Greek Life Member Management System

10.26.2018

Software Engineering 1 - Group 9 - Report 2
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

1 Interaction Diagrams

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

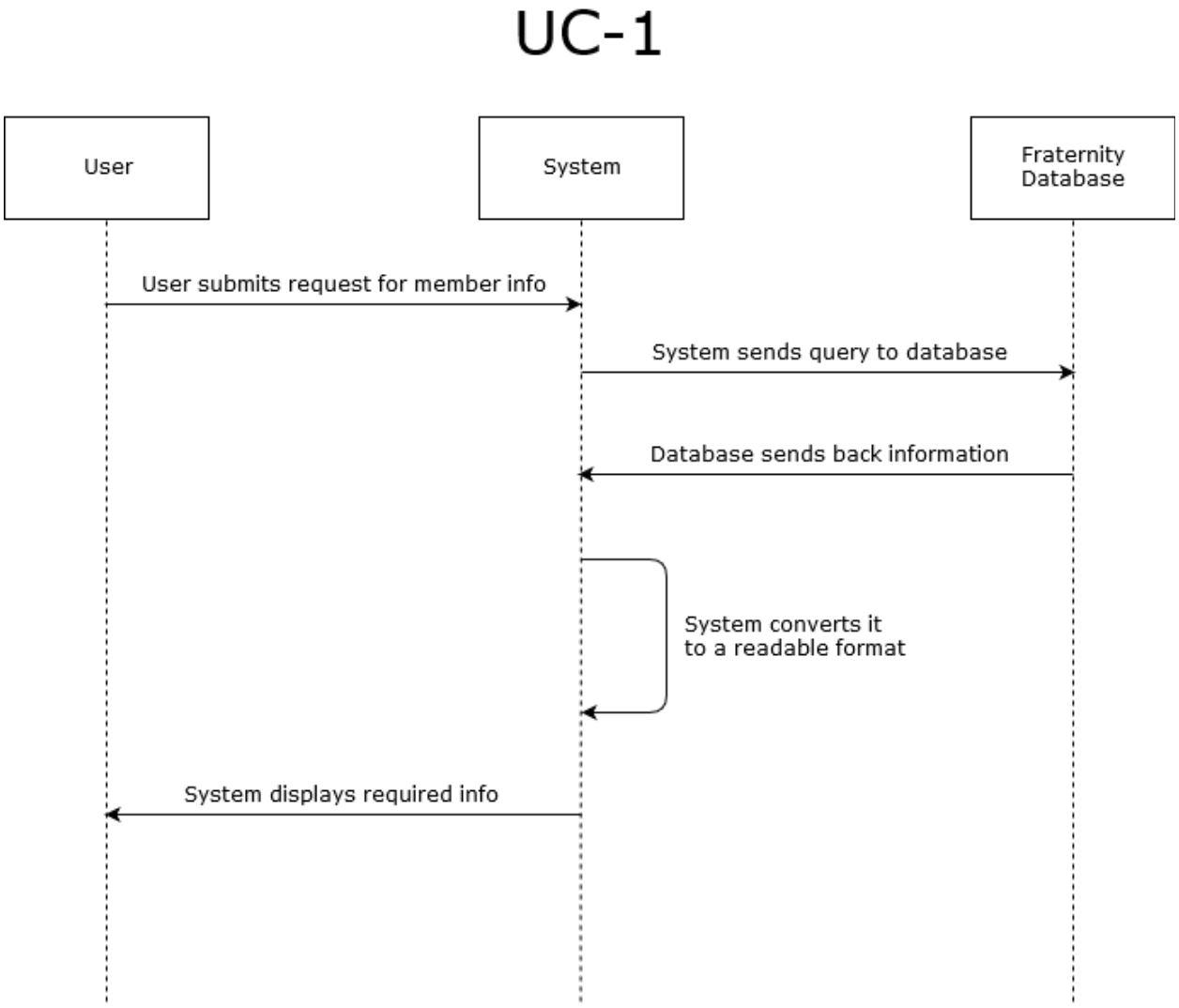

Figure 1.1 System Sequence Diagram for Use Case 1

Figure 1.1 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.

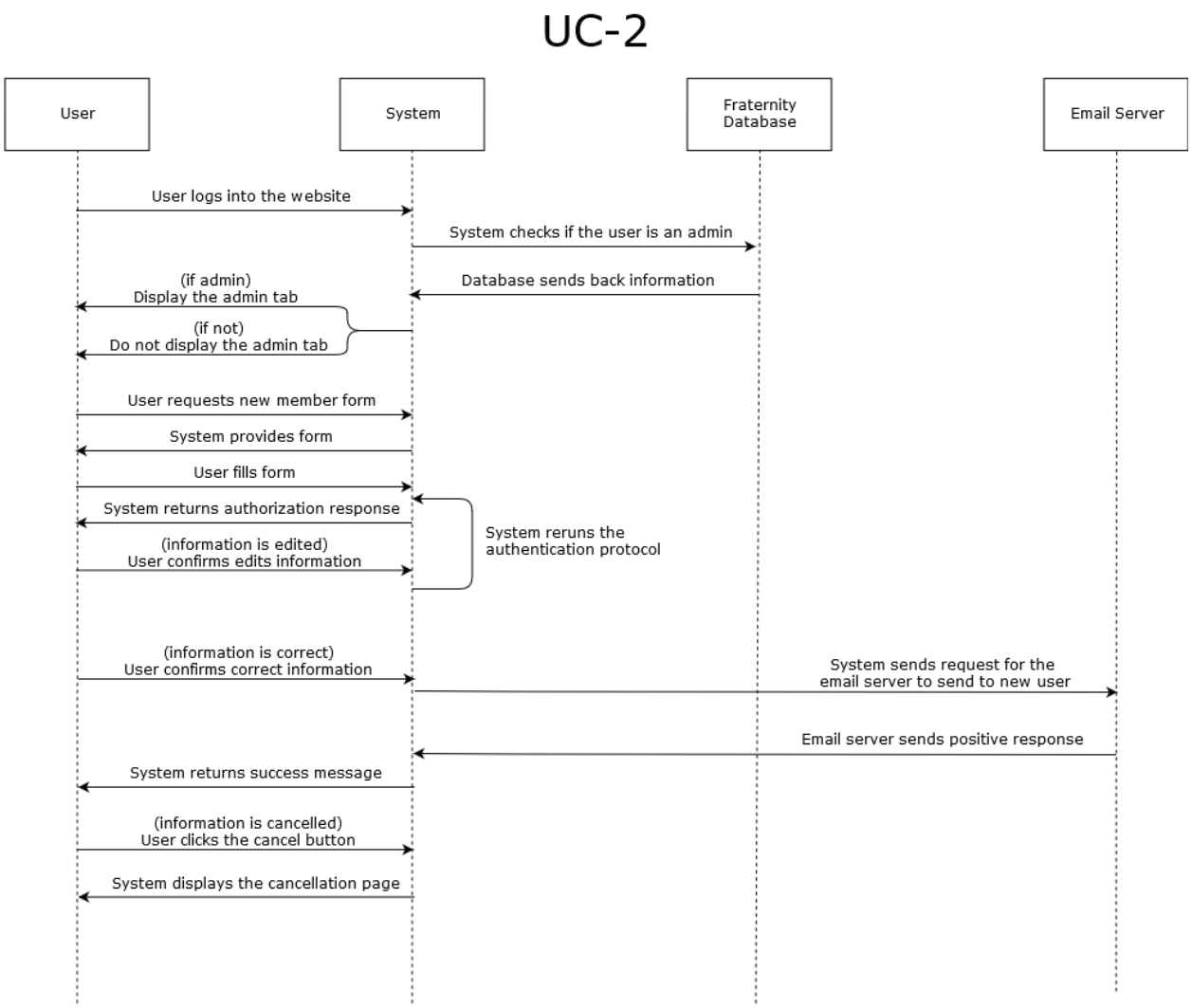

Figure 1.2 System Sequence Diagram for Use Case 2

Figure 1.2 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**).**

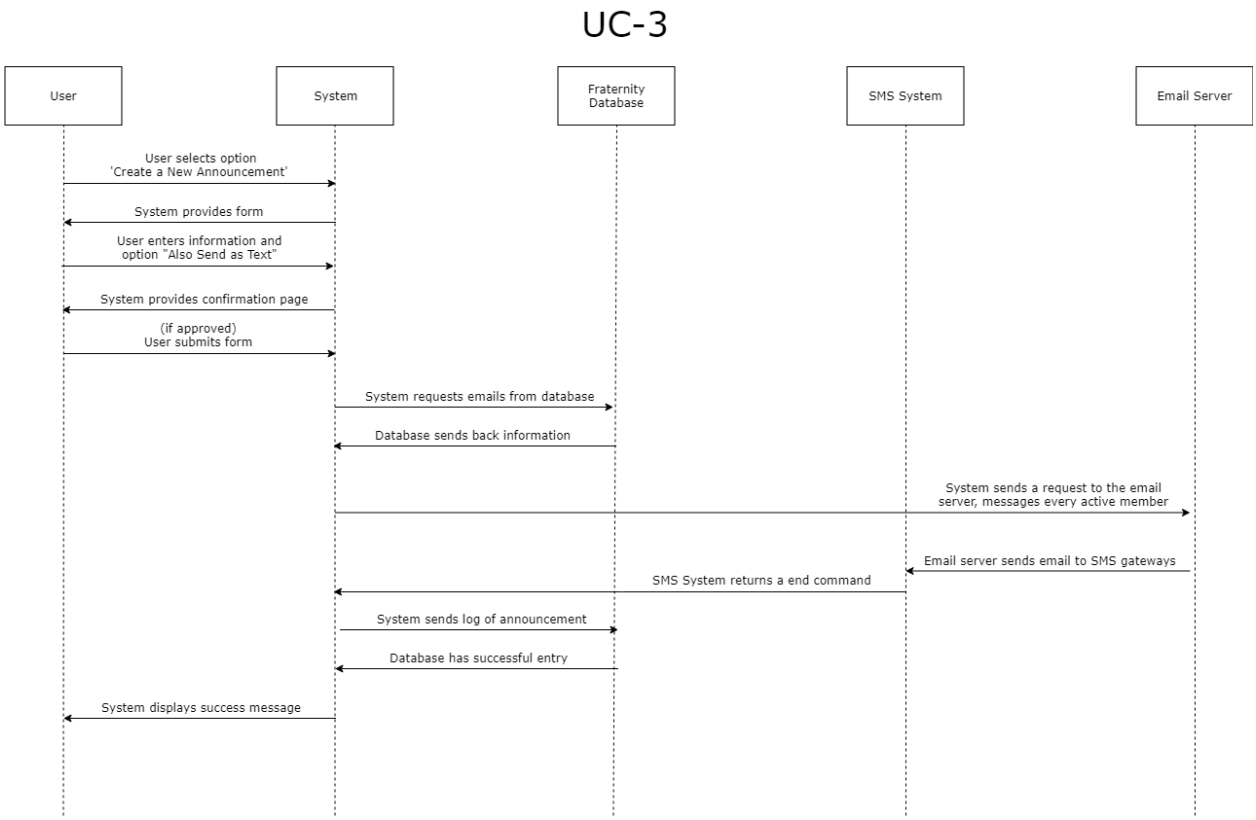

Figure 1.3 - Sequence Diagram of Use Case 3

Figure 1.3 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**)**.

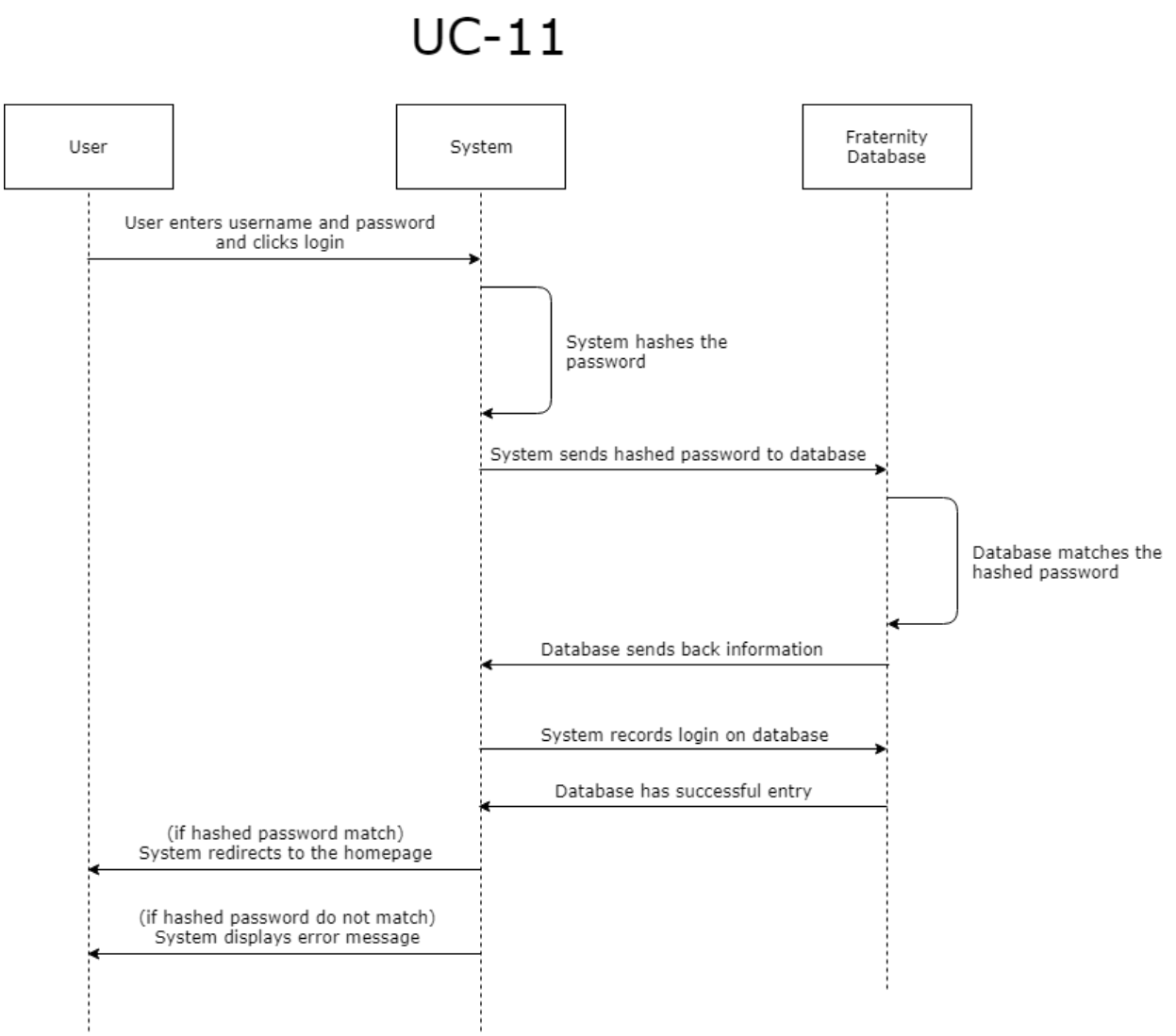

Figure 1.4 - Sequence Diagram of Use Case 11

Figure 1.4 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**)**.

2 Class Diagrams and Interface Specifications

2.1 Class Diagram

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

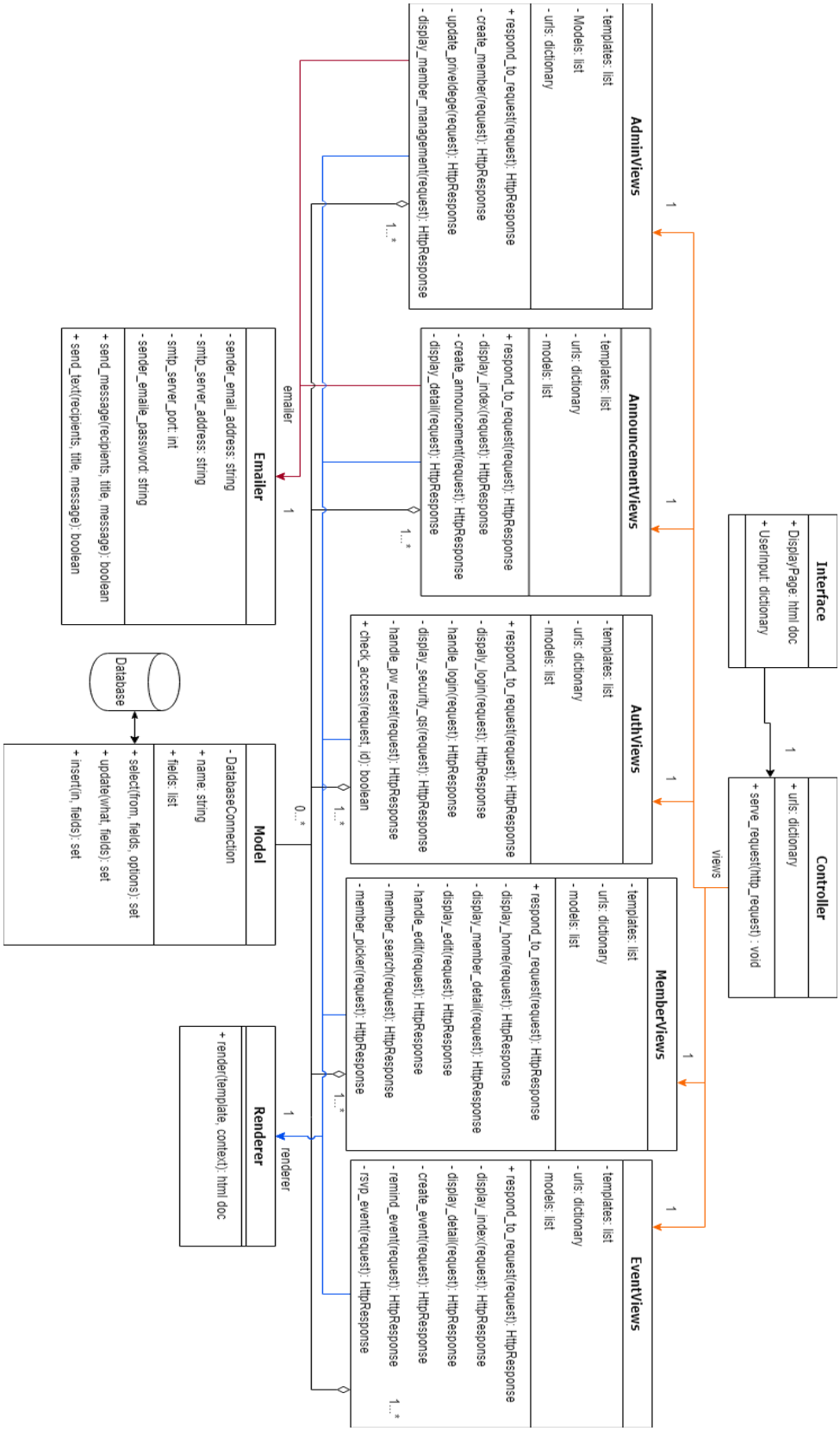

Figure 2.1.1 - Class Diagram. Zoom in for better view.

2.2 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
Attributes: ● displayPage: html doc; The displayPage is the html document currently displayed to the user. ● 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.
Methods ● N/A

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

Controller
Attributes: ● urls: dictionary; The urls attribute is a dictionary, mapping each url that the user can request to the correct View to handle the request.
Methods ● 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.

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

AdminViews
Attributes: ● templates: list; The templates attribute is a list of templates for how to display data for each webpage this view is responsible for creating. ● models: list; The models attribute is a list of Models representing the database tables that this View interacts with. ● urls: dictionary; The urls attribute is a dictionary that maps a url to the appropriate method to respond to that url.
Methods ● 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.
- update_priveldege(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.
- 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.

AnnouncementViews
Attributes: ● templates: list; same as AdminViews.templates. ● models: list; same as AdminViews.models. ● urls: dictionary; Same as AdminViews.urls.
Methods ● respond_to_request(request): HttpResponse; same as AdminViews.respond_to_request(request) ● display_index(request): HttpResponse; This method generates the index page for announcements. ● create_announcement(request): HttpResponse; This method handles the request to create a new announcement using the Announcements Model. Responds with either a success or failure page. ● display_detail(request): HttpResponse; This method responds with a page showing the full details for a requested announcement.

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

AuthViews
Attributes: ● templates: list; same as AdminViews.templates. ● models: list; same as AdminViews.models. ● urls: dictionary; Same as AdminViews.urls.
Methods ● respond_to_request(request): HttpResponse; same as AdminViews.respond_to_request(request) ● dispaly_login(request): HttpResponse; This method responds with the User login

page for the System.
- handle_login(request): HttpResponse; 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.
- display_security_qs(request): HttpResponse; This method generates the password reset page with the security questions and the new password fields.
- handle_pw_reset(request): HttpResponse; This method checks the answers to the security questions and updates the Auth Model with the new password. It responds with a success or failure page.
- 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 **MeberViews** class is the View responsible for responding to actions that all levels of Users can execute on the System.

MemberViews
Attributes: ● templates: list; same as AdminViews.templates. ● models: list; same as AdminViews.models. ● urls: dictionary; Same as AdminViews.urls.
Methods ● respond_to_request(request): HttpResponse; same functionality as AdminViews.respond_to_request(request) ● display_home(request): HttpResponse; This method responds with the homepage of the application. ● display_member(request): HttpResponse; This method responds with a page displaying the full details of a member using the Member Model. ● display_edit(request): HttpResponse; This page generates the form to edit a member's info based on the Member Model. ● handle_edit(request): HttpResponse; 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. ● member_search(request): HttpResponse; 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. ● member_picker(request): HttpResponse; This method generates a random list of active members and responds with a web page displaying them.

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

EventViews
Attributes: ● templates: list; same as AdminViews.templates. ● models: list; same as AdminViews.models. ● urls: dictionary; Same as AdminViews.urls.
Methods ● 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 **Emailer** class is responsible for encapsulating the functionality to send emails and SMS messages to members.

Emailer
Attributes: ● sender_email_address: string; This is the email address to use to send the emails ● smtp_server_address: string; This is the address of the SMTP server to use to send the email. ● smtp_server_port: int; This is the port number on the SMTP server to send the message to. ● sender_email_password: string; This is the password to the email address being used to send the emails.
Methods ● send_message(recipients, title, message): boolean; This method is invoked to send an email to the list of recipients with the provided title and message. ● send_text(recipients, title, message): 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.

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.

Model
Attributes: ● databaseConnection: A connection to the database. ● name: string; The name attribute is the name of the database table. ● fields: list; The fields attribute is a list of tuples of the columns' names and data types in the database.
Methods ● 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.

Renderer
Attributes: ● N/A
Methods ● 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.

2.3 Traceability Matrix

	Class									
Domain Concepts	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						X				
SearchOperator						X				
ElectionTallier										
FileArchiveManager										
MemberInfoUpdater						X				
Announcement Manager				X						
EventManager							X			
PositionHolderController		X								
OfficerReportFormSubmitter										
ExecController		X								
ElectionCreator										
ProfileCreator			X							
AdminController		X								
PriveledgeUpdater			X							
FamilyTreeGenerator										

Table 2.3.1 - 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 conepts -> 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 LoginVerifer 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 3. More information about MVC pattern can be found at the links in Section 8. References of this report.

Some concepts have not been converted to classes yet. This is because we are under heavy time constraints due to the constant workload of generating Reports for this class. Thus, some Use Cases were determined to be of lesser value to the project and have been deferred until we can determine if development time will allows us to complete them. These Use Cases are UC-8 , UC-9, UC-12, UC-13, UC-14. The concepts corresponding to these use cases are FamilyTreeGenerator, ElectionCreator, OfficerReportFormSubmitter, FileArchiveManager, and ElectionTallier.

3 System Architecture and System Design

3.1 Architecture Style

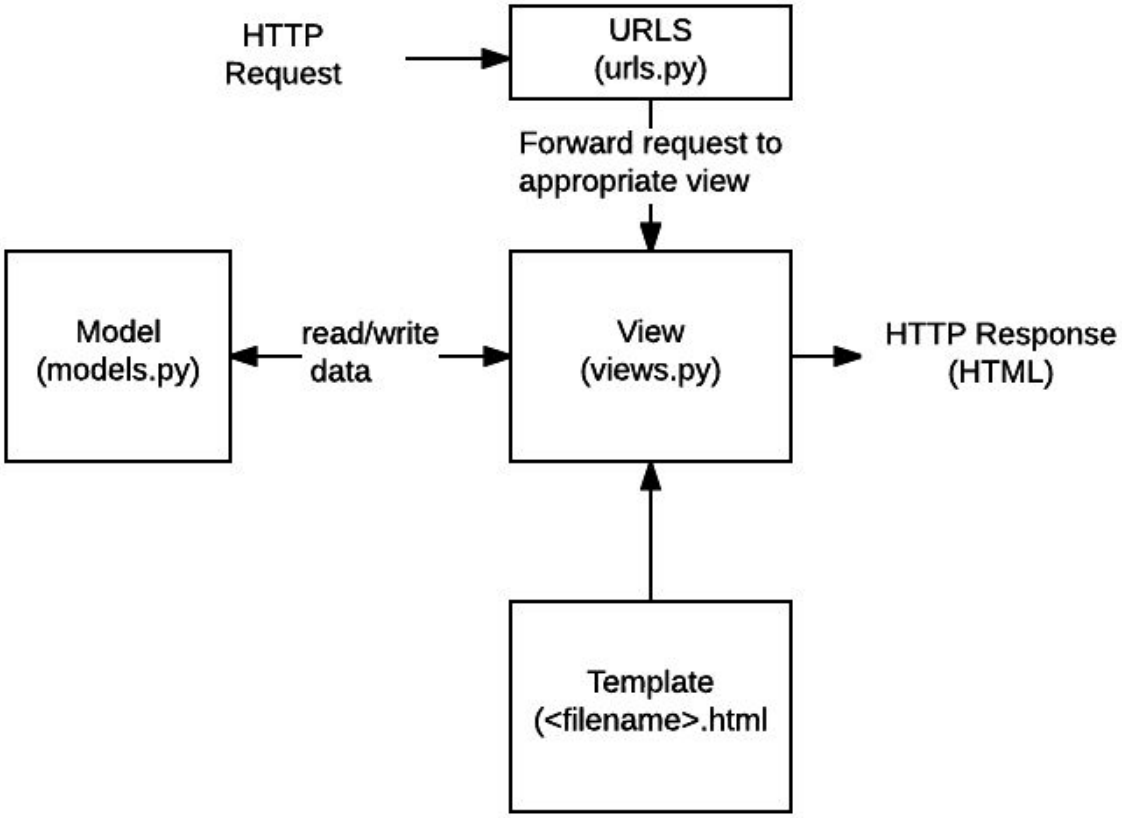

Figure 3.1.1 - Architecture Style

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.[12] 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.

3.2 Subsystems

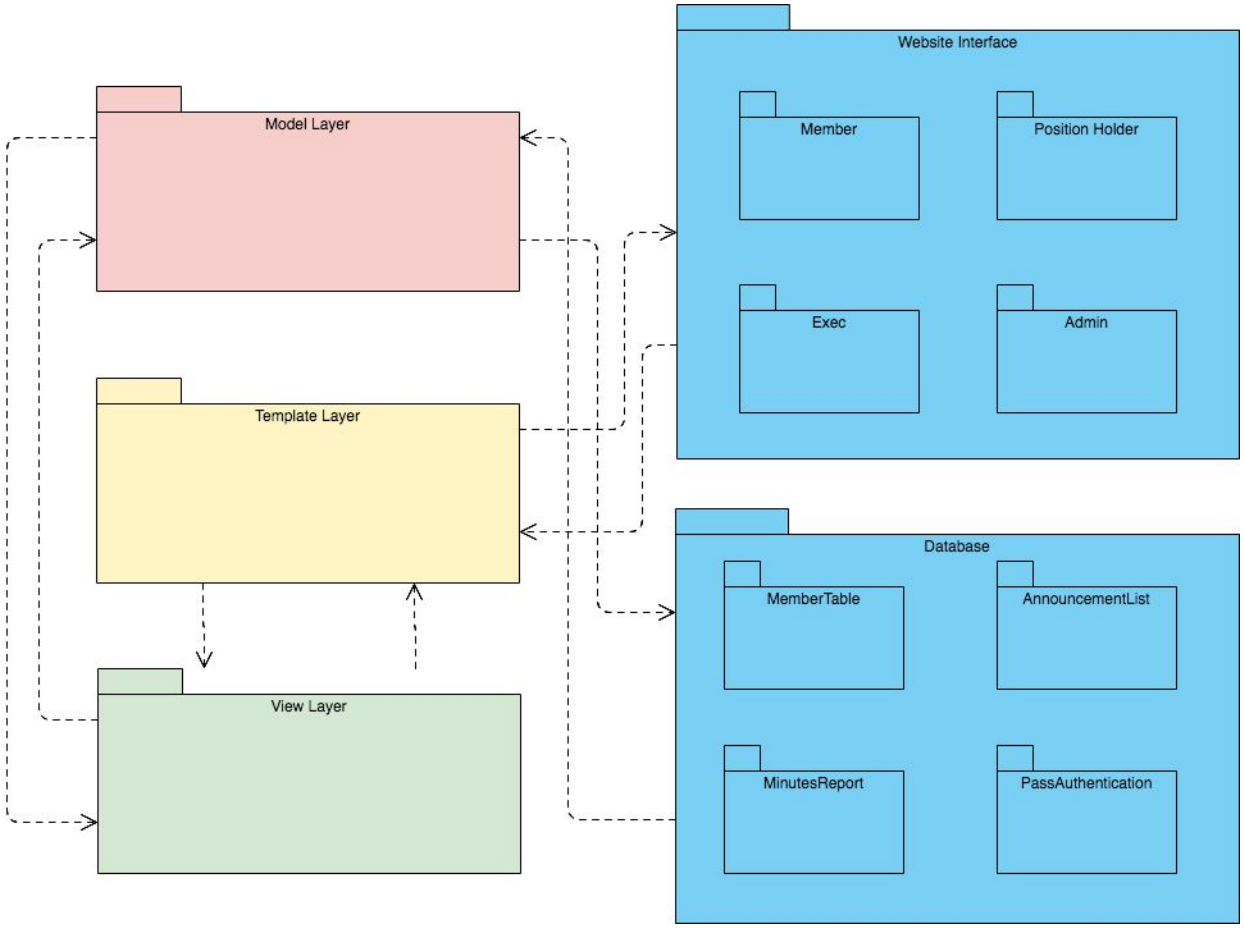

Figure 3.2.1 - Subsystems Used in the System

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 DatabaseConncetor. 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.

3.3 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.

3.4 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.

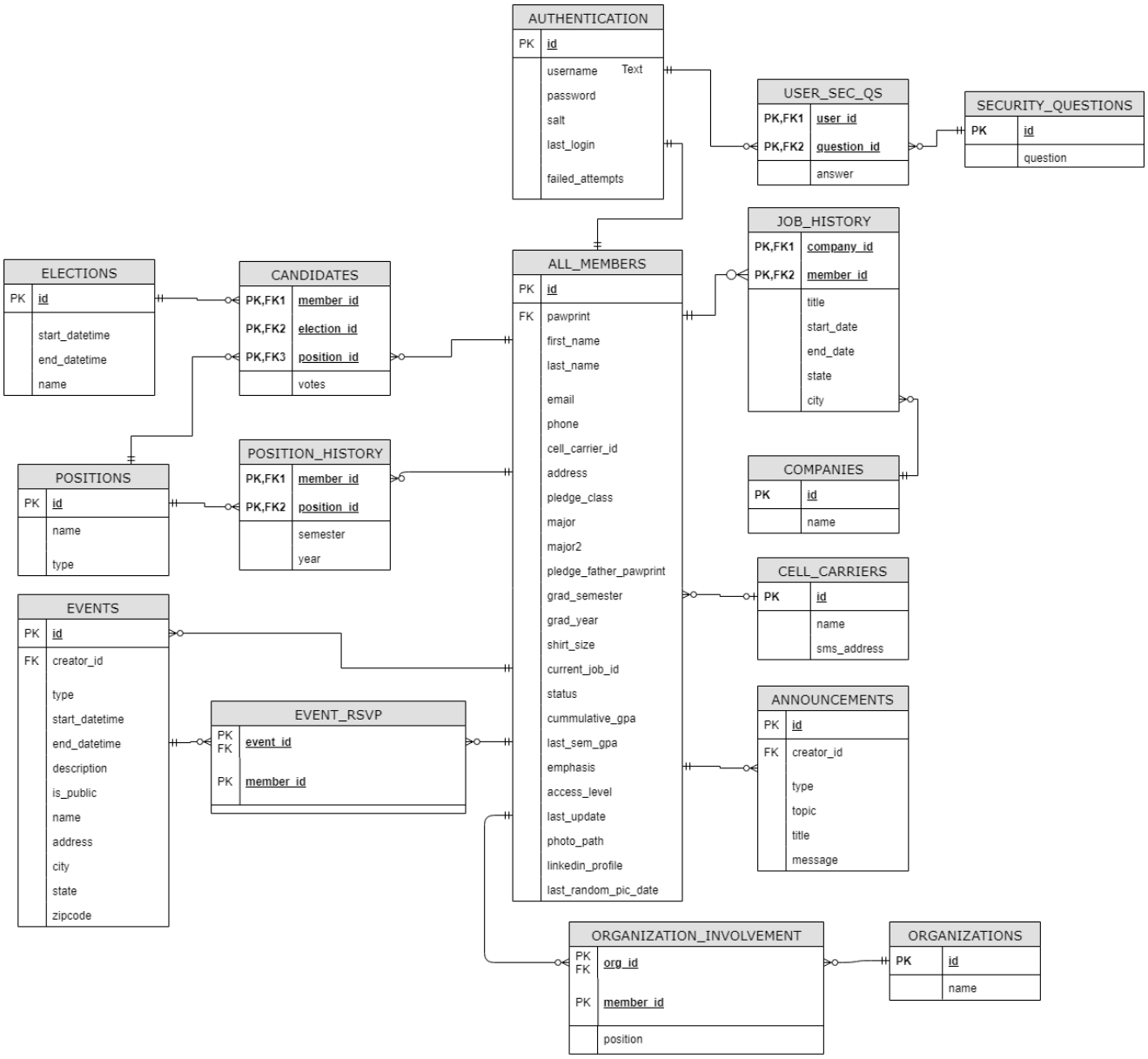

Figure 3.4.1 - Database Schema. Zoom for a better view.

3.5 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.

3.6 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.

3.7 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.

4 Algorithms and Data Structures

4.1 Algorithms

There are no mathematical models to implement in this project.

4.2 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.

5 User Interface Design and Specification

No significant changes since the preliminary designs in Report 1 Section 4, User Interface Specification.

6 Design of Tests

6.1 Test Cases

TC-01 **Use-Case tested: UC-01** **Searching Member Information**
Test Covers: • Database searches **Assumption:** • Database is populated with information and user is on the search member page **Steps:** • Enter search criteria in text entry area • Select search button **Expected:** • Data will be returned to the screen with correct search criteria represented **Fails if:** • If no information is returned or information that does not match search criteria is returned

TC-02a **Use-Case tested: UC-2** **Using Administrator Powers**
Test Covers: • Add new members **Assumption:** • Administrator has propper login credentials and is able to login **Steps:** • Select users in main menu • Select a user • Enter user information • Select Save

Expected:
- User will be added to the users page

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 propper 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
- Preform 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 Google Calendar 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, Google Calendar 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.

6.2 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.

6.3 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 it working properly such that we can create accounts and ensure the proper information is being stored into the database.

7 Plan Of Work

7.1 Merging Individual Contributions

This document was written individually and in groups throughout the project development period. Google Docs allowed us to have one single stream of content without having any conflicts of interest. Group members would occasionally change other group members work, but that was usually after discussion in a group meeting. The only issues that members would run into is confusion on content another group member was working on, this was easily solved by a quick explanation.

7.2 Project Coordination and Project Report

Due to our group only having three members, the practical application is still very much in development for Demo-1. The report writing has taken up all of the group members' time, which has had an impact on the practical project.

The functionality that we have in place is that there is a web application that is visible, with links to the required blog, contact information and documentation in place. The groups' major focus of the class has been getting the reports in on time, detracting from our ability to develop the System.

As of this report, none of the use cases have been met, but we plan to have the minimum functionality for the fully dressed use cases (UC-1, UC-2, UC-3, U-11) by Demo 1. However, this is dependent on the volume of paperwork that is required along with this demo.

7.3 Timeline

Our plan of work involves increasing coding efforts and continuing documentation work on the report deliverables each week. We will work in week long sprint cycles starting and ending at our meetings on Tuesday nights. The plan is to devote the week days to documentation of the reports and the weekends to coding for the project. The most important item to finish right now is the Use Cases for Demo-1. Each team member will be assigned a separate Use Case of the System so that if one member fails, it will not hinder all team members. For Demo-1 we plan to complete the four most important Use Cases 1,2,3, and 11. After Demo-1, we will assess our progress on the project and decide what we will be able to reasonably accomplish by the end of the semester. This Gantt Chart serves as a roadmap for our plan of milestones for the rest of the semester.

7.3.1 Gantt Charts

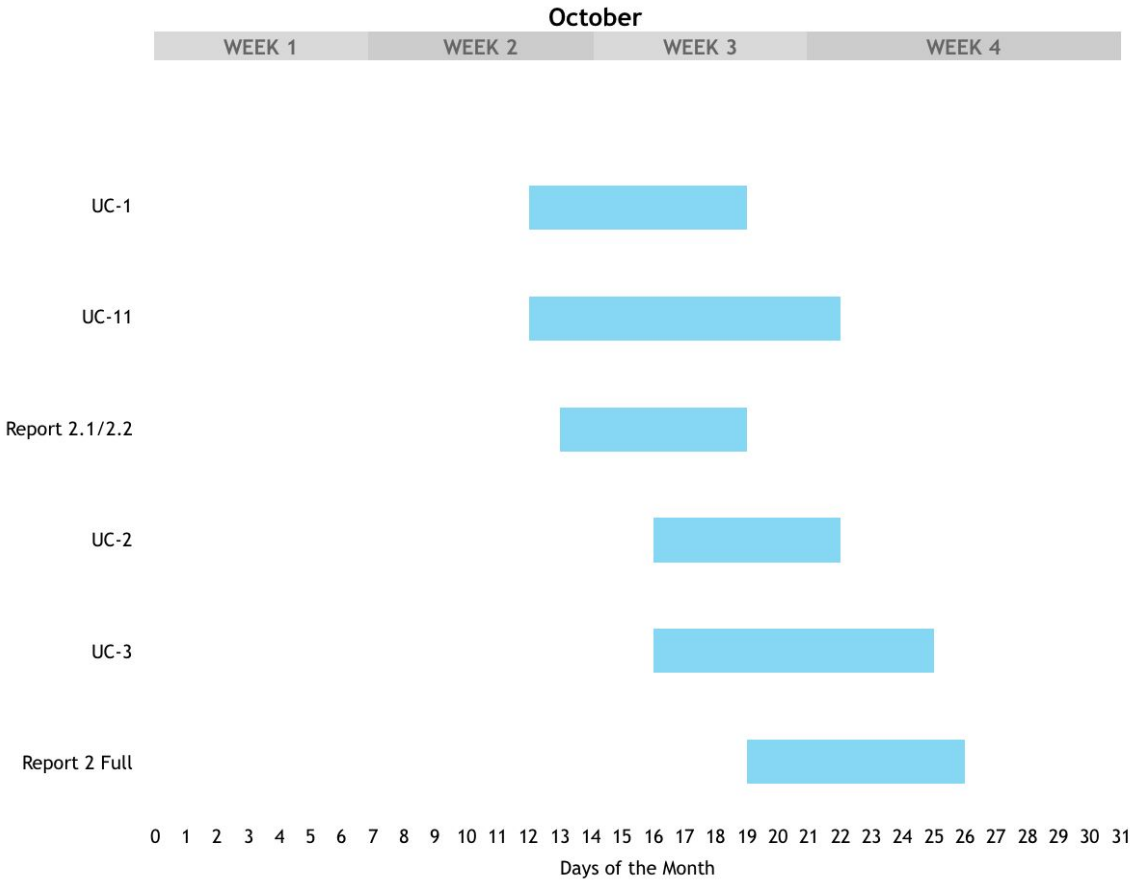

Figure 7.3.1: Gantt Chart for October

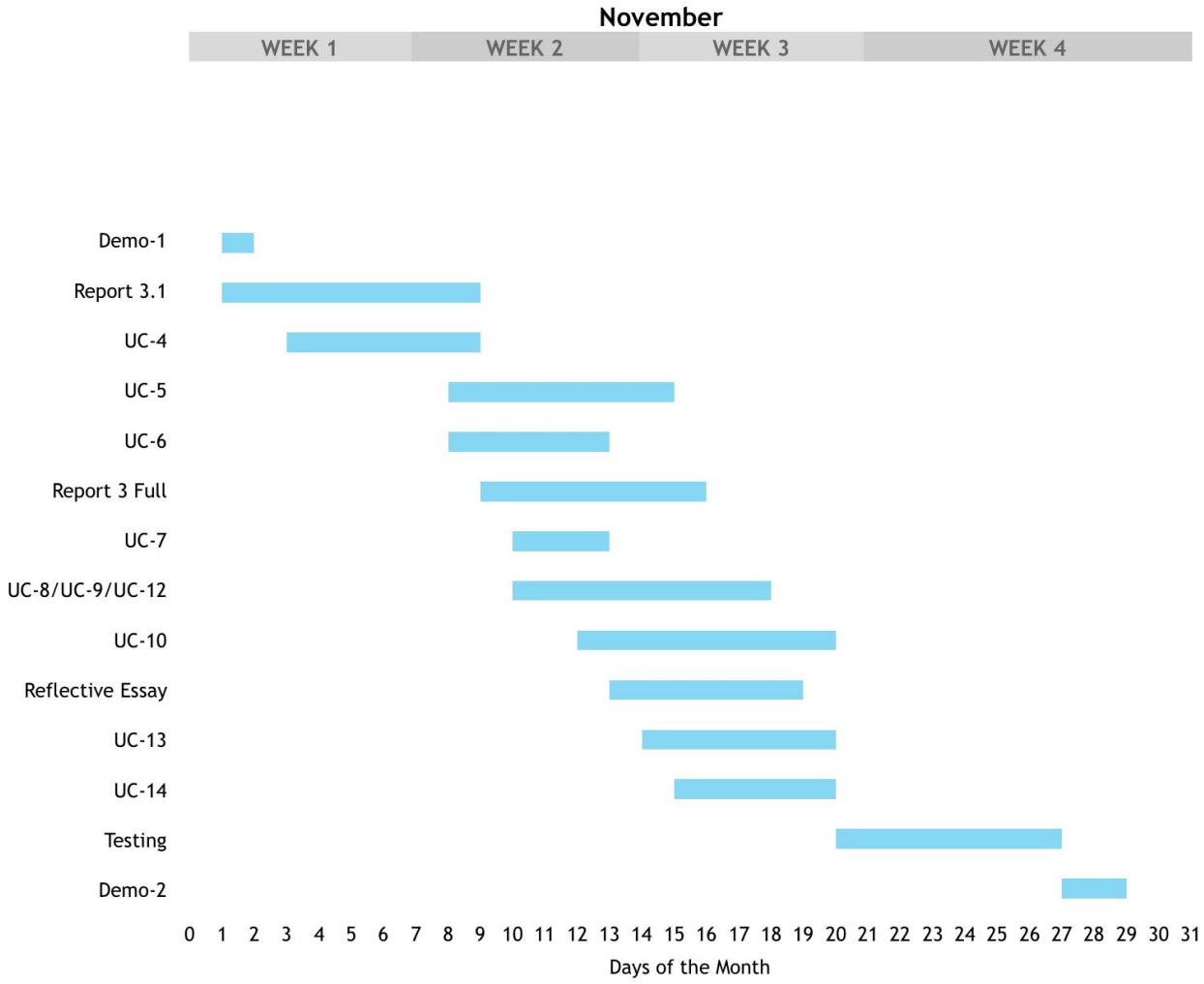

FIgure 7.3.2: Gantt Chart for November

7.4 Breakdown of Responsibilities

Name	Past	Present	Future
Jason Pulis	Created and maintains the development server and the blog. Glossary of Terms, User Interface Specification, Domain Model, Association	Development for the Login System (UC-11) and Administrator Power (UC-2), Demo Deliverables for Demo-1, Jason will be responsible for integrating	Project Management, Report 2, Development of other products

	Definitions, Product Ownership, Gantt Chart. Design of Tests	individual work.	
Christopher Whetsel	Problem Diagnosis, Enumerated Requirements, Stakeholders, Casual Description, Traceability Matrix, Fully Dressed Use Cases, Concept Definitions, System Operations Contracts, Breakdown of Responsibilities, Interaction Diagrams, Class Diagrams	Searching Member Information (UC-1), Demo Deliverables for Demo-1, Integration testing	Project Management, Report 2, Development of other products
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	Demo Deliverables and UI design for Demo-1, Creating a Text Announcement (UC-3), Integration Testing	Report 3 section and the features described in the Product Ownership

Table 7.4.1 - Breakdown of Responsibilities. .

8 References

1."Use case", Wikipedia http://en.wikipedia.org/wiki/Use_case

2."System requirements", Wikipedia http://en.wikipedia.org/wiki/System_requirements

3."User interface specification", Wikipedia
http://en.wikipedia.org/wiki/User_interface_specification

4. “System Sequence Diagram”, Wikipedia
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

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