Indian Institute of Technology Gandhinagar

Report: Assignment 4

Authors

22110050	Birudugadda Srivibhav	20110106	Shriyash Mandavekar
22110260	Vamsi Chaturvedula	20110071	Haikoo Khandor
22110124	Sriman reddy	20110104	Madhav Kanda
22110162	Nikhilesh Myanapuri	19110009	Deep Thakkar

Under the Supervision of
Prof. Mayank Singh

April 16, 2024

Github repo link: https://github.com/Shriyash1234/Food_Delivery_System
Video Link: [Website Video]

3.1.1 Implementing the suggestions received during feedback:

Feedback 1 - From Restaurant

Among the feedback that we received from various stakeholders, such as the Just Chill Cafe owner, possible users, etc. We chose to implement the following suggestions:

- Adding the bank details of the restaurant to the database - One of the stakeholders suggested that the bank account details like account number, IFSC code, and bank name be included on the restaurant page, which was accessible through restaurant credentials only.

- Adding a balance earned section of a particular restaurant in the database - Another unique suggestion that was made by one of the stakeholders was that there ought to be a section that enables the restaurant owner or the manager (the one with the credentials to log in) to keep a tab of the profits made through the app. We created a column balance_earned, which could take integer values and would store the details. It can only be updated by the restaurant.

- Adding a review of the restaurant section in the database - another suggestion we received was a review section for the restaurant, which we added.

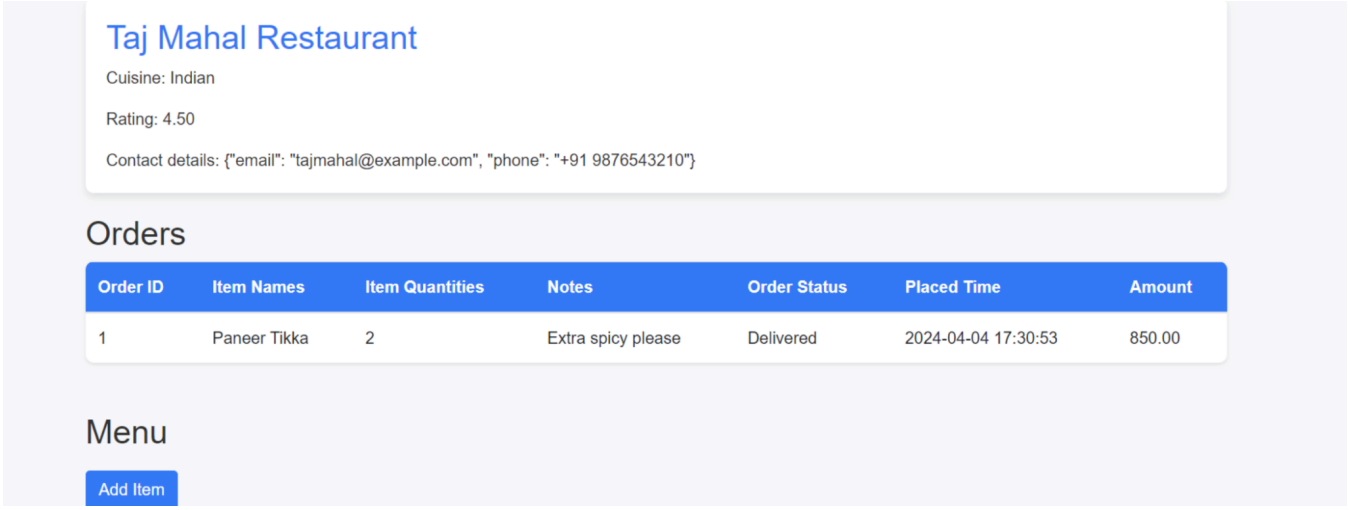
All these changes need to be made in the database's restaurant table and must be visible on our webpage.

Before implementing the suggestions:

Our SQL script before implementing the suggestions looked like the following :

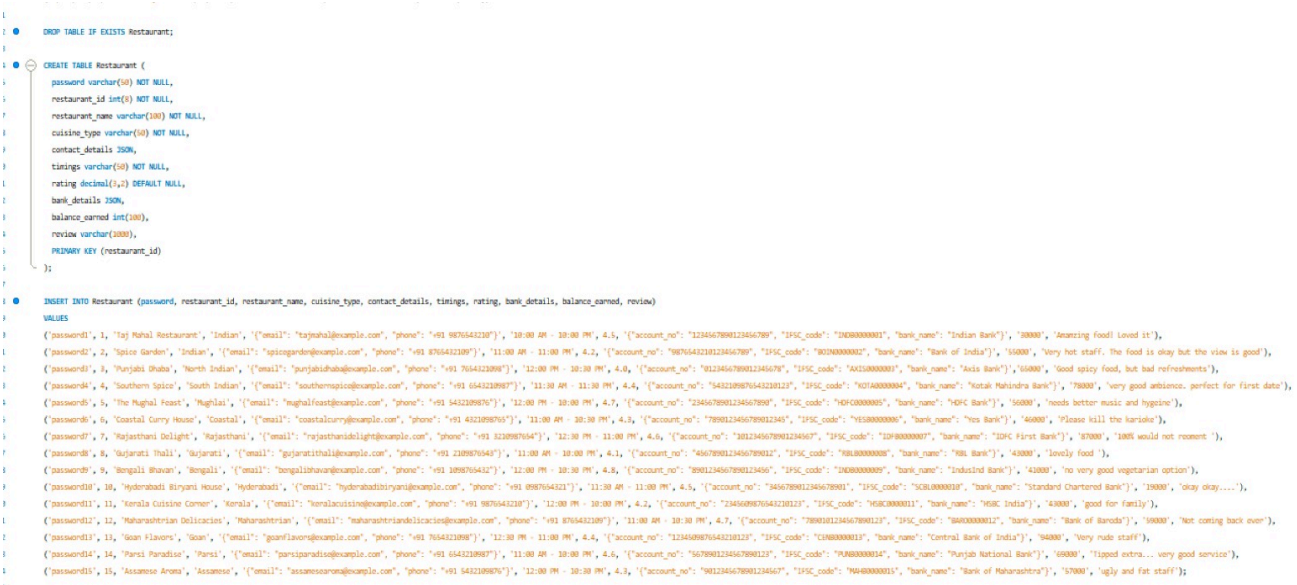
```
DROP TABLE IF EXISTS Restaurant;

CREATE TABLE Restaurant (
  password varchar(50) NOT NULL,
  restaurant_id int(8) NOT NULL,
  restaurant_name varchar(100) NOT NULL,
  cuisine_type varchar(50) NOT NULL,
  contact_details JSON,
  timings varchar(50) NOT NULL,
  rating decimal(3,2) DEFAULT NULL,
  PRIMARY KEY (restaurant_id)
);

INSERT INTO Restaurant (password, restaurant_id, restaurant_name, cuisine_type, contact_details, timings, rating)
VALUES
('password1', 1, 'Taj Mahal Restaurant', 'Indian', '{"email": "tajmahal@example.com", "phone": "+91 9876543210"}', '10:00 AM - 10:00 PM', 4.5),
('password2', 2, 'Spice Garden', 'Indian', '{"email": "spicegarden@example.com", "phone": "+91 8765432109"}', '11:00 AM - 11:00 PM', 4.2),
('password3', 3, 'Punjabi Dhaba', 'North Indian', '{"email": "punjabidhaba@example.com", "phone": "+91 7654321098"}', '12:00 PM - 10:30 PM', 4.0),
('password4', 4, 'Southern Spice', 'South Indian', '{"email": "southernspice@example.com", "phone": "+91 6543210987"}', '11:30 AM - 11:30 PM', 4.4),
('password5', 5, 'The Mughal Feast', 'Mughlai', '{"email": "mughalfeast@example.com", "phone": "+91 5432109876"}', '12:00 PM - 10:00 PM', 4.7),
('password6', 6, 'Coastal Curry House', 'Coastal', '{"email": "coastalcurry@example.com", "phone": "+91 4321098765"}', '11:00 AM - 10:30 PM', 4.3),
('password7', 7, 'Rajasthani Delight', 'Rajasthani', '{"email": "rajasthanidelight@example.com", "phone": "+91 3210987654"}', '12:30 PM - 11:00 PM', 4.6),
('password8', 8, 'Gujarati Thali', 'Gujarati', '{"email": "gujaratithali@example.com", "phone": "+91 2109876543"}', '11:00 AM - 10:00 PM', 4.1),
('password9', 9, 'Bengali Bhavan', 'Bengali', '{"email": "bengalibhavan@example.com", "phone": "+91 1098765432"}', '12:00 PM - 10:30 PM', 4.8),
('password10', 10, 'Hyderabadi Biryani House', 'Hyderabadi', '{"email": "hyderabadibiryani@example.com", "phone": "+91 0987654321"}', '11:30 AM - 11:00 PM', 4.5),
('password11', 11, 'Kerala Cuisine Corner', 'Kerala', '{"email": "keralacuisine@example.com", "phone": "+91 9876543210"}', '12:00 PM - 10:00 PM', 4.2),
('password12', 12, 'Maharashtrian Delicacies', 'Maharashtrian', '{"email": "maharashtriandelicacies@example.com", "phone": "+91 8765432109"}', '11:00 AM - 10:30 PM', 4.7),
('password13', 13, 'Goan Flavors', 'Goan', '{"email": "goanflavors@example.com", "phone": "+91 7654321098"}', '12:30 PM - 11:00 PM', 4.4),
('password14', 14, 'Parsi Paradise', 'Parsi', '{"email": "parsiparadise@example.com", "phone": "+91 6543210987"}', '11:00 AM - 10:00 PM', 4.6),
('password15', 15, 'Assamese Aroma', 'Assamese', '{"email": "assamesearoma@example.com", "phone": "+91 5432109876"}', '12:00 PM - 10:30 PM', 4.3);
```

Our web application looks like this:



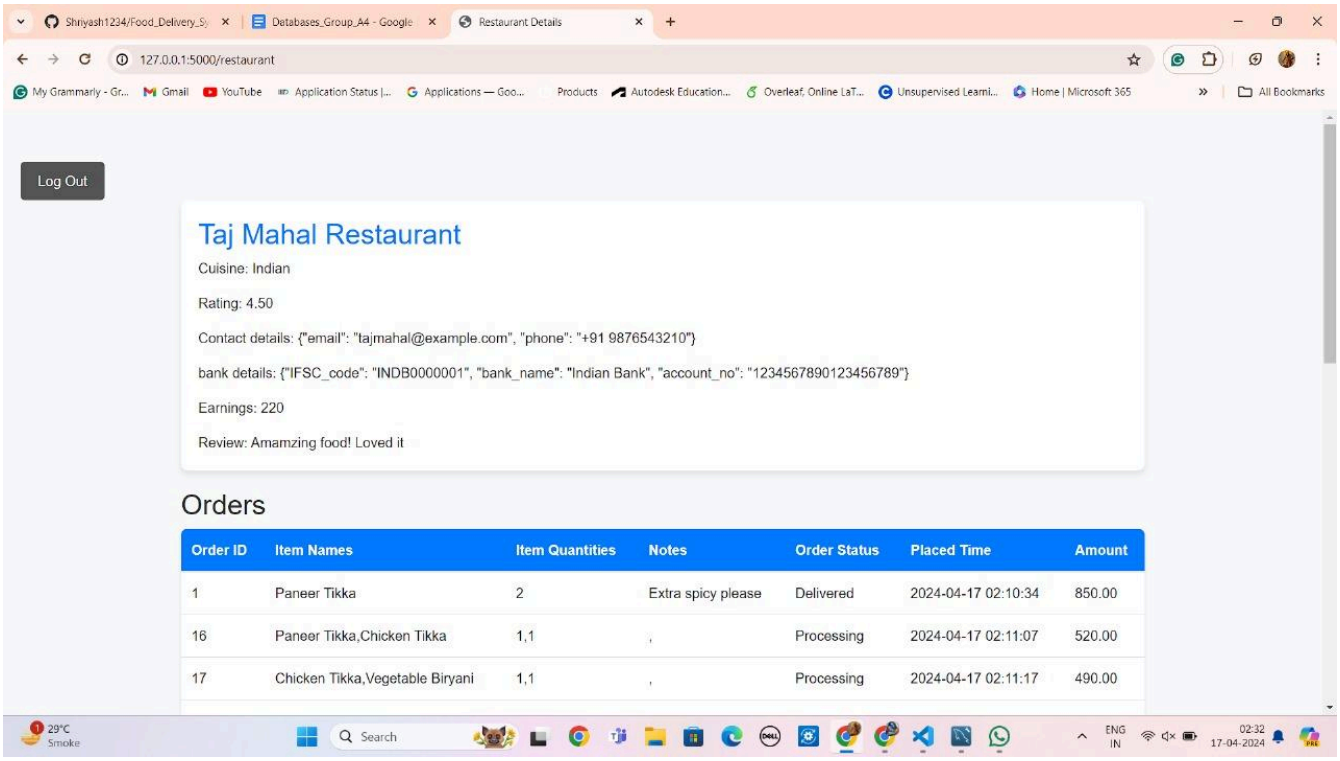
After implementing the suggestions:

Our SQL script, after implementing the suggestions, looks like the following:

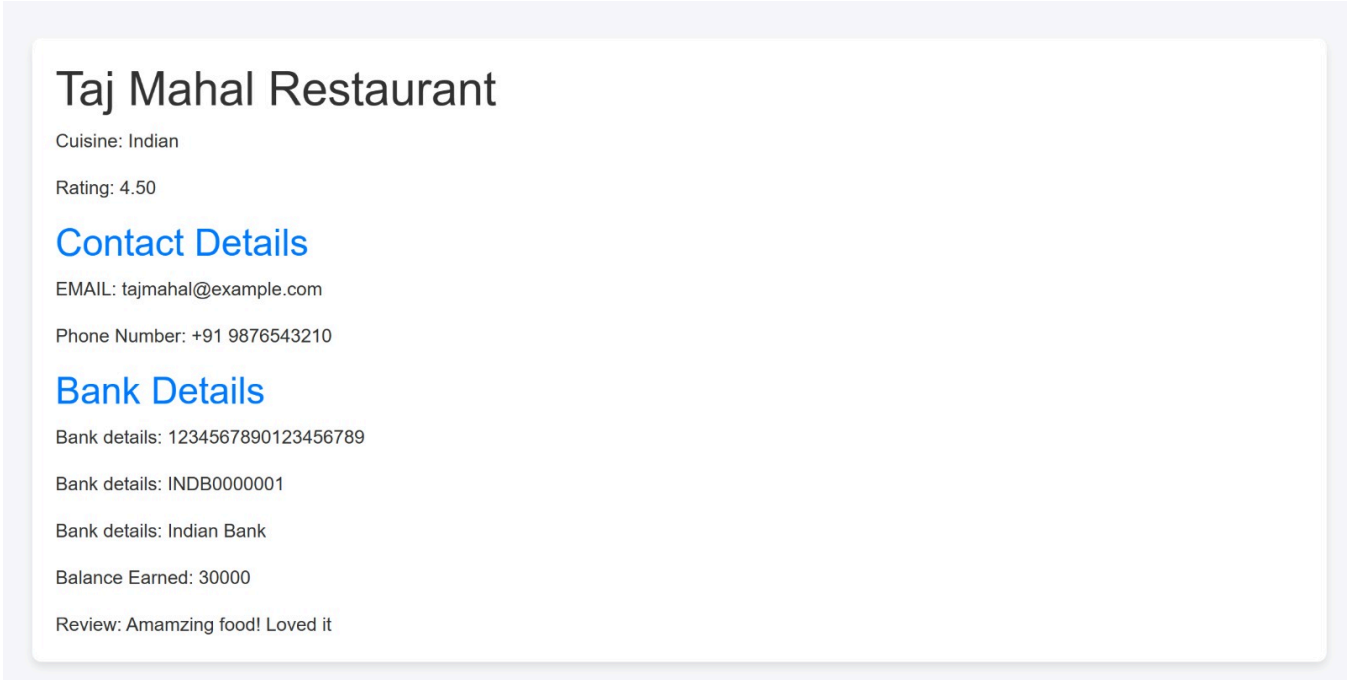


```
DROP TABLE IF EXISTS Restaurant;

CREATE TABLE Restaurant (
 password varchar(50) NOT NULL,
 restaurant_id int(8) NOT NULL,
 restaurant_name varchar(100) NOT NULL,
 cuisine_type varchar(50) NOT NULL,
 contact_details JSON,
 timings varchar(50) NOT NULL,
 rating decimal(3,2) DEFAULT NULL,
 bank_details JSON,
 balance_earned int(100),
 review varchar(1000),
 PRIMARY KEY (restaurant_id)
 );

INSERT INTO Restaurant (password, restaurant_id, restaurant_name, cuisine_type, contact_details, timings, rating, bank_details, balance_earned, review)
VALUES
('password1', 1, 'Taj Mahal Restaurant', 'Indian', '{"email": "tajmahal@example.com", "phone": "+91 9876543210"}', '10:00 AM - 10:00 PM', 4.5, '{"account_no": "123456789012345678", "IFSC_code": "INDB0000001", "bank_name": "Indian Bank"}', '30000', 'Amazing food! Loved it'),
('password2', 2, 'Spice Garden', 'Indian', '{"email": "spicegarden@example.com", "phone": "+91 8765432109"}', '11:00 AM - 11:00 PM', 4.2, '{"account_no": "987654321012345678", "IFSC_code": "BOIN0000002", "bank_name": "Bank of India"}', '65000', 'Very hot staff. The food is okay but the view is good'),
('password3', 3, 'Punjabi Dhaba', 'North Indian', '{"email": "punjabidhaba@example.com", "phone": "+91 7654321098"}', '12:00 PM - 10:30 PM', 4.0, '{"account_no": "012345678901234678", "IFSC_code": "AXIS0000003", "bank_name": "Axis Bank"}','65000', 'Good spicy food, but bad refreshments'),
('password4', 4, 'Southern Spice', 'South Indian', '{"email": "southernspice@example.com", "phone": "+91 6543210987"}', '11:30 AM - 11:30 PM', 4.4, '{"account_no": "543210987654321123", "IFSC_code": "KOTA0000004", "bank_name": "Kotak Mahindra Bank"}', '78000', 'very good ambience. perfect for first date'),
('password5', 5, 'The Mughal Feast', 'Mughlai', '{"email": "mughalfeast@example.com", "phone": "+91 5432109876"}', '12:00 PM - 10:00 PM', 4.7, '{"account_no": "234567890123456789", "IFSC_code": "HDFC0000005", "bank_name": "HDFC Bank"}', '56000', 'needs better music and hygeine'),
('password6', 6, 'Coastal Curry House', 'Coastal', '{"email": "coastalcurry@example.com", "phone": "+91 4321098765"}', '11:00 AM - 10:30 PM', 4.8, '{"account_no": "789012345678901345", "IFSC_code": "YES0000006", "bank_name": "Yes Bank"}', '46000', 'Please kill the karioke'),
('password7', 7, 'Rajasthani Delight', 'Rajasthani', '{"email": "rajasthanidelight@example.com", "phone": "+91 3210987654"}', '12:30 PM - 11:00 PM', 4.6, '{"account_no": "101234567890123457", "IFSC_code": "IDFB0000007", "bank_name": "IDFC First Bank"}', '87000', '100% would not reomemt '),
('password8', 8, 'Gujarati Thali', 'Gujarati', '{"email": "gujaratithali@example.com", "phone": "+91 2109876543"}', '11:00 AM - 10:00 PM', 4.1, '{"account_no": "456789012345678912", "IFSC_code": "KBL0000008", "bank_name": "KBL Bank"}', '43000', 'lovely food '),
('password9', 9, 'Bengali Bhavan', 'Bengali', '{"email": "bengalibhavan@example.com", "phone": "+91 1098765432"}', '12:00 PM - 10:30 PM', 4.8, '{"account_no": "890123456789012456", "IFSC_code": "IND0000009", "bank_name": "IndusInd Bank"}', '41000', 'no very good vegetarian option'),
('password10', 10, 'Hyderabadi Biryani House', 'Hyderabadi', '{"email": "hyderabadibiryani@example.com", "phone": "+91 0987654321"}', '11:30 AM - 11:00 PM', 4.5, '{"account_no": "345678901234567891", "IFSC_code": "SCBL0000010", "bank_name": "Standard Chartered Bank"}', '19000', 'okay okay....'),
('password11', 11, 'Kerala Cuisine Corner', 'Kerala', '{"email": "keralacuisine@example.com", "phone": "+91 9876543210"}', '12:00 PM - 10:00 PM', 4.2, '{"account_no": "234560987654320123", "IFSC_code": "HSBC0000011", "bank_name": "HSBC India"}', '43000', 'good for family'),
('password12', 12, 'Maharashtrian Delicacies', 'Maharashtrian', '{"email": "maharashtriandelicacies@example.com", "phone": "+91 8765432109"}', '11:00 AM - 10:30 PM', 4.7, '{"account_no": "789012123456789123", "IFSC_code": "BARB0000012", "bank_name": "Bank of Baroda"}', '59000', 'Not coming back ever'),
('password13', 13, 'Goan Flavors', 'Goan', '{"email": "goanflavors@example.com", "phone": "+91 7654321098"}', '12:30 PM - 11:00 PM', 4.4, '{"account_no": "123450987654320123", "IFSC_code": "CNRB0000013", "bank_name": "Central Bank of India"}', '94000', 'Very rude staff'),
('password14', 14, 'Parsi Paradise', 'Parsi', '{"email": "parsiparadise@example.com", "phone": "+91 6543210987"}', '11:00 AM - 10:00 PM', 4.6, '{"account_no": "567890123456789123", "IFSC_code": "PUNB0000014", "bank_name": "Punjab National Bank"}', '69000', 'Tipped extra... very good service'),
('password15', 15, 'Assamese Aroma', 'Assamese', '{"email": "assamesearoma@example.com", "phone": "+91 5432109876"}', '12:00 PM - 10:30 PM', 4.3, '{"account_no": "901234567890123457", "IFSC_code": "MAHB0000015", "bank_name": "Bank of Maharashtra"}', '57000', 'ugly and fat staff');
```

Our web application looks:



After the implementation, there are no major concerns among the stakeholders. A small update suggested was that on the view for the restaurant, the data for contact details and bank details were not represented properly. It was updated for ease of readability as shown:



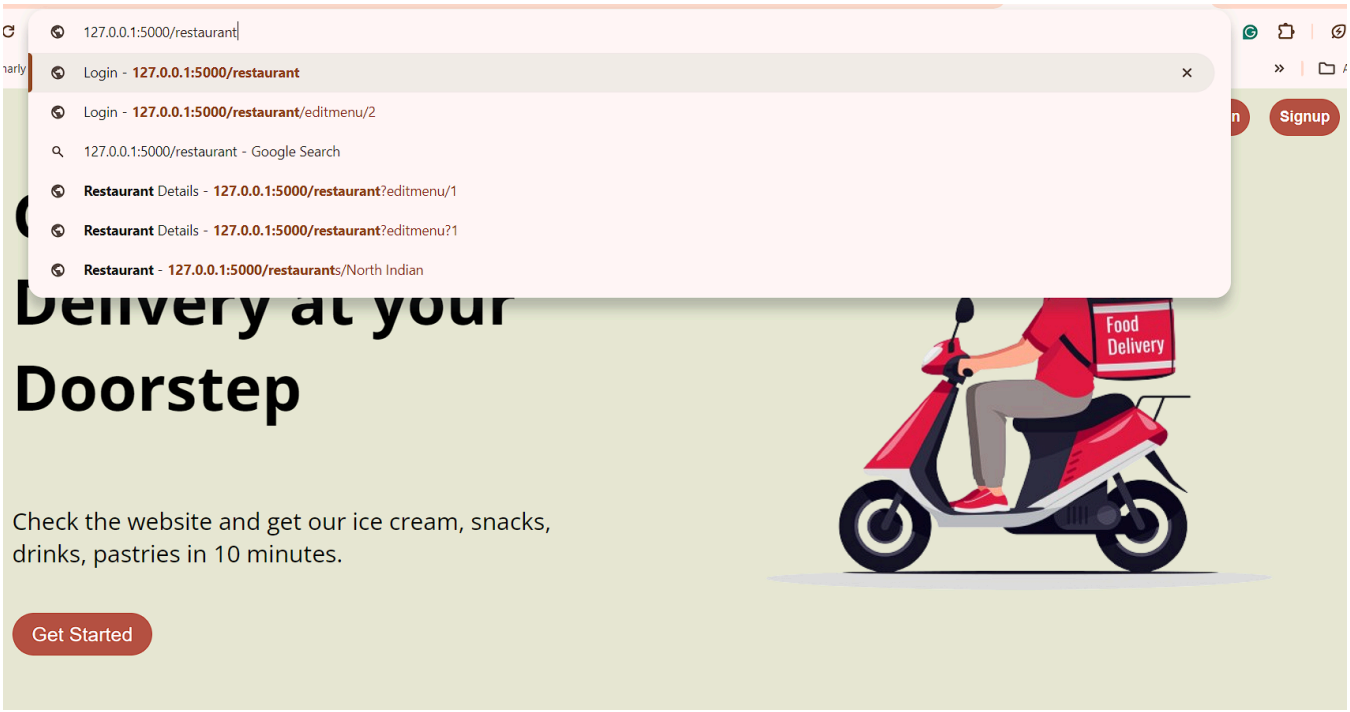
Feedback 2: From User

The users have found some minor bugs in the website, where users were able to bypass the login page by altering the URL from "http://127.0.0.1:5000/" to "http://127.0.0.1:5000/restaurant,"

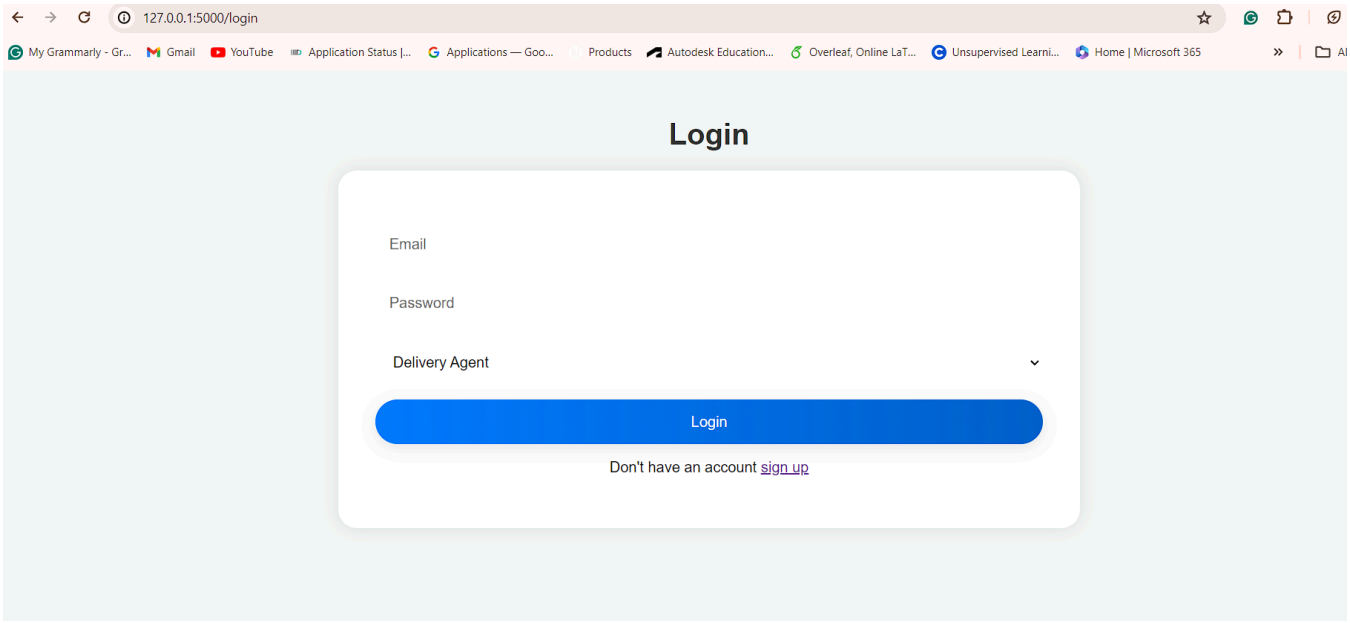
corrective measures have been implemented. Specifically, new functions have been introduced to enforce login authentication when accessing different sections of the website.

Three distinct login functions have been integrated: "login_cust," "login_rest," and "login_agent." Each function corresponds to a specific user role, ensuring that appropriate authentication is required when navigating to relevant pages. By implementing these functions, access to sensitive sections, such as the restaurant owner page, is now properly restricted, requiring users to authenticate via the login page before proceeding further.

```python
# Decorator function to check if user is logged in
def login_rest(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'restaurant_ID' not in session:
            # Redirect to login page if user is not logged in
            return redirect(url_for('googlelogin'))
        return f(*args, **kwargs)
    return decorated_function

def login_cust(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'customer_id' not in session:
            # Redirect to login page if user is not logged in
            return redirect(url_for('googlelogin'))
        return f(*args, **kwargs)
    return decorated_function

def login_agent(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'agent_ID' not in session:
            # Redirect to login page if user is not logged in
            return redirect(url_for('googlelogin'))
        return f(*args, **kwargs)
    return decorated_function
```



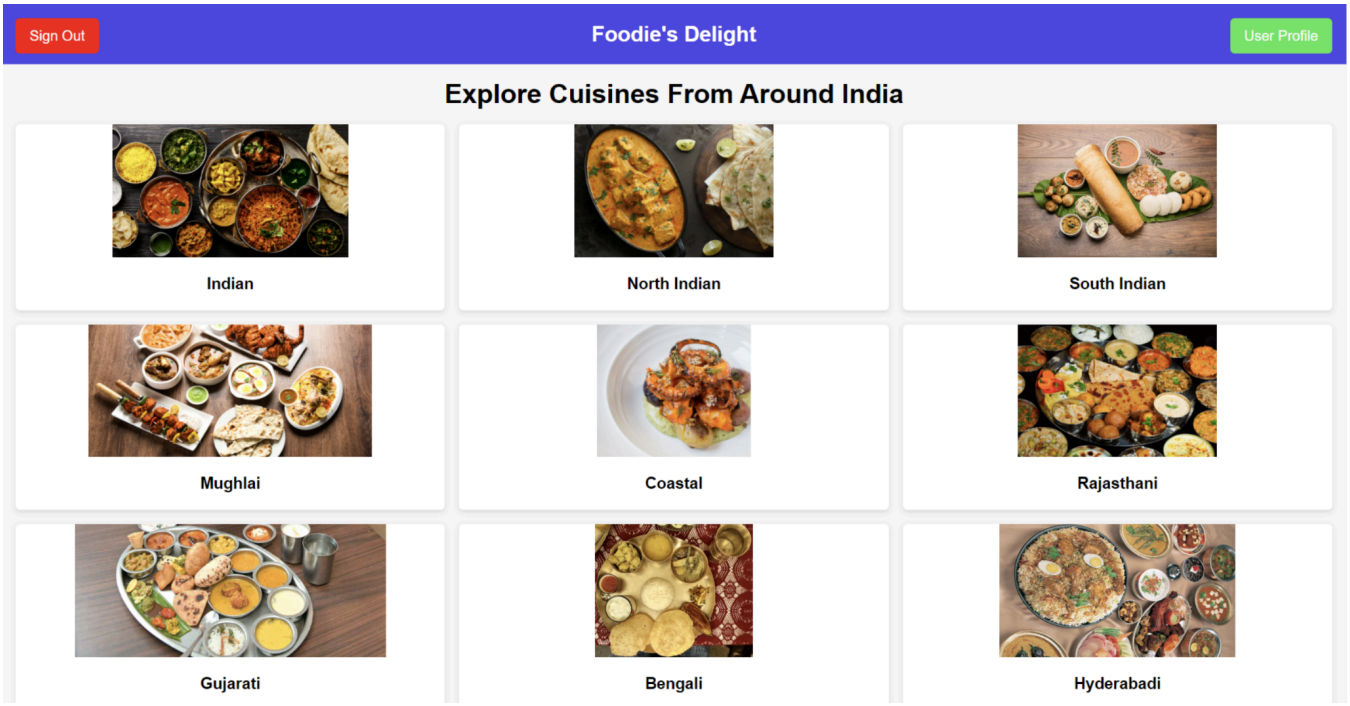
As you can see in the above picture, when typed "http://127.0.0.1:5000/restaurant", we are redirected to the login page:



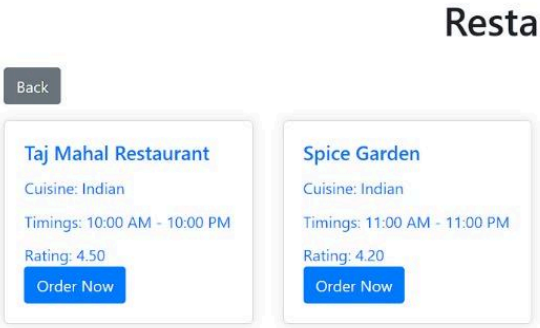
3.1.2 Views and Privileges

Customer:

- Place an order by viewing various cuisines, allowing them to read data from the cuisine-type table.

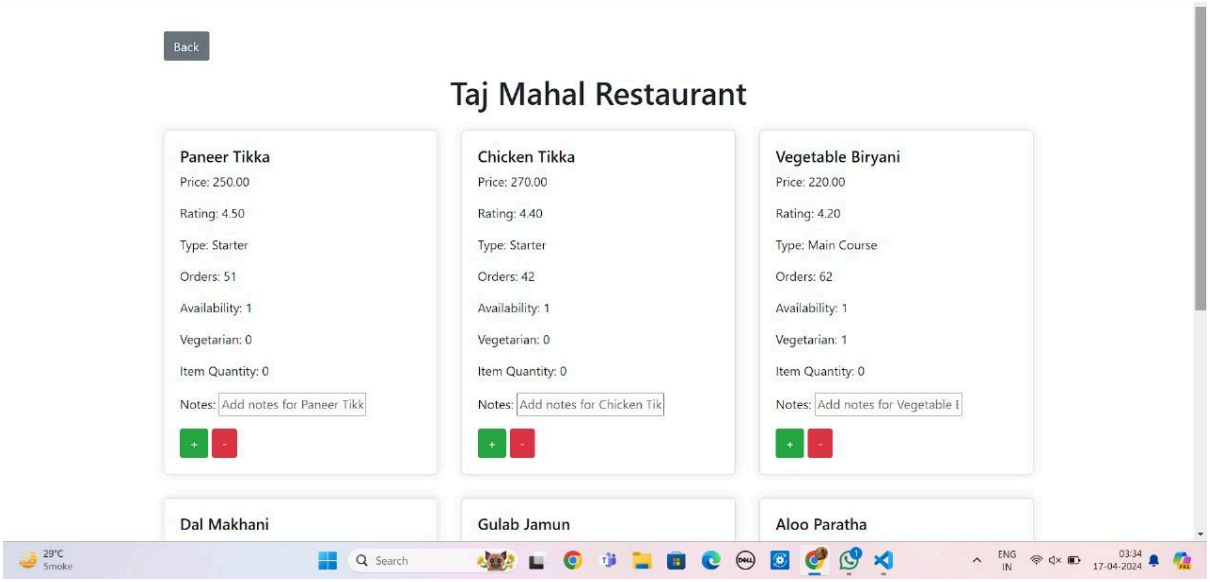


- The customer then can view and select the restaurants based on the cuisine chosen.

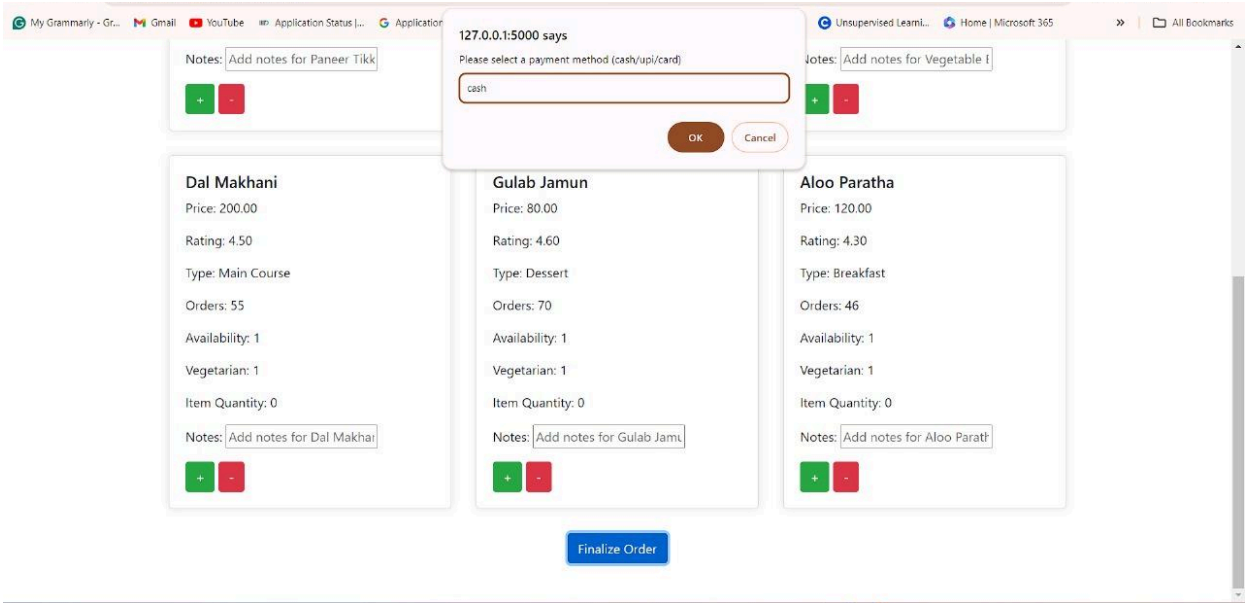


- The customers can view the menu of the restaurant and will be able to view all the required information like type, rating, number of orders, and availability. The customers have the

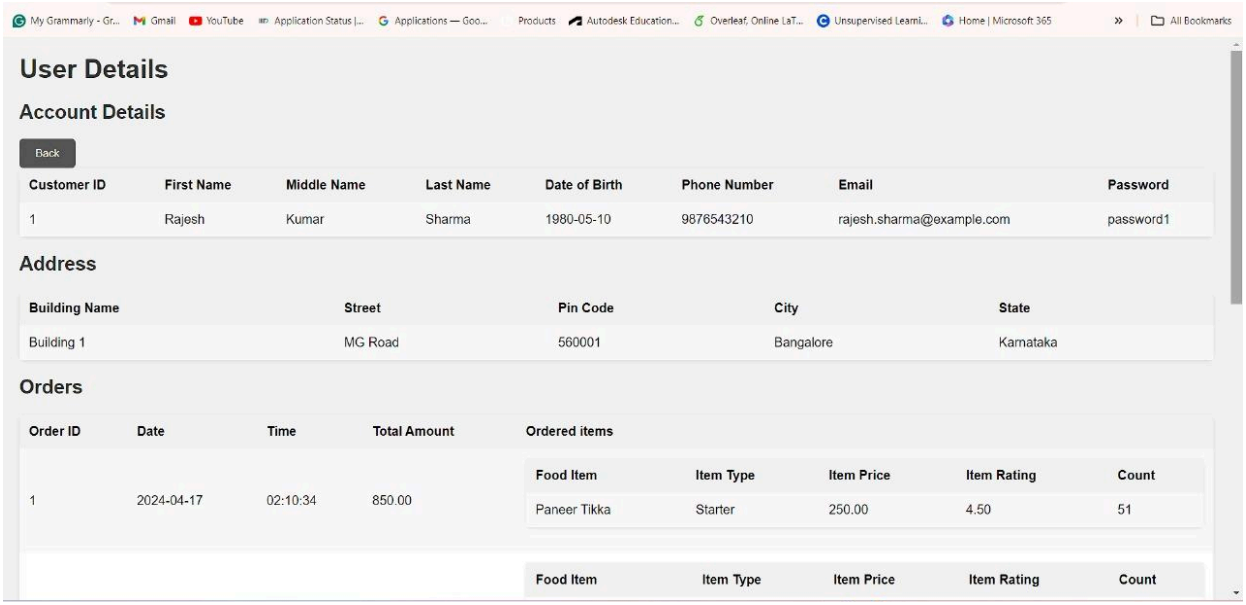
privilege of choosing the item quantity and ordering multiple items, and they can also specify special requirements through the note.



- The customer also has the privilege of choosing the mode of payment as well.



- Also, the customer has the privilege of viewing the complete account information, address, and list of orders.



User Details

Account Details

Back

Customer ID	First Name	Middle Name	Last Name	Date of Birth	Phone Number	Email	Password
1	Rajesh	Kumar	Sharma	1980-05-10	9876543210	rajesh.sharma@example.com	password1

Address

Building Name	Street	Pin Code	City	State
Building 1	MG Road	560001	Bangalore	Karnataka

Orders

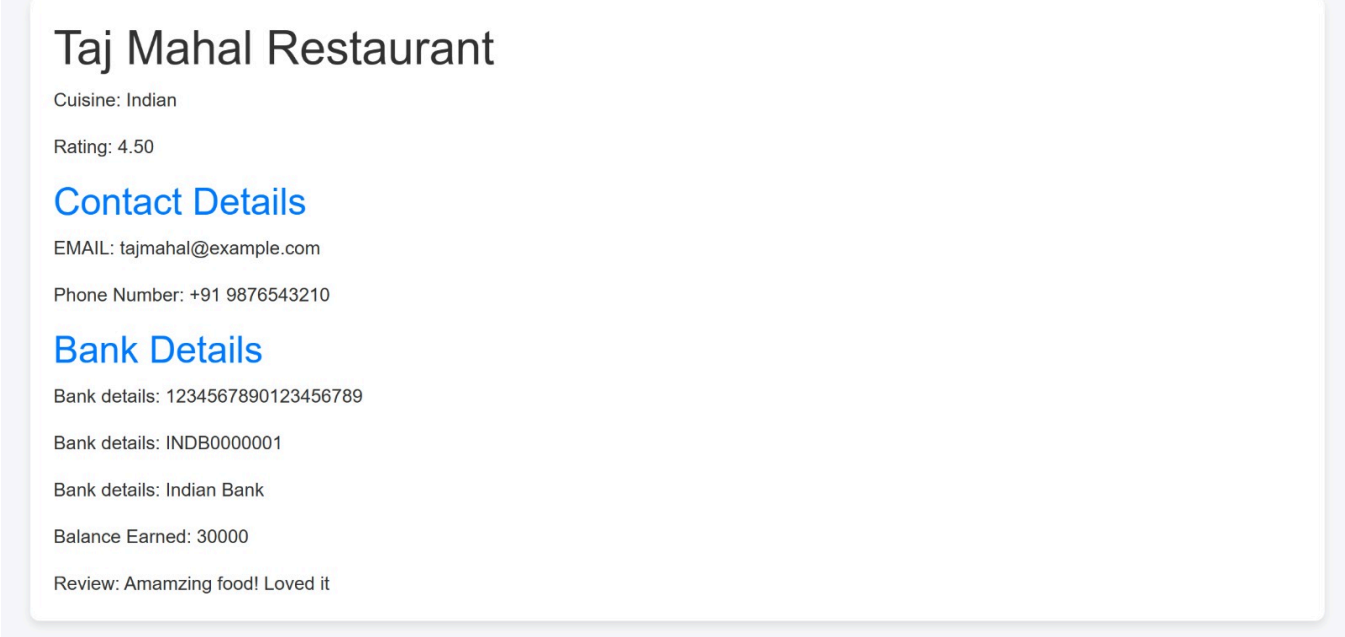
Order ID	Date	Time	Total Amount	Ordered items
1	2024-04-17	02:10:34	850.00	

Food Item	Item Type	Item Price	Item Rating	Count
Paneer Tikka	Starter	250.00	4.50	51

Food Item	Item Type	Item Price	Item Rating	Count

Restaurant Management:

- The restaurant owner can manage his bank account details, view his earnings, and view the reviews that he has received.



Taj Mahal Restaurant

Cuisine: Indian

Rating: 4.50

Contact Details

EMAIL: tajmahal@example.com

Phone Number: +91 9876543210

Bank Details

Bank details: 1234567890123456789

Bank details: INDB0000001

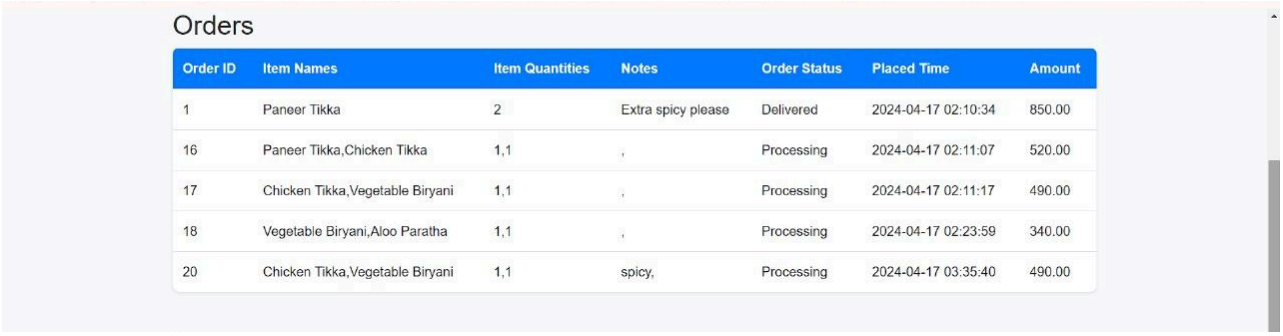
Bank details: Indian Bank

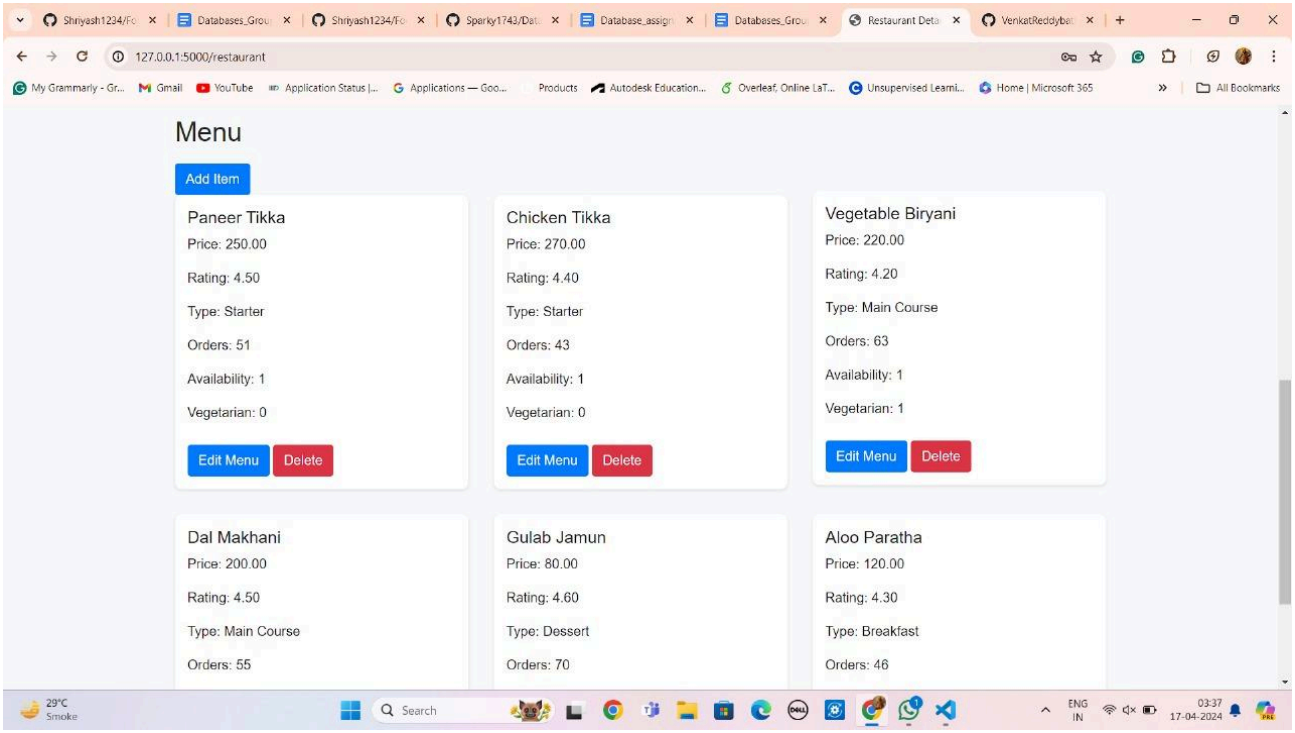
Balance Earned: 30000

Review: Amamzing food! Loved it

- The restaurant owner can view the orders and their status running from his restaurant.

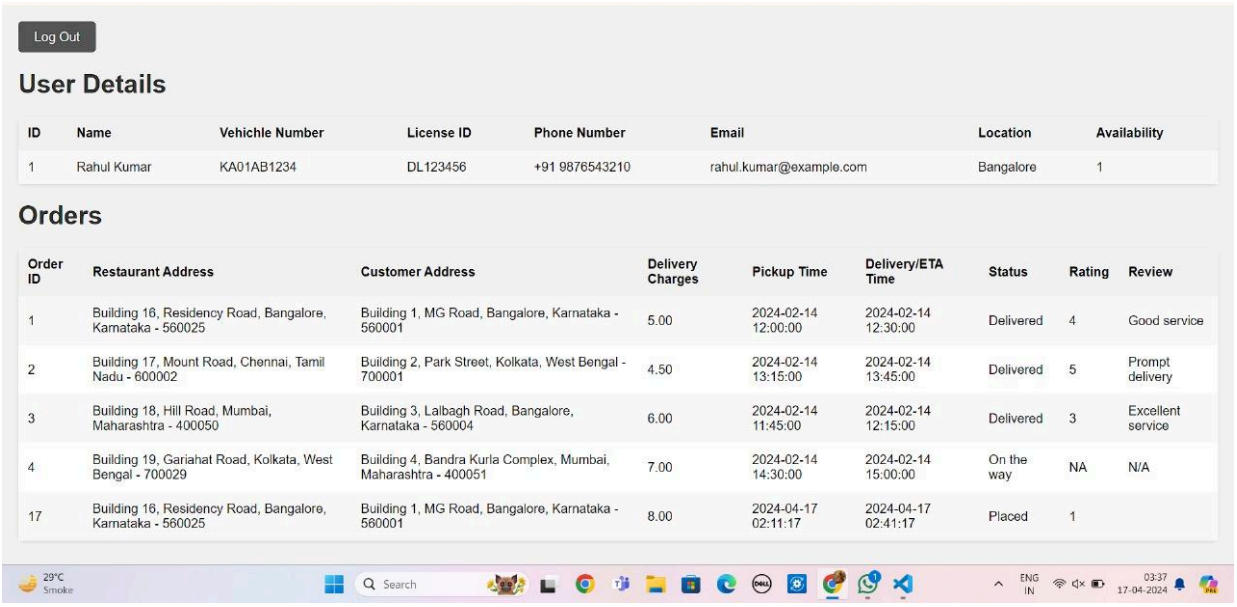
- The restaurant owner has the privilege of editing the menu of all the items in the database in real time.





Delivery Agent:

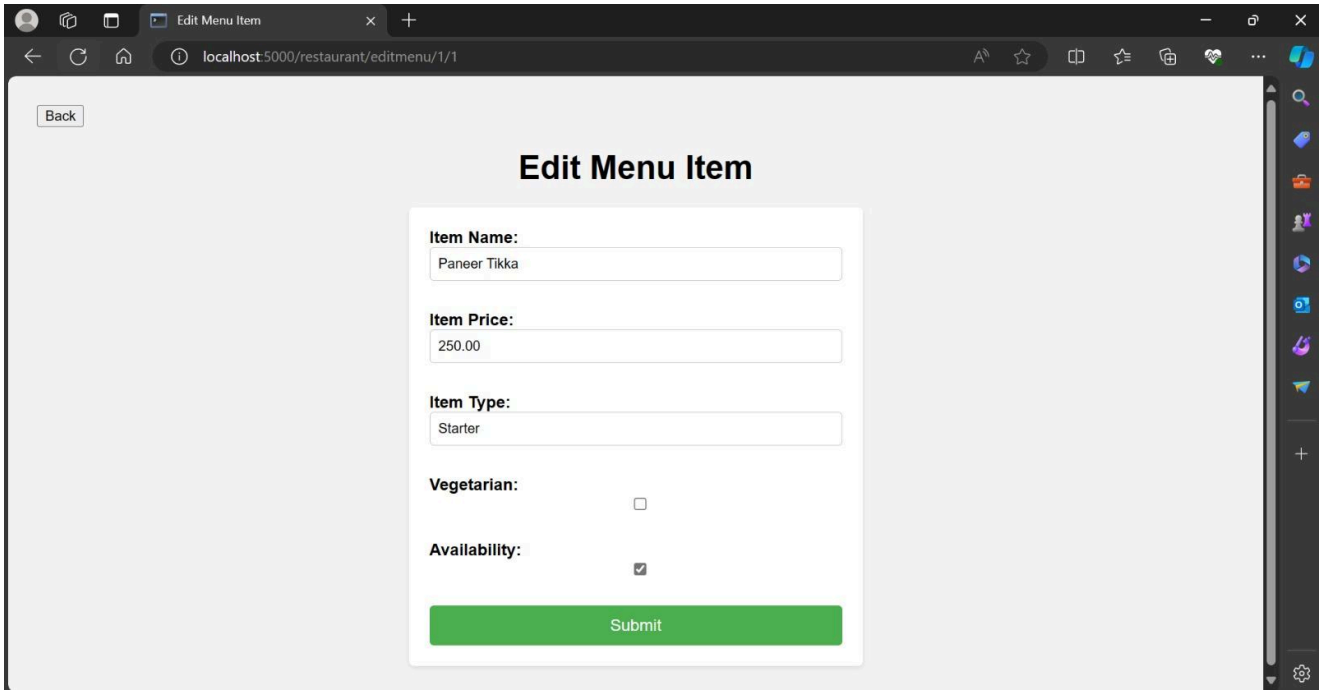
- The delivery agent has the privilege to view the information about his history of orders along with his information on the account.



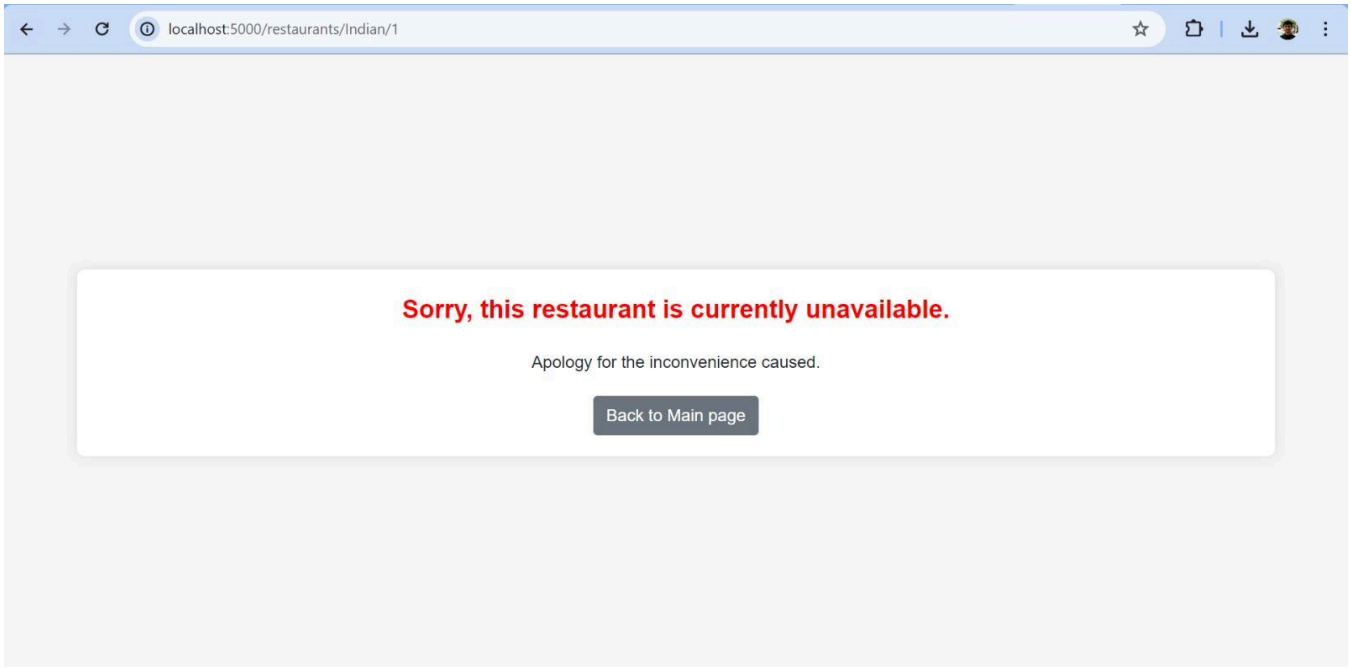
3.2.1 Concurrent Multi-User Access:

It might happen that the restaurant owner could be updating the restaurant menu at the same time as the customer is about to place the order. In that case, what the user is planning to order might not be available in the updated menu anymore. For such a concurrent multi-user access case between the restaurant owner and the customer, we need to ensure that if the restaurant is updating the menu, then the customer is unable to access the menu of that restaurant so that he/she doesn't place any order that the restaurant cannot process after updating the menu.

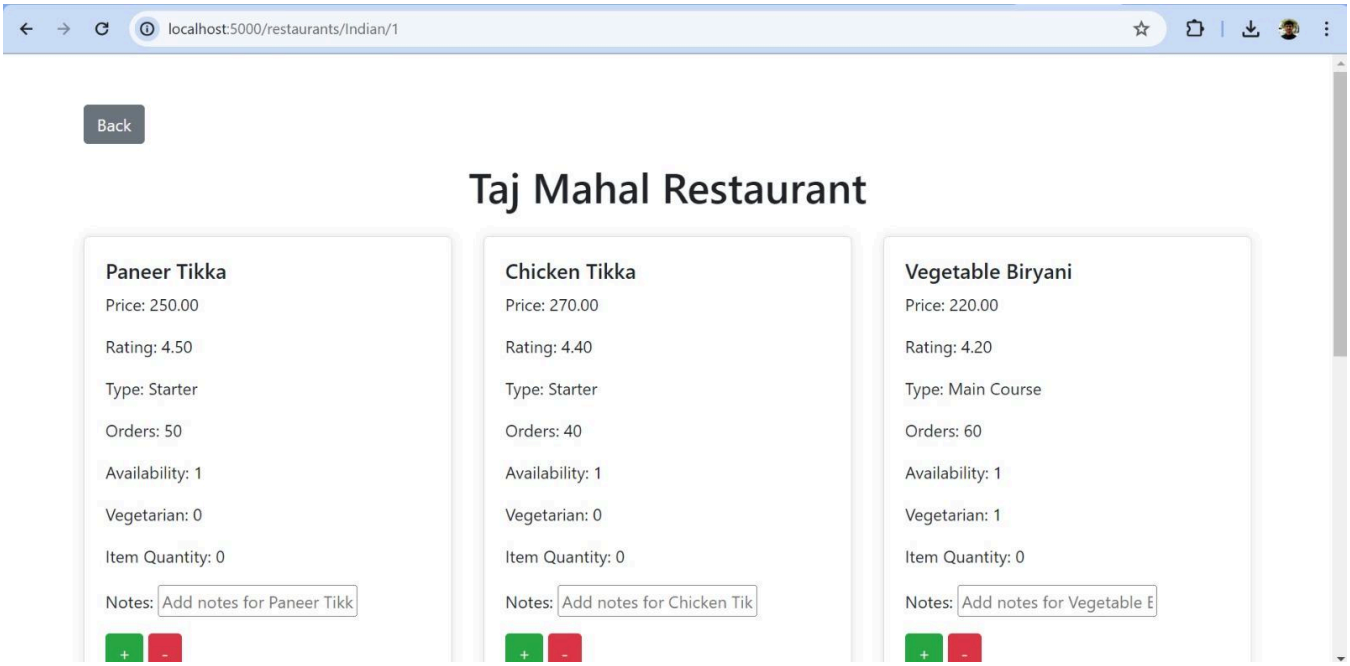
To ensure that such cases are tackled easily, we ensure that the restaurant isn't accessed concurrently by both the restaurant owner and the customer. The restaurant owner will update the data, while the customer will only read the data. To ensure that when the restaurant owner updates the data, the customer doesn't view the data, we implement the concept of lock. So, whenever the restaurant owner is updating the menu, that is the screen is the following for any restaurant:



At the same time, if any user is trying to access the menu of the same restaurant, they will be displayed the following screen to ensure proper concurrent multi-user access:



But, once the restaurant owner completely fills up the Edit Menu and hits the submit button or the back button, the customer is able to access the screen for placing the order as follows:



To implement this logic, we used a variable named ‘editing_menu’, which keeps track of whether the restaurant owner is updating or not. This value is stored in the restaurant schema itself and helps us maintain concurrent multi-user access. We have implemented the following code based on the logic explained above:

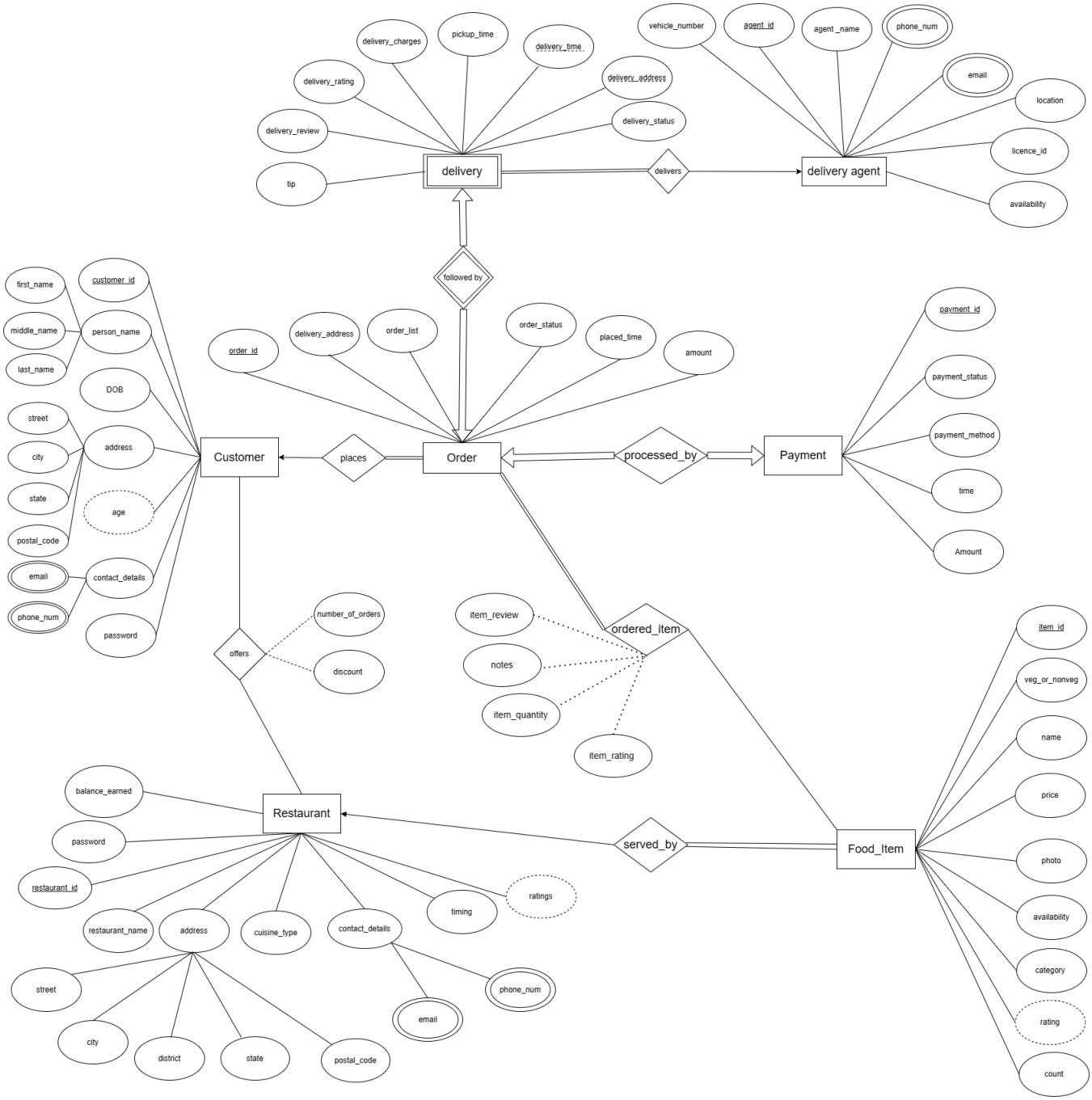
```
@app.route('/restaurants/<cuisine_type>/<restaurant_id>')
def restaurant_menu(restaurant_id, cuisine_type):
    cur = mysql.connection.cursor()
    cur.execute('''
    SELECT *
    FROM food_item
    JOIN restaurant ON food_item.restaurant_id = restaurant.restaurant_id
    WHERE restaurant.restaurant_id = %s
    ''', (restaurant_id,))
    food_item_data = cur.fetchall()
    food_item_columns = [col[0] for col in cur.description]
    food_items = [dict(zip(food_item_columns, row)) for row in food_item_data]

    cur.execute('''
    SELECT restaurant_name, editing_menu
    from restaurant
    WHERE restaurant.restaurant_id = %s
    ''', (restaurant_id))
    restaurant_name_data = cur.fetchall()
    print(restaurant_name_data)
    restaurant_name_columns = [col[0] for col in cur.description]
    restaurant_name = [dict(zip(restaurant_name_columns, row)) for row in restaurant_name_data]
    if(restaurant_name_data[0][1]):
        return render_template("/customers/rest_unavailable.html")
    else:
        return render_template("/customers/menu.html",food_items=food_items,restaurant_name=restaurant_name)
```

3.2.2 Implementing the feedback

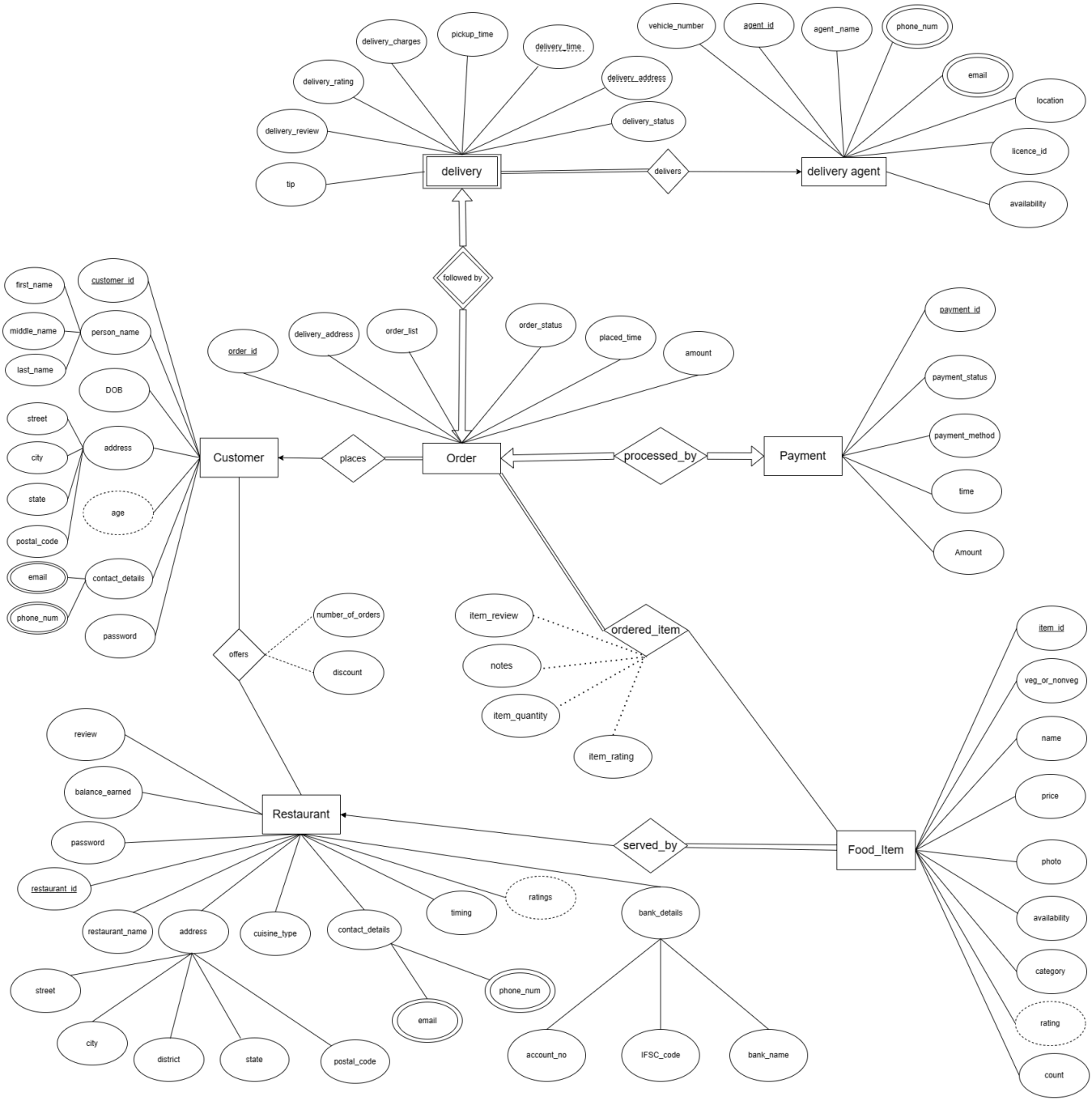
Described in detail in section 3.1.1. As the task was similar

The changes in ER diagrams are:

Before:



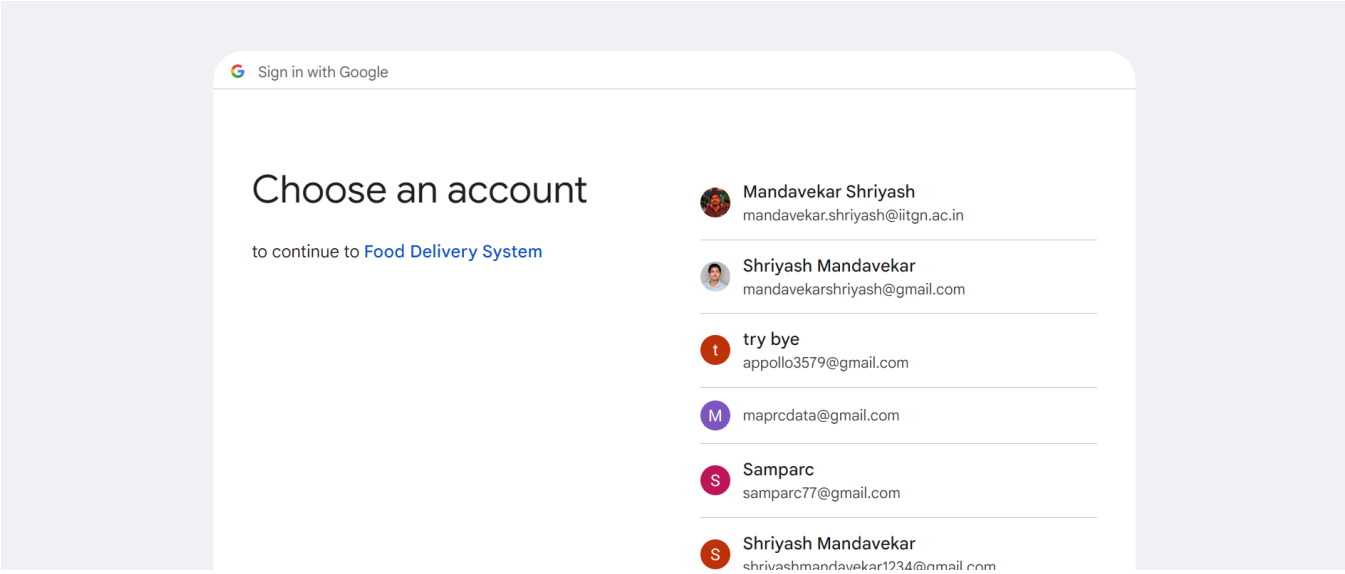
After: The change can be observed in the Restaurant entity, as additional attributes like review and bank details have been added.



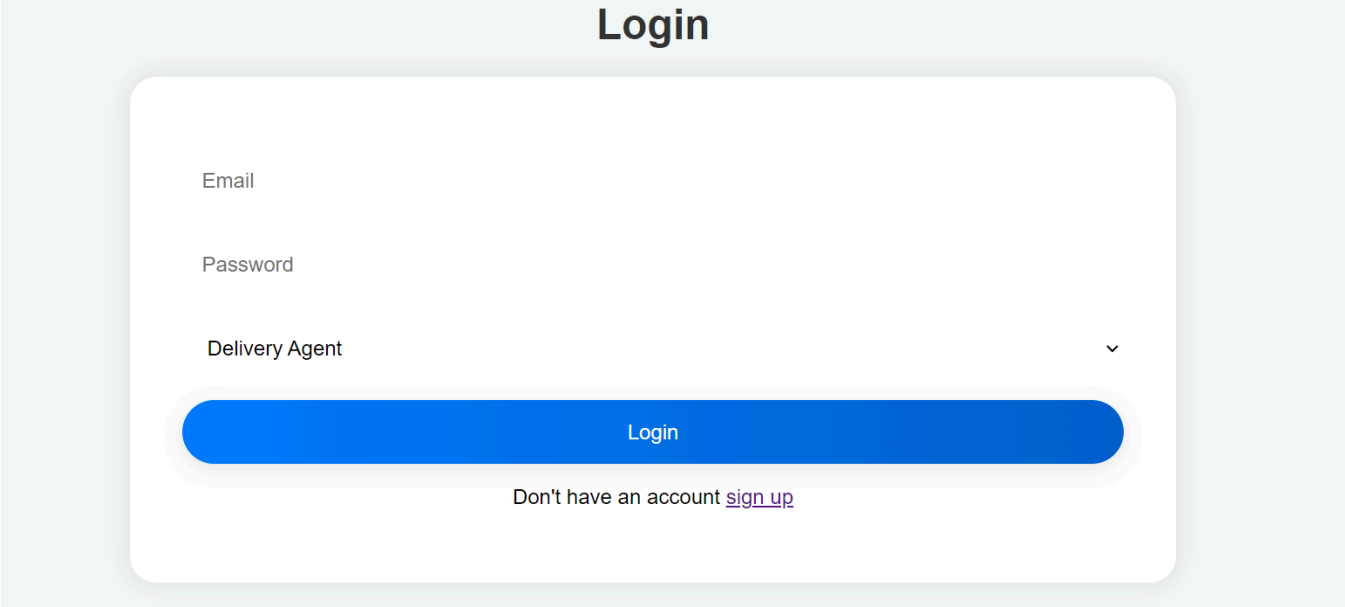
3.2.3 Google sign-in

We added the Google sign-in to the flask. Each user has to go through the Google sign-in before logging in to the account with their Gmail ID. If the user is from outside the IITGN, then the user will not be given further access to the website. The following are steps for Google sign-up.

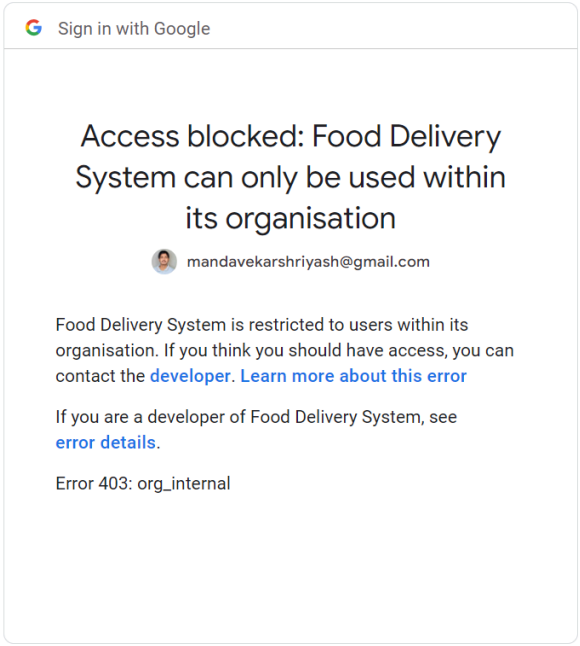
1. Google sign-up page



2. After the sign-up page, the user will go to the login page.



3. If anyone outside the IITGN tries to access the website, the website will throw an error message.




```python
@app.route("/callback")
def callback():
    flow.fetch_token(authorization_response=request.url)

    credentials = flow.credentials
    request_session = requests.session()
    cached_session = cachecontrol.CacheControl(request_session)
    token_request = google.auth.transport.requests.Request(session=cached_session)

    id_info = id_token.verify_oauth2_token(
        id_token=credentials._id_token,
        request=token_request,
        audience=GOOGLE_CLIENT_ID,
        clock_skew_in_seconds=10
    )

    session["google_id"] = id_info.get("sub")
    session["name"] = id_info.get("name")
    return redirect("/login")

@app.route("/googlelogin")
def googlelogin():
    authorization_url, state = flow.authorization_url()
    session["state"] = state
    return redirect(authorization_url)
```

The callback routes check if the user is authorised or not, and if the user is authorised, then it collects the user's name and redirects the user to the login page.

3.3 Responsibility of G1 & G2:

3.3.1 Attacks and Defence

1) Incorrect Queries (That fail on runtime):

● **Description**: This is an SQL Injection attack. We enter logically and grammatically correct query(s) into the field. However, we make sure that they will fail when we submit, i.e., at runtime. This will cause the system to throw an error, which will leave a trace that we will be able to see. From the traceback, we can extract the information about the back-end database system, like the name of the table, columns, rows' data, etc. We collect this info so that we can target some specific aspects of the database instead of the entire database. It can also be useful in building other attacks, more particularly.

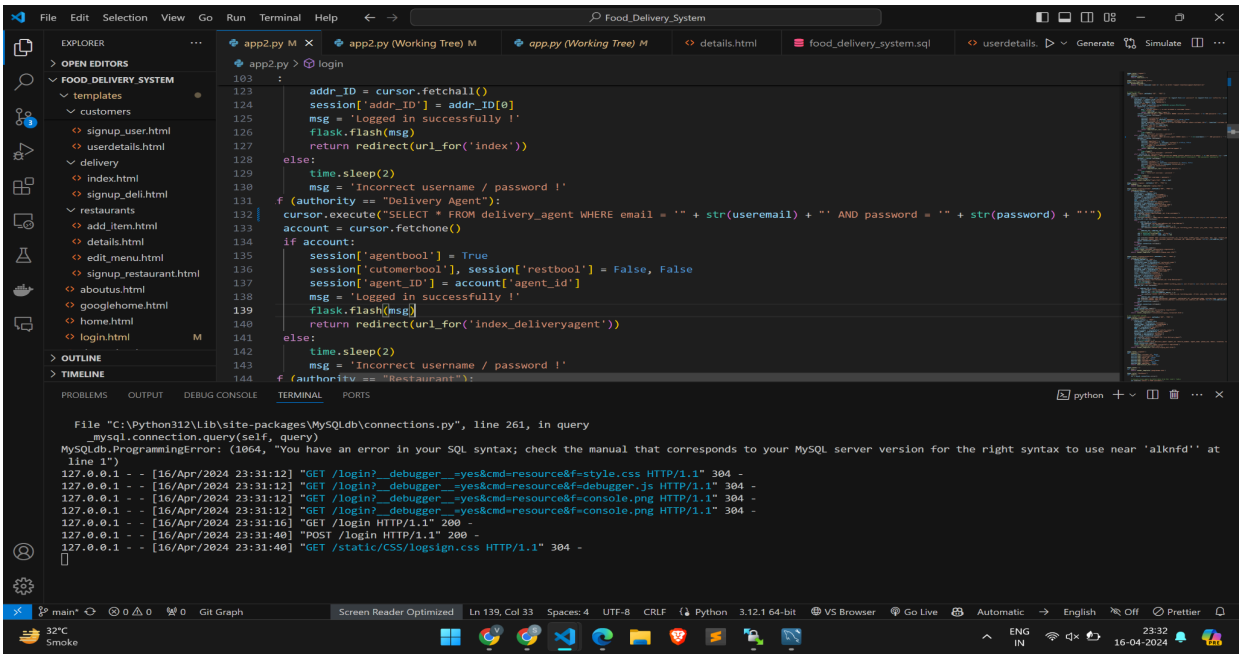
● **Attack**:

1) Email: ‘ OR select * from nsdkjfnfn –

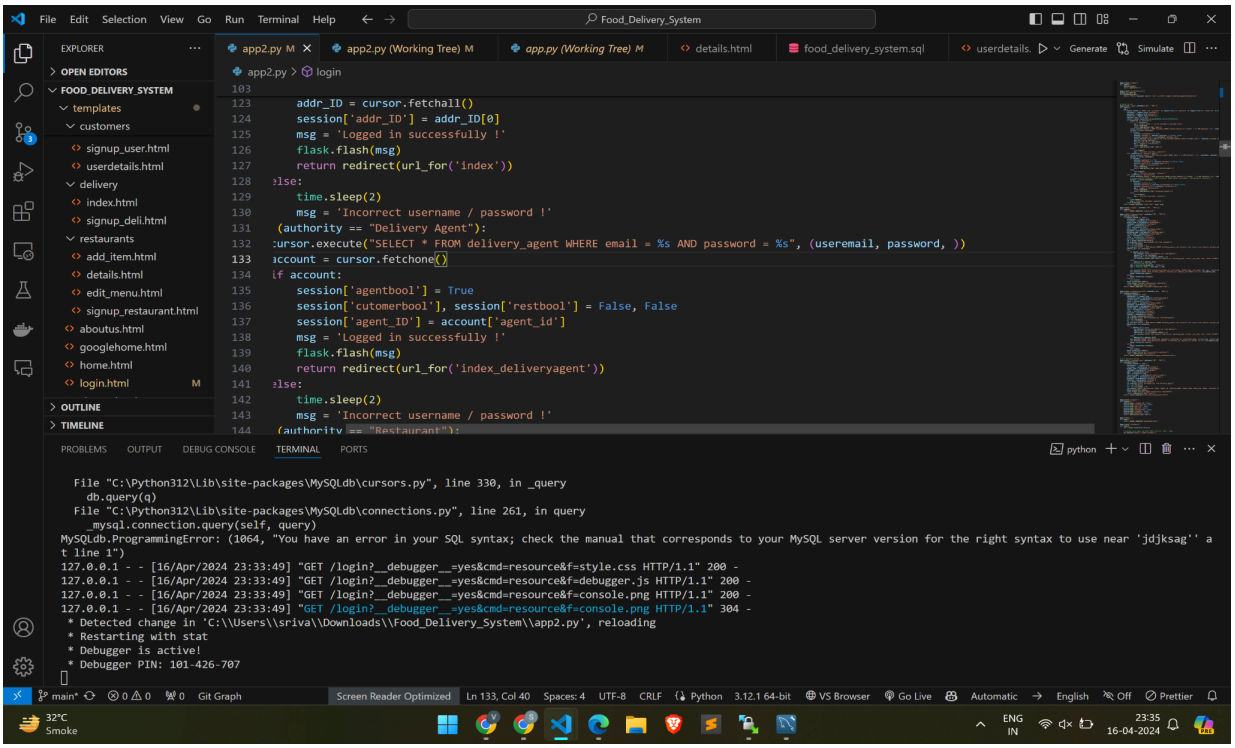
2) Email : ‘ OR SELECT accounts FROM users WHERE login = convert ((select name from sysobjects where xtype='U' limit 1), unsigned integer)

● **Defense**: Do not allow single quotes (‘) or double quotes (“) in the input fields, except in the case that the input is just going to be stored by the database and not going to be used in the query execution.

Before:



After:



2) Cross-Site Scripting (XSS) :

● **Description**: Cross-site scripting (XSS) is a type of security vulnerability that allows an attacker to inject malicious HTML/JavaScript code into the web page. The attacker can exploit a vulnerability in a web application by injecting a script or code that is executed when the user visits the compromised page. This can result in the theft of user data, such as cookies or sensitive information, or the execution of malicious actions on behalf of the user. In persistent XSS, the malicious code is stored in a server-side database, and every time the web page is loaded, the code is executed. In non-persistent XSS, the malicious code is injected into a URL or a form and is executed when the user interacts with the affected page.

● **Attack** :

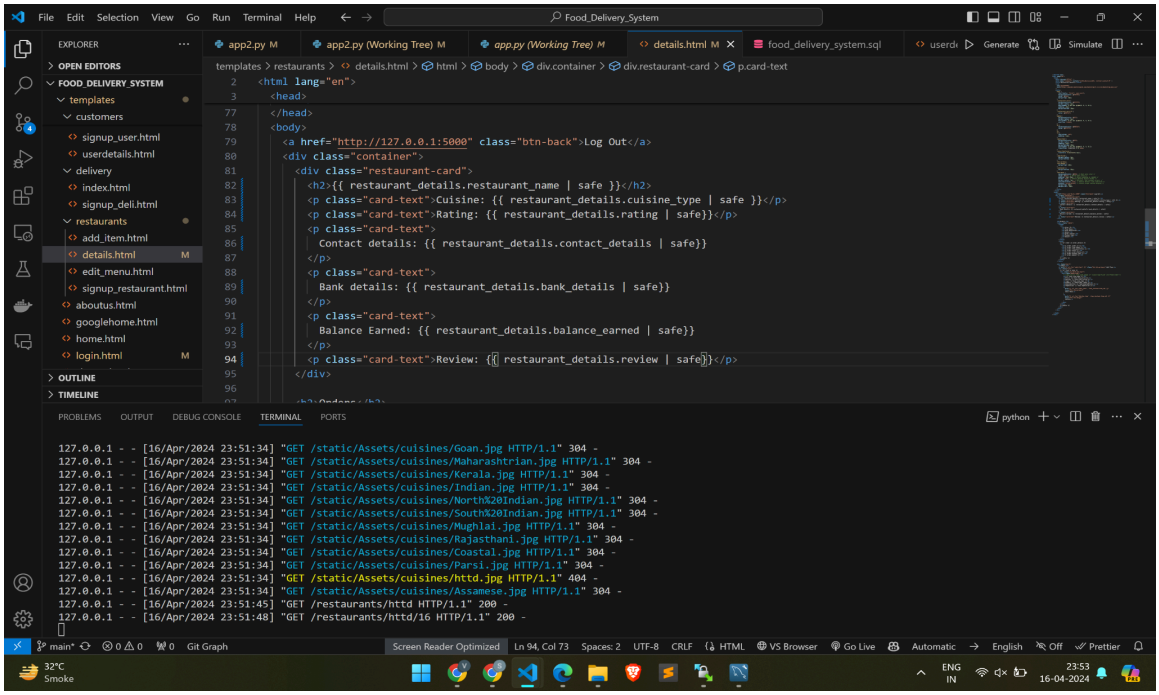
1) Adding a restaurant named as follows: <script> alert(“YOUR ACCOUNT IS HACKED”); </script>

2) If this name is added, then when any other customer wants a restaurant list, this name is loaded in HTML from MySQL, and the HTML code runs this as a script instead of just displaying it as plain text.

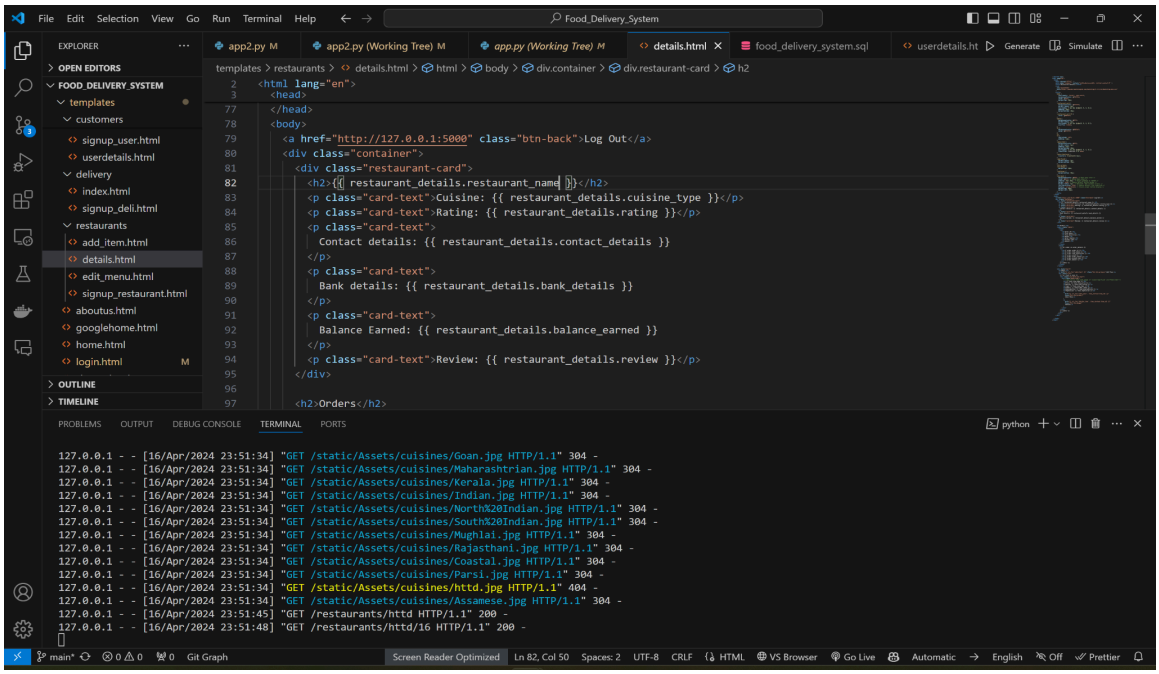
3) This script will give an alert if there is some other script written, and then the user can be forwarded to some other malicious website.

- **Defense**: We used the “safe” keyword function in the HTML code file for displaying the data collected from the database as {{rest[‘name’] | safe}}. This safe function considers the data from the database to be safe and doesn't use any filters. For defence against XSS, we removed all such “safe” functions. After this change, the HTML will consider data from the database as plain text and simply display it without running it as a script.

Before:



After:



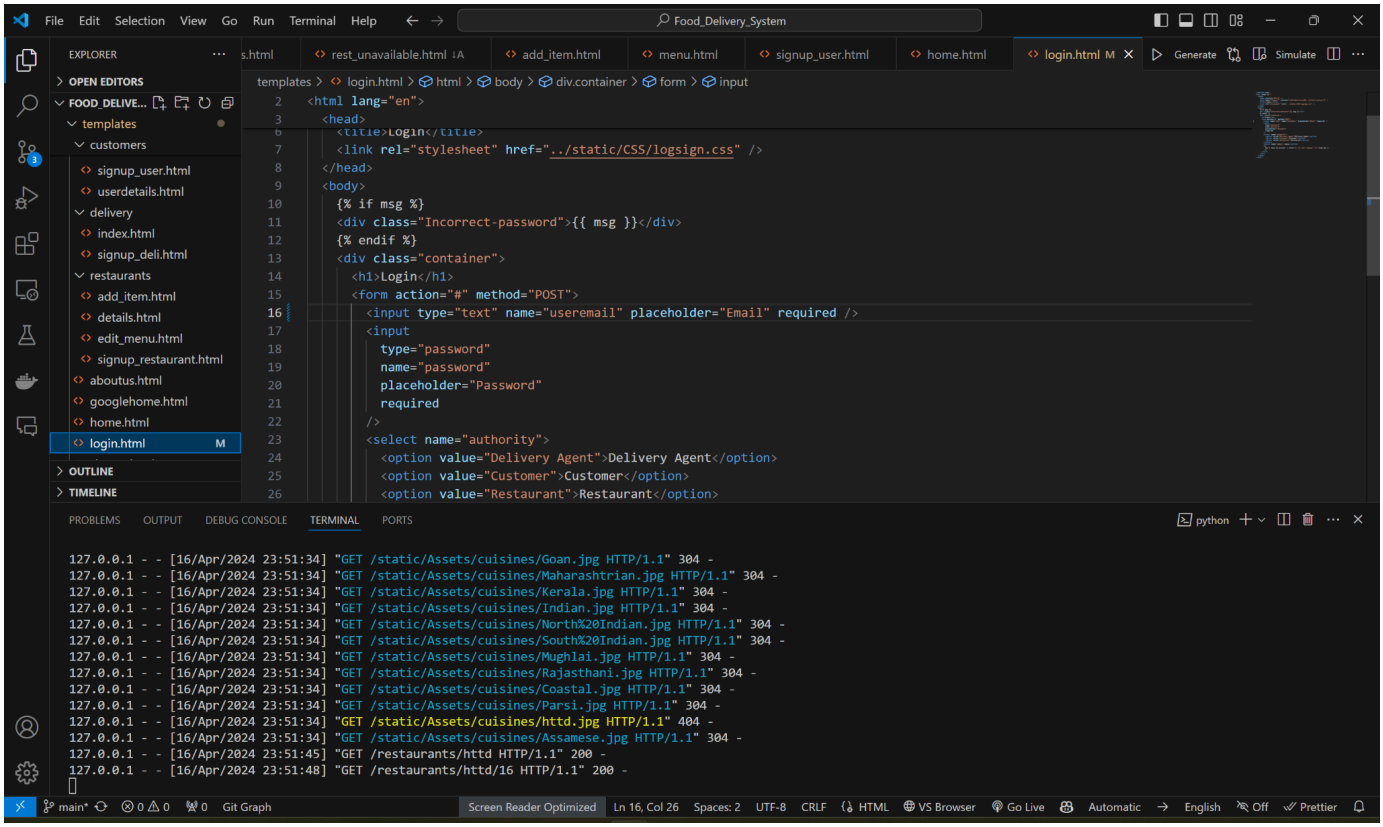
3) **Tautology** :

● **Description**: This is an SQL Injection attack. We write a query that always evaluates as TRUE to allow us to proceed anyway.

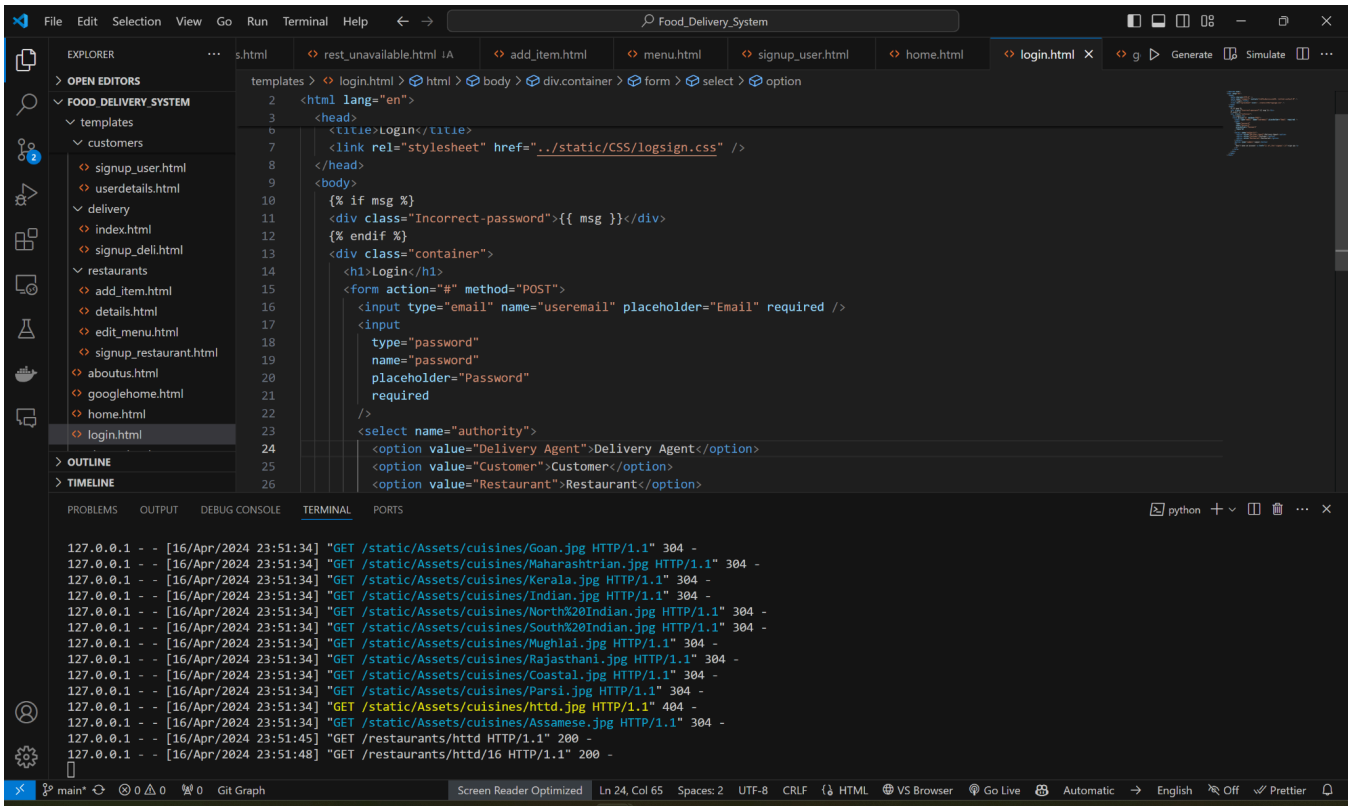
● **Attack** : ○ Email : admin’ OR ‘1’=’1’ --

● **Defense**: Change the input field from text to email/phone_number so that an email/number verification test is executed.

Before:



After:



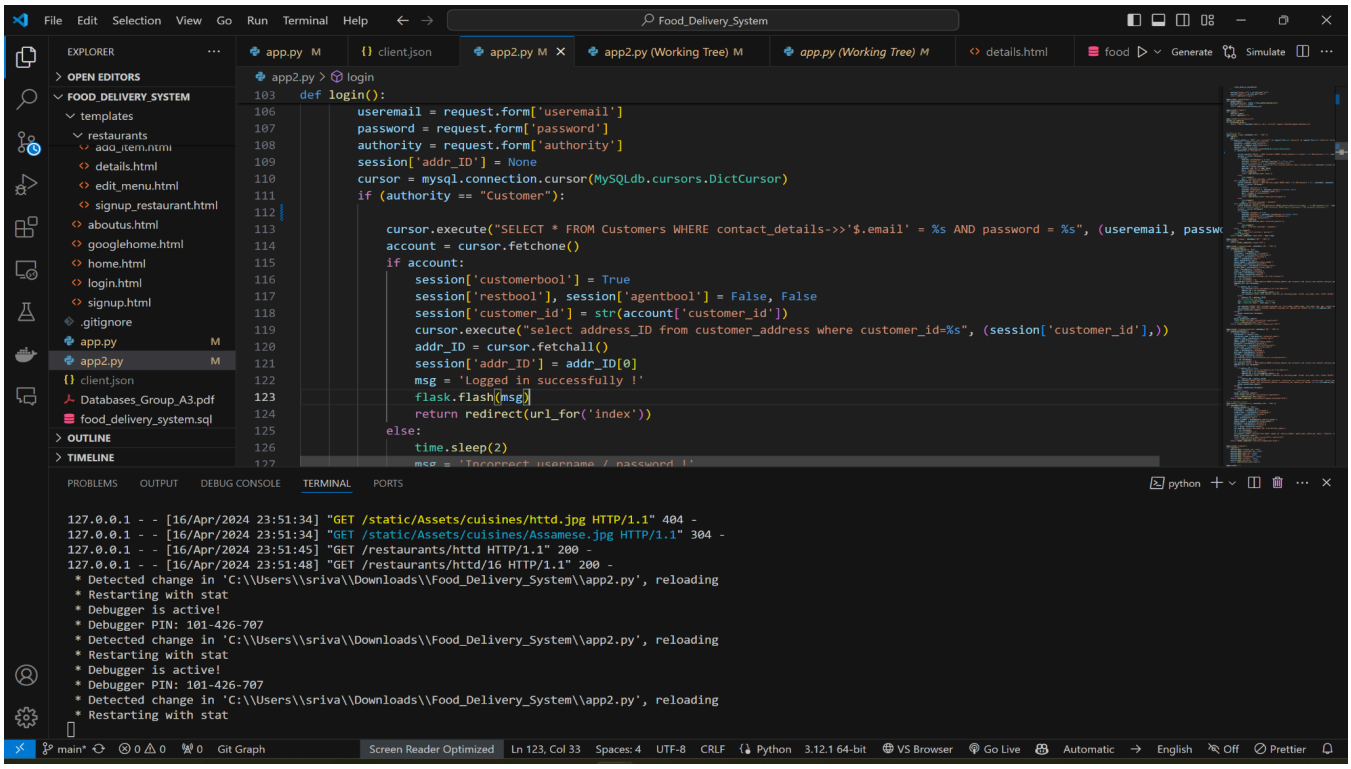
4) **Piggy-Backed Queries** :

● **Description**: This, too, is an SQL Injection attack. In this type of attack, we let the original query run as usual and correctly. But, we instead insert a new query after the original one. The original query is run and executed as usual. But, after that, the injected query is also run, which, thus, piggybacks the original query.

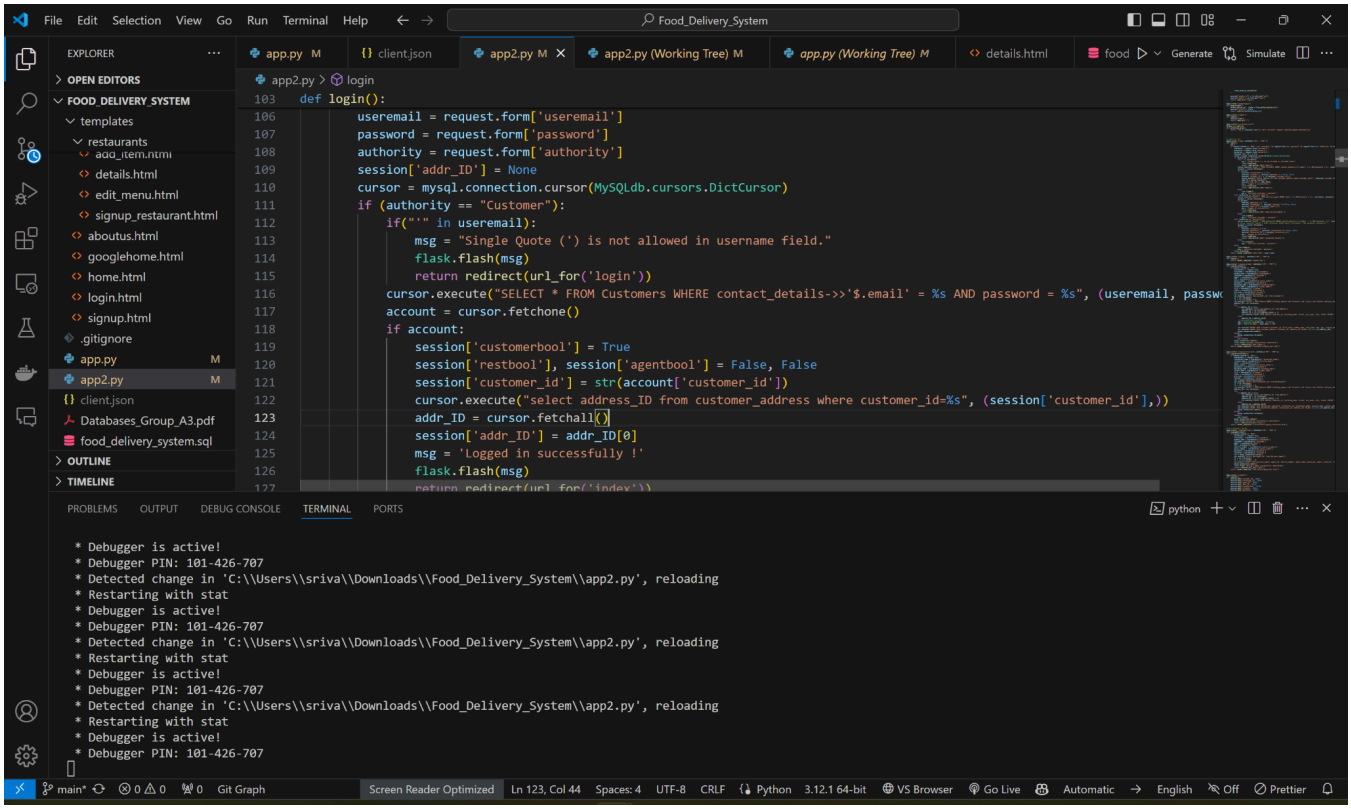
● **Attack** : ○ Password : some_pass_word ’ ; DROP TABLE customers ; --

● **Defense**: We need to look for special characters in the input fields like semicolon(;), quotes(‘ and “) etc. Defences from previous attacks can also be used for this type of attack.

Before:



After:



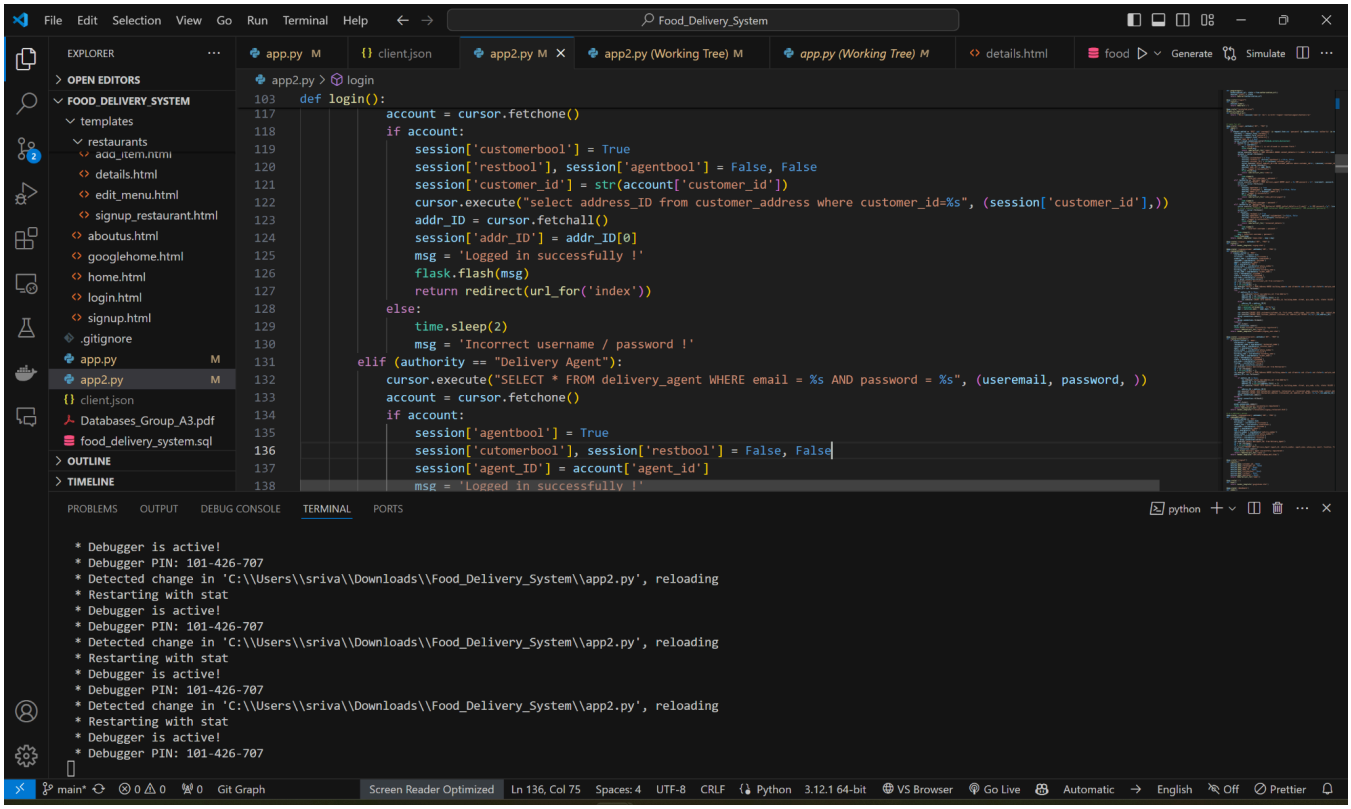
5) **Brute Force** :

○ **Description**: A brute force attack is a type of cyber attack where an attacker attempts to gain unauthorised access to a system or data by trying every possible password or encryption key until the correct one is found.

○ **Attack** :

1) For some valid usernames, we will try all possible password combinations. If the possible combinations are very few or if the user has a weak password, then we can brute force the password.
2) We have used the “hydra tool” for this purpose.
3) We had username.txt and password.txt as possible passwords and usernames
4) $ hydra -l username.txt -p password.txt 127.0.0.1 8000 -o safe.txt
5) This code will go through combinations of usernames and passwords and return correct combinations in the safe.txt file.

○ **Defense**: To defend against brute force attacks, we need to make passwords stronger. For this, we can force the users to choose a password which has a length of more than 8 or 10 characters. In addition to that, we can impose a constraint that those passwords should include at least one unit of the following: capital letter, small letter, digit and special character. We have also added a three 3-second delay between the submit button click and the displaying of the same webpage, in case the input username and password are incorrect. After this, the brute force attack will take 3 seconds per check. In this way, we have optimised our defence against brute-force attacks.



3.3.2 Validity of relationships and constraints after feedback

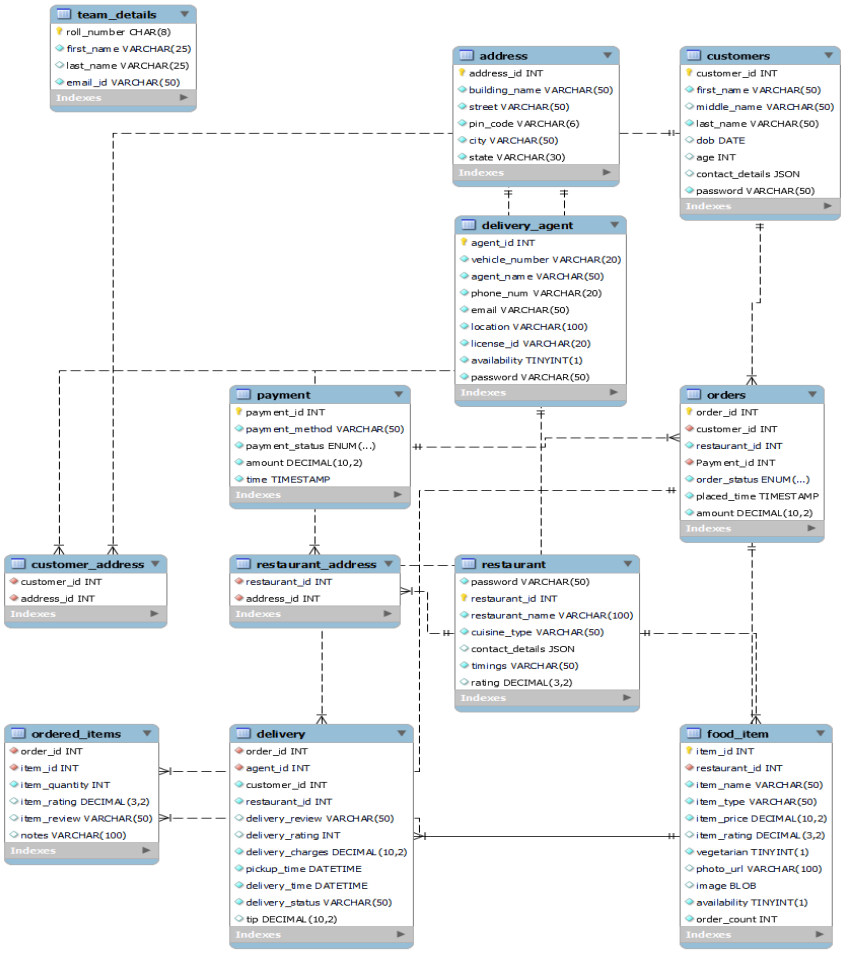
From our original database, there were no constraints in the old database, which we have changed in the new database. All changes have been made in only one table, the restaurant table. Also, based on the feedback, we did not identify any need to add any additional constraints in this table of the database.

Also, we can interpret from comparing both the ER diagrams that no constraints or relationships have been validated.

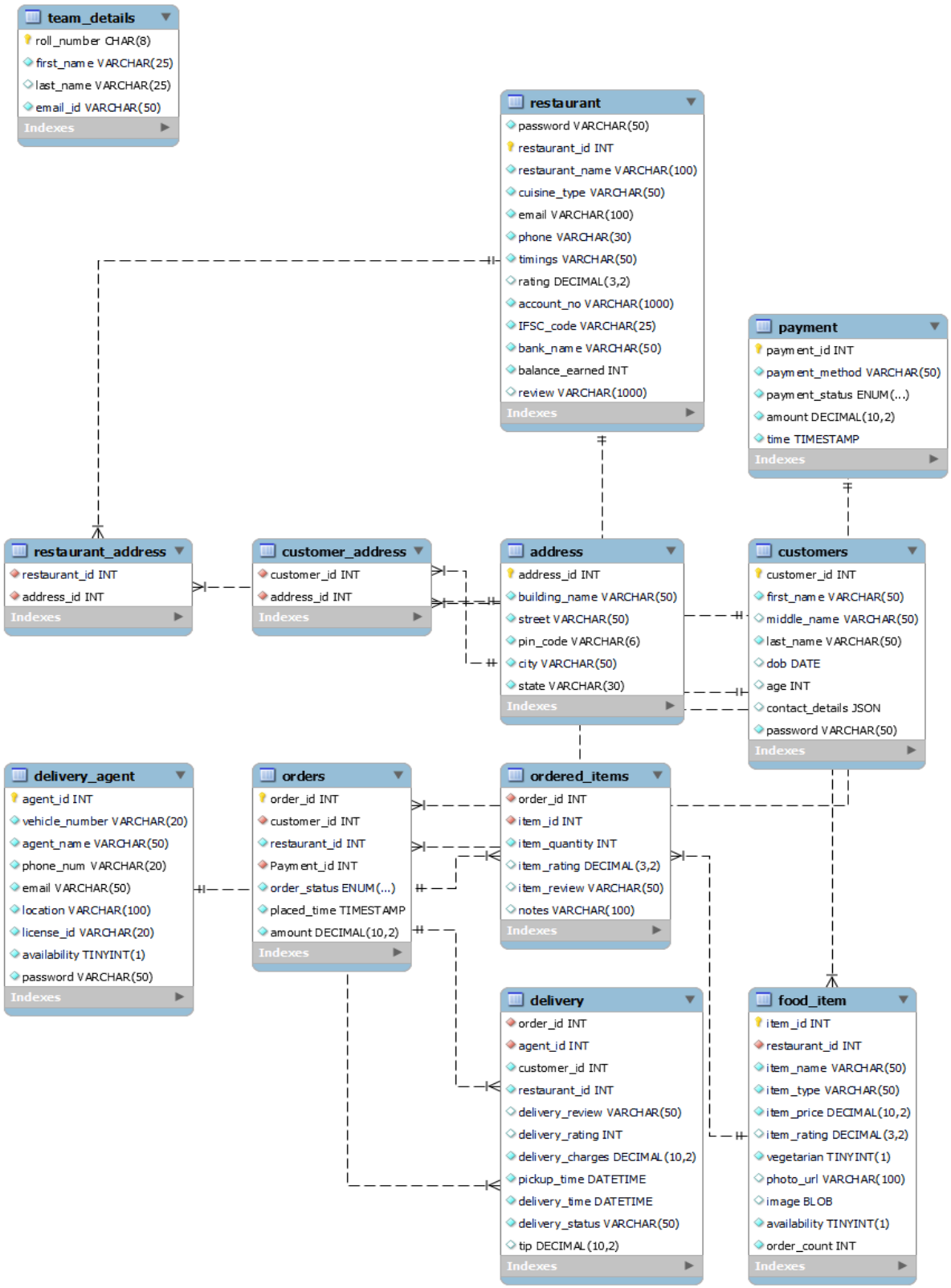
Among the new attributes added to the restaurant table are the following:

- account_no: type Integer, unique
- IFSC_code: type - varchar, can be common for multiple entries
- bank_name: type - varchar, can be common for multiple entries
- review: type - varchar, can be common for multiple entries
- balance_earned: type - Integer, can be common for multiple entries

Our OLD ER Diagram :



Our new ER diagram:



References

- Classroom slides
- SQL Queries
- Flask
- SQL injection
- hydra tool

Work Distribution

Team Member	Contribution
1. Birudugadda Srivibhav	Collecting feedback forms. Did SQL injection part. Making up the final report, pointing out bugs and recording video.
2. Srivathsa Vamsi	Collecting feedback forms, SQL Injection part, pointing out bugs, recording video, making the final report and code base
3. Sriman Reddy	Collecting feedback forms, SQL Injection part, Making up the final report, pointing out bugs and recording video.
4. Nikhilesh Myanapuri	Collecting feedback forms, SQL Injection part, pointing out bugs, recording video, making the final report and code base
5. Haikoo Khandor	Contributed to the concurrent multi-user access and helped in making the final report and code base.
6. Madhav Kanda	Contributed to the concurrent multi-user access and helped in making the final report and code base.
7. Shriyash Mandavekar	Added Google authentication. Added earning by restaurants functionality. Some minor changes on the home page.
8. Deep Thakkar	Doing the 3.2.2 and 3.3.2 questions and helping in doing the final report.