QuickBites: Food Delivery System

Dipesh Patidar , Piyush Singh , Pratyaksh Bhayre , Mrugank Patil , Abhimanyu Abhimanyu

1. Introduction

QuickBites is a campus-based food delivery system designed to streamline food ordering and delivery from various outlets within the institute. The system facilitates a seamless ordering process, tracks real-time order status, and provides analytics for better decision-making. The goal is to enhance efficiency for both customers and vendors, reducing wait times and optimizing food delivery.

---

2. Objective

To integrate the QuickBites project database with the centralized CS432CIMS system, ensure secure API operations through session validation, implement role-based access control (RBAC), log all critical changes, and prevent unauthorized data tampering.

Assignment Details

1. SQL Server Location: http://10.0.116.125/phpmyadmin
2. Authentical APIs Location: http://10.0.116.125:5000/
3. Centralized Database Name: cs432cims
4. Centralized Tables in CIMS Database:
   (a) members: Contains details of all members (e.g., member id, name, email).
   (b) members group mapping: Maps members to groups (to maintain integrity).
   (c) payments: Contains payment-related information.
   (d) login: Contains user credentials (userid, password) and active sessions (session)

---

3. Tasks

Task 1: Member Creation

When a new member is created in the centralized members table, create a relevant entry in the login table for this member with default credentials (userid, password). This allows the member to log in and obtain a session token using the authentication API (authUser).

- First we have to run the flask application using the main.py file. Using the thunder client we have to establish the connection with the SQL Server on the http://10.0.116.125/phpmyadmin.

main.py | Login.py | New Request | AddUser.py

GET http://127.0.0.1:5000/dbcon Send

Query | Headers 2 | Auth | Body 1 | Tests | Pre Run

Status: 200 OK Size: 77 Bytes Time: 175 ms Response

```
{
  "database": "cs432cims",
  "message": "Database connection successful"
}
```

Copy

- As we have successfully connected to the SQL server now we have to add the admin details so that we can add the admin and modify anything required.

main.py | Login.py | New Request | AddUser.py

POST http://127.0.0.1:5000/addUser Send

Query | Headers 2 | Auth | Body 1 | Tests | Pre Run

JSON | XML | Text | Form | Form-encode | GraphQL | Binary

```
{
    "username": "dipesh",
    "password": "volley_ball#34",
    "role": "admin",
    "email": "dipesh@example.com",
    "DoB": "2004-01-23"
}
```

Status: 200 OK Size: 44 Bytes Time: 84 ms Response

```
{
  "message": "admin added successfully"
}
```

Copy

- We can confirm this by checking the phpmyadmin. Username Dipesh have been added as a admin

Show all | Number of rows: 500 Filter rows: dipes Sort by key: None

+ Options

		UserName	ID	emailID	DoB
Edit Copy Delete		Dipesh	2095	dipesh@example.co	1990-01-01

- Now we have to login with the username and the password for the admin.

main.py | Login.py | New Request | AddUser.py

POST http://127.0.0.1:5000/login Send

Query | Headers 2 | Auth | Body 1 | Tests | Pre Run

JSON | XML | Text | Form | Form-encode | GraphQL | Binary

```
{
    "username": "dipesh",
    "password": "volley_ball#34"
}
```

Status: 200 OK Size: 0 Bytes Time: 98 ms Response

Copy


```
{
  "group": null,
  "message": "Login successful",
  "role": "admin",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
      .eyJ1c2VyIjoiZGlwZXNoIiwicm9sZSI6ImFkbWluIiwiZXhwIjoxNzQ0NjUyNjkxLCJncm91cCI6bnVsbCwic2Vzc2lvbl9pZCI6MjA5
      NX0.45eLfZZx-gk0afdKMuh5_cRmRpPcVcpEZ5EJl9u9M10",
  "username": "dipesh"
}
```


- Using isAuth we can obtain a session token using the authentication API. The user "dipesh" is successfully authenticated. Has "admin" role permissions. Session token is valid until the expiry timestamp (1744656374)

main.py | Login.py | DeleteUser.py | New Request | AddUser.py

GET http://127.0.0.1:5000/isAuth Send

Query | Headers 2 | Auth | Body 1 | Tests | Pre Run

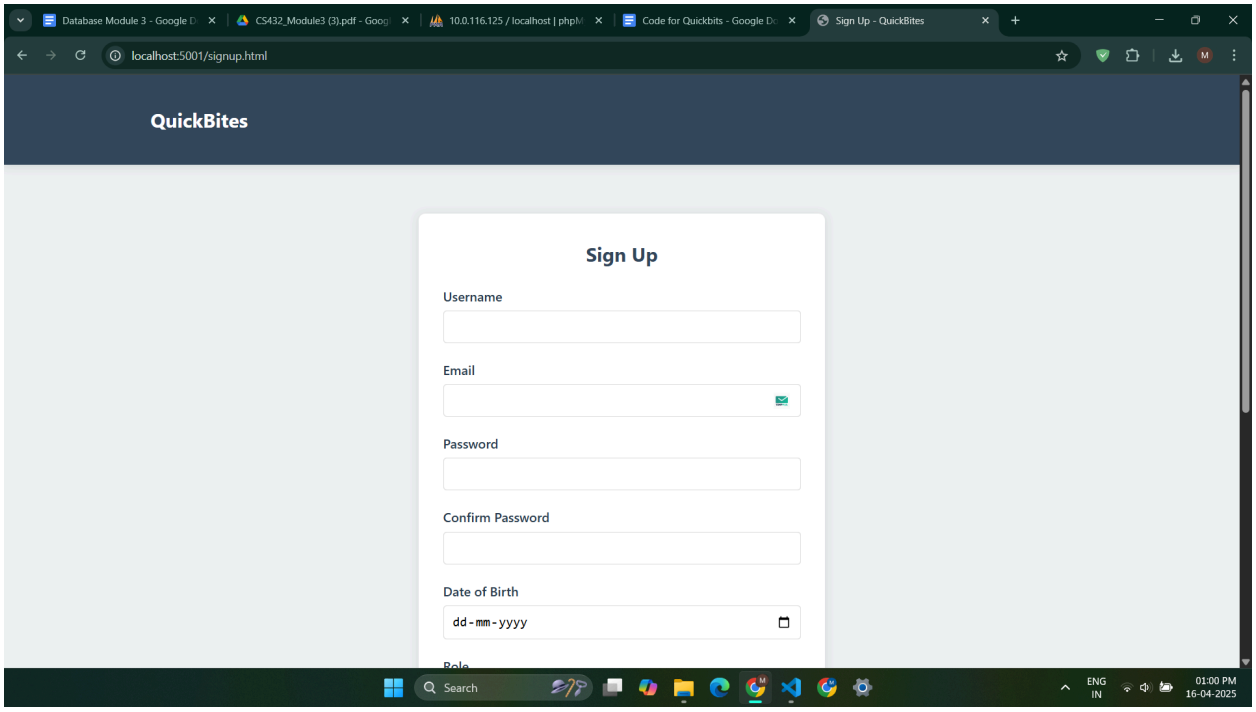
JSON | XML | Text | Form | Form-encode | GraphQL | Binary

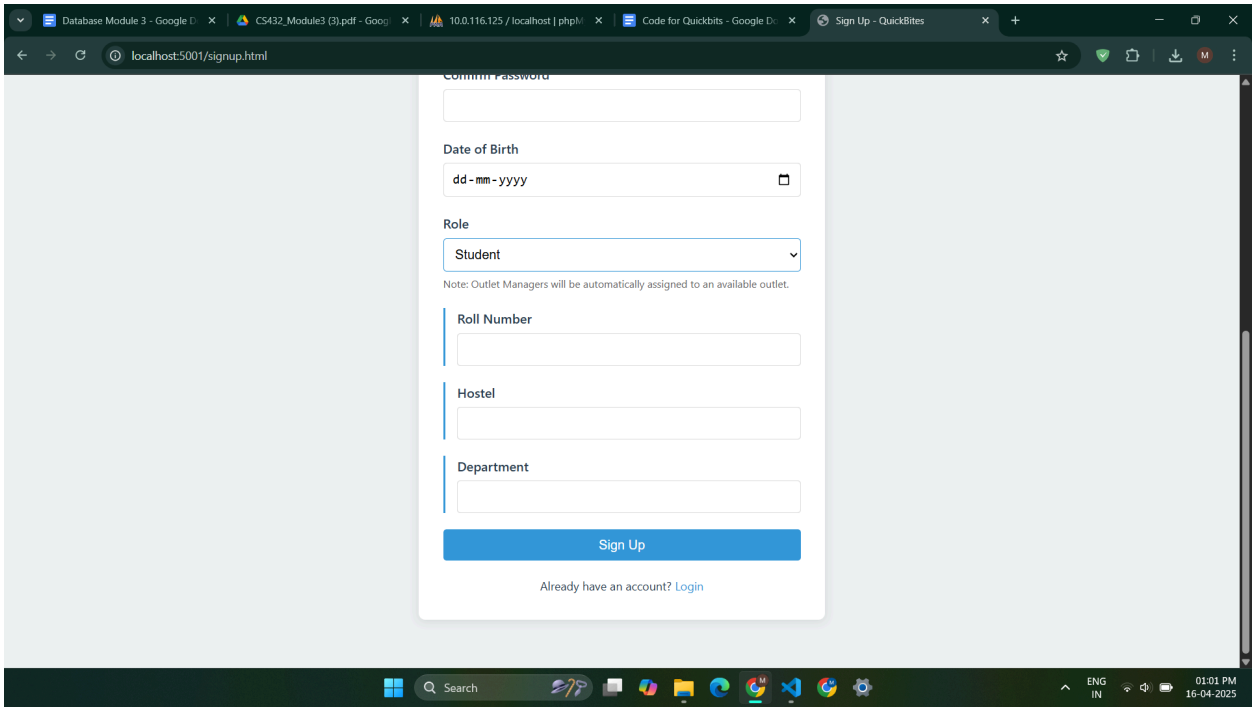
```
{
    "username": "dipesh",
    "password": "volley_ball#34"
}
```

Status: 200 OK Size: 147 Bytes Time: 4 ms Response


```
{
  "expiry": 1744656374,
  "group": null,
  "message": "User is authenticated",
  "role": "admin",
  "session_id": 2095,
  "username": "dipesh"
}
```

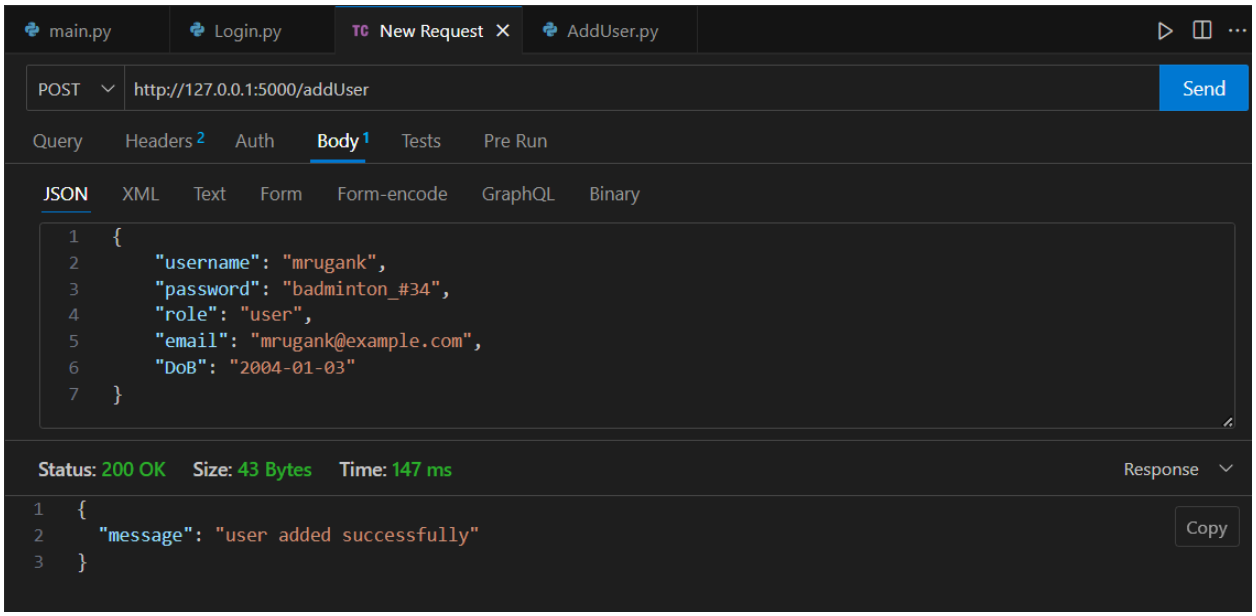
UI Implementation



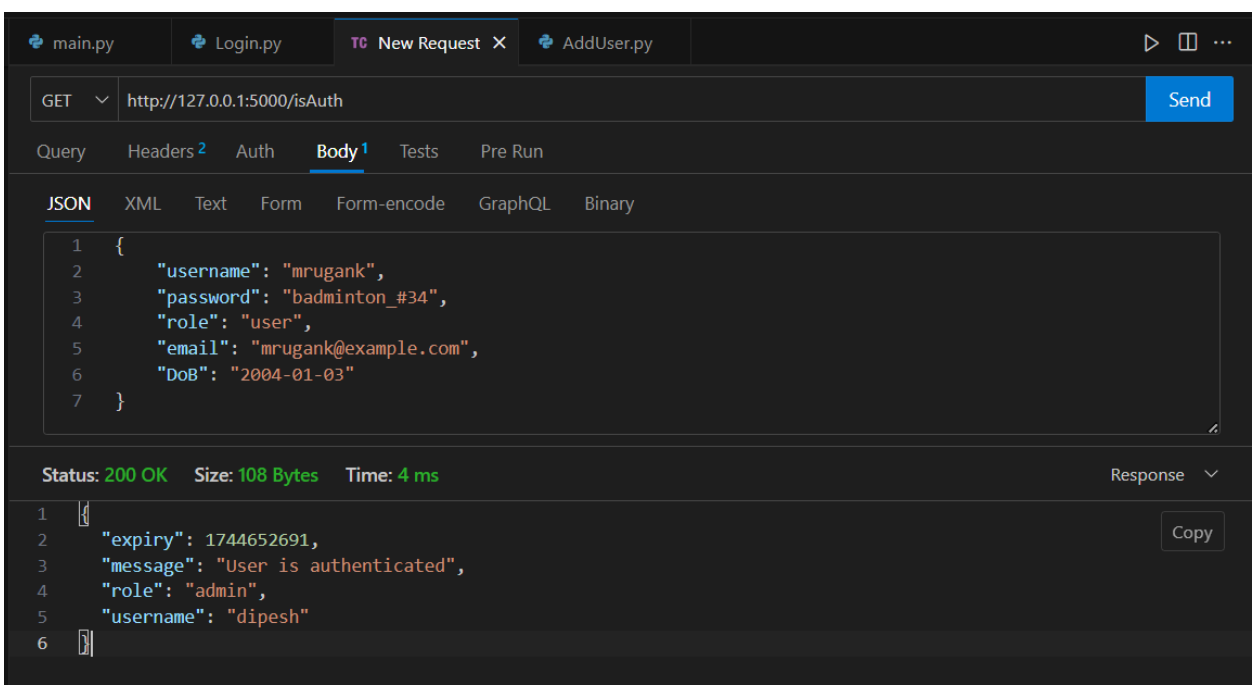


Task 2: Role-Based Access Control – Implement role-based access control (RBAC)

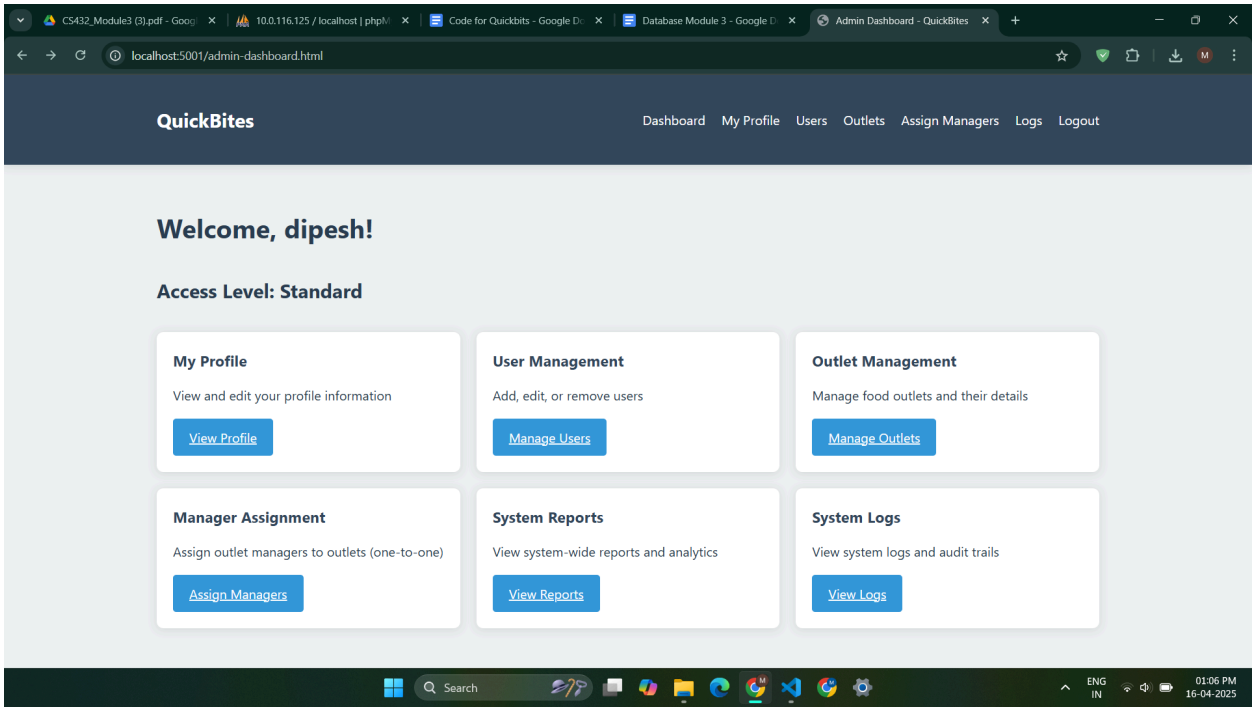
1. Admins should have full access to perform actions like adding/deleting members and accessing admin-level data.
    - We have added the username as “mrugank” as an user by the admin.

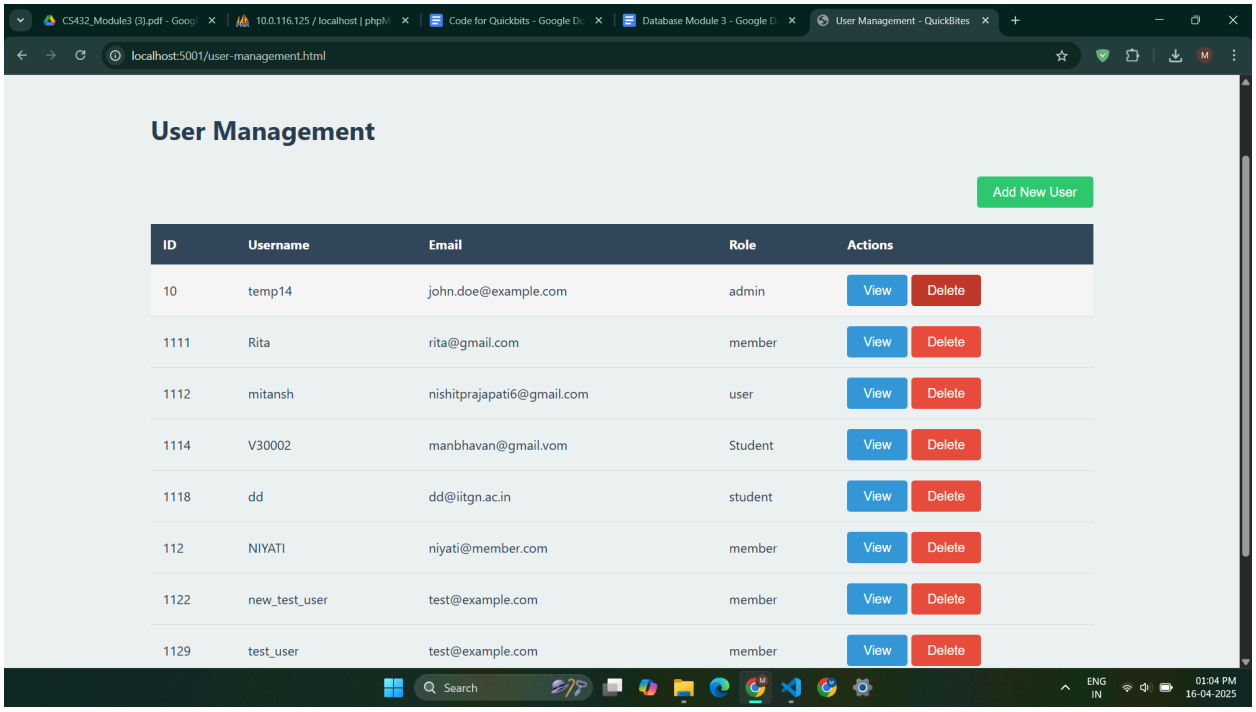


- We can authenticate it using the isAuth as who is the person which is doing the modification



UI Implementation





2. Regular users should not be allowed to perform admin-level actions such as adding/deleting members or accessing any CIMS database directly.

- Now we have logged in using the user “mrugank”.

main.py | Login.py | New Request | AddUser.py

POST http://127.0.0.1:5000/login Send

Query | Headers 2 | Auth | Body 1 | Tests | Pre Run

JSON | XML | Text | Form | Form-encode | GraphQL | Binary

```
{
    "username": "mrugank",
    "password": "badminton_#34"
}
```

Status: 200 OK Size: 0 Bytes Time: 103 ms Response

Copy


```
{
  "group": null,
  "message": "Login successful",
  "role": "user",
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
      .eyJ1c2VyIjoibXJ1Z2FuayIsInJvbGUiOiJ1c2VyIiwiZXhwIjoxNzQ0NjU0MTMxLCJncm91cCI6bnVsbCwic2Vzc2lvbl9pZCI6MjE0
      MH0.najpK5Wdw0Df9VwhamZaae5j93OLHcSNWoIATlouIWE",
  "username": "mrugank"
}
```


- Now we will try to modify the means to add other users to the database but it will not allow the user to do it. It will only be modified by the admin only. We will get an error as *"Unauthorized - Admin access required"*.

main.py | Login.py | New Request | AddUser.py

POST http://127.0.0.1:5000/addUser Send

Query | Headers 2 | Auth | Body 1 | Tests | Pre Run

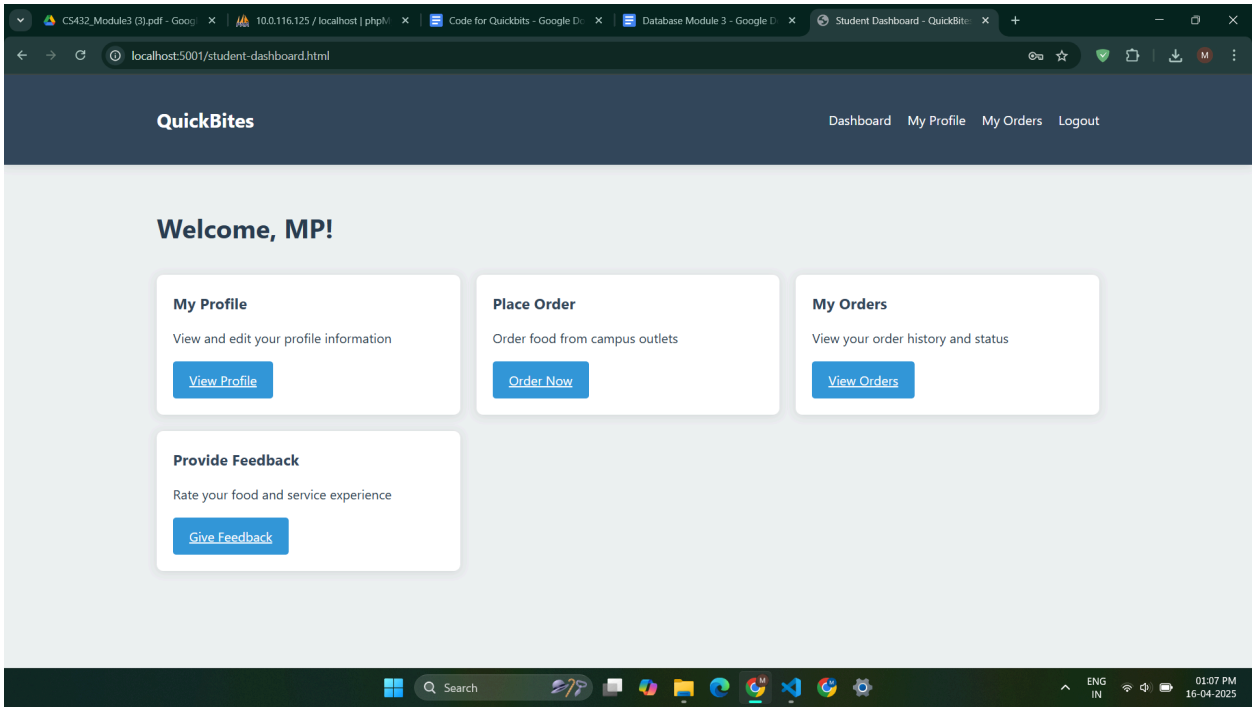
JSON | XML | Text | Form | Form-encode | GraphQL | Binary

```
{
    "username": "piyush",
    "password": "cricket_#34",
    "role": "user",
    "email": "piyush@example.com",
    "DoB": "2004-01-05"
}
```

Status: 403 FORBIDDEN Size: 54 Bytes Time: 12 ms Response

```
{
  "error": "Unauthorized - Admin access required"
}
```

UI Implementation



Task 3: Member Deletion – When deleting a member from the centralized members table:

1. Check if the member is associated with any other group using the members group mapping table.
   - For this first make the DeleteUser.py file to add the above functionality.
     The below does does the following

```
            # Check if member exists in other groups
            cursor.execute("""
                SELECT COUNT(*) as group_count
                FROM members_group_mapping
                WHERE MemberID = %s AND GroupID != %s
            """, (self.member_id, self.group_id))

            result = cursor.fetchone()

            if result['group_count'] > 0:
                # Member exists in other groups, only remove specific
group mapping
                cursor.execute("""
                    DELETE FROM members_group_mapping
                    WHERE MemberID = %s AND GroupID = %s
                """, (self.member_id, self.group_id))
```

- Queries the members_group_mapping table to count how many other groups the member belongs to (excluding the current group)
- If group_count > 0, it means the member is associated with other groups
- In this case, it only removes the specific group mapping instead of deleting the member completely
- If group_count = 0, it proceeds to delete the member completely from all tables

2. If the member is not related to any group, delete the member from both the members table and the corresponding entry in the login table.

```
# If member is not in other groups (group_count = 0), delete completely
else:
    # Delete from login table first (foreign key constraint)
    cursor.execute("DELETE FROM Login WHERE MemberID = %s",
                   (self.member_id,))

    # Delete from group mapping
    cursor.execute("""
        DELETE FROM members_group_mapping
        WHERE MemberID = %s
    """, (self.member_id,))

    # Finally delete from members table
    cursor.execute("DELETE FROM members WHERE ID = %s",
                   (self.member_id,))
```

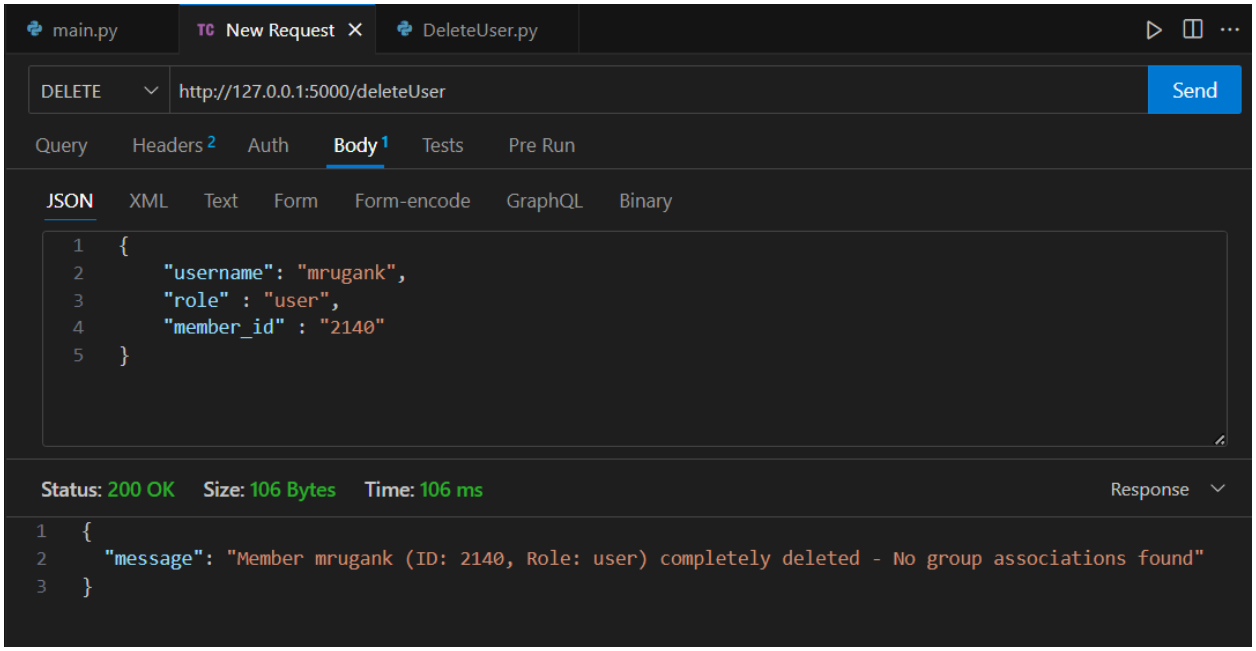
- Deletes from Login table first (due to foreign key constraints)
- Removes any group mappings from members_group_mapping table
- Finally deletes the member from the members table
- Uses transactions to ensure all operations complete successfully or roll back if there's an error

3. If the member is associated with other groups, only remove the specific group mapping from the members group mapping table.

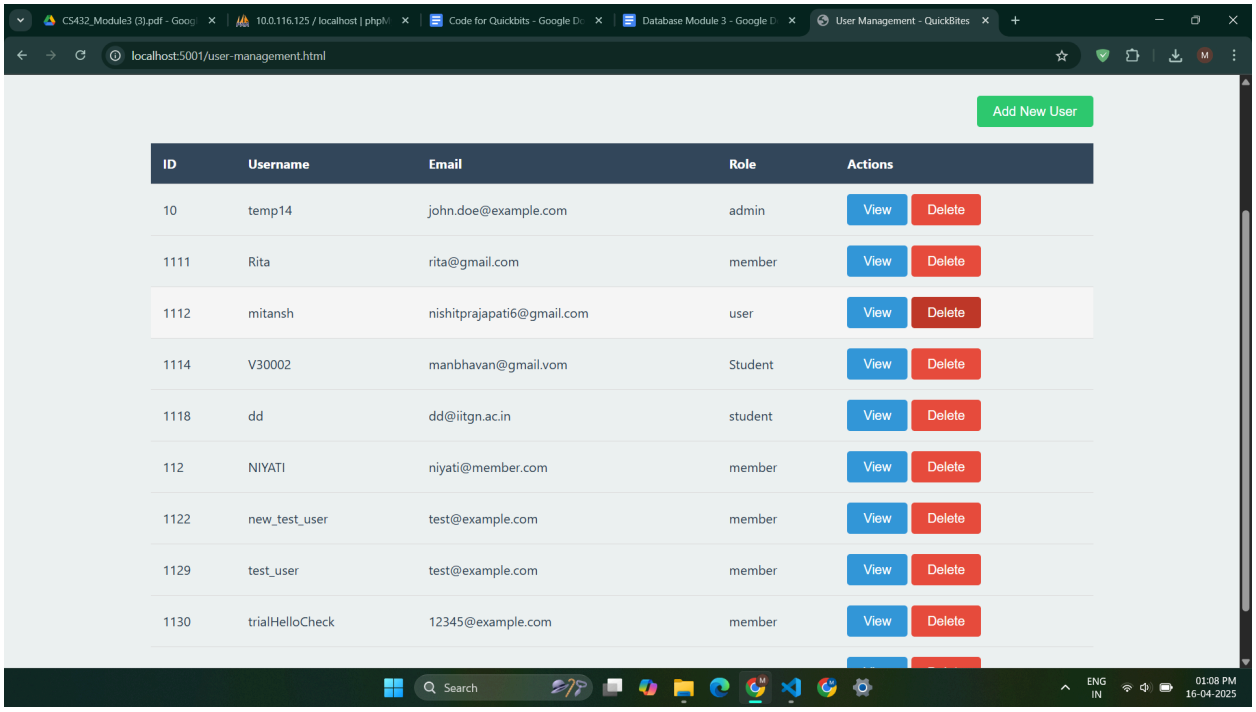
```
if result['group_count'] > 0:
    # Member exists in other groups, only remove specific group mapping
    cursor.execute("""
        DELETE FROM members_group_mapping
        WHERE MemberID = %s AND GroupID = %s
    """, (self.member_id, self.group_id))
```

```
    self.message = {'message': 'Member removed from group'}
    self.logging.info(f"Member {self.member_id} removed from group
{self.group_id}")
```

- Checks if group_count > 0 (meaning member exists in other groups)
- Only deletes the specific group mapping using both MemberID and GroupID
- Keeps the member's records in:
    - members table
    - Login table
    - Other group mappings in members_group_mapping
- Logs the action
- Returns a message indicating the member was only removed from the specific group
- This is the output for deleting the user from the member table using admin

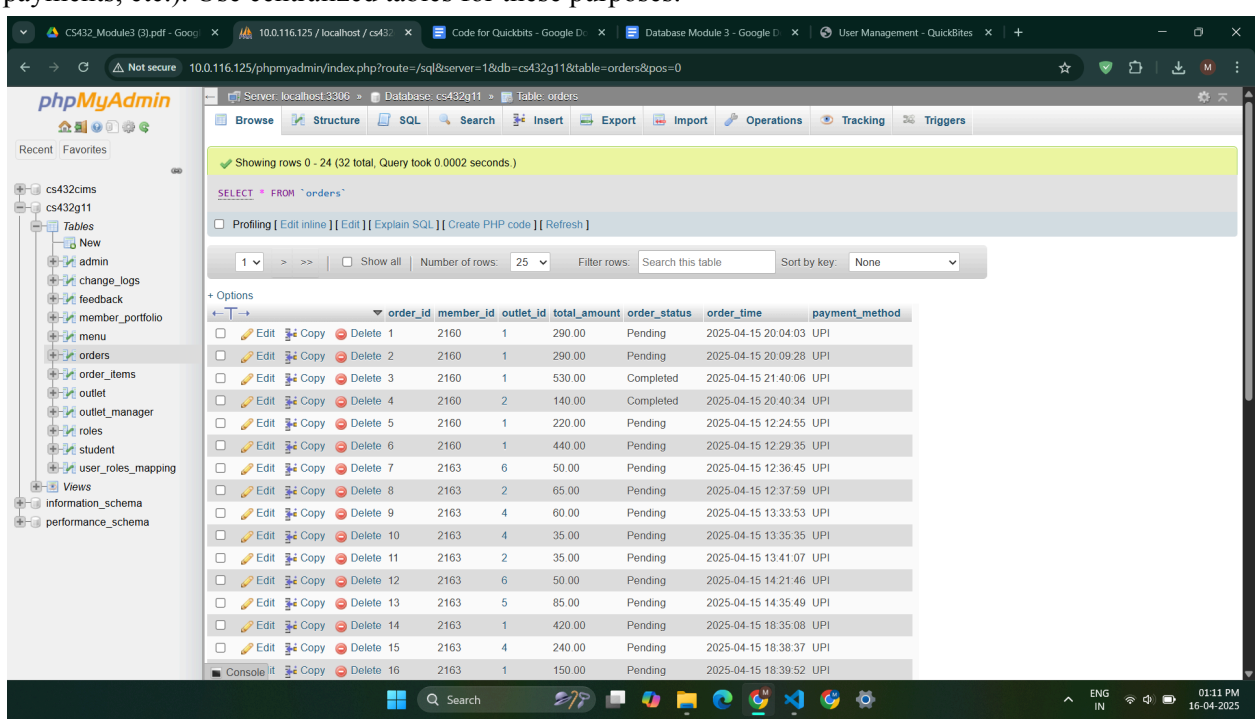


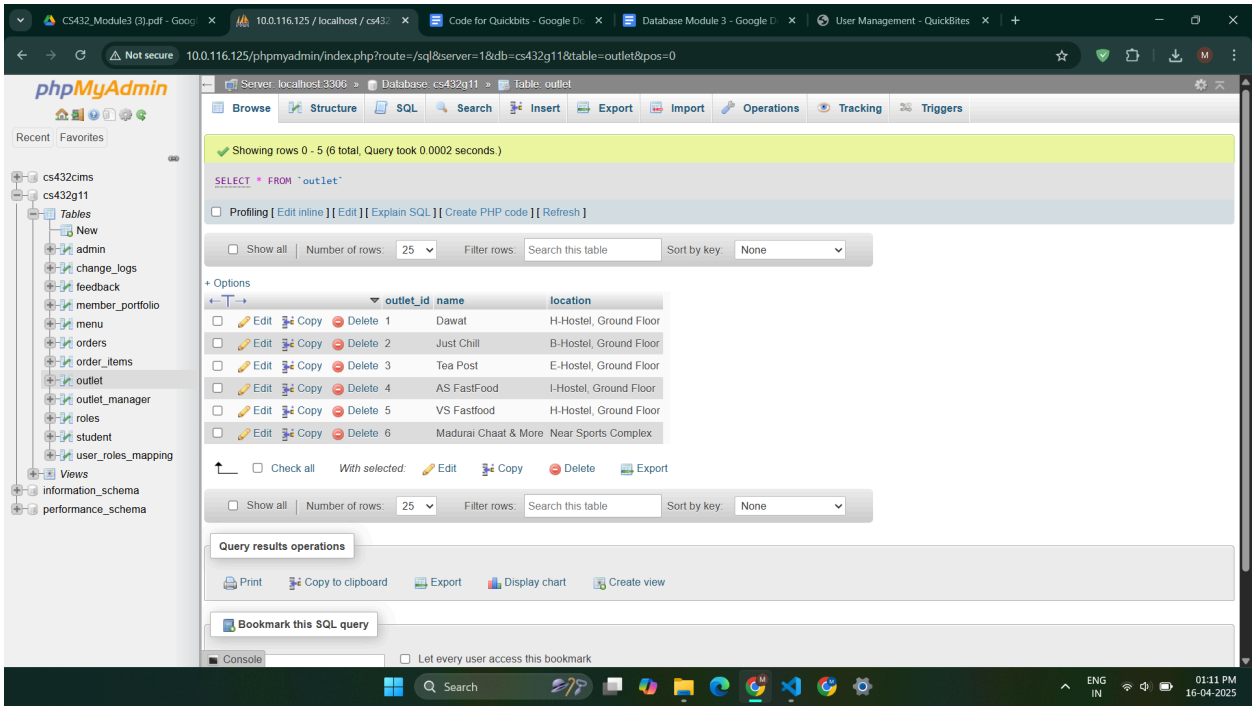
.UI Implementation



Task 4: Database Table Creation

Create the required tables in your project-specific database (CS432 Gx, where x is your group number). Do not duplicate any table that already exists in the CIMS database (e.g., do not create members, payments, etc.). Use centralized tables for these purposes.





Task 5: API Development

Develop web APIs locally on your system to perform operations as per your project requirements (e.g., CRUD operations). Ensure that:

1. Each API call validates the session using the provided centralized API (isValidSession(session)).
2. Admin-level actions are restricted to admin users only.
3. Unauthorized modifications (i.e., direct database writes without session validation) should be logged and flagged.

Session Validation Implementation

We implemented a session validation decorator that wraps all API endpoints to ensure proper authentication and authorization:

```python
def require_valid_session(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        # Get token from Authorization header
        auth_header = request.headers.get('Authorization')
        token = None
        if auth_header and auth_header.startswith('Bearer '):
            token = auth_header.split(' ')[1]

        # If no token in header, try to get from cookies
        if not token:
            token = request.cookies.get('session_token')

        if not token:
            logging.warning(f"Access attempt without token: {request.path}")
            return jsonify({"error": "Authentication required"}), 401

        try:
            # Call centralized session validation API
            response = requests.get(
                f"{API_BASE_URL}/isAuth",
                headers={"Authorization": f"Bearer {token}"},
                cookies={"session_token": token}
            )

            if not response.ok:
                logging.warning(f"Invalid session token: {token[:10]}...")
                return jsonify({"error": "Invalid or expired session"}), 401

            # Extract user data from response
            user_data = response.json()
            user_id = user_data.get('member_id')
            user_role = user_data.get('role', '').lower()

            # Add user data to kwargs for the endpoint function
            kwargs['user_id'] = user_id
            kwargs['user_role'] = user_role
```

Role-Based Access Control

All API endpoints implement role-based access control to ensure that only authorized users can perform specific actions:

```python
@app.route('/api/outlets', methods=['POST'])
@require_valid_session
def create_outlet(user_id=None, user_role=None, **kwargs):
    try:
        # Verify admin role
        if user_role != 'admin':
            logging.warning(f"Unauthorized access attempt: User {user_id} with role {user_role}
            tried to create outlet")
            return jsonify({"error": "Unauthorized. Only admins can create outlets"}), 403

        # Process the request
        data = request.json
        name = data.get('name')
        location = data.get('location')

        # Validate required fields
        if not all([name, location]):
            return jsonify({"error": "Missing required fields"}), 400

        # Insert into database
        conn = get_db_connection(False)  # Use project database
        cursor = conn.cursor()

        query = "INSERT INTO outlet (name, location) VALUES (%s, %s)"
        cursor.execute(query, (name, location))
        conn.commit()

        new_outlet_id = cursor.lastrowid

        cursor.close()
        conn.close()

        # Log successful creation
        logging.info(f"Admin {user_id} created new outlet: {name}")

        return jsonify({
            "message": "Outlet created successfully",
```

Unauthorized Access Logging

We implemented a database access logging system to track and flag unauthorized modifications:

```python
class DatabaseAccessLogger:
    def __init__(self, db_config):
        self.db_config = db_config
        self.setup_logging()

    def setup_logging(self):
        # Configure logging
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(levelname)s - %(message)s',
            handlers=[
                logging.FileHandler('database_access.log'),
                logging.StreamHandler()
            ]
        )

    def log_access(self, user_id, action, table, query, is_authorized):
        """Log database access with authorization status"""
        try:
            conn = mysql.connector.connect(**self.db_config)
            cursor = conn.cursor()

            # Insert log entry
            query = """
            INSERT INTO access_logs
            (user_id, action, table_name, query, timestamp, is_authorized)
            VALUES (%s, %s, %s, %s, NOW(), %s)
            """
            cursor.execute(query, (user_id, action, table, query,
            is_authorized))
            conn.commit()

            # If unauthorized, also log to file with warning level
            if not is_authorized:
                logging.warning(
                    f"UNAUTHORIZED ACCESS: User {user_id} performed {action} on
                    {table} with query: {query}"
```

Task 6: Logging Changes to CIMS Database

Whenever an API call modifies data in the centralized CIMS database (e.g., adding payments or updating member details), ensure logs are printed locally on your system and also logged to the server

- If any transaction is made without proper session validation, it will not appear in the server logs, revealing unauthorized modifications.

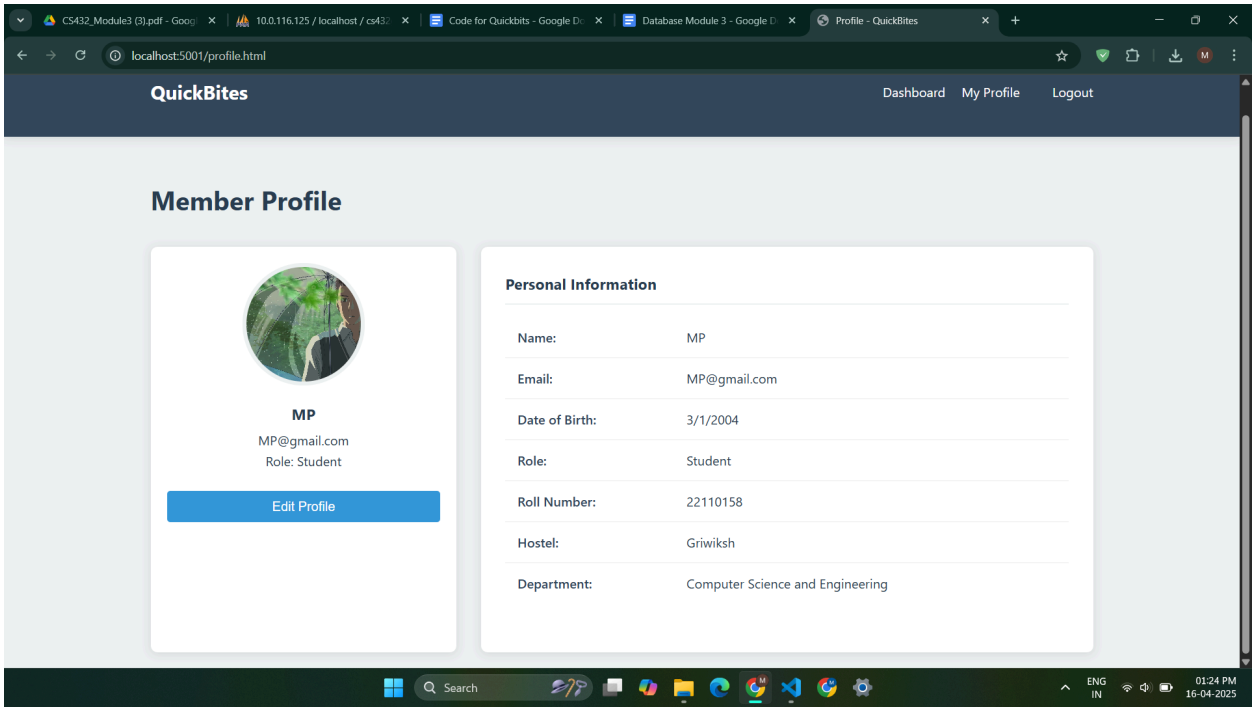
```
app.log
auth.js HTTP/1.1ESC[0m" 304 -
23142  2025-04-16 04:45:03,774 - INFO - Validating token: eyJ0eXAiOi...
23143  2025-04-16 04:45:03,775 - INFO - Authentication successful for user: dipesh
23144  2025-04-16 04:45:03,776 - INFO - 127.0.0.1 - - [16/Apr/2025 04:45:03] "GET /isAuth HTTP/1.1"
       200 -
23145  2025-04-16 04:45:04,104 - INFO - 127.0.0.1 - - [16/Apr/2025 04:45:04] "ESC[33mGET /api/admin/
       undefined HTTP/1.1ESC[0m" 404 -
23146  2025-04-16 04:45:06,565 - INFO - 127.0.0.1 - - [16/Apr/2025 04:45:06] "GET /
       outlet-manager-assignment.html HTTP/1.1" 200 -
23147  2025-04-16 04:45:06,878 - INFO - 127.0.0.1 - - [16/Apr/2025 04:45:06] "ESC[36mGET /static/css/
       styles.css HTTP/1.1ESC[0m" 304 -
23148  2025-04-16 04:45:06,901 - INFO - 127.0.0.1 - - [16/Apr/2025 04:45:06] "ESC[36mGET /static/js/
       auth.js HTTP/1.1ESC[0m" 304 -
23149  2025-04-16 04:45:07,222 - INFO - Validating token: eyJ0eXAiOi...
23150  2025-04-16 04:45:07,222 - INFO - Authentication successful for user: dipesh
23151  2025-04-16 04:45:07,222 - INFO - 127.0.0.1 - - [16/Apr/2025 04:45:07] "GET /isAuth HTTP/1.1"
       200 -
23152  2025-04-16 04:45:07,482 - INFO - Cookie token: None
23153  2025-04-16 04:45:07,482 - INFO - Authorization header: Bearer null
23154  2025-04-16 04:45:07,482 - INFO - Using token from Authorization header: null...
23155  2025-04-16 04:45:07,482 - INFO - All request headers: {'Host': 'localhost:5001',
       'Connection': 'keep-alive', 'Sec-Ch-Ua-Platform': '"Windows"', 'Authorization': 'Bearer
       null', 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
       like Gecko) Chrome/135.0.0.0 Safari/537.36', 'Sec-Ch-Ua': '"Google Chrome";v="135", "Not-A.
       Brand";v="8", "Chromium";v="135"', 'Content-Type': 'application/json', 'Sec-Ch-Ua-Mobile': '?
       0', 'Accept': '*/*', 'Sec-Fetch-Site': 'same-origin', 'Sec-Fetch-Mode': 'cors',
       'Sec-Fetch-Dest': 'empty', 'Referer': 'http://localhost:5001/outlet-manager-assignment.html',
       'Accept-Encoding': 'gzip, deflate, br, zstd', 'Accept-Language': 'en-US,en;q=0.9'}
23156  2025-04-16 04:45:07,497 - INFO - Validating session token: null...
23157  2025-04-16 04:45:07,497 - WARNING - Invalid session token
23158  2025-04-16 04:45:07,497 - WARNING - Invalid token, but allowing access for debugging
23159  2025-04-16 04:45:07,543 - INFO - Cookie token: None
23160  2025-04-16 04:45:07,543 - INFO - Authorization header: Bearer null
23161  2025-04-16 04:45:07,543 - INFO - Using token from Authorization header: null...
23162  2025-04-16 04:45:07,543 - INFO - All request headers: {'Host': 'localhost:5001',
       'Connection': 'keep-alive', 'Sec-Ch-Ua-Platform': '"Windows"', 'Authorization': 'Bearer
       null', 'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
```

Task 7: Member Portfolio Management

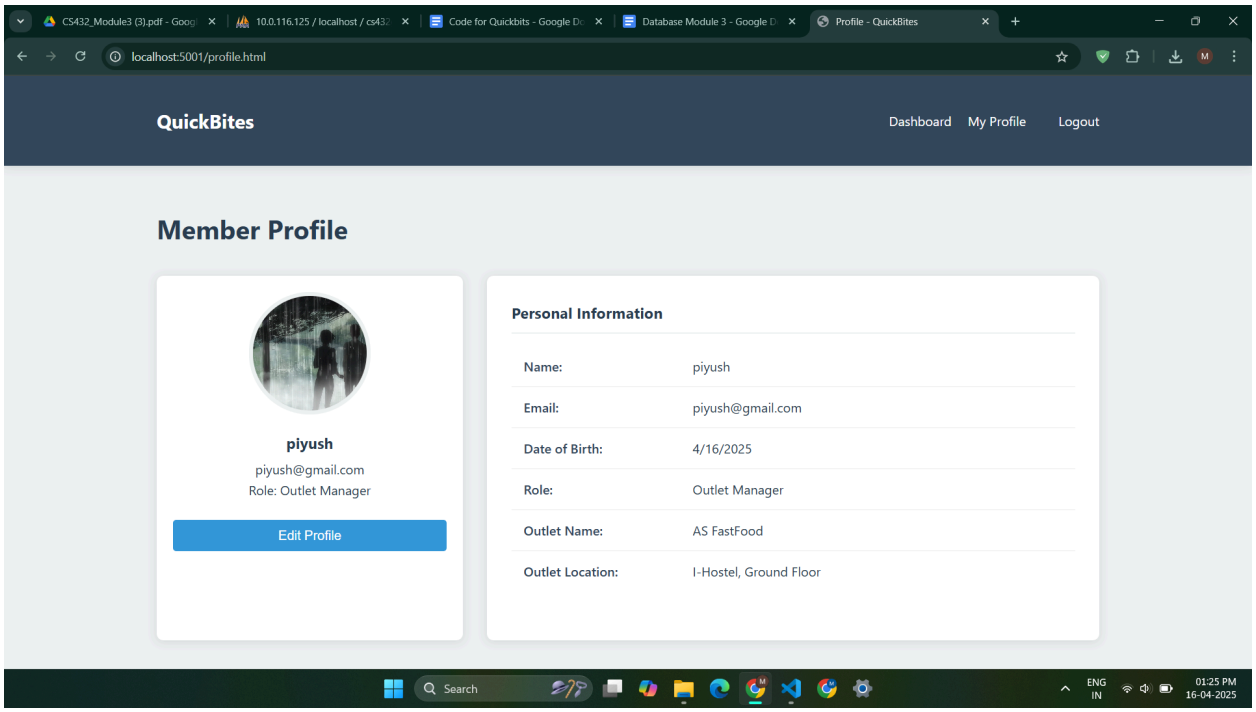
Students must create a portfolio feature for members belonging to their own project. The portfolio should:

- Display relevant details of members within the project.
- Restrict access so that profiles of members from other projects remain hidden.
- Ensure that only authenticated users can view the portfolio.
- Implement appropriate role-based access (e.g., project admins can edit portfolio details, but regular members can only view their own profiles).

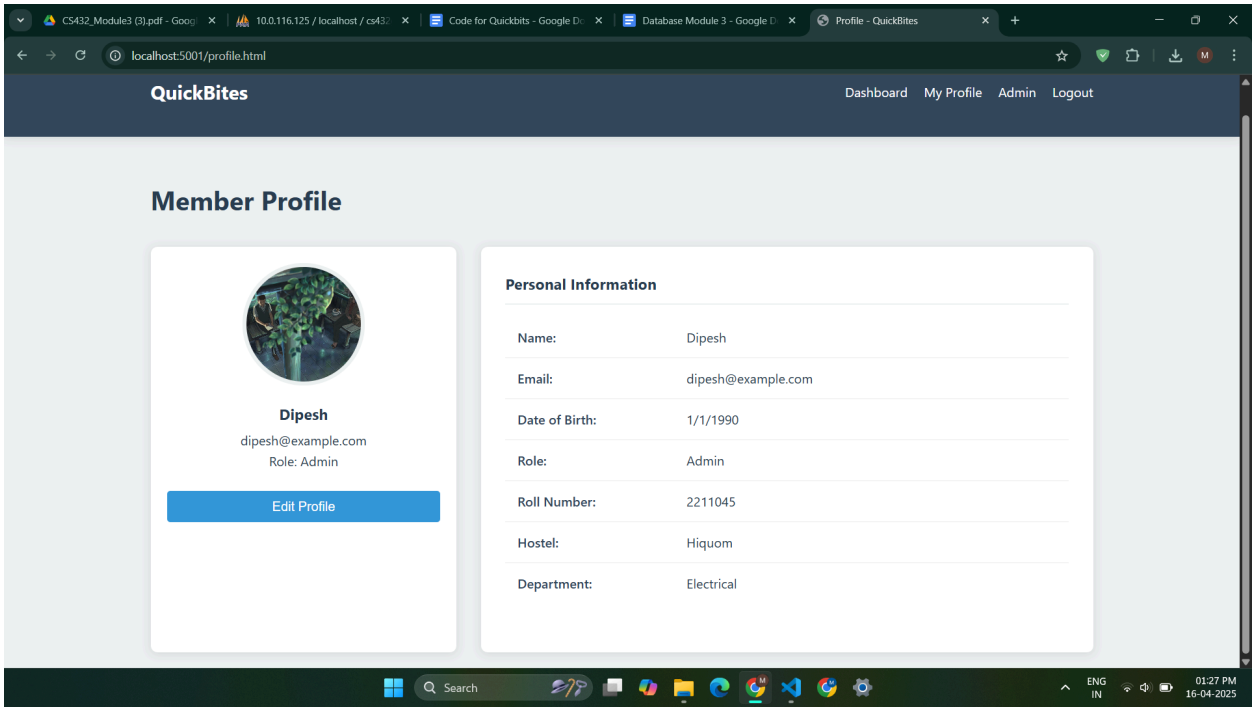
Student Profile

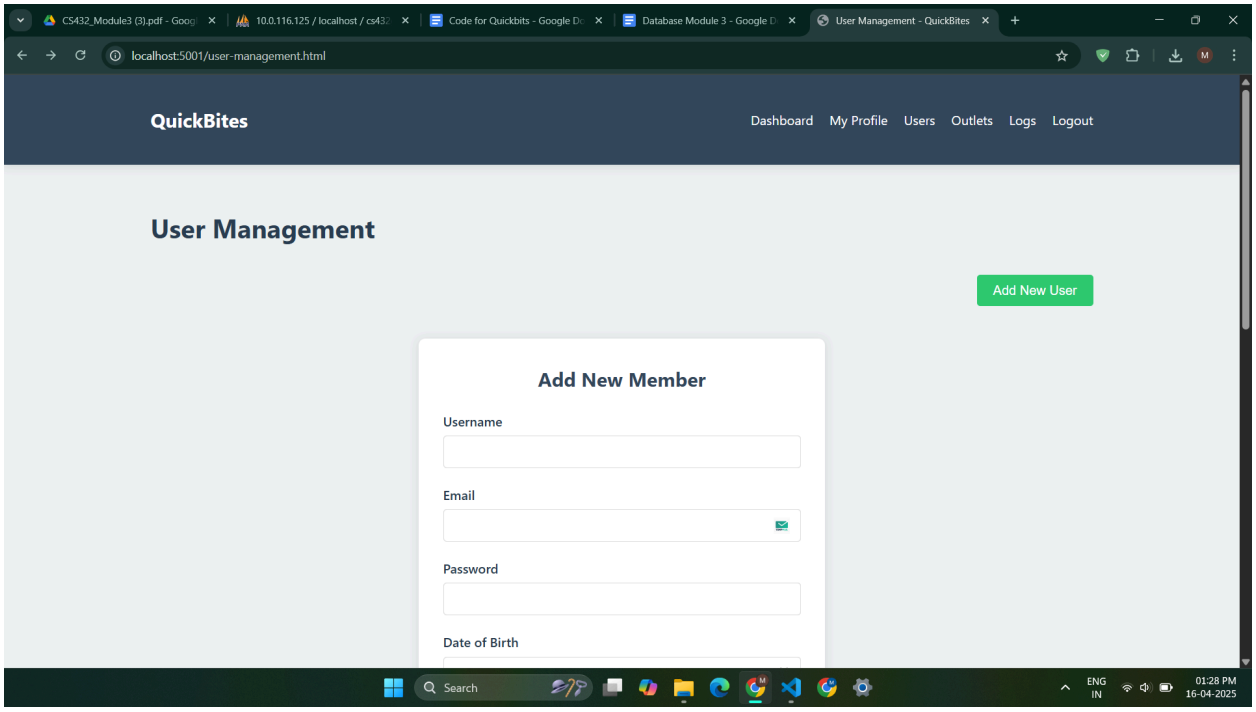


Outlet Manager



Admin


Profile - QuickBites
localhost:5001/profile.html
QuickBites
Dashboard
My Profile
Admin
Logout
Member Profile
Dipesh
dipesh@example.com
Role: Admin
Edit Profile
Personal Information
Name: Dipesh
Email: dipesh@example.com
Date of Birth: 1/1/1990
Role: Admin
Roll Number: 2211045
Hostel: Hiquom
Department: Electrical


User Management - QuickBites
localhost:5001/user-management.html
QuickBites
Dashboard
My Profile
Users
Outlets
Logs
Logout
User Management
Add New User
Add New Member
Username
Email
Password
Date of Birth

Design Principles

- Separation of Concerns: Created tables specific to food ordering functionality while leveraging centralized tables for common data
- Normalization: Applied database normalization principles to minimize redundancy and maintain data integrity
- Referential Integrity: Implemented foreign key constraints to ensure data consistency across tables
- Role-Based Structure: Designed tables to support the three main user roles (student, outlet manager, admin)

Key Tables

- outlet: Stores information about campus food outlets
- outlet_manager: Maps managers to their assigned outlets (one-to-one relationship)
- menu: Contains food items available at each outlet
- orders: Tracks order information including status and payment details
- order_items: Stores individual items within each order

Views

Created views to simplify common queries:

- menu_with_outlet: Combines menu items with outlet information
- order_details: Provides comprehensive order information with customer and outlet details

CIMS Database Integration

Integration Strategy

- Reference, Don't Duplicate: Used foreign keys to reference CIMS tables rather than duplicating data
- Centralized Authentication: Leveraged CIMS authentication system for user management
- Consistent Data Model: Ensured data model consistency between project-specific and CIMS tables

Key Integration Points

- User Management: Referenced the CIMS members table for user information
- Authentication: Used the CIMS Login table for credentials and session management
- Profile Images: Utilized the CIMS images table for user profile pictures

Cross-Database Queries

- Implemented JOIN operations across databases to retrieve comprehensive information
- Used consistent data types and naming conventions to ensure compatibility
- Created views to abstract the complexity of cross-database queries

Session Validation and Security Implementation

Session Validation

- Centralized Validation: Used the isValidSession API to validate all requests
- Decorator Pattern: Implemented a session validation decorator that wraps all API endpoints
- Token Management: Supported both header-based and cookie-based token validation

Role-Based Access Control

- Role Verification: Checked user roles before allowing access to protected endpoints
- Contextual Authorization: For outlet managers, verified they only access their assigned outlet's data
- Admin Privileges: Implemented special checks for admin-level operations

Preventing Data Leaks

- Input Validation: Validated all input parameters to prevent injection attacks
- Output Filtering: Ensured sensitive data is not included in API responses
- Error Handling: Implemented proper error handling to avoid exposing system details

Security Logging

- Comprehensive Logging: Recorded all API calls with user ID, role, and action details
- Unauthorized Access Detection: Flagged and logged unauthorized access attempts
- Audit Trail: Maintained an audit trail of all database modifications for accountability

Conclusion

The QuickBites project successfully implements a secure, integrated food ordering system that leverages the centralized CIMS database while maintaining proper separation of concerns. The design ensures data integrity, prevents unauthorized access, and provides a solid foundation for future enhancements.