mySQL BACKEND  
FOR ORDERING SYSTEM

Nathan Weber

Database Overview

The databasing solution that has been chosen for the backend of this project is an AWS RDS instance running mySQL 8.0.17. It is a micro class instance with 20 GB of storage and 1GB of RAM, for demonstration and bug-testing purposes. It is configured to allow mySQL traffic from anywhere, as long as there is a valid user connection established. The only user connection allowed besides the master account is a user named “program”, which has Select, Insert, Update, and Delete privileges.  
 This project is meant to emulate a program that could be connected to in multiple locations at the same time. It would also theoretically contain a large amount of orders, so the ability to have a short response time would be necessary as well. As such, a file I/O system as was designed for this project earlier in the timeline would not be suitable for this use case. The database gives us the ability to handle large back catalogues of orders for later analysis, while preserving performance. It also allows multiple users to contribute to a shared volume of information, rather than relying on updating local files periodically, which would cause a plethora of collision issues. It is for these reasons that the move to a remote databasing solution was made.

Several options for remote databasing were considered. mySQL proved to be similar to the SQL language that I was already partially familiar with, includes a powerful Python connector library by Oracle, and is very simple to set up on a variety of platforms. Digital Ocean and AWS were attractive opportunities, but I am in contact with a person that is familiar with working with AWS for databasing and authentication, so it was selected. Unfortunately, at the time that the RDS instance was set up, I was unaware of the student credit option that AWS offers through the GitHub student pack.

Database Schema

A screenshot of a cell phone

Description automatically generated

*E-R diagram provided by the reverse-engineering output of mySQL Workbench.*

This database is fairly simple in terms of construction. There are three tables contained within the order system database: orderlist, itemslist, and the inventory table. Orderlist and itemslist comprise the majority of the workings of the backend, containing all of the orders that are submitted to the database and the items within those orders. The Inventory table is the storage for the inventory tracking portion of the project, containing items ordered by an auto-incrementing ID.

Orderlist has four columns, orderid, recipient, active, and working. OrderID is the primary key for the table, and it increments by 1 every time that an order is inserted into the table. This key is used to address a specific row from the outside, as all other columns in the table are mutable from the m\_OrderIOLink connection. The “recipient” column describes the name of the person whom the order belongs to, or alternatively, the number of the table that the order belongs to. “Active” is a Boolean (automatically translated to a TinyInt type in mySQL, the two are synonyms) which designates if a specific order has been place but has yet to be fulfilled. While this boolean is set to true (1), the order will show up in the list of active orders to be displayed in the overview. When the boolean is set to 0, i.e., when the order has been completed and the recipient has checked out, the order is said to be archived for later analysis. The “working” boolean works as a partner to the “active” column, indicating if the kitchen has already started work on the order. The function of this column is to ensure that orders can no longer be edited after the kitchen starts the order.

Itemslist has five columns: OwnerOrder, name, quantity, price, and sreq. OwnerOrder works as the first part of the three-element compound primary key for the itemslist table. It is also a foreign key, referencing an extant orderid element in the orderlist table. This foreign key’s behavior is set to CASCADE on deletion of the referenced ordered, which prevents orphaned items from lingering in the table. Name describes the name of the item specified; quantity is self-explanatory as well. Price contains a double describing the price of an individual instance of the object. The total price of an order is calculated by the product of Price and Quantity. Sreq is a string that contains the “special requests” for the specific items in an order, specific ordering requests that may not be handled simply by name, such as a request for a hamburger to not have cheese or lettuce or how much to cook meats.

The compound primary key in the itemslist table is necessary so that the items can be addressed smoothly despite their columns being mutable, as well as allowing multiple items in a single order to be addressed despite having the same name, but different special requests.

The inventory table is independent of the above tables. The inventory functionality is implemented to track items like bags of chips, or other pre-packaged items that can be added to an order but may fall in and out of stock at any given time. Rows in the inventory table are given an auto-incrementing ID, a name, and a quantity.

Core Objects

Python classes for use in and outside of the backend are defined in the Order.py file. The classes within are the “Item” and “Order” classes. These classes function in a similar relationship that the tables exist in above; one Order can contain one or more Items. Orders have a field for their orderID, their recipient, their price, a list of all of the items they own, and booleans for working and active as above. Items have a field for their name, price, quantity, and special requests as above.

Items can be constructed as follows:

item = Item(*name, price, quantity*)

or

item = Item(*name, price, quantity, sreq*)

where name is a string, price is a real number, quantity is an int, and sreq is a string. If no sreq is specified in the construction, the special request field will be recorded as “NSR”, for “No Special Requests”. These objects are then added to an order or are used in conjunction with an index to update a specific item in an order. Items have no functions attached to them.

Orders can be constructed as follows:

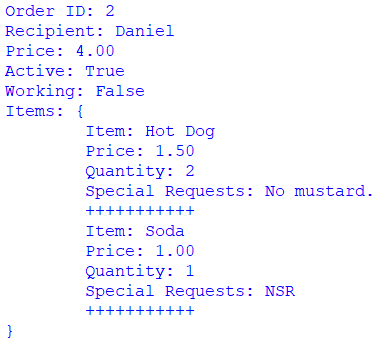
order = Order(*recipient*)

or

order = Order(*recipient, orderid*)

where recipient is a string and ordered is an int. Please note that the constructor that specifies an orderid should only be used in the m\_OrderIOLink.py file, as the orderid that is passed into this must correlate to the orderid for this object in the database itself. Any Order object that is instantiated without an orderid is given the default orderid of -1. An orderid will be assigned to orders once they are added to the database.

Order objects have a number of functions associated with them. There is a \_\_str\_\_ function that allows an order to be printed as a string, as seen below.



*An example of an order with two items being printed.*

addItem(i):  
Adds the Item object *i* to the Items[] list of the order object. It also adds the product of the item’s unit price and quantity to the price attribute of the order.

removeItem(i):  
Removes an Item object *i* from the Items[] list of the order object. It also subtracts the product of the item’s unit price and quantity from the price attribute of the order.

getItemIndex(i):  
Returns the index of an Item object *i* in the order object’s Items[] list. For use with getItem(ind) and updateItem(i, ind).

getItem(ind):  
Returns an Item object located at the index *ind* in the Order’s Items[] list.

updateItem(i, ind):  
Replaces the Item object at index *ind* in the Order’s Items[] list with the Item object *i*.

setWorking(b):  
Sets the working status of the Order object to the value indicated by boolean *b*.

setActive(b):  
Sets the activity status of the Order object to the value indicated by boolean *b*.

orderOut():  
Prints the object to the console in a similar manner to the \_\_str\_\_ function.

There is an additional object used in the backend, described in the InvItem.py file provided by Kayla Welton. This object describes the object that is used in the inventory functionality of this project, as well as the object used to interface with the Inventory table in the database. It contains the fields name, quantity, and ID. There are getter functions for each of these fields, as well as a \_\_str\_\_ definition.

The constructors are:

inv = InvItem(*name, quantity*)

and

inv = InvItem(*name, quantity, id*)

As with the order function, the constructor that includes an ID for the item should only be used in the connector file, as the id is supplied by response from the backend exclusively.

Database link: OrderIOLink object

Interfacing with the database is controlled by the OrderIOLink object. This object utilizes the Connector/Python library provided by Oracle for interface between python and a mySQL database. The primary design goal for this object was to provide the other developers with a set of easy to understand and read functions that execute queries that manipulate the SQL database that has been constructed for this project. This object facilitates the translation of Order and InvItem objects into a form that the database can understand and store. It also allows the user to request certain information from the database and package that information into Order and InvItem Objects for use in the main program. In total, the object is able to facilitate insertion, deletion, and updates of orders, specific order retrieval, retrieval of all active orders, specific inventory item retrieval, insertion, deletion, and update of inventory items, and the retrieval of all inventory items.

There is only one constructor for this object. It opens a connection with the database, and once the connection is established, creates a cursor for this connection. It is important to note that once the this object has been constructed, it must have the close() function performed before closure of the program or else the connection will linger. The constructor is called as follows:

link = OrderIOLink()

close():  
Closes the cursor, then the connection. Necessary to prevent the database from becoming overwhelmed with lingering connections.

getOrder(oid):  
Given an order ID *oid*, retrieves the order from the orderlist table, creates an empty Order object with it, and populates it with the items associated with that order in the itemslist table, then returns the Order object.

getAllActiveOrders():  
Retrieves all of the orders in the orderlist table that are marked as “active”, then constructs Order objects for each of them that have been populated with the items associated with each order. It then returns all of these Order objects in a list.

checkWork(o):  
Checks if the order *o* is marked as “working”. This is a helper function to determine if the user is allowed to update an order. If the order is working, returns true. If it is not, it returns false.

addOrder(o):  
Adds the Order object *o* and all of its items to the database.

removeOrder(oid):  
Removes the order specified by order id *oid* from the database. The database has been configured to automatically remove all associated items from the database as well when a deletion occurs like this.

updateOrder(o):  
Modifies the order in the system associated with the object *o* to match the object *o* passed in. Please note that the way that the database determines which object this refers to is by the object’s order ID. To make sure that this performs correctly, the usage of the function should be as follows:

o1 = getOrder(`oid`)

*##modify o1 here*

link.updateOrder(o1)

This preserves a correct oid.  
Additionally, please note that this function will not update the order if the order is marked as “working” using the checkWork(o) function. If the order has been erroneously marked as “working”, please use the updateOrderWorkingState(o) function to update the order’s working status beforehand.

updateOrderWorkingState(o):  
Updates the order *o* in the database to reflect its modified working state. Should be used to modify the working state of the order to lock it from being updated further, or to rectify an erroneous working state update.

getInvItem(name):  
Returns an InvItem object from the Inventory table in the database that is associated with *name*.

addInvItem(inv):  
Adds a given InvItem object *inv* to the Inventory table in the database.

removeInvItem(inv):  
Removes a given InvItem matching given object *inv* in the Inventory table in the database.

updateInvItem(inv):  
Updates an item in the Inventory table matching given InvItem object *inv* to match *inv*.

getAllInvItems():  
Returns a list populated with InvItem objects created from all of the rows in the Inventory table.