Clothing Store Point of Sale System

Tiffany Truc
Danny Kim

CS-250-1002

System Description

The Clothing Store Point of Sale System allows companies to organize store inventory alongside the customer transaction history. The point of sale system is designed to make inventory management more efficient by automatically updating the item inventory given purchases and returns. The item inventory system is searchable given an item's unique ID, price, size, color, and quantity. By working with external payment processors, employees are able to accept debit, credit, and cash payments with purchase totals being automatically calculated. Transaction history and sales numbers are stored separately in a secure cloud-based database which can only be accessed by administrative employees. This system is also supported by iOS and Android operating systems, and utilizes phones, tablets, and barcode scanners to connect to the shared item inventory.

Software Architecture Overview

Architectural Diagram Description

Clothing Store POS System

User Interface – The design implemented to make the interface intuitive and simple to use.

Credential System – User login system that determines if the user is an administrative user or regular user.

USE CASE – A bundle of features implemented into the POS application that all connect to one another. In this the Purchase/Return, Employee Tracking, Inventory Management, and General Transaction Report systems will all communicate with each other in some way.

- **Purchase/Return System** – Implementation for the process of purchasing/returning clothing store products from/to the store.

- **Employee Tracking System** – A system that will allow employees to clock in/clock out and keep track of their general actions on the POS system.

- **Inventory Management System** – Implementation of managing inventory in various manners ranging from unique ID to the variables that consists of clothing item or other purchasable items from the clothing store. It allows the user to add/remove or customize various inventory items.

- **General Transaction Report System** – A log that records all transactions that were made within the system.

External Device Compatibility System – An abstract block of compatibility features required to be able to allow access to the POS system by various external hardware.

- Barcode Scanner
- Receipt Printer
- Keyboard and Mouse
- Tablet/PC
- Payment System

Point of Sale API – A storage of data that holds all manner of information regarding the product.

Credit/Debit API – A storage of all data involving payment methods such as credit cards, debit cards, and the name of the holders.

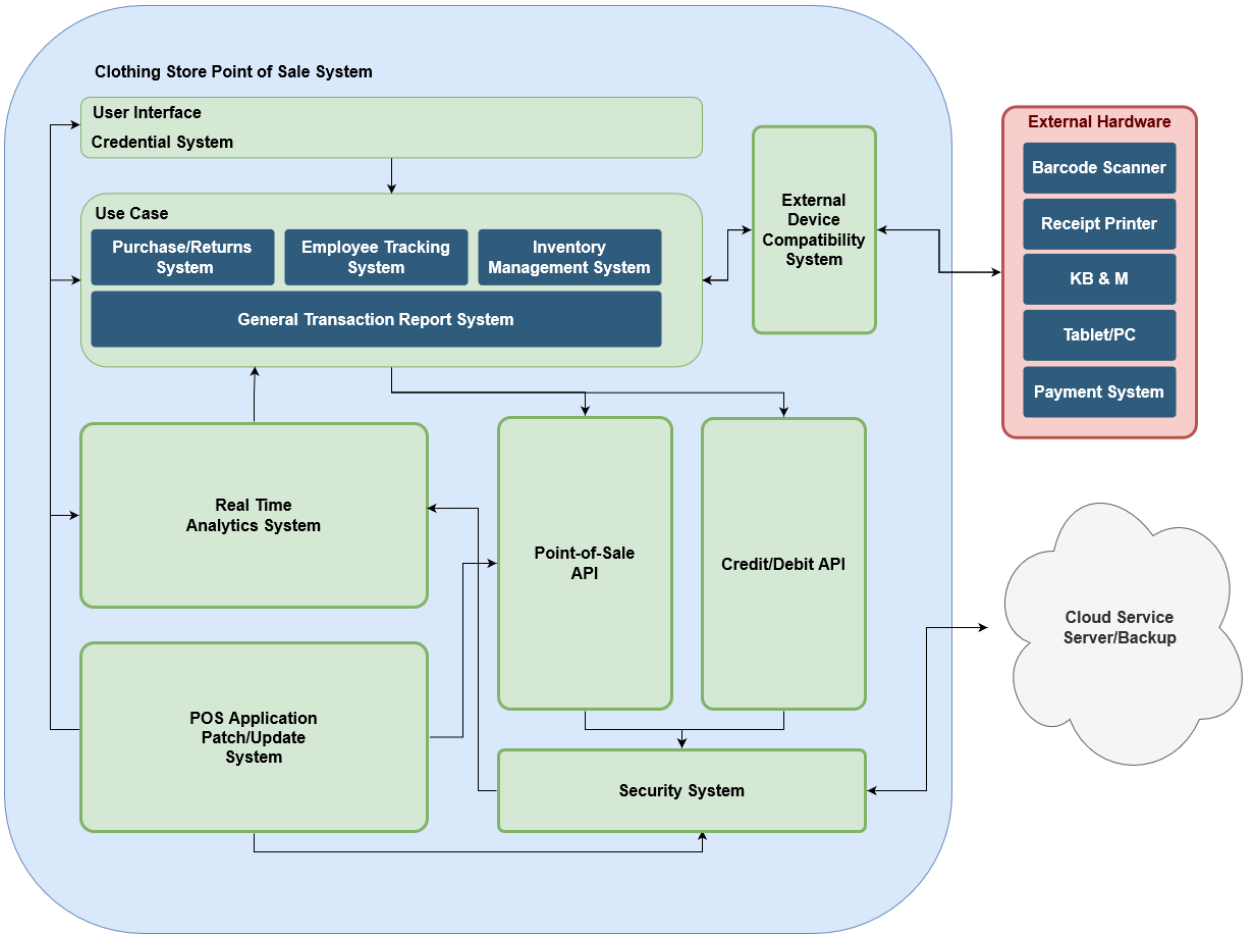
Security System – A system made to combat hacking of store information regarding their products and payment information.

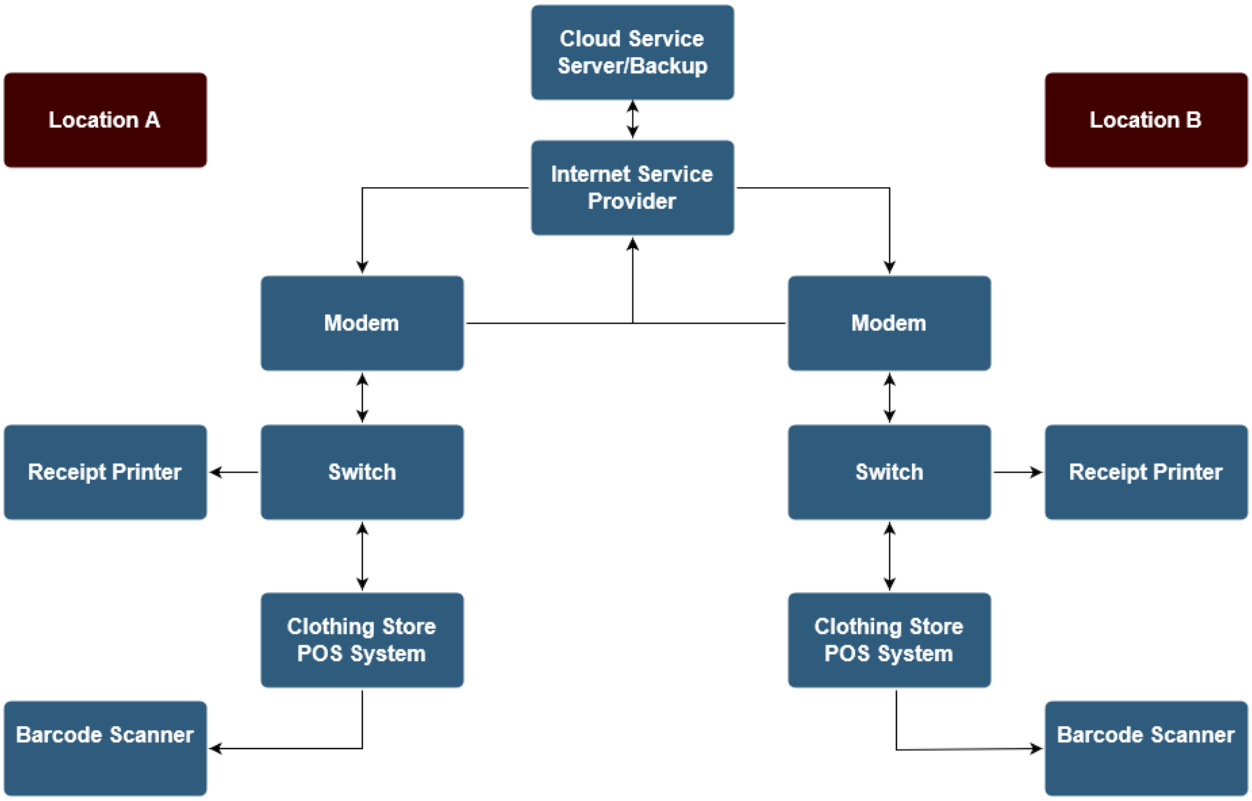
Cloud Service Server/BackUp – A cloud implementation that automatically updates and backs up data involving anything that happens in the POS system.

Real Time Analytics System – The part of the POS system that calculates any mathematical functions instantly. It will calculate net costs for purchases which includes tax and update inventory after every purchase/returns. Another example would be the ability to calculate the total time an employee has worked in that week at the clothing store.

POS Application Patch/Update System – The system that allows for the POS application to be updated through a series of patches after the initial release into the market.

Original Software Architecture Diagram





UPDATED UML Diagram and Descriptions

The Item class deals with in store items and their associated characteristics, the class itself has eight public attributes. These attributes include idNumber, itemName, price, quantity, size, color, dateAdded, and itemType. The attributes itemName, size, color, and itemType are strings. The idNumber, quantity, and dateAdded are ints, and the price is a float. These attributes help categorize the items in the Clothing Point of Sale System, and are connected to the inventory, and payment classes. The employee and administrator classes are connected to this class, as employees and administrators can edit, add, and remove items. The operations in the item class include buyItem(), returnItem(), addItem(), and removeItem(). These operations all return as void.

The Administrative User class deals with management and employees granted administrative privileges. This class has four private attributes: name, adminID, password, and email. Password, name and email is stored as a string, and adminID is stored as an int. The Administrative User class has three operations: viewTransactions(), viewSales(), and viewItems(). These operations all return void, and allow administrative users to access the transaction history, sales numbers, and item inventory databases. Only administrative users are allowed the permission to access these databases.

The Employee class deals with employee login information, and the ability to add, remove, and edit item information. The Employee class has four private attributes: name, staffID, password and email. The attributes name, password, and email are strings, and the staffID is stored as an int. The operations of the Employee class include: viewItem(), searchItem(), addItem(), removeItem(), assignAttributes(), clockIn(), and clockOut(). These operations are used to check in and check out employees, view items, and assign item characteristics, edit items, and add and remove items. These operations all return as void.

The Payment class deals with customer transactions, and recording the payment, as well as updating the inventory, and automatically getting the total for the transaction. The Payment class has six private attributes: amount, saleDate, type, saleTime, paymentInfo, and transactionID. The attributes saleDate, saleTime, and transactionID are ints, while type is a string, and amount is a float. The operations of the Payment class include getTotal(), setTotal(float), setPaymentInfo(int creditNumber), getPayment(), and readBarCode(int idNumber). All of the operations return as void except for getTotal() which returns as a float and getPayment() which returns an int. The function readBarCode takes in the item idNumber to input into the system and add to the customer's transaction. The operations automatically update the item inventory, sales numbers, and transaction history after every customer transaction. Returns are included under customer transactions. Furthermore, the operations automatically get the total of the transaction, including sales tax. The readBarCode operation receives the item's idNumber as a parameter, and returns void.

The Transaction History class has records detailing each customer transaction given their payment type, store location, cashier information, name, sale time and date, and transaction ID. eight private attributes. The attributes are the name, saleDate, saleTime, customerPaymentType, customerCardInfo, staffID, storeID, and transactionID. The attributes saleDate, saleTime, staffID, storeID, transactionID, and customerCardInfo are all ints. The name and customerPaymentType are strings. The operations include getTransactionInfo(), editTransaction(int transactionID), getPaymentInfo(), and updateTransactionHistory() which all return as void, except for getTransactionInfo() which returns as a string, and getPaymentInfo() which returns as an int. The operations receive transaction information, edit the transaction attributes, and update the transaction history anytime an item is bought or returned.

The Sales Number class records the sales per store given the transaction history of the store, and is accessible by administrative users. The class contains three public attributes, and three private attributes. The public attributes include quantity, idNumber, and itemName, and the private attributes include saleDate, saleTime, and storeID. All of the attributes are ints except for itemName, which is a string. The operations are: getSalesInfo(), updateSales(), setSalesQuantity(int), setSalesName(string), and setSalesIdNumber(int). These operations return as void, except for getSalesInfo() which returns as a string. The operations retrieve the sale

analytics, updates sale analytics given each purchase and return, and removes/adds sales given a purchase or return.

The Inventory class deals with the inventory of items per store location, and the characteristics of items, alongside operations making the inventory system more efficient, organized, and searchable for employees and management. The public attributes include itemNumber, quantity, itemName, itemAttributes, dateAdded, and itemType. The private attribute is the storeID. The itemNumber, quantity, storeID, and dateAdded are all ints. The itemType, itemAttributes and itemName are strings. The operations include getInventoryInfo(), updateInventory(), setItemQuantity(int), setItemName(string), and setItemAttributes(string). These operations all return as void, except for getInventoryInfo() which returns as a string. The operations allow employees and management to get the inventory details, update the item inventory, and remove and add items to the inventory.

There is a Cash class, which is utilized in the Payment class. There are no attributes, and there are no operations for this class.

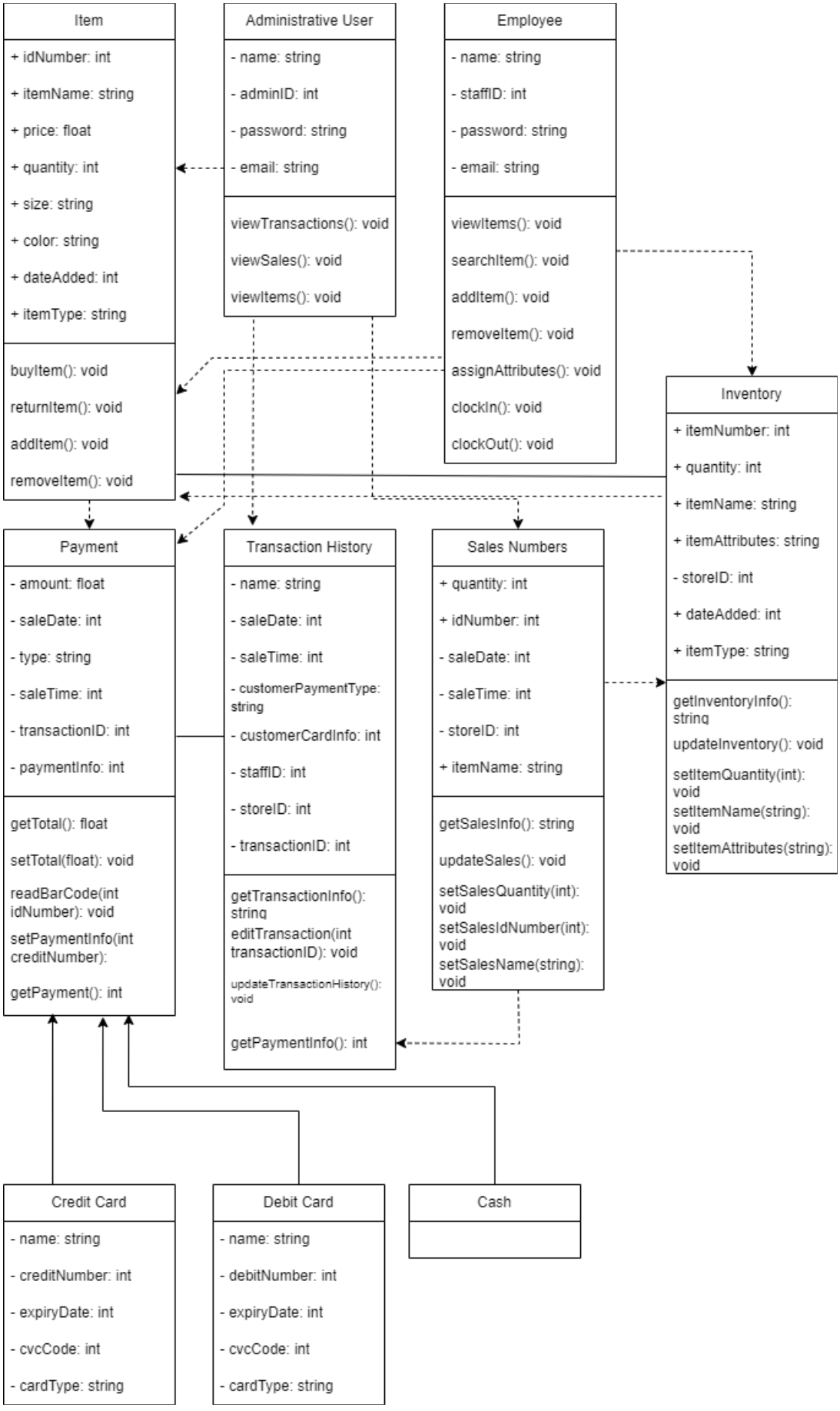
The Credit Card class stores customer information, specifically their credit card information. There are five private attributes to this class: name, creditNumber, expiryDate, cvcCode, and cardType. The name and cardType are stored as strings, with the creditNumber, expiryDate and cvcCode stored as ints. There are no operations in this class.

The Debit Card class stores the customer debit card information when used in a transaction. There are five private attributes to this class: name, debitNumber, expiryDate, cvcCode, and cardType. The name and cardType are stored as strings, with the debitNumber, expiryDate and cvcCode stored as ints. There are no operations in this class.

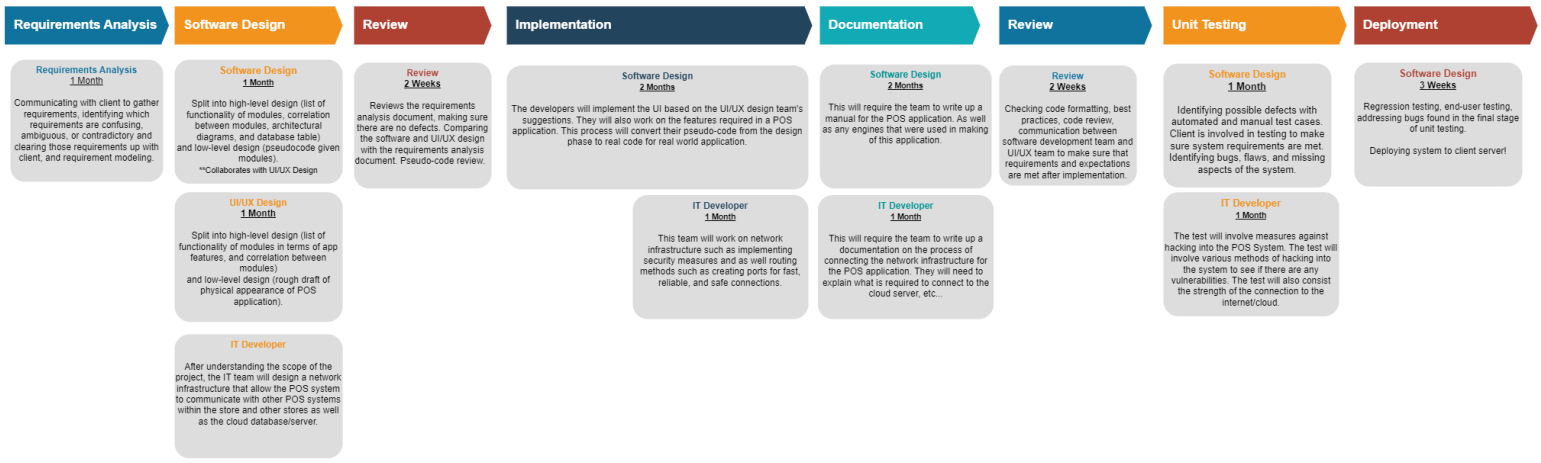
Explanation of Modifications:

In the Payment class, getPaymentInfo() was added to the operations, and will retrieve the payment attributes and characteristics, such as the saleTime and saleDate, getPayment() was added, which returns the customer's card information, and setPaymentInfo(int creditNumber) was also added. The operations: updateInventory() was moved to the Inventory class, updateTransactionHistory() was moved to the Transaction History class, and updateSales() was moved to the Sales Number class to automatically update their respective systems alongside a purchase or refund being made. In the Transaction class, returnItem(int idNumber) was removed, and instead replaced with editTransaction(int transactionID), which would allow the user to edit the status of the transaction in cases where the customer returns an item. The operation getTransactionInfo() was added, and will return as a string of the transaction attributes. In the Inventory class, searchInventory() was replaced with getInventoryInfo() for clarification, and clarity, as getInventoryInfo() would retrieve the attributes of the item inventory, such as the

name of an item, quantity, and the unique item ID of the respective item. The operations setItemQuantity(int), setItemName(string), and setItemAttributes(string) have been added to set the characteristics of the item in inventory. The Sales Number class had the said operations added: setSalesQuantity(int), to set the quantity of sales per store given purchases or returns by a customer, setSalesIdNumber(int), to input the ID Number of the set of sales of a store given a date, and setSalesName(string) to set the item name under sales analytics, and getSalesInfo() which returns as a string of sales attributes per store. A setter was also added to the Payment class, setTotal(float) ,which sets the total of the transaction and returns as void, and the paymentInfo variable was also added to the Payment class.


Item
+ idNumber: int
+ itemName: string
+ price: float
+ quantity: int
+ size: string
+ color: string
+ dateAdded: int
+ itemType: string
buyItem(): void
returnItem(): void
addItem(): void
removeItem(): void
Administrative User
- name: string
- adminID: int
- password: string
- email: string
viewTransactions(): void
viewSales(): void
viewItems(): void
Employee
- name: string
- staffID: int
- password: string
- email: string
viewItems(): void
searchItem(): void
addItem(): void
removeItem(): void
assignAttributes(): void
clockIn(): void
clockOut(): void
Inventory
+ itemNumber: int
+ quantity: int
+ itemName: string
+ itemAttributes: string
- storeID: int
+ dateAdded: int
+ itemType: string
getInventoryInfo(): string
updateInventory(): void
setItemQuantity(int): void
setItemName(string): void
setItemAttributes(string): void
Payment
- amount: float
- saleDate: int
- type: string
- saleTime: int
- transactionID: int
- paymentInfo: int
getTotal(): float
setTotal(float): void
readBarCode(int idNumber): void
setPaymentInfo(int creditNumber):
getPayment(): int
Transaction History
- name: string
- saleDate: int
- saleTime: int
- customerPaymentType: string
- customerCardInfo: int
- staffID: int
- storeID: int
- transactionID: int
getTransactionInfo(): string
editTransaction(int transactionID): void
updateTransactionHistory(): void
getPaymentInfo(): int
Sales Numbers
+ quantity: int
+ idNumber: int
- saleDate: int
- saleTime: int
- storeID: int
+ itemName: string
getSalesInfo(): string
updateSales(): void
setSalesQuantity(int): void
setSalesIdNumber(int): void
setSalesName(string): void
Credit Card
- name: string
- creditNumber: int
- expiryDate: int
- cvcCode: int
- cardType: string
Debit Card
- name: string
- debitNumber: int
- expiryDate: int
- cvcCode: int
- cardType: string
Cash

Software Development Timeline



Test Plan

Unit Tests

Test Case: UnitTest 1
Test Name: Payment readBarCode() Unit Test
Description: The function readBarCode() directly works with a barcode scanner, which displays a beam of light across a barcode and measures the amount of light and the pattern of light reflected back. This reflected light is then converted into data by the function through decoding, and the data is used to efficiently read in the item's unique barcode ID. This function utilizes the Universal Product Code (UPC), which is the standard barcode symbology for multiple countries. The UPC consists of 12 digits, with one check digit, and 11 data digits. This unit test seeks to test if the method properly reads in the expected digits.
Steps: The user would either input the barcode ID manually, or use the BarCode scanner to input the barcode ID into the function. This results in readBarCode(554149166525) being called, which should return the item ID (554149166525).
Input: 554149166525 -> readBarCode(554149166525)
Expected Output: 554149166525
Actual Output: 554149166525
Pass/Fail: Pass

Test Case: UnitTest 2
Test Name: Payment readBarCode() Unit Test
Description: The function readBarCode() directly works with a barcode scanner, which displays

a beam of light across a barcode and measures the amount of light and the pattern of light reflected back. This reflected light is then converted into data by the function through decoding, and the data is used to efficiently read in the item's unique barcode ID. This function utilizes the Universal Product Code (UPC), which is the standard barcode symbology for multiple countries. The UPC consists of 12 digits, with one check digit, and 11 data digits. This unit test seeks to test exception handling in the case of a 13 digit barcode ID.
Steps: The user would either input the barcode ID manually, or use the BarCode scanner to input the barcode ID into the function. In the case of a barcode ID != 12, the method would throw an exception that would output "Error: Maximum digits are 12." for a barcodeID > 12, and "Error: Minimum digits are 12." for a barcodeID < 12.
Input: 1228411015533
Expected Output: Error: Maximum digits are 12.
Actual Output: Error: Maximum digits are 12.
Pass/Fail: Fail

Test Case: UnitTest 3
Test Name: SalesNumbers setSalesIdNumber() Unit Test
Description: The setSalesIdNumber() function seeks to pair the sale analytics of a store to a unique ID searchable throughout the system by an administrative user. That is, if a store makes a quantity of sales in a day, it will be paired with a unique sales ID number that's searchable within the store's system. This unique sales ID will be 9 numbers long. A sales ID less than or greater than 9 numbers will result in an exception being thrown.
Input: storeID.setSalesIDNumber(736369041)
Expected Output: 736369041
Actual Output: 736369041
Pass/Fail: Pass

Input: storeID.setSalesIDNumber(02117348)
Expected Output: 02117348
Actual Output: Error: Unique Sale ID requires 9 numbers.
Pass/Fail: Fail

<u>Integration Tests</u>

Test Case: IntegrationTest 1
Test Name: SalesNumber getSalesInfo() Integration Test
Description: Testing the getSalesInfo() function will ensure that administrative users are able to access the sales analytics for their respective stores in the system. The method retrieves the information from each store's sales analytics, containing vital information such as an item's unique ID, the quantity of said items sold, and the item's Name. This information is used when

deciding whether to or whether to not order more stock of an item given the demand from customers. This information is also used to see whether a store location is thriving in business, or vice versa given the quantity of sales per location. This function is located in the Sales Number class, can be accessed from the Administrative User class, and accesses the Transaction History class and Inventory class. The Sales Number class accesses the Inventory class to update the inventory stock with the quantity of items sold, and uses the Transaction History class to know the quantity of items sold, alongside which items are being sold.
Steps: Using the setter function, set each variable to a desired value. Create a tester class with SalesNumber object x. Using this object, call the following functions with a test set of inputs: x.setSalesName("Slacks"), x.setSalesQuantity(27), x.setSalesIdNumber(0123456). After calling these setters, the result should be printed to x.getSalesInfo(), which returns as a string containing the resulting sales attributes.
Input(s): x.setSalesName("Slacks"), x.setSalesQuantity(27), x.setSalesIdNumber(0123456)
Expected Result(s): The result of calling the getSalesInfo() function should return a string that outputs: "Item Name: Slacks, Item Quantity Sold: 27, Sales ID: 0123456"
Actual Result(s): "Item Name: Slacks, Item Quantity Sold: 27, Sales ID: 0123456"
Pass/Fail: Pass

Test Case: IntegrationTest 2
Test Name: Payment getPayment() Integration Test
Description: Testing the Payment class' getPayment() function will check if the customer's transaction information is stored properly within the Transaction History class. This function references the Credit Card, Debit Card, or Cash class. By using customer information within the aforementioned classes, it stores thecustomerPaymentType variable, and the customerCardInfo variable in the Transaction History class. This method is important as refunds through credit and debit cards are processed through the customer's card information that's stored in the Transaction History class.
Steps: Using the setter functions in the Payment class, the paymentInfo variable will contain the customer's credit card or debit card information. We'll be testing if the input passed onto the setter class in the Payment class will properly transfer over into the Transaction class' getter class. Our first step is to call setPaymentInfo(3300815134028233). This is then printed into transactionHistory.getPaymentInfo() as an int of the customer's 16-digit credit card number.
Input(s): 3300815134028233 is inputted into setPaymentInfo(3300815134028233)
Expected Output(s): 3300815134028233
Actual Output(s): 3300815134028233
Pass/Fail: Pass

<u>System Tests</u>

Test Case: SystemTest 1

Test Name: GUI functional testing.

Description: Graphical User Interface (GUI) testing. Doing a thorough examination of all functional aspects of the interface such as buttons, text boxes, list boxes, menus, and toolbars.
Steps:

1. Clicking on every button that is available to observe the behavior of the application.

2. Checking all boxes that can readily accept inputs from the user.

3. Clicking on menus and toolbars to see if it opens the correct page or bar as intended.

4. After reviewing the results and recording it, go back to step 1 and repeat the process multiple times to check for consistency.

Test Data: Clicking through all available buttons using mouse & keyboard and touch screen responsiveness. Typing in words, numbers, symbols, and other inputs available on keyboard/tablets/smartphones.

Expected Results: All buttons are responsive through every hardware that will be utilizing this application. Typing into the software is responsive and outputs correctly when inputted from the user. Moving through this application utilizing the menu, and other features in the toolbar are responsive.

Actual Results: All buttons were responsive, and inputting various text, symbols, and other keyboard inputs were outputting correctly. Clicking through the menu and toolbars worked as intended.

Pass/Fail: Pass

Remarks: Due to the POS system being partly reliant on a stable internet connection, I found that some elements on the menu/toolbar did not work if the system was not connected to the internet. Otherwise, everything worked just as intended when a reliant internet connection was present.

Test Case: SystemTest2

Test Name: GUI load testing.

Description: Doing a thorough examination of graphical aspects of the interfaces such as icons, logos, colors, and page layouts are loaded properly.
Steps:

1. Refer to the Design Team's documentation for the proper layout of the GUI.

2. Install the software on all types of hardware.

    a. PC (1080p, 1440p, 4k)

    b. Phones/Tablets (Various resolution from vertical/horizontal POV)

    c. POS hardware with cash register implementation. (Various Resolutions)

3. Load into the application and verify that all icons, logos, colors, and layouts are loading properly.

4. Input data into list boxes and text boxes of the POS system to check if the list is outputting correctly when adding new data into it.

5. Check if uploading a logo from the user affects the output of the page layout.

6. Restart the application and repeat the process to be certain of consistency.

Test Data: Typing in words, symbols, and other inputs available from keyboards/tablets/phones. Utilizing different logos with differing dimensions to upload for testing purposes. Testing various resolutions from various devices.

Expected Results: All inputs from any hardware will not change the layout of the GUI. All resolutions from various devices are outputting all elements in the GUI properly.

Actual Result: There were unintended stretching in the logos and icons when utilizing 1440p or 4k resolution screens. All colors were implemented correctly. The page layouts were displayed correctly on almost all devices. There seemed to be inconsistent loading of the page layout on mobile devices such as tablets/phones when switching from horizontal to vertical view as well the opposite way.

Pass/Fail: Fail

Remarks: The unintended stretching comes from uploading logos and icons that the client requested. Testing a logo made in 1080p resolution and uploading it to a system that is utilizing 4k resolution resulted in some stretching of the image. Therefore, the POS system requires some image scaling technology to compensate for the various number of variables in resolution, or the client must provide the hardware specs before we upload the correct image resolution to the software.

Test Case: SystemTest3

Test Name: Installation and Compatibility Testing

Description: Testing the POS application compatibility for all operating systems and hardware.

Steps:

1. Create an environment to replicate all hardware that is required to operate a clothing store business.
    a. PC/Tablet/Phone (Windows/Android/MacOS/IOS)
    b. Cash register
    c. Receipt Printer
    d. Scanner
    e. Printer
2. Install the POS application onto the hardware.
3. Verify that the installation process is working as intended.
4. Verify after installation that the application is working as intended.
5. Uninstall application to verify that the uninstallation process is working as intended.

Test Data: Using all the various hardware that replicates a clothing store environment. Each hardware has its own operating system. Using different types of receipt printers/printers/scanners/cash registers available and frequently utilized in the clothing store business.

Expected Result: The application is compatible with all various OS, and hardware listed in the testing.

Actual Result: The POS system is compatible with the various OS listed in the testing. The installation/uninstallation worked as intended on all the compatible devices. Unfortunately, there were some hardware such as scanners/cash registers/printers that did not work with our POS system. These devices were recorded and logged in a separate file for review.

Pass/Fail: Fail

Remarks: There were some compatibility issues that should be fixed with a patch. Once this POS application is out in the market, I am worried that new hardware might have compatibility issues as well. It would be wise to allow an avenue for patches to future hardware that need to work with our software.

Test Case: SystemTest4

Test Name: Payment Exception Handling

Description: Testing exception handling if an unexpected error occurs when processing a payment from a customer in a clothing store. Such as, if their credit/debit card does not have any money, or if there is a system error where the card was not read correctly, the pos system should alert the user that the payment did not go through and display the correct error message. The system then should process the incorrect transaction and recover accordingly to be able to process payment again. If the payment does go through, the test will check if an exception will be thrown if there is any corruption of data when updating transactional and sales data. This can occur due to hardware malfunctioning, network error, or unknown circumstances.

Steps:

1. Utilize various payment methods that can be processed by the POS system.

2. Test various inputs such as negative balance in the credit/debit card, hardware issues where the credit/debit card reader does not scan properly or anything with human error such as inserting a card that is not a form of payment such as library cards, etc.…

3. Observe the POS application behavior when the payment process does not go through.

4. Observe the POS application when the payment process does go through.

5. Record the observation and repeat the test and make sure the exception handling is working as intended.

6. Check if the system is recovering properly after an exception handling is set off.

7. Observe the POS application transaction history update.

8. Emulate a test to see if an exception is thrown if the customer data from the transaction is updated incorrectly due to corruption of data.

9. Observe the POS application sales data update.

10. Emulate a test to see if an exception is thrown if the sales data from the transaction is updated incorrectly due to corruption of data.

Test Data: Different dummy credit/debit cards with various amounts of money. Which includes negative balance, or a small balance which can only pay a portion of the net amount required for the transaction. Emulation of credit/debit card readers that can duplicate common errors that occur regularly. Emulating network error, hardware malfunction, and any other creative ideas of causing corruption in the system.

Expected Result: During the POS transaction process, the exception handling should alert the user if there were any errors that occurred during the payment process.

Actual Result: Exception Error was thrown when expected and the application recovered accordingly well. Through the various emulations, whenever any sort of corruption of data occurred the software alerted accordingly and reverted any changes that might negatively affect the data set. The only issue that occurred was when we emulated a power outage during the payment process. When restarting the application there was some corrupted data present, and no exception handling was thrown.

Pass/Fail: Pass

Remarks: It would be wise to alert the client that in an event of a power outage that they will have to revert any changes that occurred during the outage manually.

Test Case: SystemTest 5

Test Name: POS Application Volume Testing

Description: Testing the system performance when it must process an enormous amount of data. For a clothing store POS system, it will require processing a long list of information on products which include ID, attributes, and pricing, as well as customer data which includes payment info, customer info, and sales data. The data set is uploaded and downloaded from the cloud system and therefore this test observes and verifies the capabilities of this POS system when handling various amounts of data.

Steps:

1. Create a data set of various amounts.
2. Upload a data set into the POS system.
3. Check the performance of the system when the data set is entered.

4. Check if the POS system outputs at the required speeds when requesting data.
5. Check if the POS system is working at the required speeds when inputting new data.
6. Check if the POS system is deleting at the required speeds when erasing data.
7. Check if the POS system is optimized well enough to upload to the cloud at required speeds.
8. Repeat the test by entering a new data set with a different size of data.

Test Data: A various amount of data sets that include product/customer/sales info. The data sets range from a small amount to an extremely large amount.

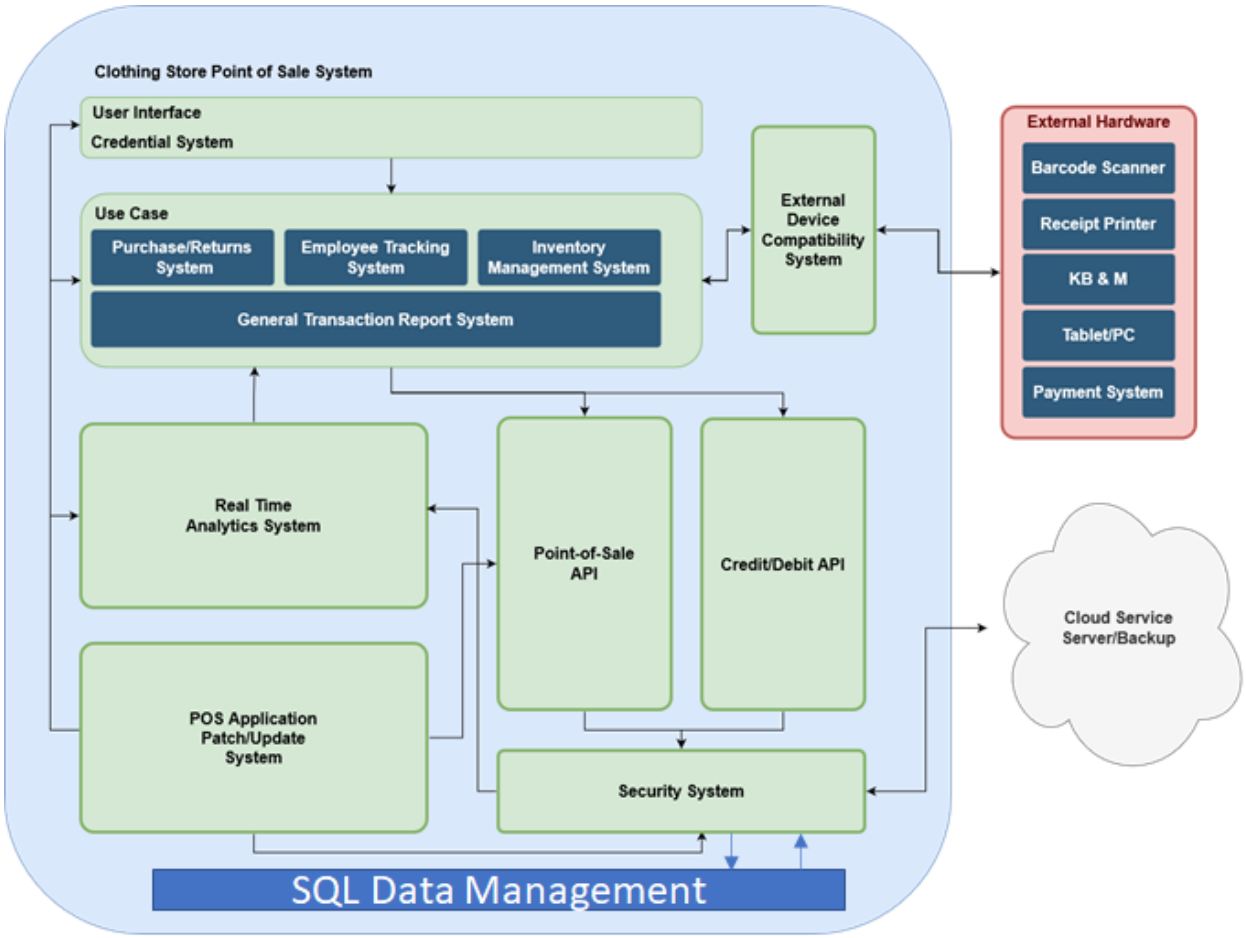
Expected Results: As we test the various amounts of data set, the POS application should be able to handle large amounts of volume that are expected for a clothing store.

Actual Result: During the volume testing, the application was able to handle the required data sets to operate a clothing store franchise. The system performance on various features our POS system offers ran accordingly to the required speeds. When we tested an excessive amount of data is when we observed performance slowing.

Pass/Fail: Pass

Remarks: Due to the POS system requiring only string/int/float data types, it would be extremely difficult and require an enormous amount of time to input the amount that would cause a system performance decrease.

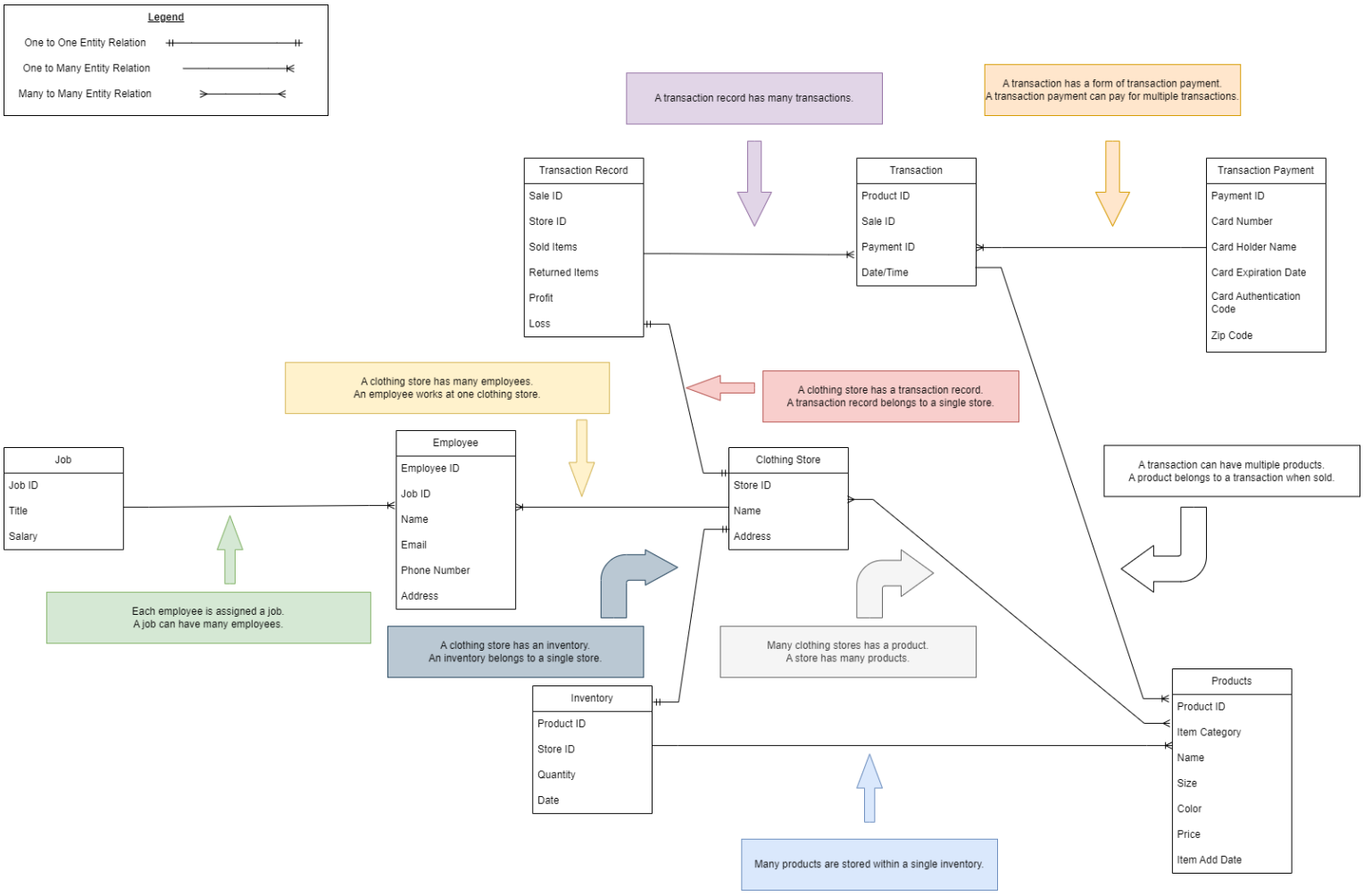
UPDATED Software Architecture Design



Explanation of Modifications

One database has been added to the updated architecture diagram—the SQL Data Management database. This database is to make up for the lack of data management in the APIs and systems included in the architecture diagram. An API isn't a data storage method, and neither are the systems listed on the original architecture diagram, hence us adding the SQL Data Management to the updated version of the architecture diagram.

SQL Data Management Strategy



Explanation of Architecture Diagram/SQL Database Diagram Logic

We decided to utilize only one SQL database for our POS system because it is necessary for all information to be contained in one database. Having one database will make searching for information, and the application of said information efficient and easier. This POS application is going to be sold to clients that run multi-chain clothing stores and will require it to be able to communicate between all the stores. Having a single database ensures that all the client's POS applications will contain all the information they require to make sure their stores are sharing information correctly. It also adds benefits such as "store A" not having something in stock but being able to check inventory in "store B" to give options to the client's customers to buy.

The design layout of our database was organized to ensure that there were minimal issues of collision when our client's employees start using our POS application. For example, we

separated data for “products” apart from “inventory” to ensure that when the real-time analytics start working to calculate the number of products in an inventory, it will not collide when clients add new products to the database. We followed this design philosophy throughout our entire database. We also designed our “employee” and “job” sets for a secure database platform that will ensure limited access and authorization depending on the job of the user that is utilizing the POS system.

Trade-Offs Between SQL and NoSQL

We ultimately decided on SQL based on the ACID properties of SQL, and the security that using SQL brings compared to NoSQL. The ACID properties of SQL mean that there won’t be “bad data,” and in the case that our database crashes, there won’t be data corruption. Due to our business application being an inventory/accounting/POS system, it’s ultimately suited for an SQL-based system because of the consistency and security of data. It’s *extremely* important that we prioritize security, especially given that the database contains sensitive customer payment information and confidential transactional data. The scalability of NoSQL isn’t required for our POS System because SQL provides enough space for a clothing store. In the case of other systems that require scalability, NoSQL is cheaper and better for cloud-based systems with its horizontal scaling. In terms of scalability for SQL, SQL is more expensive as it requires additional physical hardware to be added, as it scales vertically. The POS system is focused on a clothing store, so we have a well-structured data model that SQL databases can efficiently utilize. A NoSQL database might be suited for data sets with a model in advance, but this isn’t the case for us.

Possible Alternatives

A possible alternative for organization could be having a separate SQL database for each individual store. However, having a separate database for each store would make access authorization, updating, and query searching less efficient, and more troublesome than having a single database. Furthermore, a connection to a server can only access the data in the database specified in the connection request. This means that the client would have to reconnect to their desired database separately. An alternative for grouping would be having Employee and Job, Transaction Record, Transaction, and Transaction Payment, and Products and Inventory together. However, this grouping could potentially result in more instances of collision. Due to the inefficiency of the alternatives listed above, we have decided to not go with these instances.