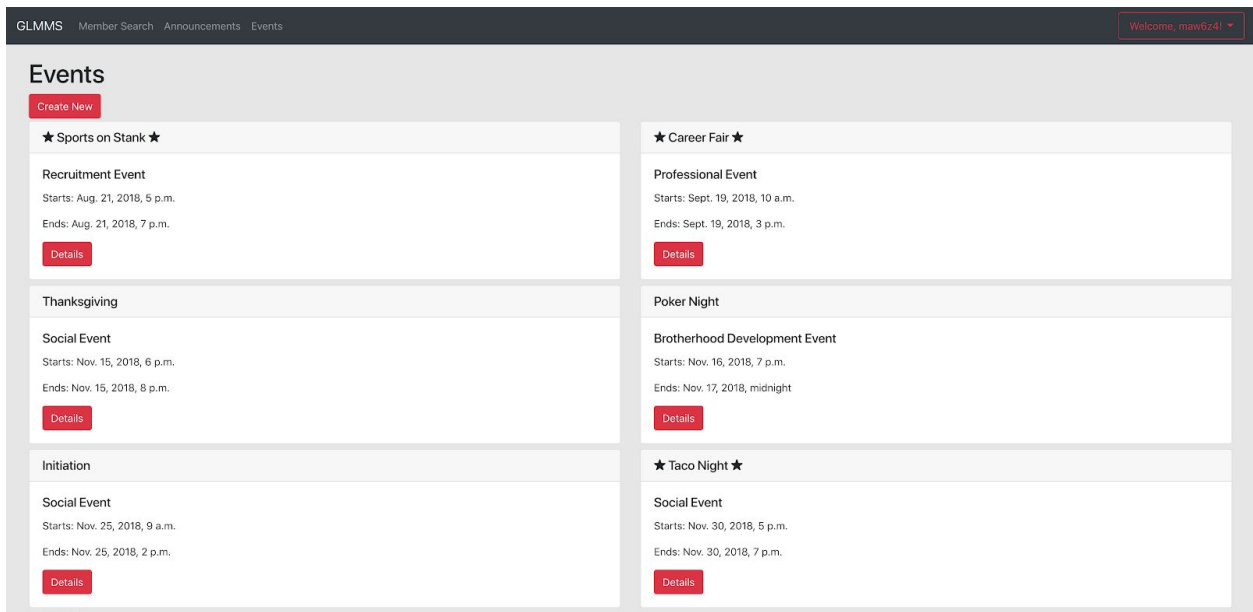


Figure 10.1.19 Event index page.



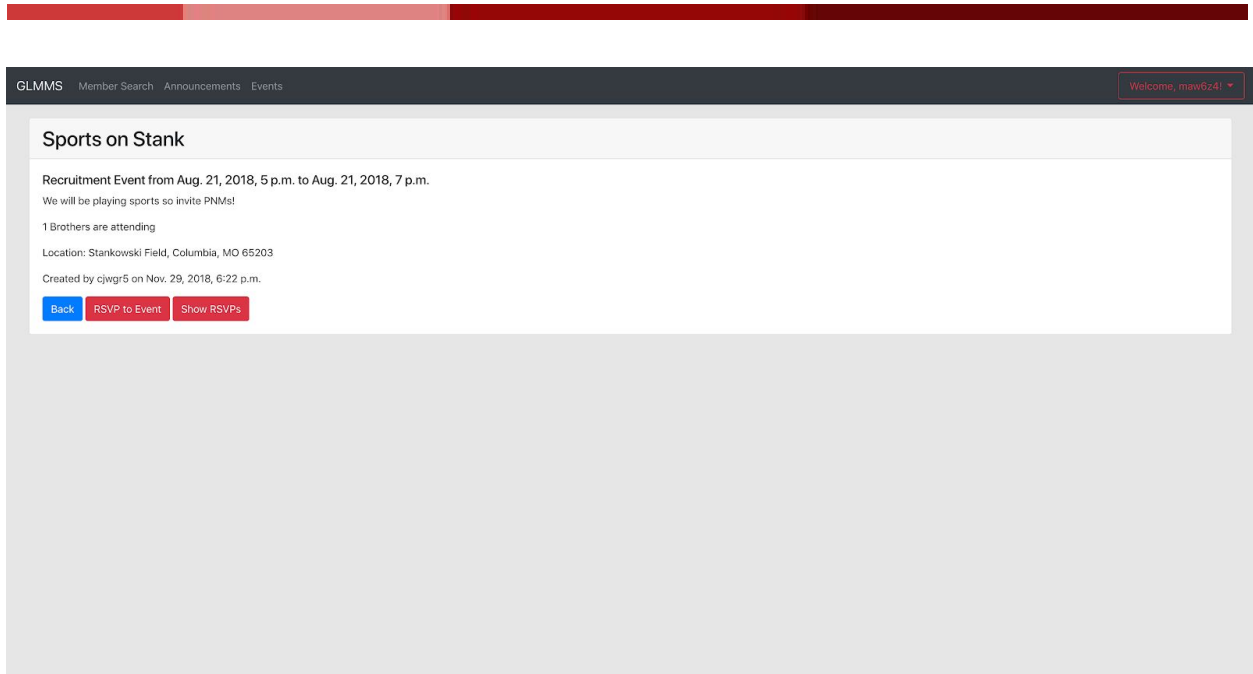


Figure 10.1.20 Event detail page.

Figure 10.1.21 Event creation page.

### 10.1.11 Use Case UC-11: Logging into the System

The user login will ask for the User's username and password with a submit button. A link will be below the submit button for account recovery detailed in UC-5.

GLMMS

Please login to see this page.

Username\*

Password\*

Login

[Lost password?](#)

Figure 10.1.22 Event creation page.

## 10.2 User Effort Estimation

The User effort estimation for several possible uses is shown below.

### 10.2.1 Scenario 1

Generate List of Active Members' Phone Numbers (UC-1):


1. Select "Member Search" from the navigation menu.
2. Select "Multi Member Search" tab.
3. Choose "Current Members" from the "Group" dropdown list.
4. Choose "Phone" from the "Info" dropdown list.
5. Choose "Last Name" from the "Sort" dropdown list.
6. Press "Search" button.
7. Press "Generate Spreadsheet" button

Clicks: ~10; Keystrokes: ~0

### 10.2.2 Scenario 2

Adding a Member (UC-2):

1. Select "Member Management" from the navigation menu.
2. Click the "Add Member" button.

- 
3. Click the “pawprint” field.
  4. Enter the pawprint.
  5. Click the email field.
  6. Enter the email
  7. Click “Create” button.
  8. Click “Confirm” button.

Clicks: ~6; Keystrokes: ~30

### **10.2.3 Scenario 3**

Send Text Announcement that Chapter has Moved (UC-3):

1. Select “Announcements” from navigation menu.
2. On “Announcements” page, enter your message in the “New Announcement” field.
3. Select a topic from the dropdown list.
4. Click “Urgent Text” button.
5. Click the “Confirm” button.

Clicks: ~5; Keystrokes: ~150

### **10.2.4 Scenario 4**

User Login To System (UC-11):

1. From spdmizzou.com homepage, select “Login”
2. Click the “username” field
3. Enter username
4. Click the “password” field
5. Enter password
6. Press <Enter> to login

Clicks: ~3; Keystrokes: ~20

## 11 Design of Tests

### 11.1 Test Cases

<b>TC-01</b> <b>Use-Case tested: UC-01</b> <b>Searching Member Information</b>
<p><b>Test Covers:</b></p> <ul style="list-style-type: none"><li>• Database searches</li></ul> <p><b>Assumption:</b></p> <ul style="list-style-type: none"><li>• Database is populated with information and user is on the search member page</li></ul> <p><b>Steps:</b></p> <ul style="list-style-type: none"><li>• Enter search criteria in text entry area</li><li>• Select search button</li></ul> <p><b>Expected:</b></p> <ul style="list-style-type: none"><li>• Data will be returned to the screen with correct search criteria represented</li></ul> <p><b>Fails if:</b></p> <ul style="list-style-type: none"><li>• If no information is returned or information that does not match search criteria is returned</li></ul>

<b>TC-02a</b> <b>Use-Case tested: UC-2</b> <b>Using Administrator Powers</b>
<p><b>Test Covers:</b></p> <ul style="list-style-type: none"><li>• Add new members</li></ul> <p><b>Assumption:</b></p> <ul style="list-style-type: none"><li>• Administrator has proper login credentials and is able to login</li></ul> <p><b>Steps:</b></p> <ul style="list-style-type: none"><li>• Select users in main menu</li><li>• Select a user</li><li>• Enter user information</li><li>• Select Save</li></ul> <p><b>Expected:</b></p> <ul style="list-style-type: none"><li>• User will be added to the users page</li></ul>

**Fails if:**

- A new user is not added to the users page

**TC-02b**  
**Use-Case tested: UC-2**  
**Using Administrator Powers**

**Test Covers:**

- Change member information

**Assumption:**

- Administrator has proper login credentials and is able to login

**Steps:**

- Select users in main menu
- Select a user
- Update user information
- Select Save

**Expected:**

- User information is updated on the users page

**Fails if:**

- Information is not updated on the users page

**TC-03**  
**Use-Case tested: UC-03**  
**Creating a Text Announcement**

**Test Covers:**

- Send urgent announcements via text message from the System to all active members

**Assumption:**

- Active members have provided their cell phone carrier information in their profile, the user has permissions for creating announcements, and are on the announcement page

**Steps:**

- Enter a message in the textbox
- Select the urgent text button

**Expected:**

- A message is sent to active members cell phones

**Fails if:**

- Messages are not sent to active members cell phones or are also sent to inactive

members

**TC-04**  
**Use-Case tested: UC-04**  
**Updating Member Information**

**Test Covers:**

- Users updating their personal information

**Assumption:**

- Users are logged in and are on their profile page

**Steps:**

- Select the edit button
- Update information in the editable areas
- Select the save button

**Expected:**

- Users information will be updated on profile page

**Fails if:**

- Users information is not updated on the profile page or is updated incorrectly

**TC-05**  
**Use-Case tested: UC-05**  
**Recovering Password**

**Test Covers:**

- Recovering a user account after forgotten password

**Assumption:**

- User is at the main page and has access to the email address that is on file

**Steps:**

- Select lost password button
- Enter email address
- Select reset password button
- Follow the link in the email sent to the email address on file
- Enter new password in the new password text field
- Enter same password in confirm password text field
- Select change my password button
- Perform login attempt as described in TC-9

**Expected:**

- Users password will be reset

**Fails if:**

- Users email is not reset

**TC-06**  
**Use-Case tested: UC-06**  
**Picking random members**

**Test Covers:**

- Randomly choose members from the Database

**Assumption:**

- Users have permissions for selecting random members and are on the random member selector screen

**Steps:**

- Select the number of members needed from the dropdown menu
- Select submit button

**Expected:**

- A list of members will be populated

**Fails if:**

- A list of members is not generated, the wrong number of members are generated, or inactive members are included in the generations

**TC-07a**  
**Use-Case tested: UC-07**  
**Creating and Viewing Announcements**

**Test Covers:**

- Create an announcement

**Assumption:**

- The user has permissions for creating announcements, and are on the announcement page

**Steps:**

- Enter a message in the textbox
- Select the submit button

**Expected:**

- A new announcement will be added to the announcements page

**Fails if:**

- A new announcement is not created or the announcement is created incorrectly

**TC-07b**  
**Use-Case tested: UC-07**  
**Creating and Viewing Announcements**

**Test Covers:**

- access a filterable announcements feed with topics

**Assumption:**

- User is on the announcement page

**Steps:**

- The announcements are auto generated on the page for view
- Select a filter from the filter side bar
  - or
- Select a hashtag from an announcement
  - or
- Enter search results in the search textbox
- Select the submit button

**Expected:**

- The desired filter option will display the information based on the filter that was used

**Fails if:**

- One or more filter options does not function or returns incorrect results

**TC-08**  
**Use-Case tested: UC-10**  
**Managing Events**

**Test Covers:**

- Create an event that users can RSVP to and the event creator can send email reminders to the RSVP-ed users. Created events will be linked to the fraternity Google Calendar

**Assumption:**

- Members have an active email address in the system, user has permissions to create an even, user is on the event management page

**Steps:**

- Select members to invite from the who dropdown menu
- Enter information into the what textarea
- Choose a date from the when-date dropdown menu
- Choose a time from the when-time dropdown menu
- Enter an address in the where textarea
- Select submit button
  - or
- Select previously made event reminder button

**Expected:**



- An email containing the information will be sent to all selected members with an embedded RSVP link that sends a response to the system, the Calendar App is updated to reflect the new event, and a reminder email is sent prior to the event.

**Fails if:**

- Emails are not sent at all, emails are not sent to the selected members, emails are sent to non-selected members, emails contain incorrect information, RSVP's do not return to the system, Calendar App is not updated or incorrect information is updated, reminder email is not sent prior to the event or not sent to correct people

**TC-09**  
**Use-Case tested: UC-11**  
**Login**

**Test Covers:**

- Login procedure

**Assumption:**

- User is already populated in the database

**Steps:**

- Enter Username and Password
- Select Login button

**Expected:**

- User will be logged into the system if the credentials are valid and denied if they are invalid

**Fails if:**

- Login passes with incorrect info or fails with correct info

The remainder of our program does not need a test, due to the visual nature of this project. When a user requests information it should immediately be populated on the screen. Failure of this will be seen immediately and will have the ability to be fixed without any extra testing needed.

## 11.2 Test Coverage

The testing coverage has two objectives. The first is to ensure that input is handled correctly and the system handles the invalid inputs correctly. The second is to ensure that no errors are the fault of the software. To ensure this we have developed multiple tests that cover the full range of user interaction with the GLMMS. We have created these tests to account for all pass/fail possibilities and will add more testing in the future if it becomes obvious that it is necessary to more thoroughly test these cases.



## 11.3 Integration Testing Strategy

When the units are complete we will slowly integrate a few units together at a time to ensure proper integration of the whole system. We plan to complete the database first and test if we can properly store information into it. While this is happening we will also make sure that our website is working properly such that we can create accounts and ensure the proper information is being stored into the database.

## 12 History of Work

### 12.1 History of Work

The major milestones for this project were Project Proposal, Report 1, Report 2, Demo-1, Report 3, and Demo 2. We were able to meet the deadlines for the Proposal, Reports 1 and 2, and we completed the implementation of Use cases 1, 2, 3, and 11 for Demo 1. We are on pace to complete this report on time and we are confident we will be able to demo the rest of the chosen Use cases for Demo-2. The original Plan of Work from Report 2 is below. A full list of weekly meeting agendas can be found on the development blog.

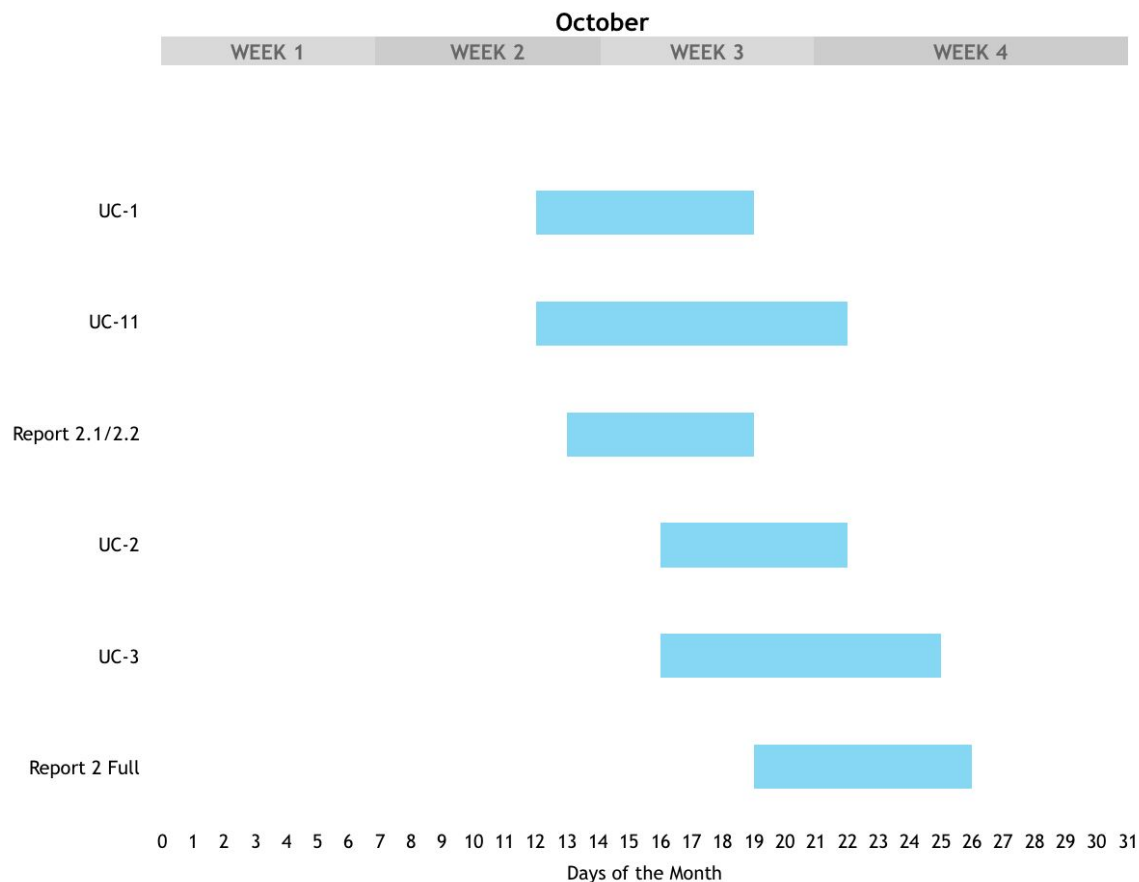


Figure 12.1.1: Original Gantt Chart for October

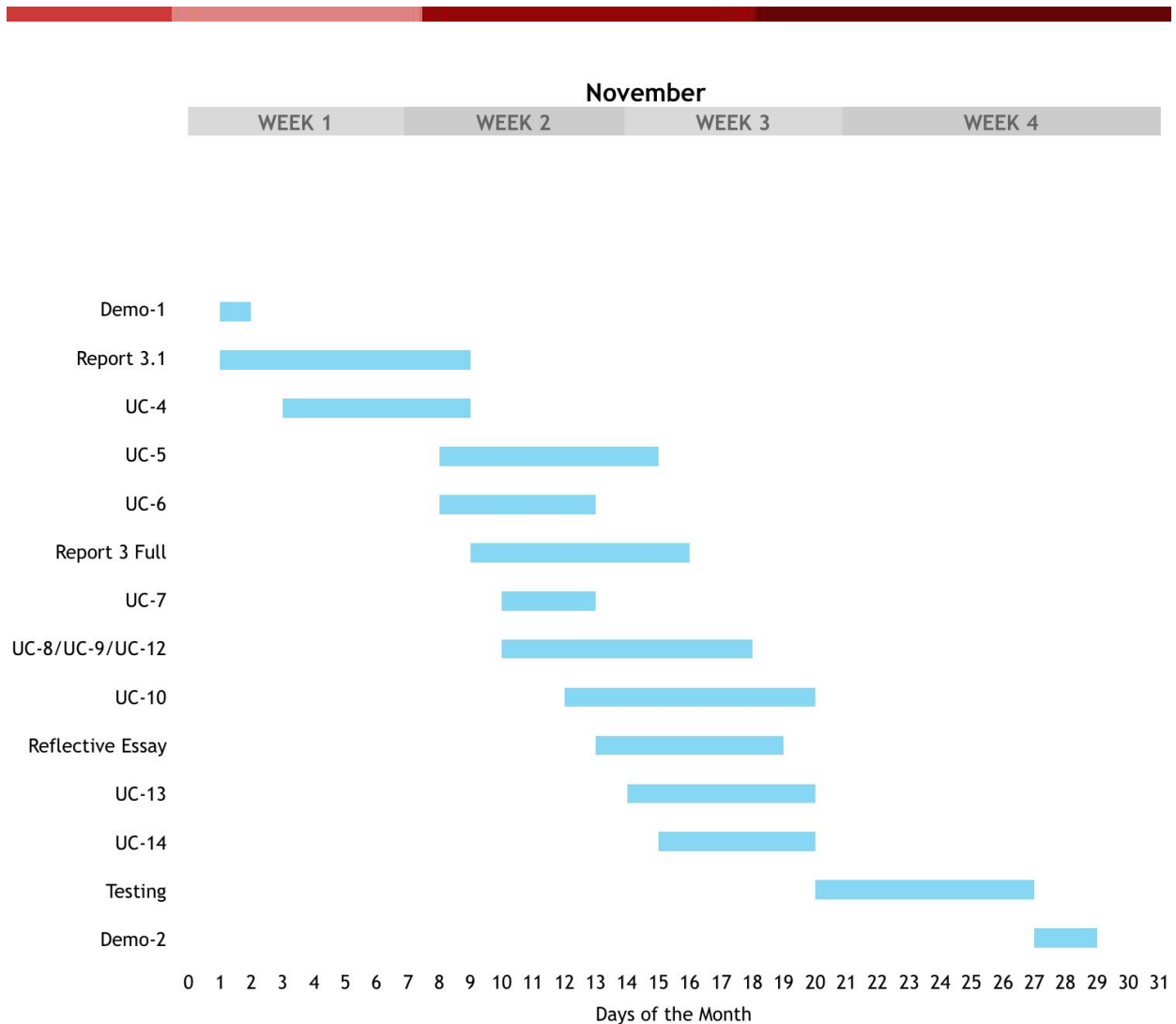



Figure 12.1.2: Original Gantt Chart for November

We were able to meet the deadlines as described in the charts above for Demo-1 and Report 2. However, the development of the Use Cases in the October chart ran long for all of them. They ran up until the day before Demo-1. We were fighting to find time to dedicate to development because we all had difficult schedules this semester and most of the time we set aside for this class went to the writing of the reports and other documentation. That being said, we did meet the deadline for the demo, but well behind our planned development deadlines.

After Demo-1 we realized that our original goals for this project were far too ambitious for the size of our group and the time we had to commit to this project. We scaled back the planned Use cases significantly for Demo-2. This helped but did not eliminate the problems with missing development deadlines. We were fortunate that the deadline for Report 3 was moved back to November 30th and the Demo-2 date was also pushed back by a week. We are well positioned to finish Report 3 by the deadline, but are sliding on our planned deadlines for the remaining Use Cases 4, 5, 7, 10. The week for Testing was removed from the plan in favor of continuous testing as the code was developed which has given us some room to work with the schedule. Use Cases



4, 5, and 7 are nearly finished ahead of schedule, but Use Cases 3 and 10 are behind. We plan to have them finished two days before our Demo (demo is December 6th) in order to practice for the demo and perform exploratory testing on the System.

Overall, we have encountered consistent problems with being behind schedule, but are confident that with dedication we will be ready for Demo-2.

## 12.2 Accomplishments

- We created a functional web application using the Django framework and Bootstrap.
- Designed a polished UI.
- Wrote an extensive Project Report detailing the design and implementation of our System.
- We learned the Django framework and the Model View Template architecture style without ever using it before.
- Successfully leveraged git to manage complex group-developed code base.

## 12.3 Future Work

We plan to continue developing this project after the class is over for Sigma Phi Delta. We plan to implement those test cases that were listed in Section 3.3 Use Cases (UC 6, 8, 9, 12, 13, 14) but not implemented. Also, we will continue to improve the features already implemented. Some possible improvements would be to provide filters for the announcements list, add the spreadsheet generation to the member search, and link the events to a calendar. We will also continue to look into new ways to design the UI to make it pretty and easy to use. There is still a lot that can be done with this project in the future.

## 12.4 Breakdown of Final Responsibilities

Jason Pulis

- Maintains the Development server and blog
- User Interface Specification
- Domain Model
- Association Definitions
- Product Ownership
- Gantt Chart
- Design of Tests
- Effort Estimation Using Use Case Points
- Developed the Login System, Password Recovery, Administrator Powers, and the Announcements app.

Christopher Whetsel

- Problem Diagnosis
- Enumerated Requirements
- Stakeholders
- Casual Description
- UC Traceability Matrix

- 
- Fully Dressed Use Cases
  - Concept Definitions
  - System Operations Contracts
  - Breakdown of Responsibilities
  - Interaction Diagrams
  - Class Diagrams
  - Design Patterns
  - History of Work
  - Developed Member application and Events application.

Mike Winkelmann

- Solution
- Onscreen Appearance Requirements
- Actors and Goals
- Use Case Diagrams
- System Sequence Diagrams
- Attribute Definitions
- Timeline
- System Architecture
- Project Management and Plan of Work
- Exploratory Testing of System
- Designed the UI for the whole application.
- Developed Member application.

## 13 References

1. "Use case", Wikipedia [http://en.wikipedia.org/wiki/Use\\_case](http://en.wikipedia.org/wiki/Use_case)
2. "System requirements", Wikipedia [http://en.wikipedia.org/wiki/System\\_requirements](http://en.wikipedia.org/wiki/System_requirements)
3. "User interface specification", Wikipedia [http://en.wikipedia.org/wiki/User\\_interface\\_specification](http://en.wikipedia.org/wiki/User_interface_specification)
4. "System Sequence Diagram", Wikipedia [https://en.wikipedia.org/wiki/System\\_sequence\\_diagram](https://en.wikipedia.org/wiki/System_sequence_diagram)
5. "Software Engineering book", Ivan Marsic [http://www.ece.rutgers.edu/~marsic/books/SE/book-SE\\_marsic.pdf](http://www.ece.rutgers.edu/~marsic/books/SE/book-SE_marsic.pdf)
6. "Sigma Phi Delta National - Home" <https://sigmaphidelta.2stayconnected.com/>
7. "ΣΦΔ Beta-Omicron" <http://spdmizzou.com/>
8. "FitBit: Health Monitoring Analytics" <http://www.ece.rutgers.edu/~marsic/books/SE/projects/HealthMonitor/2014-g5-report3.pdf>
9. "HeartBPM" <http://www.ece.rutgers.edu/~marsic/books/SE/projects/HealthMonitor/2014-g12-report3.pdf>
10. "Restaurant Automation" <http://www.ece.rutgers.edu/~marsic/books/SE/projects/Restaurant/2015-g3-report3.pdf>
11. "Gantt Chart" <https://www.teamgantt.com/free-gantt-chart-excel-template>
12. "Model View Controller", Wikipedia <https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>
13. "Django" <https://www.djangoproject.com/>
14. "Python" <https://www.python.org/>
15. "Educational Networking Tool for College Students" <http://eceweb1.rutgers.edu/~marsic/books/SE/projects/OTHER/2015-g9-report3.pdf>
16. "Object Constraint Language" <https://www.omg.org/spec/OCL/2.0/>
17. "Software Design Patterns", Wikipedia [https://en.wikipedia.org/wiki/Software\\_design\\_pattern](https://en.wikipedia.org/wiki/Software_design_pattern)
18. "Front controller", Wikipedia [https://en.wikipedia.org/wiki/Front\\_controller](https://en.wikipedia.org/wiki/Front_controller)
19. "Command pattern", Wikipedia [https://en.wikipedia.org/wiki/Command\\_pattern](https://en.wikipedia.org/wiki/Command_pattern)
20. "Module pattern", Wikipedia [https://en.wikipedia.org/wiki/Module\\_pattern](https://en.wikipedia.org/wiki/Module_pattern)
21. "Servant (design pattern)", Wikipedia [https://en.wikipedia.org/wiki/Servant\\_\(design\\_pattern\)](https://en.wikipedia.org/wiki/Servant_(design_pattern))