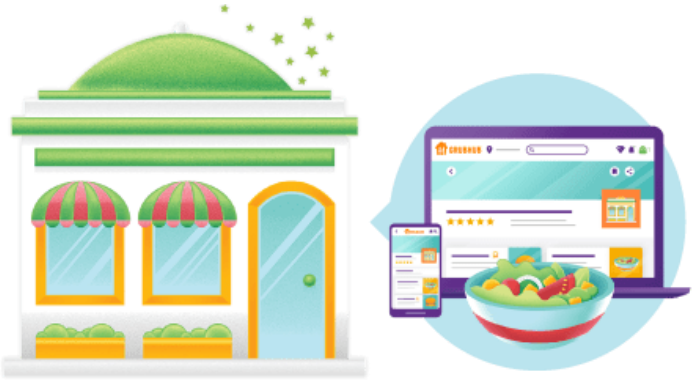
Virtual Restaurant

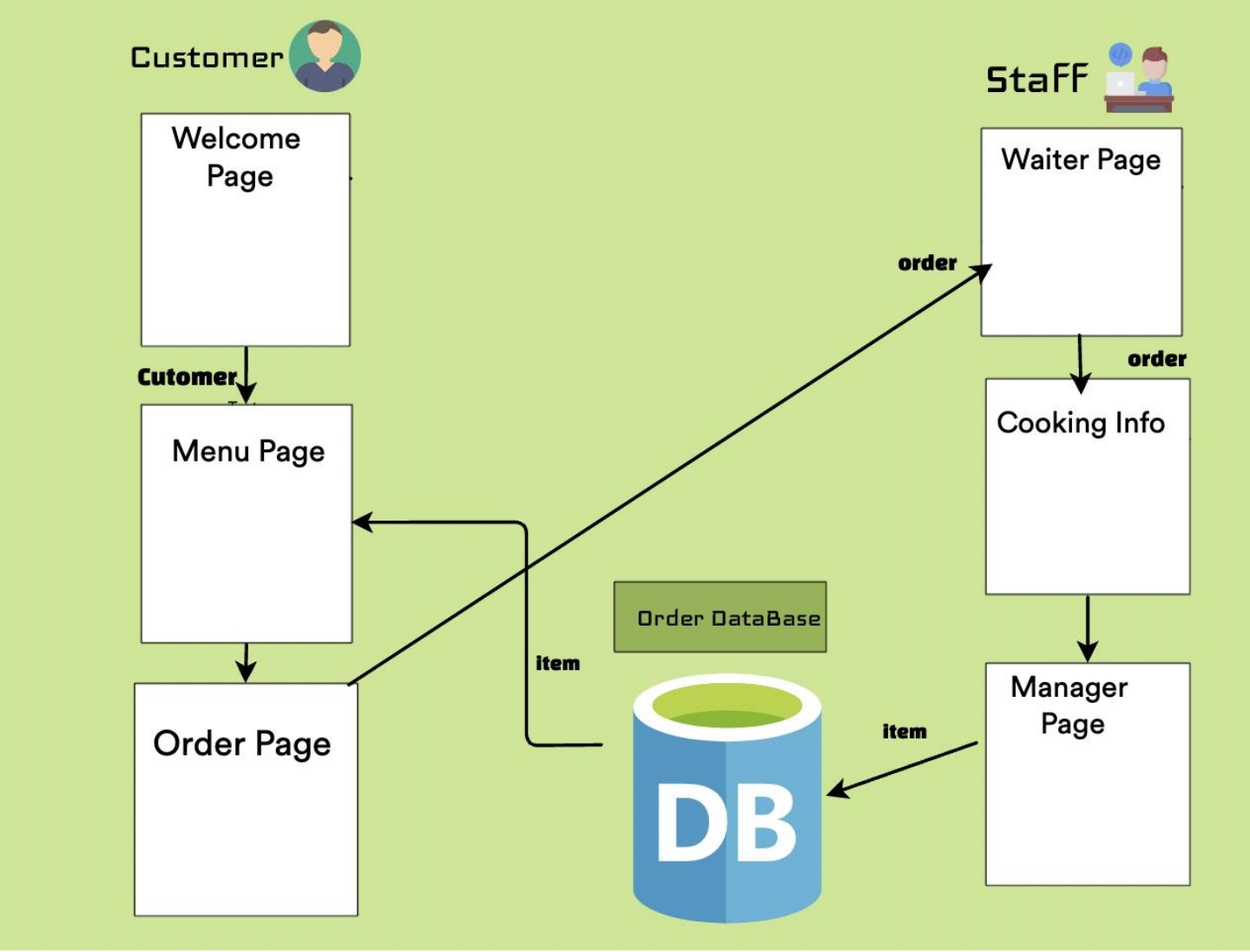


Team Members: Edgar Navarro, Sarah Azzalddin, Joshua Brown.

Software Description

Imagine yourself visiting a restaurant that functions virtually. This software will be the implementation of this imagination. The virtual restaurant is intended to accept customers who are seated at tables and waiters accept food orders. Meanwhile, there is a kitchen that has a menu of orders and prepares meals for customers. Our dearest hope is that this restaurant targets beyond what you expect!

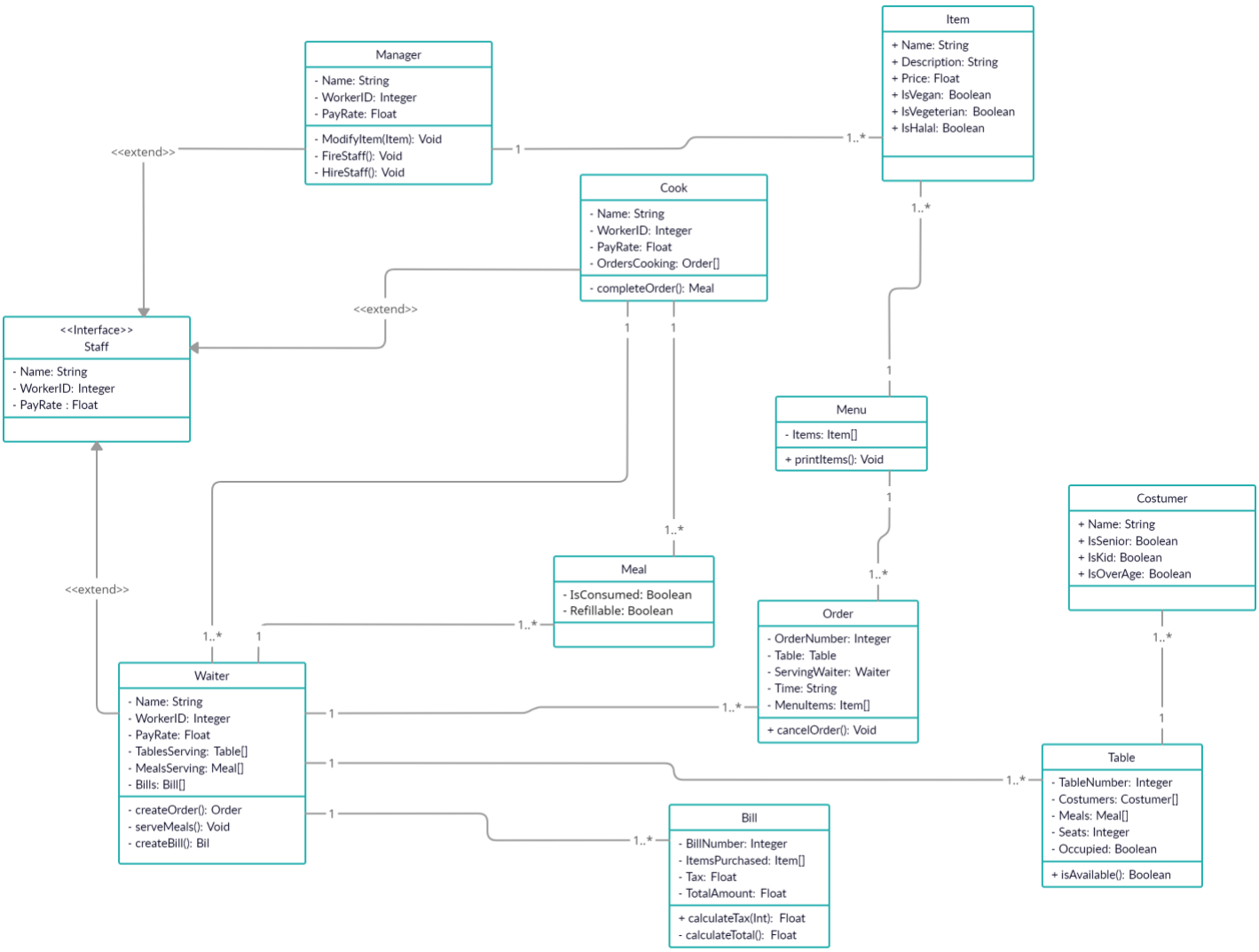
1. Architecture Diagram



Architecture Diagram Description:

This is a visual representation of the software system for Virtual Restaurant. It includes the process of a walkthrough for a customer. For example, the welcome page is the first page that the customer will see and can enter the customer information. Then, the Menu Page is where the customer chooses food items from the menu. Next, the waiter (staff) will receive the order information from the Order Page (customer). In essence, customer interaction ends on the Order Page. In addition, there is Order Database that holds the data for Items' descriptions and names.

Class Diagram



Description of Classes

There are 10 classes which are Customer, Table, Bill, Waiter, Item, Menu, Order, Meal, Cook, and Manager. Each at a varying level is intercorrelated with others to serve as a virtual restaurant. An instance of the Customer class is created with the name and age of the customer as login information and the Table class holds an array of Customer class objects. Now, the waiter collects the incoming data from the customer and sends it to the Cook class for the food to be prepared. The Meal object is created by the Cook and the Waiter will then serve the meal to the table. As for the Staff interface, this is where it serves as a template for the staff which are the Waiter, Cook, and Manager.

Description of Attributes

Staff

- Name (string): it contains the **Staff** member's name.
- WorkerID (int): it is a **Staff** member's unique ID number.
- Payrate (float): **Staff** member's wage (calculated by hourly pay rate * position rate).
    - Extends these attributes to **Cook**, **Waiter**, and **Manager** classes
- **Waiter**
    - TablesServing[]: It is stored as a list of **Tables**, tracks which **Tables** are assigned to the **Waiter.**
    - MealsServing[]: It is stored as a list of **Meals**, tracks which meals the waiter is currently bringing to the **Table.**
    - Bills[]: It is stored as a list of **Bills**, contains all **Bills** charged to a specific table.
- **Cook**
    - OrdersCooking[]: It is stored as a list of **Orders**, and contains all meals the **Cook** is currently preparing.
- **Manager**
    - It inherits all attributes from **Staff**: it does not contain unique attributes.

Customer

- Name (string): It contains the **Customer's** name.
- IsSenior (boolean): It is **Customers** over the age of 55 who receive a discount when it is true.
- IsKid (boolean): It is **Customers** under the age of 10 who receive a discount when it is true.
- IsOverage (boolean): It is **Customers** over the age of 21 can purchase alcohol when it is true.

Table

- TableNumber (int): It is a unique identifier for each **Table**
- Customers[]: It is stored as a list of **Customers,** tracks which customers are seated at the **Table**
- Meals[]: It is stored as a list of **Meals,** tracks which meals are currently on the **Table.**
- Seats (int): It serves as a size limit for Customers[], ie. It is how many seats each table has.
- Occupied (boolean): When it is true, the table has at least one seat occupied.

Meal

- IsConsumed (boolean): When it is true, signals to the **Waiter** that the customer has finished their meal
- Refillable (boolean): When it is true, customers are given the option to have another portion served to them

Item

- Name (string): It contains the name of the **Item.**
- Description (string): It is a brief description of the **Item.**
- Price (float): It is how much the **Item** cost.
- IsVegan, IsVegetarian, IsHalal (boolean): When it is true, it signifies if an **item** falls within certain dietary restrictions.

Menu

- Items[]: It is stored as a list of **Items.** It contains all items currently on the menu.

Order

- OrderNumber (int): It is a unique identifier for each **Order.**
- Table (Table): It contains the **Table** from where the **Order** comes.
- ServingWaiter (Waiter): It contains the **Waiter** that is responsible for the **order.**
- Time (String): It stores the time that the **Order** was made.
- MenuItems[]: It stores a list of **Items**. It contains every item that the **Customer** has ordered.

Bill

- BillNumber (int): It is a unique identifier for each **Bill.**
- ItemPurchased[]: It is an array of **Item** objects that contains the items that the customer.
- Tax (float): It is the total amount of tax that is charged to the **Table.**
- Total (float): It is the total cost including tax charged to the **Table.**

Description of Operations

Manager

- ModifyItems(Item) *void*: It updates **items** on the **Menu**. It can also add or remove **Menu Items.**
- HireStaff() *void*: It creates a new instance of **Staff** and it designates whether they become **Cook** or **Waiter.**
- FireStaff() *void*: It removes **Staff** members from the restaurant.

Cook

- completeOrder() *Meal*: It provides the **Waiter** with a **Meal,** to bring to its respective **Table.**

Waiter

- createOrder() *Order*: It creates a new instance of **Order,** with each **Menu Item,** specified by the **Customer.**
- serveMeals() *void*: It populates the **Table** with each **Meal** the **Customer** has ordered.
- createBill() *Bill*: It compiles all charges and taxes for the **Table** the **Waiter** is currently serving.

Menu

- printItems() *void*: It displays the entire list of **Items** on the **Menu** to the **Customer**.

Order

- cancelOrder() *void*: It cancels an **Order** in case there was an error made by the **Customer** or the **Waiter**.

Table

- IsAvailable() *boolean*: Returns the status of the table to whoever calls it, without having to make this information public.

Bill

- calculateTax(Int), calculateTotal() *float*: It generates the correct cost for the **Table** that the **Bill** is assigned to.

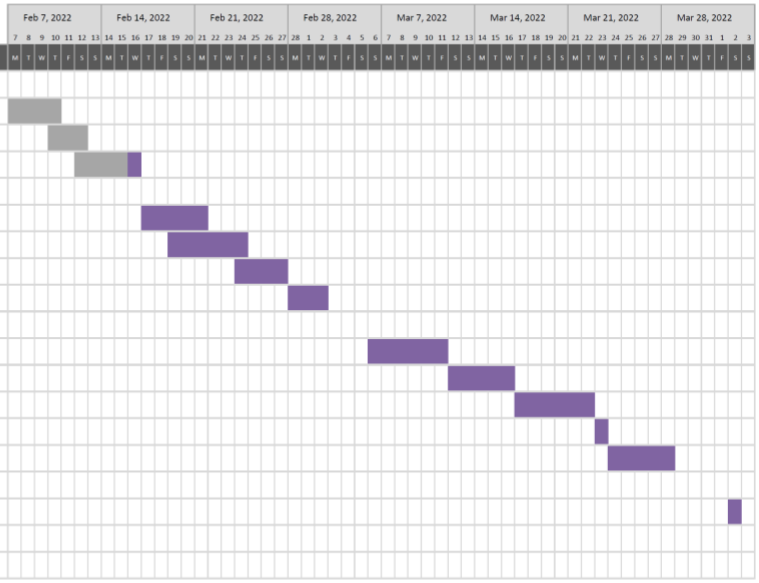
Development Timeline

Virtual Restuarant

Project Start: Thu, 2/10/2022

Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	END
Requirment Gathering				
Client meetings	Sarah A.	100%	2/7/22	2/10/22
Requirment Specification	Sarah A.	100%	2/10/22	2/12/22
Software Design Specification	Sarah A.	80%	2/12/22	2/16/22
Development				
Complete Staff classes	Edgar N.	0%	2/17/22	2/21/22
Complete Menu System	Edgar N. & Joshua B.	0%	2/19/22	2/24/22
Complete Customer and Table System	Joshua B.	0%	2/24/22	2/27/22
Implement Relations	Edgar N. & Joshua B.	0%	2/28/22	3/2/22
Testing				
Establish Functioning Systems	Edgar N. & Joshua B.	0%	3/6/22	3/11/22
Complete Debugging	Edgar N. & Joshua B.	0%	3/12/22	3/16/22
Distribute Functioning Protoype	Edgar N. & Joshua B.	0%	3/17/22	3/22/22
Client Meeting	Sarah A.		3/23/22	3/23/22
Implement Feedback	Edgar N. & Joshua B.	0%	3/24/22	3/28/22
Release and Maintain				
Release Functioning System	Edgar N. & Joshua B.	0%	4/2/22	-
Release Patch	TBD	0%	TBD	-



2. Test Plan

Brief Introduction: In this part of the assignment, we will "exercise software to try and make it fail." This means that we implement software testing methods when the expected output does *not* equal the observed input. The goal of the testing is to detect and locate any possible errors, failures, or faults. There are three essential testing levels that are: **Unit Testing** (single unit), **Integration Testing** (relationship between integrated units), and **System Testing** (the entire system).

Unit Testing

Based on the provided class diagram above, we choose to test the method of CalculateTax() in the class 'Bill.'

```
 If (Bill ≤ 0) {
          return null;
          print "Error";
}
 else if (Bill > 0) {
          return = (CalculateTax (3));
}
      if (return (1)){
         pass;
          else {
              fail;
}
```

Integration Testing

Integration Test for Manager

First, we will check the ModifyItem() method, to verify if the Manager class can use this method to update Items within the Menu class, as well as add or delete Items. We will do this by creating a new instance of Manager (Gordon) and a new instance of Item (Hamburger). Then we will we will change the price of the item through ModifyItem()'s built in functionality. Next we plan to test both the HireStaff() and FireStaff() methods to make sure that the Manager class has the power to create new instances of Waiter and Cook, while also having the power to remove them. We will do this by calling the HireStaff() method twice, creating a new instance of Cook and Waiter, then we will call the FireStaff() method twice to fire those two new employees.

```
Manager Gordon;
Item Hamburger; // default price set by Item constructor $0.00
Gordon.ModifyItem(Hamburger)
        **Change price from $0.00 to $9.00
Gordon.HireStaff();
        **Hire a new Waiter
Gordon.HireStaff();
        **Hire a new Cook
Gordon.FireStaff();
        **Fire Watier
Gordon.FireStaff()
        **Fire Cook
```

System Testing

This is a System level test. It will follow the expected procedure, through the Restaurant system. The system begins with a number of Customer objects being seated to

their tables. Once seated the Table object's data will be updated to contain the array of customers seated there. The Waiter object's responsibility is to attend their assigned tables by creating orders, serving meals, and creating bills. The Waiter object therefore interacts with multiple other objects including: Order objects, Table objects, Bill objects, Meal objects, and Cook objects. Primary interactions include those of giving Order objects to the Cook object, and receiving Meal objects from the Cook object, finally giving the Meal object to the respective Table object to be consumed by the Customer. After this process is complete the Waiter object is checking constantly on the status of Meal objects until they are consumed, then creates a Bill object with the respective information.

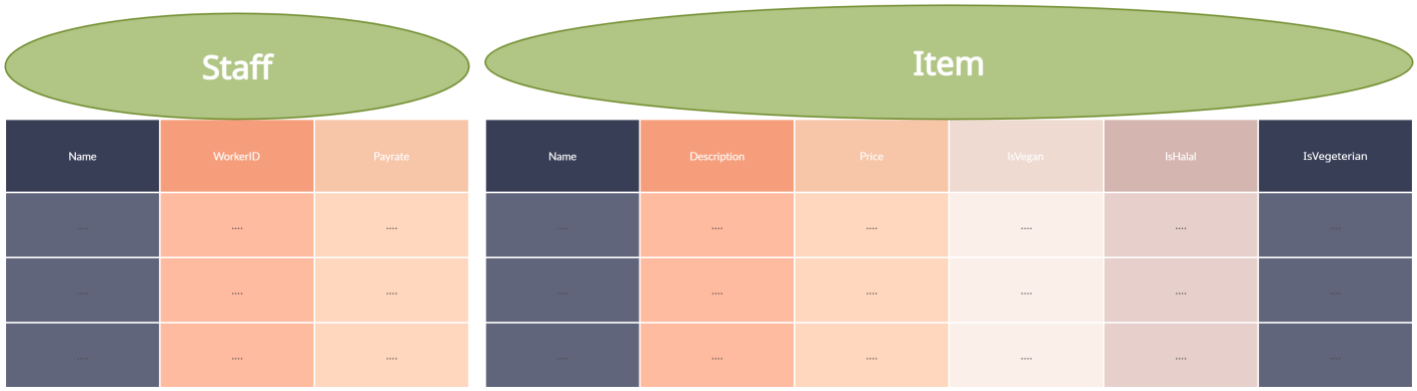
The System also involves operations that do not directly affect the Customer objects, such as the Manager object's interactions. The Manager class can be seen as a Factory class, because its job is to generate new Staff members or remove existing ones, essentially deleting and creating objects from their specified classes. However, the Manager object also has the ability to interact with Item objects, that form part of the Menu object.

Together these functions make up the Restaurant system, and enable essential functionality to operate a Restaurant. From Waiters creating Orders for Tables and Customers consuming their Meals, to Managers hiring and firing staff and editing the existing Menu.

3. Data Management Strategy

Staff:
Name: Sarah
Description: Virtual Restaurant Manager
Type: Table
Primary Index: Name
Secondary Index**:** WorkerID

Item
Name: Vegan Bowl
Description: Virtual Restaurant Item
Type: Table
Primary Index: Name
Secondary Index: Description



We will be implementing a modular data collecting/storing system in order to organize our data into relevant sections. All data will be stored in one of three distinct databases, Staff data, Restaurant data, and User data. This divided approach will allow our system to have more efficient access to the data it needs while simultaneously reducing the amount of overlap in our system. This design will also make sure to hide any sensitive data from users of the system.

- **Staff Data:** this database is responsible for holding the data of all staff members of our Virtual Restaurant, namely each staff member's Name, WorkerID, and Payrate. This database, however, will not hold class-specific data held in the Waiter and Cook classes (ex. TablesServing[], OrdersCooking[]), as these will be handled by the Restaurant data section. This database will not be accessed as much as the other two databases in our system, as it will primarily serve as a staff directory for our Virtual Restaurant. The manager class is the only class in our system that can modify this database.

- **Restaurant Data:** this database is responsible for holding all data that pertains to our Restaurant's operations, such as Menu, Item, Meal, Order, Bill, and Table class data, along with the aforementioned data from Cook and Waiter classes. Since these classes will be constantly interacting with each other as our system runs, we cannot afford to store this information in separate locations.

- **User data:** this database contains all data that is relevant to the User, and will be the only database that contains Customer class data. This database will not hold any data pertaining to Bill or Table classes, in an effort to hide as much data from the User as possible, but during runtime, this data will be made available to the User as needed through calls from the Restaurant database.

These databases will be implemented using SQL, since we are dealing with tabular data, as opposed to a NoSQL approach which is far better suited for non-tabular data (graphs, doc trees, key-value). Another merit of the SQL approach is that transactions are ACID-compliant (Atomicity, Consistency, Isolation, Durability) so our data changes as expected, and it is always in a consistent state across all the databases (at least the ones that share data). ACID also guarantees that transactions are always invisible, i.e. irrelevant data is hidden on one end of a transaction, while the relevant data on the other end is shown to the receiver and vice-versa. Furthermore, a NoSQL approach is optimized for the availability of data which is advantageous for large databases, but since our project is relatively small it can benefit from a SQL approach that favors consistency, so we deal with fewer discrepancies between the databases overall. While SQL databases do require vertical scaling to update systems, this may not be necessary for our project since it will require very limited hardware and CPU resources to run efficiently.