**Final SQL Code Database Review**

**Create the Database:**

A screen shot of a computer

Description automatically generated

**Explanation:**

* This creates the database schema named LittleLemonDB if it doesn’t already exist.
* The utf8mb3 character set is used for backward compatibility.
* The USE command sets the active database context for all subsequent operations.

**Create the Audit Log Table:**

**A screen shot of a computer program

Description automatically generated**

**Purpose**

* Audit log table is used for tracking all critical changes to other tables in the database, it helps maintain accountability, transparency, useful for debugging or auditing purposes.

**Column Descriptions:**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| AuditID | INT (Auto Increment) | Primary key. Uniquely identifies each audit entry. |
| TableName | VARCHAR(50) | The name of the table where the change occurred. |
| ActionType | ENUM('INSERT', 'UPDATE', 'DELETE') | Describes the type of action performed. |
| RecordID | INT | The primary key of the affected record in the referenced table. |
| OldData | MEDIUMTEXT | Stores the previous data before the action (if applicable). |
| NewData | MEDIUMTEXT | Stores the new data after the action (if applicable). |
| ActionTimestamp | TIMESTAMP | Automatically logs the time of the action. |
| PerformedBy | VARCHAR(100) | Captures the user/system who performed the action. |

**Indexes:**

* idx\_action\_timestamp: Optimizes queries by timestamp (e.g., fetch latest changes).
* fk\_audit\_log\_orders: Helps filter audit logs by RecordID. Despite the name, this is **not a foreign key**, just an index.

**Storage Engine & Charset:**

* **Engine**: InnoDB — supports transactions and row-level locking.
* **Charset**: utf8mb4 — allows full Unicode (including emojis), ideal for logs that may contain diverse data.
* **Collation**: utf8mb4\_unicode\_ci — case-insensitive comparison with proper Unicode handling.

**Create Customer Table:**

A screen shot of a computer

Description automatically generated

**Purpose**

* The Customers table stores basic contact information about customers, uniquely identifying each one via a CustomerID. It serves as a key reference for other operations like bookings and orders.

**Column Descriptions:**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| CustomerID | INT (Auto Increment) | Unique identifier for each customer (Primary Key). |
| FirstName | VARCHAR(50) | The customer’s first name. |
| LastName | VARCHAR(50) | The customer’s last name. |
| PhoneNumber | VARCHAR(15) | Contact phone number (must be unique). |
| Email | VARCHAR(100) | Email address (must be unique). |

**Constraints & indexes**

* Primary Key: CustomerID
* Unique Indexes:
  + PhoneNumber\_UNIQUE: Prevents duplicate phone numbers
  + Email\_UNIQUE: Ensures email addresess are unique
* NOT NULL – ensures a column must always have a value – it cannot be left empty

**Why Use NOT NULL?**

* **Data integrity**: Ensures important fields are always filled (like names, emails, or IDs).
* **Prevents errors**: You avoid having NULL (unknown) values where meaningful data is expected.
* **Supports logic**: Many business rules (e.g., sending emails) depend on having complete data.

**Create Tables Table:**

**A screen shot of a computer

Description automatically generated**

**Purpose**

* This table stores details about restaurant tables, including their unique number, seating capacity, and available status. It is used to manage table bookings and assignments

**Column Descriptions**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| TableID | INT (Auto Increment) | Primary key, uniquely identifies each table in the restaurant. |
| TableNumber | INT | Public-facing table number (e.g., Table 1, Table 2). Must be unique. |
| Capacity | INT | Number of people the table can seat. |
| IsAvailable | TINYINT(1) | Indicates if the table is currently available (1 = yes, 0 = no). |
| LastUpdated | TIMESTAMP | Tracks when the table’s availability or other info was last updated. |

**Constraints & Indexes:**

* **Primary Key**: TableID
* **Unique Index**: TableNumber\_UNIQUE — ensures that each table number is used only once.
* **Default Values**:
* IsAvailable: Defaults to 1 (available).
* LastUpdated: Automatically updates whenever the row is modified.

**Create Bookings Table:**

A screen shot of a computer program

Description automatically generated

**Purpose:**

* The Bookings table stores records of **table reservations** made by customers. It links a customer to a table on a specific date, and tracks whether the booking is active or cancelled.

**Column Descriptions**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| BookingID | INT (Auto Increment) | Unique identifier for each booking (Primary Key). |
| CustomerID | INT | Foreign key reference to the customer who made the booking. |
| BookingDate | DATE | The date for which the booking is scheduled. |
| TableID | INT | Foreign key reference to the table being booked. |
| Status | ENUM('New Booking', 'Cancelled') | Indicates if the booking is active or cancelled. |

**Constraints & Indexes**

* **Primary Key**: BookingID
* **Foreign Keys**:
  + CustomerID → Customer\_Details(CustomerID)
  + TableID → Tables(TableID)

**Cascade Rules**:

* **ON DELETE CASCADE**: If a customer or table is deleted, their bookings are automatically deleted.
* **ON UPDATE CASCADE**: If a customer/table ID changes, the booking updates too.

**Indexes**:

* CustomerID\_idx — for fast lookup of bookings by customer.
* idx\_BookingDate — optimizes queries by date.
* fk\_bookings\_tableid — assists in finding all bookings for a specific table.

**AUTO\_INCREMENT = 200 (explained)**

**Why Would You Set It to 200?**

There are a few reasons you might do this:

**Skip Reserved IDs**

* You may want to **reserve** BookingIDs 1–199 for legacy data, testing, or special cases.

**Mask Real Volume**

* Setting it to a higher number gives the **appearance of a larger customer base** or business volume — e.g., if a new system is starting, it doesn’t look like the first-ever booking.

**Avoid Conflicts**

* If records were **imported from another system** where IDs went up to 199, you set it to 200 to avoid duplicate keys.

**Create Menu Table:**

A screen shot of a computer

Description automatically generated

**Purpose:**

* The Menu table stores the list of available food and drink items that customers can order. Each has a name, category and price.

**Column Descriptions:**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| MenuID | INT (Auto Increment) | Primary key that uniquely identifies each menu item. |
| ItemName | VARCHAR(100) | Name of the menu item. |
| Category | ENUM('Starters', 'Courses', 'Drinks', 'Desserts') | Classifies the item type. |
| Price | DECIMAL(10,2) | Cost of the menu item, with two decimal precision. |

**Constraints & Indexes**

* **Primary Key**: MenuID
* No foreign keys or additional indexes are defined — the table is kept simple and flat.

**Create Staff Information Table:**

A screen shot of a computer program

Description automatically generated

**Purpose:**

* The Staff Information table is used to **store employee data** for all staff working at Little Lemon. It includes names, roles (e.g., Manager, Waiter, Chef), and salaries. This table supports HR operations and is referenced when assigning staff to customer orders.

**Column Descriptions**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| StaffID | INT (Auto Increment) | Unique identifier for each staff member. |
| FirstName | VARCHAR(50) | Staff member’s first name. |
| LastName | VARCHAR(50) | Staff member’s last name. |
| Role | VARCHAR(45) | The job position or title (e.g., Chef, Server, Manager). |
| Salary | DECIMAL(10,2) | Salary or wage (supports up to 10 digits, 2 decimal places). |

**Constraints & Indexes**

* **Primary Key**: StaffID — ensures each staff record is uniquely identified.
* There are **no unique or foreign keys** in this table.
* Role is stored as free text (VARCHAR), which provides flexibility but no validation.

**Create Orders Table:**

A screenshot of a computer

Description automatically generated

**Purpose:**

* The orders table stores all customer orders placed at little lemon. It links each order to a customer, the staff member who handled it, and optionally to a booking.

**Column Descriptions:**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| OrderID | INT (Auto Increment) | Unique ID for the order (Primary Key). |
| CustomerID | INT | Foreign key linking to the customer who placed the order. |
| StaffID | INT | Foreign key indicating who handled the order. |
| OrderDate | DATE | Date when the order was placed. |
| TotalCost | DECIMAL(10,2) | Total cost of the entire order. |
| BookingID | INT (nullable) | Optional foreign key linking the order to a booking. |

**Constraints & Relationships:**

* **Primary Key**: OrderID
* **Foreign Keys**:
  + CustomerID → Customer\_Details(CustomerID) — **CASCADE on delete/update**
  + StaffID → Staff\_Information(StaffID) — **CASCADE on delete/update**
  + BookingID → Bookings(BookingID) — **SET NULL on delete**, **CASCADE on update**

If a booking is deleted, the BookingID in the order is set to NULL (order is preserved).

**Indexes:**

* CustomerID\_idx: Optimizes queries by customer.
* StaffID\_idx: Helps filter orders handled by a particular staff member.
* fk\_orders\_bookings: Index to speed up joins on BookingID.

**Create Order Delivery Status Table:**

A screen shot of a computer

Description automatically generated

**Purpose:**

* The order delivery status table tracks the current delivery status of each customer order. It allows the restaurant to manage and update the state of orders as they move through the fulfilment process

**Column Descriptions:**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| DeliveryID | INT (Auto Increment) | Primary key that uniquely identifies the status entry. |
| OrderID | INT | Foreign key referencing the Orders table. |
| Date | DATE (Nullable) | The date associated with the delivery status update. |
| Status | ENUM | Current state of the order: New Order, Preparing, Out for Delivery, Delivered, or Cancelled. |

**Constraints & Relationships:**

* **Primary Key**: DeliveryID
* **Foreign Key**:
  + OrderID → Orders(OrderID) — **CASCADE on delete/update**

If the order is deleted, its delivery status is deleted as well.

**Indexes**

* OrderID\_idx: Helps efficiently join or filter records by order.
* idx\_Status: Speeds up queries by delivery status (e.g., filter all “Out for Delivery” orders).

**Create Order Items Table:**

A screen shot of a computer

Description automatically generated

**Purpose:**

* The Order Items table captures individual items within an order — including the menu item, quantity, price per item, and the total cost for each line. This allows detailed tracking of what was ordered and for how much.

**Column Descriptions**

| **Column Name** | **Data Type** | **Description** |
| --- | --- | --- |
| OrderItemID | INT (Auto Increment) | Unique ID for each line item (Primary Key). |
| OrderID | INT | Foreign key linking to the Orders table. |
| MenuID | INT | Foreign key linking to the Menu table. |
| Quantity | INT | Number of units of the item ordered. |
| ItemPrice | DECIMAL(10,2) | Unit price at the time of the order. |
| TotalItemCost | DECIMAL(10,2) | **Generated column**: Quantity × ItemPrice, automatically calculated. |

**Constraints & Relationships:**

* **Primary Key**: OrderItemID
* **Foreign Keys**:
  + OrderID → Orders(OrderID) — **CASCADE on delete**
  + MenuID → Menu(MenuID) — **CASCADE on delete/update**
* **Generated Column**:
  + TotalItemCost is **stored** (not virtual), and recalculated based on quantity and price.

**Indexes:**

* OrderID\_idx: Speeds up retrieval of all items in a specific order.
* MenuID\_idx: Helps find all orders that include a specific menu item.

**Stored Procedures Breakdown**

* AddMultipleItemsToOrder
* AddValidBooking
* CancelAllBookingsForDate
* CancelBooking
* ChangeBookingTable
* CheckBooking
* CreateOrder
* ErrorHandler
* FindAvailableTables
* GetBookingsForDate
* GetMaxQuantity
* UpdateBooking
* UpdateOrderStatus

**AddMultipleItemsToOrder – Procedure 1**

(refer to SQL final script for the full code)

**Purpose:**

* This procedure enables adding multiple menu items to an existing order in a single call, without requiring multiple insert statements from the application layer.
* It accepts comma-separated values for both MenuIDs and Quantities, processes each pair, and inserts them into the order items table

**Step-by-step Breakdown**

Procedure Header

*DELIMITER $$*

*CREATE PROCEDURE `AddMultipleItemsToOrder`(*

*IN p\_OrderID INT,*

*IN p\_MenuIDs TEXT,*

*IN p\_Quantities TEXT*

*)*

**Explanation:**

* We define a procedure named AddMultipleItemsToOrder.
* p\_OrderID – the target order to add items to.
* p\_MenuIDs and p\_Quantities – comma-separated lists like '2,4,5' and '1,3,2' representing item IDs and their quantities.

Start of Procedure Block

*BEGIN*

* Opens the main procedure logic block. All declarations and operations happen inside this block

Declare Variables

*DECLARE done INT DEFAULT 0;*

*DECLARE v\_MenuID INT;*

*DECLARE v\_Quantity INT;*

*DECLARE v\_ItemPrice DECIMAL(10,2);*

**Explanation:**

* done: A control flag to stop the cursor loop.
* v\_MenuID: Holds the current menu item ID from the parsed string.
* v\_Quantity: Holds the quantity for that item.
* v\_ItemPrice: Temporarily stores the unit price for the current item.

Declare Cursor (for parsing strings)

*DECLARE menu\_cursor CURSOR FOR*

*SELECT menu\_id, quantity FROM (*

*SELECT*

*SUBSTRING\_INDEX(SUBSTRING\_INDEX(p\_MenuIDs, ',', numbers.n), ',', -1) AS menu\_id,*

*SUBSTRING\_INDEX(SUBSTRING\_INDEX(p\_Quantities, ',', numbers.n), ',', -1) AS quantity*

*FROM*

*(SELECT 1 n UNION ALL SELECT 2 UNION ALL SELECT 3 UNION ALL SELECT 4 UNION ALL SELECT 5*

*UNION ALL SELECT 6 UNION ALL SELECT 7 UNION ALL SELECT 8 UNION ALL SELECT 9 UNION ALL SELECT 10) numbers*

*WHERE CHAR\_LENGTH(p\_MenuIDs) - CHAR\_LENGTH(REPLACE(p\_MenuIDs, ',', '')) >= numbers.n - 1*

*) AS temp\_data;*

**Explanation:**

* This builds a **cursor** that simulates iterating over arrays in SQL.
* It breaks apart the comma-separated p\_MenuIDs and p\_Quantities by using SUBSTRING\_INDEX.
* The derived numbers table simulates looping 10 times (enough for up to 10 items).
* Each loop extracts the nth value from both strings.
* The WHERE clause makes sure we don’t go beyond the length of the strings.

Define Cursor Exit Handler

*DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;*

**Explanation:**

* Tells MySQL: *“When no more rows are found in the cursor, set done to 1 instead of throwing an error.”*
* This allows us to exit the loop cleanly.

Open Cursor

*OPEN menu\_cursor;*

* Opens the cursor so we can start fetching values from it

Cursor Loops Begins

*read\_loop: LOOP*

*FETCH menu\_cursor INTO v\_MenuID, v\_Quantity;*

*IF done THEN*

*LEAVE read\_loop;*

*END IF;*

**Explanation:**

* Named loop: read\_loop.
* FETCH reads the next row of parsed menu ID and quantity into variables.
* If done is set (no more rows), it breaks the loop using LEAVE.

Get Item Price

*SELECT Price INTO v\_ItemPrice FROM Menu WHERE MenuID = v\_MenuID;*

* Looks up the current price for the menu item from the menu table

Insert Into Order Items

*INSERT INTO Order\_Items (OrderID, MenuID, Quantity, ItemPrice)*

*VALUES (p\_OrderID, v\_MenuID, v\_Quantity, v\_ItemPrice);*

* Add the current item into the Order\_Items table with wuantity and price
* TotalItemCost is automatically calculated because it’s a generated column

Close the Loop

*END LOOP;*

* Marks the end of the cursor-based loop

Close the Cursor

*CLOSE menu\_cursor;*

* Releases resources used by the cursor

Update Delivery Status

*UPDATE Order\_Delivery\_Status*

*SET Status = 'Preparing'*

*WHERE OrderID = p\_OrderID AND Status = 'New Order';*

* Once items are added, we update the delivery status from ‘New Order’ to ‘Preparing’. This helps downstream processes know the kitchen is working on it.

End of Procedure

*END*

* Marks the end of the procedure logic

DELIMITER $$/;

DELIMITER $$

...

DELIMITER ;

* Used to change the statement delimiter while writing the procedure.
* This prevents ; inside the procedure from prematurely ending it.

Re-Cap of AddMultipleItemsToOrder Procedure (with example data)

Goal:

* You want to add multiple menu items to one existing order using this procedure, instead of running multiple INSERT statements.