

## **CIS 344 FINAL PROJECT “Restaurant Portal”**

**Submitted by: Mohiuddin Hasan**

**Prof. Yanilda Peralta Ramos**

For the database queries:

First we created the database, restaurant\_reservations and the tables.

We create a "customers" table using the “create table”, and create the following attributes - customerId, customerName and contactInfo. The customer ID column is set to auto-increment, which will increment the ID automatically every time we insert a new account and set it as the primary key.

The "reservations" table is created using the same queries. But the customer Id is the foreign key here, which links to the table “customers” which references the customerId of the customer table.

After creating the tables, we populate the tables using insert into “reservations” values().

We create the procedures findReservations, addSpecialRequest and addReservation. The procedure starts with named findReservations with one input of customerId of type integer. This stored procedure is designed to retrieve reservation records from a database based on the specified customer ID. Then we stored another procedure named addSpecialRequest with two inputs: reservationId of type integer and requests of type varchar(200). This stored procedure is designed to update the special requests field in the reservations table for a specific reservation. To add a new reservation to a restaurant's database system, including customer information, reservation details, and optionally, dining preferences we created another procedure named addReservation with several input parameters.

The SQL procedures that I used at the beginning had a couple of issues related to the use of parameter names and variable scoping. Specifically, the parameters customerId and reservationId inside the procedures conflict with the column names in the tables. To avoid this, I used a different naming convention for the parameters. The delimiter is changed to // to properly define the end of the procedure without conflicting with the default ; delimiter.

For the procedure addSpecialRequest, begin ... end is a block that encloses the statements which make up the body of the procedure. In this query the UPDATE statement is used to modify existing records in a table. Here the set key specifies the column(s) to be updated. specialRequests = requests, sets the value of the specialRequests column to the value passed in the requests parameter. The where clause specifies which row(s) to update.

### **Error and changes:**

At the beginning the restaurant database homepage on localhost was not getting connected. Later on, I figured that I had to use the database first before running the restaurantdatabase.py and restaurantserver.py on the localhost:8000/.

## Restaurant Portal

[Home](#) [Add Reservation](#) [View Reservations](#)

### All Reservations

Reservation ID	Customer ID	Reservation Time	Number of Guests	Special Requests
1	1	2024-05-11 18:00:00	4	Window seat preferred
2	2	2024-06-12 19:30:00	2	Vegetarian meal for one guest
3	3	2024-07-10 15:30:00	10	Please serve halal Food
4	4	2024-06-11 18:30:00	5	Vegetarian meal for one guest
5	5	2024-06-10 16:00:00	6	Please sing Happy Birthday for my daughter

However, I had multiple errors while creating the procedure. Since I haven't used the procedure in a SQL before, I had to study on how to create a procedure and why delimiters are used while creating a procedure. Once I inserted the values In the tables, I was able to see the output on the homepage of the Restaurant Database. But I was not able to insert a value in the beginning. The add reservation and view reservation page wasn't working. So based on the provided source file, RestaurantServer.py, I added more rows and columns for the add reservation, view reservation and add customer. Then I added a few more methods on the RestaurantDatabase.py code. addDiningPreference, updateSpecialRequest, and findCustomer methods are the extra methods that I added on the source code. After adding more methods I used the "call procedure" in the mwb file, which essentially invokes that named procedure to execute its code, which can perform various actions such as querying the database, updating data, or performing calculations, depending on what it's designed to do.



### This page isn't working

localhost didn't send any data.

ERR\_EMPTY\_RESPONSE

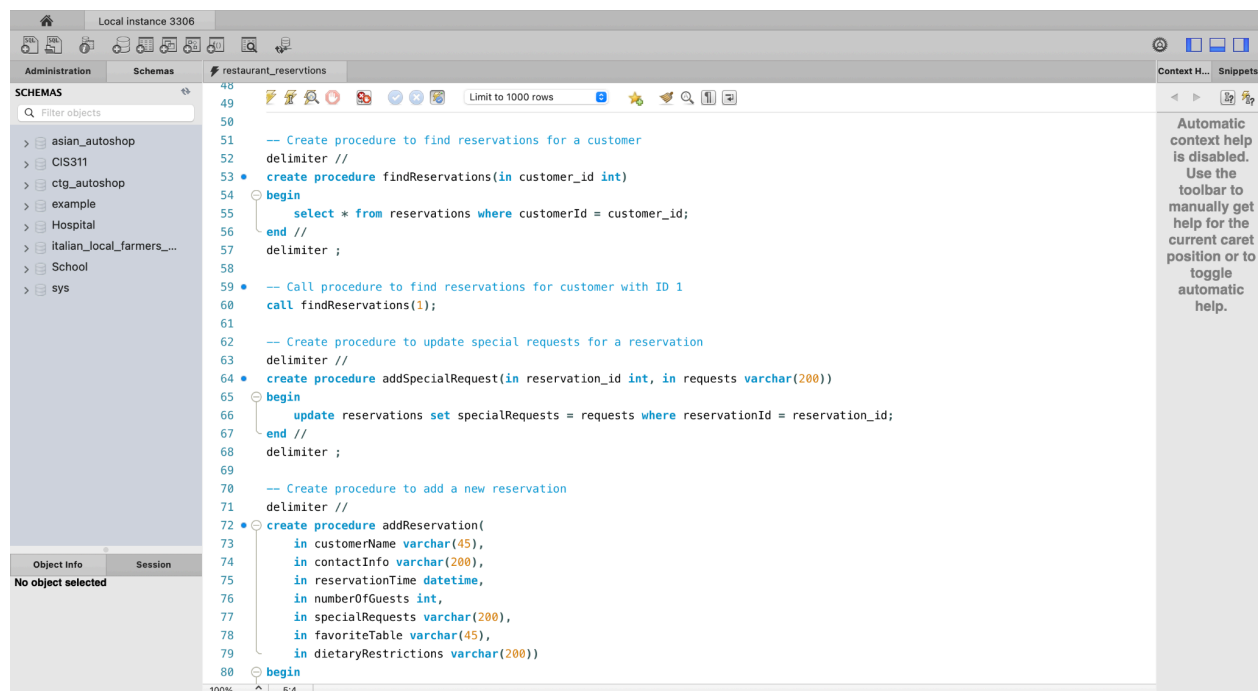
Reload

## Procedure findReservation:

This script sets the delimiter to // to handle the creation of the procedure and then resets the delimiter back to ';' after the procedure is created. It also includes an example call to the procedure with a customer\_id of 1.

## Procedure addSpecialRequest

This stored procedure allows for the efficient management of special requests associated with specific reservations. We use the stored procedure 'addSpecialRequest' with two input parameters, 'reservation\_id' of type int and 'requests' of type varchar(200). The 'begin' keyword marks the beginning of the procedure's executable block. All statements within this block are executed as part of the stored procedure. This SQL statement updates the 'specialRequests' column of the reservations table. It sets the value of 'specialRequests' to the provided requests parameter (requests) for the reservation identified by the reservation\_id parameter (reservation\_id). The end keyword marks the end of the procedure's executable block.



The screenshot shows a SQL IDE window titled 'Local instance 3306'. The 'Schemas' panel on the left lists various databases, including 'restaurant\_reservations'. The main editor displays the following SQL script:

```
48
49
50
51 -- Create procedure to find reservations for a customer
52 delimiter //
53 • create procedure findReservations(in customer_id int)
54 begin
55     select * from reservations where customerId = customer_id;
56 end //
57 delimiter ;
58
59 • -- Call procedure to find reservations for customer with ID 1
60 call findReservations(1);
61
62 -- Create procedure to update special requests for a reservation
63 delimiter //
64 • create procedure addSpecialRequest(in reservation_id int, in requests varchar(200))
65 begin
66     update reservations set specialRequests = requests where reservationId = reservation_id;
67 end //
68 delimiter ;
69
70 -- Create procedure to add a new reservation
71 delimiter //
72 • create procedure addReservation(
73     in customerName varchar(45),
74     in contactInfo varchar(200),
75     in reservationTime datetime,
76     in numberOfGuests int,
77     in specialRequests varchar(200),
78     in favoriteTable varchar(45),
79     in dietaryRestrictions varchar(200))
80 begin
```

The right sidebar contains a message: 'Automatic context help is disabled. Use the toolbar to manually get help for the current caret position or to toggle automatic help.'

## Procedure addReservation

Create procedure initiates the creation of a new stored procedure named addReservation. The 'addReservation' stored procedure is made to add a new reservation to the database, handling scenarios where the customer may or may not already exist. It also provides the flexibility to insert dining preferences along with the reservation details. The create procedure addReservation uses seven input parameters: customerName, contactInfo, reservationTime, numberOfGuests, specialRequests, favoriteTable, dietaryRestrictions. The 'in' keyword is used to define input parameters for the stored procedure. The 'declare' keyword defines a local variable 'custId' of type int is declared to store the customer ID retrieved from the database The select query checks

if a customer with the provided name and contact information already exists in the customers table. If found, it assigns the customer ID to the custId variable. The if condition checks that if the customer does not exist (custId is null), this block inserts a new customer into the customers table using the provided name and contact information. The last\_insert\_id() function retrieves the auto-generated customer ID and assigns it to custId. The insert into statement inserts a new reservation into the reservations table, associating it with the customer identified by 'custId'. It includes reservation time, number of guests, and any special requests provided.

```

-- Update special requests for a reservation
update reservations set specialRequests = requests where reservationId = reservation_id;
end //
delimiter ;

-- Create procedure to add a new reservation
delimiter //
create procedure addReservation(
    in customerName varchar(45),
    in contactInfo varchar(200),
    in reservationTime datetime,
    in numberOfGuests int,
    in specialRequests varchar(200),
    in favoriteTable varchar(45),
    in dietaryRestrictions varchar(200))
79
80 begin
81     declare custId int;
82     -- Check if customer exists
83     select customerId into custId from customers where customerName = customerName and contactInfo = contactInfo limit 1;
84
85     -- If customer does not exist, insert new customer
86     if custId is null then
87         insert into customers (customerName, contactInfo) values (customerName, contactInfo);
88         set custId = last_insert_id();
89     end if;
90
91     -- Insert reservation
92     insert into reservations (customerId, reservationTime, numberOfGuests, specialRequests)
93     values (custId, reservationTime, numberOfGuests, specialRequests);
94
95     -- Insert dining preferences if provided
96     if favoriteTable is not null or dietaryRestrictions is not null then

```

## Restaurantserver.py

The RestaurantPortalHandler class initializes the RestaurantDatabase within the \_\_init\_\_ method and calls the BaseHTTPRequestHandler initializer. The do\_POST method is defined to handle POST requests. It includes parsing the form data for the /addReservation path. And adds a reservation to the database. It sends a simple HTML response indicating the reservation has been added.

The do\_GET method is defined to handle GET requests. It displays all reservations for the root path /. It is used as a placeholder for handling other paths such as /addCustomer and /findReservations.

In the server code I made some changes which included a broader error handling mechanism with self.send\_error(404, 'Path Not Found: %s' % self.path) for unknown paths, making the server more robust. Added forms for /addCustomer and /addSpecialRequests paths enhance the functionality, allowing users to add customers and update special requests directly from the web interface.

## Restaurantdatabase.py

First, we checked if the port number provided in the restaurantDatabase.py file is the same or not of our database server. Then we changed the password of the database with the password of our database server.

#### addCustomer method

The addCustomer method is designed to add a new customer to the customers table in the database. This method ensures that customer information is recorded and stored, allowing the restaurant to keep track of its customers' details. The method first checks if the connection to the database is active using `self.connection.is_connected()`. This ensures that any database operations are only performed when there is a valid connection. If the connection is active, a cursor object is created using `self.connection.cursor()`. The cursor is used to execute SQL queries on the database. Then we use the SQL query string for inserting a new customer, i.e. `insert into customers (customerName, contactInfo) values (%s, %s)`. This query uses placeholders (`%s`) for the values to be inserted. This approach helps prevent SQL injection attacks by using parameterized queries. Then the cursor executes the SQL query with the provided `customer_name` and `contact_info` values. The transaction is committed to the database using `self.connection.commit()`. This step ensures that the changes are saved and made permanent in the database. A success message is printed to the console to indicate that the customer has been added successfully. Finally the method returns the `customer_id` of the newly added customer using `self.cursor.lastrowid`. This value is the unique identifier assigned by the database to the new customer record.

#### findCustomer method

The findCustomer method is used to locate a customer in the customers table of the database based on the customer's name and contact information. If the customer is found, the method returns the customer's unique identifier (`customerId`). If the customer is not found, the method returns `None`. The cursor fetches a single result from the query using `self.cursor.fetchone()`. This method returns a tuple containing the `customerId` if a matching customer is found, or `None` if no match is found. The method returns the `customerId` from the result tuple if a customer is found and if no matching customer is found, the method returns `None`.

#### addReservation method

The addReservation method is used to insert a new reservation into the reservations table of the database. The method handles the entire process of checking for an existing customer and adding the customer if they do not exist, before inserting the reservation details. The method attempts to find the customer in the database using the findCustomer method. If the customer does not exist (`customer_id` is `None`), the method adds the customer using the addCustomer method and retrieves the newly generated `customer_id`. A new cursor is created for executing the SQL query. The SQL query string is defined to insert a new record into the reservations table. The execute method runs the query, inserting the provided reservation details along with the `customer_id`. The changes are committed to the database. A confirmation message is printed to indicate successful insertion.

#### getAllReservations method

The `getAllReservations` method is designed to retrieve all reservation records from the reservations table in the database. This method fetches the entire list of reservations, providing a comprehensive view of all the entries in the table. The SQL query string `'select * from reservations'` selects all columns from the reservations table. The `'execute'` method of the cursor runs the query against the database. The `'fetchall'` method retrieves all rows returned by the executed query. These rows are stored in the `'records'` variable, which is then returned by the method.

#### getCustomerPreferences method

The `'getCustomerPreferences'` method is used to retrieve the dining preferences associated with a specific customer from the `diningPreferences` table in the database. This method allows for the retrieval of information regarding a customer's preferred dining choices, such as favorite table and dietary restrictions. This method is similar to that of `getAllReservations` except we added a `where` clause in the query.

#### addDiningPreference method

The `'addDiningPreference'` method allows for the addition of dining preferences for a specific customer into the `'diningPreferences'` table in the database. This method enables the recording of a customer's favorite table and dietary restrictions, facilitating personalized dining experiences. The SQL query string `'INSERT INTO diningPreferences (customerId, favoriteTable, dietaryRestrictions) VALUES (%s, %s, %s)'` inserts a new row into the `diningPreferences` table, specifying the `customerId`, `favoriteTable`, and `dietaryRestrictions` columns with corresponding values.

#### updateSpecialRequest method

The `'updateSpecialRequest'` method enables the modification of special requests associated with a specific reservation in the reservations table of the database. This method facilitates the updating of special requests made by customers for their reservations, ensuring that the latest information is reflected in the database. `UPDATE` is used to specify the table to be updated, which is the reservations table. The `WHERE` condition identifies the specific reservation to be updated based on its `'reservationId'`. The SQL query string `'UPDATE reservations SET specialRequests = %s WHERE reservationId = %s'` updates the `'specialRequests'` column of the reservations table for the specified `'reservation_id'`. The `execute` method of the cursor executes the query, with the `'special_requests'` and `'reservation_id'` parameters passed as a tuple.

Link to github:

[github.com/mohiuddinhasan1/CIS-344](https://github.com/mohiuddinhasan1/CIS-344)