Criteria C: Development

List of techniques used

1. Token-based authentication (JWT)
2. Complex SQL query for fetching messages
3. Nested loop for fetching mutual sign-ups
4. Manipulation of Document Object Model (DOM) for calendar function

Explanation of techniques and evidence

Token-based authentication

To fulfill SC #1 and to record the information of the current use, an authentication method is needed for the web application. Initially, I decided to use cookies as a way of storing. However, based on my research about cookies and testing in the browser, I found out that the cookie is not a secure way of storing the user information since this is a variable stored in the browser on the client side, which can be easily changed, causing identity theft if not hashed. So my solution to this issue was to change from client to server side by using sessions.

The code below shows that I could store the current user in the dictionary session.

```python
session['current_user'] = users[username]
return redirect(url_for('profile'))
```

In order to use the session variable, I needed to define an initial secret in the variable of the Flask application. I did this in the code below and generated a random string as a secure key.

```python
app = Flask(__name__)
```

```
app.secret_key = "randomtextwith12345"
```

However, after doing some research, I realized that simply assigning user IDs to sessions is not a secure practice. A user can easily access the browser's inspector and change the user ID, thus gaining unauthorized access to other users' accounts if it is saved in plain texts.

In order to solve this problem, I decided to use the JWT token. Each subsequent request after the user is logged in will include the JWT, allowing the user to access the routes that are permitted [1]. A user ID encoded with JWT looks like this:

```python
jwt.encode({'user_id': user_id}, token_encryption_key,
algorithm='HS256')
```

The login system now is safe and free of data leaks so that personal information is protected.

To ensure that some parts of the website are only accessible for authorized users, the login status of the user needs to be confirmed in some routes. In order to validate the login status of the current user, I designed a user validation function called “token_authorization”.

```python
def token_authorization(f):

    @wraps(f)
    def decorated(*args, **kwargs):
        if 'token' not in session:
            return redirect(url_for('login'))
        try:
            data = jwt.decode(session['token'], token_key,
algorithms=['HS256'])
        except:
            return redirect(url_for('login'))

        # If the token is valid, continue to the original function with
the original arguments
        return f(*args, **kwargs)
    # Return the new decorated function
    return decorated
```

First, the `@wraps` function is used to preserve the metadata of the original function, such as the name and docstring. Then, the next lines check if the 'token' key exists in session. If not, the user is redirected to the login page. Next, a `try` block attempts to decode the token using `jwt.decode()`. The try/except block tests the first statement, and provides a solution if it doesn't work. This solution provides a solution for safe login/registration systems and effectively addresses the authorization issue using the complex method of token authentication, showing the skill of **decomposition**: the problem is broken down into two parts, protection of current user’s information and validating that the user is logged in.

<u>Complex SQL queries for fetching messages</u>

In order to achieve the message function described by SC #4, I decided to use a SQL query. This query is to fetch the past messages from the database that the user received to display on the web application. This is part of the message function since the user needs to be able to check message history as well as sending new messages to others.

```python
db = database_worker("frisbeecrew.db")
users = db.search(f"""SELECT name, id FROM users WHERE id !=
{user_id}""")

messages = db.search(f"""
    SELECT messages.*, users.name AS sender_name
    FROM messages
    INNER JOIN users ON messages.sender_id = users.id
    WHERE receiver_id = {user_id}
    ORDER BY timestamp DESC
""")
# results are ordered by the timestamp field in descending order using
ORDER BY and DESC to ensure most recent messages appear first
```

The variable messages is defined by the search result of a complex SQL query. I had to use a complex INNER JOIN because I had the difficulty of combining two tables: message and users. INNER JOIN fetches rows that have matching values, which makes it possible to retrieve the message sender's information, including name(which is stored in "users" table) and id.

First, the query selects all columns from the message table using the asterisk(*) symbol and name column from users table. I decided to alias the name column as sender_name to make it more descriptive and relevant. Then, an INNER JOIN is used to combine rows from messages table and users table based on a common column. It matches the "sender_id" in the messages table with the "id" in users table. This allows the query to retrieve the sender's information as "sender_id" is a foreign key pointing to the users table. WHERE is used to filter results to only show messages where "receiver_id" and "user_id" are the same. I used curly brackets to dynamically replace {user_id} with the actual "user_id" in the python code. This code uses various SQL commands, showing a high level of **abstraction** and complexity.

Nested loop

I came across a challenge when trying to achieve the view sign-up participants function described by SC #7: There are multiple sets of data that need to be compared to see the sign-ups from other users. The ids of other users need to be fetched first, then each compared to every event that the current user has signed up for. Considering the complexity of this action and the layers of data, I decided to use a nested loop to solve this problem and fulfill SC #7.

```python
@app.route('/mutual_signups/<int:user_id>', methods=['GET'])
@token_required
def mutual_signups(user_id):
    # Fetch all users except the current user
    all_users_query = f"SELECT id, name FROM users WHERE id != 
{current_user_id}"
    all_users = db.search(query=all_users_query)

    mutual_signups_result = {}

    # In the dictionary, the keys will be the other users’ names, and 
the values will be the count of mutual sign-ups.

    for user in all_users:
        other_user_id = user[0]
        other_user_name = user[1]

        # Fetch events signed up by the current user
        current_user_signups_query = f"SELECT event_id FROM signups 
WHERE user_id = {current_user_id}"
        current_user_signups = 
db.search(query=current_user_signups_query)


        other_user_signups_query = f"SELECT event_id FROM signups WHERE 
user_id = {other_user_id}"
        other_user_signups = db.search(query=other_user_signups_query)

        mutual_signups_count = 0
      # First for loop goes through each event the current user has 
signed up for, second for loop goes through the events that the other 
user in the loop  has signed up for
```

```
for current_user_signup in current_user_signups:
    for other_user_signup in other_user_signups:
        if current_user_signup[0] == other_user_signup[0]:
            mutual_signups_count += 1

# Store result for each user
mutual_signups_result[other_user_name] = mutual_signups_count
```

Now, the outer loop is initiated to loop through every other user from the “all_users” variable. I extracted the id and name of every other user. Sign-ups of the current user and that of the other user in the loop are retrieved. A “mutual_signups_count” variable that counts the number of mutual sign-ups is initiated. Then, I designed two for loops. The if statement checks if both users signed up for the same event. If so, the counter is incremented. The nested loop shows a high level of **abstraction**, especially in the inner loops where sign-ups are compared.

Document Object Model (DOM)

A challenge that I came across while developing the app for my client was the calendar function(SC #6). I planned to create a static calendar. However, as described in SC #3, events include date, time, and content, and displaying all this information on a static calendar grid will make the page look cluttered. I needed a way of changing the page dynamically. I decided to use JavaScript to manipulate the DOM directly to display the event info.

```
Python
<script>
    document.addEventListener('DOMContentLoaded', function() {
        var calendarEl = document.getElementById('calendar');

        // Initialize the calendar
        var calendar = new FullCalendar.Calendar(calendarEl, {
            initialView: 'dayGridMonth',  // default view is month
            headerToolbar: {
                left: 'prev,next today',
                center: 'title',
                right: 'dayGridMonth,timeGridWeek,timeGridDay'  // views
            },
            events: [
                {% for event in events %}
                {
                    title: '{{ event[1] }}',
                    start: '{{ event[2] }} {{ event[3] }}',
                },
                {% endfor %}
            ],
            eventClick: function(info) {
                var eventTitle = info.event.title;
                var eventStart = info.event.start.toLocaleString();  //
Format date and time
                alert('Event: ' + eventTitle + '\nStart: ' +
eventStart);  // Display event title and start time
            }
        });

        // Render the calendar
        calendar.render();
    });
</script>
```

On the first line, “DOMContentLoaded” ensures that the code inside only runs after the calendar.html page has been fully loaded. Then, I had to use the getElementById() method to get the HTML element with the id “calendar.” A calendar is initialized using the “calendarEl” element. The code then sets the default view using “initialView” and configures the navigation bar using “headerToolbar.” Here the “events” variable is passed from app.py.

Then, I decided to implement an eventClick function to make sure that it only runs after an event on calendar has been clicked. I had to use this function to display event info without making the calendar look cluttered.

WORD COUNT: 1064

Work Cited

[1] JSON Web Token. “JSON Web Token Introduction - jwt.io.” *JWT.io*, https://jwt.io/introduction. Accessed 23 September 2024.