# Point of Sale System for General Restaurant

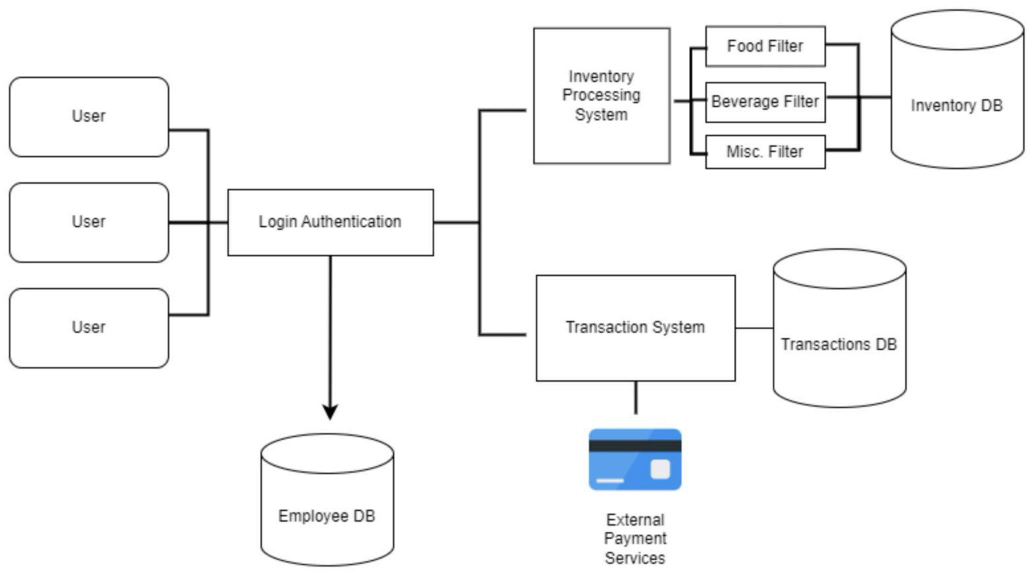
Team Members: Husain Patanwala, Andrew-Jacob Santos, Cody Tran

## System Description

This system is being developed and designed to be able to support general restaurant management functions. It will be able to take input from users, and then the system will interact with the backend updating and fetching information from databases to support daily restaurant operations. Ensuring smoother workflow between front-of-the-house and back-of-the-house employees. The users will access this system through tablet stations throughout the restaurant or through portable tablets.

## Software Architecture Overview

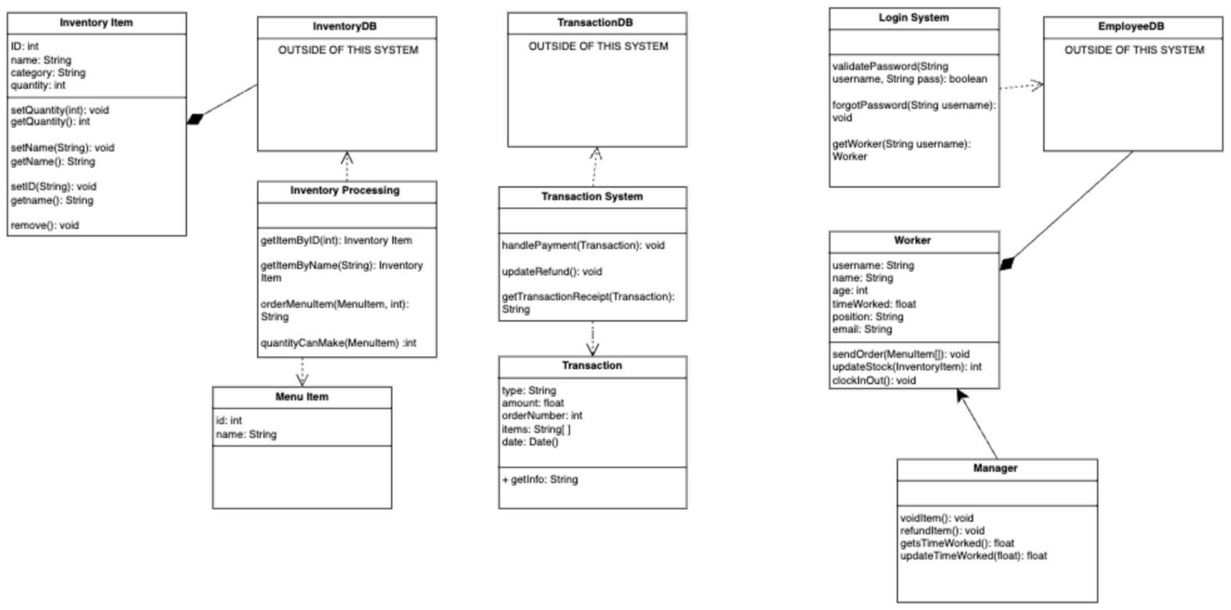
### Architectural Diagram of Major Components:



The leftmost component in the diagram represents the users (in this case restaurant staff) who will be provided input to the system through various event triggers. Initially, users will have to enter the PoS through the login authentication system which depends on the Employee database. This db houses login info for each staff member, paycheck, position, and other relevant information. The other components of the system can only be accessed after passing through this step. Afterward, users can either follow one of two paths: accessing inventory or using the transaction system.

The IPT (inventory processing system) includes the functions necessary to interact with system data.The IPT is also associated with some general filters that interact with the inventory database. These filters limit the scope of what the user might be searching for, before entering the database itself. On the other hand, the transaction system will have necessary functionality to interact with third party payment systems. However, it depends on the info stored in the database to issue refunds and such.

## UML Class Diagram:



### Inventory Item:

The class that represents a particular raw inventory item. These will include raw ingredients, utensils, and any other restockable, purchasable item.

- Attributes:
    - **ID**: A unique integer representing this item. Used for quick reference by the database and other systems
    - **name**: The name of the item in question. Examples: “Short Grain White Rice”, “Napkin”, “Arugula”
    - **category**: What kind of classification the item falls under. Classifications are Food, Beverage, or Miscellaneous. Miscellaneous encompasses non-edible items like utensils and napkins.
    - **quantity**: How many items the restaurant has in storage.
- Operations:
    - **getQuantity**(): returns the quantity of the item
    - **setQuantity**(): sets the quantity of the item to an inputted number
    - **getName**(): returns the name of the item
    - **setName**(): changes the name of the item to an inputted name
    - **getID**(): returns the id of the item
    - **setID**(): sets the id of the item to an inputted integer
    - **remove**(): removes the item from database and storage

### Inventory Processing:

The system that processes the quantity of inventory items available. It interacts with the database to receive information about availability of inventory items. It provides information about how many menu items can be created with the current stock.

- Operations:
    - **getItemByID**(int): Given an ID, fetches the corresponding inventory item from the database and returns it. Allows for faster querying compared to searching by name.
    - **getItemByName**(String): Searches for the inventory item with a given name in the database
    - **orderMenuItem**(MenuItem, int): Will change the count of items in the inventory database after a particular Menu Item has been created. For example, if “Sushi” is ordered, will recalculate the amount of rice, fish, seaweed, and other ingredients in storage. Returns a string formatted for use with the Transaction System, includes name of Menu Item and quantity purchased.
    - **quantityCanMake**(MenuItem): returns the total number of Menu Items that can be made based on how many ingredients are in the inventory database. For example, if “Sushi” is inputted and there is only enough rice for 5 Sushi rolls, then the function returns 5.

**Worker:**

This class represents any worker in a particular restaurant.

- Attributes:
    - **username**: a unique string attribute used to represent the user’s username when logging into the system through the login system
    - **name**: A string that represents the name of the user
    - **age**: An integer that represents the age of the user
    - **position**: A string that represents the position of the worker (Manager, server, hostess, chef)
    - **timeWorked**: A floating point value that represents how long the user has worked in a given time frame
    - **email:** The worker’s email address. They provide this when they create their account.
- Operations:
    - Any worker in our system can use two operations
        - **sendOrder**(MenuItem[]): a function that sends an order to every user in the system so that everyone can view a customer’s order.
            - The purpose of this function is mainly for seamless communication between chefs and servers. Chefs can view the orders of tables taken by the server. It also allows anyone to view a table’s order so that delivering food to a table becomes easier.
        - **updateStock**(InventoryItem): a function that allows the user to update the quantity of certain items
            - The purpose of this function is to update the stock so that staff won’t be misinformed about an item is in stock and make the correct adjustments when informing a customer

                - The worker class acts as a superclass to two subclasses:
                    - Managers have certain operations that a regular worker does not have
                        - **voidItem():** This function can void any item that has been already ordered
                            - The purpose is if a user sends an order by mistake, the manager can void the item so the customer won't be charged.
                        - **refundItem**(): This function can refund an item that has already been purchased
                        - **shiftTimeWorked:** This function allows for the manager to change the time clock of a worker so that their hours worked can be correctly calculated when calculating hourly pay.

**Menu Item:**

This class represents the objects stored inside the Inventory Database.

- Attributes:
    - **id**: A unique integer value that is associated with a specific product. Users can access Inventory objects either by getting the id or name.
    - **name**: A String associated with a specific product used to access Inventory items from the Inventory database.

**Transaction:**

This class represents the object stored inside the Transaction Database.

- Attributes:
    - **Type**: A string that specifies the parties involved in the transaction. "Order" means that the transaction is a customer purchasing a menu item, such as food or drink. "Restock" is a purchase from a supplier for ingredients.
    - **Amount**: A float that is the amount paid for the specific transaction.
    - **orderNumber**: An integer that associated with the specific transaction
    - **items**: This in an array of Strings in the with each String having the below format:
        - "ID: number purchased"
    - **date**: A Date object that reflects the time of purchase.
    - 
- Operations:
    - **getInfo()**:This will return a string representation of the values held by the members of an inventory object. This method will be used by the user when trying to refund a customer.

**Transaction System:**

The system that interacts with the Transaction Database to process and provide relevant information about restaurant-customer and restaurant–supplier transactions. A transaction is defined as an exchange of money for an item.

- Operations:

  - **handlePayment**(Transaction): adds the provided Transaction to the Transaction Database. Attempts to process the payment between each party involved. More arguments may be added depending on how the payment is processed.
  - **updateRefund(**Transaction): Attempts to process a refund between each party involved in a given Transaction.
  - **getTransactionReceipt**(Transaction): returns a String that represents the text to be printed on a receipt of the transaction. This will largely be used for restaurant-customer transactions.

**Login System:**
The system that validates restaurant staff logins. Interacts with the Employee Database to receive and send information about workers' accounts, usernames, and passwords.

- Operations:
  - **validatePassword**(String user, String pass): The first string is the username of an employee's account. The second string is a password. This function returns whether or not the inputted password is the correct one for the user's account.
  - **forgotPassword**(String username): Sends a form to the inputted user's email to reset their password.
  - **getWorker**(String username): Returns the Worker object representing an employee's account. Used to access their account information.

## Development Plan and Timeline

Start Date: November 1st 2024
End Date: December 1st 2024

The time frame being allocated for this project is approximately one month. The initial two weeks will consist of team members working on their individual projects or components. The following list shows the divvying up of tasks amongst team members:

Cody ⇒ Inventory: This encompasses creating the Inventory Objects and associated function to interact with the database.
Andrew ⇒ Login: Creation of the login authentication component that interacts with an employee database to ensure workers can access the overall system.
Husain ⇒ Payment: Working with external third party system to accept transactions and track info through Transaction objects stored in the Transaction DB.

After doing unit testing and such to ensure our respective components are working, the second two weeks are going to be integration and system testing. During this time, team members will be collaborating actively to ensure seamless integration of components for the final deliverable.

# Test Plan

## Updates

added clockInOut() method under worker class in the UML

- A worker should have the ability to clock in and out so that their shiftTimeWorked can be updated and we can manipulate that data if needed.

added updateTimeWorked() method under manager in the UML

- A manager should be able to shift the time clock of the worker, this is so that the worker time clocks are correct respective to the time they actually worked and any worker that tries to stay on the clock longer than they are supposed to will not be paid for those hours on the clock.

changed validatePassword() to validateUser in the UML

- This method name made more sense because a user would have to enter both a username and a password which means both fields would have to be validated not just the password field.

change Transaction.type to mean “Order” (customer buying food) or “Restock” (buying ingredients from supplier)

- This usage of the term “type” was more relevant for the user than its previous usage. The type of payment processor could be abstracted out, and knowing if the transaction was between a customer or supplier was more important.

change orderMenuItem to return a string, used for Transactions, also added the description to specify that the int is the amount of items ordered

- Changing the orderMenuItem to return a string allowed for interoperability with the Transaction system. Adding the description for the int was just for clarification, and was meant to be added in the last version

## Verification Test Plan

### Inventory Processing

#### Unit test

```
InventoryProcessing ip = new InventoryProcessing()
item = ip.getItemByID(13)
assertEquals(“Sushi”, item.name)
```

This unit test is to ensure the functionality of the Inventory Processing class getItemByID method. I instantiate the IP class, call the respective method, and store the object in a variable called item. I then check if the name attribute of the object matches the item I was trying to retrieve from the database.

### Integration test

```
InventoryProcessing ip = new InventoryProcessing()
MenuItem salmon = new MenuItem(“SalmonRoll”, 6)
String s1 = ip.orderMenuItem(salmon, 2)
// price of salmon roll is $6, we’re ordering 2 of them
MenuItem tuna = new MenuItem(“TunaRoll”, 6)
String s2 = ip.orderMenuItem(tuna, 3)
// price of tuna roll is $5, ordering 3 of them

Transaction t = new Transaction(type = “Order”, items = [s1,
s2])
assertEquals(t.date, Date.Today)
assertEquals(amount, 27)
// make sure the transaction totals to $27 (2 * 6 + 3 * 5)
```

This test ensures proper functionality between the Inventory and Transaction systems. A couple of MenuItems are created that represent Salmon Rolls and Tuna Rolls. They are then ordered through an InventoryProcessing class.  The outputs of the orderMenuItem calls are then used to create a Transaction class that represents a customer’s order of 2 Salmon Rolls and 3 Tuna Rolls. The test checks that the date of the Transaction is accurate, and the total price is accurate.

### System Test

A customer gives the server their order (example: 1 sushi, 2 sodas, 2 fried rice, 1 miso soup), and the server provides the proper inputs to the IPS to reflect the order items requested. The IPS internally uses filters to cut down the search scope and then updates the respective inventory objects' quantity attributes. The customer order has to then be passed to the transaction system to create the proper transaction object, and stored in the database. To check if the IPS worked correctly, we would have to get the quantity before and after the object attributes are updated. If prevQuantity - numberOrdered = obj. quantity, then we can see that the system worked as intended.

## Worker/User

### Unit test

```
Worker andrew = new Worker()
timeWorked = andrew.getTimeWorked()
newTimeWorked = andrew.updateTimeWorked(5.14)
if(assertEquals(newTimeWorked, timeWorked) == True)
      return FAIL
```

```
else
     Return PASS
```

This unit test tests the method a manager can access, “updateTimeWorked()”. We are checking if the time clocked in from the worker is the same as the updated time worked. If we are updating the time a worker is clocked in we expect a different value compared to the new updated time. If the values are equal it results in a failed test, if they are not equal it results in a passed test.

### Integration test

```
Worker w = new Worker()
InventoryProcessing ip = new InventoryProcessing()

prevQuanity = ip.getItemByName(“Fish”).quantity()
w.updateStock(“Fish”, 21)


assertEquals(ip.getItemByName(“Fish”).quantity(),prevQuantity+21)
```

This integration tests if a worker can update inventory and if those changes are reflected in the inventory database. A worker, represented by the worker object, calls the updateStock method that accesses an Inventory item quantity attribute and modifies it. To check if the worker method worked, we use the IPS to get the object quantity before the update. We can claim that this integration test worked.  the new quantity == prev quantity + new stock.

### System Test

A worker logs into the POS system (providing a username and password input), and the system checks if the inputs provided are stored in the EmployeeDB. If the inputs were not found, the system would prompt the user to re-enter their information. If the user forgot their password, they could select an option to recover or change their password. When the user correctly inputs a valid login, they can now access functions under the worker class. On the other hand, if the user logs in with the manager’s information, they will be able to access all functions of a worker in addition to the manager.

## Data Management Strategy

We felt that the Software Architecture Diagram (SWA) did not need any changes, as the current design properly reflected the three databases we went with for our data management strategy.

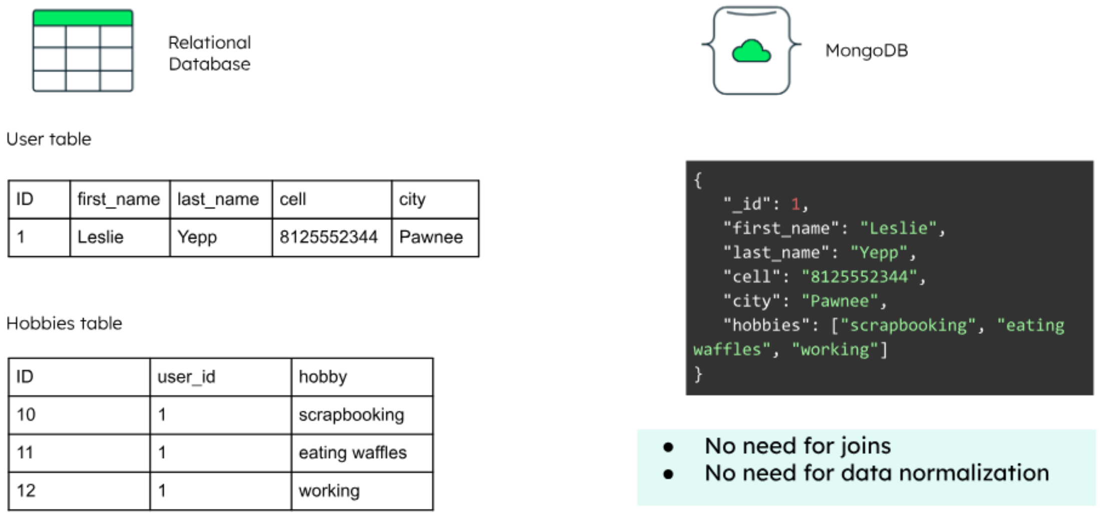
As demonstrated in our SWA, we implemented three separate databases for this system: Inventory, Transaction, and Employee. The inventory database includes the quantity of food items, beverages, and miscellaneous restaurant items. We grouped these together because logically, these different kinds of items are all classified as “inventory” and are treated by the system in the same way.

Maintaining security in a system when dealing with payments is a must, and as such, we separated all the transaction information from the other system data.

Employee information was also separated into its own database, and we chose to split the data to work with our planned software architecture diagram. We have a login authentication system that only needs to interact with relative employee info, and not data pertaining to restaurant inventory or sales.

Our specified data management strategy was SQL rather than NoSQL because the data needed for our restaurant POS is highly relational. Each record can represent a particular item, transaction, or employee, with each field representing a particular piece of information about it. Furthermore, this data is not large or rapidly changing, so we can confine it to relational table representation.

A possible alternative to using SQL would be to use document-based NoSQL such as MongoDB to store our data. Compared to SQL, the document-based data management system would be beneficial as you can use arrays in the key-value pairs. Right now we have to parse out the information from an array containing order items to a string before it can be stored in the Transaction database. Even though this alternative allows for greater flexibility with data scaling and types, it has a lot more overhead and is more complicated than SQL. The simplicity and ease of use of SQL is ideal for a relatively simple and small system like ours.



User table

| ID | first_name | last_name | cell | city |
|---|---|---|---|---|
| 1 | Leslie | Yepp | 8125552344 | Pawnee |

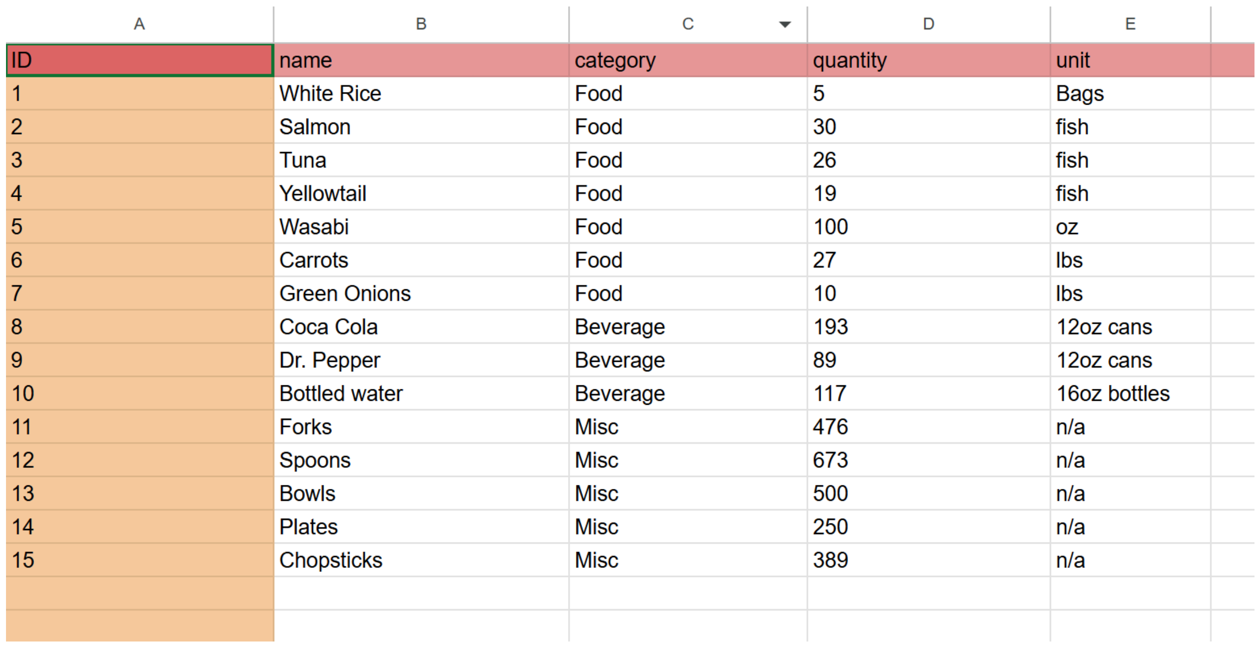
Hobbies table

| ID | user_id | hobby |
|---|---|---|
| 10 | 1 | scrapbooking |
| 11 | 1 | eating waffles |
| 12 | 1 | working |

```
{
   "_id": 1,
   "first_name": "Leslie",
   "last_name": "Yepp",
   "cell": "8125552344",
   "city": "Pawnee",
   "hobbies": ["scrapbooking", "eating
waffles", "working"]
}
```

- No need for joins
- No need for data normalization

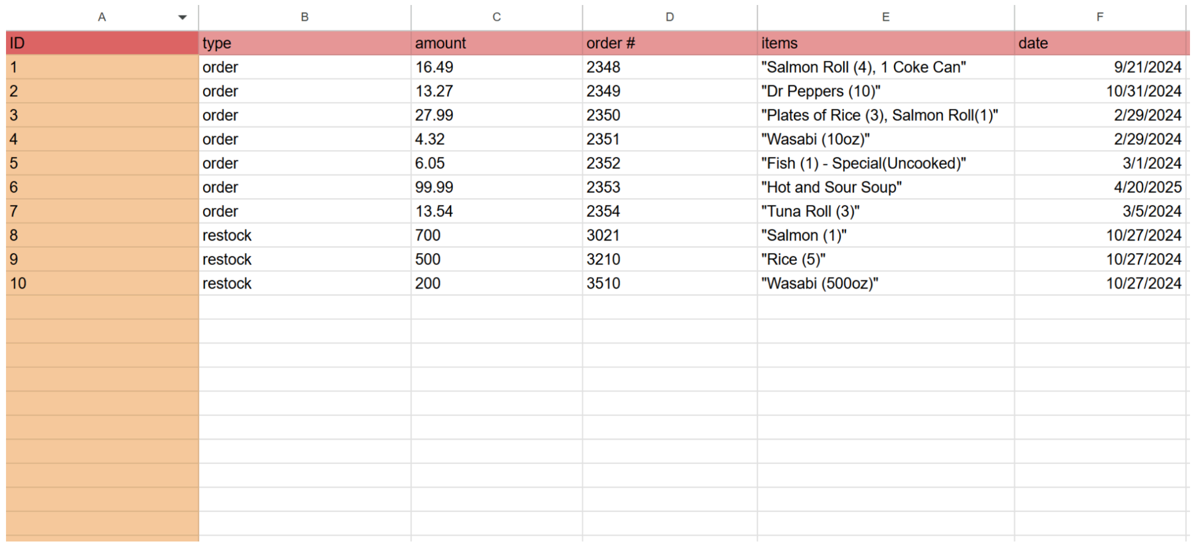
Inventory Database



| ID | name | category | quantity | unit |
|---|---|---|---|---|
| 1 | White Rice | Food | 5 | Bags |
| 2 | Salmon | Food | 30 | fish |
| 3 | Tuna | Food | 26 | fish |
| 4 | Yellowtail | Food | 19 | fish |
| 5 | Wasabi | Food | 100 | oz |
| 6 | Carrots | Food | 27 | lbs |
| 7 | Green Onions | Food | 10 | lbs |
| 8 | Coca Cola | Beverage | 193 | 12oz cans |
| 9 | Dr. Pepper | Beverage | 89 | 12oz cans |
| 10 | Bottled water | Beverage | 117 | 16oz bottles |
| 11 | Forks | Misc | 476 | n/a |
| 12 | Spoons | Misc | 673 | n/a |
| 13 | Bowls | Misc | 500 | n/a |
| 14 | Plates | Misc | 250 | n/a |
| 15 | Chopsticks | Misc | 389 | n/a |

This is a SQL database that stores information for the restaurant's inventory items. Each record (the rows of the table) represents an item.
**ID:** A unique integer that corresponds to this entry, for SQL lookup
**Name:** The name of the item
**Category:** Whether the item is food, a beverage, or miscellaneous (utensils, napkins, etc).
**Quantity:** How many of this item is in storage
**Unit:** The unit of measurement for quantity

Transaction Database

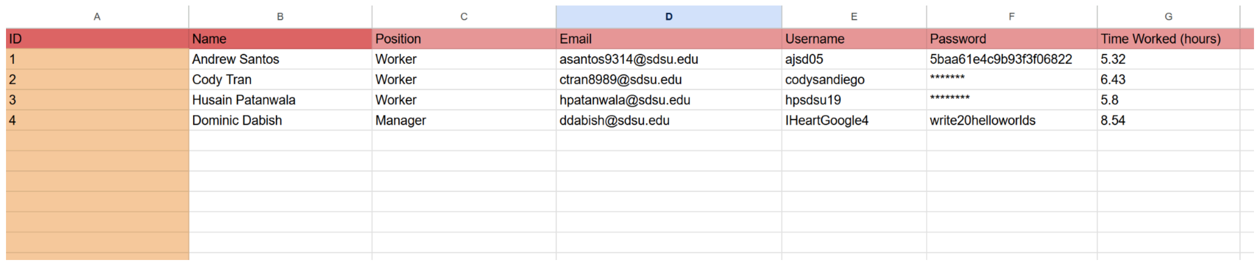


| ID | type | amount | order # | items | date |
|---|---|---|---|---|---|
| 1 | order | 16.49 | 2348 | "Salmon Roll (4), 1 Coke Can" | 9/21/2024 |
| 2 | order | 13.27 | 2349 | "Dr Peppers (10)" | 10/31/2024 |
| 3 | order | 27.99 | 2350 | "Plates of Rice (3), Salmon Roll(1)" | 2/29/2024 |
| 4 | order | 4.32 | 2351 | "Wasabi (10oz)" | 2/29/2024 |
| 5 | order | 6.05 | 2352 | "Fish (1) - Special(Uncooked)" | 3/1/2024 |
| 6 | order | 99.99 | 2353 | "Hot and Sour Soup" | 4/20/2025 |
| 7 | order | 13.54 | 2354 | "Tuna Roll (3)" | 3/5/2024 |
| 8 | restock | 700 | 3021 | "Salmon (1)" | 10/27/2024 |
| 9 | restock | 500 | 3210 | "Rice (5)" | 10/27/2024 |
| 10 | restock | 200 | 3510 | "Wasabi (500oz)" | 10/27/2024 |

This is our SQL Transaction database which stores the transactions made in our restaurant. We store the customer transactions and the purchases made by our restaurant like restocking on a certain ingredient. Each row represents a transaction and the information associated with it.

**ID:** A unique integer that corresponds to this entry, for SQL lookup
**Type:** There can be two types: whether it is a *customer* ordering a menu item or a *restock* order from a supplier to buy more ingredients
**Amount:** The cost of the order
**Order Number:** A number associated with each order used by customers to identify their order
**Item:** The items purchased in that particular order
**Date:** The date when the order was made

Employee Database



| A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|
| ID | Name | Position | Email | Username | Password | Time Worked (hours) |
| 1 | Andrew Santos | Worker | asantos9314@sdsu.edu | ajsd05 | 5baa61e4c9b93f3f06822 | 5.32 |
| 2 | Cody Tran | Worker | ctran8989@sdsu.edu | codysandiego | ******* | 6.43 |
| 3 | Husain Patanwala | Worker | hpatanwala@sdsu.edu | hpsdsu19 | ******** | 5.8 |
| 4 | Dominic Dabish | Manager | ddabish@sdsu.edu | IHeartGoogle4 | write20helloworlds | 8.54 |

This is our SQL Employee Database which stores all of the employees of our restaurant. Each employee has information tied to them such as their position, email, username and password to log into our system, and the amount of time worked in that given week. Each row represents a singular employee and the data associated with them.
**ID:** A unique integer that corresponds to this entry, for SQL lookup
**Name:** Name of the employee
**Position:** What position they are in the restaurant (Worker, server, manager, chef)
**Email:** The email associated with that employee
**Username:** The username the employee uses to login to the system
**Password:** The password the employee uses to login to the system
**Time Worked:** The amount of hours the employee worked in that week

https://docs.google.com/spreadsheets/d/1VvZSXmh0XLxKqtGOVCKUFB9FB_o8asNpSMQIfxG3mvI/edit?usp=drive_link