CS 432 Final Report

Food Delivery

Osborg

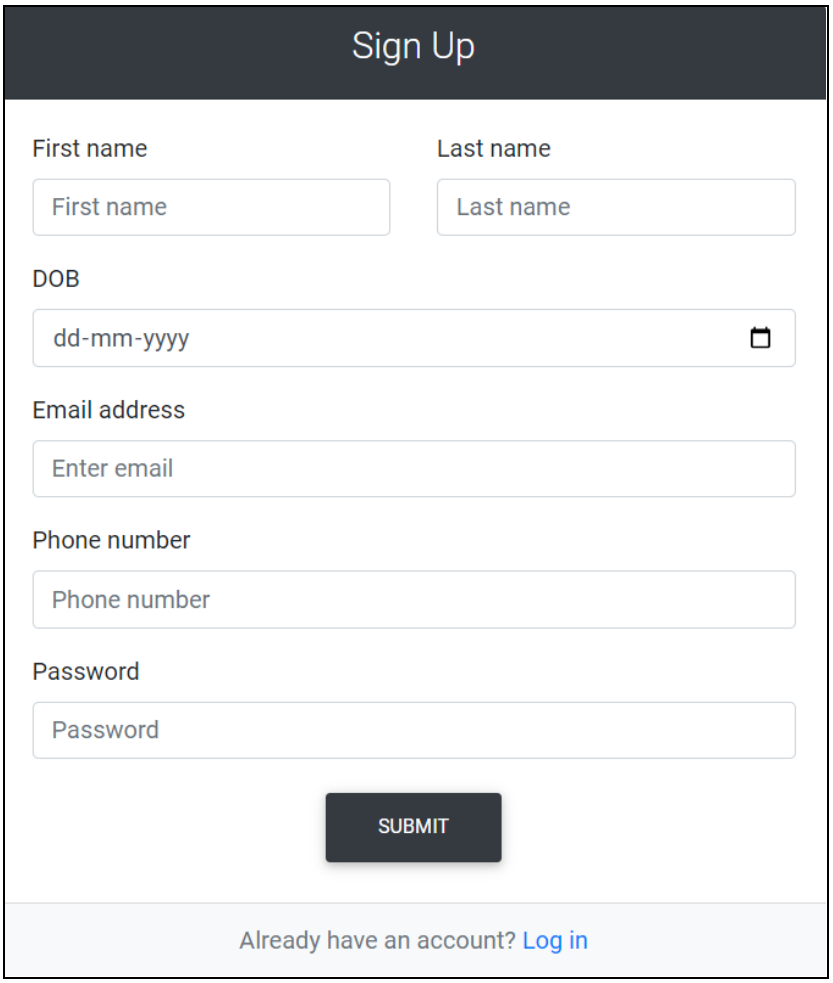
4th September, 20XX

3.1.1

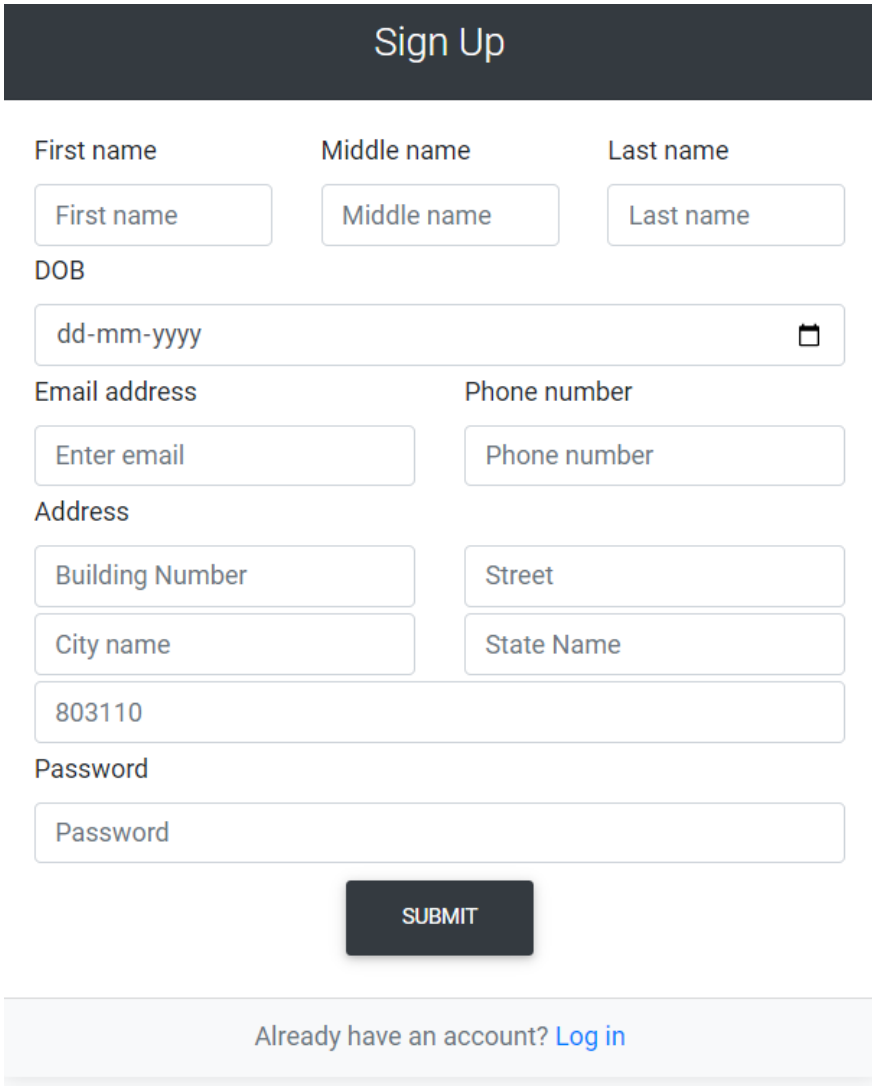
We took two rounds of feedback from all stakeholders and, based on the review, made the following changes in our food delivery system. Changes post the first feedback, compared to the initial version.

Customer:

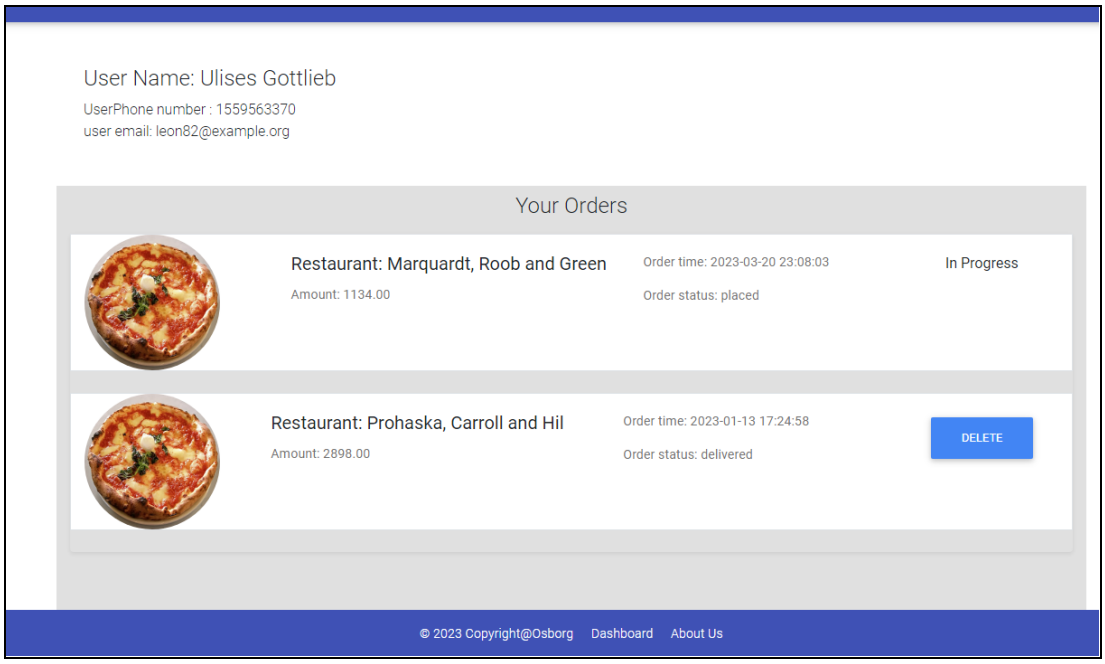
Filling up signup details. All the mentioned fields are mandatory. Each signup page is customized as per the user's authorization.



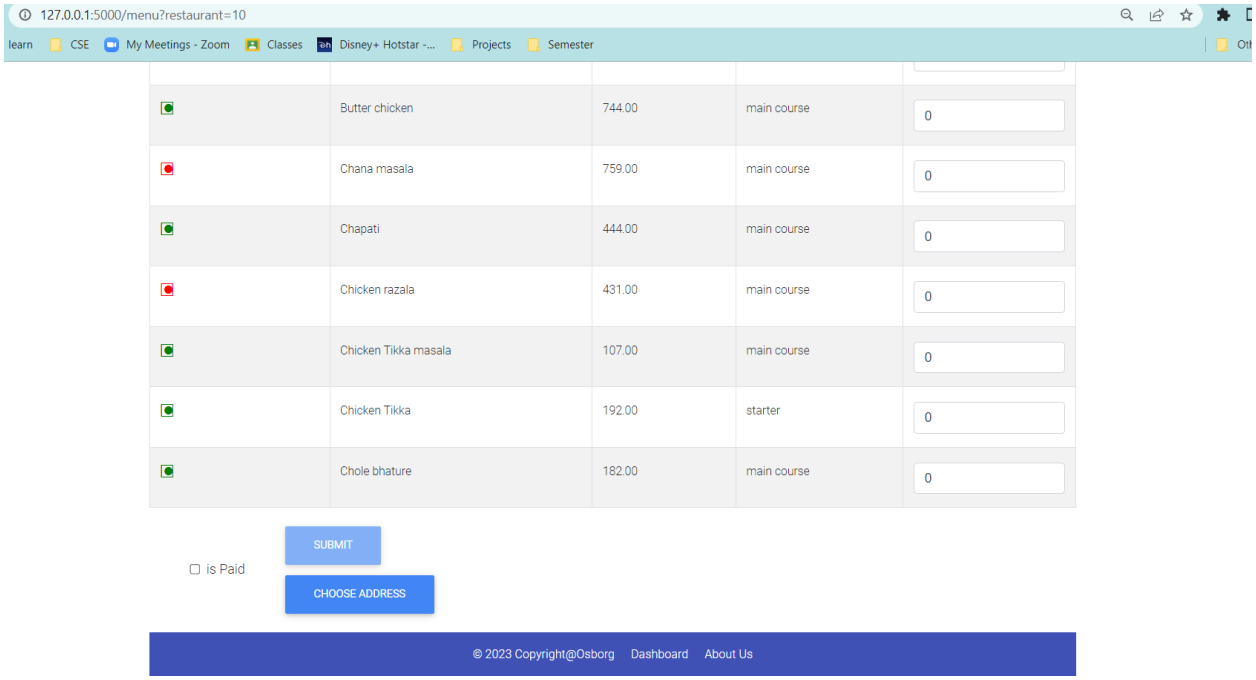
Based on feedback, the modification was made so that stakeholders can now enter their addresses at the time of signup.



This page showcases the personal details of the user and the status of the orders they have placed.

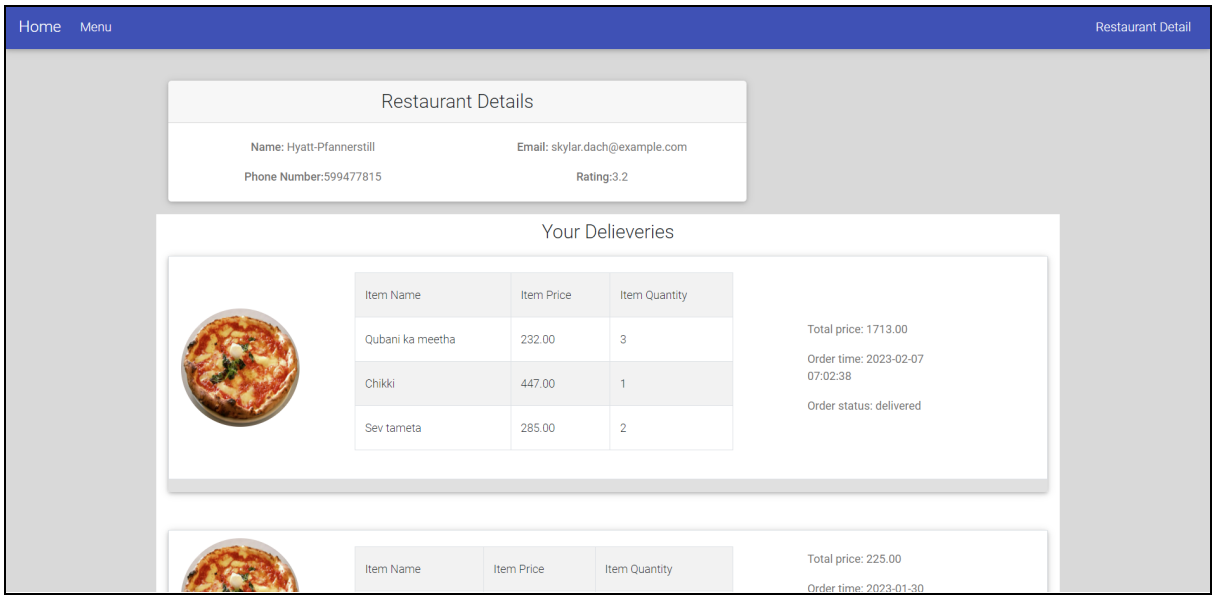


The view was modified so that the user can now choose the address for their account at the time of placing the order.

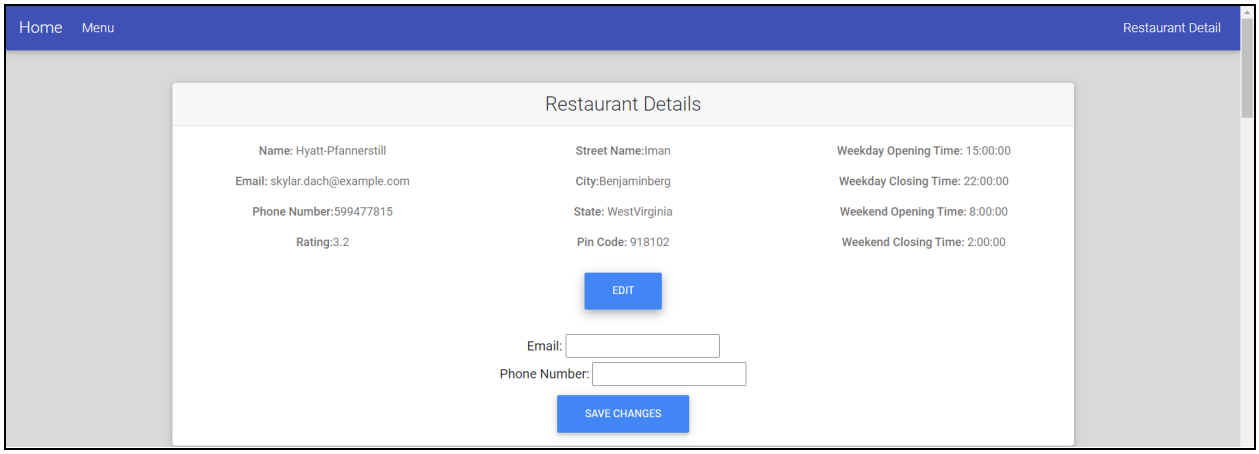


Restaurant:

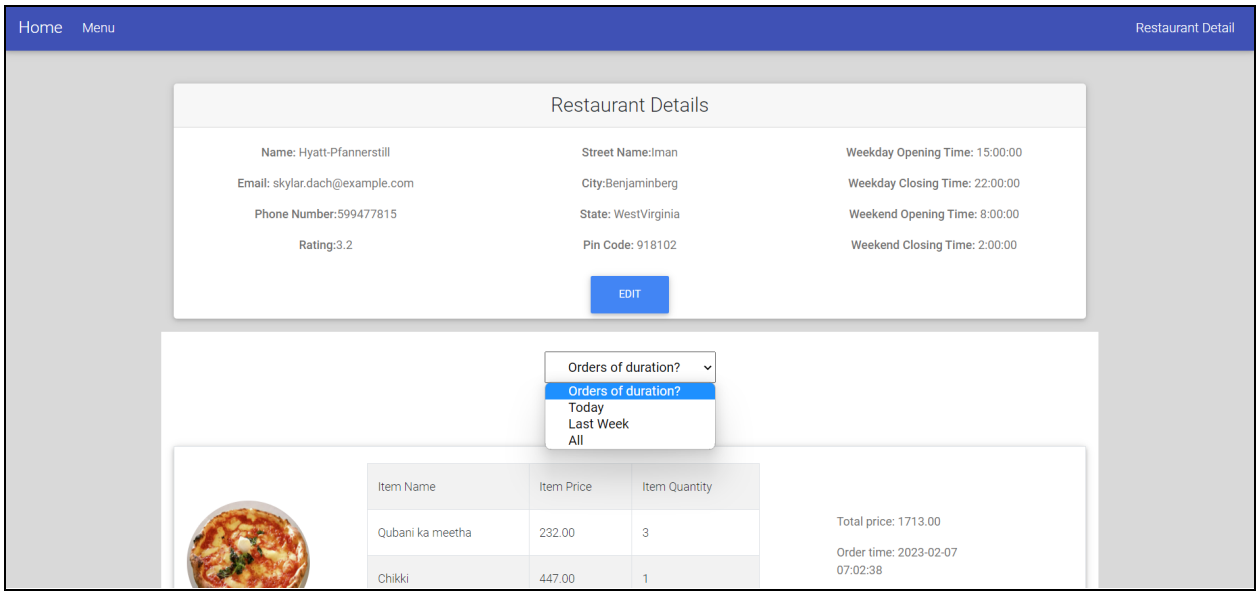
Restaurant owners can access these pages. They will be able to see the "Name," "Address," "Email," and "Rating" of the restaurant. They can view the latest 10 orders given to this restaurant.



This was modified to give them access to edit their email address and phone numbers at any given time. Also, the opening and closing time of the restaurant is mentioned separately for weekdays and weekends.

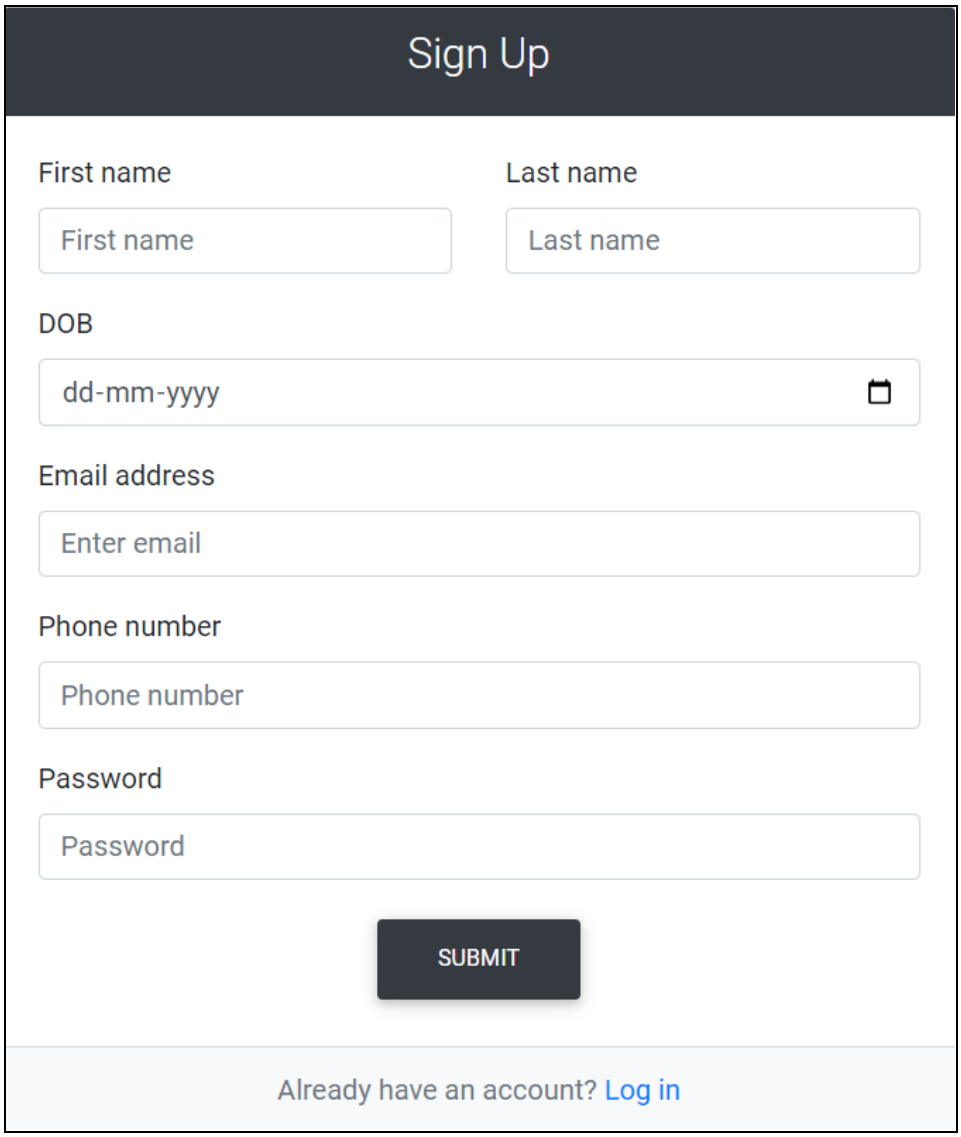


Moreover, a feature to sort orders according to duration is also provided. They can filter orders for the day or the week.

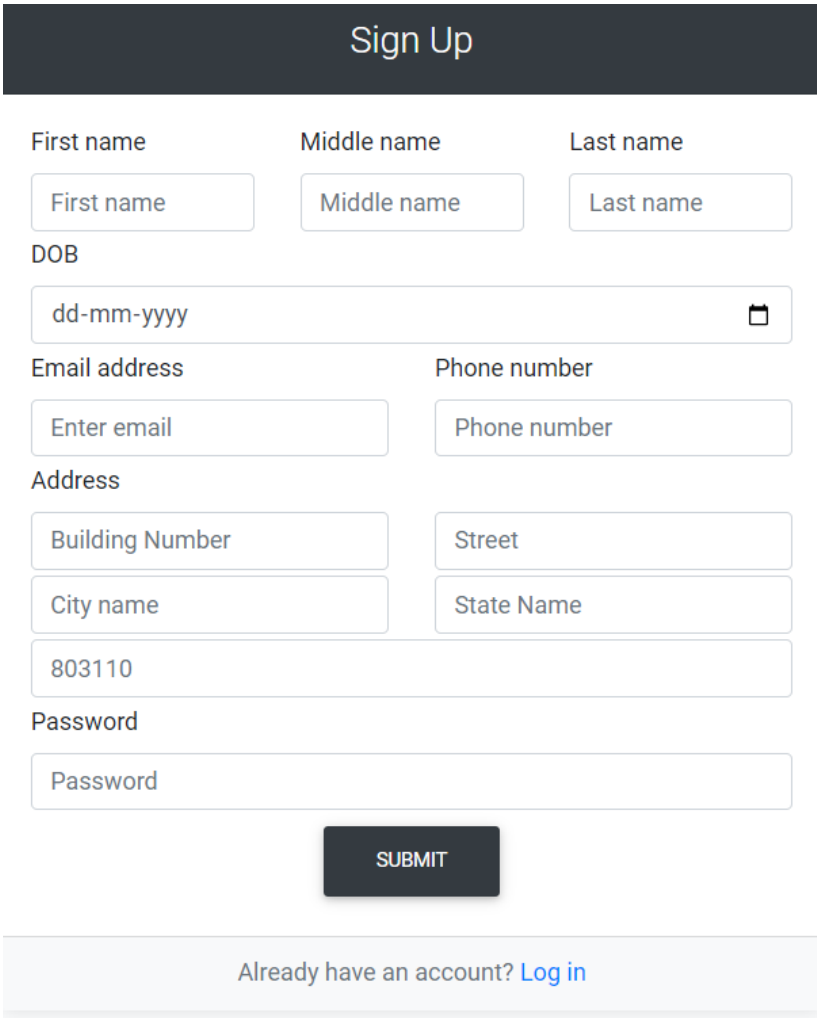


Delivery Agent:

Filling up signup details. All the mentioned fields are mandatory. Each signup page is customized as per the user's authorization.



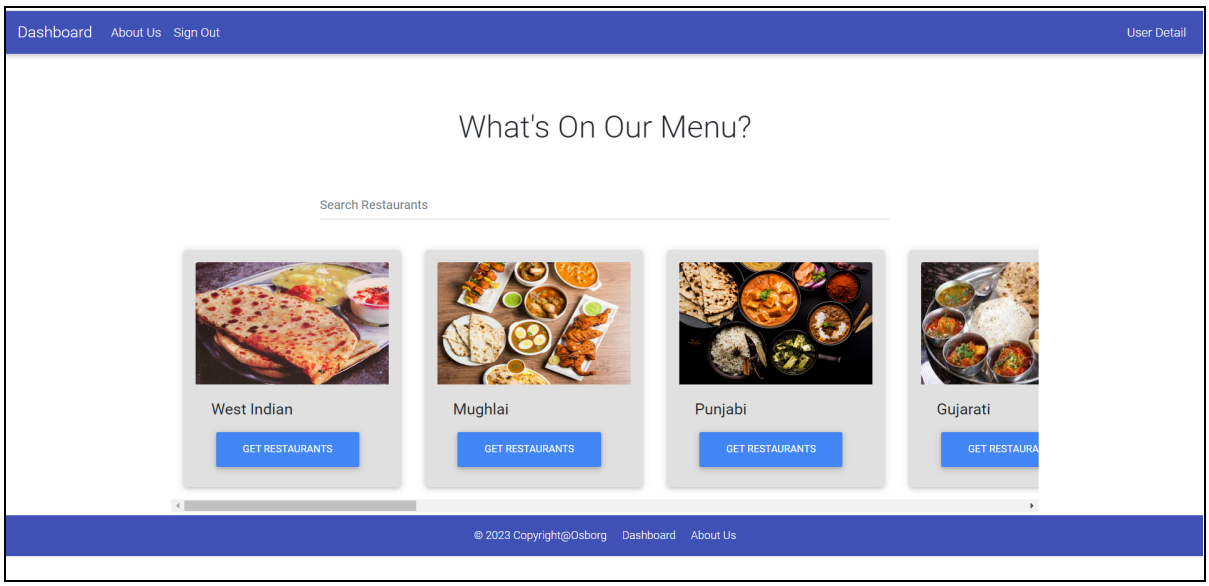
Based on feedback, the modification was made so that delivery agents can enter their addresses at signup.



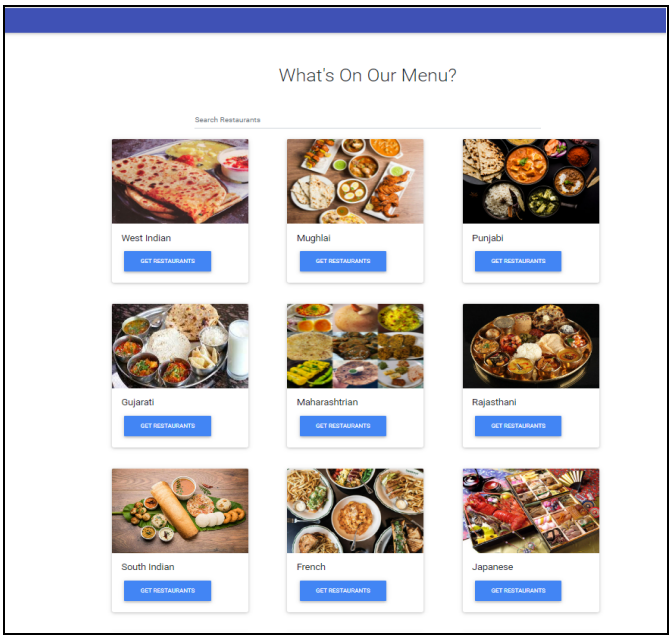
Further changes introduced after the second round of feedback:

Customer:

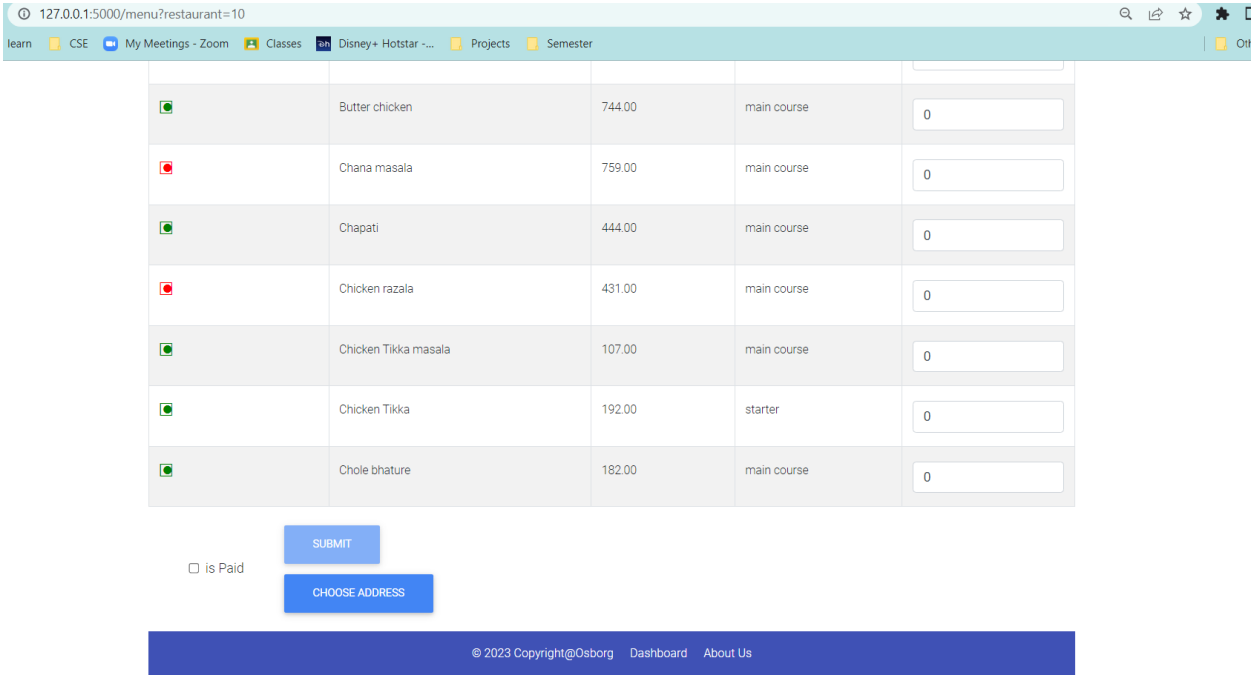
Dashboard for customers. It contains a search bar and cuisine types being served by restaurants. Earlier, the customers had to slide through a horizontal column to view all types being served.



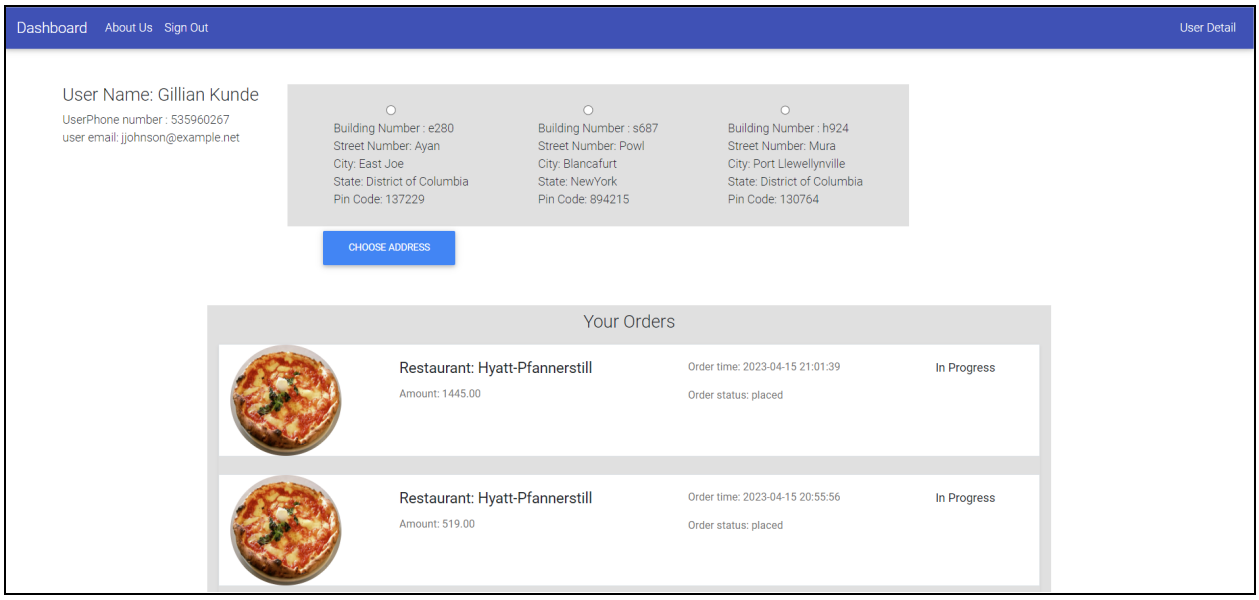
It is now modified vertically, such that the users can view all cuisine types at the same time for a better customer experience.



The view was modified so that instead of choosing the order through a button, we can change it using a simple radio button for a better customer experience. This enables them to order from different places like home, office, etc.

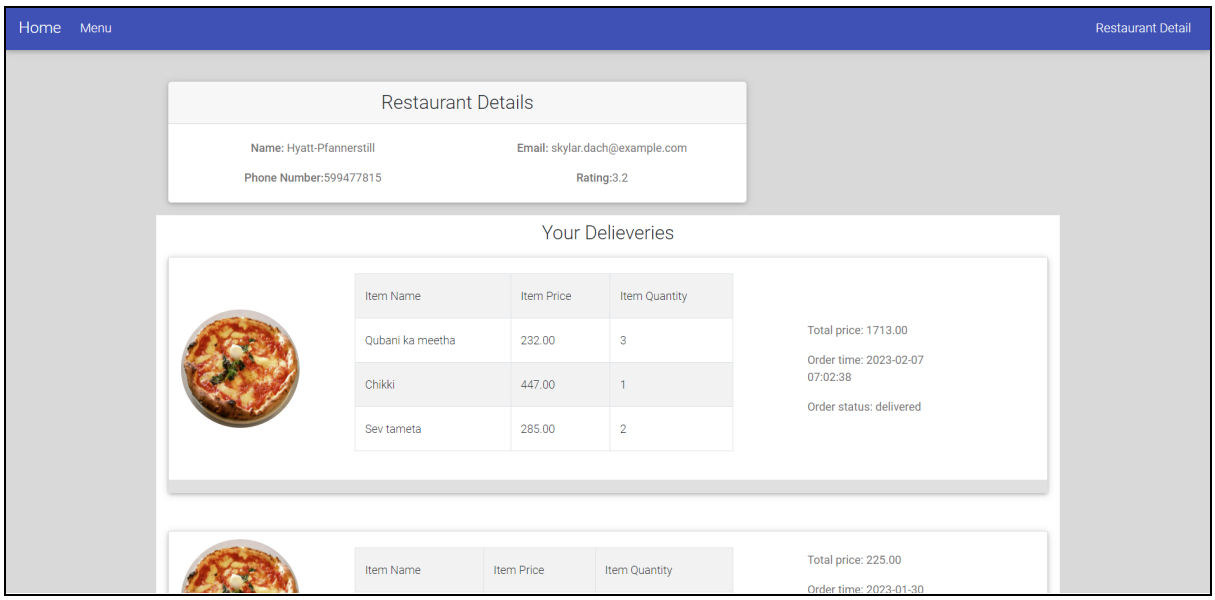


The changed view with the radio buttons.

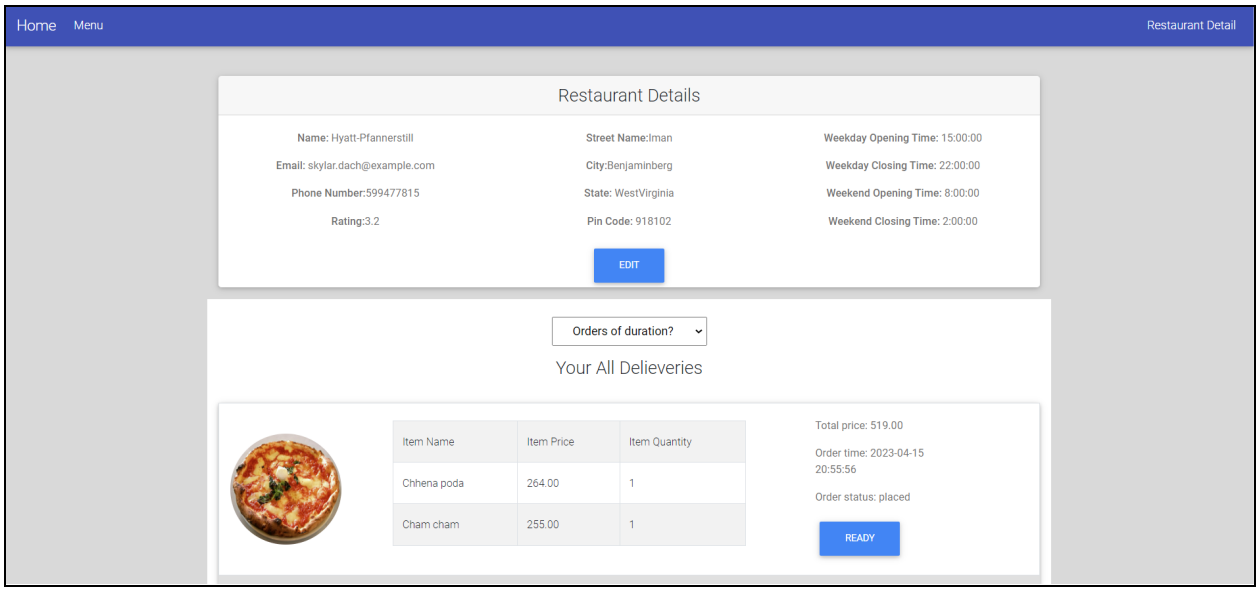


Restaurant:

Restaurant owners can access these pages. They will be able to see “Name”, “Address”, “Email”, and “Rating” of the restaurant. They can view the latest 10 orders given to this restaurant.



This was changed so that restaurant owners could update the status of the order, and change it to ready whenever the food is prepared.

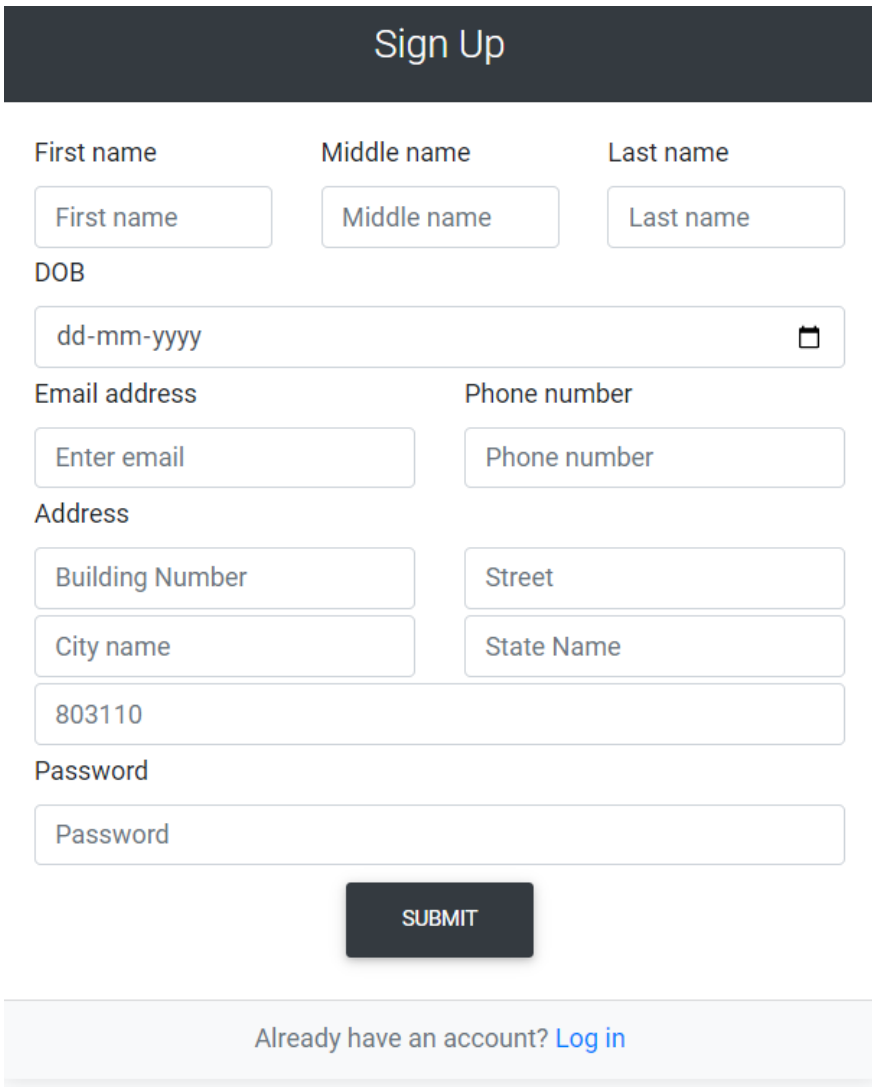


3.1.2

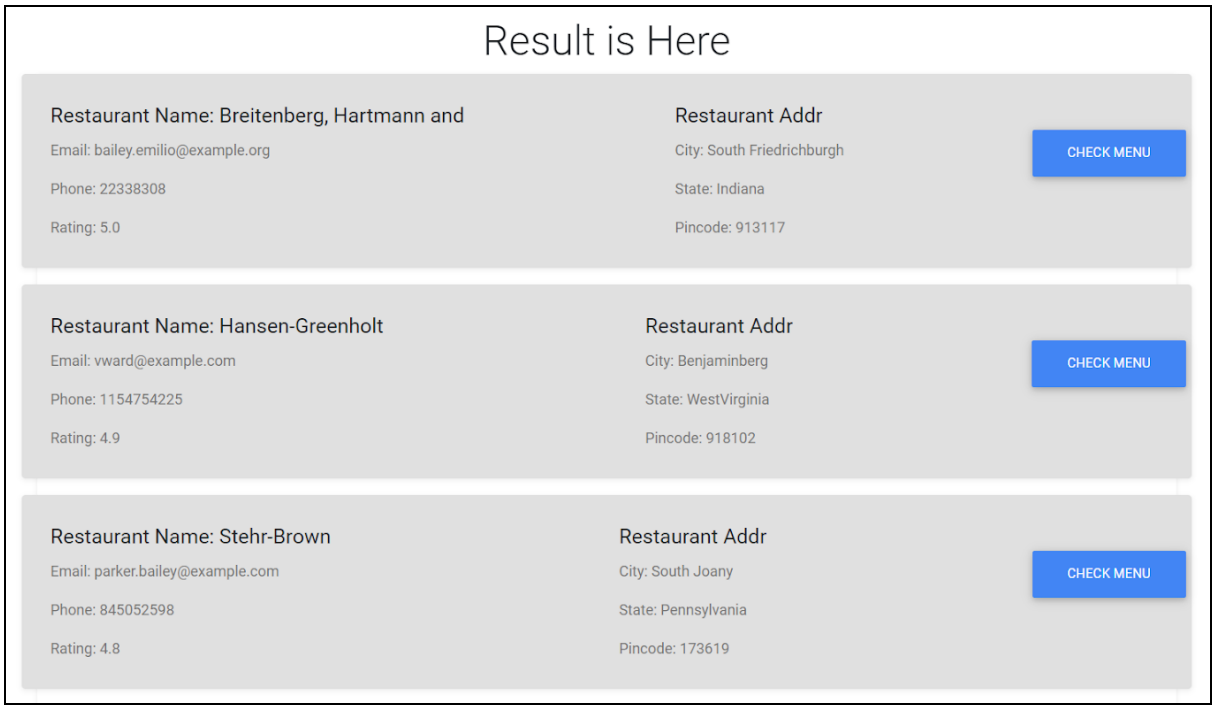
Customer:

The customer has the following rights and privileges that enable them to make the following changes to the database:

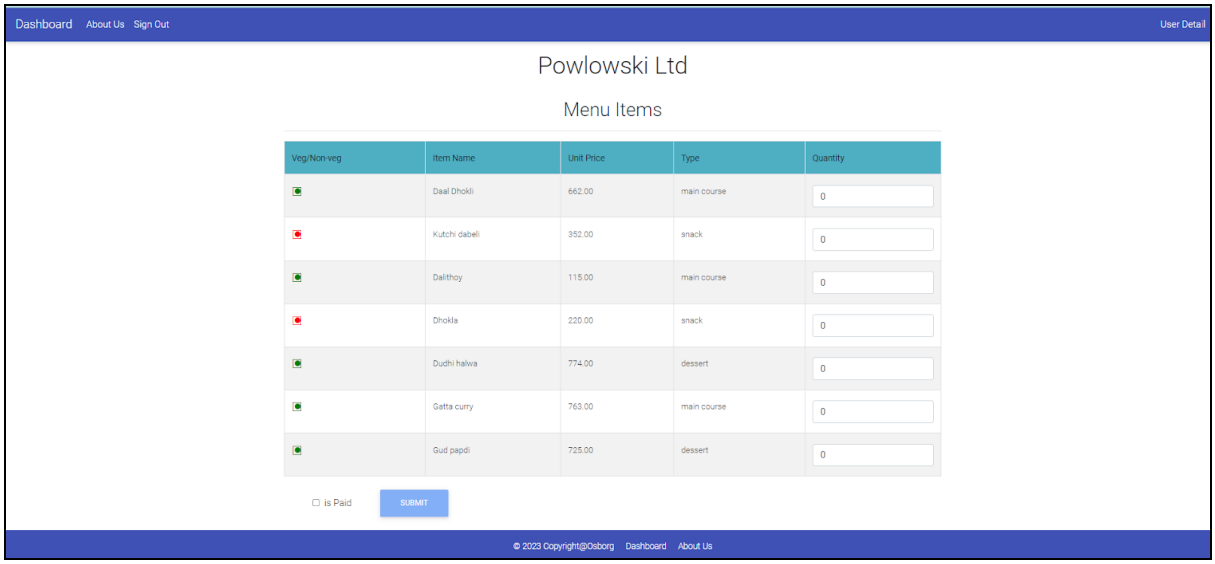
- At the time of signup, they enter their personal details. These changes are directly reflected in the customer's table. Since customers can add multiple addresses, they can also access the customer address table.



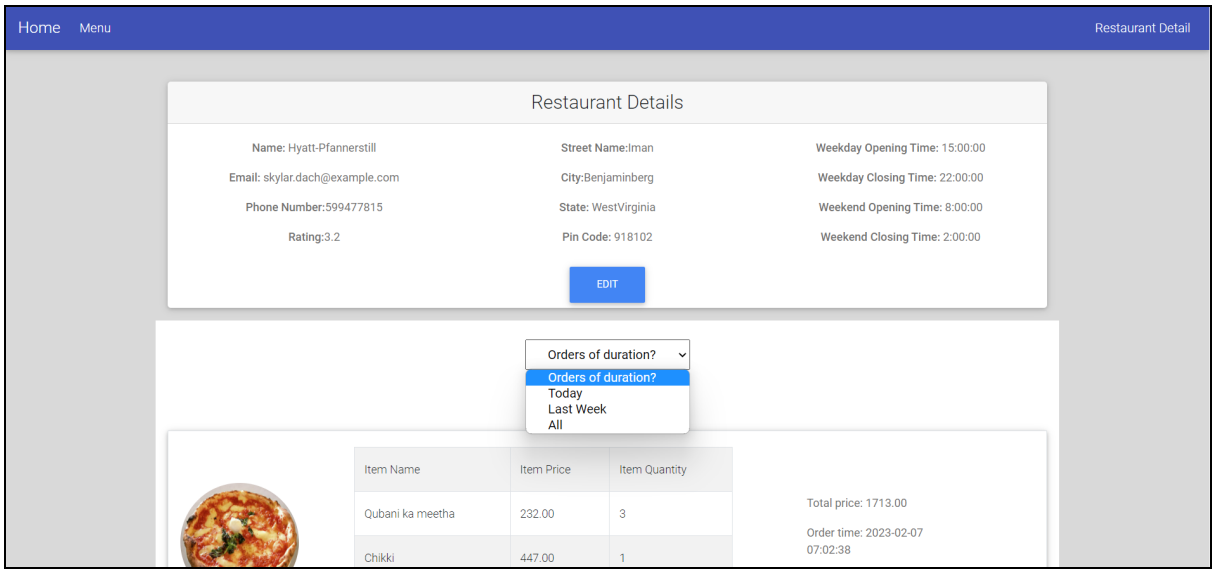
- Place an order from any restaurant to create a new entry in the orders table. They can also view various cuisines, allowing them to read data from the cuisine-type table.



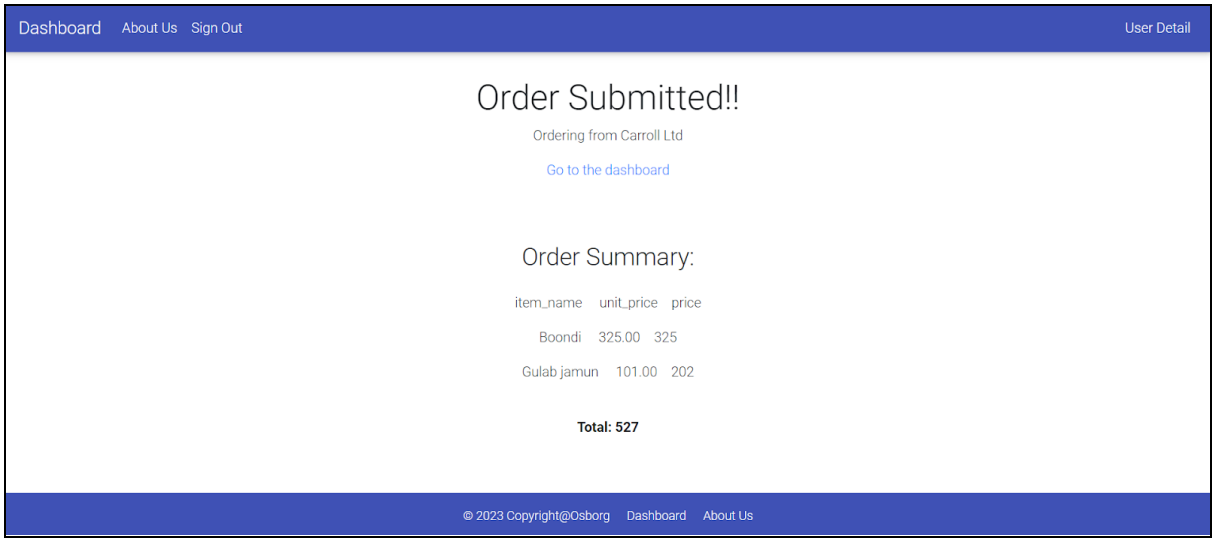
- They can view the menu for any restaurant, thus having read access to the menu items table.



- Choose any addresses they provided for the order, reflecting the corresponding change in the delivery_detail table.



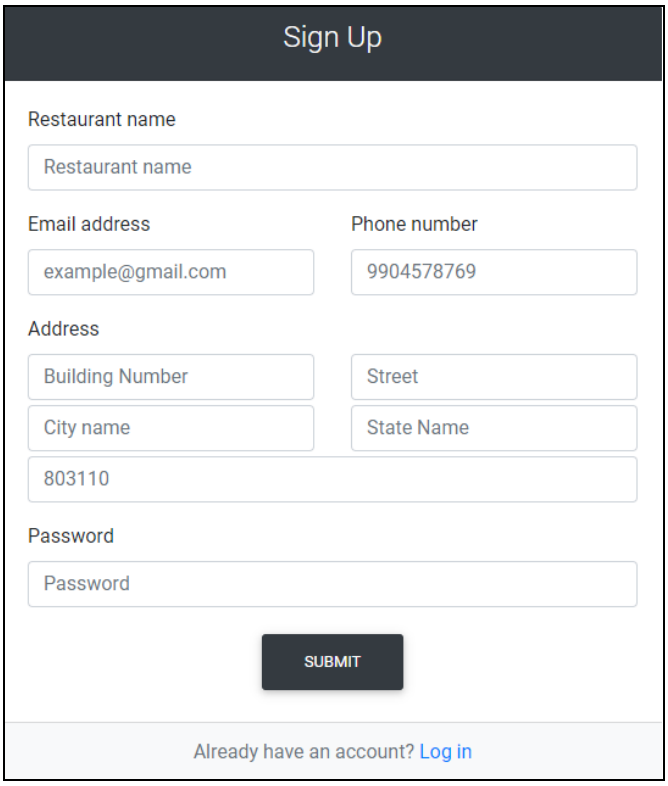
- Customers can also edit their order's payment details, thus having access to the payments table. They also can view their earlier orders, whose data they can view from the order_items table.



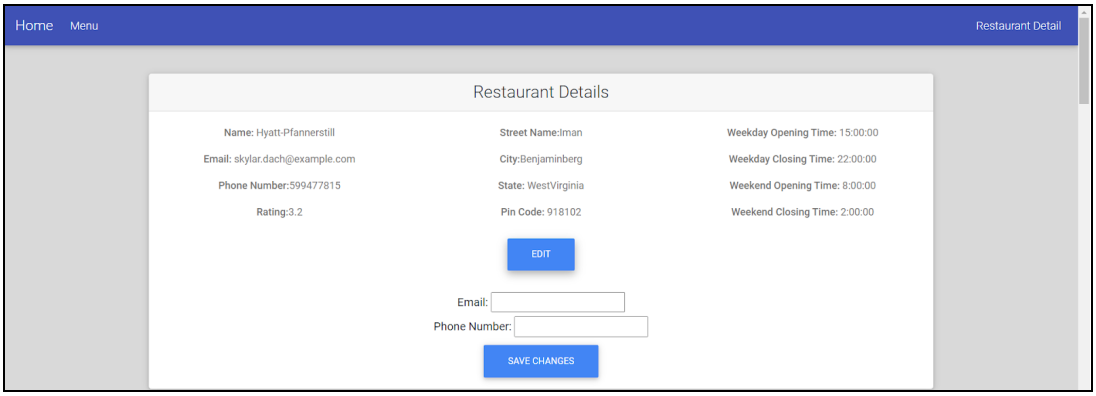
Restaurant:

The restaurant owners have the following rights and privileges that enable them to make the following changes to the database:

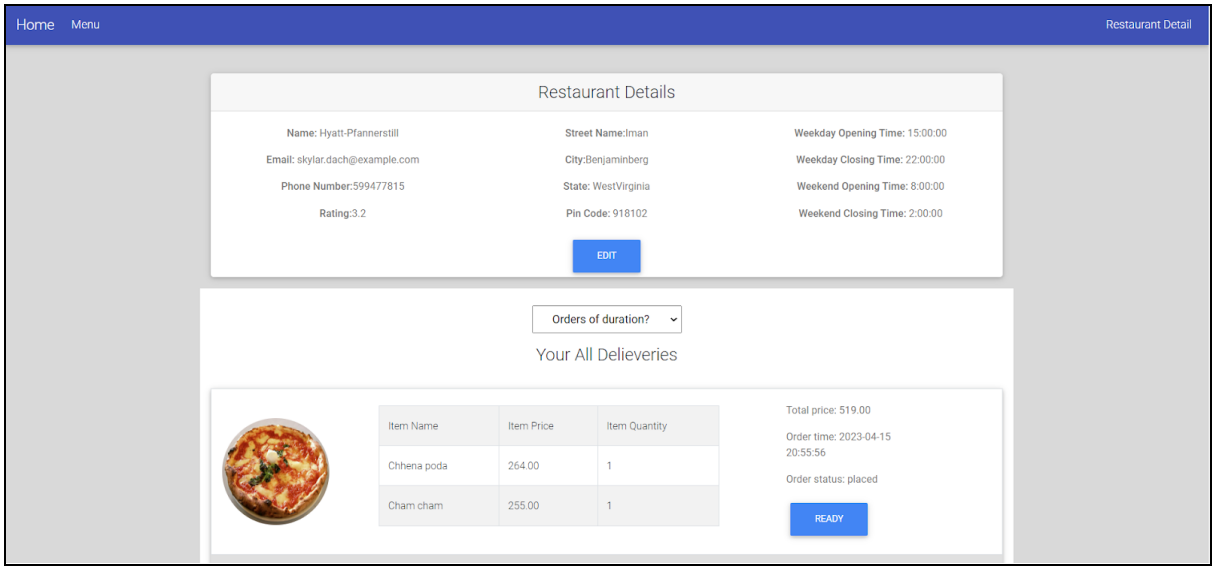
- At the time of signup, they enter their personal details. These changes are directly reflected in the restaurant table.



- They can edit their personal details, like email, phone number, etc., at any point. These changes will be directly reflected in the restaurant's table.



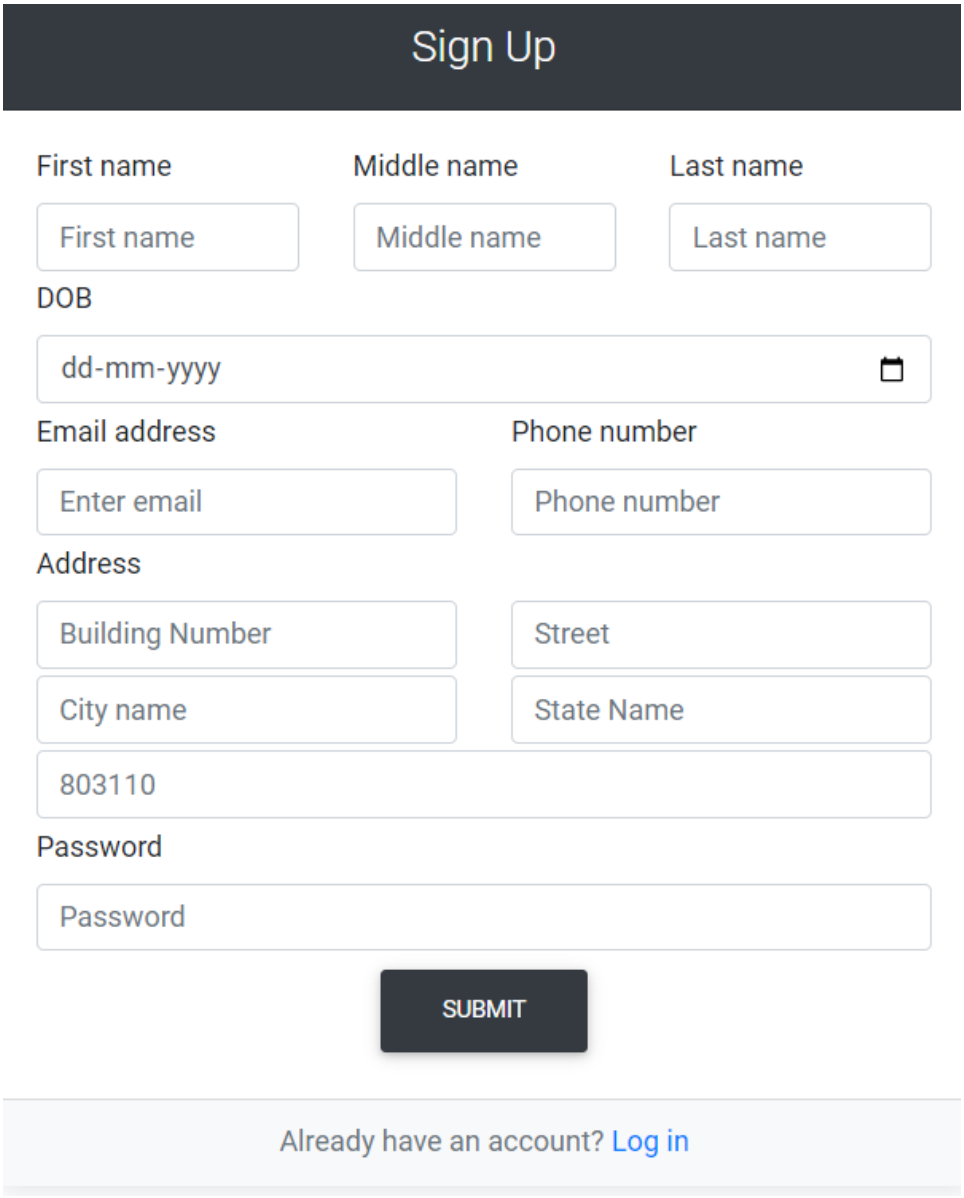
- They can change the order status by flagging when the food is ready. This change is directly reflected in the orders table.



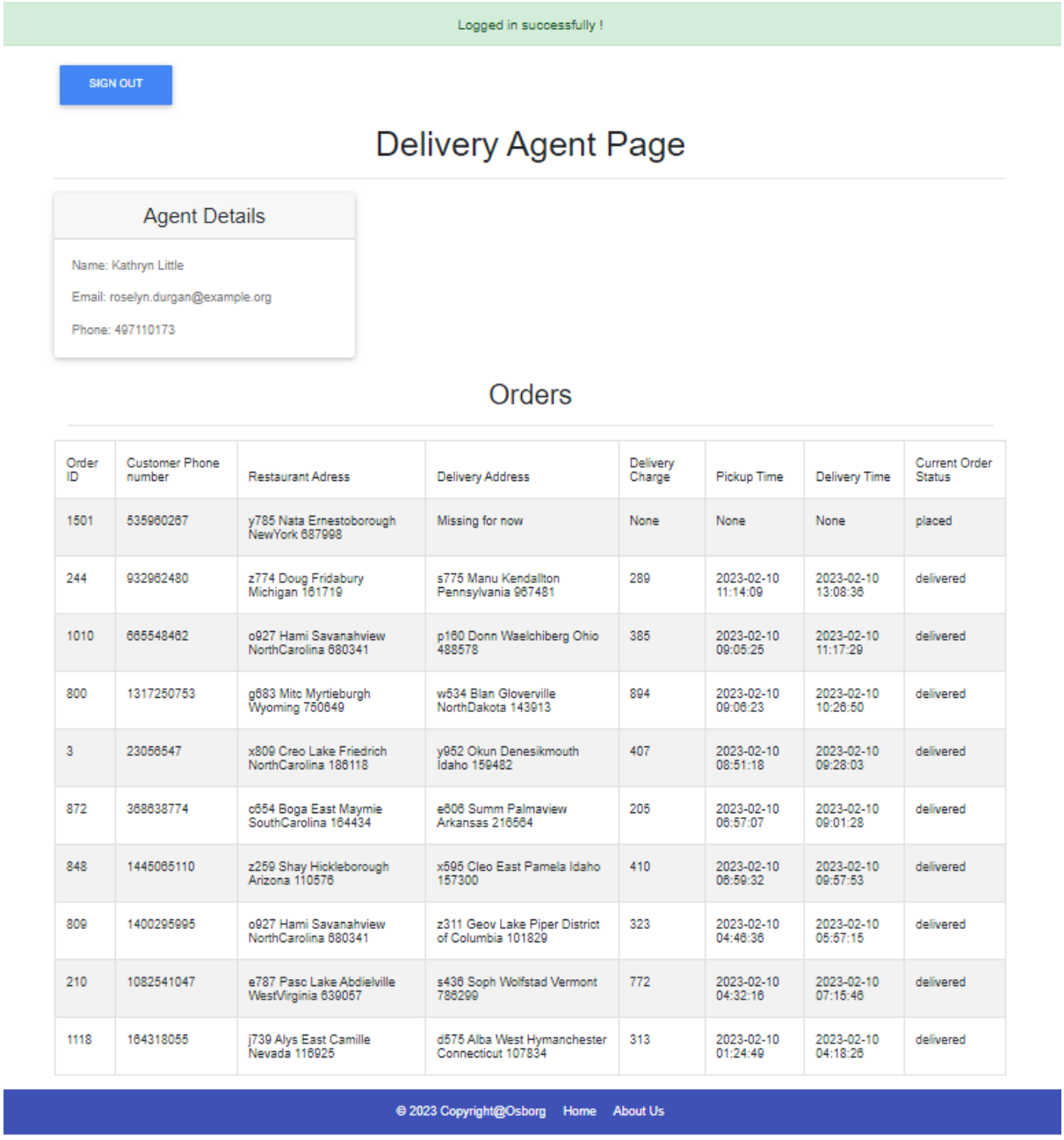
- They can also change and update the items from their menu. That gives them access to edit menu_items and cuisine_type tables.

Delivery Agent:

- At the time of signup, they enter their personal details. These changes are directly reflected in the delivery agent table.



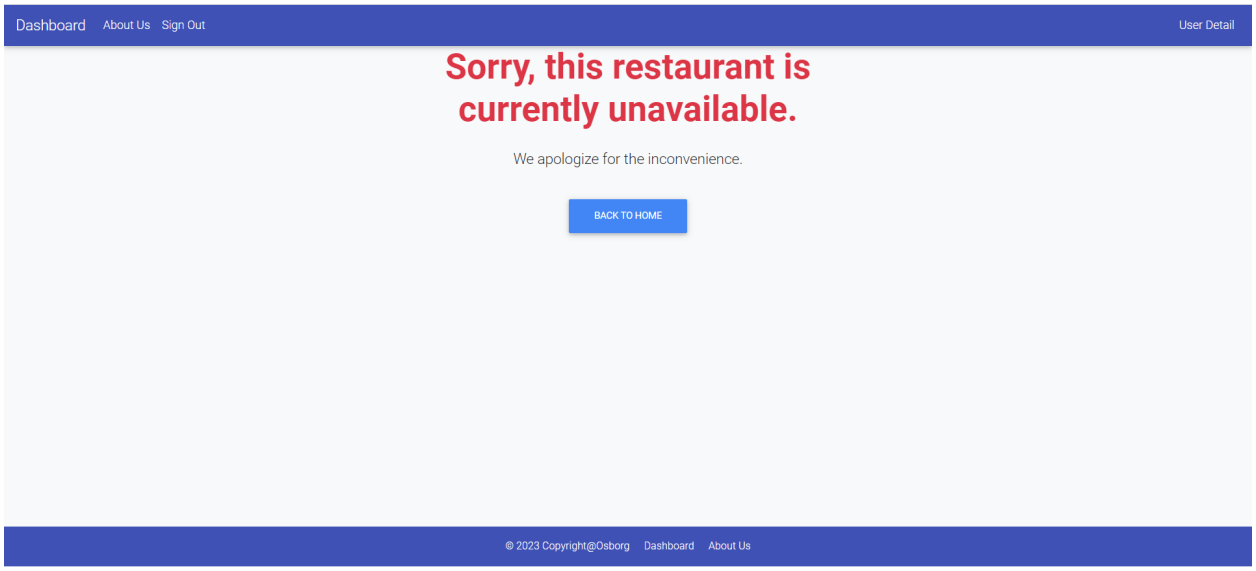
- They can view the orders to be picked and choose to take up the order. This gives them access to the view and edits access to the order table



3.2.1

We have a table named 'menu_item' that is accessed by both customers and restaurants. Customers only read from it, while restaurants write to it. To prevent conflicts, we need to use a locking mechanism to ensure that these operations do not occur simultaneously. This is important because we do not want a customer to read menu items that are being updated by a restaurant.

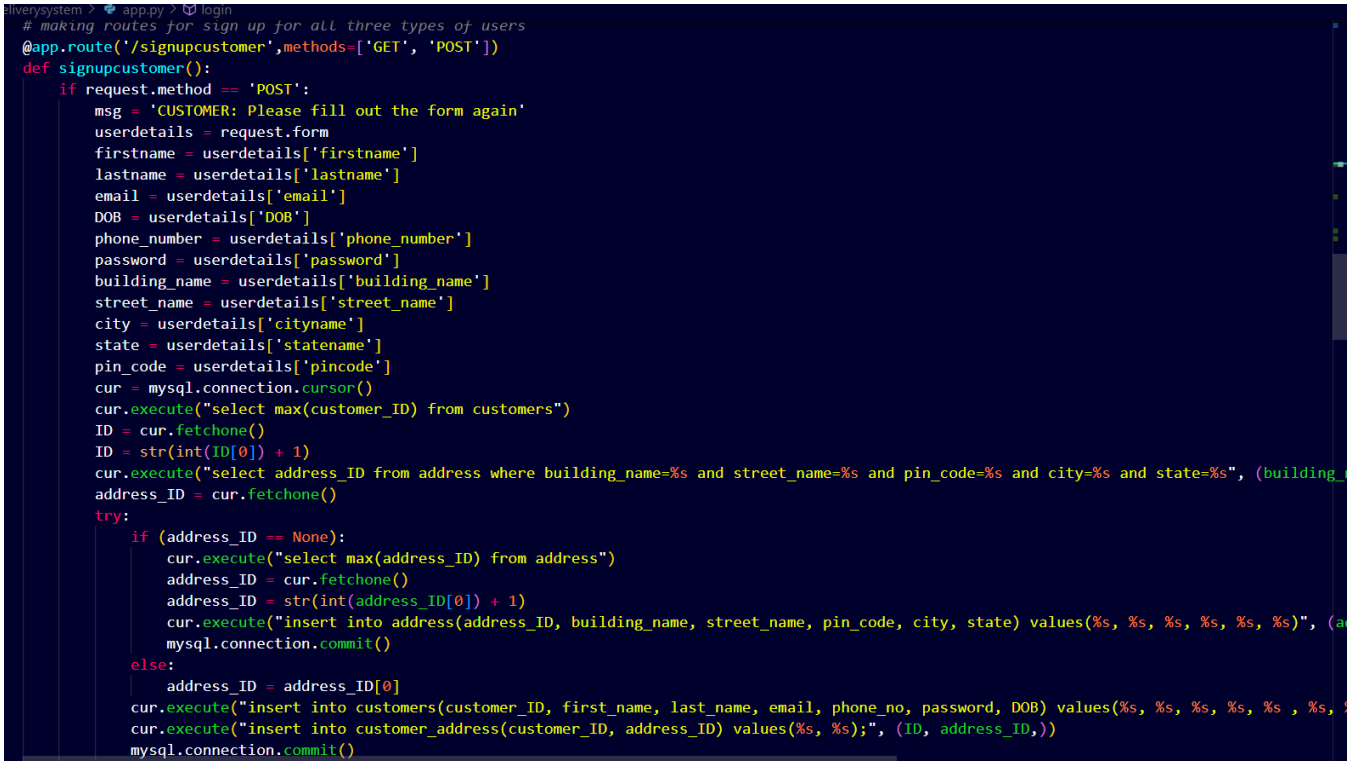
To implement this, we use a lock as soon as a restaurant begins editing their menu. If a customer tries to access that restaurant's menu during this time, we redirect them to a page that says 'restaurant currently unavailable'. We release the lock once the menu update is complete.



This is the page a customer sees when they are trying to access the menu of a restaurant that the restaurant is currently updating.

3.2.2

We have made multiple changes in the frontend part also but as the question asks about the changes in the backend part we are mentioning the snippets of the code for the backend changes.
Adding customer address on the sign-up page.



```
eliverysystem > app.py > login
# making routes for sign up for all three types of users
@app.route('/signupcustomer',methods=['GET', 'POST'])
def signupcustomer():
    if request.method == 'POST':
        msg = 'CUSTOMER: Please fill out the form again'
        userdetails = request.form
        firstname = userdetails['firstname']
        lastname = userdetails['lastname']
        email = userdetails['email']
        DOB = userdetails['DOB']
        phone_number = userdetails['phone_number']
        password = userdetails['password']
        building_name = userdetails['building_name']
        street_name = userdetails['street_name']
        city = userdetails['cityname']
        state = userdetails['statename']
        pin_code = userdetails['pincode']
        cur = mysql.connection.cursor()
        cur.execute("select max(customer_ID) from customers")
        ID = cur.fetchone()
        ID = str(int(ID[0]) + 1)
        cur.execute("select address_ID from address where building_name=%s and street_name=%s and pin_code=%s and city=%s and state=%s", (building_
        address_ID = cur.fetchone()
        try:
            if (address_ID == None):
                cur.execute("select max(address_ID) from address")
                address_ID = cur.fetchone()
                address_ID = str(int(address_ID[0]) + 1)
                cur.execute("insert into address(address_ID, building_name, street_name, pin_code, city, state) values(%s, %s, %s, %s, %s, %s)", (a
                mysql.connection.commit()
            else:
                address_ID = address_ID[0]
            cur.execute("insert into customers(customer_ID, first_name, last_name, email, phone_no, password, DOB) values(%s, %s, %s, %s, %s , %s, %
            cur.execute("insert into customer_address(customer_ID, address_ID) values(%s, %s);", (ID, address_ID,))
            mysql.connection.commit()
```

Adding restaurant address on the signup page.

```
@app.route('/signuprestaurant', methods=['GET', 'POST'])
def signuprestaurant():
    if request.method == 'POST':
        msg = 'Restaurant please fill out the form again'
        restdetail = request.form
        name = restdetail['name']
        email = restdetail['email']
        phoneno = restdetail['Phone number']
        password = restdetail['password']
        # Keeping weekend_time and weekday_time as null values
        # For now keeping rest_address from our side
        cur = mysql.connection.cursor()
        cur.execute('select max(restaurant_ID) from restaurant')
        ID = cur.fetchone()
        ID = str(int(ID[0]) + 1)
        building_name = restdetail['buildingnumber']
        street_name = restdetail['streetname']
        city = restdetail['cityname']
        state = restdetail['statename']
        pin_code = restdetail['pincode']
        cur.execute("select address_ID from address where building_name=%s and street_name=%s and pin_code=%s and city=%s and state=%s", (building_
        address_ID = cur.fetchone()
        try:
            if (address_ID == None):
                cur.execute("select max(address_ID) from address")
                address_ID = cur.fetchone()
                rest_address = str(int(address_ID[0]) + 1)
                cur.execute("insert into address(address_ID, building_name, street_name, pin_code, city, state) values(%s, %s, %s, %s, %s, %s)", (r
                mysql.connection.commit()
            else:
                rest_address = address_ID[0]
            cur.execute('insert into restaurant(restaurant_ID, name, email, phone_number, rest_address, password, weekday_time, weekend_time) value
        except:
```

Storing multiple addresses and allowing customers to choose a particular address for delivery.

```
    cursor.execute("SELECT * FROM Customers WHERE customer_ID = %s", (ID,))
    account = cursor.fetchone()
    if account:
        cursor.execute("select a.address_ID, building_name, street_name, city, state, pin_code from address a inner join customer_address c  on a.a
        addressCust = cursor.fetchall()
        addresses = []
        for address in addressCust:
            temp={
                'address_ID':address[0],
                'building_no':address[1],
                'street_no':address[2],
                'city':address[3],
                'state':address[4],
                'pin_code':address[5]
            }
            addresses.append(temp)

        cursor.execute("SELECT Orders.order_ID, Orders.order_placed_time, Orders.order_status, restaurant.name as restaurant_name, order_totals.net_
        orders_by_cust = cursor.fetchall()
        orders = []
        for placed_order in orders_by_cust:
            temp = {
                    'ID': placed_order[0],
                    'rest_name': placed_order[3],
                    'time': placed_order[1].strftime('%Y-%m-%d %H:%M:%S'),
                    'status': placed_order[2],
                    'price': placed_order[4]
            }
            orders.append(temp)
        context = {
            "first_name":account[1],
            "last_name": account[2],
            "email": account[4],
```

We have added the opening time and closing time for the restaurants. We also provide the option to edit the email id and phone number of a restaurant. Moreover, we also allow restaurants to do multiple queries on their placed/completed orders based on today, last week and all.

```
# Update for assign 04

order_update_id = request.form.get('order_update')
if (order_update_id != None):
    cur.execute("update orders set order_status=%s where order_ID=%s", ("ready", order_update_id,))
    mysql.connection.commit()

try:
    query = "SELECT Orders.order_ID, Orders.order_placed_time, Orders.order_status, order_totals.net_price FROM Orders inner join restaurant on
    if request.method == 'POST' and 'email' in request.form and 'phone' in request.form:
        email = request.form['email']
        phone = request.form['phone']
        cur.execute("update restaurant set email=%s where restaurant_ID=%s;",(email, rest_ID,))
        cur.execute("update restaurant set phone_number=%s where restaurant_ID=%s;",(phone,rest_ID,))
        mysql.connection.commit()
        cur.execute(query, (rest_ID,))
    elif request.method == 'POST' and 'duration' in request.form:
        duration = request.form['duration']
        if(duration=="all"):
            cur.execute(query, (rest_ID,))
        elif(duration=="today"):
            duration = str(datetime.date.today())
            cur.execute("SELECT Orders.order_ID, Orders.order_placed_time, Orders.order_status, order_totals.net_price FROM Orders inner join r
        elif (duration=="lastweek"):
            base = datetime.datetime.today()
            week_time = str(base - datetime.timedelta(days=6)).split(" ")[0]
            cur.execute("SELECT Orders.order_ID, Orders.order_placed_time, Orders.order_status, order_totals.net_price FROM Orders inner join r
    else:
        cur.execute(query, (rest_ID,))
    orders_rest = cur.fetchall()
except:
    cur.execute(query, (rest_ID,))
    orders_rest = cur.fetchall()
# update ends for assign 04
```

3.3.1

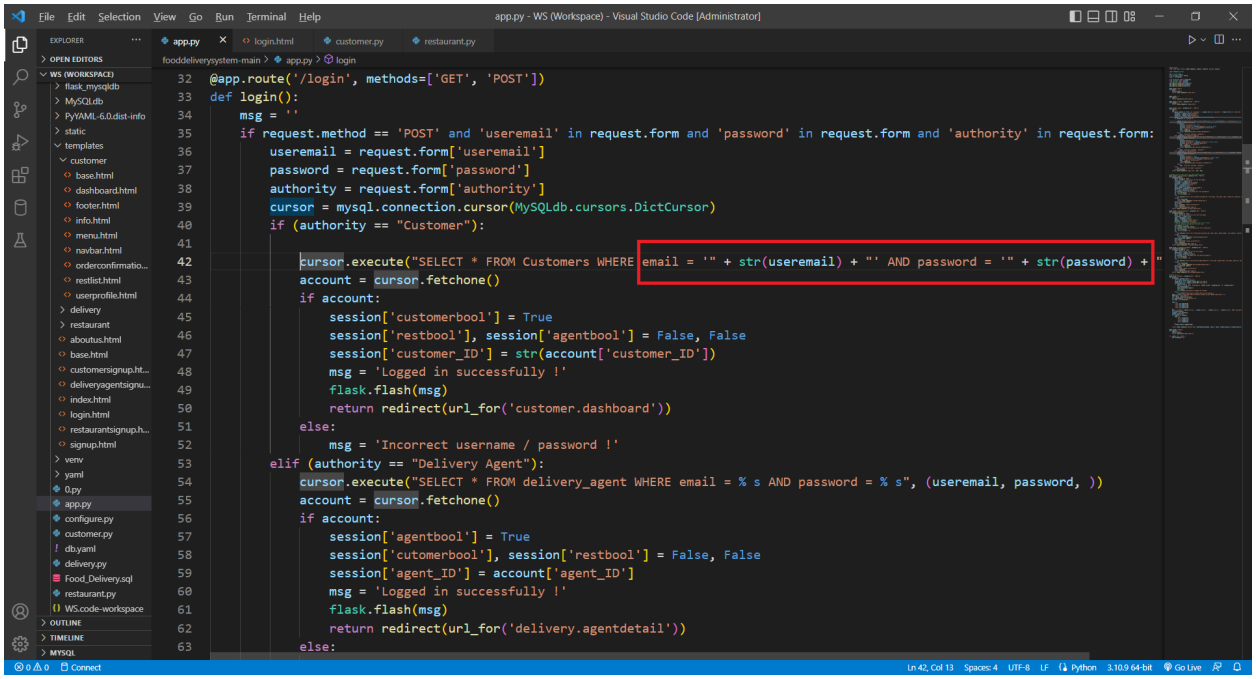
Attacks and Defenses

Link to the Folder containing Recordings/Snapshots of Attacks, Defenses, Results

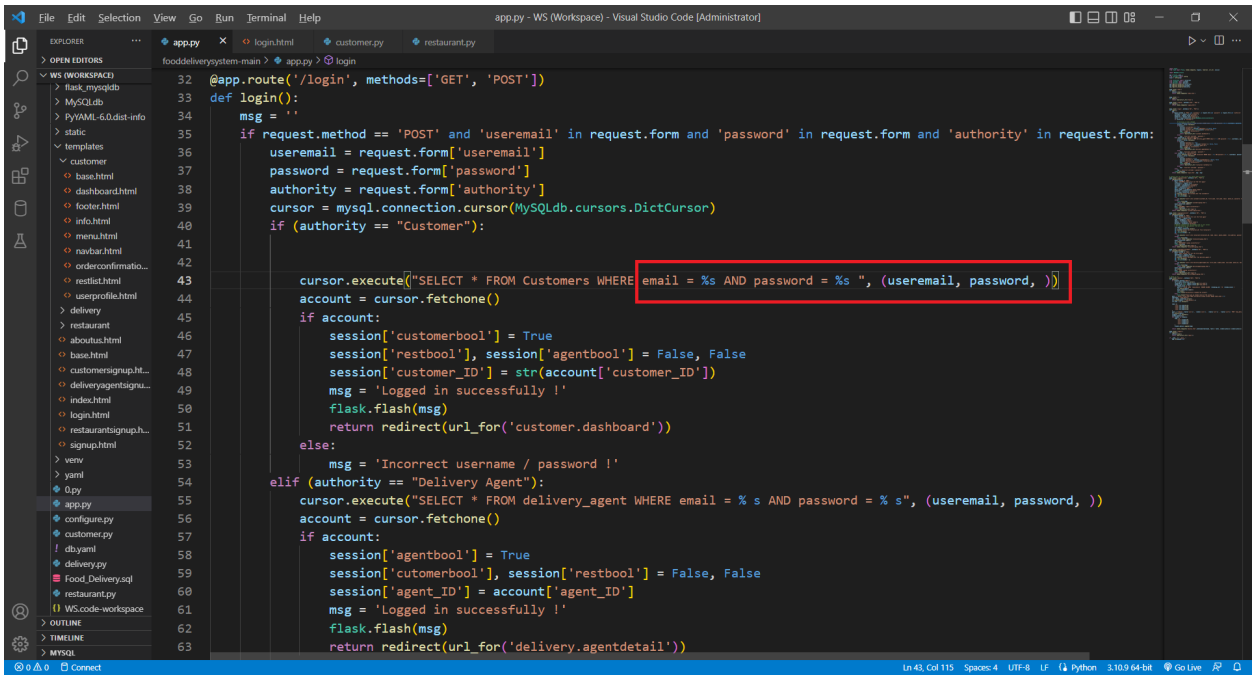
1) **Incorrect Queries** (That fail on runtime):

- **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**:
    - Email : ' OR select * from nsdkjfnfn --
    - 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 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:

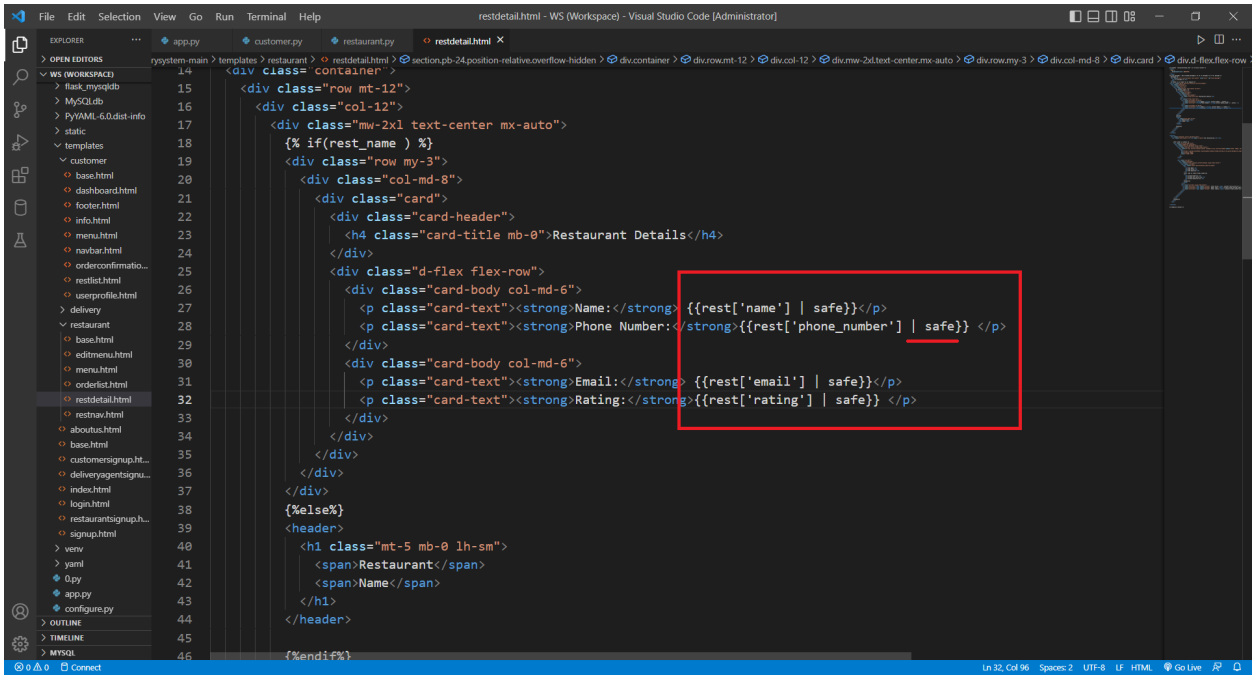


2) **Cross Site Scripting** (XSS) :

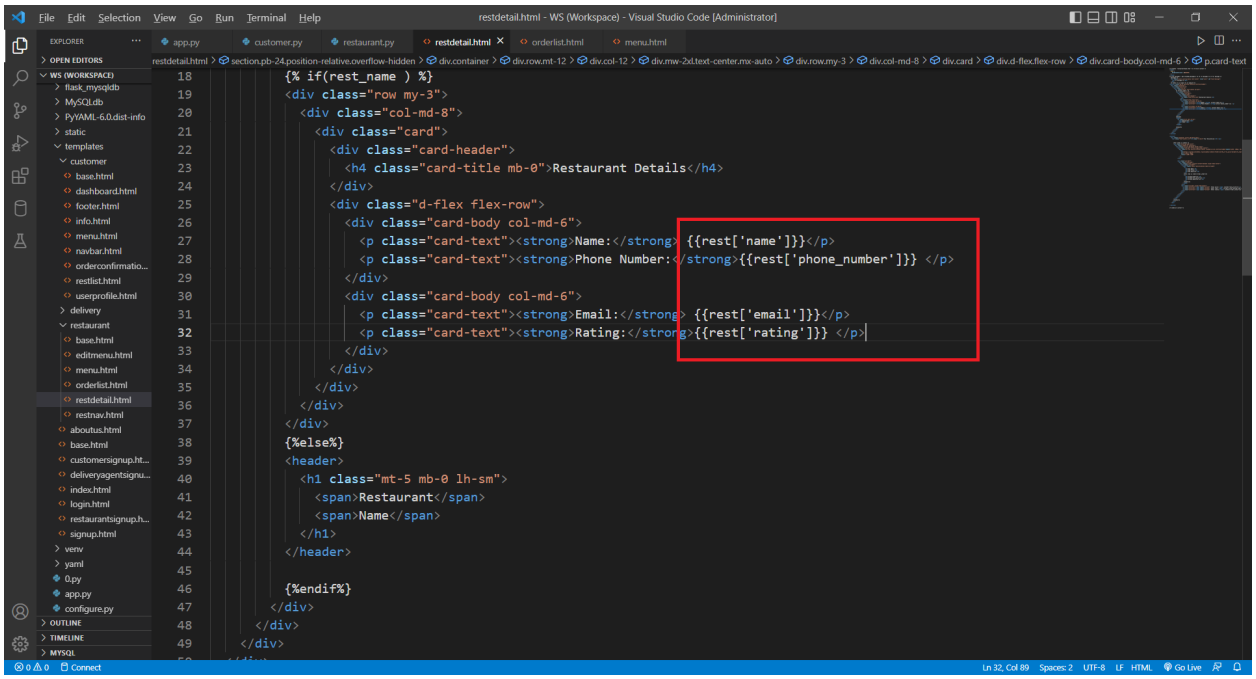
- **Description** : Cross-site scripting (XSS) is a type of security vulnerability that allows an attacker to inject malicious HTML/JavaScript code into the web page. The attacker can exploit a vulnerability in a web application by injecting a script or code that is executed when the user visits the 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** :
    - Adding a restaurant named as follow:
      <script> alert("YOUR ACCOUNT IS HACKED"); </script>
    - 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 script instead of just displaying it as plain text.
    - This script will give alert if there is some other script written then the user can be forwarded to some other malicious website.
- **Defense** : We had used “safe” keyword-function in them html code-file for displaying the data collected from database, as {{rest[‘name’] | safe}}. This safe function considers the data from the database to be safe and doesn't use any filters. For defense against XSS, we removed all such “safe” functions. After this change the html will consider data from the database as a plain text and simply display it without running it as a script.

Before:



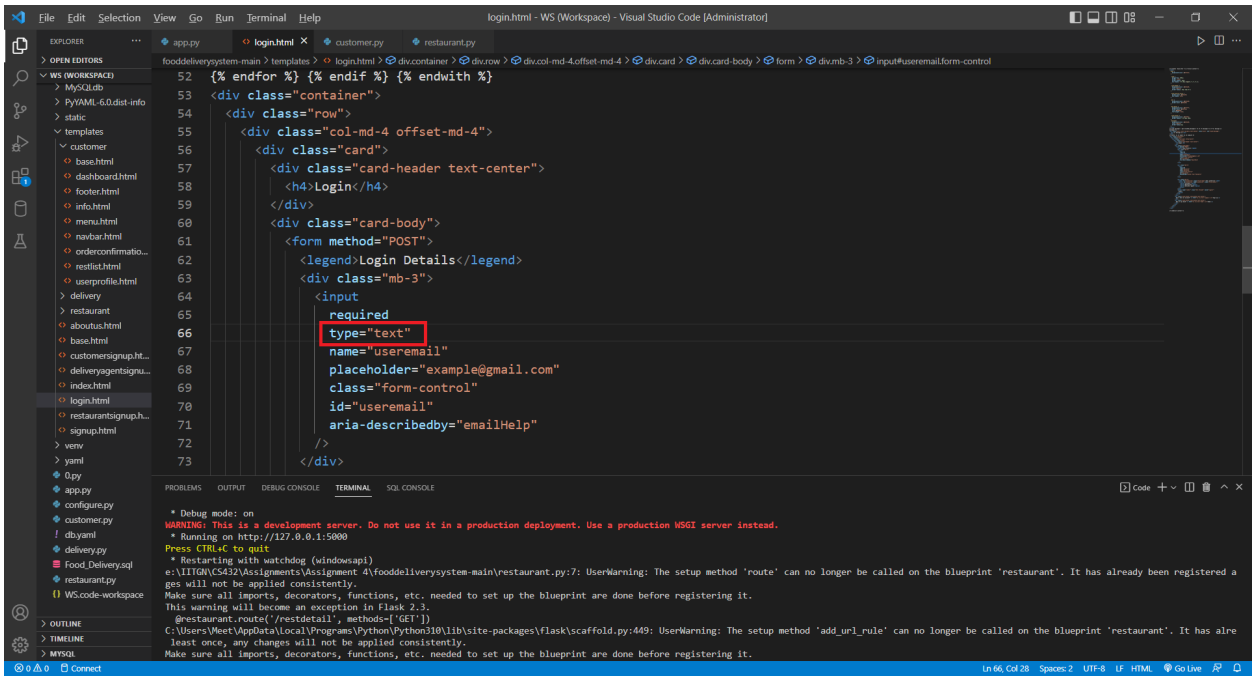
After:



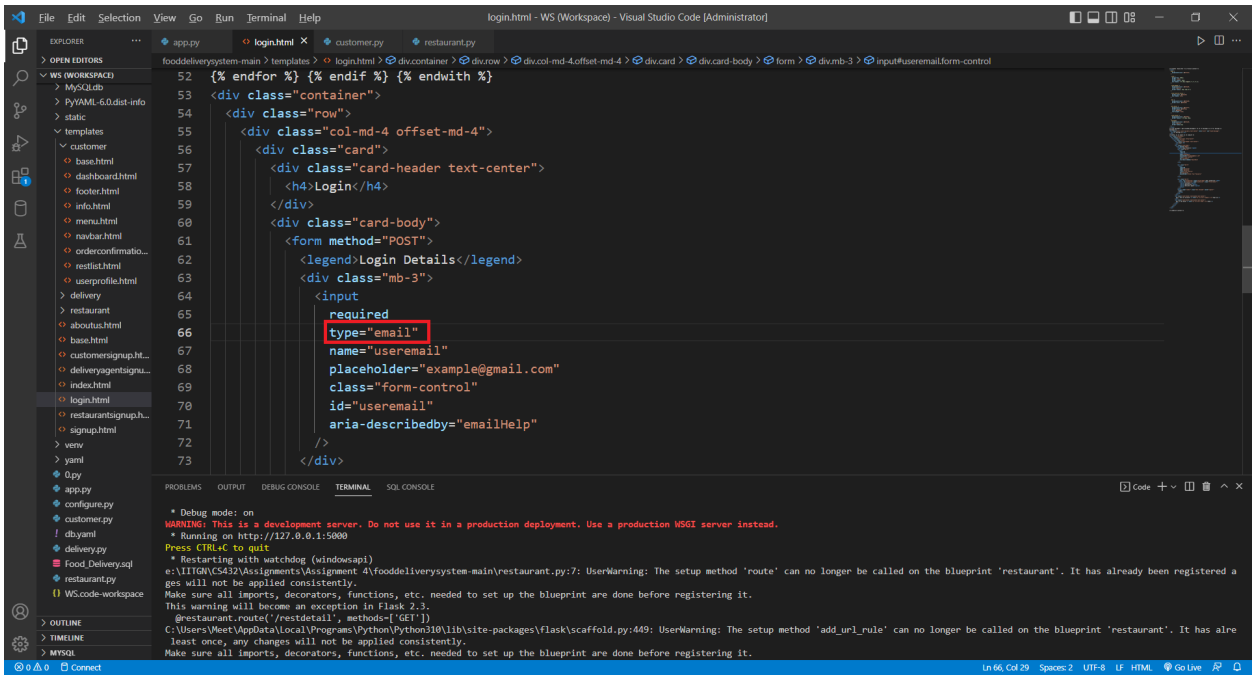
3) **Tautology** :

- **Description** : This is an SQL Injection attack. We write a query that always evaluates to TRUE, so that it will allow us to proceed anyways.
- **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:



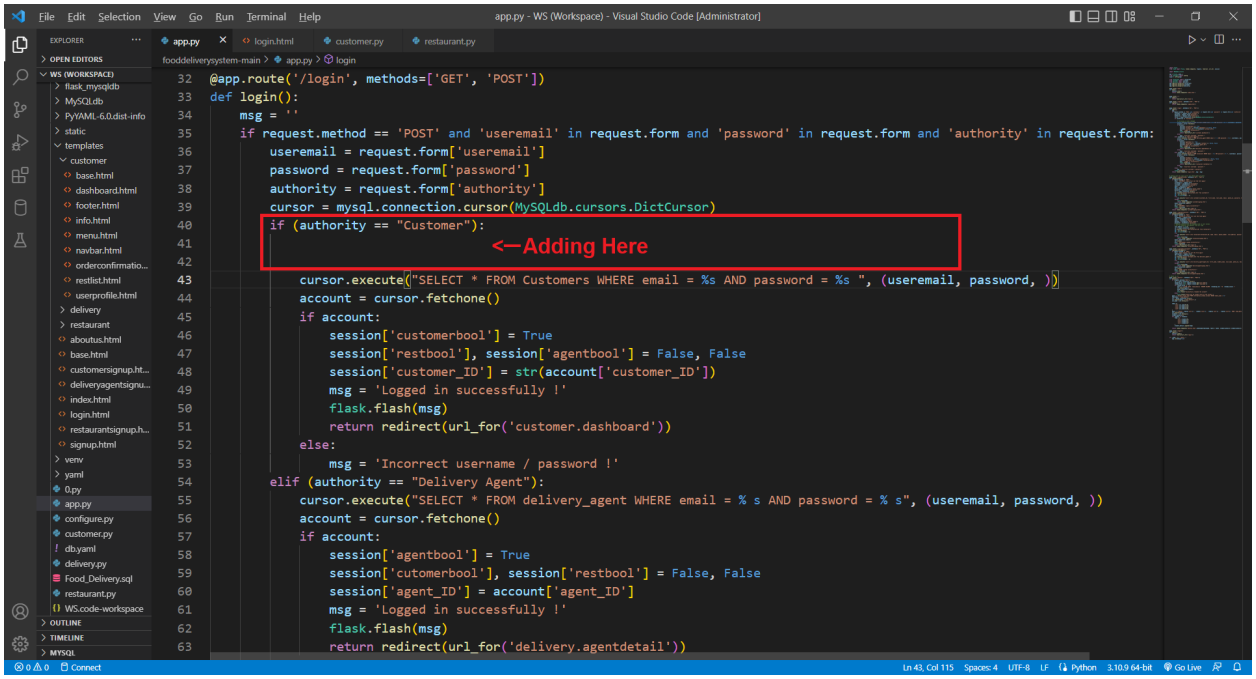
4) **Union Query** :

- **Description** : This also is an SQL Injection attack. We do not care about the query that is already inside the database system. Instead, we finish the original query in a way that its output is NULL. Then, we put the UNION keyword, and write another query that we want to be executed. (Note : We use the information obtained from the first attack to write these queries.)
- **Attack** :
    - Email : ' UNION select * from customers where cusomer_ID=5 --
    - Email : ' UNION select name, phone_no from customers --
    - Email : ' UNION drop table customers --
- **Defense** : We can add both the defenses mentioned above. Instead of them, we can perform a quality check on the input obtained from the fields, about whether or not they contain any sub queries. We search for the keyword UNION and check if the following statement can be executed as another query. If so, we reject showing the output and throw an alert signal.

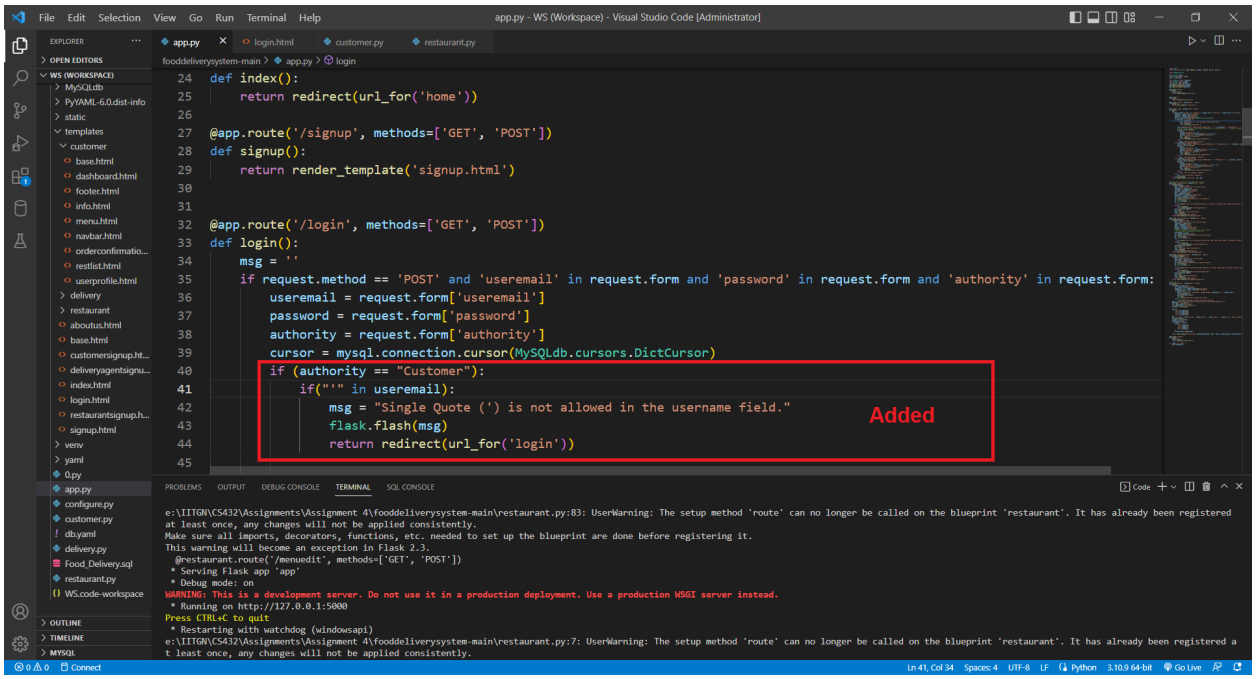
5) **Piggy-Backed Queries** :

- **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, piggy-backs 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. Defenses from the previous attacks can also be used for this type of attack.

Before:



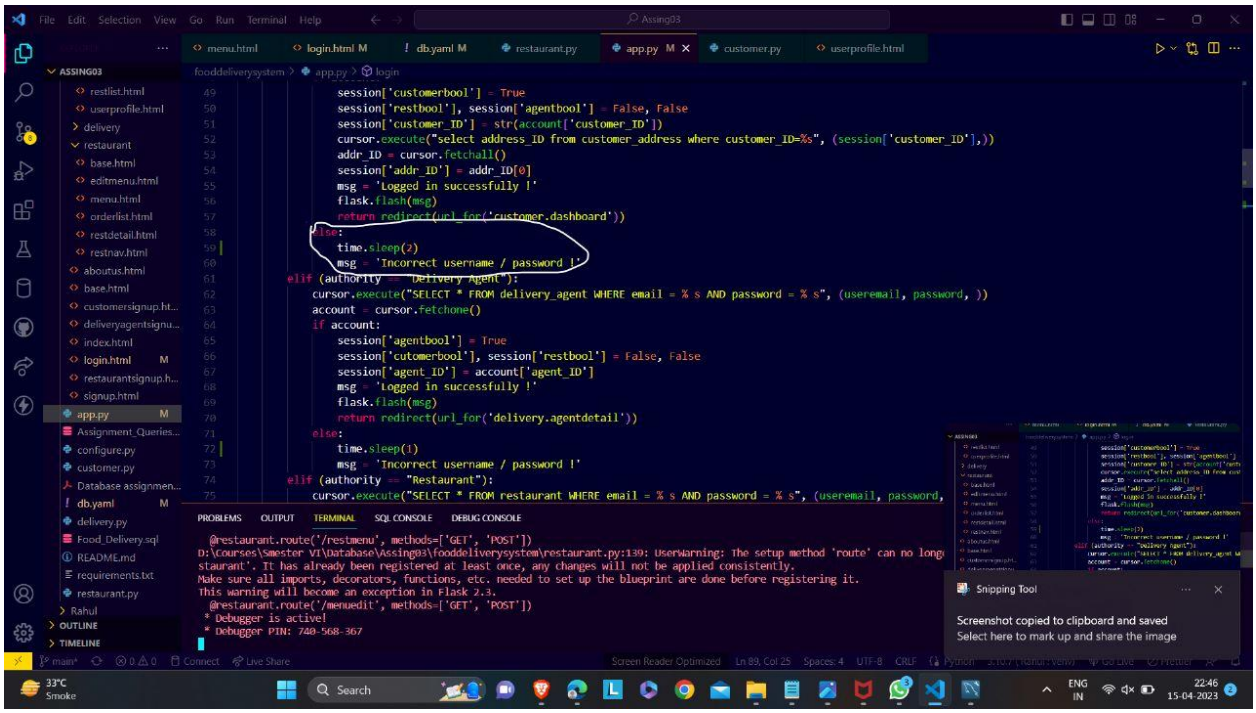
After:



6) **Brute Force** :

- **Description** : A brute force attack is a type of cyber attack where an attacker attempts to gain unauthorized access to a system or data via trying every possible password or encryption key until the correct one is found.
- **Attack** :
  - For some valid username we will try all possible password combinations. If the possible combinations are very less or if the user has a weak password then we can brute force the password.
  - We have used “hydra tool” for this purpose.
  - We had username.txt and password.txt as possible passwords and usernames.
  - `$ hydra -l username.txt -p password.txt 127.0.0.1 8000 -o safe.txt`
  - This code will go through combinations of username and passwords and return correct combinations in the safe.txt file.

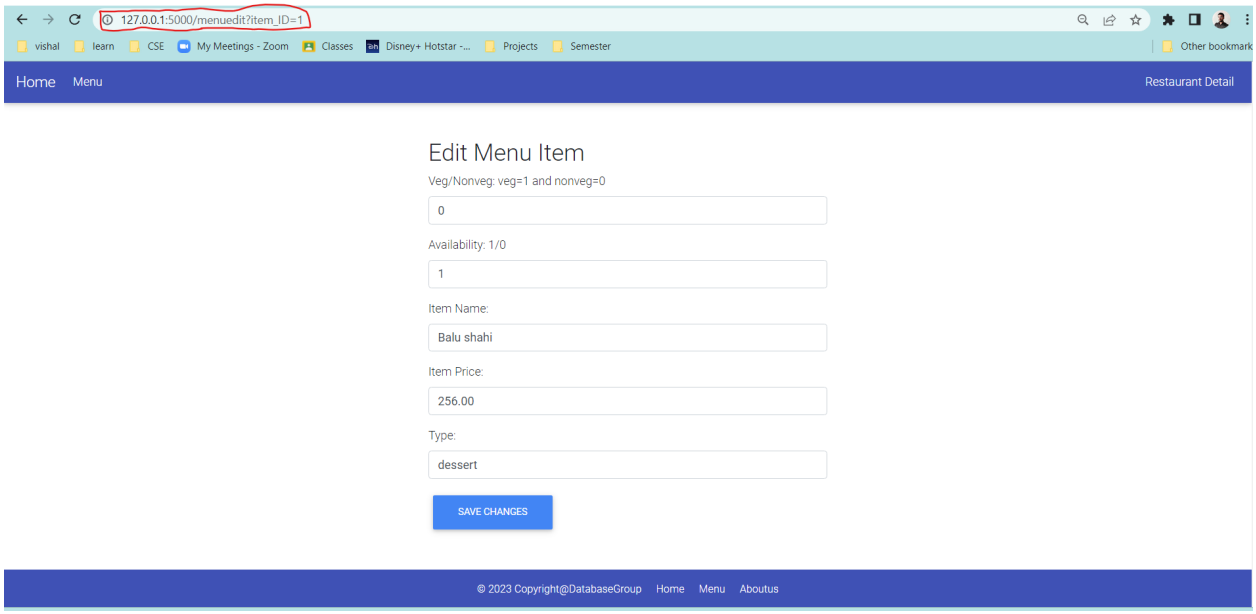
- **Defense** : To defend against brute force attacks we need to make passwords stronger. For this we can force the users to choose a password which has length more than 8 or 10 characters. In addition to that, we can impose a constraint that those passwords should include at least one unit of the following : capital letter, small letter, digit and special character. We have also added 3 seconds delay between 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 optimized our defense against brute force attacks.



7) **Frontend attack**:

- When the restaurant menu goes to edit a certain menu item then through playing with url a restaurant is able to change the details of any menu item in the database.
- We prevented it by checking whether the menu_item belongs to the current restaurant_ID stored in session details.
- We flash up the message if the session's restaurant id doesn't match with the one related to the menu_item being edited.

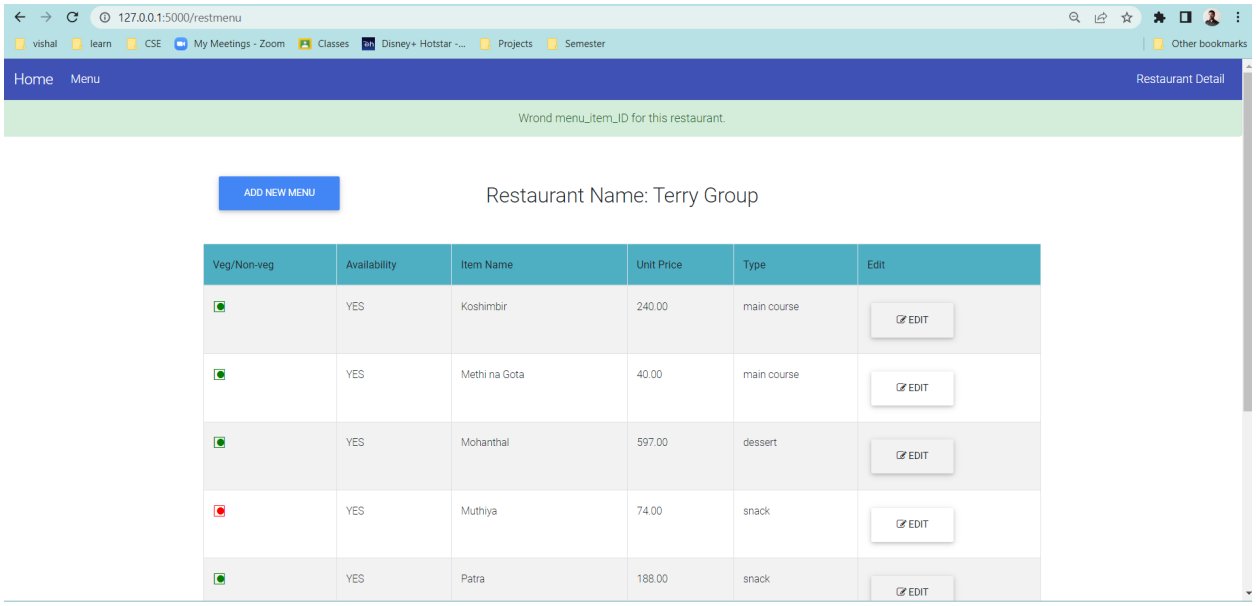
Location of vulnerability:



Before and after:

```
    item_ID = request.values.get("item_ID")
    print(item_ID)
    cur = mysql.connection.cursor()
    cur.execute("select item_ID, veg, availability, name,
    item_detail = cur.fetchone()
    if (item_detail is None):
        flash("Need to login as Restaurant")
        return redirect(url_for('login'))
    item = {
        'ID': item_detail[0],
        'veg': item_detail[1],
        'availability': item_detail[2],
        'name': item_detail[3],
        'unit_price': item_detail[4],
        'item_type': item_detail[5]
    }
    return render_template('restaurant/editmenu.html', it
if request.method == 'POST':
    item_ID = request.values.get("item_ID")
    veg_novveg = request.values.get("veg")
    availability = request.values.get("availability")
    item_name = request.values.get("name")
```

```
163
164     item_ID = request.values.get("item_ID")
165     rest_ID = session['restaurant_ID']
166     cur = mysql.connection.cursor()
167     cur.execute("select item_ID, veg, availability,
168     item_detail = cur.fetchone()
169     if (item_detail is None):
170         flash("Need to login as Restaurant")
171         return redirect(url_for('login'))
172     if (item_detail[6] == session['restaurant_ID']):
173         item = {
174             'ID': item_detail[0],
175             'veg': item_detail[1],
176             'availability': item_detail[2],
177             'name': item_detail[3],
178             'unit_price': item_detail[4],
179             'item_type': item_detail[5]
180         }
181         #setting the is_editing_menu to 1
182         cur.execute("UPDATE restaurant SET is_editi
183         mysql.connection.commit()
184         return render_template('restaurant/editmenu
```



3.3.2

These are constraints in our database management system that enforce rules and restrictions on the data stored in the database tables. These we have implemented in the submitted SQL-dump. They hold true even after the changes implemented after the second feedback.

1. delivery_rating: This constraint ensures that the delivery rating given by the customer is between 0 and 5 (inclusive).
2. delivery_charges: This constraint ensures that the delivery charges for an order are non-negative (i.e., greater than or equal to 0).
3. delivery_time: This constraint ensures that the delivery time for an order is greater than the pickup time.
4. rating: This constraint ensures that the restaurant rating the customer gives is between 0 and 5 (inclusive).
5. payment_method: This constraint ensures that the payment method used for an order is one of the allowed options (UPI, Netbanking, Debit Card, or Credit Card).
6. amount: This constraint ensures that the amount paid for an order is greater than 0.
7. order_status: This constraint ensures that the order status is one of the allowed options (placed, ready, picked, or delivered). This is the only change committed in our database from the previous version as we have added the option to have the statues “ready”.
8. order_rating: This constraint ensures that the order rating given by the customer is between 0 and 5 (inclusive).

Contributions:

Index	Tasks	Members taking responsibility
1	Getting Feedback	Vishal, Pratik and Juhil
2	Implementing the feedback	Rahul and Vishal
3	SQL injection and XSS and frontend attacks	Prey, Meet and Tejas
4	Prevention of concurrent transaction	Dhruv, Prakram
5	Checking constraints and relations after second feedback	Juhil, Dhiren
6	Documentation	Dablu and Dhiren