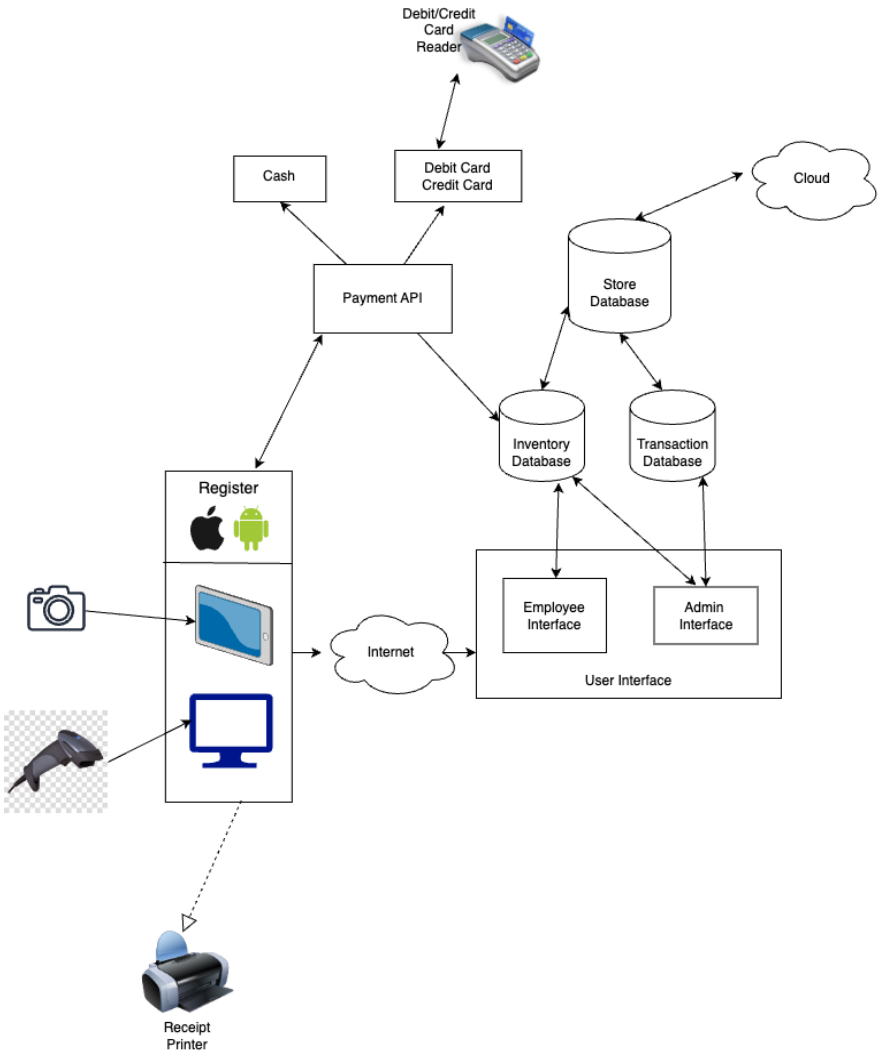
Clothing Store Point of Sale System

Prepared by: Steven Pun, Jared Robles, and Tammy Dahl

System Description

The Clothing Store Point of Sale (POS) System is designed to facilitate smooth and efficient retail operations for both employees and management. The versatile software can be accessed via store registers and portable devices with connectivity to the admin or employee interfaces, depending on user roles. The system centralizes transaction data and inventory management, offering real-time updates and synchronization across multiple store locations via cloud-based services. It also integrates payment processing for various methods, including cards and cash, alongside essential sales functions such as barcode scanning and receipt printing. This system streamlines in-store processes, ensuring operational efficiency and enhanced customer service.

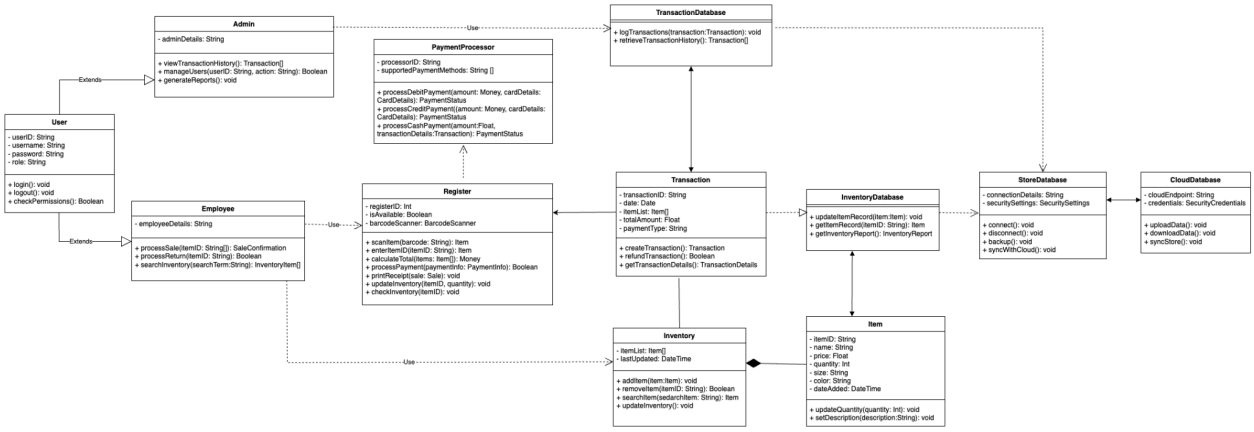
Software Architecture Diagram



Software Architecture Diagram Description

The Point of Sale System for the clothing store is designed to make sales efficient and organized for employees. The software can be accessed through store registers as well as phones and tablets with Apple or Android operating systems. These devices must first be connected to the internet, which will allow the user to access either an admin interface (for managers) or an employee interface. The admin interface allows the user to access the transaction database, which contains all the transaction history and data. Admin and regular employees can both access the inventory database, which they can update and search. These two databases make up the store database that is backed up by a cloud system, synchronizing data across different store locations. The inventory database is automatically updated through the Payment API, which handles payment from debit and credit cards, as well as cash. Credit and debit cards are processed through card readers. Employees are also able to scan items through barcode scanners. The items will then be visible on the device being used. After payment is processed, the transaction information will be printed out through a receipt printer.

UML Class Diagram



UML Class Diagram Description:

This UML diagram effectively visualizes the structure of a clothing store POS software system, outlining the relationships between users, transactions, payments, inventory, and data storage, both locally and in the cloud.

Connections:
Solid lines represent associations, showing how instances of one class are connected to instances of another, indicating relationships or collaboration between classes.
Dashed lines represent dependencies, indicating that a class uses another class in some capacity, but not necessarily holding a reference to it.

User (Abstract Class)

Overview: Represents a generic user in the system, serving as a base class for more specific types of users.

Attributes:
userID: Int - A unique identifier for the user.
username: String - The user's login name.
password: String - The user's password for authentication.
role: String - The role of the user within the system, e.g., Admin or Employee.

Operations:
login(): void - A method for the user to log into the system.
logout(): void - A method for the user to log out of the system.
checkPermissions(): Boolean - A method to check if the user has the permissions to perform a certain operation.

Admin (Extends User)
Overview: A specialized type of User with administrative privileges.

Attributes:
Inherits all attributes from User.

Operations:
viewTransactionHistory(): Transaction[] - Allows the admin to view the history of all transactions.
manageUsers(userID: String, action: String): Boolean - Enables the admin to manage users, including actions like create, update, or delete.
generateReport(): void - Allows the admin to generate different types of reports.

Employee (Extends User)
Overview: A specialized type of User who handles daily transactions and inventory management.

Attributes:
Inherits all attributes from User.

Operations:
processSale(itemIDs: String[]): SaleConfirmation - Processes the sale of items and returns a sale confirmation.
processReturn(itemID: String): Boolean - Processes the return of an item and indicates success with a boolean.
searchInventory(searchTerm: String): InventoryItem[] - Searches the inventory with a given term and returns a list of items.

Register
Overview: The point of interaction for processing sales and managing payments.

Attributes:
registerID: Int - A unique identifier for the register.
isAvailable: Boolean - Indicates if the register is available for use.
barcodeScanner: BarcodeScanner - An associated barcode scanner for inputting item data.

Operations:
scanItem(barcode: String): Item - Scans an item's barcode and retrieves the item details.
enterItemID(itemID: String): Item - Manually enters an item's ID and retrieves the item details.
calculateTotal(items: Item[]): Money - Calculates the total cost of items including taxes.
processPayment(paymentInfo: PaymentInfo): Boolean - Processes the payment and returns success status.
printReceipt(sale: Sale): void - Prints a receipt for the completed sale.
updateInventory(item: Item, quantity: Int): void - Updates the inventory based on the sale or return.
checkInventory(itemID: String): void - Checks the inventory level of a specific item.

PaymentProcessor

Description: Handles the processing of various types of payments, interfacing with external payment systems as necessary.

Attributes:

processorID (String): A unique identifier for the payment processor, which may be used to track transactions or configure processor settings.
supportedPaymentMethods (String[]): An array of strings representing the different payment methods supported by this processor (e.g., "debit", "credit", "cash").
Operations:

processDebitPayment(amount: Money, cardDetails: CardDetails): PaymentStatus - Processes a payment made by a debit card. The amount is a Money object representing the value of the transaction. cardDetails is a CardDetails object containing information such as card number, expiry date, and CVV. The method communicates with a bank or payment gateway to complete the transaction and returns a PaymentStatus object indicating the result of the operation.

processCreditPayment(amount: Money, cardDetails: CardDetails): PaymentStatus - Processes a payment made by a credit card. It uses the same parameter types as processDebitPayment and performs a similar action but may involve different processing rules or checks specific to credit transactions, such as pre-authorization or credit checks. It returns a PaymentStatus object that details the outcome of the credit payment.

processCashPayment(amount: Float, transactionDetails: Transaction): PaymentStatus - Records the details of a transaction paid with cash. While no electronic authorization is required, this method logs the transaction for record-keeping, ensuring cash payments are accounted for within the system. The amount is the cash received, and transactionDetails includes data such as transaction ID and items involved. The method returns a PaymentStatus that would typically indicate a successful recording of the transaction.

TransactionDatabase
Overview: Stores and manages the transaction records.

Operations:
logTransactions(): Records new transactions in the database.
retrieveTransactionHistory(): Retrieves the history of transactions.

Transaction

Overview: Represents a single financial transaction.

Attributes:
transactionID: String - A unique identifier for the transaction.
date: Date - The date the transaction took place.
itemList: Item[] - The list of items involved in the transaction.
totalAmount: Float - The total amount of the transaction.
paymentType: String - The method of payment used.

Operations:
createTransaction(), refundTransaction(), getTransactionDetails() - Creates a new transaction, processes a refund, and retrieves details of transactions.

InventoryDatabase
Overview: Manages the storage and retrieval of inventory records.

Operations:
updateItemRecord(), getItemRecord(), getInventoryReport() - Updates an item record, retrieves an item's record, and generates an inventory report.

Inventory

Overview: Represents the collection of all items available for sale.

Attributes:
itemList: Item[] - A list of all items in the inventory.

lastUpdated: DateTime - The last time the inventory was updated.

Operations:
addItem(), removeItem(), searchItem(), updateInventory() - Adds a new item, removes an item, searches for items, and updates the inventory.

Item
Overview: Represents an individual item in the inventory.

Attributes:
itemID: String - A unique identifier for the item.
name: String - The name of the item.
price: Float - The price of the item.
quantity: Int - The quantity of the item available.
size: String - The size of the item.
color: String - The color of the item.
dateAdded: DateTime - The date the item was added to the inventory.

Operations:
updateQuantity(), setDescription() - Updates the quantity of an item and sets its description.

StoreDatabase
Overview: Represents the local database where store-related data are kept.

Attributes:
connectionDetails: String - Information required to establish a connection to the database.
securitySettings - The settings to ensure secure access and data storage.

Operations:
connect(), disconnect(), backup(), syncWithCloud() - Connects to, disconnects from, backs up, and synchronizes the database with the cloud.

CloudDatabase
Overview: Represents the cloud-based service used for data backup and synchronization.

Attributes:
cloudEndpoint: String - The endpoint for connecting to the cloud service.
credentials - Authentication credentials for accessing the cloud service.

Operations:
uploadData(), downloadData(), syncStore() - Uploads data to, downloads data from, and synchronizes the store's data with the cloud.

Development plan and timeline

For this project, with building the interface as well as testing all of the functionalities, we anticipate that this project will take about 15 months to complete.

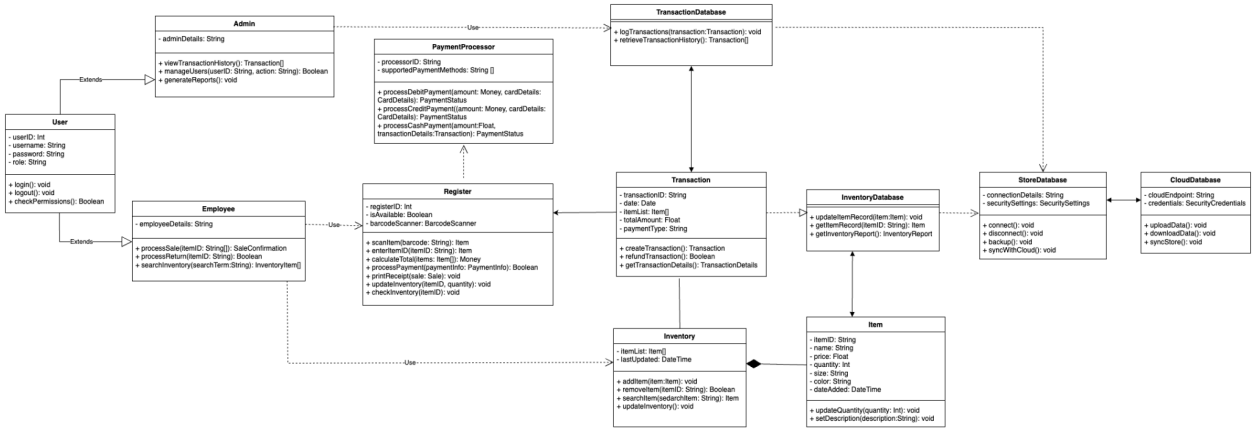
First, the software engineering team should develop the initial page that the employees interact with. The front-end developers would work on a system that the employees and admins would be able to see and interact with. The different functions that would be expected from them are creating a page for employees to log into the system, clock in for their shifts, and altering information about the clothing stock. On this page, the employees would be able to complete customer purchases, process returns as well and have a function to be able to sort through the different filters to categorize the clothing and make checking customers out easier. We anticipate that this development will take around 3-4 months.

On the other side of the program, the backend developers would work on creating the system that supports what the employees are interacting with. Here, the backend developers would make sure that the system is connected to the internet, running on multiple different servers, making sure that the system is secure and does not allow anybody without proper permissions to enter the system, as well as making sure that everything is backed up and makes it so that the system would not need maintenance during operational hours. A trivial part of the system would be the server that stores information about the clothing and should be able to keep information about every item, including the size, color, ID number, price, quantity of the item, date added, etc. This is to ensure that the employees can categorize each item while being able to filter through them while checking customers out. The system should have a third-party API that would connect to external services allowing the company to process payments through Apple Pay, other digital wallets, as well as various debit and credit cards. The backend developers would also be responsible for making sure that the inventory count is accounted for and for making sure that the items are properly tracked for when sales and refunds are made. We expect that this process will take around 8 months to complete.

Finally, the testing phase would be where the product is tested. We want to try and test every possible scenario with different items and variations to get a true understanding of what may be seen as an issue and what needs to be improved on. This would require both the front-end and back-end developers to test every possible scenario and report back to ensure that the program doesn't have any outstanding bugs and can fix any issues that present themselves during this testing phase. This phase should take around 3 months to ensure that everything is working properly and satisfies all needs.

Test Plan

Updated Software Design Specifications



Changes Made:

- In the user class, the userID was changed from a String to an Int. This change was made since the username would take in a string while the userID would only include a numerical ID.

Overall, there were not any major changes to our previous design. We decided to not make any more modifications because we felt that our Software Architecture Overview sufficiently addressed all necessary components without inaccuracies or missing information.

Test Set 1 : User Account Management

Unit Testing

Test Case: checkPermissions()
Validate that the checkPermissions() method accurately identifies an admin user's permissions

Input: Instantiate a User object for admin Greg with userID, username, password, and role as Admin. Then, instantiate a User object for employee Sue with userID, username, password, and role as Employee.

User checkPermissions()
User a
a.userID = "123" // changed userID type to Int on UML
a.username = "greg"
a.password = "test"
a.role = "Admin"

User b

```
b.userID = “456” // changed userID type to Int on UML
b.username = “sue”
b.password = “test”
b.role = “Employee”

IF(a.checkPermissions() == true)
        Return PASS
ELSE
        Return FAIL
```

Expected output: The method returns a pass for Greg, confirming he has admin permissions, and a fail for Sue since she has employee permissions.

How the test covers the targeted feature:
The checkPermissions() test case checks if the system correctly identifies user roles, such as distinguishing between admin Greg and employee Sue. By creating two users with different roles and testing if they have the correct permissions, we’re making sure that the system works as it should for role-based access. When Greg, the admin, is tested we expect the system to confirm that he has admin rights, which means he can access the transaction database. For Sue, the employee, we expect the system to show she doesn’t have permission. This allows us to test both scenarios and confirms that our system correctly gives access based on a user’s role. Which is key to making sure the right people have access to certain actions and information.

Functional Testing (Integration)

Test Case: viewTransactionHistory(): Transaction[] and retrieveTransactionHistory(): Transaction[] integration
Ensure that the viewTransactionHistory() method correctly retrieves the transaction history through the retrieveTransactionHistory() method and returns a correctly formatted list of transactions from the database.

Input:
Instantiate an Admin object for admin Greg with necessary details to log in and view transaction history.

```
SET admin = NEW Admin
admin.adminDetails = "Details about admin Greg"
SET transactionHistory = admin.viewTransactionHistory()

IF (transactionHistory != NULL)
        IF (transactionHistory is an array of Transaction)
            FOR each transaction in transactionHistory
                PRINT transaction
```

```
            EXIT
        PRINT "PASS: Transaction history successfully retrieved."
    ELSE
        PRINT "FAIL: Incorrect data type for transaction history."
ELSE
    PRINT "FAIL: Transaction history is null."
```

Expected output:
The expected output is non-null, and is correctly formatted as an array of Transaction objects. To confirm this, each transaction in the transactionHistory is printed, then the system will print "PASS: Transaction history successfully retrieved." If transactionHistory is null or not formatted as an array of Transaction objects, print “FAIL” with the appropriate failure reason.

How the test covers the targeted feature:
This test validates the integration between viewTransactionHistory() and retrieveTransactionHistory(), ensuring that the system can retrieve and display the transaction history accurately for an admin user like Greg. By checking that transactionHistory is non-null and correctly formatted we are also verifying that the data is presented in the expected structure. This is essential for admin users who rely on accessing and viewing transaction history for management and reporting purposes. The test confirms the systems ability to handle database interactions and present data correctly, which is fundamental for operational integrity.

System Testing

The system test should start with the admin attempting login into the Register using their ID, username, and password. Upon successful login, the system should verify the admin’s permissions before granting access to the transaction database. The admin should then be able to retrieve a list of transactions, select any transaction to view it’s details, and receive appropriate notifications in the event of retrieval issues, such as attempting to access transactions that don’t exist in the database. The test should cover logout procedures and ensure no unauthorized access is permitted once the session is terminated.

Test Set 2 - Inventory

Unit Testing

Test Case: searchItem()
Input: Input the itemID
Output: Item

Inventory i
i.addItem(“123”)
Inventory searchItem()

```
i.searchItem(“123”)
If (i.searchItem == i)
        Return Item
Else
        Return “Item is not found”
```

Features Tested:

1) Item Search - testing if employees are able to search for items in inventory
2) Accuracy - testing if the results are accurate and relevant to what is being searched

Test Sets:

1) Item that exists in inventory
2) Item that does not exist in inventory

How the Tests Cover the Targeted Features:

We first add an item to the inventory in order for it to be searched. After the item is added, we search the inventory using an itemID connected to the item. After the item is searched for, we either get the item or a message that states, “Item is not found”. This tests if the employees are actually able to search for items in inventory. Also, this tests the accuracy of the searches, because if we get a result that is not the item that we searched for, then we can conclude that the search feature is not accurate.

Functional Testing (Integration)

```
checkInventory (itemID): void
Inventory i
i.addItem(“123”)
Inventory searchItem()

i.searchItem(“123”)
checkInventory c
if (c.checkInventory(i) == searchItem)
        return item
else
        return “Item not found”
```

For the integration test, we are testing the feature for an employee or manager to be able to check on the stock of an item in the store on the kiosk as well as testing how accurate the system is at displaying the information. The selected test sets that will be applied will be to check whether the item is in stock or if the item does not exist/is out of stock. First, an item would be entered into the inventory information and the checkInventory function would exist within the program where an itemID can be entered into the system. The system will then

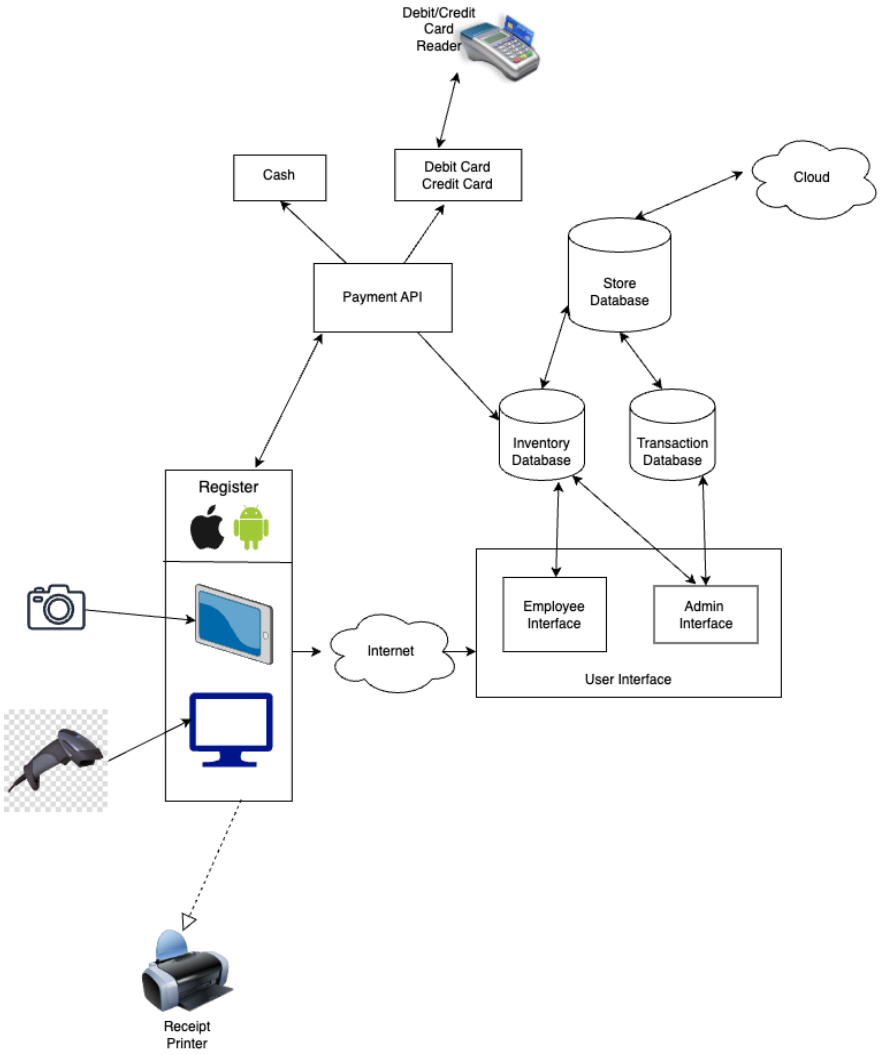
compare the entered itemID with the existing itemID's in the inventory and will return the item qualities such as size, color, stock, etc. If an itemID is entered that does not exist, the screen will display "item not found". If the item does exist within the system but the stock count is 0, the screen would display "out of stock".

System Testing

The employee or manager should be able to walk up to the kiosk and log in with their employee account. They are prompted to a screen where they can navigate to a screen where they can search for items in the inventory. They should be able to enter the itemID and click the search button where the screen will give them information about the stock of the item. If an item is not found, the system will display a message saying that the item is not found or that the item is not currently in stock. After the message is displayed, the employee is able to click a button that will return to the home screen.

Software Design 2.0

Architecture Diagram

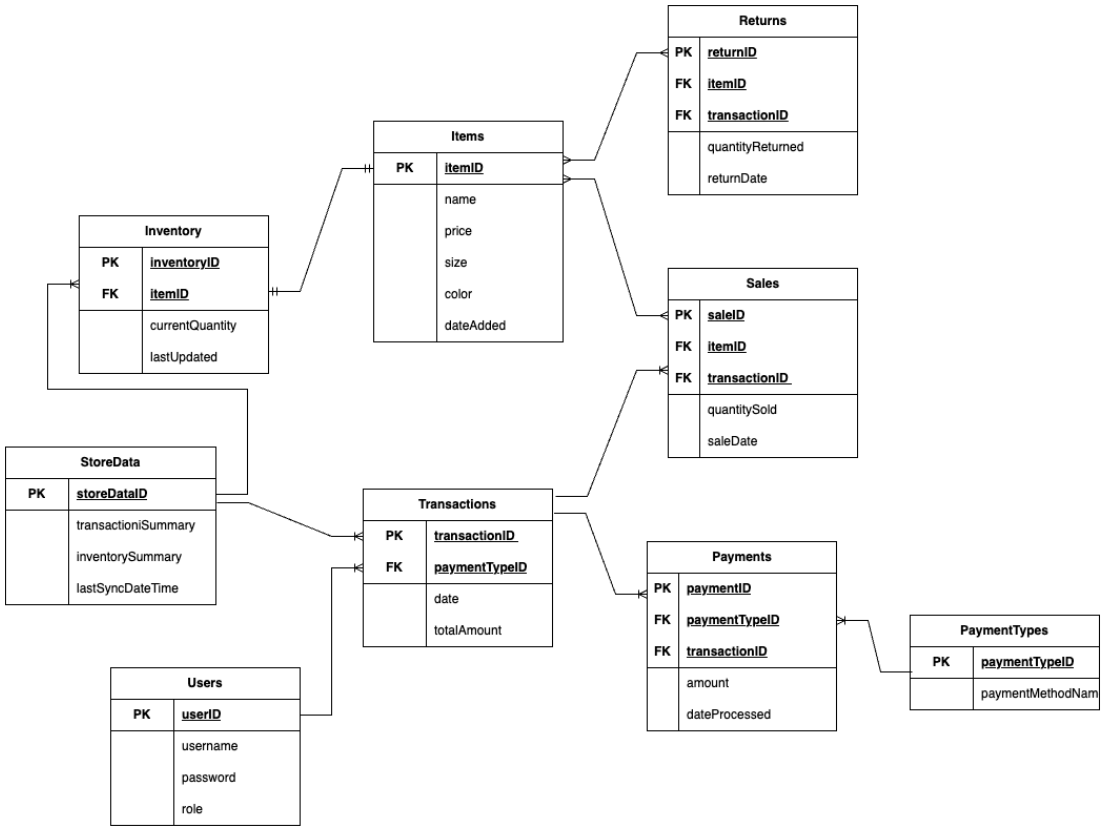


Explain why we are/aren't changing Architecture Diagram:

We are not going to change our software architecture diagram because we believe that our existing diagram contains all necessary information. In terms of data management, we decided to keep the number of databases the same because we wanted the Transaction Database to only be accessed by the admin and not the employee. There will be one large store database with all the inventory and transaction data that will be stored in the cloud.

Data Management Strategy

Entity Relationship Diagram:



Data Dictionary:

StoreData

Name	Description	Type	Length Max	Range of Values
storeDataID (PK)	Unique identifier for each store data record	Int		Auto-increment, Unique
transactionSummary	Summary details of recent transactions	Varchar	255	Varies
inventorySummary	Summary details of current inventory	Varchar	255	Varies
latestSyncDateTime	The last date and time of synchronization	DateTime		Date and Time format

Inventory

Name	Description	Type	Length Max	Range of Values
inventoryID (PK)	Unique identifier for each inventory record.	Int	-	Auto-increment, Unique
itemID (FK)	Links to the Items table	Int	-	Foreign Key
currentQuantity	Current stock level of the item	Int	-	Varies
lastUpdated	The last date and time the inventory was updated	DateTime	-	Date and Time format

Items

Name	Description	Type	Length Max	Range of Values
itemID (PK)	Unique identifier for each item	Int	-	Unique
name	Name of the item	Varchar	255	
price	Price of the item	Float	-	
size	Size classification of the item	Varchar	255	
color	Color of the item	Varchar	255	
dateAdded	Date item was first added to inventory	DateTime		Date and Time format

Returns

Name	Description	Type	Length Max	Range of Values
returnID (PK)	Unique identifier for a return	Int	-	Auto-increment, Unique
itemID (FK)	Foreign key linking to the	Int	-	Foreign Key

	Items table.			
transactionID (FK)	Foreign key linking to the Transactions table.	Int	-	Foreign Key
quantityReturned	Quantity of the item returned	Int	-	
returnDate	Date item was returned	DateTime		Date and Time format

Sales

Name	Description	Type	Length Max	Range of Values
saleID (PK)	Unique identifier for a sale	Int	-	Auto-increment, Unique
itemID (FK)	Foreign key linking to the Items table.	Int	-	Foreign Key
transactionID (FK)	Foreign key linking to the Transactions table.	String	-	Foreign Key
quantitySold	Quantity of the item sold	Int	-	
saleDate	Date item was sold	DateTime		Date and Time format

Users

Name	Description	Type	Length Max	Range of Values
userID (PK)	Unique identifier for a user	Int		Unique
username	Username of the user	Varchar	255	
password	Password of the user	Varchar	255	
role	The role of the user (e.g., admin, employee)	Varchar	255	

Transactions

Name	Description	Type	Length Max	Range of Values
transactionID (PK)	Unique identifier for a transaction	Int	-	Unique
paymentTypeID (FK)	Foreign key linking to the PaymentTypes table.	Int	-	Foreign Key
date	Date of the transaction	DateTime	-	Date and Time format
totalAmount	Total amount of the transaction	Float	-	

Payments

Name	Description	Type	Length Max	Range of Values
paymentID (PK)	Unique identifier for a payment	Int	-	Auto-increment, Unique
paymentTypeID (FK)	Foreign key linking to the PaymentTypes table.	Int	-	Foreign Key
transactionID (FK)	Foreign key linking to the Transaction table	String	-	Foreign Key
amount	The amount of the payment	Float	-	
dateProcessed	Date the payment was processed	DateTime	-	Date and Time format

Payment Types

Name	Description	Type	Length Max	Range of Values
paymentTypeID (PK)	Unique identifier for the type of payment	Int	-	Auto-increment, Unique
paymentMethodName	The name of the payment method	Varchar	255	

Data Management Description:

In our diagram, we have the main **StoreData** entity which contains the primary key storeDataID. This unique identification number helps differentiate between the different stores in each location. The entity creates the data about a given store which helps the employees track the stores sales, what items are selling and need to be restocked, as well as when the information was last updated. The information is then uploaded to the cloud database connected to other store locations.

The StoreData entity takes information from the **transactions** entity which contains a unique identification number for each different transaction that allows employees and customers to keep track of purchases and helps to set similar purchases apart from one another. The transactionID contains information about the date that the transaction took place on, as well as the total amount of said transaction. All this information is then uploaded and saved into the main server's system to track all sales. The paymentTypeID borrows information from the payments and payment types entities that detail the specific information about the payment.

The **Inventory** entity is also utilized by the StoreData which helps track the items that a store has in stock, and what they are out of. Some primary information that each item has is the inventoryID which allows the store as a whole to track the history of certain items to determine the stock of an item as well as the last time the information was updated. This information is important to track as it allows employees to see which items are out of stock and need to be replaced. Each item is then given an itemID, with specific information regarding each item from the items entity.

For the inventory to be properly tracked, it must borrow information from the **items** entity, which contains a unique itemID and assigns specific properties to each item such as size, color, name price, last time the item was updated, and the quantity of each item given its size and color. The items entity is trivial to the inventory class as it provides important information that allows workers to sort and store information about the items in an easily accessible way as well as being able to provide the information to customers.

The **Users** entity manages system access for employees at various levels. Each employee is assigned a unique userID, along with a username and password, enabling them to log into the system. This setup facilitates the execution of specific tasks aligned with the employee's role—such as transaction processing, inventory updates, and accessing sales reports. Roles are clearly defined (e.g., employee, admin) to ensure that employees can only access features necessary for their duties.

Given that payments are an important part of sales, the **payments** entity contains various information that tracks the payment's total amount and the date the payment was made and processed. This is done by giving each payment a unique numerical value stored as the paymentID. This entity also stores information about the payment type laid out in the paymentTypes entity and links the payment with a specific transactionID, making a connection between each payment and each transaction.

The **paymentTypes** entity contains information about each transaction's payment storing an int value for the payment type as well as a string value containing the method of payment called paymentMethodName. This information is valuable to the store as it helps the store track where payments are made from, whether it was made with cash, debit, or credit card as well as the name of the institution. The name of the institution will be of importance as it allows the store to keep track of payments and allows them information in the case of a payment dispute or their systems not working with a certain institution.

With the many, various transactions processed throughout the day for a store, the transactions draw information from the **sales** entity which contains primary information such as the saleID containing the quantity of sales made, as well as the date of those sales. The sales are tracked by linking them with each item's unique itemID, as well as providing information to the transactionID.

When customers wish to return items, the **returns** entity is utilized which creates a unique returnID that allows the system to track which unique items have been returned and prevents double returns. This entity also tracks how many items are made in a certain return exchange as well as the date the return took place. Each return also contains information about the item's unique ID as well as the transaction ID to determine which item was returned, and the initial transaction that took place.

Trade Off Discussion

In our system, we have chosen a SQL database framework over a noSQL option such as MongoDB, guided by our specific demands for solid transactional integrity and the ability to handle complex queries. SQL databases excel in maintaining data integrity and facilitating relationships between data elements, while providing native support for ACID-compliant transactions and an affinity for structured data. These features are essential for the transaction-heavy operations that our system is designed to handle.

While NoSQL databases offer greater flexibility and are adept at handling large volumes of unstructured data, our system does not anticipate the need for such scalability in the foreseeable future. The structured nature of our data, the necessity for detailed queries, and the absence of large-scale changes in data volume steered us towards SQL.

Our design choice also extends to using multiple databases to logically separate concerns—such as separating transaction processing from inventory management—enhancing maintainability and clarity. The ERD and data dictionary serve as integral tools in this design, enabling a clear visualization of data relationships and providing a detailed description of table contents, which complements the structured nature of SQL databases.