DEPARTMENT OF
COMPUTER SCIENCE

Spring 2022

Senior Project Report

Department of Computer Science

California State University, Dominguez Hills

Chatting App

Prepared By

Ricardo Mondragon Rodriguez

Johnny Castillo

In
Partial Fulfillment of the requirements

For
Senior Design – CSC 492

Department of Computer Science
California State University, Dominguez Hills
Spring 2022

Committee Members/Approval

Amlan Chatterjee	_____	_____
Faculty Advisor	***Signature***	***Date***
_____	_____	_____
Committee Member	***Signature***	***Date***
_____	_____	_____
Committee Member	***Signature***	***Date***
Mohsen Beheshti	_____	_____
Department Chair	***Signature***	***Date***

ABSTRACT

In a world where communication is of utmost importance, people spend a majority of their life on the web communicating with others. The process of communicating helps facilitate sharing information and knowledge with others, as well as develop relationships. We've been doing it ever since we started to make noise with our vocal cords. It is a sense of connection for why we do it in the first place. Would we be able to survive with the inability to communicate with one another? No, in fact it would be impossible to live a life we currently do if communication wasn't available. However, the leap forward that brought communication to the new frontier has to be technological advancements. In 1831, the electric telegraph was created that allowed people to communicate with each other over long distances. Then in 1973, Motorola created the first mobile phone that led to many predecessors to follow suit. It was primarily created for the enterprise market and then it was brought towards the commercial market for people to purchase. However the most important advancement towards communication has to be the internet. In the early 80s the internet was invented. It made communication easier and achieved longer distances of communication. Apart from social communication enterprises were also able to market the internet as well. They were able to better advertise their product because of the internet. Overall communication such as the internet has allowed people across the world to communicate with each other but as well for people create a sense of community in an endless world.

Table of Contents

List of Figures

1 Introduction

We decided to create a Chatting Web App that would allow users to communicate with one another. The reason for this is a chatting website like WhatsApp or Discord is well known to allow a sense of community. Also allow potential Dominguez Hill's students, but not limited to communicate with one another. However, the main reason we decided to create this chatting website was to push ourselves out of our comfort zone as well as help create our portfolio for future job interviews. This chatting website will give us back-end experience as well as front-end development. That being said, the functionality of our website will allow users to experience a homepage where they are able to make an account using a username of their choosing and setting up a password. We then would request the user to once again login. If the username or password was incorrect it would redirect the user that their credentials are wrong. We would like to add future account information from profile image to potential username that user could change in the future. After the user has login he will be brought to a room selection interface. The user can input which room they would like to join. It can be an integer or even string. Finally they will be brought to the chatting interface and we then can now send messages to others in the specified room. Part of the chatting interface are home buttons from which they can return home or even sign out of their account. We would like to add the future ability to create a private room but that required some sort of finesse over the channels being used. We would also like to add the ability to direct message users, that would require a friend list. However as of now we are keeping it pretty simple and adding new features if time allows.

2 Background

To help construct the project and put it all together, we decided to use Python. Python has web based frameworks such as Flask which allowed us to easily set up our app via a web browser tab. Python also has support for HTML and CSS which would allow us to change the front-end display within Python. Flask is used as both the front-end and back-end of our chatting app consisting of Python, HTML, CSS, and JavaScript. Flask serves as the back-end since it is able to route to different links via app.routes that we create for specific tasks within the project. Sockets in Python would serve as our host for communication within the app for users to connect with one another and start a conversation. SocketIO actually allowed for easier implementation of ideas we had in mind with the use of JavaScript.

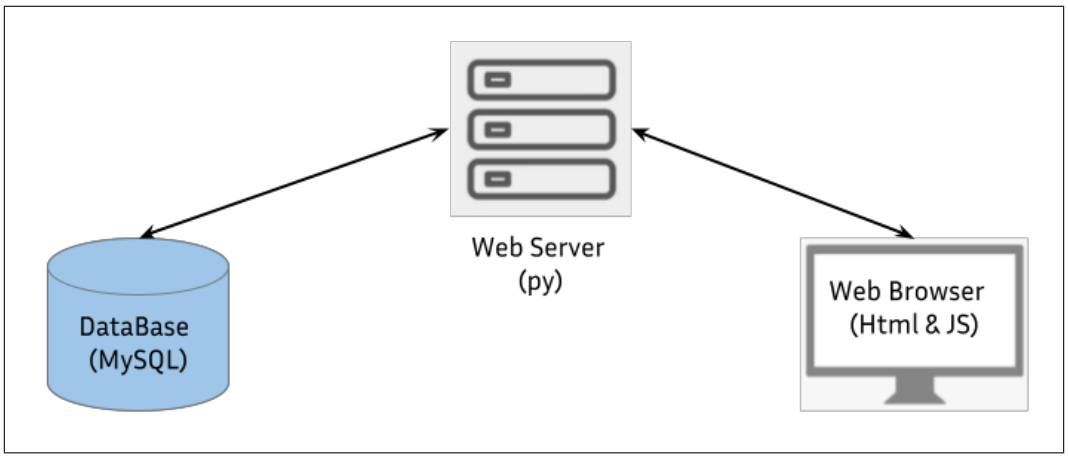


Figure 1: Building Block Diagram

3 Tools Used

This section will primarily focus on the tools we used to commence our project

3.1 GitLab

A GitLab repository was created in order to allow file sharing, multi development and insure that or program was in one central place. As well as of allowing to have a permanent backup place for in cases of file lost. GitLab also allowed us to push code onto branches along with a message where we would write any changes that we had made to the files for that push onto the branch.

3.1.1 Implementation

We created one main branch and only created additional branches when newer implementation could have potential catastrophes. The branches we created were used for testing new features so that the main branch would contain a fully functioning version of our chatting application. Having separate branches for development and testing purposes prevented us from running into merge conflicts as we had our own separate branches to push code onto and overall prevented any errors from occurring.

3.1.2 Challenges

We did face one challenge but it was easily resolved. I was unable to push on to the main branch because it was protected, but Johnny was just able to change the main branch and allow me access. Another challenge we had was that when we would commit, it would trying pushing multiple files that we weren't using onto the branch. Other than that, we had no issues.

4 Development

This section will primarily focus on the steps we need to take in order to implement our Chatting App.

4.1 Setup Web Browser

4.1.1 Implementation

Initially, we first setup a virtual environment to keep all packages and downloads that we were using locally within the project. Once we set that up, we began installing all the packages we would be using such as Flask, Flask SQLAlchemy, Flask SocketIO, Flask Session, etc. After doing so, we began to setup our app by creating a Flask object so that we could run the script and it would allow us to open it up in a web browser. All requirements for the project to run are located in the GitLab repository in the requirements.txt file.

After setting up Flask, we began to create templates for what each page would contain. Some of the pages we created were a homepage (home.html), login page (login.html), logout page (logout.html), a sign up page (signup.html), a room page (room.html), and our chatting room area (chat.html). Within each template html file, we added html code for what each page was responsible for displaying to the users screen. Figure 2 is a example of how home.html.

home.html 559 bytes

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chatting Application</title>
    <link rel="stylesheet" type="text/css" href="/static/main.css">
</head>
<body style="background-color: #5555dd">
<h1 style="color:black;"><center>WELCOME TO THE CHATTING APPLICATION</center></h1>
<ul class="homeNavBar">
    <li><a href="/login"> Sign in</a></li>
    <li><a href="/signup"> Sign up</a></li>
    <li><a href="/logout"> Sign out</a></li>
</ul>
</body>
</html>
```

Figure 2: Home.HTML

Flask uses the render_template() function to render the specified HTML file to be displayed for a given app.route(). Using Figure 2, render_template('home.html') would be the first template to be displayed upon opening the app. An example of how it looks in Python is shown below. In Figure 3, @app.route('/') is the first URL that Flask runs when the app is opened. Each route function that we add will have their own URL that is ran. For example, for our login page, Flask runs @app.route('/login') and our Python function defined for that URL runs and renders the 'login.html' template.

```
@app.route('/')
def home():
    return render_template('home.html')
```

Figure 3: Flask-Route

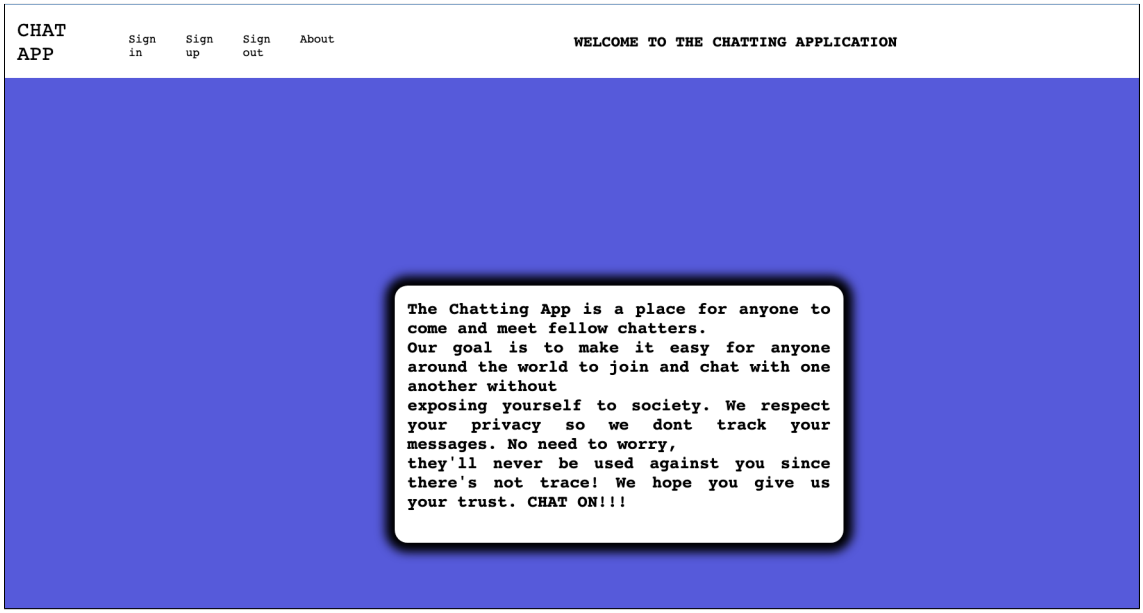


Figure 4: App Home Screen

Figure 4 is the first thing that the user is presented with upon arrival to the app URL. We have a navigation bar that has redirect links to the sign in page, sign up, and logout. The user is able to navigate to different parts of the app via direct URL input in the browser or through links displayed on the page. The redirects for each link in the navigation bar can be seen in Figure 2. The href shown in Figure 2 are used for Flask to know what function to call. For example, href='/login' calls the app.route('/login') URL which then runs the Python code below it. An example can be seen in Figure 5.

```
@app.route('/login', methods=['POST', 'GET'])
def login():
    '''
    :return: html templates based on the task
    '''
    if request.method == 'POST': # if user submits a login check if it exists
        username = request.form.get('username')
        password = request.form.get('password')

        users = Users.query.filter_by(username=username).first() # get all the usernames that are

        # enocode both since we encoded when storing in db. will compare bytes
        if users and bcrypt.checkpw(password.encode('utf-8'), users.password.encode('utf-8')): # p
            session['username'] = username
            print("sessions ", session['username'])
            flash('Login successful!')
            return redirect(url_for('room'))
        else:
            flash('Username or password are incorrect')
            return render_template('login.html')
    else:
        if 'username' in session: # user is still signed in, no need to re-sign in
            return redirect('/room')
        return render_template('login.html')
```

Figure 5: Flask routing via login page

Figure 5 also shows how we handle a user logging in. If a user enters both a username and password, we get it from the forms we created using HTML, then grab the username from the database. If the user exists, then we check the entered password and compare it to the hashed password that is stored in the database. If the credentials entered match to those in the database, then we grant them access to the chatting room area where they can begin conversations. If the username does not exist or the password is incorrect, we notify the user trying to login and have them either retry or direct them to the sign up page.

We used sessions to keep track of who was logged in so that if you went to home and clicked sign in, you wouldn't need to sign in again since you did not log out. We only made a user sign in if they were no longer in the session which happens when the user logs out.

4.2 Database

The Database that we have created and used was done in MySQL. The purpose of the Database for the chatting app is to store the users credentials such as username and password along with date and time the account was created. The database that stores this content is

titled "logins" where it contains a table titled "Users" containing the credentials.

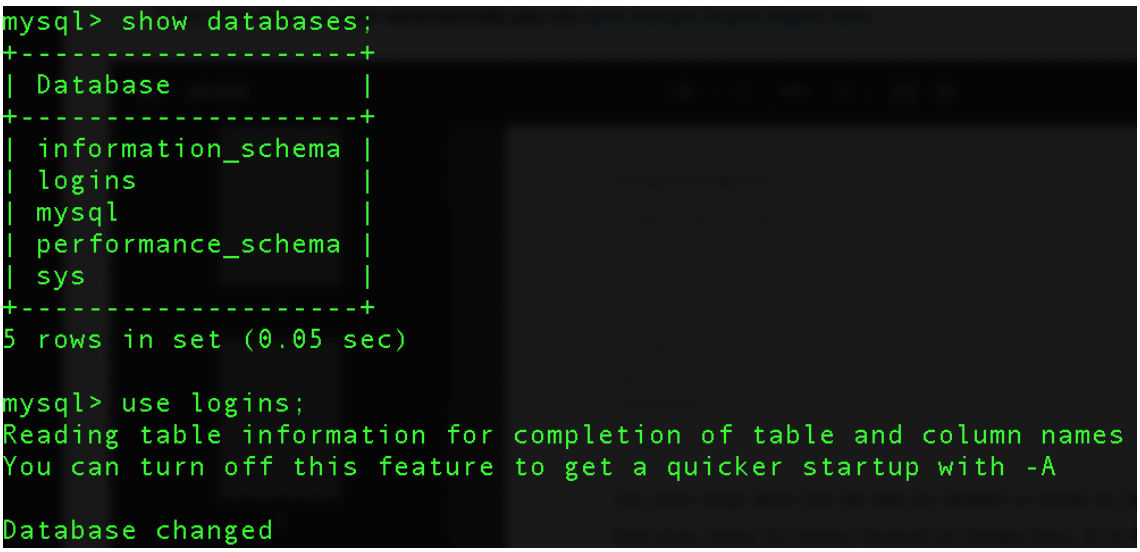


Figure 6: Database

The above image shows that our database actually exists within our local MySQL server.

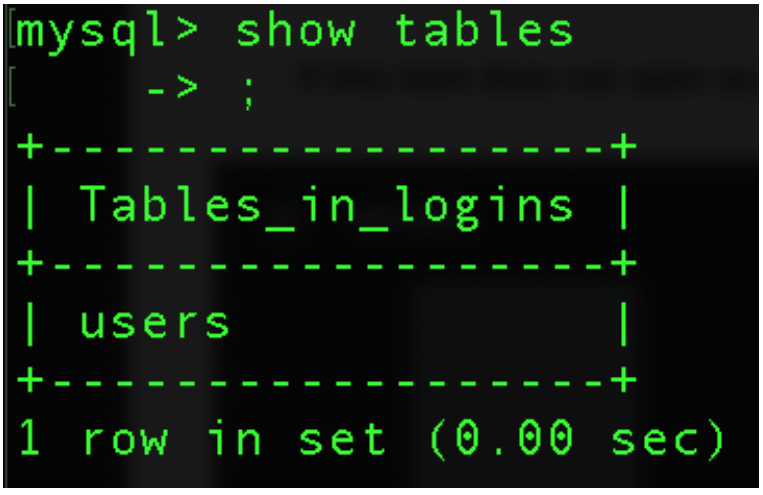
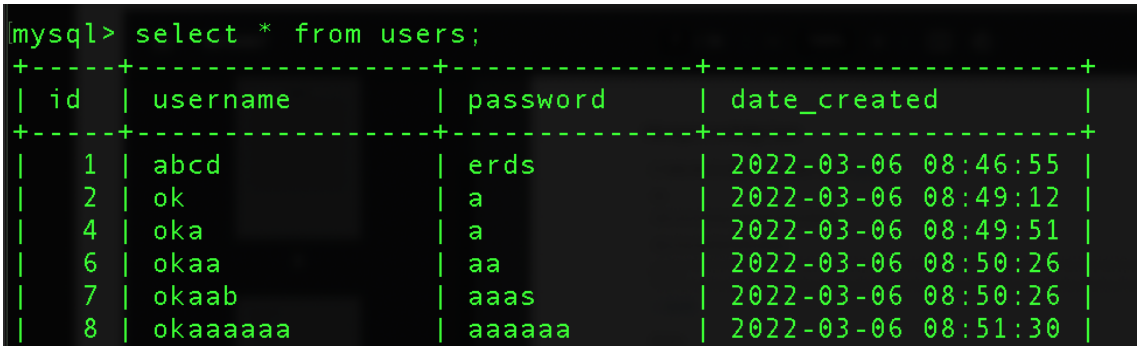


Figure 7: Tables

Figure 7 shows the tables that are in our "logins" database. The content inside the table that we collect from the users is shown in Figure 8.

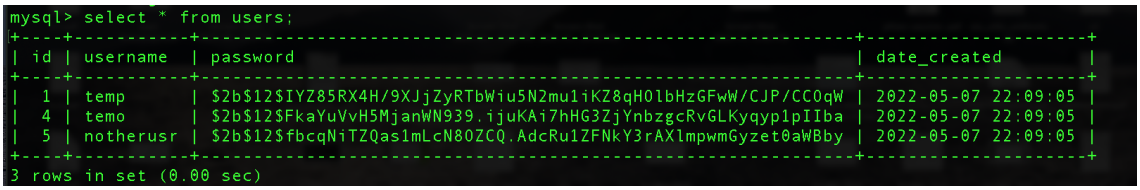


```
mysql> select * from users;
+-----+------------------+--------------+----------------------+
| id  | username         | password     | date_created         |
+-----+------------------+--------------+----------------------+
|   1 | abcd             | erds         | 2022-03-06 08:46:55 |
|   2 | ok               | a            | 2022-03-06 08:49:12 |
|   4 | oka              | a            | 2022-03-06 08:49:51 |
|   6 | okaa             | aa           | 2022-03-06 08:50:26 |
|   7 | okaab            | aaas         | 2022-03-06 08:50:26 |
|   8 | okaaaaaa         | aaaaaa       | 2022-03-06 08:51:30 |
```

Figure 8: Table Content

ID in Figure 8 is the primary key for the users table which holds the username and password that the user entered in the app. Each username is checked before storing it into the table to prevent duplicate usernames. The date created column was populated at time the credentials were stored into the database using the datetime library from Python.

As we know, storing the users raw password is not secure. To help protect our users and keep their credentials secure, we decided to encrypt the users passwords using the bcrypt library available in Python. An example of how users passwords are stored in the database is shown in figure 9.



```
mysql> select * from users;
+----+-----------+--------------------------------------------------------------+---------------------+
| id | username  | password                                                     | date_created        |
+----+-----------+--------------------------------------------------------------+---------------------+
|  1 | temp      | $2b$12$IYZ85RX4H/9XJjZyRTbWiu5N2mu1iKZ8qHOlbHzGFwW/CJP/CCOqW | 2022-05-07 22:09:05 |
|  4 | temo      | $2b$12$FkaYuVvH5MjanWN939.ijuKAi7hHG3ZjYnbzgcRvGLKyqyp1pIIba | 2022-05-07 22:09:05 |
|  5 | notherusr | $2b$12$fbcqNiTZQas1mLcN8OZCQ.AdcRu1ZFNkY3rAXlmpwmGyzet0aWBby | 2022-05-07 22:09:05 |
+----+-----------+--------------------------------------------------------------+---------------------+
3 rows in set (0.00 sec)
```

Figure 9: Encrypted Passwords

Bcrypt is a password salting and hashing library available in Python. It uses salting which is a random string that is attached to the entered password to make it more secure. Once the salt is added to the entered password, it then runs through the hash function. Bcrypt, by default, adds the salt to the password and even defaults to a cost function of 12. The cost function is the amount of times the password is hashed. The more your cost

is, the more secure the password will be. Figure 10 demonstrates the process of encrypting a password similar to the way bcrypt works. The hashed password is then stored into the MySQL database after verifying if the user already exists.

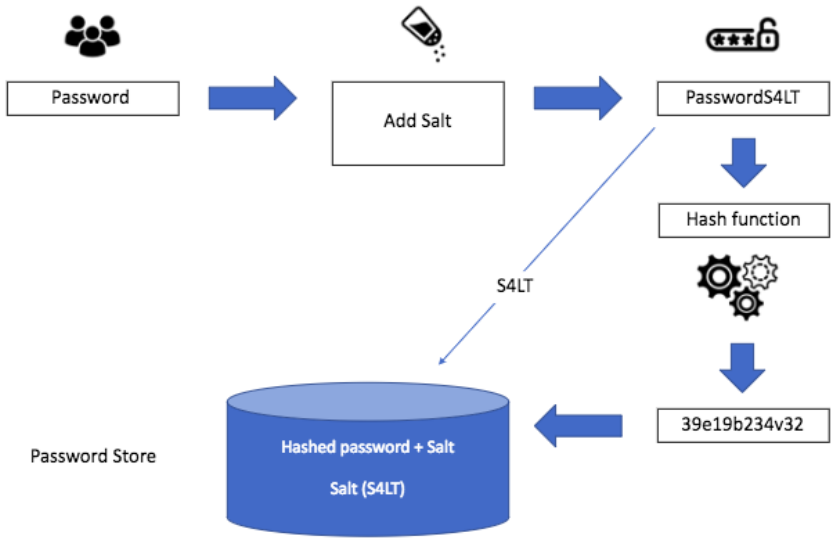


Figure 10: Bcrypt encryption process

Figure 11 shows the format of the passwords after the process of salting and hashing is finished. See figure 10.

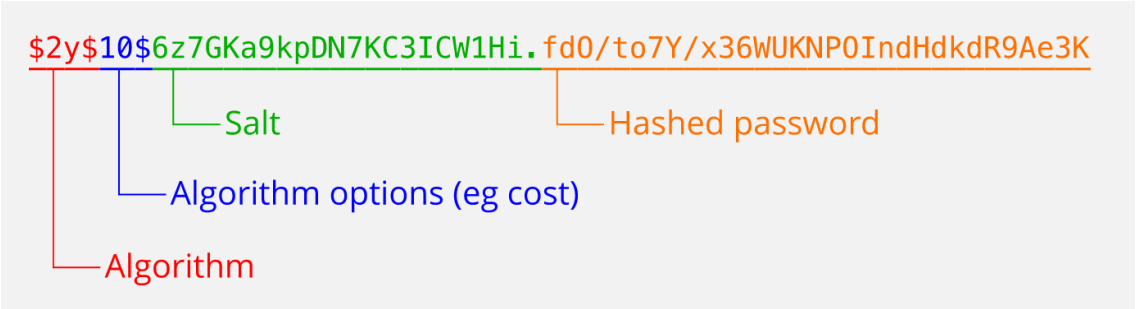


Figure 11: Bcrypt hashed password

The figure above represents to overall format of the new encrypted password. For our encryption process, we generate a new salt for each password. We accomplish this by using the genSalt() function provided in the bcrypt library.

4.2.1 Database Implementation

To implement MySQL with Python, we had to download the SQLAlchemy library in Python which allows for communication between Python and databases. Once we got that working, we had to configure the Flask app object that we created with the path to connect with our MySQL database. We also had to use a connector specifically for MySQL and Python which is called pymsql. This established the actual connection to our database from Python. This connector basically told python what database to use and grant access to that database since it required the username and password for access.

To create the tables from Python is MySQL, we created a Users class in Python with the specified attributes shown in Figure 8. Python fully created the table for us simply by running a few commands inside the Python shell. Since it had prior access to the database with SQLAlchemy, it was able to create the table exactly where we wanted it.

```
class Users(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(20), nullable=False, unique=True)
    password = db.Column(db.String(100), nullable=False)
    date_created = db.Column(db.DateTime, default=datetime.utcnow())

    def __init__(self, username, password):
        self.username = username
        self.password = password
```

Figure 12: Users class

4.3 Server & Client

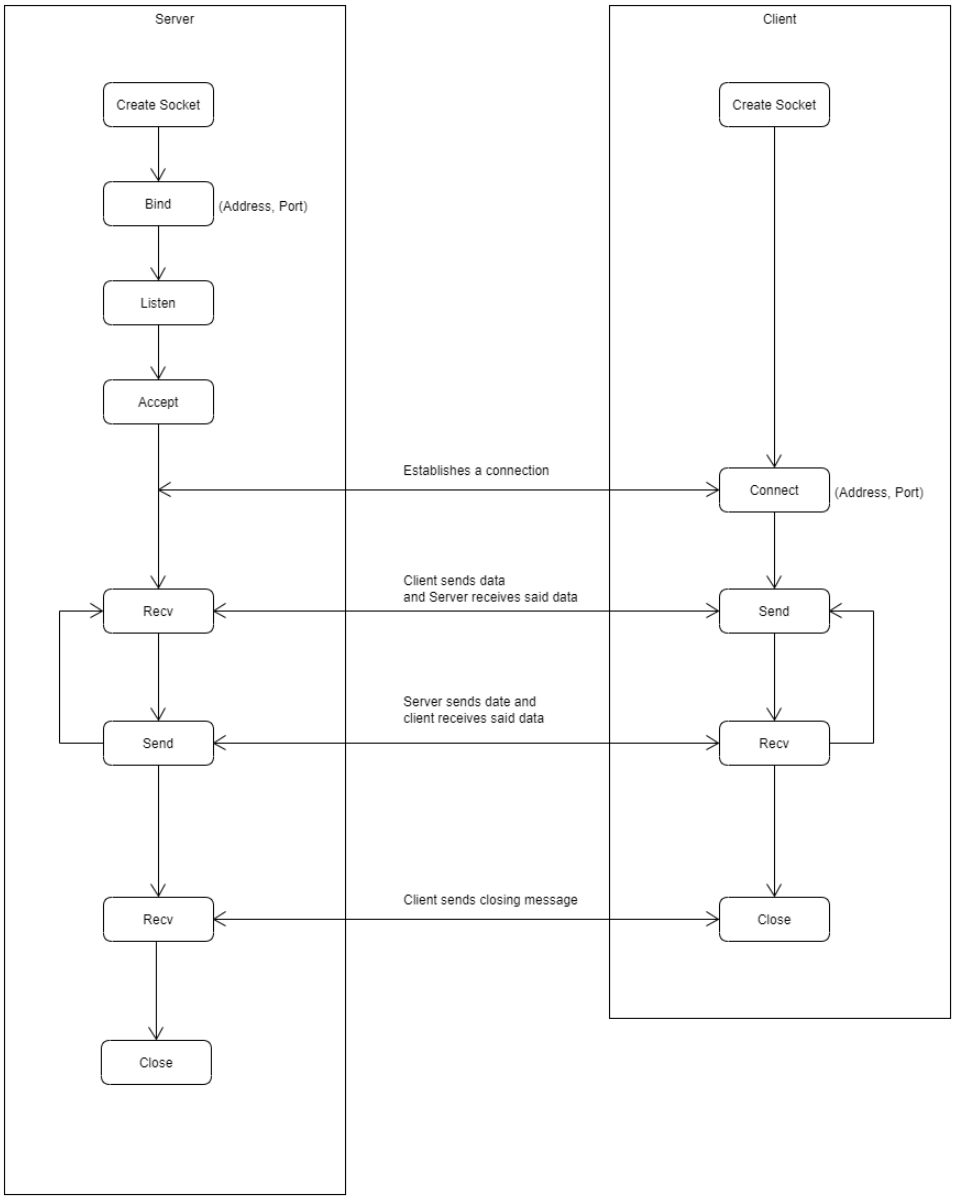


Figure 13: Server & Client UML Activity Diagram

4.3.1 First Implementation - Socket

In order to establish our Server-Client communication we decided to use the Python socket module. Socket is a low level networking interface originated from the Berkeley socket API that would help us as an introduction into networking. Figure 13 demonstrates the flow of interaction between server and client using socket programming. In these instances the server acts as the listener on a particular IP while the client reaches out to that connection. Both handshake with one another to establish connection. Afterwards the client or server is able to send data to one another over this connection.

```
SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
SERVER.bind((SEVER_HOST, PORT))
```

Figure 14: Socket Initialization

Figure 14 demonstrates how our socket is initialized. AF_INET refers to IPV4 and SOCK_STREAM means we're using TCP protocol. SOL_REUSEADDR allows the local addresses to be reusable. SERVER_HOST is the IP address that we specify and in this case we set it to our local machine/local-host. In real world implementations we would have specified specific IP addresses we wanted to use, but since we only plan on running it on our machine we don't need to. PORT is straightforward and specifies software-defined numbers associated with a network protocol where the process will occur, in this instance it's 5000. Not every port is available on a local machine like our personal computers. Ports 491152-65535 are usually free of use.

We handle the message passing and receiving using the Multi-thread process otherwise we would create a wait and request system where the server would only respond when a request is made. We don't want that otherwise new messages would only be sent to the user when they sent a message. Meaning users could be unaware of a conversation going until they decided to message something. Figure 13 & 16 shows how the structure of threads are.

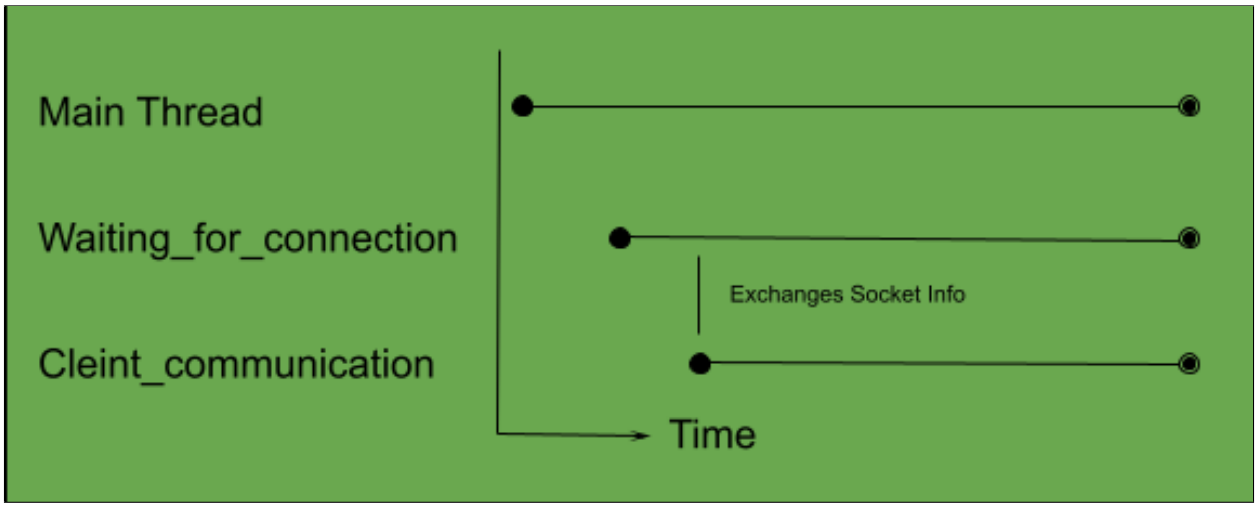


Figure 15: Server Thread Diagram

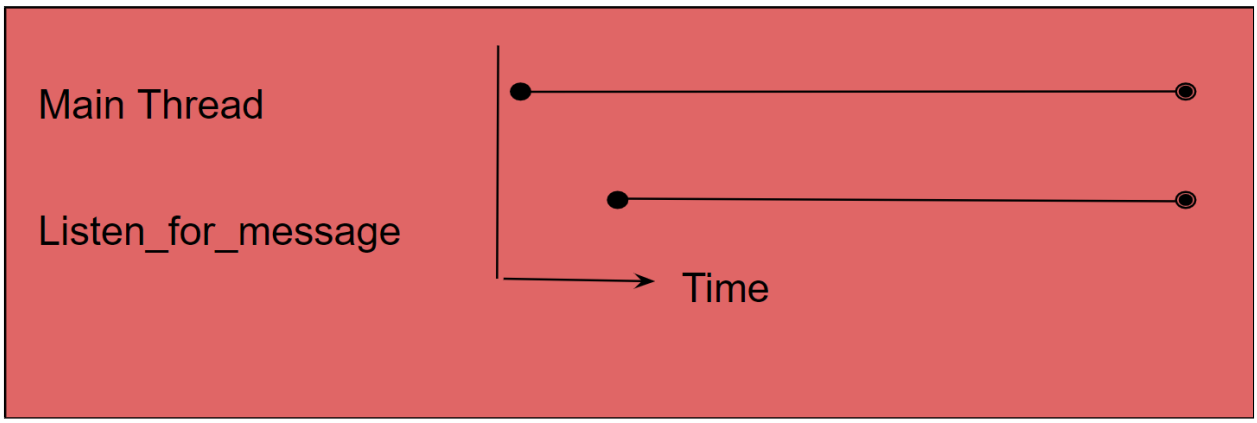


Figure 16: Client Thread Diagram

Python socket only allows for data to be transferred in binary format. So part of our process is to convert strings to UTF-8 encoding and reverse engineer incoming messages. In order to broadcast, our server keeps an array of User class we created that are connected to our server and broadcasts to them every time the server receives a message from any of those users. Our User class stores the name of the User, the socket they created, and their IP address for future reference.

4.3.2 Challenges

When we attempted to combine both our flask and server system we ran into some issues. We converted our python client script into a class that would be created and called along with our flask code. In order to imitate users communicating between each other we opened two browsers and launched our app. We first login to the first and then second. However we ran into an issue when sending messages where the name of users being shared between the client and server was wrong. The people communicating with each were being displayed with the same name. For example if Joe and Timmy login in Joe would be displayed as Timmy even though he's not Timmy. We determined that it had to do with the flask session and cookies on the web browser. It seems as if the previous user's socket was being overridden with the new one.

4.3.3 Resolve - Socket-IO

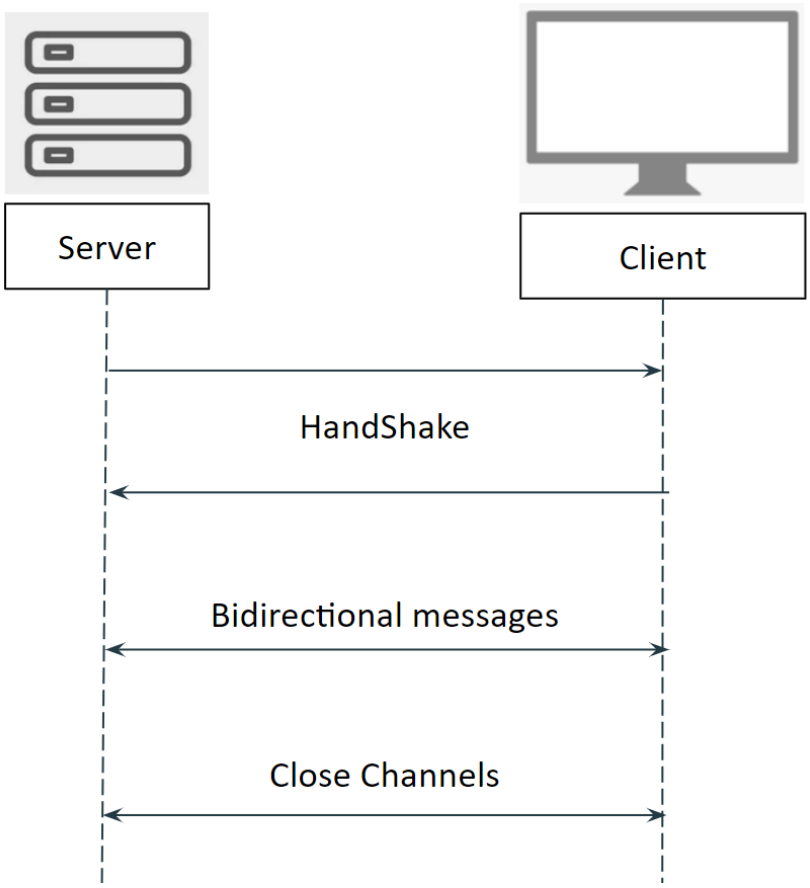


Figure 17: New Server-Client-UML-Diagram

In order to resolve this issue we looked more into web sockets and came across Socket-IO. There are two ways Socket-IO works from HTTP to Web-socket transport. In this instance we used HTTP for our application. However if we were to implement using a web-socket, Socket-IO allows for HTTP roll back when web-socket can not be established. This HTTP long-polling back was created when the web browser didn't't allow for such implementation, but that's changed. There are some benefits to using socket-IO from automatic re-connection, implementation of a Full Duplex Communication system as in figure 17, Low latency and HTTP roll back. Python socket-IO wraps around Python flask making the flask application act as a server.

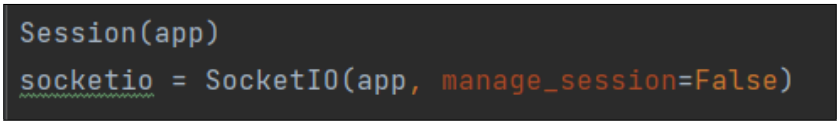


Figure 18: Socket-IO Initialization

Upon initialization of our Socket-IO the socket will create its own User Session, if any changes to the Flask session occur our server will not have access to it. We don't want that so as can see in figure 18 manage_session = false ensures that we use the Flask session not the Socket-IO session. Python Socket-IO is an event driven system. When the client wants to communicate with the server it emits an event name and a list of arguments.

Figure 19: Socket-IO Event-Handling Example

Figure 19 demonstrates an event handler we created to process when a User client joins rooms. In figure 19 an event is listener when its initialize as socketio.on/socket.on However in order to implement the client aspect of the server-client model we use JavaScript Socket-IO embedded in our HTML as a client. To demonstrate a flow of events in figure 19 we see at the end of the event "join" that an emit is called with the event name "status". The event "status" is embedded in our JavaScript Socket IO as shown in figure 20, it receives data and displays it onto our text area HTML.

Figure 20: JavaScript Socket-IO Event-Handling Example

Essentially our new implementation with Socket-IO still works the same way as our previous client-server scripts did, but the new implementation handled a lot more at once.

We use Python Socket-IO to establish our server and JavaScript Socket-IO for our client. Socket-IO allowed us to do more at a much quicker rate. With Socket-IO, we were able to handle multiple users joining rooms, sending messages and displaying them to all users within the room, and even leaving rooms. The way rooms are handled is by keeping a channel of users in the specific room and only broadcasting to those users. Since JavaScript was used for SocketIO communication, we were able to combine it with our html so that when a button was clicked, JavaScript code would be ran. An example is the leave button in a chat room. When the leave button is clicked, the JavaScript function 'leave_room()' is called from our socket.js file. The function essentially emits the 'left' function from our app.py file and disconnects the socket from that room.

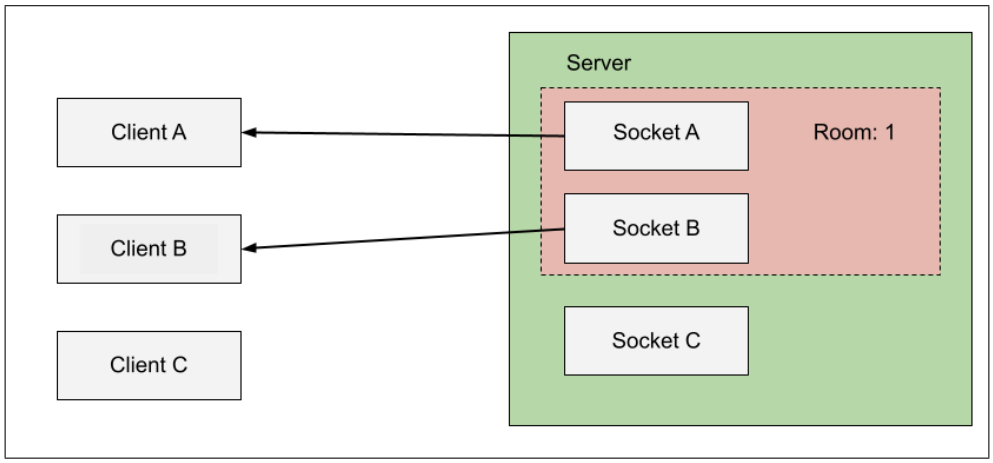


Figure 21: Room Diagram

4.4 Deployment

4.4.1 Linode Server Setup

To set up the server, I registered for a free $100 credit on Linode's site. After, I purchased a nanode for $5 a month, but only get charged for the time it is running. The next thing I did was setup the server with a password and the operating system (Ubuntu) then ssh into that server from my computer to login. Once I logged in, I downloaded a few things that I needed in order for it to be semi-secure and to install MySQL.

4.4.2 Project Setup

Upon successfully setting up the Linode Server, it was time to setup our flask project. To set it up, I had to transfer the project from my computer to the server using the SCP command. I then proceeded to create a virtual environment in the server as well where I installed all the necessary libraries from the requirements.txt file. After all necessary libraries downloaded, I setup the MySQL database which was a bit of a hassle. I had to create a user, and change the path to the database from our app.py file since it was now running from the server. After doing some MySQL configuration, it was finally working and all the user's credentials were being stored as intended.

4.4.3 Testing

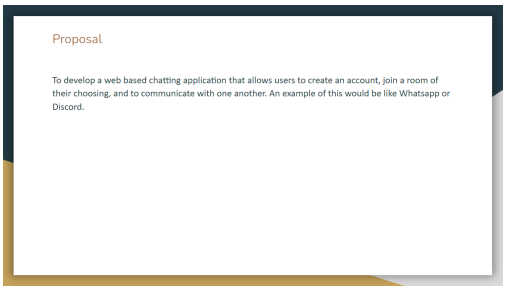
Testing of the project running off the Linode Server was done as normal. I would just enter the IP address that Linode assigned when setting up the server to the URL search bar. The app worked on all devices connected to any network. The real test came during our live demo since we were not sure if the server would be able to handle multiple users at once. Thankfully, it could and it all went well as classmates were able to register and login to use the app in real time and chat with one another.

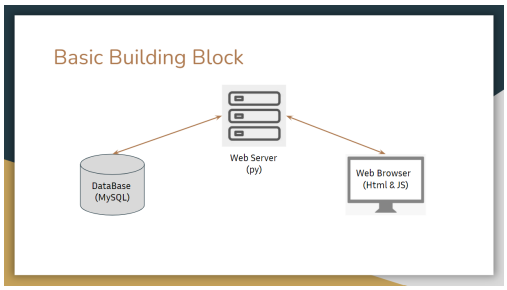
5 Conclusion & Future Work

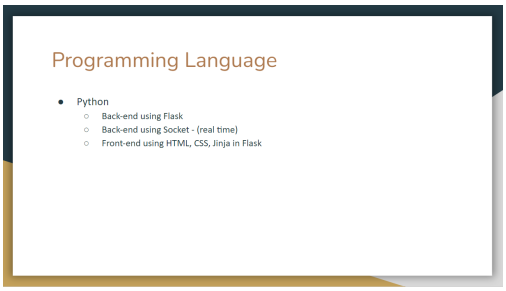
In conclusion we were able to implement all basic functions of our chatting app and even some that we were planning on implementing, but didn't know if we'd be able to. We would like to implement more animations, like a loading circle after the user enters their credentials. The reason for this is it would apply a visual cue so the user comprehends that the site is loading based on their inputs. We would also like to have the ability for users to change their password or restore it in the future. The reason is that users will not always remember their passwords, and since the passwords are now being encrypted, there is no way to gain access to that account anymore. Another thing that we would like to have implemented is the ability to add friends and make private rooms with invite links and direct messaging. Overall, we'd like to make the front end a bit more appealing using JavaScript. One thing that we would do different is use Postman for testing purposes. Postman seems to make the testing of a project much easier and faster by sending requests directly to the app. An example of where we would use it would be with the sign up and login features that we have. Other than that, we were successful in developing our app and even deploying it onto a server for classmates to use from their environment.

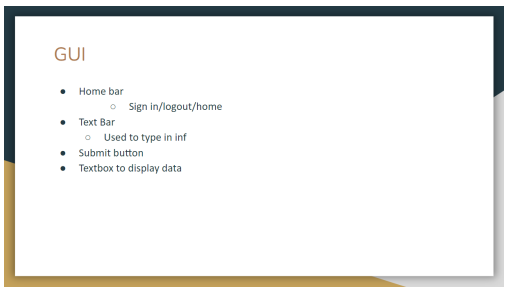
6 Appendix

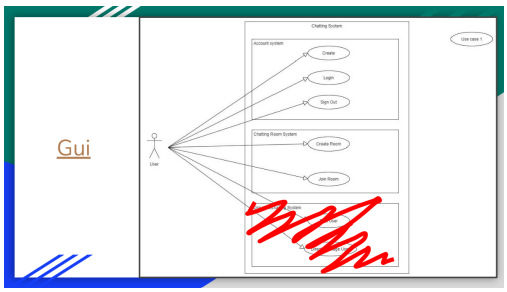
6.1 Final Presentation











Before:

After:

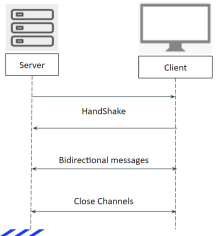
Before:

After:

DataBase

DataBase

- MySQL with python SQL Alchemy
- Create a database called Users
- Table call Users
- Stores Username and Password Credentials
- Date Account was created
- With Primary Key User-Id



Server-Client

Challenges faced

Socket vs SocketIO

- Issues with names in session
- SocketIO wraps around flask package
- SocketIO allows for reconnection automatically
- Full Duplex Communication
  - simultaneous data transmission and receptions
- LOW-LATENCY
- EVENT-BASED

What we wish we would have done differently

Done Differently

- Learn more front-end
  - JavaScript
  - Better Animation