Chase Toyofuku-Souza
CPSC 408
5/19/2021

Restaurant POS System

Menya Le Nood is a small ramen restaurant located in Honolulu, Hawaii that is in need of a new, low-cost, highly reliable POS system. Our current POS is unable to run fully offline, and it does incur monthly fees. Additionally, it has a large amount of features that employees are unaware of, so a lot of time is needed to train employees. I developed an application to manage reservations and orders in a simple and efficient manner.

There are many POS systems available that offer both software and hardware, including options for employee management and scheduling. Menya's current provider is *Lavu*, an iPad POS that comes with a terminal for swiping credit cards and opening a cash drawer. Because Lavu utilizes an iPad as the interface, we are able to use a touch screen interface, which is convenient for faster order processing. Additionally, it is able to send orders from the front of the house to the back of the house over WiFi. The most outstanding part of Lavu is it has an offline mode, that prevents data from being lost when the WiFi is down.

Although Lavu is a very robust solution for restaurants looking for an all-in-one POS and management system, there are a few downsides. The most noticeable downside is the system is reliant on connection to the internet, and Menya often has internet problems. Although Lavu describes the offline mode as being able to sync orders across several terminals without an internet connection, this functionality is not particularly useful for a restaurant that only uses one terminal. Additionally, due to Lavu being hosted on an iPad, they rely on WiFi to connect to the printers (for receipts and sending orders to the back of house), which causes additional strain on the staff when the internet goes out during busy hours.

Another popular POS system is *Square*, which offers various interfaces depending on the type of business the customer has, including retail, food and beverage, professional services, and even enterprise businesses. Similar to Lavu, Square offers a iPad compatible display system, dubbed as a *Kitchen Display System*, and offers a lot more dynamic features than the Lavu. Square integrates all orders into one system, whether it be from in-person to Postmates, all tickets are easily accessible from the Kitchen Display System. Square also offers more customizations to ticket layouts, and offers advanced features like timers and alarms for orders.

To simplify the features of our current POS, while minimizing costs and keeping the same functionality, I developed a restaurant-optimized POS system built in Python using Tkinter as the framework for the GUI. Although Python is not compatible with our current system (iPad), Python and Tkinter have the advantage of being cross-platform, so we could use Linux (free) or any existing computers to run this software.

There are several GUI frameworks available for Python, however I chose Tkinter because it is a library that comes ported with the standard installation of Python. Additionally, it does not have as complicated of an API to interact with when compared to other frameworks such as PyQt, which simplifies the design process. A disadvantage of Tkinter is that it does not have a modern look, but this is perfectly acceptable for creating a functional yet simplified GUI. Another disadvantage of Tkinter is that it does not have advanced widgets that PyQt5 may have such as treeview, a widget used to represent a table. To remedy the lack of modern features and advanced widgets, Tkinter supports another module, *ttk*, which is a themed version of Tkinter.

Tkinter has a very simple design structure, where there typically exists one “root” window, and then a hierarchy of widgets that are placed on top of that window. These widgets include familiar objects such as buttons, labels, scrollbars, and frames. Following this structure, I have two frames, the *MainFrame* and the *OrderFrame*, as well as the root window (*App*). Each of these are subclasses from the original Tkinter classes, where I configure the initial geometries and layouts, then define functions necessary for each frame's behavior.

This application's root window is purely responsible for housing the two frames, having functions that create the necessary frames as well as switch to/show the proper frame. The root window has a dictionary as a member variable, the key being the name of the frame's class, and the value being the frame object itself. In order to properly order the frames, they are placed in a grid on the same row and column, so one frame will always be covering the other. I can then call "tkraise()" to raise the specified frame above the other, as well as destroying it/reinitialize it to refresh the values if anything changed.

The main frame simply consists of eight buttons, one for each table, and a button for generating reports. In order to automate and simplify the design, I created a subclass of the tk.Button and tk.Label, which allows me to specify size, position, text, and font, all in one line.

The order frame is similar, having elements such as buttons and labels, but the difference here is that there are widgets in this frame that are only visible at certain times. For example, selecting a table that does not have an open tab will take you to a screen to create a reservation, and pressing "submit" on that screen will finally take you to the order page. However, this reservation creation and order submission are actually in the same frame, I just destroy the unneeded elements when it is no longer required to show them.

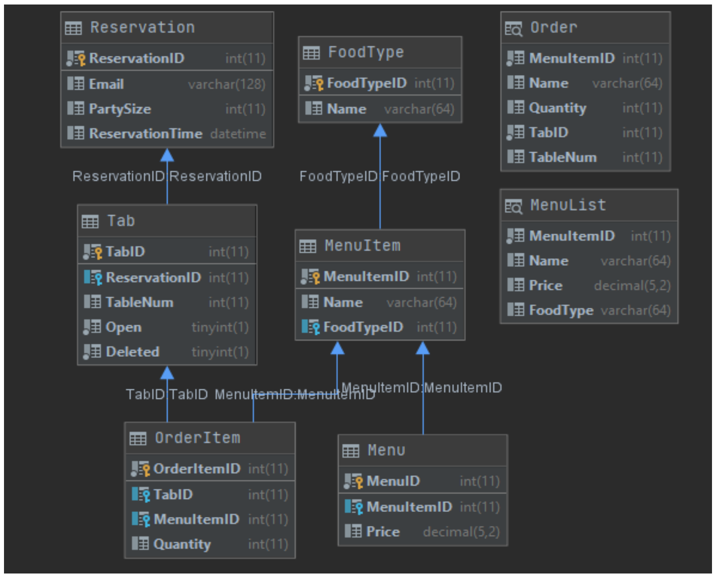
Tkinter is surprisingly flexible when it comes to placing widgets in your window. There are three different methods to accomplish this, pack, place, and grid. I previously mentioned that I used grid to place the two frames on top of each other, but that was the extent of it. The different methods to place widgets do not interact well with each other, and after many failed attempts to place widgets properly with grid, I eventually switched to “place”, which is the more tedious approach of the bunch, where you have to individually specify every single dimension of the item, rather than just a simple grid layout or an automatic pack layout. This difficulty came from the frames being in the same window, so trying to place them in a grid or via pack would also move the other frames widgets.

The back-end for this POS system is Cloud SQL, with the database being MySQL. One of the benefits of using Cloud SQL is that it is a reliable cloud database service with fast processing, and it has a relatively low cost for storage and processing. As I previously mentioned, the internet connection at the restaurant is unstable, so this application must be able to work flawlessly both online and offline. As such, a more optimized solution in the future may be to host a local MySQL server instead, or have a local backup as well as a remote connection.

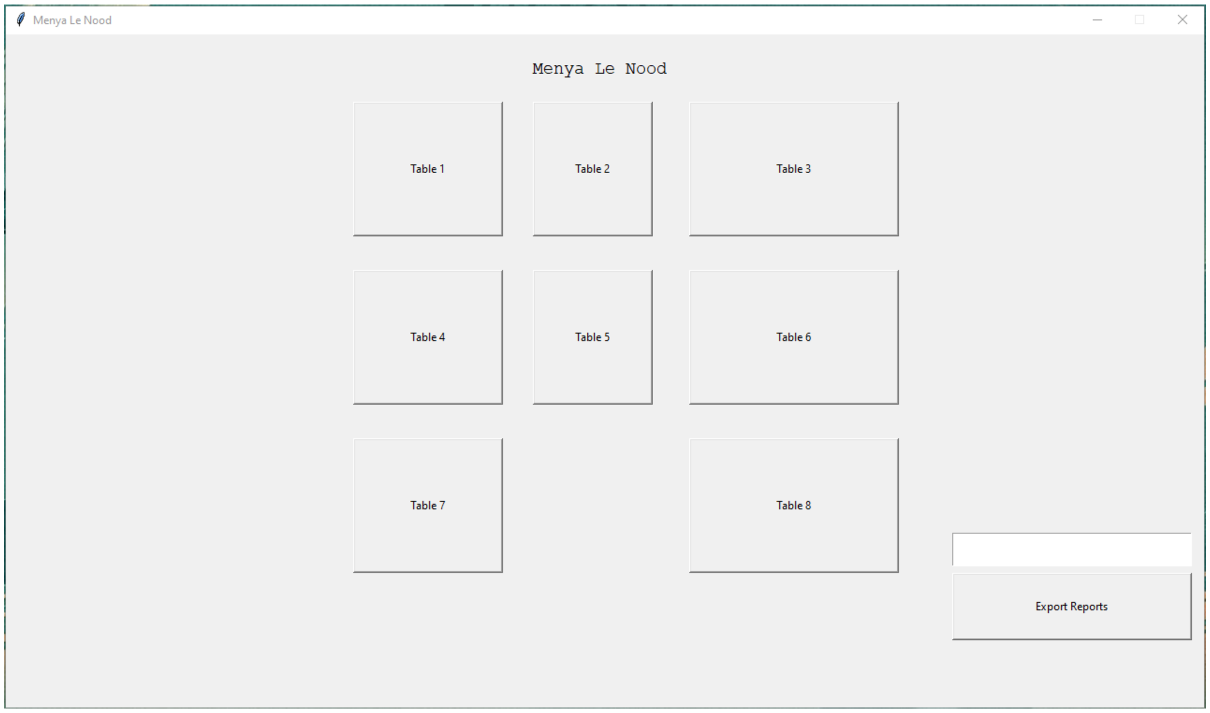
The schema for my database consists of six tables and two views, all table’s information being spread out to ensure 3NF. In terms of customers, it is required to create a reservation and use that “ReservationID” along with a “TableNum” to then create a Tab. This tab can be opened or closed, to represent if a customer has checked out or not. Tabs can also be deleted, in the event that a customer leaves or the server inputs an order incorrectly.

As for the menu of the restaurant, it is spread across three tables, *FoodType, MenuItem,* and *Menu*. The FoodType table allows me to define different categories of the menu items, which then allows reports to be made on food type, as well as easily sorting the menu in-app. To connect the food type to an actual item, there is the MenuItem table, which simply takes the FoodTypeID and adds an item name. Lastly, there is the Menu table, which combines the MenuItem with a price value. In the end, we have a menu which is sorted by type of food, and the prices for each item.

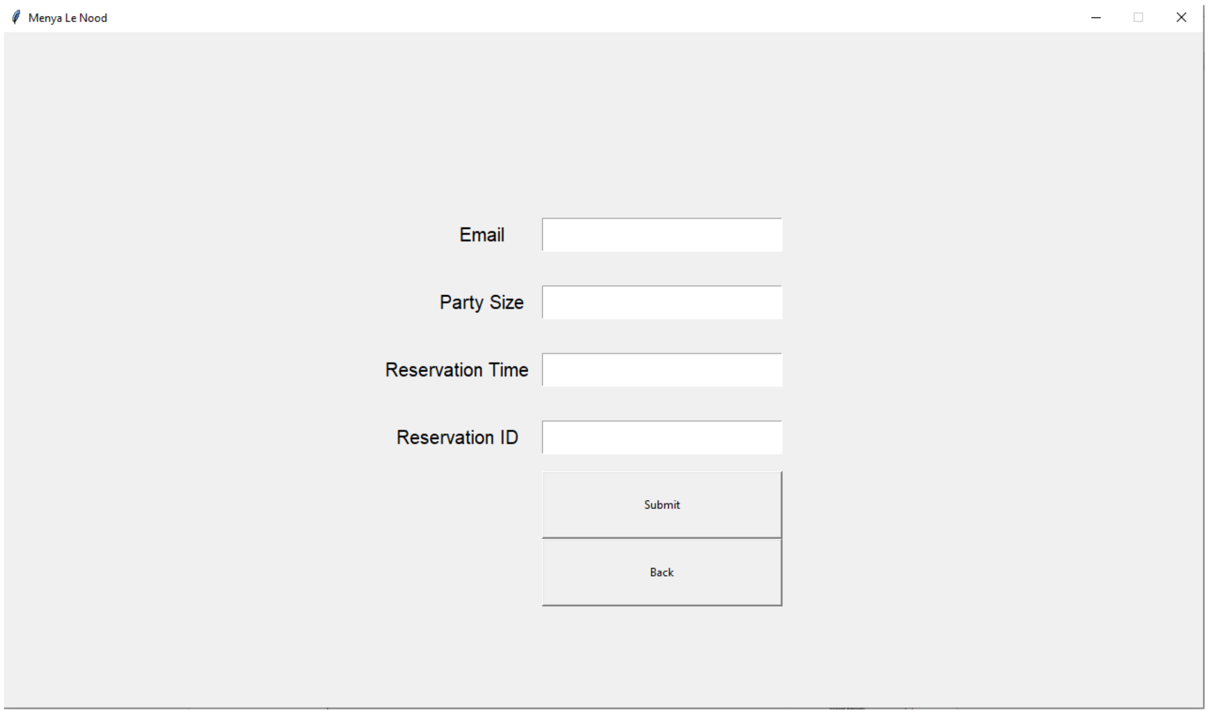
Finally, to connect the menu to the customer, we have the *OrderItem* table. This table takes in the “TabID” and “MenuItemID” and adds in a quantity that represents how much of that item they ordered. The views are there to combine the information spread across the separate tables and present it in a readable format. For example, the *MenuList* view will combine the type of the food, name of the food, and price of the food into one concise table.



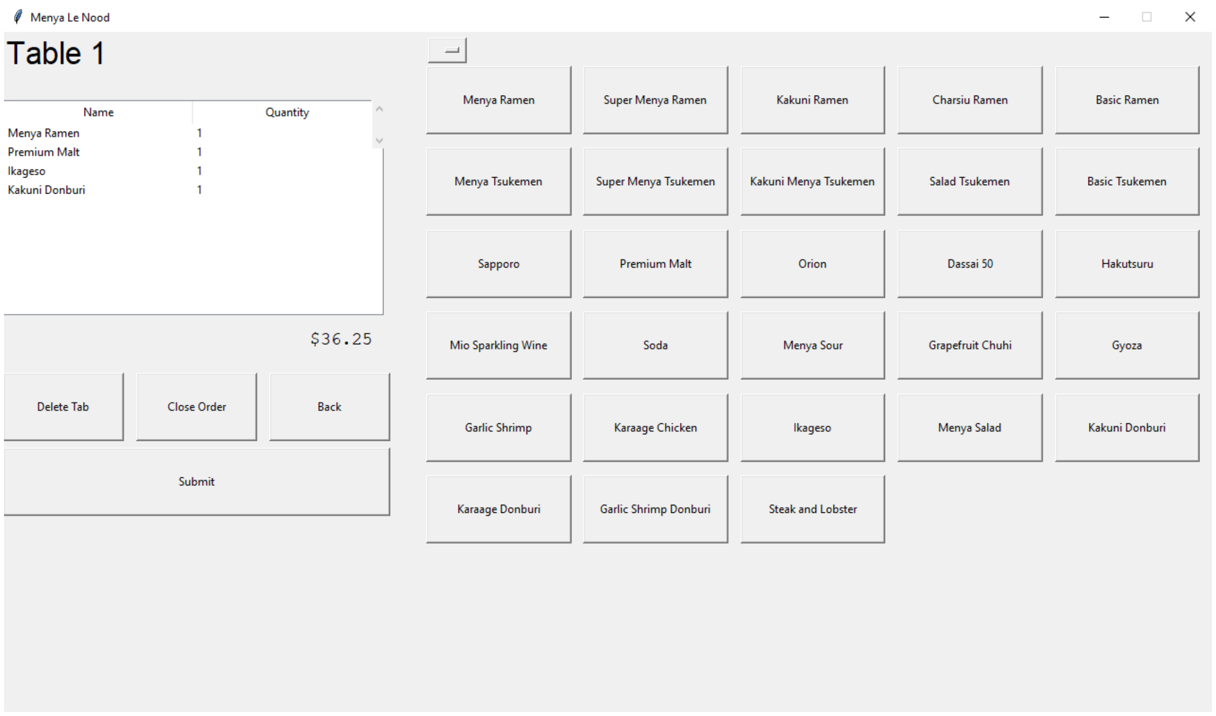
Schema diagram built with DataGrip



Home screen, choose tables and export reports



Create a reservation or choose an existing reservation



Name	Quantity
Menya Ramen	1
Premium Malt	1
Ikageso	1
Kakuni Donburi	1

$36.25

Menu for ordering items