



NEW YORK INSTITUTE OF TECHNOLOGY

CONNECTIONS APPLICATION BETWEEN HIGH SCHOOLERS AND COLLEGE MENTORS

TECHNICAL REPORT

Prepared for: M. Akhtar, Professor of CSCI-380 M02/W02
New York Institute of Technology
Department of Computer Science

Prepared by: Group SMMMT
Sean McNamee, Leader, 1270817 smcnamee@nyit.edu
Mohammad Ishtiaq, 1208754, mishtiaq@nyit.edu
Matthew Nielebock, 1243279, mnielebo@nyit.edu
Max Piff, 1098544, mpiff@nyit.edu
Timothy Cameron, 1159614 tcameron@nyit.edu

December 1, 2020

TABLE OF CONTENTS

Abstract	4
Key Terms	4
1.0 Introduction	5
1.1 Existing Systems	5
1.2 Proposed System	5
1.21 Advantage of proposed system	6
1.22 Changes to proposed system	6
1.3 Software Engineering	7
1.4 Purpose	7
1.5 Project Objective, Goals, and Scope	7
1.6 Technologies & Tools	8
1.7 Users	10
1.71 Privilege or access level	10
1.8 Motivation	10
1.9 Timeline	10
2.0 Analysis of the System	16
2.1 Activity List	16
2.1.1 Functional and Non-Functional Requirements	18
2.2 Context Diagram	19
2.3 Data Flow Diagram	21
2.4 State Diagram	21
2.5 Use-Case Diagram	22
2.6 Swimlane Diagram	23
3.0 Database Design	27
3.1 Table Schema	28
3.2 Entity Relationship Diagram	31
4.0 Functionality and Implementation	32
4.1 Features	34
4.2 Security	33
4.2.1 Privilege Levels	33
4.3 File Structure	34
4.3.1 Main Menu UI Example	35
4.3.2 Chat System Page	37
4.3.3 Registration	41
5.0 Earned Value Analysis	50

6.0 Testing	54
6.1 Unit Testing	54
6.1.1 Chat System	61
6.1.2 Security and Authentication	61
6.1.3 User Interface	61
6.1.4 Database Control	61
7.0 Conclusion	63
7.3 Team Members	65
7.3.1 Sean McNamee	66
7.3.2 Mohammad Ishtiaq	66
7.3.3 Matthew Nielebock	67
7.3.4 Max Piff	67
7.3.5 Timothy Cameron	67
8.0 References	68

Abstract

This project is meant to replace an existing system which was being utilized by our employers. By using an internet connection, the users would be able to have real time communication with college advisors and current students at specific colleges. The creation of a login system will help distinguish the hierarchy of users and provide security measures for our application. While the chat rooms can be accessed once verified, users may choose from countless college chat rooms that interest them and get started chatting away. Once complete the student or advisors may have a copy of their chat sent to them through email, allowing them to keep their conversations for future reference or for security purposes.

Key Terms

Unity Application, Channel-based, Real-Time Communications, Login Credential Authentication, Identity Verification, Data Storage.

1.0 Introduction

1.1 Existing Systems

When beginning the design and development of this application we wanted to look at products that are currently on the market that we could draw inspiration from. The applications were the best comparisons Discord, Slack, Google Hangouts, and Zoom. Discord is a popular chatting service that provides many interesting features; in Discord you can create “servers” that store role based chat rooms that support text based chatting, voice calls, and video calls. Slack has very similar features however where Discord is more casual Slack is a more professional platform that is used by many companies. Slack also utilizes the role locked chat rooms. Google Hangouts is a popular video chatting platform from Google. We drew inspiration from the polished look that Google put onto their product. We want our product to look as good as it functions so as such we took cues from popular industry standard applications. Zoom has risen to popularity in this new online era due to the ongoing global pandemic; so as such we had almost no choice but to consider this application when designing our system. Zoom is another application that has a heavy focus on video chatting and holding meetings. A unique feature that Zoom introduced was breakout rooms. Break out rooms allow the owner of the call to set up several other rooms that would send users in the meeting to, for the purpose of having smaller discussions. What makes break out rooms so unique is that users never have to leave the main call they simply go to a sub-meeting. Once the owner decides the break out rooms have run their course they can close the rooms and all users will return to the main call. Once again this product has a very professional and simplistic feel when using it, so Zoom was a great piece of software to use in part of our framework for the design of our application.

1.2 Proposed System

After taking into consideration the features that Discord, Google Hangouts, and Zoom offer to connect users to each other, we decided that we wanted to narrow the scope of our project significantly. Video and voice calling, file sharing, and breakout rooms are unnecessary features for our connections application. The project two abstract specifies that our system must help connect high schoolers with college mentors and alumni. Therefore, we decided that the communication between these users could be limited to chatting.

Typical use of our desktop application would allow high school students to log into the system, enter college chat rooms, and chat with college mentors. To achieve this, we decided to implement standard messaging and channels. Unlike many existing systems, our system will have a verification step that will ensure real high school students are engaging with real college mentors. On top of this user verification, our system will have channel verification, allowing only colleges with verified faculty to have a college channel. Therefore, regular users can focus on interacting with others rather than looking for legitimate channels. Due to time constraints, our project’s chatting services will remain local to a user’s computer.

For high school students and college mentors to gain access to the system, they must register for an account and be verified by their school’s faculty. For a school’s faculty to gain

access to the system, they must do the same, and be verified by the system's developer. On registration, a verification code along with the user's information is sent in an email to the verifier. The verifier can then log into the system, go to the verification page, and enter in the user's name and the verification code they received for that user. Only then will that user have access to more than just the menu, registration, and login pages.

In terms of use, High schoolers have simple chat privilege in all channels, while college mentors have moderator privilege and access for their college's channel only. Before exiting a channel, users are asked if they would like a transcript of the messages from their session sent to their email. If declined, the messages are no longer accessible to that user.

1.21 Advantage of proposed system

There is not an abundance of features that come along with this proposed application, however, the ones that are included boast success. One of these features is the verification process which is crucial to the application so our users are not speaking with random people who are users with heinous intentions. We took pride in this feature because of its importance and how necessary it is to something as prestigious as college help and education.

Another feature that we thought carried enough weight to include was our chat history system which sends past chats to a user through email if they would like to have a record of them. This is good in case a user asked for valuable information but managed to forget or maybe misunderstand the advisor and would like to look back at their messages. Additionally, in the situation that they would like to talk to the same advisor they could find out easier since this advisor would most likely introduce themselves within the messages.

Although something that might be overlooked, our application is simple and user-friendly which makes it easier for users to traverse the app while getting the information that they strive to acquire. This supports them going into their college career and gives them invaluable conversations and possibly even connections with these advisors and mentors.

1.22 Changes to proposed system

When looking back to our proposed system we found that we had set the goal posts at a very reachable distance. Before we started working on the technical portion of the project we sat down as a group to discuss what we could realistically accomplish within the constraints of both time and the different experience levels of all members. As a result of our thorough planning of the project we were able to accomplish all but one of the original goals we outlined in our proposed system. The only aspect we did not incorporate from our proposed system was achieving functional chatting between two people. In its place, we allow the system to automatically respond to specific phrases provided by the user.

1.3 Software Engineering

The software model we decided to use is the waterfall model. Since we know exactly what we are trying to make and it most likely will not change, it is our best choice. To successfully create this connections application we split the work between everyone in the group and gave everyone fields they are inexperienced with. This immediately eliminates RAD, Iterative, and Spiral. The Iterative model would not work because of the time constraints. A prototype would not be necessary because we already have an idea of our UI. Spiral would be unsuccessful because our team does not have enough developers to split work between two fully-functional models. Finally, the Concurrent model does not apply here because it is not a feature intensive project and we are creating the project from scratch. Based on these deductions, we decided waterfall was the most beneficial to work with for our application.

1.4 Purpose

The team was tasked to build a new and refined version of a currently operational connecting system. This system would allow potential college applications to connect with college advisors, who then would help answer various questions. The system would allow the users to sign up, log in, choose a chat room based on the college of choice, and start chatting with an advisor there. These were the key aspects we focused on for this connection system, but we had created a system to save the messages that were being sent throughout the time that the user was in the chat room and email them after they had left the respective chat room. This allowed potential college applicants and college advisors to refer back to their chats for the time frame they were active for safety, convenience, and administration work.

1.5 Project Objective, Goals, and Scope

The goal for this project is to help potential college students in their search for the right college or university. In the current day and age, many colleges are not holding on-campus tours for incoming or prospective students. Our system would allow these students the ability to speak to college advisors that are from their respective universities safely and securely. In addition, having other students there in the chat room while you are asking questions is beneficial to everyone because everyone can help each other with their proposed questions. During a typical college open house, you would meet other students, but since you can not do that due to the pandemic, we can allow these students to meet in a more comfortable space. This will help make friendships and allow them to meet peers who are interested in the same things they are interested in. The scope for this project is to have a working login system for users, advisors, and admins. Following that we will have a working chat system in which an 'Admin' may respond to specific questions or phrases given by the user. Lastly, we would be able to send the chat history to the user if they feel as though they require it.

1.6 Technologies & Tools

Unity:

We decided to use Unity for most of the build of this project. Unity allowed everyone to piece together all the parts of the project one by one. A wide array of free extensions allows us to easily integrate SQL servers, GitHub, and programming extensions into one project. Unity was a program that 80% of the group did not have any history with, so this was a very big learning experience. Not only did we learn to build this application, but we also learned how to connect many different technologies to make a nice user experience.

Building the UI in unity is highly flexible because it is a drag and drop system. Since this is a prototype of the final project, we went with a simple UI system because we want to keep the focus on the functionality of the chat system. The addition of auto-adjust allows the background to scale up or down based on how the user scales the application window.

C#:

The main reason for choosing C# as the primary programming language is because it worked well with Unity. Some of the group members had no prior knowledge of C# but have worked with C and Java. This allowed everyone to quickly learn the syntax of C# and allowed us to get working on our project very quickly. C# is also heavily documented on the unity help center, and the unity forums. This allowed us to find answers to questions about programming and other extensions fast and easy

GitHub Desktop:

To allow everyone access to the main program files 24/7, Sean set up a GitHub repository. After downloading the GitHub desktop application, we all were able to connect to the main folder where the scripts and programs were stored. This allowed the team to work on their specific parts independently without having to wait for everyone to be online.

Discord:

The team's main communication system was Discord. Discord allowed us to make many different rooms for different purposes and goals. Allowing everyone to speak in the respected room for a specific task, allowed there to be less confusion if there was only one chatroom. For example, there is a chat room called "Database", which only allows chat for database related questions, code, or advancements.

The team would also take time out of their week to meet in a group call. Discord made this extremely easy on us as well since we could make a voice chat room. It is a lot easier to use

than many other systems in the market and is more responsive for us. We were able to share our screens and work with our teammates quickly, easily, and effectively.

Google Drive:

The purpose of using google drive is to share our work quickly and efficiently. Google drive is already well known for file storage and sharing, so this was a big part of our project day today. We organized everything through folders labeled with the specific tasks and filled those folders with the appropriate information. Google drive is also free for everyone with a Gmail account, and we all get one for free with our designated NYIT email account(s). We were able to work on documents at the same time, which helped make group discussions more productive.

MySQL:

For our database, we used MySQL. Since Tim is currently in a database class, we felt that it would be best for him to start with a standard database system that most of the team was familiar with. MySQL allowed us to the Command Line Client and Workbench. The workbench's GUI system made it easy to create new tables with appropriate attribute data types and manipulate tables. One difficulty with the workbench was foreign keys, thus the command-line client was used to directly interface with the database. Once the database was integrated with the GUI, the command line client was used to quickly retrieve or update records in the tables.

SMTP (Simple Mail Transfer Protocol):

To send verification codes and message logs, an automatic email system was needed. SMTP is a useful protocol that's implemented in many languages. After creating a Gmail account and disabling the account's security features, the SMTP made it very easy to send emails to any user. Preset email bodies were created so that crucial information could be inserted. One example of this is that the bottom of all emails will end with "Regards, College Connections Team". The Gmail that sends the emails is csci380.smmmt@gmail.com. The limit of emails that could be sent daily remotely by a Gmail account was one hundred, and we felt that this was more than sufficient for our application's use (Google, 2020).

When2Meet:

This online tool is like other sharing programs but is meant to make scheduling for meetings easier. A shared link would be provided by our team leader and everyone apart of the team would then highlight the times in the day they can meet for the project. After everyone has finished inputting their times, you can hover around on the data table and the output will display which team members are free at that specific time.

1.7 Users

The users of our product are high school students, college moderators, and advisors. Each of these users has a different set of rules associated with the role. The bulk of users using our system is intended to be high school students. They will be able to traverse between chatrooms to speak with the various college moderators waiting to provide college-specific information. This information will be close to if not completely up-to-date because these college moderators will be alumni or current students. College moderators can as well moderate chat. Advisors supersede the predefined users, granting them access to the system through verification. Developers, a specialization of advisors who belong to no school, can verify advisors.

1.71 Privilege or access level

In terms of access within the system, users have the following capabilities: register, login, verify users, join chat room, chat, moderate chat, exit chatroom, receive chat transcript, and logout. Unregistered users can only register. Once registered, users have the type of high school student, college moderator, or advisor. Once they are verified, they will all be able to login. High school students can browse the chatrooms they are assigned to and join any of them. Within the chatroom, they can send and receive chats. When exiting a room, they will be given the option to receive a transcript of their message history. After fully exiting, that message history is cleared. College moderators can perform all the aforementioned tasks, but can also moderate chat within a room. Advisors have the capabilities of all users previously stated, however, they gain the ability to verify users from their college that they inputted during registration. Since we need to clarify that the advisors are legit, we allow developers to verify them. Another plus for being a developer is that they can access all possible chatrooms, not just the ones they are assigned to. These are the only differences between the Advisor and the Developer user type. Overall, these user privileges limit user access accordingly and ensure identity when using the system.

1.8 Motivation

We were motivated to do this challenge simply because it seemed like the best option for our group's skill set. The main goal of the project was for people to learn new things, and this allowed people to learn topics that they weren't strong in, all while being able to ask questions to others who do know the topics. One example of this is that only one person in our group knew how to use Unity, Mohammad. Multiple people were trying to learn the program and if they were stuck and couldn't find an answer on Google they would ask Mohammad for help. The main reason we used Unity was because four out the five group members knew how to make html pages which was the original idea for the project, but we wanted to challenge ourselves.

1.9 Timeline

Before getting far into the project, we knew that partitioning our workload would be crucial to creating a successful system. Therefore, we set specific dates that individual components for a deliverable would be due on top of the actual due date. As well, we set internal

dates for the system's component updates. These, in addition to important dates for our team's success, have been included in our timeline.

Our timeline starts on October 5th when the original team was formed. As shown in figure 1.9.1, we worked together on the second assignment of the semester, partitioning our tasks of research and writing evenly among the group. Hurried efforts in the days leading up to the project deadline provided foresight to create internal due dates. By moving these internal dates further from class-designated deadlines, we provided ourselves with more time to cross check each other and ask questions in class.

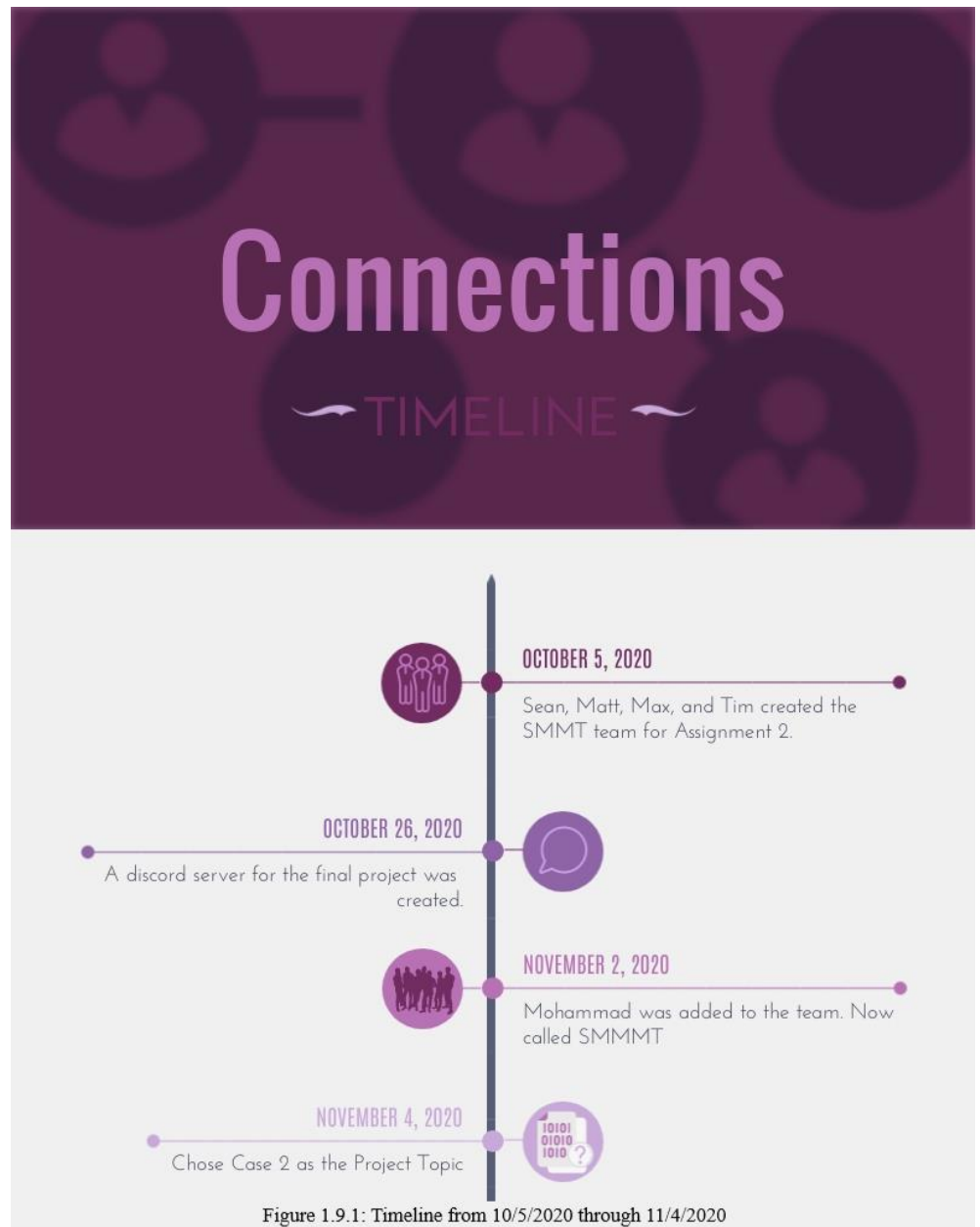
As the term project approached we created a discord server, as we all foresaw the deficiencies a singular discord chat would amass. The discord server provided multiple categories and channels, which we utilized for overall concepts and deliverables. Our discord server was initially separated into the following categories: Information, General Communication, and Deliverables. As the project progressed, we added categories for each of

the implementation components of our project. The discord server allowed the team to stay focussed and task oriented throughout the project.

On November 2nd, Mohammad joined, and the team was finalized to SMMMT. His entrance created a shift of our team's focus towards Unity, as he expressed his expertise in it and boasted its capabilities with UI creation. After choosing case two for our project topic on the 4th, the team quickly began communications to identify possible solutions. Our discussions included HTML-based web pages, a Java-based GUI, and a Unity-based GUI.

As Figure 1.9.2 notes, we created an official location for our project's files on November 5th, and came to a conclusion on our technology just two days later. We decided to utilize Unity for our GUI with Java as backend logic. Unfortunately, on November 9th, we learned that authentication had to become an integral part of our system. In response, the most focus was put towards that, and we split our project into four parts: Security, Database, UI, and Chat.

As our workload increased, we created a location for diagram files, then assigned responsibilities for Deliverable 2. On the same day, November 12th, we submitted Deliverable 1. Ahead of pace with Deliverable 2, we moved our focus back towards the project, splitting our team so that each member would be working on a component they were unfamiliar



with. We decided each member who was familiar with a component could act as a secondary, providing assistance to the primary member whenever needed.

Work on the project began quickly, with a GUI prototype being created and uploaded to the Google drive on November 14th. Testing with that project on the 17th then brought forward the problem of Java's compatibility with Unity. To retain the already completed work in Unity,

the rest of the project would have to switch to C#.

Therefore, the setup of C# and solutions utilizing it started being explored.

Meanwhile, our group wrapped up our Deliverable 2 on November 18th as a result of a misunderstanding of its required deadline. With the extra time, we sent an email to gain further understanding on one of its requirements. We finalized and submitted Deliverable 2 on November 23rd, as seen on Table 1.9.3.

Submitting Deliverable 2 allowed the team to move focus towards the project yet again. Each Component was worked on, and an internal due date of November 30th was set to have all components as prototypes and in the Google Drive. The next day, to further centralize the project, all the prototyped

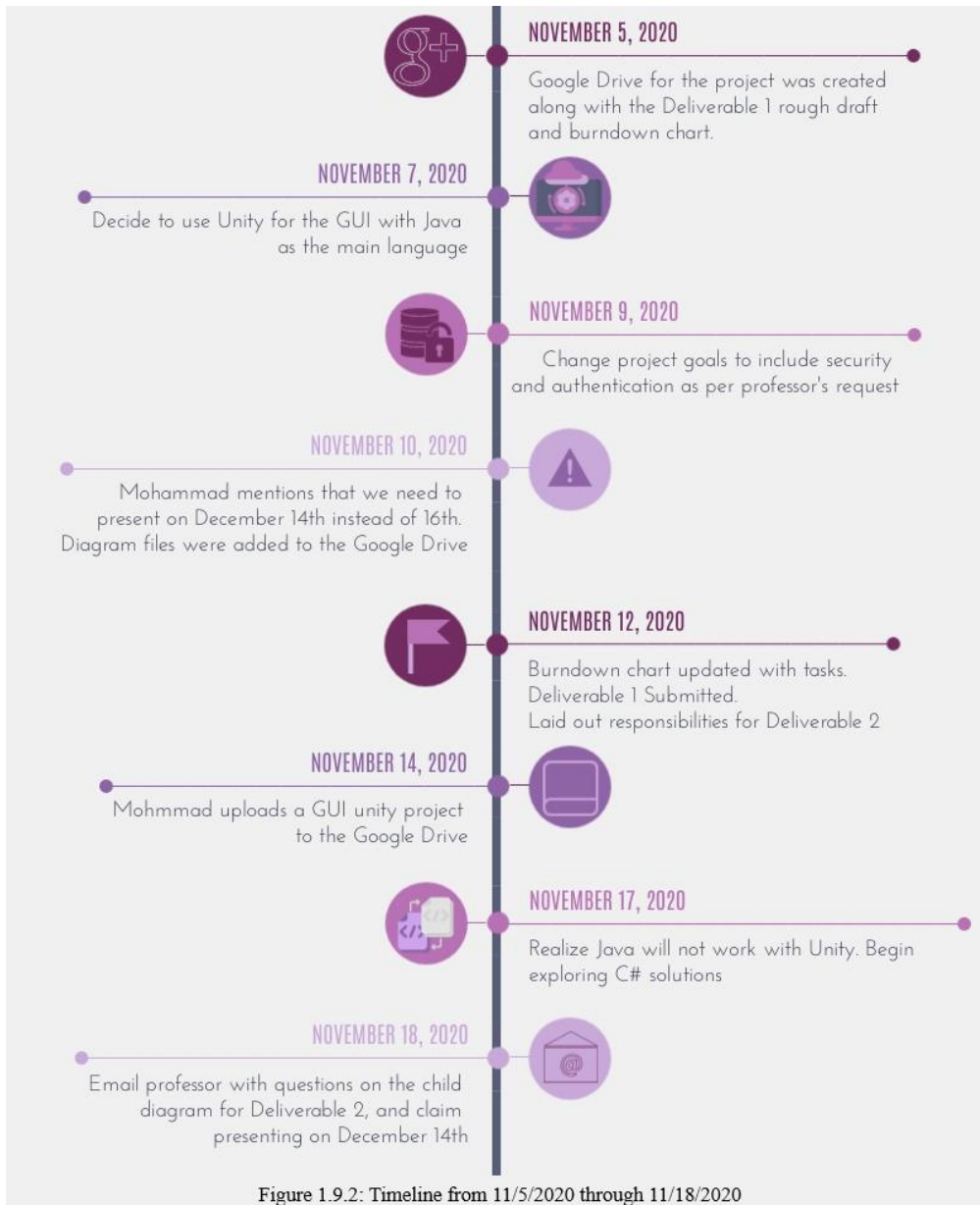


Figure 1.9.2: Timeline from 11/5/2020 through 11/18/2020

components were merged into one project and posted to GitHub. Since each component was still a prototype, functionality was limited. However, setting this up paved the way for uniform access to the project.

On December 2nd, shown in Figure 1.9.3, the responsibilities for the last stretch of our project were laid out: Deliverables 3a and 3b. We created a frame for the Technical Report following the provided format, and used that to create a presentation. The presentation was then modified to fit the guidelines explained in class. Once each member had their personal responsibilities, they could work at their own pace, updating the group of any problems via Disord.

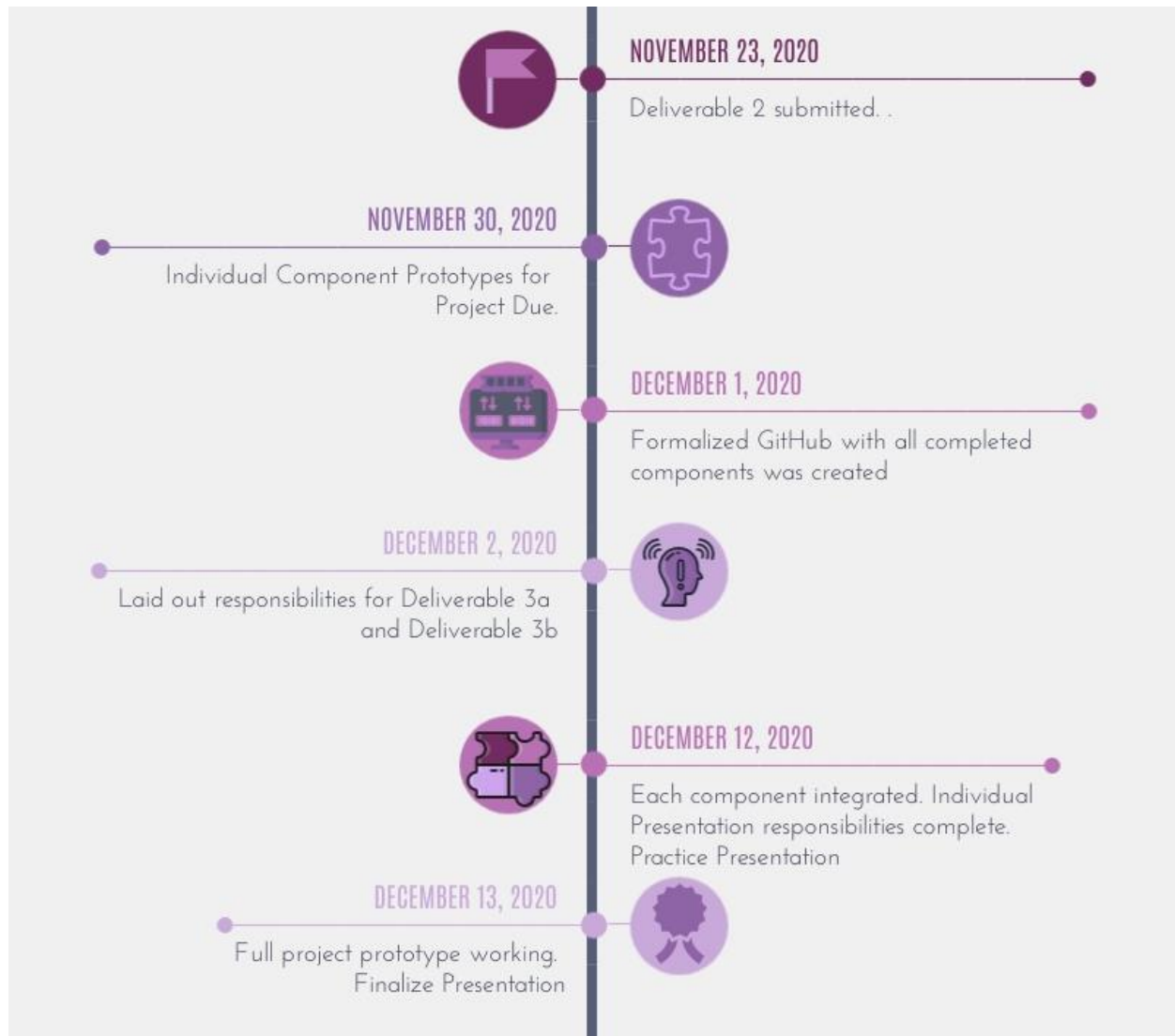


Figure 1.9.3: Timeline from 11/23/2020 through 12/13/2020

The internal deadline of December 12th came, to which each component was successfully connected to the others, the individual responsibilities for the presentation were complete, and the presentation was practiced. By the next day, the presentation was finalized and the full project was working.

That Monday, December 14th, as seen in figure 1.9.4, the project was presented, edited as per recommendations, and submitted. The following day, individual components for the

Technical Report were completed. On December 16th, Deliverable 3a and 3b were completely submitted, finishing SMMMT's work on the Connections Application.

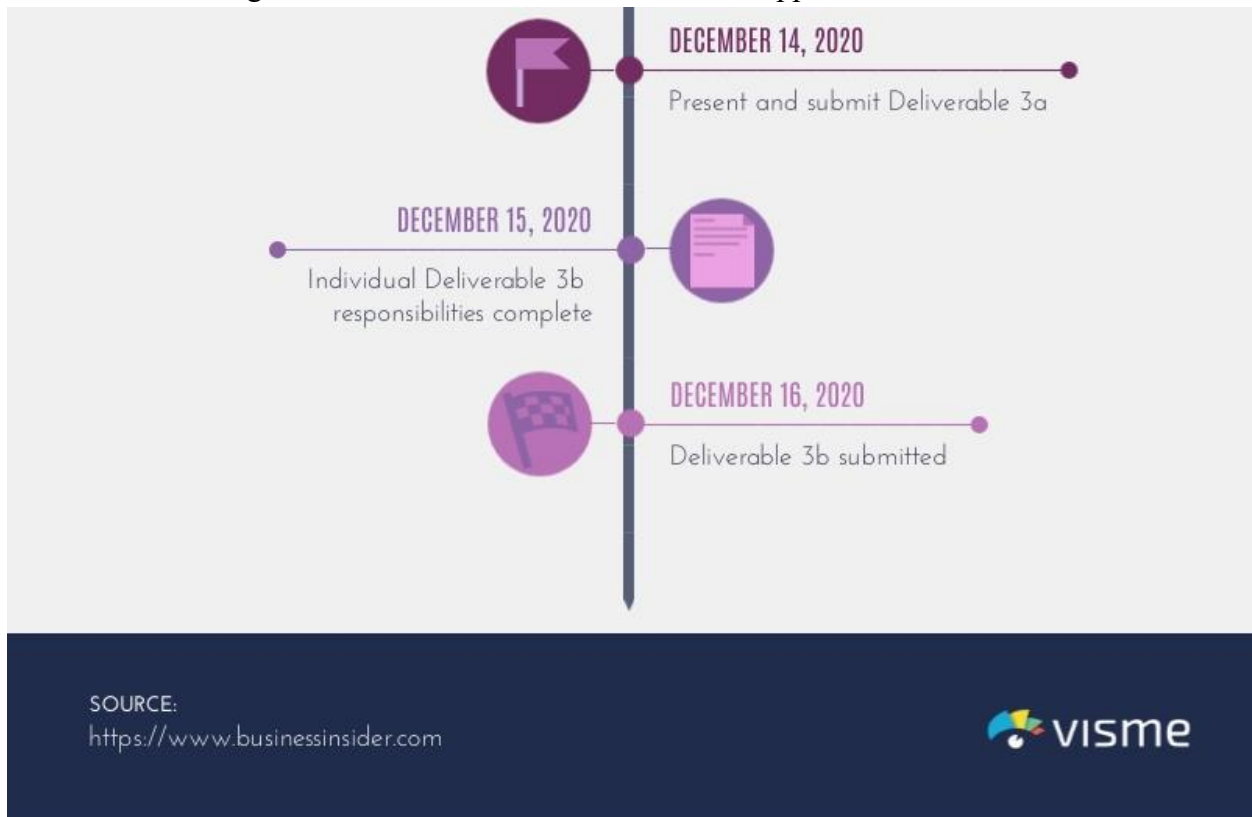


Figure 1.9.4: Timeline from 12/14/2020 through 12/16/2020

2.0 Analysis of the System

2.1 Activity List

In terms of our project's progress, the first prototyped component was the GUI. Created using Notepad++ files in C#, it generalized our team's idea of varying chatrooms for the system. Soon thereafter, we discussed further implementations, and decided to use Unity as our user interface. The other components of our project were the database access, chat system, and security. While these components separated tasks for our group members, these components could still be broken down, as shown in Figure 2.1.1

This activity list shows the dependencies of various tasks, and used spreadsheet calculations to be dynamic. One can see that for the database component to be complete, schema, a database account, the working queries, a connection through C#, and the queries in C# had to be completed. Carefully reading down the list of task task names, one can see their organization. Database dependencies, Authentication dependencies, chatting dependencies, then user interface dependencies take up tasks 1 through 28. If one wanted to go more in depth on tasks, they could break down tasks such as Queries into each individual query, or tasks such as a single page into the input fields needed. One note to make on this list is that while the receiving and broadcast of messages was meant to be through a socket, it was only implemented locally.

Continuing with the task names, the overall completed components were followed by the integration of those components with each other, and the completed prototype. One downside to specifying responsibilities based on components was that once integration of those parts began, uniform work ceased, and pressure was placed on few members. After the prototype was completed, extreme testing of the overall functionality began. The reason unit testing is not included is because the successful implementation of a component assumes that unit testing was performed and passed. After testing was completed, fixes were made, and the system was deemed finalized.

Task Number	Task Name	Task Description	Task Length	Start Time	Completion Time	Task Dependencies
-1	Start	Calculating	0	0	0	
1	Database schema	Database	3	0	3	-1
2	External DB account	Database	1	0	1	-1
3	Working Queries	Database	2	3	5	1
4	Connection via C#	Database	2	5	7	1, 2, 3
5	Queries via C#	Database	2	7	9	4
6	Project Email Account	Security	0.5	0	0.5	-1
7	Email Sender	Security	2	0.5	2.5	6
8	User class	Security	2	0	2	-1
9	Account Registration	Security	2	2.5	4.5	7

10	Verification Code Creation	Security	1	0	1	-1
11	Account Verification	Security	1	2.5	3.5	7
12	Login Authentication	Security	2	2.5	4.5	7
13	Password Encryptor	Security	2	0	2	-1
14	Message Class	Security	0.25	0	0.25	-1
15	Add message to room chat log	Chat	0.25	0.25	0.5	14
16	Delete message from room chat log	Chat	0.25	0.5	0.75	15
17	Add message for user chat log	Chat	0.25	2	2.25	8
18	Delete message from user chat log	Chat	0.25	2	2.25	8
19	Receive message from user	Chat	2	2	4	8
20	Broadcast Message	Chat	2	0.25	2.25	14
21	Response to user messages	Chat	1	0.25	1.25	14
22	Main Page	GUI	0.5	0	0.5	-1
23	Login Page	GUI	0.5	0	0.5	-1
24	Register Page	GUI	1	0	1	-1
25	Verify Page	GUI	0.5	0	0.5	-1
26	Browse Room Page	GUI	0.5	0	0.5	-1
27	Chatroom Page	GUI	1	0	1	-1
28	Connect pages together	GUI	2	1	3	22, 23, 24, 25, 26, 27
29	UI Prototype	Prototypes	1	3	4	22, 23, 24, 25, 26, 27, 28
30	Chat Prototype	Prototypes	1	4	5	14, 15, 16, 17, 18, 19, 20, 21
31	Security Prototype	Prototypes	1	4.5	5.5	6, 7, 8, 9, 10, 11, 12, 13
32	Database Prototype	Prototypes	1	9	10	1, 2, 3, 4, 5
33	Register adds to DB and sends email	Integration	2	10	12	31, 32
34	Verification updates user account	Integration	2	10	12	31, 32

35	Login only after verified	Integration	1	12	13	34, 33
36	Browse rooms only after login	Integration	1	13	14	35
37	View all user chats for chatroom	Integration	2	5	7	29, 30
38	User messages sent via email on exit	Integration	1	7	8	37, 31
39	DB integrated with Security	Integration	4	13	17	33, 34, 35
40	Security integrated with UI	Integration	3	14	17	35, 36, 38
41	Security integrated with Chat	Integration	1	7	8	37
42	Chat integrated with UI	Integration	2	8	10	37, 38
43	Finalize Prototype	Integration	3	17	20	39, 40, 41, 42
44	Test Chat System	Testing	2	20	22	43
45	Test SQL Injections	Testing	3	20	23	43
46	Test UI Usability	Testing	3	20	23	43
47	Functional Testing	Testing	4	20	24	43
48	Finalize Testing	Testing	1	24	25	43, 44, 45, 46, 47
49	Fix SQL Injections	Improvement	2	25	27	48
50	Increase UI Usability	Improvement	2	25	27	48
51	Finalize System	Finalize	1	27	28	48, 49, 50
Figure 2.1.1: Activity List						

2.1.1 Functional and Non-Functional Requirements

A functional requirement is what the system does, how it performs and overall covers the technical aspects of a system. A non-functional requirement on the other hand defines how a system will accomplish what it has set out to do; non-functional requirements focus on the usability of a system. The functional requirements for our system would be the database, the chat system, and the security and authentication. The reason these aspects have been identified as functional requirements is that without any of these features the application would simply not function at all. The database is a necessary technical component due to the fact that it stores all

of the user information; and that information is what allows the users to properly access the channels and roles that they have. The chat system, which is the focal point of the entire application, is obviously a functional requirement because without it working our application might as well just not exist. The chat system allows the high schoolers to connect with the college students and advisors which is the goal of what we set out to do when designing and producing this application. The last functional requirement that has been identified is the security and authentication aspect of the application. Without our security the application would become compromised. We cannot in good conscience produce an application that puts our own users at risk. The authentication that we have implemented is particularly important due to the emphasis we place on the verification on each individual that would be accessing our platform. This aspect is a vital aspect to how our application functions at a technical level. Aside from the developers each user on our platform has been verified by either a developer or an advisor; this is a functional requirement due to how our application was proposed. Our non-functional requirements would include the user interface (UI). When discussing the UI we will have to examine a few different aspects of it. These aspects would include the overall look of the application, and the layout of the menus. The way that the application visually looks does not interfere with how the application functions at a technical level; however, if the application does not look nice it will impede how easy it is for a user to navigate and use our software. The layout of the menus is important when thinking about non-functional requirements due to the fact that if no one is able to navigate the application, they cannot use said application and thus the functional requirements don't even matter.

2.2 Context Diagram

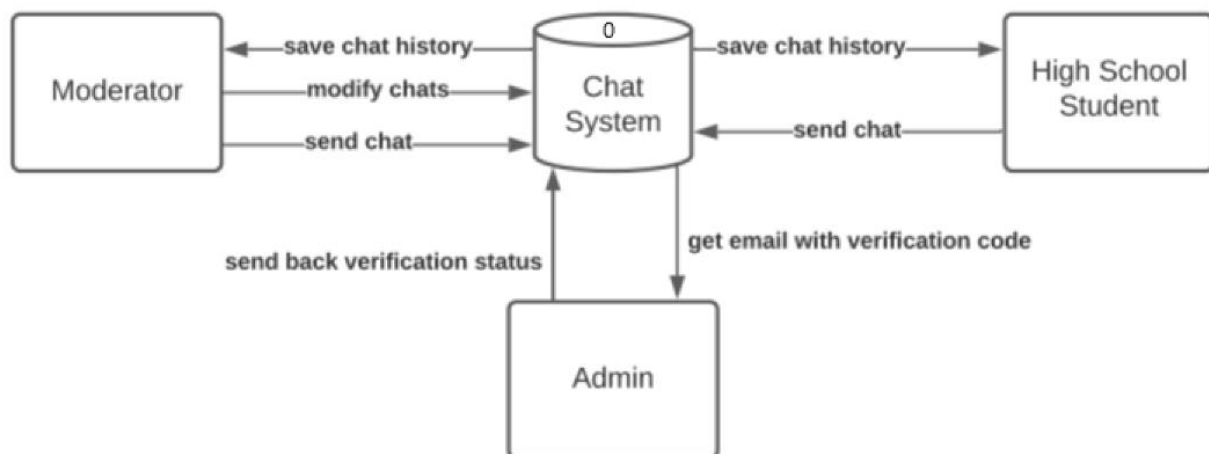


Figure 2.2.1: Chat System Context Diagram

The context diagram in Figure 2.2.1 shows the functionalities of the main entities of the system - the users. High school students are able to send and receive back chat history and

moderators have the same capabilities but can also moderate chat. Admins are able to get emailed the verification codes and then verify the students if they are eligible.

2.3 Data Flow Diagram

Data Flow Diagram 1

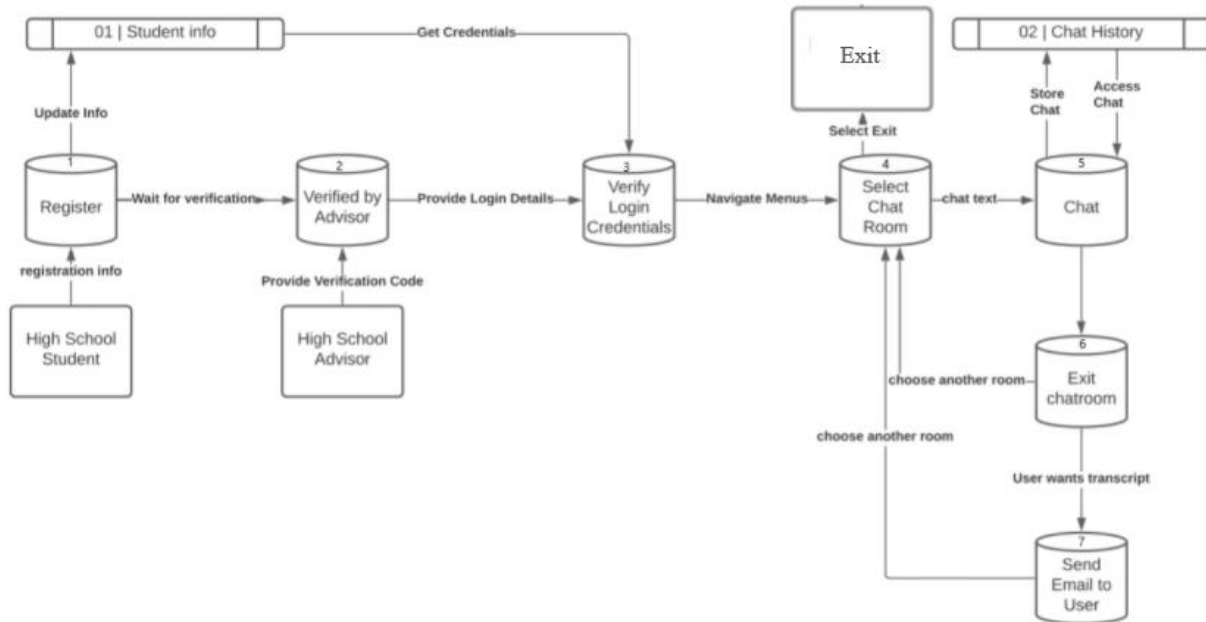


Figure 2.3.1: High schooler DFD

In our first data flow diagram seen in Figure 2.3.1, we focus on the high school students and high school advisors. A high school student can register, which then updates the first data store, student info. After this the student must wait to be verified by the advisor. Once the student is verified, they must provide login details to get access to the information. The provided login information is checked from the student info data store and once it is proven to be correct, the student will have full access to join and use chat rooms. Once a chat room is selected they are linked to an advisor from the school that they chose to speak with. These chats are stored in the chat history data store in order to be used for a chat transcript. When the users decide to exit the chat room, the user gets a prompt asking whether or not they want a transcript of the chats. If they choose that they want a transcript then it is sent to the email that they provided with their account. Regardless of the answer to the prompt the user is sent back to the select chat room screen, their chat history is erased, and they can either exit to the main menu or choose another chat room.

Data Flow Diagram 2

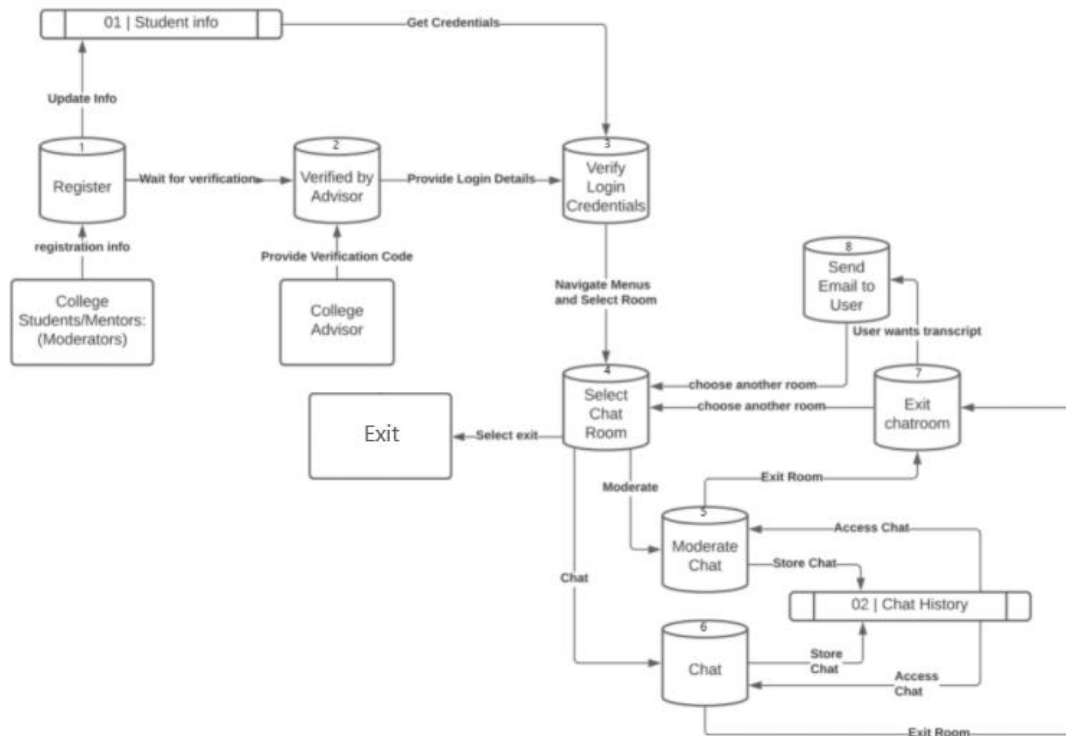


Figure 2.3.2: College Advisors/Moderators DFD

The second data flow diagram is identical to the first data flow diagram for the login processes, however, the chat system adds a process into the mix. The fifth process, seen in Figure 2.3.2, is the only change to the diagram, adding the moderate chat process. This process allows for moderators to join chats to moderate them and they access the second data store chat history once again for chat transcript purposes.

2.4 State Diagram

To better depict the functionality of the components of our project, we created a state diagram as shown in Figure 2.4.1. It structured the various variables and functions that each class would have, along with their visibility. As one can see, the Authenticator deals with the registration and login process. This includes the initial registration, generating and sending a verification code sent in an email, checking if a verification code is valid for a user, and logging in the user. As well, the Authenticator helps to ensure inputs and emails are valid. This Authenticator utilized the Queries class. The Queries class created a connection with the database, ran statements and queries, then closed. The various functions allowed the Authenticator to quickly make calls and abstract its logic away from MySQL syntax. The Login function of the Authenticator returns a User. This user contains all the information within the user table of the database, with an exception to the password. One should note that the firstName and lastName attributes in the database are combined to create the User class' username variable. One of the User's variables is a messageList. This is a record of all the user's

chat's and those who chatted with them in a room. A simple call to the SendMessageHistory utilizes the message history and email variable to send the transcript to the person's account. While in a chatroom, the Chat class is utilized. This updates the User interface with the provided messages and saves them to the user. Internally, it uses the Message class to pass the information quickly.

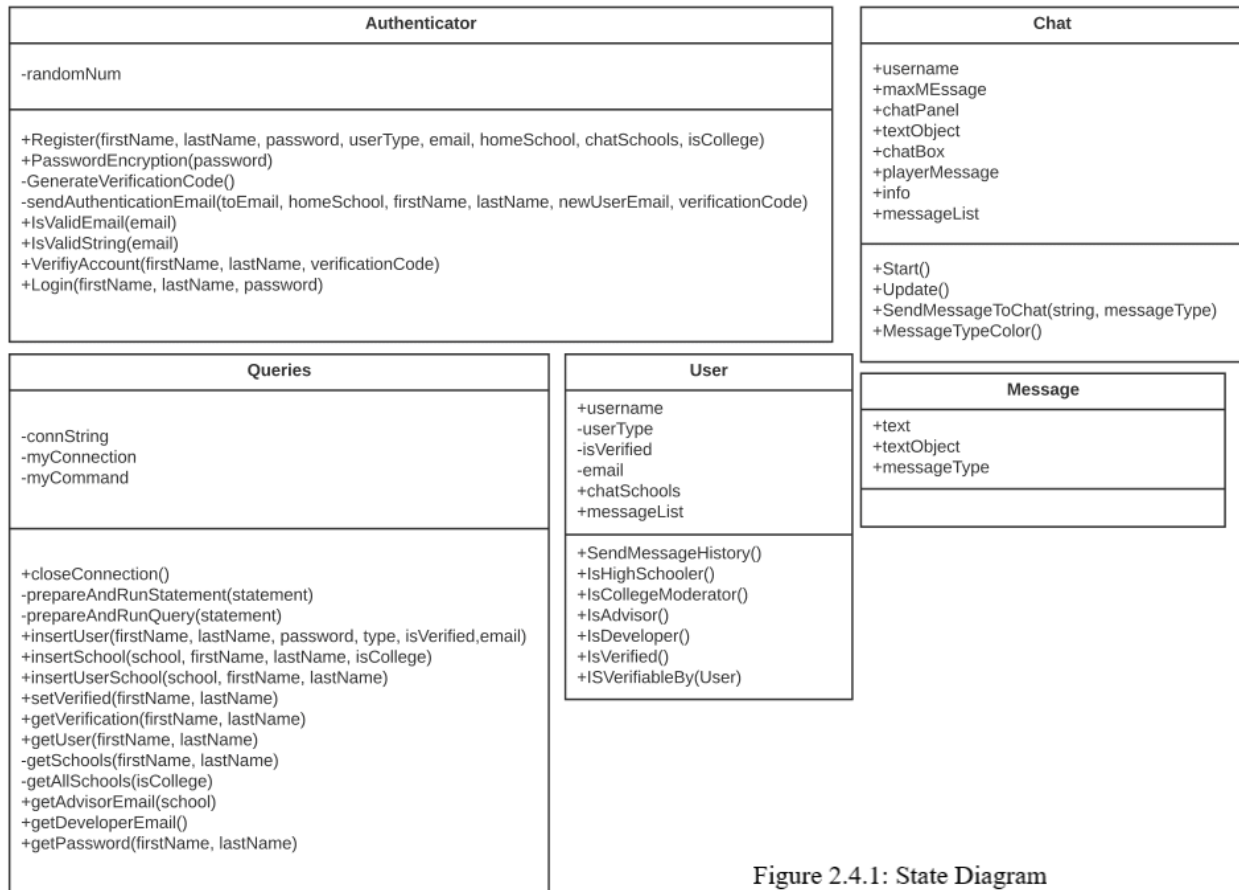


Figure 2.4.1: State Diagram

2.5 Use-Case Diagram

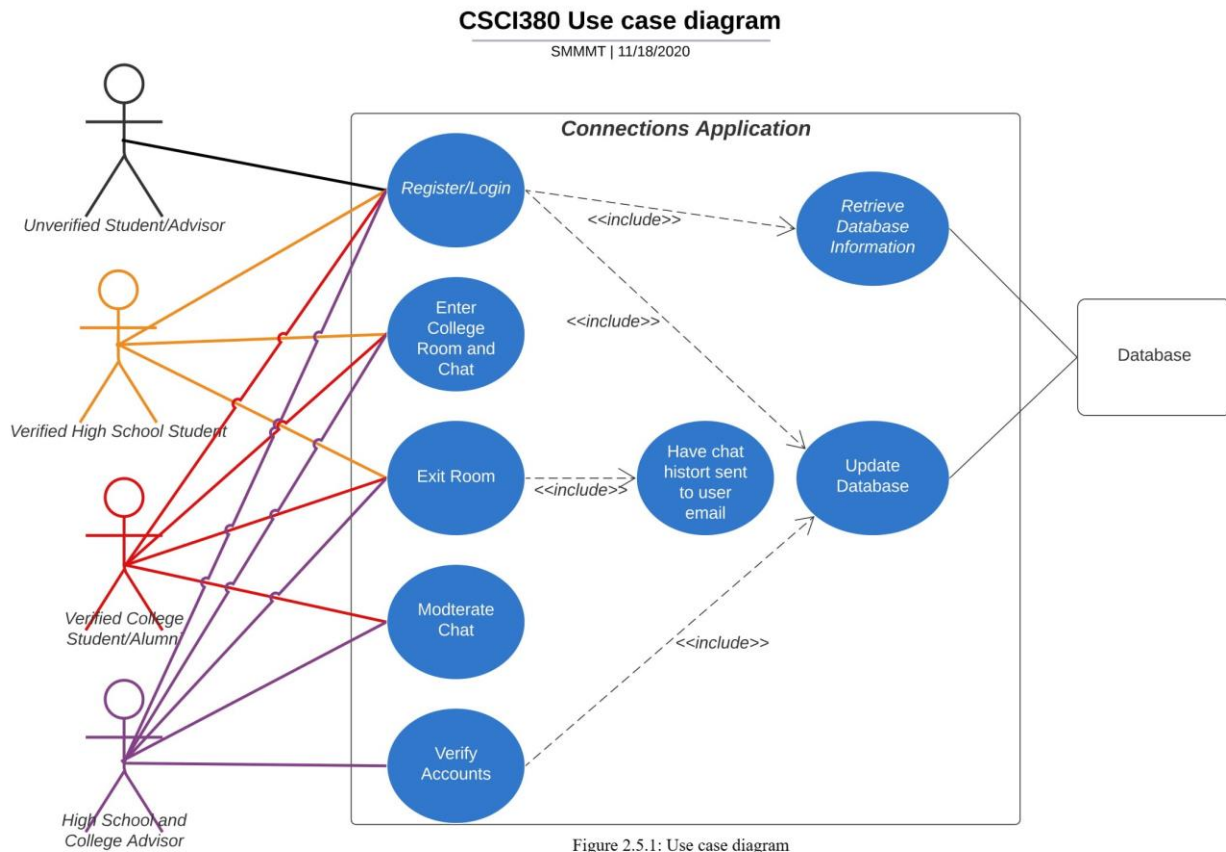
The Use-Case Diagram, as shown in Figure 2.5.1, shows what users types can do what actions. The users shown are unverified, verified high schooler, verified college moderator, and advisor.

Unverified users are the most basic and only have one option, and that is to register themselves in the system. Once they register their account it will store that information in the database, and then they will be assigned to the correct user type which will update the database.

The next user is the Verified High School Student, these users are able to login to the system, the main thing that this user can do is enter the chat rooms and message other highschool/college students, alumni and or advisors. The final option for this user is the ability to exit the room and request a transcript, and then they can log out.

The third user is the Verified College Student/Alumni, these users are able to login, enter chat rooms, chat in the rooms, moderate the chat rooms, and request a transcript when they exit the room.

The highest type of user is the High School/College Advisors, these users are able to do the most. The advisors are able to login, enter and chat in the chatrooms, moderate the chat rooms, verify accounts, and request chat transcripts. Since these users can verify accounts they are also able to update the database. A specialized type of advisor is the “Developer”. While not shown because their abstracted permissions are identical, at a closer level, the developer is the one who verified other advisors



2.6 Swimlane Diagram

The swim lane diagram shows the flow of a user's actions in the system. For our swimlane seen in figure 2.6.1, we separated it into two lanes; the user, and system. Once a user is logged in, the focus shifts to the system, which carries out tasks based on input.

If the system does not see that the user's credentials are already in the database it will require that user to register and have their information verified. Once the user/information is verified it will be added to the database. This will allow the user to skip this step the next time they log into the application.

If the user is already verified they will skip the previous step and the system will move straight into verifying that the username and password that were entered are correct, if they are then the user is logged into their account and then are free to navigate the app. If the information is not correct it will loop back and tell the user that the information they input was not correct and they will need to try again.

Once you log in, if you are not verified it will not allow you to move forward. Instead, it will give the user a message telling them to contact their school so that they can gain access. If the verification check comes back in good standing it will then move onto determining what type of user the account should be.

A highschooler is most limited in their actions. They can enter a room and chat, or exit a current room and possibly receive an email with their chat history before exiting. Once they enter

a room and chat, the diagram shows a loop that goes back to check the UserType and wait for the highschooler's next choice.

The College student/advisor is the next type of user, these users have more privilege than the highschoolers. The College students/advisors are able to moderate the chat rooms, enter the

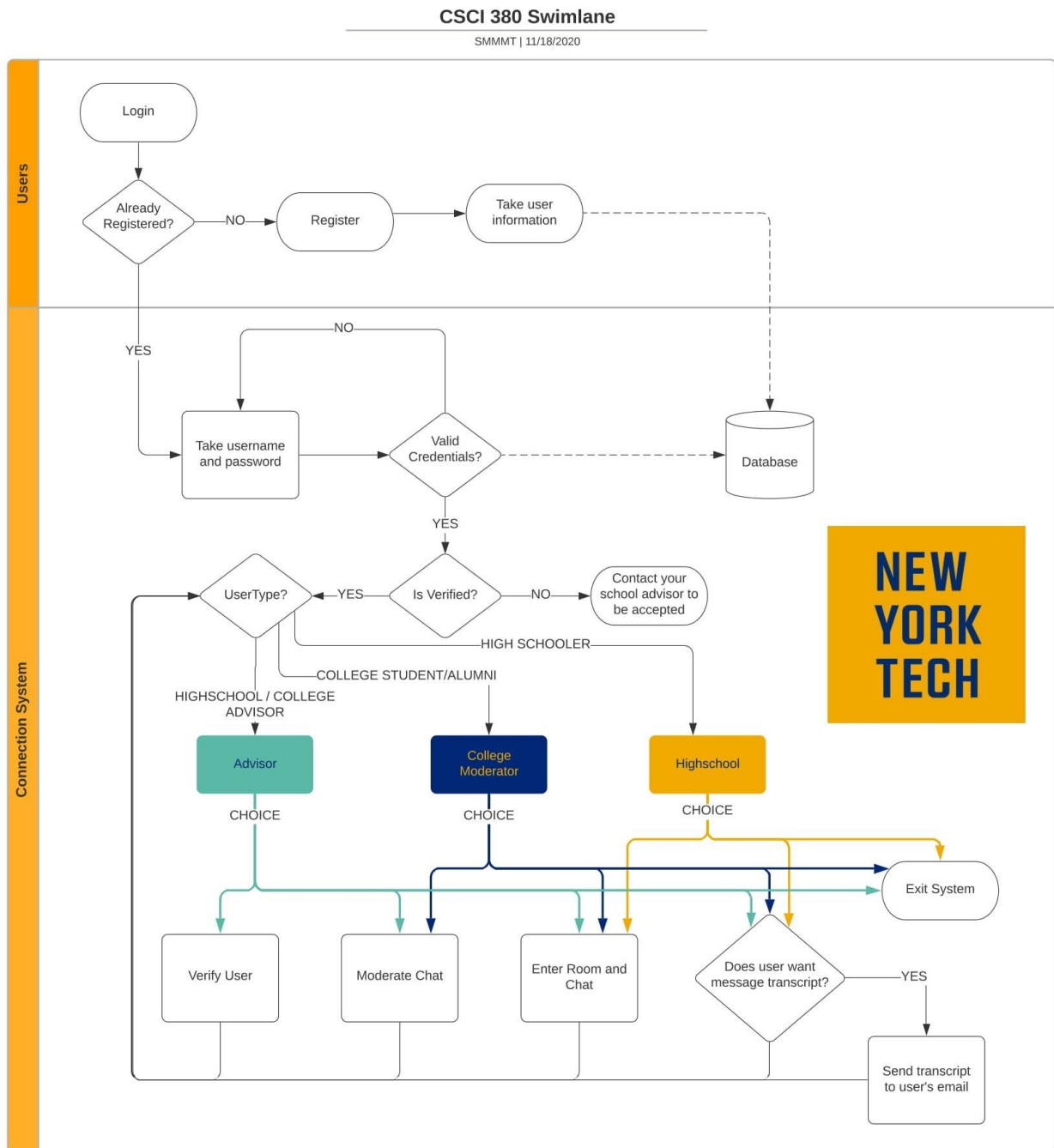


Figure 2.6.1: Swimlane Diagram

chat rooms and send messages to other users, lastly they can request a message transcript of the chat room when they exit.

The final type of use is the advisor. They can access chat rooms, send chats, modify chats, verify users, and request their message transcript on exit. The type of advisor will dictate the type of users the advisor can verify. High School advisors can verify high school students. College advisors can verify College students and moderators. A specialized type of advisor is the “Developer”. While not shown because their abstracted permissions are identical, at a closer level, the developer is the one who verified other advisors. Like the other users, once an action is taken, their UserType is rechecked and waits for their next choice.

When any of these users exit the chat rooms they will then be prompted on whether they want to receive a chat transcript from the room they are exiting. If they reply yes, it will be sent in an email, while if they say no, it will skip this step. Either answer will take them back to the home page. On the home page, all users will also have an option that will let them decide if they want to log out. Every user will have an option to enter a new room or log out of the system once they exit the room.

3.0 Database Design

In terms of our system, we needed a way to store structured information quickly and efficiently. Thus, we turned towards MySQL. Since it was Tim's first time creating a system using SQL, the team wanted to make sure he had the proper support if he needed it. Since the majority of the team had experience in MySQL, it became the obvious choice.

The structured nature of SQL allowed queries to return specified data, rather than an entire JSON file. Thus, in its implementation within C#, the specified needs of the security component could be translated into functions, each with their own queries. To make a connection to the database within C#, the MySql.Data API was used. After setting a string with the database information and opening the connection, statements could be executed. Specifically we split the statements into 'queries' and 'non-queries', with the first returning database information, and the second only manipulating the database.

While layout the structure of our schema, we quickly realized that the tables could be broken into two major entities: users and schools. A user-school table was also included to allow communication between the user and school tables.

Users include any person logging into the system, whether it be a high school student, college mentor, advisor, or a system developer. The information that has to be recorded for a user includes their first, last name, password, user type, email, and a verification field. The first name and last name were then concatenated together to become a username while one chats. These along with a password were needed for authorization into the system. An email is needed so that users can be sent verification codes or emailed message transcripts. The user type helped define what one could access throughout the system, while a verification field told us if the user had been verified, and if not, what their verification code was.

The school entity merely stores the school name and school advisor's first and last name. This allows the system to quickly send a verification code to the proper advisor for a registering user. It as well confirms that any school on file has an appropriate advisor and is thus a verified school.

For our physical implementation of the design into the database, we chose to go with two different data types; we decided upon using mostly the data type 'varchar' with a single enum in tow. We decided upon this because of the versatility of the 'varchar' type. This data type, the variable character, can store numbers, traditional characters, and special characters. This allows us to handle any sort of data that the user might attempt to enter into the entry fields. Another benefit of using this data type is that we as developers can set the maximum amount of characters that the user is allowed to enter, with the maximum as set by the tech specifications being 8000 characters, but who really needs that many characters. As a result of how customizable this type is it was a perfect fit to use for all but one of our entities. The last data type we used is an enum. An enum is a set of values that are stored in a container, the enum. The enum was selected to be used for the 'type' entity. The reason this was selected to be an enum was due to the fact that there are four different types of users that our system can have within it, a HighSchooler, CollegeModerator, Advisor, or Developer. Using an enum allowed us to easily

assign new users a role without having additional clutter on the backend of our system, thereby making it more efficient by having less overhead. However due to the way that an enum works this type needs to have a set default value or else it would give an error, this is different from the varchar which we could leave as null until we set a value to it. We set the default value to be HighSchooler because this role has the lowest access to the system. This is a small but important detail due to the fact that if a new user is not properly identified if they have any other role than HighSchooler they could potentially cause unwanted harm or changes to the application.

3.1 Table Schema

To ensure that our schema removed repeating groups, partial dependencies, and transitive dependencies, we simplified our unnormalized data into third normal form (3NF).

firstName	lastName	password	isVerified	userType	email	schools	school advisor firstNames	school advisor lastNames
Sean	McNamee	testPassword	Verified	Developer	smcnamee@nyit.edu			
Tim	Cameron	7355608	Verified	Developer	tcameron@nyit.edu			
Maherukh	Akhtar	veryNiceSecure123	Verified	Advisor	makhtar@nyit.edu	NYIT	Maherukh	Akhtar
Hal	Abelson	doopDeDop	Verified	Advisor	hal@mit.edu	MIT	Hal	Abelson
Boaz	Barak	noPassNeeded	Verified	Advisor	boaz@seas.harvard.edu	Harvard	Boaz	Barak
Michelle	Falco	smortPass	Verified	Advisor	m.falco@wi.k12.ny.us			
Mary	Lapid	scoopityWoop	Verified	Advisor	mlapid@bayshoreschools.org			
NYIT	Moderator	examplePassword	Verified	CollegeModerator	NYIT@nyit.edu	NYIT	Maherukh	Akhtar
MIT	Moderator	exampleMITPassword	Verified	CollegeModerator	MIT@mit.edu	MIT	Hal	Abelson
Harvard	Moderator	exampleHarvardPassword	Verified	CollegeModerator	Harvard@harvard.edu	Harvard	Boaz	Barak
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org	NYIT, Harvard	Maherukh, Boaz, Hal	Akhtar, Barak, Abelson
Brad	Person	thisIsntSecure2	Verified	HighSchooler	bperson@bshs.org	MIT, Harvard	Hal, Boaz	Abelson, Barak
Clark	Person	thisIsMoreSecure	918274	HighSchooler	cperson@wiufsd.org	Harvard	Boaz	Barak
Dorothy	Person	tHiSiSmOReSeCuRe	182264	HighSchooler	dperson@bshs.org	NYIT, MIT	Maherukh, Hal	Akhtar, Abelson

Figure 3.1.1: Unnormalized schema with data

One will notice immediately that figure 3.1.1's primary key is the combination of firstName and lastName. As well, it is apparent that there are repeating groups within it. To turn this unnormalized relationship into a normalized relation, these groups were separated into multiple records, as seen in figure 3.1.2.

firstName	lastName	password	isVerified	userType	email	schoolName	school advisor firstName	school advisor lastName
Sean	McNamee	testPassword	Verified	Developer	smcnamee@nyit.edu			
Tim	Cameron	7355608	Verified	Developer	tcameron@nyit.edu			
Maherukh	Akhtar	veryNiceSecure123	Verified	Advisor	makhtar@nyit.edu	NYIT	Maherukh	Akhtar
Hal	Abelson	doopDeDop	Verified	Advisor	hal@mit.edu	MIT	Hal	Abelson
Boaz	Barak	noPassNeeded	Verified	Advisor	boaz@seas.harvard.edu	Harvard	Boaz	Barak
Michelle	Falco	smortPass	Verified	Advisor	m.falco@wi.k12.ny.us			
Mary	Lapid	scoopityWoop	Verified	Advisor	mlapid@bayshoreschools.org			
NYIT	Moderator	examplePassword	Verified	CollegeModerator	NYIT@nyit.edu	NYIT	Maherukh	Akhtar
MIT	Moderator	exampleMITPassword	Verified	CollegeModerator	MIT@mit.edu	MIT	Hal	Abelson
Harvard	Moderator	exampleHarvardPassword	Verified	CollegeModerator	Harvard@harvard.edu	Harvard	Boaz	Barak
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org	NYIT	Maherukh	Akhtar
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org	MIT	Hal	Abelson
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org	Harvard	Boaz	Barak
Brad	Person	thisIsntSecure2	Verified	HighSchooler	bperson@bshs.org	MIT	Hal	Abelson
Brad	Person	thisIsntSecure2	Verified	HighSchooler	bperson@bshs.org	Harvard	Boaz	Barak
Clark	Person	thisIsMoreSecure	918274	HighSchooler	cperson@wiufsd.org	Harvard	Boaz	Barak
Dorothy	Person	tHiSiSmOReSeCuRe	182264	HighSchooler	dperson@bshs.org	NYIT	Maherukh	Akhtar
Dorothy	Person	tHiSiSmOReSeCuRe	182264	HighSchooler	dperson@bshs.org	MIT	Hal	Abelson

Figure 3.12: Table schema with data in 1NF

Since all values were now atomic and thus in first normal form (1NF), the next focus was on partial dependencies. However, the separation of repeating groups has forced the primary key to also include schoolName. At a closer look, one can tell that the school advisor firstName and school advisor lastName are dependent only on schoolName rather than the entire primary key. Thus, schoolName, school advisor firstName, and school advisor lastName were extracted into a second relation. Since schoolName was still a part of the table's primary key, it became an attribute of both tables, as seen in figure 3.1.3. This puts the schema into second normal form (2NF).

firstName	lastName	password	isVerified	userType	email	schoolName	schoolName	school advisor firstName	school advisor lastName
Sean	McNamee	testPassword	Verified	Developer	smcnamee@nyit.edu		NYIT	Maheerukh	Akhtar
Tim	Cameron	7355608	Verified	Developer	tcameron@nyit.edu		MIT	Hal	Abelson
Maheerukh	Akhtar	veryNiceSecure123	Verified	Advisor	makhtar@nyit.edu	NYIT	Harvard	Boaz	Barak
Hal	Abelson	doopDeDop	Verified	Advisor	hal@mit.edu	MIT			
Boaz	Barak	noPassNeeded	Verified	Advisor	boaz@seas.harvard.edu	Harvard			
Michelle	Falco	smortPass	Verified	Advisor	m.falco@wi.k12.ny.us				
Mary	Lapid	scoopyWoop	Verified	Advisor	mlapid@bayshoreschools.org				
NYIT	Moderator	examplePassword	Verified	CollegeModerator	NYIT@nyit.edu	NYIT			
MIT	Moderator	exampleMITPassword	Verified	CollegeModerator	MIT@mit.edu	MIT			
Harvard	Moderator	exampleHarvardPassword	Verified	CollegeModerator	Harvard@harvard.edu	Harvard			
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org	NYIT			
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org	MIT			
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org	Harvard			
Brad	Person	thisIsntSecure2	Verified	HighSchooler	bperson@bshs.org	MIT			
Brad	Person	thisIsntSecure2	Verified	HighSchooler	bperson@bshs.org	Harvard			
Clark	Person	thisIsMoreSecure	918274	HighSchooler	cperson@wiufsd.org	Harvard			
Dorothy	Person	thisIsSmOReSeCuRe	182264	HighSchooler	dperson@bshs.org	NYIT			
Dorothy	Person	thisIsSmOReSeCuRe	182264	HighSchooler	dperson@bshs.org	MIT			

Figure 3.1.3: Table schema with data in 2NF

Simplifying the 2NF schema into third normal form (3NF) required removing transitive dependencies. To spot these, one must only find repetitive data across records or data that is not part of the entity. Taking a closer look at figure 3.1.3 reveals that all but the schoolName is repetitive for Alan Person and Dorothy Person. Thus, the schoolName can be removed from the left relation, copying the firstName and lastName values to a new table with it so it can still be referenced. The finalized 3NF schema with data can be seen in figure 3.1.4. It eliminates redundant data storage, and maintains data integrity and consistency.

firstName	lastName	password	isVerified	userType	email	schoolName	school advisor firstName	school advisor lastName
Sean	McNamee	testPassword	Verified	Developer	smcnamee@nyit.edu	NYIT	Maheerukh	Akhtar
Tim	Cameron	7355608	Verified	Developer	tcameron@nyit.edu	MIT	Hal	Abelson
Maheerukh	Akhtar	veryNiceSecure123	Verified	Advisor	makhtar@nyit.edu	Harvard	Boaz	Barak
Hal	Abelson	doopDeDop	Verified	Advisor	hal@mit.edu			
Boaz	Barak	noPassNeeded	Verified	Advisor	boaz@seas.harvard.edu			
Michelle	Falco	smortPass	Verified	Advisor	m.falco@wi.k12.ny.us			
Mary	Lapid	scoopyWoop	Verified	Advisor	mlapid@bayshoreschools.org			
NYIT	Moderator	examplePassword	Verified	CollegeModerator	NYIT@nyit.edu			
MIT	Moderator	exampleMITPassword	Verified	CollegeModerator	MIT@mit.edu			
Harvard	Moderator	exampleHarvardPassword	Verified	CollegeModerator	Harvard@harvard.edu			
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org			
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org			
Allan	Person	thisIsntSecure	Verified	HighSchooler	aperson@wiufsd.org			
Brad	Person	thisIsntSecure2	Verified	HighSchooler	bperson@bshs.org			
Brad	Person	thisIsntSecure2	Verified	HighSchooler	bperson@bshs.org			
Clark	Person	thisIsMoreSecure	918274	HighSchooler	cperson@wiufsd.org			
Dorothy	Person	thisIsSmOReSeCuRe	182264	HighSchooler	dperson@bshs.org			
Dorothy	Person	thisIsSmOReSeCuRe	182264	HighSchooler	dperson@bshs.org			

firstName	lastName	schoolName
Maheerukh	Akhtar	NYIT
Hal	Abelson	MIT
Boaz	Barak	Harvard
NYIT	Moderator	NYIT
MIT	Moderator	MIT
Harvard	Moderator	Harvard
Allan	Person	NYIT
Allan	Person	MIT
Brad	Person	Harvard
Brad	Person	MIT
Clark	Person	Harvard
Dorothy	Person	NYIT
Dorothy	Person	MIT

Figure 3.1.4: Table schema with data in 3NF

To increase readability and make it easier to denormalize for our implementation into MySQL, we simplified these tables into figure 3.1.5. Note that the primary keys are colored in,

while the foreign keys are bolded. The relations were as well labeled according to the entities they represent or connect: user, school, and user-school.

USER					
firstname	lastname	password	userType	isVerified	email
SCHOOL					
schoolName	advisorfirstName	advisorLastName			
USER-SCHOOLS					
firstname	lastname	schoolName			

Figure 3.1.5: Table Schema in 3NF

To further showcase the functionality of the structured information of our schema, we have provided a sample of our queries that provide functionality for the security component of our system.

```
select schoolName
FROM `csci380`.`user-school`
WHERE (firstName, lastName)=('Tim', 'Cameron');
```

This query shows how to return a list of all the schools associated with a requested user. The query takes in the firstName and lastName of the individual whose records we seek and then searches the user-school table for a location in which the input data matches a data store. Once that location is identified within the database, the query will return a list containing the name(s) of the school(s) that is associated with that individual to the application. The data retrieved from this query will then be used to populate the dropdown menu to select the accessible chat rooms in which the user is allowed to chat in.

```
select email
FROM `csci380`.`user`
WHERE (firstName, lastName)=(
    select advisorFirstName, advisorLastName
    FROM `csci380`.`school`
    WHERE school='TestSchool');
```

This second query is used when a user registers for an account. When a user first registers, they are unverified and can't access any of the major components of the system. To become verified, their advisor must enter that user's firstName, lastName, and verification code.

To receive this information, the advisor is sent an email. Thus, this query gets the email of the advisor associated with a given school. To achieve this result, a subquery gets the advisorFirstName and advisorLastName for a specific school from the school table. Since the advisorFirstName and advisorLastName are foreign keys to the user table, they are used to locate the advisor's email that resides within the user table. This query showcases the organization and structure of our schema.

3.2 Entity Relationship Diagram

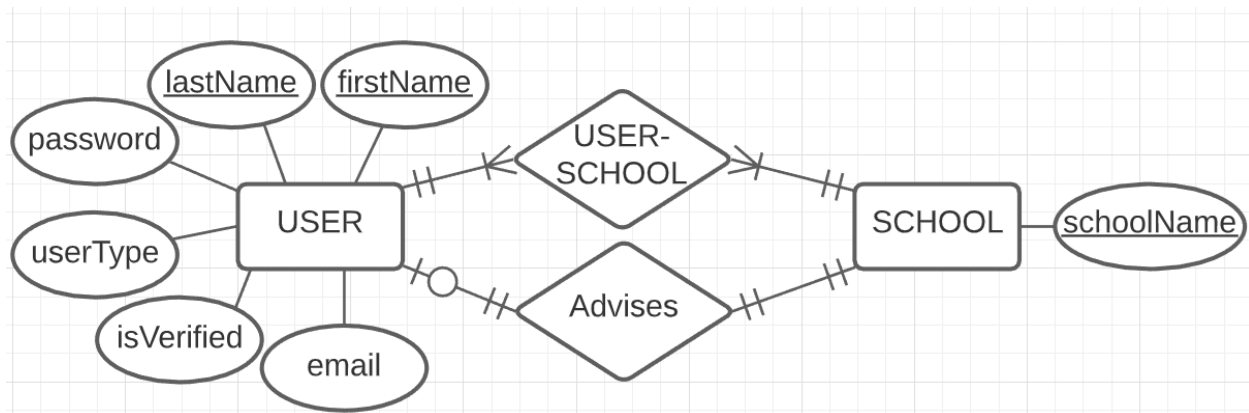


Figure 3.2.1: Entity Relationship Diagram

While the table schema provides a data-driven representation of the relations, the ER diagram provides a visual representation. At first, one may question the non-obligatory relationship between 'Advises' and 'USER'. However, a closer look at the intricacies reveal the reasoning. While all schools require an advisor for verification, not all users are advisors of a school; high schoolers and college moderators only chat, and never worry about verifying other users. This contrast to the schools a user can chat in allows for there to be two connections between the school and user entities. A school and user have a cardinality of many to many in terms of chattable schools, while the cardinality in terms of advising is one to zero or one. These cardinalities also show that user-school should contain a primary key composed of two sets of foreign keys, one that points to the user table primary keys, and one that points to the school table primary key. They also show that a school should contain the firstName and lastName of an advisor that resides in the user table.

4.0 Functionality and Implementation

The reason we choose to use Unity is because of the challenge. Many of the group members have never used Unity before and it was going to be a good learning experience for everyone. Unity makes creating UI easy and implementation of MySQL databases easy as well. This was very good for our project as we wanted to store and access data constantly. We decided to use C# as our main programming language only because of its Unity integration. Unity was built around C# and is heavily documented throughout the years. This provided the team with many resources to investigate, especially the only unity scripting API. Other online tools like MySQL and SMTP were documented in C#, which allowed everything to come together piece by piece, with the utilization of only one programming language.

4.1 Features

Features are the components that makeup a piece of software, and like any good piece of software our application is filled with features both large and small. Our crowning jewel of a feature is the chatting system. This feature allows different users to communicate with one another as it was laid out in our proposed system statement. Within the chatting system there are three other features that have been built in, chat in different rooms based on roles, moderate chat, and send email transcripts. By having chat rooms be locked by roles it allows only the intended individuals to access the appropriate channels. Moderating chat is an exclusive feature only accessible to users above high schooler. By having moderators in chat rooms they make sure that the conversations are kept appropriate and on topic. Once users are completed chatting and moderating they are given the option to have an email sent with a transcript of the conversation. This allows the user to go back and look at previous conversations since they cannot look at them within the application. However, prior to being able to chat a user must first be registered. This feature entails entering in their personal information, creating a password, and assigning themselves the appropriate roles; once the user enters all of this data they will be registered within the system. Once a user is registered their data will be stored within the database. Now the user is eligible to be verified so they can access the full scope of the application. The verification process entails an approved advisor for that user going to the verify user page and entering the users first name, last name, and verification code, all of which are provided in an email to the advisor. Once the new user is verified they will have full access to the system. This then brings us to our last feature, logging in and logging out. These are very basic but overlooked features as they are quite literally the gatekeepers of an entire system. The login feature requires that a user enter their first name, last name, and password after that they will click the login button and have access to their appropriate chat rooms and privileges. The logout feature is similarly just as simple, once a user no longer wishes to be logged into their instance they will click the logout button and be logged out of the application.

4.2 Security

In many channel-based communication systems, confirming one's identity is often overlooked. Users must trust developers or admins of the system to accurately depict the identities of other users. For systems where profiles are visible, this depiction is illustrated through the use of special statuses. For both Instagram and Twitter, this is the purpose of the “Verified” check mark. However, since channel-based systems such as ours do not include viewable profiles, it is crucial to have a strong verification system to ensure identities are accurate.

To ensure security within the 6-digit verification code, we created a random number between 10^6 and $10^7 - 1$. If we ever felt that 6 digits was not enough, we could increase this by changing a single variable. Only the proper advisor would receive the verification, and thus verify the user. Only advisors have access to verify users, so this limits attempts from high school students or college moderators. Furthermore, only developers can verify typical advisors, so this creates a hierarchy of security.

One underlying error of our verification is if a Developer accidentally verifies a malicious advisor. Luckily, the most they would be able to do is verify students for their assigned school. If necessary we could just perform a delete statement for that user in the MySQL command line client. If this became a pressing issue, a page could be added for developers to remove users from the system. This error would obviously not be too overbearing since this relies on a developer to make a ‘human’ related error. Additionally, advisors can not verify other advisors, thus further limiting the impact of this error.

In terms of login credentials, passwords are encrypted by manipulating each character's ascii number, then converting back to a character within the range [32, 128]. This ensures that any weird characters from the range [0, 31] are not included in stored passwords. These encrypted passwords are stored on the database. When one tries to log in, the encrypted password is pulled down from the database. Rather than create an entire decryption method, we encrypt the entered password and check if they match.

4.2.1 Privilege Levels

The privilege levels in our system are based on a hierarchy. Advisors are at the top, and unregistered users are the entrypoint. To further understand the privilege levels of users within our system, figure 4.2.1.1 has been provided. To elaborate further on how this hierarchy functions, we'll walk through each usertype. Unregistered users do not have accounts, so they have to register. Once registered, any user can login, join rooms that they have access to, chat in those rooms, and receive a transcript of their chat history on exit. One note to make is that since high school advisors have no room to chat in, they can't join rooms, chat, or receive a transcript from those chats. The next level in the hierarchy is moderators. They can moderate chats within the system. After moderators are advisors. They are meant to verify students from their school. College advisors would verify college mentors while High school advisors would verify high

schoolers. Developers are a special type of advisor that can verify other advisors. They can also join any chat room, regardless of which chat room they are explicitly linked to.

User	Register	Login	Join Rooms	Chat	Receive Chat Transcript	Moderate Chat	Verify same-school students	Verify Advisors
Unregistered	✓	✗	✗	✗	✗	✗	✗	✗
High Schooler	✗	✓	✓	✓	✓	✗	✗	✗
College Moderator	✗	✓	✓	✓	✓	✓	✗	✗
Advisor	✗	✓	✓	✓	✓	✓	✓	✗
Developer	✗	✓	✓	✓	✓	✓	✓	✓

Figure 4.2.1.1: Privilege Levels Chart

4.3 File Structure

The basic layout for our project was mainly focused on keeping things organized. Unity by default will create a set of folders that will house the different parts of the project. The most important folder that was created is the Assets folder, this houses images, resources, scenes, scripts, and meta files.

The images folder is pretty self explanatory, it holds images that the developer is going to add into the project, in our case the folder is filled with school logos. The resources folder can be filled with information about multiple items, for our project we used it to create test users. The scenes folder is where all the scenes are stored, this allows for easy moving between all the pages. The scripts folder is next, this folder will store any scripts that are made to allow for small actions to occur, an example of this is the “SceneSwitching.cs” script. When the user clicks on a button it will call this script to move to the next/previous page. The last folder under Assets is the TextMesh Pro folder. This folder stores assets that allow you to input fonts, and other text manipulation styles, like bold, italics, and it also stores emojis. Lastly the meta files are stored inside of the assets folder because Unity will not store some data of an asset in the asset itself.

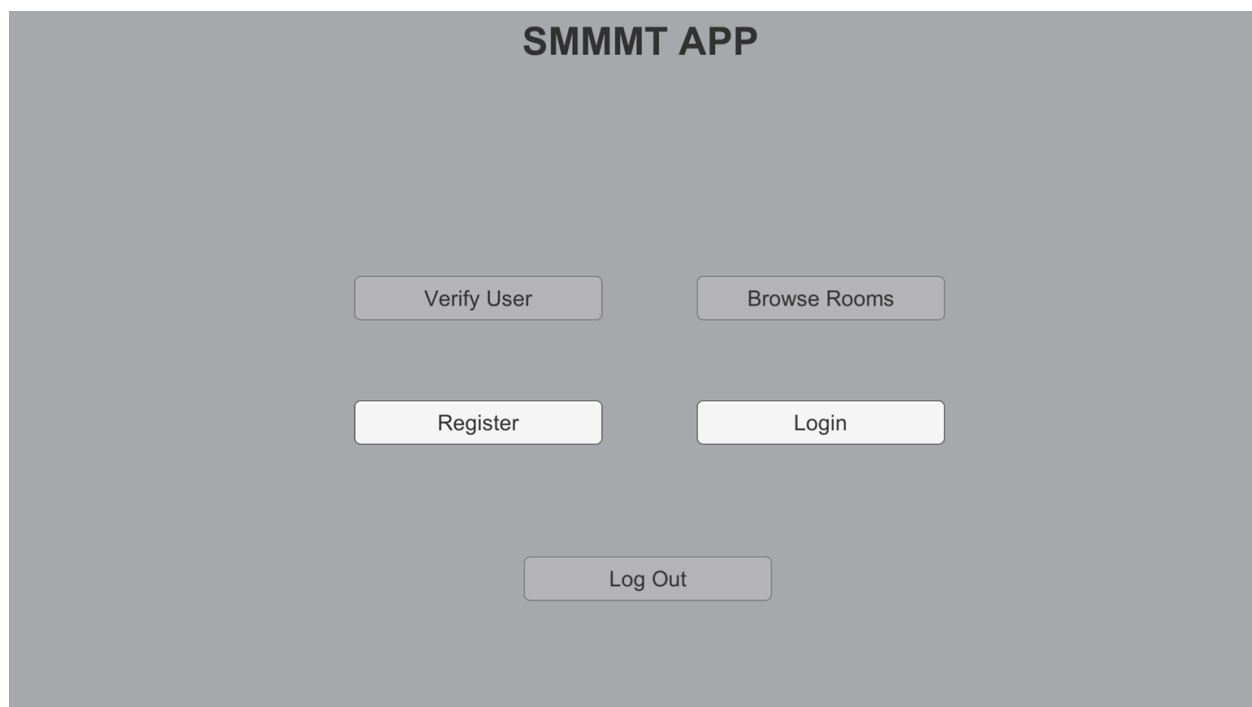
The next folder is the Library folder, this folder is the main folder for most of Unity’s tools. A lot of the tools inside of this folder are meant for people creating 3D applications, that will require legitimate scenes to be created. However there are a lot of tools in here that can be applied to something like our project, these folders contain .dll files, this is where most of the files that allow for creation and modification of 2D scripts are located.

In unity the scripts can communicate with one another by looking for classes or tags that match. An example of this is a user interacting with the UI, if the two scripts cannot talk to one

another then neither of them will work. They both operate by pulling information from each other and comparing it to guarantee that they match, ie. user information. In Unity if two scripts are meant to talk to one another and one of them gets deleted it will cause both to stop working, even if one of them wouldn't need the other to operate.

To better organize the scripts themselves, we separated them into four name spaces: Page, SecAuth, DB, and Data. The Page namespace dealt with grabbing user inputs from its respective UI page and calling methods of the SecAuth classes to perform tasks such as Login, Verify, and Register. The SecAuth classes dealt with the security components, including the Authenticator and Email Sender. They utilized the DB namespace to perform queries quickly. The Data namespace was used to pass user information throughout the program. A static variable called 'User' was able to be accessed from any page, and would provide information vital to that page's function.

4.3.1 Main Menu UI Example



This is a look at our prototype(s) main menu (Not Logged In). Here there is no programming needed to create the actual menu. All these tools are simple drag and drops, and the only bits of code will come from the "Register" & "Login" buttons. Both these buttons will have the same code, which is displayed down below:

Here we are utilizing the SceneManager library because we want to switch to the login or register “Menu”. In Unity, new scenes are called many things like game levels, menus, options, and so on. It is a simple function that simply passes a string (Which is the same as the “menu” we want to jump to) once the button is clicked. The function simply finds the appropriate “menu” and switches to that menu. This is a fast, simple, and easy way to switch between slides and not over crowded one specific scene with too much information. Keeping scenes centered around their main purpose is allowing us to make the code run smoother and the user experience to be straightforward.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class SceneSwitching : MonoBehaviour
{
    [SerializeField]
    private Dropdown dropdown = null;

    public void SwitchToMenu()
    {
        SceneManager.LoadScene(menuSwitch);
    }
}
```

Figure 4.3.1.2 : Scene Switching Program

```

public string username;

public int maxMessage = 25;

public GameObject chatPanel, textObject;
public InputField chatBox;

public Color playerMessage, info;

[SerializeField]
List<Message> messageList = new List<Message>();

```

Figure 4.3.2.1 : Variable initialization

4.3.2 Chat System Page

This first part of the code is meant for us to initialize some basic variables that are going to be used throughout the program. We start with the “username” string, which is something that can either be set in the code or the inspector window of unity. Following that, we have the “maxMessages” that can be displayed in the scroll view of the chat system. We can change this value in the code or the inspect but is

always set to 25 as default. Next, we want to be able to set up the location of where the chat messages will be sent, displayed, and inputted.

We use the “GameObject” keyword because we want to get the properties of those specific objects. After making them public, we need to set them manually in the unity hierarchy, and after this step, we can access all components of the object. We then have an “InputField”, which is where the user will be entering the chat message they want to send. This is set the same way as the “GameObjects” and allows us to get all the properties of the input field.

We make a simple Color variable because we want to be able to distinguish between user inputs and admin/system inputs. The colors are then set in the Unity inspector and read later in the “MessageTypeColor” function.

Lastly, we make a list with the name of “messageList”. We do not specify a length to this list because it will be as big or as small as we want based on the “maxMessage” variable. We do not use an array here because lists are more flexible for this situation and are easier to duplicate and send to the user later on in the emailing process.

```

void Update()
{
    if (chatBox.text != "")
    {
        if (Input.GetKeyDown(KeyCode.Return))
        {
            SendMessageToChat(username + ": " + chatBox.text, Message.MessageType.playerMessage);

            if (chatBox.text.ToLower() == "hi" || chatBox.text.ToLower() == "hello")
            {
                SendMessageToChat("Admin: Hello", Message.MessageType.info);
            }

            chatBox.text = "";
        }
    }
    else
    {
        if (!chatBox.isFocused && Input.GetKeyDown(KeyCode.Return))
        {
            chatBox.ActivateInputField();
        }
    }
}

```

Figure 4.3.2.2 : Per Frame Update

Here we check every frame for any updates in the application. First, we check if the input box where the user would input their message is selected, this makes sure that the user cannot spam messages by holding the enter key. Then we read the message and start passing everything through the “SendMessageToChat”, and this function makes sure that the message being sent is not an empty string. As a response system, if the user enters a specific message, the system will respond with a predetermined message. This allows us to simply add responses to everyday questions like office phone numbers for the specific schools. If the user has not clicked on the input field, then the screen will have no difference and wait until the input field is clicked on, or if the user interacts with the UI.

```

public void SendMessageToChat(string text, Message.MessageType messageType)
{
    //Deleting Oldest Message
    if (messageList.Count >= maxMessage)
    {
        Destroy(messageList[0].textObject.gameObject);
        messageList.Remove(messageList[0]);
    }
    //

    Message newMessage = new Message();

    newMessage.text = text;

    GameObject newText = Instantiate(textObject, chatPanel.transform);

    newMessage.textObject = newText.GetComponent<Text>();

    newMessage.textObject.text = newMessage.text;
    newMessage.textObject.color = MessageTypeColor(messageType);

    messageList.Add(newMessage);
}

```

Figure 4.3.2.3 : Sending Chat Function

This is the brains of the chatting system. We have a function here that takes in the text that the user has inputted into the input field and takes in the type of message being passed. The type will distinguish which kind of color the text will display as, and help distinguish between user inputs, and system / admin inputs.

We start by doing a check on the “messageList” list because we want to only display the number of messages designated by the “maxMessage” variable. If the “messageList” has gone beyond that limit, it will take the earliest saved message from the list and remove it. This frees up another space in the list so that any new messages being passed can be saved in the list.

Next, we make a new object from the existing “Message” object known as newMessage. Now the program takes the “text” string that is a parameter for the function and saves it to the “newMessage” text object. We then make a GameObject called “newText”, and instance its textObject and the position in which it will be in the simulation. We set the “newMessage” to the previous object called newMMessage.Text. We set the color of the message after calling the

“MessageTypeColor” function with the “messageType” variable being passed. Lastly, we add the newMessage to the “messageList” list.

```
Color MessageTypeColor(Message.MessageType messageType)
{
    Color color = info;

    switch (messageType)
    {
        case Message.MessageType.playerMessage:
            color = playerMessage;
            break;
        case Message.MessageType.info:
            color = info;
            break;
        default:
            break;
    }

    return color;
}
```

This function distinguishes what color the text will be when it is displayed in the text box. This is based on the “type” of message being sent. The switch state only distinguishes two different kinds of types, and those are info, and playerMessage (user message). At the end we return the color, and display the message based on that returned color.

Figure 4.3.2.4 : Message Coloring

This class sets up the Message. It will contain the text, textObject being projected too, and the type of the message. We have set there to be two types, which are distinguished by playerMessage, and info.

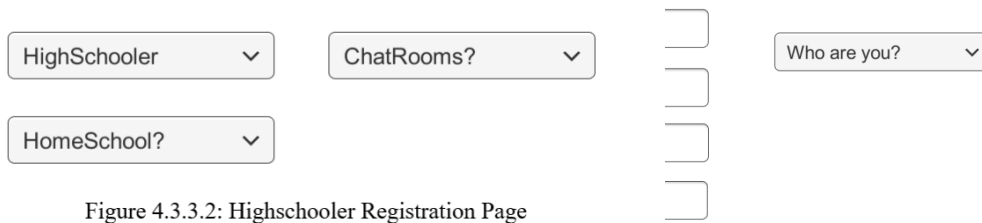
```
public class Message
{
    public string text;
    public Text textObject;
    public MessageType messageType;

    7 references
    public enum MessageType
    {
        playerMessage,
        info
    }
}
```

Figure 4.3.2.5 : Message Type

4.3.3 Registration

When a user enters the registration page, they are welcomed by inputs for their first and last name, password, email, and usertype, as shown in Figure 4.3.3.1. When selecting the usertype, more input areas appear, as shown in Figure 4.3.3.2. For a Highschooler and CollegeModerator, they have to select their home school from the bottom-most dropdown. Highschoolers must as well add colleges that they wish to chat in from the rightmost dropdown. Advisors on the other hand register a school with the system, so they manually enter the school's name in the text area.

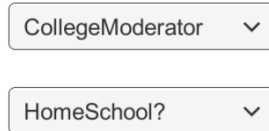


HighSchooler ▼ ChatRooms? ▼

HomeSchool? ▼

Who are you? ▼

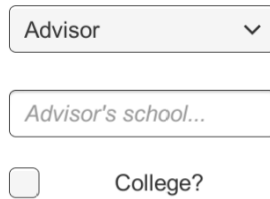
Figure 4.3.3.2: Highschooler Registration Page



CollegeModerator ▼

HomeSchool? ▼

Figure 4.3.3.3: CollegeModerator Registration Page



Advisor ▼

Advisor's school...

☐ College?

Figure 4.3.3.4: Advisor Registration Page

4.3.3.1: UI of Registration Page

The behind the ‘scene’ action when a user registers is where the real functionality begins. Before understanding how the Register page works, once has to understand how C#

files communicate with Unity. ‘SerializeField’ is used to denote any variables that can be initialized through Unity, while Unity can only call methods with no parameters.

Figure 4.3.3.5 shows the variables that were used in the RegisterPage class. Notice that all the variables with 'SerializeField' correspond directly to an element on the UI page. The variables that do not have 'SerializeField' are simply the strings that are the first value of the dropdown boxes in unity. The list 'schools' does not contain anything needing a direct connection to something happening in Unity since schools is a list of the schools that the user has chosen to be able to chat in.

In figure 4.3.3.6, the private methods created are all meant to be helper methods to be called in UserTypeSet(). showAdvisorInput(bool show) is to clean up the register page and only show the possible inputs when the user has chosen Advisor as their user type. showChatSchools(bool show) is used to show the schools that a user wants to chat in and is only used when a user is a high schooler. showHomeSchools(bool show) is a method which will show and allow high schoolers and college moderators to input their homeschools in a dropdown, however, if the user wants to be an advisor it would instead be a text box since they are creating the first instance of their school to be the advisor of it. The showSubmit(bool show) method is simple as used to by default show the submit button for Advisors and College

```
[SerializeField]
private InputField firstName;
[SerializeField]
private InputField lastName;
[SerializeField]
private InputField password;
[SerializeField]
private InputField email;
[SerializeField]
private Dropdown userType;

[SerializeField]
private Dropdown homeSchool;
[SerializeField]
private InputField advisorSchool;
[SerializeField]
private Toggle IsCollege;
[SerializeField]
private Dropdown schoolDropDown;
[SerializeField]
private Text textShowingSchools;

[SerializeField]
private Button submit;
[SerializeField]
private string menuSwitch;

private string chatrooms = "ChatRooms?";
private string homeSchoolText = "HomeSchool?";

public static List<string> schools = new List<string>();
```

Figure 4.3.3.5: RegisterPage Class Variables

moderators, but the high schoolers must choose a chat room for the submit button to appear. This is shown in the method below called AddSchool in figure 4.3.3.7.

```
public void UserTypeSet() {
    showAdvisorInput(userType.value == 3);
    showHomeSchools(userType.value == 1 || userType.value == 2);
    showChatSchools(userType.value == 1);
    showSubmit(userType.value == 2 || userType.value == 3);

    setTextForSchools();
}

private void showAdvisorInput(bool show) {
    //advisorSchool.interactable = show;
    if (!show) {
        advisorSchool.text = "";
    }
    advisorSchool.gameObject.SetActive(show);
    IsCollege.gameObject.SetActive(show);
}

private void showChatSchools(bool show) {
    if (!show) {
        schools.Clear();
    }
    schoolDropDown.gameObject.SetActive(show);
}

//you, a day ago • adding schools includes if its a college or high ...
private void showHomeSchools(bool show) {
    if (show) {
        Queries conn = new Queries();
        homeSchool.ClearOptions();
        List<string> homeSchoolsList = conn.getAllSchools(userType.value == 2);
        homeSchoolsList.Insert(0, homeSchoolText);
        homeSchool.AddOptions(homeSchoolsList);
        conn.closeConenction();
    }
    //homeSchool.interactable = true;

    homeSchool.gameObject.SetActive(show);
}

private void showSubmit(bool show) {
    submit.gameObject.SetActive(show);
}
```

Figure 4.3.3.6: RegisterPage: Actions on switching user type

```
void Start() {
    Queries conn = new Queries();
    schoolDropDown.ClearOptions();
    List<string> collegeRooms = conn.getAllSchools(true);
    collegeRooms.Insert(0, chatrooms);
    schoolDropDown.AddOptions(collegeRooms);
    conn.closeConenction();
    UserTypeSet();
}

public void AddSchool() {
    Debug.Log(schools.Count);

    if (schoolDropDown.value > 0) {
        string schoolToAdd = schoolDropDown.captionText.text;

        if (schools.Contains(schoolToAdd)) {
            schools.Remove(schoolToAdd);
        } else {
            schools.Add(schoolToAdd);
        }

        setTextForSchools();
    }

    showSubmit(schools.Count > 0);
    schoolDropDown.value = 0;
}

private void setTextForSchools() {
    string text = "";
    foreach (string school in schools) {
        text += school;
        if (school != schools[schools.Count-1]) {
            text += ", ";
        }
    }
    textShowingSchools.text = text;
}
```

Figure 4.3.3.7: RegisterPage: Start and AddSchool

When the Register Page in Unity is loaded, Start() is called, as seen in Figure 4.3.3.7. Therefore, on start, the colleges that a highschooler can select are loaded into the schoolDropDown. Thus, when a highschooler wants to choose schools to chat in, the dropdown is ready. Whenever a school in the schoolDropDown is selected, AddSchool() is called. This adds or removes the selected school to the schools list. It resets the shown school on the dropdown to the chatrooms text of “ChatRooms?”, shows the submit button if the schools list has a value, and refreshes the text area that displays the currently selected schools. Refreshing the text for schools in the setTextForSchools displays all the schools within the schools list (the chat rooms currently selected). Thus, the user can visually see their changes before submitting.

```

public void Register() {
    //Login with the provided credentials
    string strfirstName = firstName.text;
    string strlastName = lastName.text;
    string struserPassword = password.text;
    string strEmail = email.text;
    int intUserType = userType.value;

    string strhomeSchool = "";
    if (intUserType == 3) {
        //Only advisors input their own school
        strhomeSchool = advisorSchool.text;
    } else if (homeSchool.value > 0) { //All other users choose from the dropdown (0 isn't allowed)
        strhomeSchool = homeSchool.captionText.text;
    }

    //Make sure above information isn't bad
    if (Authenticator.IsValidString(strfirstName) && Authenticator.IsValidString(strlastName) &&
        Authenticator.IsValidString(struserPassword) && Authenticator.IsValidString(strEmail) &&
        Authenticator.IsValidEmail(strEmail) && (intUserType > 0) && (intUserType < 4)) {
        Authenticator.Register(strfirstName, strlastName, struserPassword,
            ((User.UserType) intUserType), strEmail, strhomeSchool, schools, IsCollege.isOn);
        SceneManager.LoadScene(menuSwitch);
    } else {
        textShowingSchools.text = "Something isn't filled out correctly";
    }
}
}

```

Figure 4.3.3.8: RegisterPage: Register if acceptable values

In figure 4.3.3.8, the Register method establishes variables which are the ones required for login credentials. The first if condition makes sure the user type is three meaning that they are an advisor in order to allow advisors to input their own school in the text box. The else if directly after allows all other users to choose their home school from the dropdown. The second if

condition checks to make sure that all user input is valid, else it will output “Something isn’t filled out correctly”.

4.3.4 Verification

The authenticator had to deal with authentication, verification, and successfully logging in to the system. After registering an email is sent which contains the verification information; The verifiers must then verify them if they are a legitimate user. Once they have been verified the user gains full access to the application. Advisors and developers are the only user types who have the capabilities to verify other users.

Figure 4.3.4.1 details the process that happens when a user registers for an account. For any user, their password is encrypted and a verification code is generated. A connection to the database is then established through the instantiation of a Queries object. With this object, the user is inserted into the users table. If the user has chat room schools, those are added to the user-school table. Next, the email of the verifier is obtained. For advisors, their verifier is a developer. For high schoolers and college moderators, their verifier is the advisor of their school. After

```
private static System.Random randomNum = new System.Random();

public static void Register(string firstName, string lastName, string password, User.UserType userType,
                           string email, string homeSchool, List<string> schools, bool isCollege) {
    password = PasswordEncryption(password);
    string verificationCode = GenerateVerificationCode();

    //Store user info into database
    Queries conn = new Queries();
    conn.insertUser(firstName, lastName, password, userType, verificationCode, email);

    //Storing USER-SCHOOL values if needed
    if (schools != null && schools.Count > 0) {
        Debug.Log("Will be added");
        foreach (string school in schools) {
            Debug.Log(school);
            //Store Username and School in USER-SCHOOL
            conn.insertUserSchool(school, firstName, lastName);
        }
    } else {
        Debug.Log("No schools added");
    }

    //Send email with verification code
    string emailOfVerifier = null;
    if (userType == User.UserType.HighSchooler || userType == User.UserType.CollegeModerator) {
        // get email of advisor (using homeSchool to find them)
        Debug.Log("Home School: " + homeSchool);
        emailOfVerifier = conn.getAdvisorEmail(homeSchool);
    } else {
        //get email of developer (search for developer userType)
        conn.insertSchool(homeSchool, firstName, lastName, isCollege);
        emailOfVerifier = conn.getDeveloperEmail();
    }

    conn.closeConenction();
    //Send email to advisor
    sendAuthenticationEmail(emailOfVerifier, homeSchool, firstName, lastName, email, verificationCode);
}
```

Figure 4.3.4.1: Verification: Registration

obtaining the advisor, the Queries object's connection is closed. Lastly, the generated verification code is sent to the verifier along with the user's information.

To ensure that the system does not get any SQL injections and that data should be accepted, IsValidString and IsValidEmail were created, as shown in Figure 4.3.4.2. IsValidString ensures a string is not null, empty, or contains a semicolon, which is used for SQL statements. IsValidEmail utilized the MailAddress class to ensure an email follows the proper format.

VerifyAccount gets the stored verification code from the database for a user and compares it to the entered verification code. If they are equal, it sets the verification code of the provided user to "Verified", meaning that users can now log in and gain full access to the system's functionality.

```
public static bool IsValidEmail(string email) {
    MailAddress address;
    try {
        address = new MailAddress(email);
        return (address.Address == email);
    } catch {
        return false;
    }
}

public static bool IsValidString(string str) {
    return (!String.IsNullOrEmpty(str) && !str.Contains(";"));
}

public static bool VerifyAccount(string firstName, string lastName, string verificationCode) {
    Queries conn = new Queries();

    //grab verification code from Database
    string verificationCodeFromDB = conn.getVerification(firstName, lastName);
    if (String.IsNullOrEmpty(verificationCodeFromDB)) {
        return false;
    }

    if (verificationCodeFromDB.Equals(verificationCode) &&
        (conn.getUser(firstName, lastName).IsVerifiableBy(SceneInstanceControl.User))) {
        //Update verification code in database
        conn.setVerified(firstName, lastName);

        conn.closeConenction();
        return true;
    }

    conn.closeConenction();
    return false;
}
```

Figure 4.3.4.2: Verification: Validity of strings, emails, account verification

```

public static User Login(string firstName, string lastName, string password) {
    //Grab password from Database
    Queries conn = new Queries();
    string passFromDB = conn.getPassword(firstName, lastName);
    if (String.IsNullOrEmpty(passFromDB)) {
        return null;
    }

    password = PasswordEncryption(password);

    if (passFromDB.Equals(password)) {
        Debug.Log("PASSWORD MATCHED!!!: " + passFromDB + " same as given " + password);
        //Get that user
        User user = conn.getUser(firstName, lastName);

        conn.closeConenction();
        return user;
    }
    conn.closeConenction();
    Debug.Log("PASSWORD: " + passFromDB + " didn't match the provided " + password);
    return null;
}

```

Figure 4.3.4.3: Verification: Login

In figure 4.3.4.3, the Login method passes through the necessary values firstName, lastName, and password. We then grab the password from the database based on the primary key (firstName, lastName) to check that it matches up correctly. The first if condition here checks to see if the string of the password from the database is null and if it is the method returns null. The second if condition checks to see if the password from the database is equal to the value of password. In the case that it is true, it adds to the debug log for debugging purposes, then gets the user from the getUser(firstName, lastName) method, proceeding to close the connection to the

database and returning user. More debug log is provided in the case the password provided does not match up, thus returning null again.

For our password encryption in Figure 4.3.4.4, we manipulated the individual characters of a string. We turned each character into an ascii number then multiplied and added two numbers: 97 and 1. We then forced this answer into a range of [32, 128] by finding the remainder

of 96 and increasing the value by 32. The divisor for the remainder makes sense because the difference between 128 and 32 is 96. Next is the `GenerateVerificationCode` method. The first thing to notice is that `n` is equal to 6. This sets the verification code to be 6 digits long. This is accomplished by taking numbers in the range [100000, 999999]. This is the same as $[10^5, 10^6-1]$. Thus, the formula for a number with `n` digits is defined by $[10^{n-1}, 10^n-1]$. Thus, this method returns a random number within that range. Lastly, is the `sendAuthenticationEmail` method. This creates a frame for an email to send to a verifier and

```
public static string PasswordEncryption(string password) {
    char[] charArr = password.ToCharArray();

    for(int i = 0; i < password.Length; i++) {
        charArr[i] = (char)((int)charArr[i]*97 + 1) % 96 + 32;
        if ((int)charArr[i] == 39 || (int)charArr[i] == 34) {
            charArr[i] = (char)((int)charArr[i]+1);
        }
    }
    return new string(charArr);
}

private static string GenerateVerificationCode() {
    //Want a n digit code
    int n = 6;
    int numNumberCode = randomNum.Next((int) Math.Pow(10, n-1), (int) Math.Pow(10, n)-1);
    return numNumberCode.ToString();
}

public static void sendAuthenticationEmail(string toEmail, string homeSchool,
    string firstName, string lastName, string newUserEmail, string verificationCode) {
    //Thanks to docs.microsoft.com
    string subject = "College Connections Verification Code";
    string body = "Hello " + homeSchool + " Advisor,\n\n" +
        "This user needs verification: " + firstName + " " + lastName +
        "\nTheir email is: " + newUserEmail +
        "\nThis is their verification code: " + verificationCode;

    EmailSender.SendEmailTo(toEmail, subject, body);
}
```

Figure 4.3.4.4: Verification: Encrypion, Verification, and Email Authentication

fills it with the appropriate information, as given. This method is called from the `Register` method. It then calls the `EmailSender`, which utilizes the SMTP.

5.0 Earned Value Analysis

Earned Value Analysis tracks a project's efficiency in terms of actual and budgeted man hours per task. Since our team did not 'clock in' when working on the project, many of the recorded man hours are estimates to the best of our ability. Note that if 4 people work together on the same thing for 3 hours, a total of 12 man hours would be recorded. Even so, many calculations were performed on the data from our scheduled and completed tasks.

To perform Earned Value Analysis, the Budget at Completion was calculated. Then, for each day, Budgeted Cost of Work Scheduled, Budgeted Cost of Work Performed, and Actual Cost of Work Performed were computed. Using these numbers, Percent Complete, Cost Variance, Scheduled Variance, Cost Performance Index, and Scheduled Performance Index were generated. On top of the variances and indices we looked at in class, we also calculated Accounting Variance, Cost Variance Index, and Schedule Variance Index. Note that the variance indices utilized the cost variance and schedule variance to center the data on zero. The formulas for these calculations were plugged into the excel sheet so that we could see daily updates(CPM, 2016), and is shown in Figure 5.1.

Date	11/10/2020	11/11/2020	11/12/2020	11/13/2020	11/14/2020	11/15/2020	11/16/2020	11/17/2020	11/18/2020	11/19/2020	11/20/2020
BCWS	0	8	24	24	24	24	27	27	27	27	47
BCWP	0	8	27	31	31	35	41	41	47	52	52
ACWP	0	8	26.5	30.5	30.5	34.5	38.5	38.5	43.5	49.5	49.5
BAC	178	178	178	178	178	178	178	178	178	178	178
Percent Complete	0	0.04494382022	0.1516853933	0.1741573034	0.1741573034	0.1966292135	0.2303370787	0.2303370787	0.2640449438	0.2921348315	0.2921348315
Expected PC	0	0.04494382022	0.1348314607	0.1348314607	0.1348314607	0.1348314607	0.1516853933	0.1516853933	0.1516853933	0.1516853933	0.2640449438
Percent Left	1	0.9550561798	0.8483146067	0.8258426966	0.8258426966	0.8033707865	0.7696629213	0.7696629213	0.7359550562	0.7078651685	0.7078651685
Expect PL	1	0.9550561798	0.8651685393	0.8651685393	0.8651685393	0.8651685393	0.8483146067	0.8483146067	0.8483146067	0.8483146067	0.7359550562
AV	0	0	-2.5	-6.5	-6.5	-10.5	-11.5	-11.5	-16.5	-22.5	-2.5
CV	0	0	0.5	0.5	0.5	0.5	2.5	2.5	3.5	2.5	2.5
SV	0	0	3	7	7	11	14	14	20	25	5
CPI	0	1	1.018867925	1.016393443	1.016393443	1.014492754	1.064935065	1.064935065	1.08045977	1.050505051	1.050505051
CVI	0	0	0.01851851852	0.01612903226	0.01612903226	0.01428571429	0.06097560976	0.06097560976	0.07446808511	0.04807692308	0.04807692308
SPI	0	1	1.125	1.291666667	1.291666667	1.458333333	1.518518519	1.518518519	1.740740741	1.925925926	1.106382979
SVI	0	0	0.125	0.2916666667	0.2916666667	0.4583333333	0.5185185185	0.5185185185	0.7407407407	0.9259259259	0.1063829787

Figure 5.1: Earned Value spreadsheet data page

These values allowed project announcements about our productivity to include real statistics from how we were doing. It proved as insight to see when we would get work done, and how much we'd get done as deadlines approached. Earned Value Analysis is a very useful tool for any large project.

The following charts utilized the calculated values so our team could better comprehend the data. In terms of the burndown chart, as shown in Figure 5.2, it shows that our percent uncompleted fluxuated above and below the scheduled percent uncompleted.

BurnDown Chart

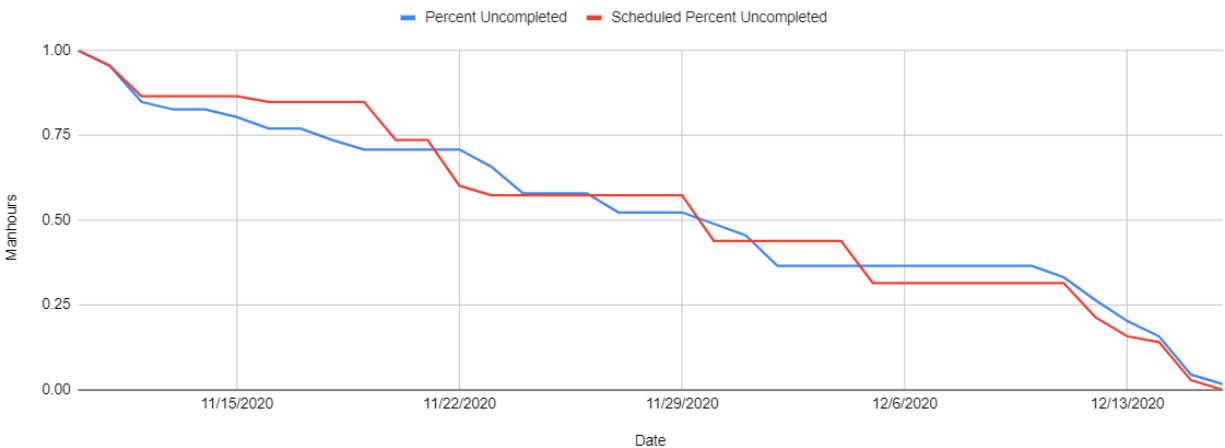


Figure 5.2: BurnDown Chart

Even with little to no knowledge of Earned Value Analysis, any member can understand the burndown chart when put in terms of “the blue is us, the red is where we should be”. Looking more closely, our percent uncompleted was higher than the scheduled percent uncompleted whenever there was a large increase in completed work, implying we were behind. Thus, this came as a warning to our team, as the largest drop came at the end of the project’s timeline. Our schedule variance index, as shown with Figure 5.3, told an identical story to our burndown chart and schedule variance, and understandably so. They all compare the current amount of completed work to the scheduled completed work. The schedule variance index differs from the schedule variance by compressing the values around the x-axis. For further analysis. These repetitive charts will be omitted, but the similarities will still be explained.

Schedule Variance Index: Positive means you earned more than scheduled

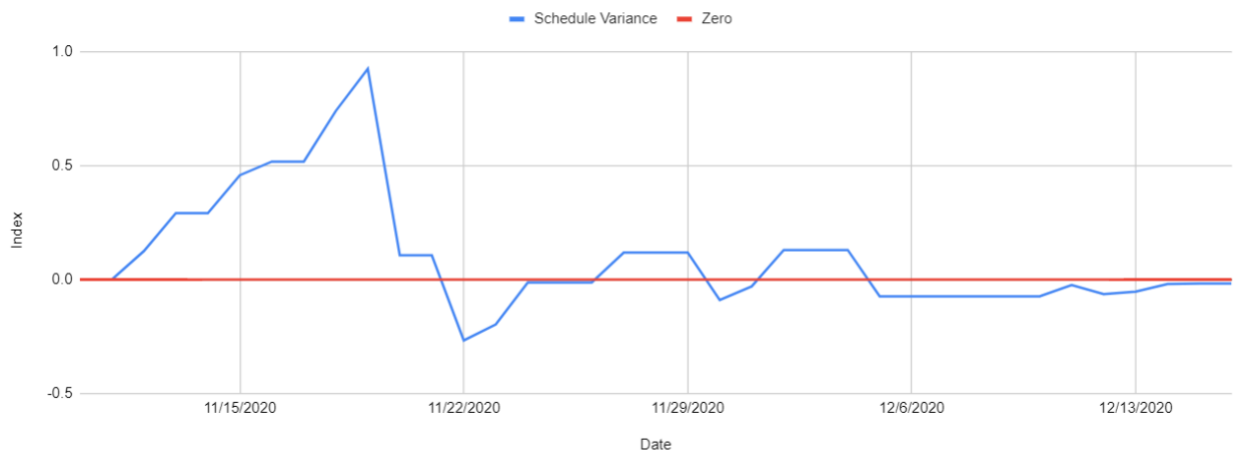


Figure 5.3: Schedule Variance Index

Our cost variance index, as shown in Figure 5.4, soared above zero throughout the project, showing that we were spending less time to complete tasks than budgeted. Near the end of the project, this value dipped significantly, but never completely reached zero. The cost

variance index (CVI) and cost value (CV) have a similar relationship to the schedule variance index and the schedule variance; the index compresses values vertically.

Cost Variance Index: Positive means Spending Less than completed



Figure 5.4: Cost Variance Index

The cost variance index also tells a similar story as the cost variance and cost performance index (CPI), and understandably. They all compare the budgeted cost of performed work to the actual cost of performed work in such a way so that positive numbers are a result of there being more budgeted cost than actual cost. As the majority of the group pushed to get in their requirements at the end, the budgeted and actual approached one another.

Accounting Variance: Positive means spending under budget



Figure 5.5: Accounting Variance

Lastly is the Accounting Variance, as shown in Figure 5.5. It compares how much work was actually performed to the budgeted amount of work scheduled. Thus, it deals with how good of a prediction was made by the creators of the estimates. At examination, one can tell that whenever the variance was more than 20 man hours away from a perfect estimation, 0, the

accounting variances shot towards 0. However, in the end, the variance did get quite close to 0. Overall, our group's evaluation of the estimates would be that they were too lenient at first, and slightly too harsh at the end.

Therefore, our team was way ahead of schedule in terms of work completed at first (SV and SVI) but as the project continued, we bounced between ahead and behind. Near the end, we reached a difference between scheduled and actual percent completed near 0, showing that as a team, we just finished all our planned work. This begs to question why we fell behind schedule in the middle of the project, and what methods could be used to remedy this. Secondly, the team seemed to be way ahead of schedule in terms of differences in man hours for tasks throughout the project, as shown with CPI, CV, and CVI. However, this dipped significantly as the project, presentation, and report came together. One reasoning for this is that while many seemed to have components completed earlier and quicker than expected, problems arose as the project continued, such as implementation of components, that required extra focus and time. Lastly, to evaluate the estimations, the Accounting Variance (AV) was used. AV depicts underestimation at the start of the project, and overestimation at the end of the project in terms of expected hours.

6.0 Testing

6.1 Unit Testing

Unit testing is a specialized way of testing your application. What is unique about unit testing compared to other types of testing is that unit testing involves testing each component of your system separately. The advantage that this gives developers is that it allows them to verify that each component of your application is functional prior to testing the entire system. When testing an entire system errors may arise from an unknown origin. The goal of unit testing is to eliminate the possibility of errors existing within the individual components of the system and isolating them to being an issue with the integration of the different components. Unit testing is essentially stress testing the different components of a system. The unit tester will attempt to break the component in anticipation of anything that a user may attempt to do, albeit in stupidity or in an attempt to compromise the system. There are three unit testing techniques, black box testing, white box testing, and gray box testing. Black box testing focuses on the input and output of data, what happens in the middle is not essential information within the constraints of this test. As a result, black box testing is an effective way to test specific features in your system. White box testing focuses on testing the code that exists within the previously mentioned black box. This type of testing is used to try and see the flexibility of the internal logic and the code. Gray box testing is a combination of black box input output based testing and the white box internal logic testing. However, in gray box testing the test is conducted from the point of view of a user rather than a developer.

6.1.1 Chat System

Our chat system is very linear and easy to implement in all our chat rooms. We made sure that the username being displayed is the same as the login name. Over that, we also made sure that the response system was always accurate. For this, we made it so that the user's message is converted to all lower cases, and then compared to a predefined string. After the user's message is compared to the predefined message, the admin will respond to the user. We added this feature in the hopes that once it is expanded, the user can ask generic questions about the specific college and an automated machine can respond to the question if no one is around. Other than that, the chat system comes in a bundle, so that it can be copy and pasted in other chat rooms.

This will allow the developers to customize the chat system in each school separately to add diversity between them all.

6.1.2 Security and Authentication

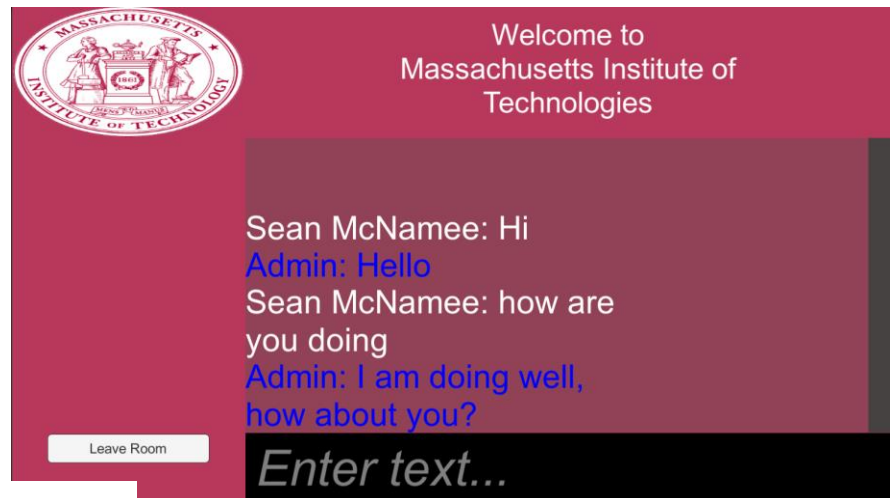
To ensure security throughout the application, the Authenticator's IsValidString and IsValidEmail methods were used. As well, checks with the status of sql queries were used to decide what course of action to take within the application.

First Name:

Last Name:

Password:

Invalid Credentials, Try again



As portrayed in figure 6.1.2.1, for completely invalid credentials the Authenticator does get any existing values in the database, therefore the credentials are invalid while trying to log in.

Here there is invalid credentials again simply because the password does not match with the password from the database, shown in figure 6.1.2.2.

First Name:

Last Name:

Password:

Invalid Credentials, Try again

First Name:

Last Name:

Password:

Not yet verified. Contact your advisor

FIGURE 6.1.2.3 User not yet verified

In the figure 6.1.2.3, all the information is checked out correctly from the database, but the user has not been verified yet. If they are to contact their advisor it could possibly boost the speed of getting verified.

First Name:

Last Name:

Password:

Email:

Advisor:

Advisor's school:

☐ College?

Something isn't filled out correctly

In figure 6.1.2.4, in the situation where the email is not put in the correct format of *****@*****.*** it will give the error shown above “Something isn’t filled out correctly”.

Figure 6.1.2.4: Invalid email format

First Name:

Last Name:

Password:

Email:

CollegeModerator:

Harvard:

Something isn't filled out correctly

Figure 6.1.2.5: Failed SQL Injection

When a user includes a semicolon in any of the inputs, the system detects that as an SQL injection attack and displays “Something isn’t filled out correctly”, as depicted in Figure 6.1.2.5. This is considered a Penetration Test. This prevents malicious users from creating verified developer accounts within the system without permission.

firstName	lastName	password	type	isVerified
Matthew	Nielebock	0, #:	Advisor	Verified
Max	Piff	#4%(HighSchooler	Verified
Professor	Akhtar	##\$%	HighSchooler	Verified
Sean	McNamee	\$4\$*e&7q#44803%	Developer	Verified
Seen	Macc	/0*\$&	Advisor	Verified
Shawn	Meek	RSTU	Advisor	123456

First Name:

Last Name:

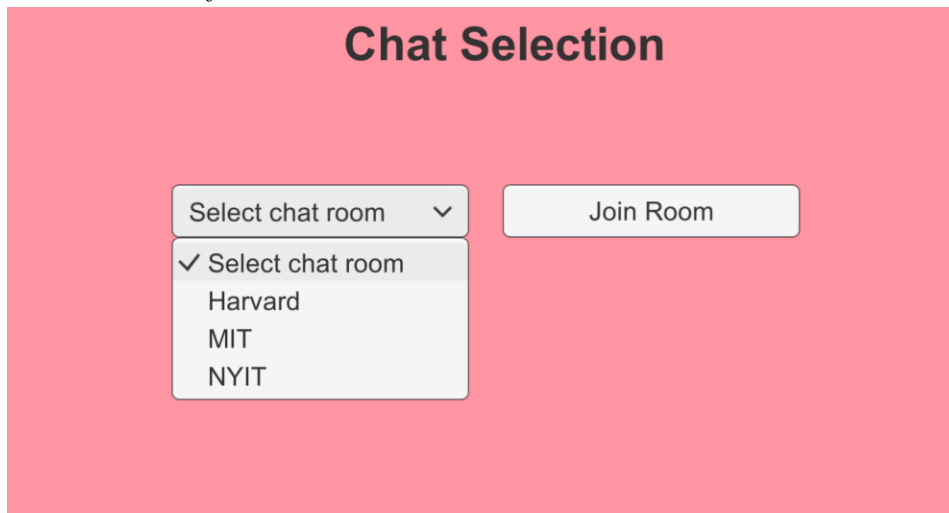
Verification Code:

Invalid Credentials, Try again

Figure 6.1.2.6: Failed Verification of Advisor by Advisor ‘Seen’ ‘Macc’

In figure 6.1.2.6, an advisor tries to verify another advisor which is not allowed to happen. Even though the advisor has inputted the correct verification code for “Shawn Meek”, the person here trying to verify this user is also an advisor. Advisors can only verify high schoolers and moderators; Advisors themselves must be verified by developers.

6.1.3 User Interface



In figure 6.1.3.1 the user has selected the school drop down menu and would like to select a chat room to enter. When the multiple rooms were first being implemented the only one that worked was the NYIT chat room. No matter what school you selected in the drop down menu it would always take you to NYIT. Now it will take the user to the correct school.

In figure 6.1.3.2 the users are able to input messages into the text field and by pressing enter can send the message into the chat room. Once the page has filled with messages the scroll bar becomes a crucial part of the page. When there are not enough messages the scroll bar is there, but does not function. With the implementation of the database it allows for users to appear as their actual name. During the early stages of testing there were no responses from an admin, and the only user that was in the system was named Mohammad. It now accurately shows the user's name. As stated in the previous figure the drop down menu had issues connecting to the correct rooms, but if you started in an unreachable chat room the leave room button worked

and would take the user back to the main menu, and still does

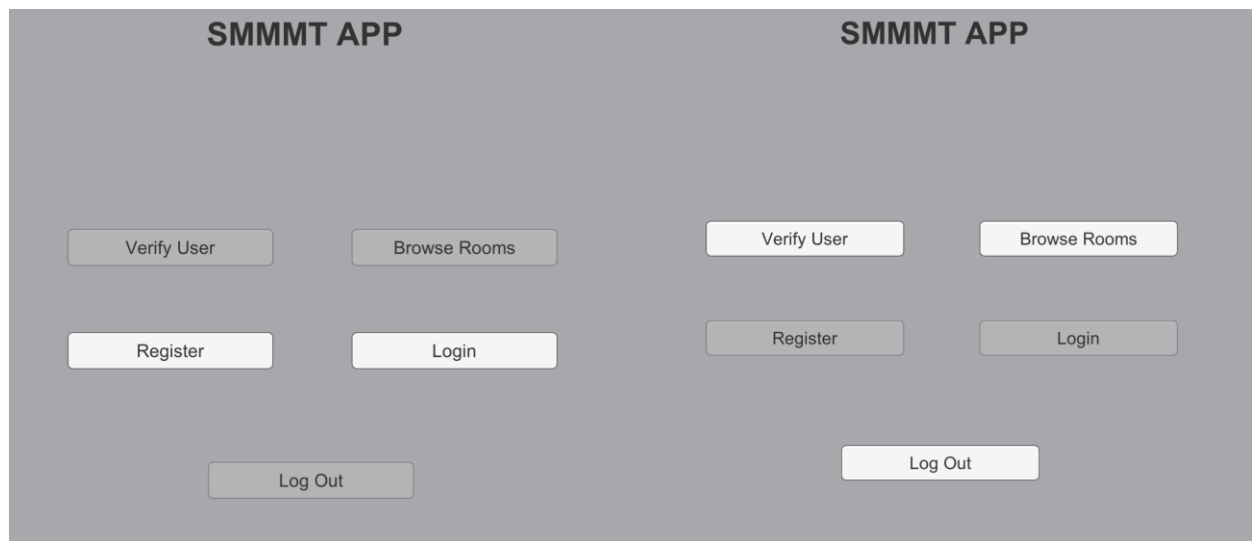


Figure 6.1.3.3 depending

In Figure 6.1.3.3 the entry conditions for the application vary depending on a couple of things. The first dependency is, is the user logged in? If they aren't there are only two options, Register and Login. Once the user is logged in is when the second dependency is once the user is logged in what level is their account? Advisors and Developers are able to verify users who are registering their accounts. But any person who is logged in will be able to select the chat rooms they have selected. Additionally any user who is logged in will be able to logout from this screen, non logged in users will not be able to use this function since nothing would come from this.

6.1.4 Database Control

school	advisorFirstName	advisorLastName	isCollege
Bay Shore HS	Seen	Macc	0
Connetquot HS	Seen	Macc	0
Half Hollow Hills HS	Seen	Macc	0
Harvard	Matthew	Nielebock	1
MIT	Matthew	Nielebock	1
Newfield HS	Shawn	Meek	0
NYIT	Matthew	Nielebock	1
Smithtown HS	Seen	Macc	0

Figure 6.1.4.1: Selection of all schools

In Figure 6.1.4.1 we are executing a basic query which pulls all of the data in the school table. The purpose of this query is to check that the data which currently resides within the school table is accurate to what is expected to be in there. This query is an important check when looking at what schools you have entered into your system, their associated advisors and whether or not it is a college.

firstName	lastName	password	type	isVerified	email
Matthew	Nielebock	0, #;	Advisor	Verified	mnielebock8@gmail.com
Max	Piff	#4%(HighSchooler	Verified	mpiff@nyit.edu
Professor	Akhtar	##\$%	HighSchooler	Verified	makhtar@nyit.edu
Sean	McNamee	\$4\$*e&7q#44803%	Developer	Verified	smcnamee@nyit.edu
Seen	Macc	/0*\$&	Advisor	Verified	sean4mc5@gmail.com
Shawn	Meek	RSTU	Advisor	Verified	seanmcnamee.45@gmail.com
Tim	Cameron	5&45	Developer	Verified	tcameron@nyit.edu

Figure 6.1.4.2: Selection of all users.

In Figure 6.1.4.2 we are executing a query to select all of the users that are entered into the user table. Executing this query will allow developers to have a list of all of the users who have access to the different roles within the application. This is important because it can be used to make sure no unauthorized users have gained access to the system, or upgraded their privileges to a type that does not fit the type that they are supposed to have.

firstName	isVerified
Tim	Verified

Figure 6.1.4.3: Selection of Tim Cameron's name and isVerified status

In Figure 6.1.4.3 the query we are executing is to determine if a user by the name of “Tim Cameron” is a verified user. Determining if a user is verified is an integral part of the application due to the fact that only verified users can gain access to the chat rooms and the other abilities granted by their ‘userType’.

In Figure 6.1.4.4 we are looking at the users table once again; however, this table will be updated to show the addition of the newest advisor to MIT, Bruce Wayne. Testing the ability to add users to the database is an essential test to run due to the fact that the goal of any application is to grow its user base.

firstName	lastName	password	type	isVerified	email
Bruce	Wayne	n0tB@t.man	Advisor	Verified	bwayne@MIT.edu
Matthew	Nielebock	0,#:.	Advisor	Verified	mnielebock8@gmail.com
Max	Piff	#4%(HighSchooler	Verified	mpiff@nyit.edu
Professor	Akhtar	##\$%	HighSchooler	Verified	makhtar@nyit.edu
Sean	McNamee	\$4\$*e&7q#44803%	Developer	154532	smcnamee@nyit.edu
Seen	Macc	/0*\$&	Advisor	Verified	sean4mc5@gmail.com
Shawn	Meek	RSTU	Advisor	Verified	seanmcnamee.45@gmail.com
Tim	Cameron	5&45	Developer	Verified	tcameron@nyit.edu

Figure 6.1.4. 4 Selection of users after an insert of Bruce Wayne.

7.0 Conclusion

As a team we gained a lot of new experience whether that be through creating our software or efficiently working as a team. Using unity was originally a software that everyone was not familiar with nor even heard of but we all agreed to use it. Even though it was bound to set back our timeline, we agreed since mohammad preached Unity. Everyone as a whole worked smoothly even though it was hard to find good chunks of time where we could all work together. One major part that we are glad professor Akhtar mentioned is implementing a security component into our project. Originally we had barely any thoughts about worrying about it, but after she mentioned it we came up with the idea of making a verification system. We harped on the fact that our system would match students up with one hundred percent verified users. In the case that we were to follow through with our system, there are not many well-known applications that bring the same features as we do here with verification. To generalize the project as a whole we all are glad to have taken some much needed steps out of our comfort zones to genuinely push ourselves.

7.1 Limitations

Obviously our application does a decent job of doing what we have proposed, but we had some limitations which held us back from creating and adding some features. The most prominent one would be the time constraints given that this was assigned with about a month to fully put together everything. Another limitation was the simple fact that every student in the group has other projects and callings that they must attend to, whether it be exams, projects, or work. This sometimes made it difficult if we needed to meet up as a group altogether to get complete a certain aspect of the project.

There were also limitations outside of project requirements and student complications. The spawn of these issues was because we all tried to take on a part of the project that we have not had much experience on in order to help us improve ourselves in that field. For instance, Tim took on the database part of the application even though he is still in the middle of taking

database as a class which he had little to no knowledge in before. We did this throughout the whole team not only with Tim and we used each other's knowledge from each aspect to help one another in the case that they got stuck. Mohammad worked on the chat system since he never created something like it which was a deciding factor in why we only have the chatting on a local system. Sean and Matt did the security/authentication of the system and used C# since our change to using Unity forced us to adapt accordingly. Max wanted to take on the user interface because he had never worked on anything similar to it and decided to further his experience with it. Overall this led to some parts taking longer than we intended and hoped to, thus it took away from time which could have been spent elsewhere.

7.2 Future Enhancements/Recommendations

With more time, our team would put a larger focus on the user interface and chatting system. Even so, we would still want to increase the functionality of the security.

Put harshly, the UI is dull. A spice to the current layout would increase user enjoyment significantly. Perhaps, a screen dedicated to logging in could be added so that the main screen does not feel as cluttered. This would veer our system inline with many of the existing chatting systems of today. Another addition for the UI could be the existence of a logout button on any screen, rather than just the main menu. This would definitely increase ease of use. Adding a compatibility mode for the UI would also be a great addition, currently it is only designed for desktop use, but as we know a lot of people use their phones more than computers. Creating mobile apps would allow for an even larger user base for this application.

For the chatting system, our goal was to have the ability for communications between multiple instances of the system. However, our current system only permits a single logged in user from chatting at once, receiving an automatically generated response at the detection of certain messages. A future goal would be implementation of this socket-based communication, so that connections can truly be made between high schoolers and college mentors. Another feature that we could add is direct messaging, where someone you chat with can be chatted to directly, outside of the college chat room. This could boost the ability of high schoolers to make deeper connections with specific members of colleges.

In terms of the Verification, with more time, we would add confirming email, security questions, and forgotten password. This would bring our application to the point where it is on par with user features as many existing applications. To boost security, we could utilize some type of private key that decrypts user information so that viewing the database does not show the username or email.

In terms of the overall application, we wish to allow highschoolers to be able to add and remove colleges from their available chat schools after registration. Perhaps the “Verify” button could be replaced with a “Add Chat Rooms” button on the main page.

7.3 Team Members

As with any team, a difference in experience and workload can shift some to take on more work than others. In our team, I believe that everyone put as much time and effort into their components as possible based on their experience and workload. The following briefing outlines the contributions of our team and our previous technical experience:

Sean:

Experience in Basic, Java, Javascript, Python, SQL, and minimal C/C++/C#. For the presentation, he worked on the proposed system and timeline. Next, deliverable 2. He Helped Matt, Tim, and Mohammad on context diagrams, and worked with Matt, Tim, and Mohammad on the DFDs. He did the table schema and ERD with Tim, and did the Swimlane, use-case, and state diagrams with Max. For the report, he did the Proposed System, Timeline, Activity List, Earned Value Analysis, and State Diagram. He worked with Max on the Swimlane. He worked with Tim on Database Design, and worked with Matt on Privilege/Access level, Security, Privilege Level, the Security Example, and Security Testing. For the project, he did the MySQL Connection with Tim and worked with Tim on the Queries. He worked with Matt on Registration, Login, Verification, the User class, Security Implementation with UI with Matt Worked with Max on UI

Mohammad:

Experience in Unity, C#, and C++. For the proposal, he worked on Tools and Technology. For the diagrams, he gave input on the Context Diagram and DFDs. For the report, he did the Purpose, Objective/Goals/Scope, Technologies & Tools, Chat System Explanation, Chat System Testing, and introduced Functionality and implementation. He also worked with Max on the UI System Explanation. For the report, he did the Messages, and Chat Auto-Response System. He worked with Sean on Understanding Unity and Connecting UI to the transcript sender. He also helped with Unity-MySQL Connection.

Matt:

Experience in Java, C++, C#, and a bit of SQL. For the proposal, he did the SE Model. For deliverable 2, he did the Context with Tim and Mohammad. He also worked with Sean, Tim, Mohammad on the DFD. For the report, he did the Proposed System advantages, SE Model, Context, DFD, Limitations, and Conclusion. He as well worked with Sean on Privilege/Access level, Security, Privilege Level, Security Example, and Security Testing. For the project, he worked with Sean on Registration, Login, Verification, and the User class. He as well helped Sean with Security Implementation with UI.

Max:

Experience in Java, C++, web development. For the proposal, he worked on Users. For the diagrams, he did the Swimlane, Use-Case, and State diagrams with Sean. He also gave input

on DFD. For the report, he worked on Users, Motivation, Use Case, and File structure. He also worked with Sean on the Swimlane and Mohammad on UI System Explanation. Lastly, he worked with Sean and Mohammad on the UI.

Tim:

Experience in Java and basic web development. For the presentation, he worked on existing systems. He worked on the table schema and ERD with Sean. For the proposal he worked on the existing systems. For the diagrams he helped create the table schema and ERD with Sean. In the report, he finalized the existing systems, changes to proposed system, functional/non-functional requirements, features, unit testing explanation, Database Unit Testing, and worked with Sean on the Database Design. For the project, he worked with Sean on the MySQL Connection and Queries.

7.3.1 Sean McNamee

As the team leader, this project has given me major insights on the type of collaboration I will face when in the field as a software developer. Unless a team is ready and willing to take initiative to work together on a certain component, that component should be split up and its smaller tasks should be delegated. As well, I've learned that in order to keep members efficient, meetings should be smaller but more frequent, with small goals stressed. In terms of the technologies of the project, I've definitely learned enough to connect backend functionality to the frontend user interface provided by Unity. I've definitely seen things that I believe are cool, and other aspects that I believe are unoptimized. I've experience C# in a deeper way and have seen the features it has to offer in comparison to Java. In terms of security, I've learned the in-depth structure required to ensure system users are identified properly. Overall, I believe this project was a great learning experience and has paved the way for future collaborative projects.

7.3.2 Mohammad Ishtiaq

After the completion of this project, I have walked away being more knowledgeable about the software engineering process of building a process in the real world. This was the first time I was given the opportunity to work in a group which was thoroughly thought out and planned. I learned skills ranging from and not limited to MySQL, Unity, C#, etc. My main takeaways were that after working on the DFDs / ERDs, I had a better understanding of what these graphs were looking for and how to make them more detailed. Furthermore, this was the first time I worked with MySQL, and SMTP, and through the help of the group, I learned how to connect databases to unity and how to create a simple login system. This will come in handy for my future game development project I have coming this winter. I would like to thank my team for their support and especially Sean for keeping the team updated and on track.

7.3.3 Matthew Nielebock

After completing this project, I have learned valuable skills which were mainly security-based and focusing on verification since this was the aspect of the project I worked on. Additionally, I learned C# during this project and how to use SMTP to send emails for verification/chat transcripts. Although I did not personally put much of my effort into the unity part since it was used through Mohammad's and Max's work, I still witnessed how to use it and could replicate it if necessary. The most important outcome though was understanding how crucial verification was to our system because it helps it stand out from other existing systems. As far as any challenges I came across personally, C# was an easy transition and the hardest thing was connecting the whole frontend and backend together in Unity. Since our part had many direct correlations with Timothy, it was definitely a challenge to help him put together the queries that we needed to connect with Sean and I's part of the Security and Authentication.

7.3.4 Max Piff

With the completion of this project I have learned valuable skills related to software engineering. I had already worked with creating user interfaces before, but I had created them with web design languages like; HTML, PHP, Javascript, and CSS. However I have never used Unity before this project, so I had to learn how to create and implement a user interface that would actually link page to page. Additionally C# is not a programming language that is familiar to me, and since our project was programmed in this language Sean helped me to work towards creating working buttons and drop down menus. I am very glad that I had the ability to try out the Unity platform, and I hope to use it again in the future. I am also very glad for the amazing hard work that my team has worked on.

7.3.5 Timothy Cameron

After completing this project I came away with new skills and greatly refined several other skills with respect to software development and working in agile with a team. Prior to embarking on this project I had minimal experience with working with databases and MySQL but through the help of my team and some research on the side I was able to become a contributing member of the team by working on the database. I learned better ways to produce queries and a better way to go about thinking about the logic regarding writing new queries, and bridging the gap between the application and the database through the usage of C#. None of this would have been possible without Sean taking time to assist me through the process when I would begin to struggle with aspects of my work. The overall experience was a positive one due to such a great atmosphere within the group and the willingness of everyone to jump on and work together. I did face some challenges when moving along with this project and those would be issues with motivation in addition to the actual challenge of learning something new and stepping out of my comfort zone; but I think this project has helped me conquer pieces of both of those challenges when looking back on the experience.

8.0 References

- CPM. (2016). *Earned value cheat sheet*. CPM Solutions Ltd. Retrieved December 9, 2020 from https://www.p6consulting.ca/wp-content/uploads/2016/01/Earned_Value_Cheatsheet.pdf
- C#. (2020). C# documentation. Retrieved December 14, 2020 from <https://docs.microsoft.com/en-us/dotnet/csharp/>
- Google. (2020). *Gmail sending limits in Google Workspace*. Retrieved December 12, 2020 from <https://support.google.com/a/answer/166852?hl=en>
- Unity. (December 7, 2020). Unity user manual (2019.4 LTS). Retrieved December 13, 2020 from <https://docs.unity3d.com/Manual/index.html>