GetThisBread

<u>Project Team Members:</u>

Richard Abrams

Ronald Abrams

Emily Ashburn

Michael Calvo

Jason Manning

Ian Wilhelmsen

Table of Contents

1. Project Definition

The GetThisBread project is needed in the industry by small businesses like diners, beauty salons, web-only storefronts, and whole hosts of businesses to direct the workflows of their employees and keep track of stock, product, and profit.
This goal is achieved by creating subsystems, featuring Point of Sales (POS), Timesheet Management System (TMS), Business Intelligence Analysis (BIA), Inventory Management (IM), all supported by a robust API which will all utilize Angular, Bootstrap, PHP and MySQL.

2. Project Requirements

Functional

The application has the following functional requirements: Be readily available for many different types of end-users of a variety of roles managed by Google's SSO. The application supports a Timecard Management System (TMS). The TMS supports features such as allowing employees to clock in and out of their appropriate jobs. The TMS also supports the scheduling of employees by role at a location and supports the viewing of schedules. The application supports a Point of Sales System (POS). The POS allows orders to be placed and tracking transactions. By tracking transactions, the POS subsystem keeps information on both credit transactions handled by a third party and cash transactions. The POS supports ticket/order viewing. The POS subsystem also supports the end-of-business day processing. Lastly, the application acts as an Inventory Management System (IMS). The IMS keeps track of incoming stock, items sold, inventory audits, creation/deletion of items, recipe costing, and estimation of utilization.

Usability

The user interface uses a web browser (preferably chrome). The team makes use of javascript libraries and UI libraries to render action and information panels to assist in point of sale operations, time scheduling, inventory management, and business analytics. The application accomplishes this by using many views with similar formatting separating key functionality of the application with clear chaining that is driving a user to complete their workflow with the least amount of scrolling, and searching and scanning the screen for their objective.

Performance

This application deals with real-time inventory, finances, and cash-specific transactions. The demand for these services is not 24hrs for all users at the same time. While the application must be accurate all the time it is in use, the application will have downtime for certain users and, at worst, expected outages of service can be arranged. The application must be accurate every time for its POS functions but there can be grace periods when dealing with the inventory or TMS functionality. Errors and continual rollouts for business-specific functionality for the TMS and IMS can be addressed with less urgency than an outage of order taking at a restaurant. Time management changes or inventory audits can be managed with pen and paper or attempted later after a hotfix.

System

Hardware Specifications

Principles would be interested in hosting the application on a VM on either a cloud or through a self-hosted farm. An MS Server 2016 or 2019 hosting a farm on Hyper-V or a centOS hosting VMWare or spinning individual VMs up via AWS or Red Hat would be enough to host the web server and DBMS given that a 100GB drive, i5 Intel 8th, a standard nic and 16 GB of SO-DIMM DDR4. The physical requirements are the equivalent of a mid-level NUC or any other brand name entry-level mini-workstation. Image snapshots, Database backups, and event logs should be stored in another location. These low-end solutions should last 5 years speculatively.

Software

The application system level can be hosted on MS server 2016 or better, CentOS 7 or Ubuntu Server 20.4. The application's web server should match the preference of the client's choice of an operating system. IIS should be used for MS server installations as LAMP or Apache should be used for Linux installations.
Database
This application was built using MySQL. MySQL is preferred because it excels at online transactions out of the box on the community level. When more transaction capacity is required and the need for analytics to be separated from the business domain and operate

on a separate server; additional features available from other Enterprise level DBMS should be considered.

Security

While in development, the application will have no security requirements. When this application becomes a production instance it will need role-based login through Google LDAP and SSO Service, SSL/TLS certificates, separation of the web server and DBMS, and secure encryption between the two preferably, encrypted tables for any sensitive information recorded from credit transactions.

---

3. Project Specification

Focus / Domain / Area

GetThisBread focuses on delivering an easy to use point of sale and timecard management systems for businesses that require it. In addition, to better serve those businesses, we added inventory management.

Libraries / Frameworks / Development Environment

In order to accomplish those goals, our team used a variety of tools to build GetThisBread. Firstly for the UI side, we used Figma to storyboard and figure out the layout of the UI. The front-end was built using Angular. For the back-end, various other tools are included. The language used is PHP for the models and MySQL for the database.

Platform

As a web application, the focus is on allowing users to run it in a browser. The target is that if anyone has a device that can run Chrome, Edge, or Firefox such as a laptop or smartphone, it would be possible to run GetThisBread. Our expected libraries and development environments to develop GetThisBread will be WAMP, PHP 7.4.12, MySQL 8.0.22, VS Code and Angular.

Genre

The genre of the project is a web application with a database.

---

4. System – Design Perspective

Identify subsystems – design point of view

Design Choices

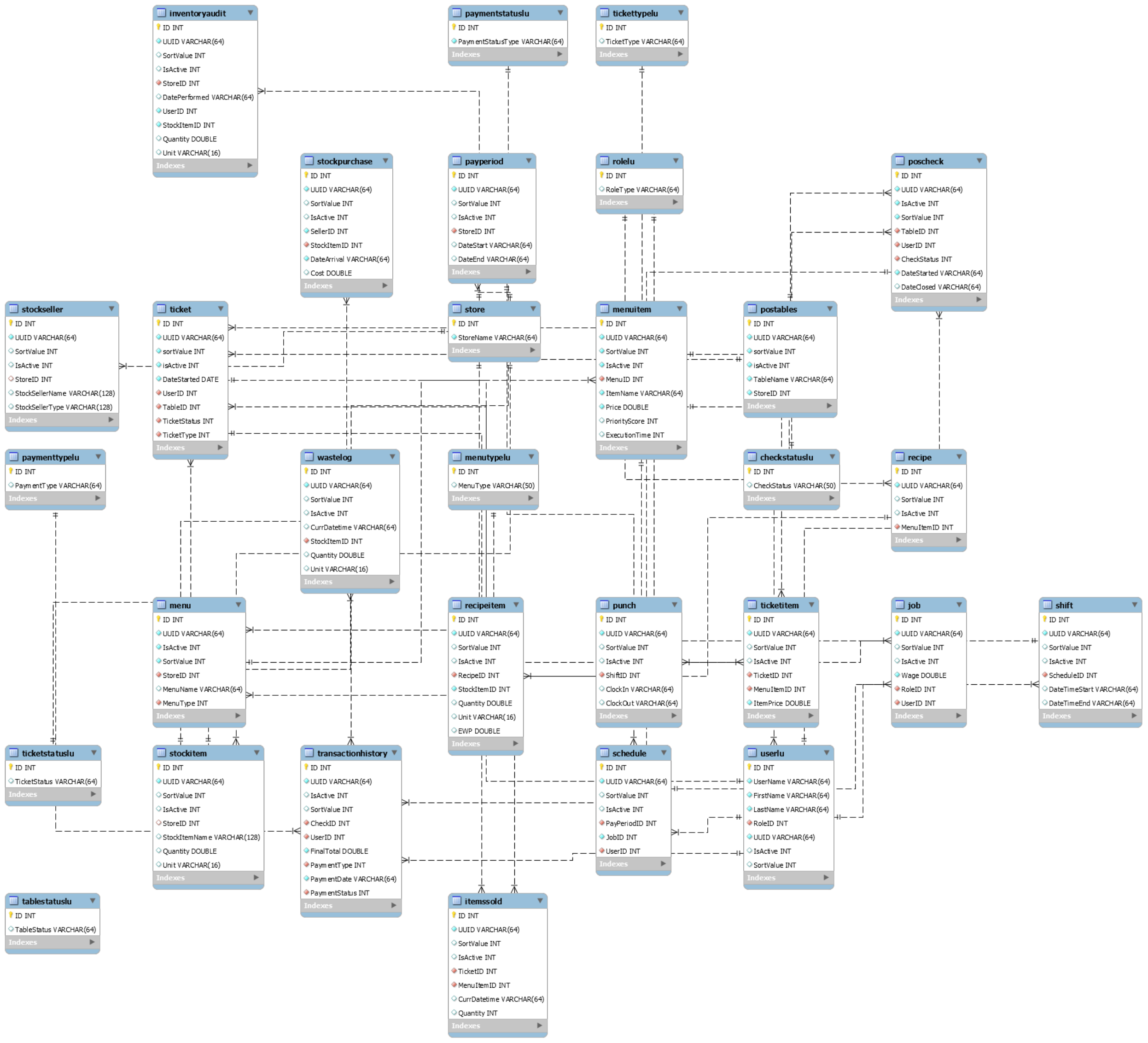
The application follows a mostly MVC and CRUD style for the bulk of the application. The advantages of this choice are clearly reusability and ease of onboarding one teammate from one component to another. Additionally, we utilized reflective class functions to minimize the amount of coding needed to access the database. The database does not follow a conventional normalization but instead relies heavily on lookup tables and mirrors the CRUD operations of the application.

Identifying Subsystems

The API and data storage subsystem, referred to as the back-end, is responsible for storing data, serving up the data back in the form of models and housing the API that supports the asynchronous calls of the rest of the subsystems.
The Point of Sales subsystem is a collection of views that fulfills all of the features laid out for the POS. The POS pulls models and data from those models from the back-end.
The Timecard Management subsystem is another collection of views that supports the features of the TMS.
The Inventory Management subsystem is a collection of views that supports the features of the IMS.
The Login subsystem is a verification of both authentication and belonging to a particular organizational unit within the Google Directory of that "store." The system is also responsible for maintaining the identity of the user throughout their session in the application.

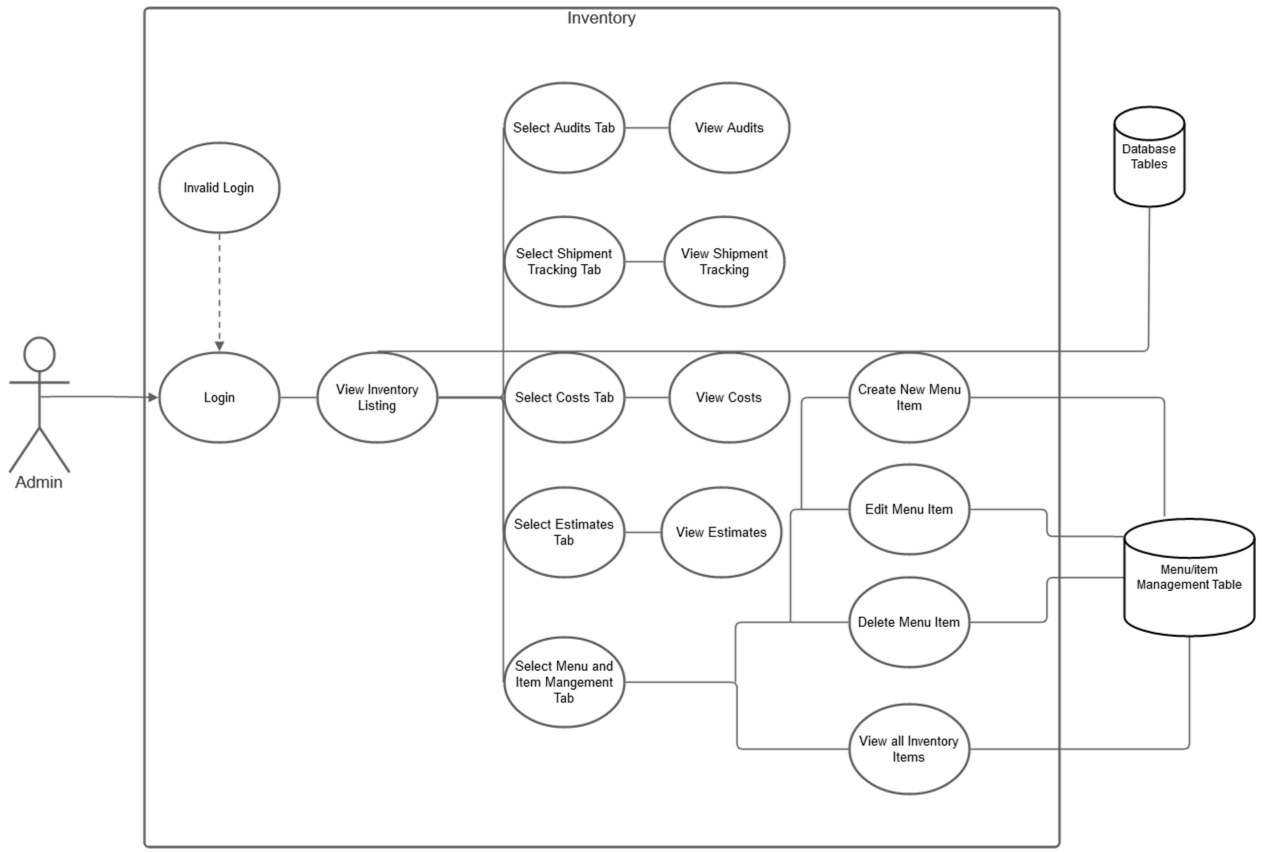
Database Subsystem

ER Diagram



Inventory Subsystem

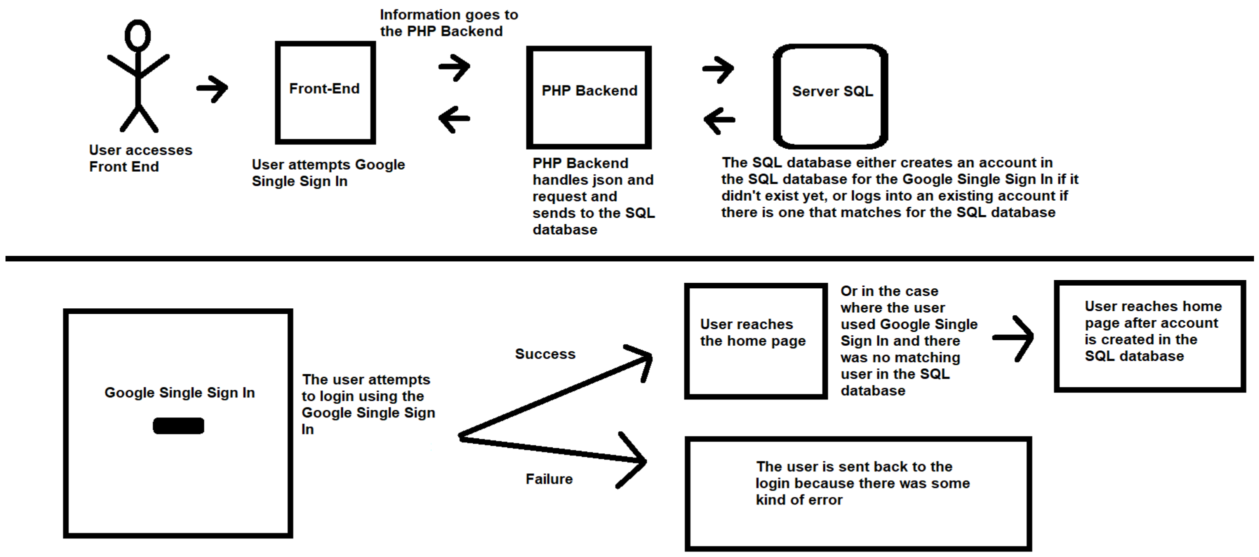
Use Case diagram:



The above diagram is a use case example of using the inventory system in GetThisBread. After the admin logs in, they can navigate to the inventory. From there the admin can navigate to view audits, track shipments, view costs, and view estimates all of which would need to pull data from the database. In addition, the admin can modify the menu from the inventory by being able to add, modify, and delete items in the menu which update the relevant table in the database. Lastly, a comprehensive inventory list can be viewed from the inventory management section.

Login Subsystem

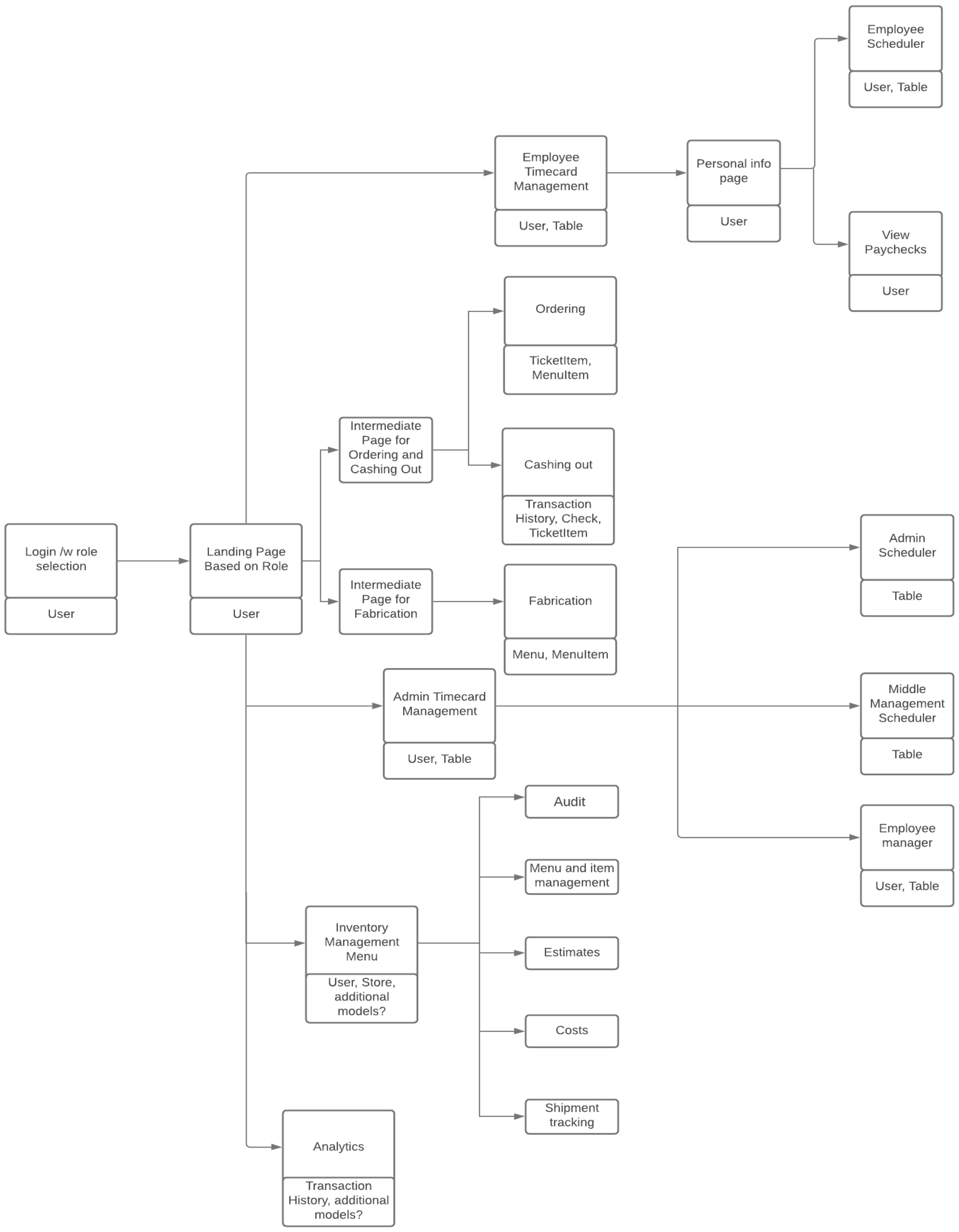
Use Case Diagram



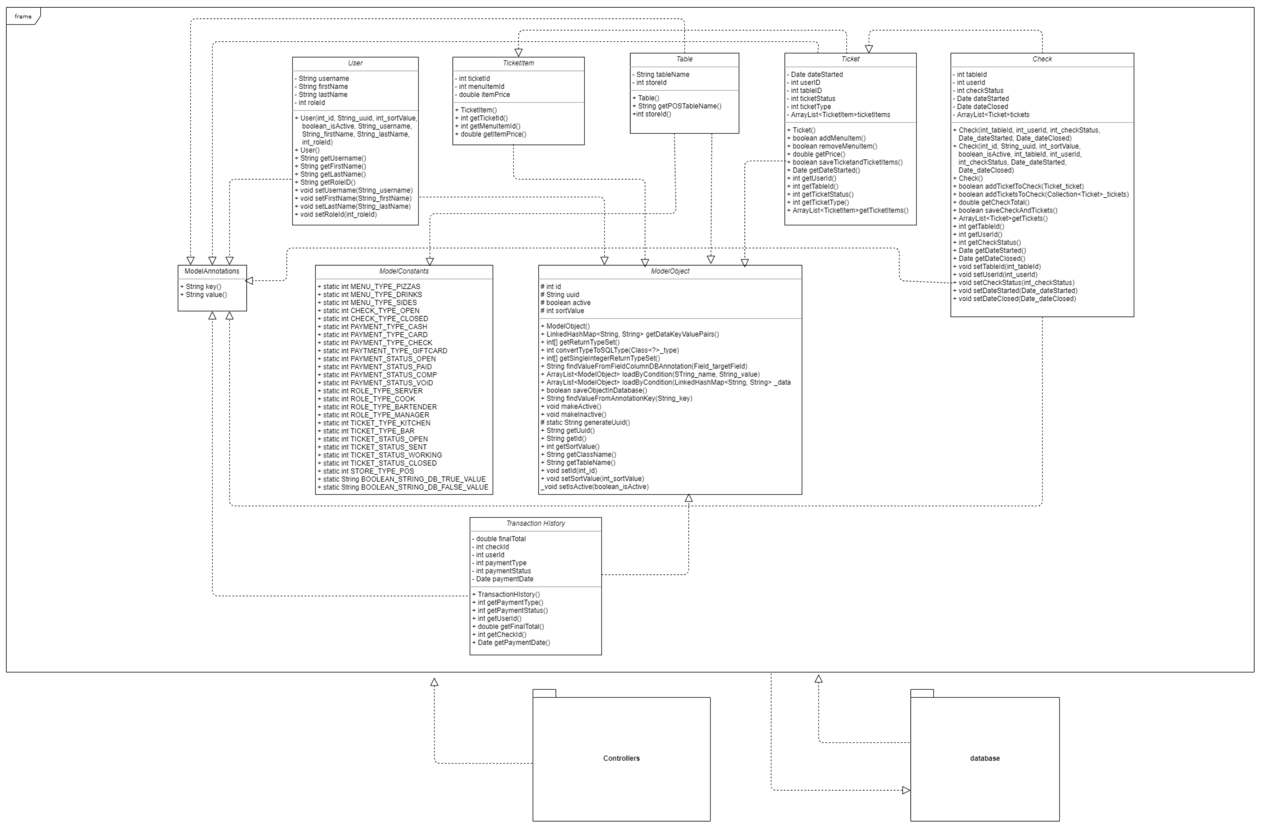
User accesses Front End
Front-End
User attempts Google Single Sign In
Information goes to the PHP Backend
PHP Backend
PHP Backend handles json and request and sends to the SQL database
Server SQL
The SQL database either creates an account in the SQL database for the Google Single Sign In if it didn't exist yet, or logs into an existing account if there is one that matches for the SQL database
Google Single Sign In
The user attempts to login using the Google Single Sign In
Success
Failure
User reaches the home page
Or in the case where the user used Google Single Sign In and there was no matching user in the SQL database
User reaches home page after account is created in the SQL database
The user is sent back to the login because there was some kind of error

PoS Subsystem

Use Case Diagram

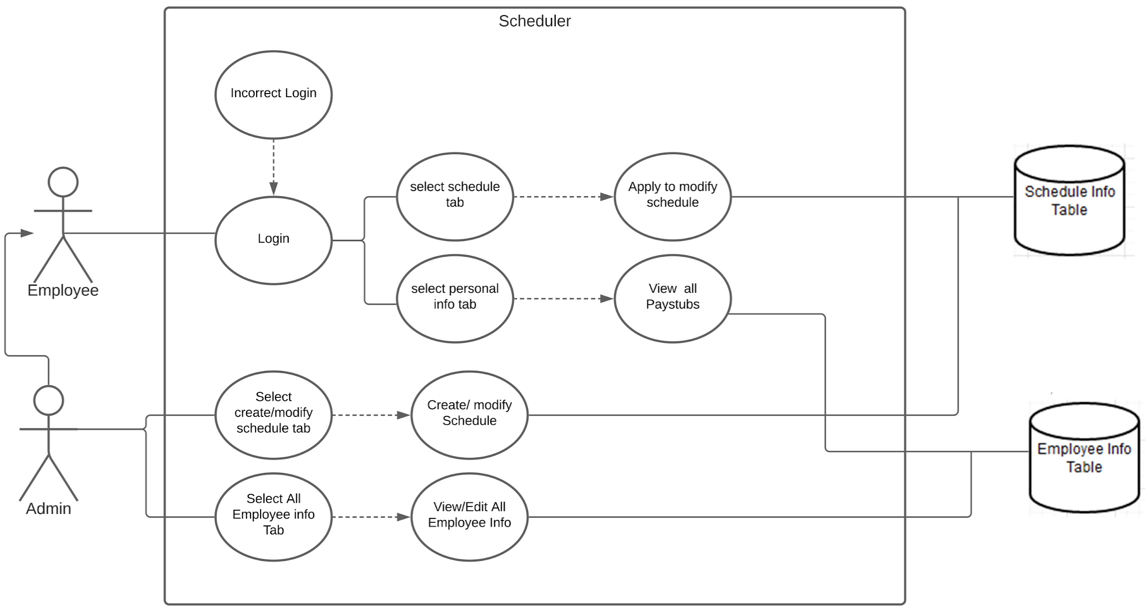

Login /w role selection
User
Landing Page Based on Role
User
Employee Timecard Management
User, Table
Personal info page
User
Employee Scheduler
User, Table
View Paychecks
User
Ordering
TicketItem, MenuItem
Intermediate Page for Ordering and Cashing Out
Cashing out
Transaction History, Check, TicketItem
Intermediate Page for Fabrication
Fabrication
Menu, MenuItem
Admin Scheduler
Table
Admin Timecard Management
User, Table
Middle Management Scheduler
Table
Employee manager
User, Table
Audit
Menu and item management
Estimates
Costs
Shipment tracking
Inventory Management Menu
User, Store, additional models?
Analytics
Transaction History, additional models?

UML Diagram


frame
User
- String username
- String firstName
- String lastName
- int roleId
+ User(int_id, String_uuid, int_sortValue, boolean_isActive, String_username, String_firstName, String_lastName, int_roleId)
+ User()
+ String getUsername()
+ String getFirstName()
+ String getLastName()
+ String getRoleID()
+ void setUsername(String_username)
+ void setFirstName(String_firstName)
+ void setLastName(String_lastName)
+ void setRoleId(int_roleId)
TicketItem
- int ticketId
- int menuItemId
- double itemPrice
+ TicketItem()
+ int getTicketId()
+ int getMenuItemId()
+ double getItemPrice()
Table
- String tableName
- int storeId
+ Table()
+ String getPOSTableName()
+int storeId()
Ticket
- Date dateStarted
- int userID
- int tableID
- int ticketStatus
- int ticketType
- ArrayList<TicketItem>ticketItems
+ Ticket()
+ boolean addMenuItem()
+ boolean removeMenuItem()
+ double getPrice()
+ boolean saveTicketandTicketItems()
+ Date getDateStarted()
+ int getUserId()
+ int getTableId()
+ int getTicketStatus()
+ int getTicketType()
+ ArrayList<TicketItem>getTicketItems()
Check
- int tableId
- int userId
- int checkStatus
- Date dateStarted
- Date dateClosed
- ArrayList<Ticket>tickets
+ Check(int_tableId, int_userId, int_checkStatus, Date_dateStarted, Date_dateClosed)
+ Check(int_id, String_uuid, int_sortValue, boolean_isActive, int_tableId, int_userId, int_checkStatus, Date_dateStarted, Date_dateClosed)
+ Check()
+ boolean addTicketToCheck(Ticket_ticket)
+ boolean addTicketsToCheck(Collection<Ticket>_tickets)
+ double getCheckTotal()
+ boolean saveCheckAndTickets()
+ ArrayList<Ticket>getTickets()
+ int getTableId()
+ int getUserId()
+ int getCheckStatus()
+ Date getDateStarted()
+ Date getDateClosed()
+ void setTableId(int_tableId)
+ void setUserId(int_userId)
+ void setCheckStatus(int_checkStatus)
+ void setDateStarted(Date_dateStarted)
+ void setDateClosed(Date_dateClosed)
ModelAnnotations
+ String key()
+ String value()
ModelConstants
+ static int MENU_TYPE_PIZZAS
+ static int MENU_TYPE_DRINKS
+ static int MENU_TYPE_SIDES
+ static int CHECK_TYPE_OPEN
+ static int CHECK_TYPE_CLOSED
+ static int PAYMENT_TYPE_CASH
+ static int PAYMENT_TYPE_CARD
+ static int PAYMENT_TYPE_CHECK
+ static int PAYTMENT_TYPE_GIFTCARD
+ static int PAYMENT_STATUS_OPEN
+ static int PAYMENT_STATUS_PAID
+ static int PAYMENT_STATUS_COMP
+ static int PAYMENT_STATUS_VOID
+ static int ROLE_TYPE_SERVER
+ static int ROLE_TYPE_COOK
+ static int ROLE_TYPE_BARTENDER
+ static int ROLE_TYPE_MANAGER
+ static int TICKET_TYPE_KITCHEN
+ static int TICKET_TYPE_BAR
+ static int TICKET_STATUS_OPEN
+ static int TICKET_STATUS_SENT
+ static int TICKET_STATUS_WORKING
+ static int TICKET_STATUS_CLOSED
+ static int STORE_TYPE_POS
+ static String BOOLEAN_STRING_DB_TRUE_VALUE
+ static String BOOLEAN_STRING_DB_FALSE_VALUE
ModelObject
# int id
# String uuid
# boolean active
# int sortValue
+ ModelObject()
+ LinkedHashMap<String, String> getDataKeyValuePairs()
+ int[] getReturnTypeSet()
+ int convertTypeToSQLType(Class<?>_type)
+ int[] getSingleIntegerReturnTypeSet()
+ String findValueFromFieldColumnDBAnnotation(Field_targetField)
+ ArrayList<ModelObject> loadByCondition(STring_name, String_value)
+ ArrayList<ModelObject> loadByCondition(LinkedHashMap<String, String> _data
+ boolean saveObjectInDatabase()
+ String findValueFromAnnotationKey(String_key)
+ void makeActive()
+ void makeInactive()
# static String generateUuid()
+ String getUuid()
+ String getId()
+ int getSortValue()
+ String getClassName()
+ String getTableName()
+ void setId(int_id)
+ void setSortValue(int_sortValue)
_void setIsActive(boolean_isActive)
Transaction HIstory
- double finalTotal
- int checkId
- int userId
- int paymentType
- int paymentStatus
- Date paymentDate
+ TransactionHIstory()
+ int getPaymentType()
+ int getPaymentStatus()
+ int getUserId()
+ double getFinalTotal()
+ int getCheckId()
+ Date getPaymentDate()
Controllers
database

Scheduling Subsystem

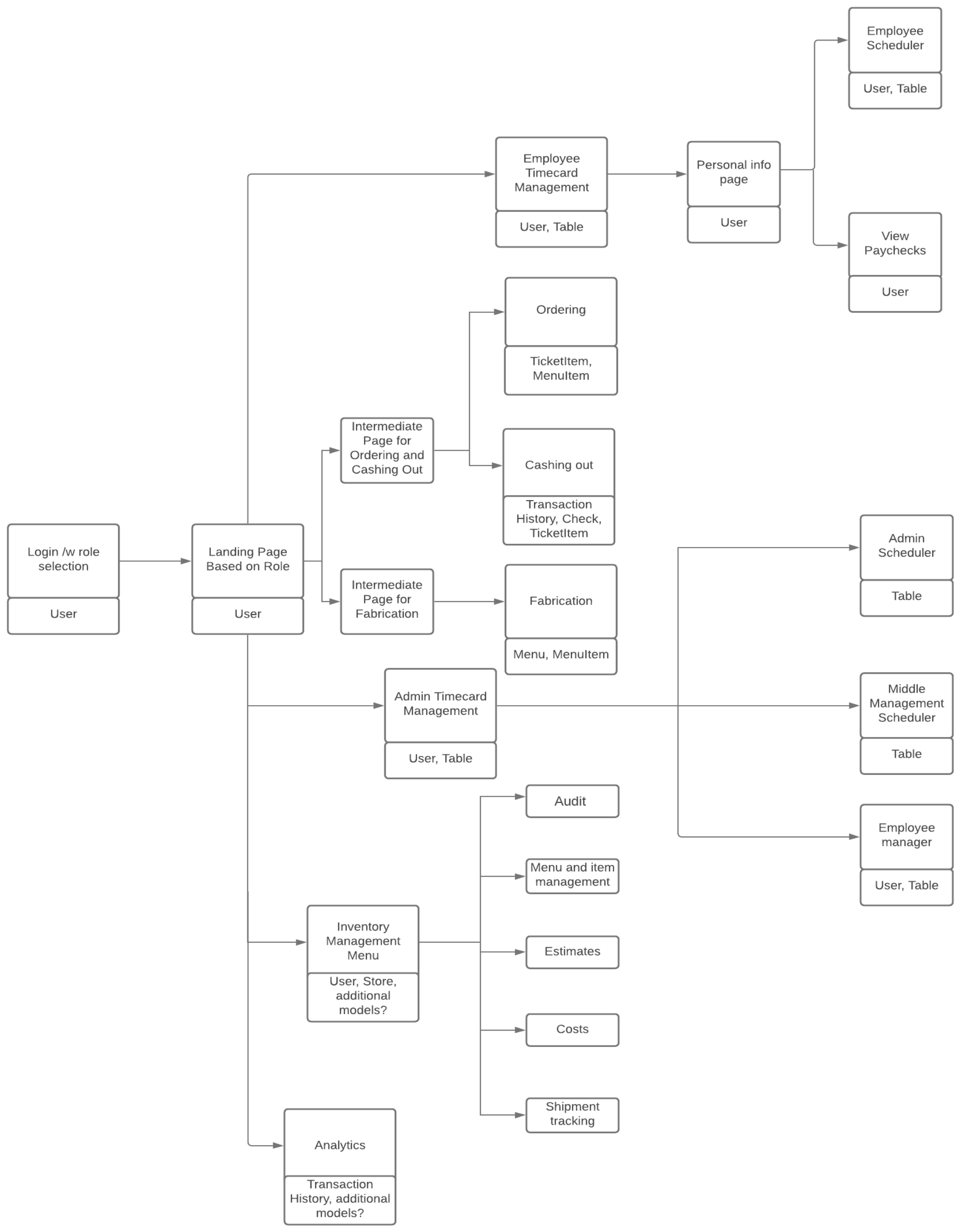
Use-case diagram:



The above diagram is a use case example of using the scheduler system in GetThisBread. It starts with a user or admin logging in. At that point, an employee can either look at their personal info which pulls data from the employee info data in our database, or update their schedule which will push out a request to the database. If you are an admin in addition to the employee's capabilities, you can also navigate to modify other employees' data and adjust the schedule which will update the relevant tables in the database.

Sub-System Communication (Diagram and Description)

Control Diagram

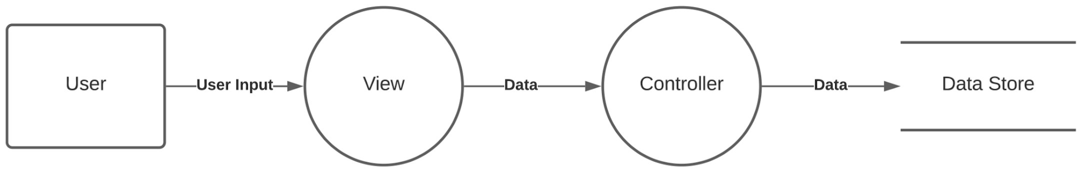

Login /w role selection
User
Landing Page Based on Role
User
Employee Timecard Management
User, Table
Personal info page
User
Employee Scheduler
User, Table
View Paychecks
User
Ordering
TicketItem, MenuItem
Intermediate Page for Ordering and Cashing Out
Cashing out
Transaction History, Check, TicketItem
Intermediate Page for Fabrication
Fabrication
Menu, MenuItem
Admin Timecard Management
User, Table
Admin Scheduler
Table
Middle Management Scheduler
Table
Employee manager
User, Table
Audit
Menu and item management
Estimates
Costs
Shipment tracking
Inventory Management Menu
User, Store, additional models?
Analytics
Transaction History, additional models?

The above diagram is an example of a control diagram for GetThisBread. The diagram shows a general idea of the current assumed flow of views and models needed for the various functional elements. The idea is to start at a login page and then have subsequent access to various other tabs and panels as necessary. From the landing page, each branch out can be considered a subsystem with the following views and models being parts of those subsystems. Lastly, the diagram also communicates the flow of expected control for those subsystems.

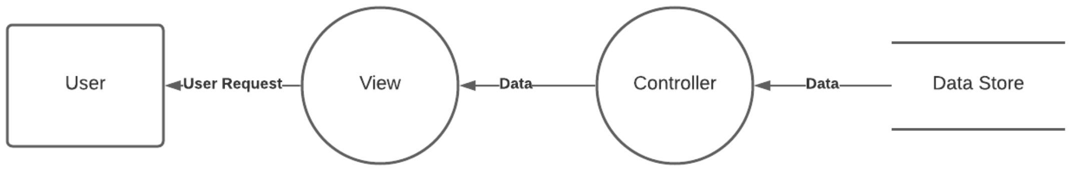
I/O - Communication between Subsystems:

Users (Managers, Employees), Google API, and the SQL Database independently request and receive information from the main GetThisBread system as shown within the context diagram. This is done with Ajax calls to subsystems. These calls hit the back-end API and serve the data in JSON format using predefined models. Upon return, the process is reversed.

Data Flow Diagrams:



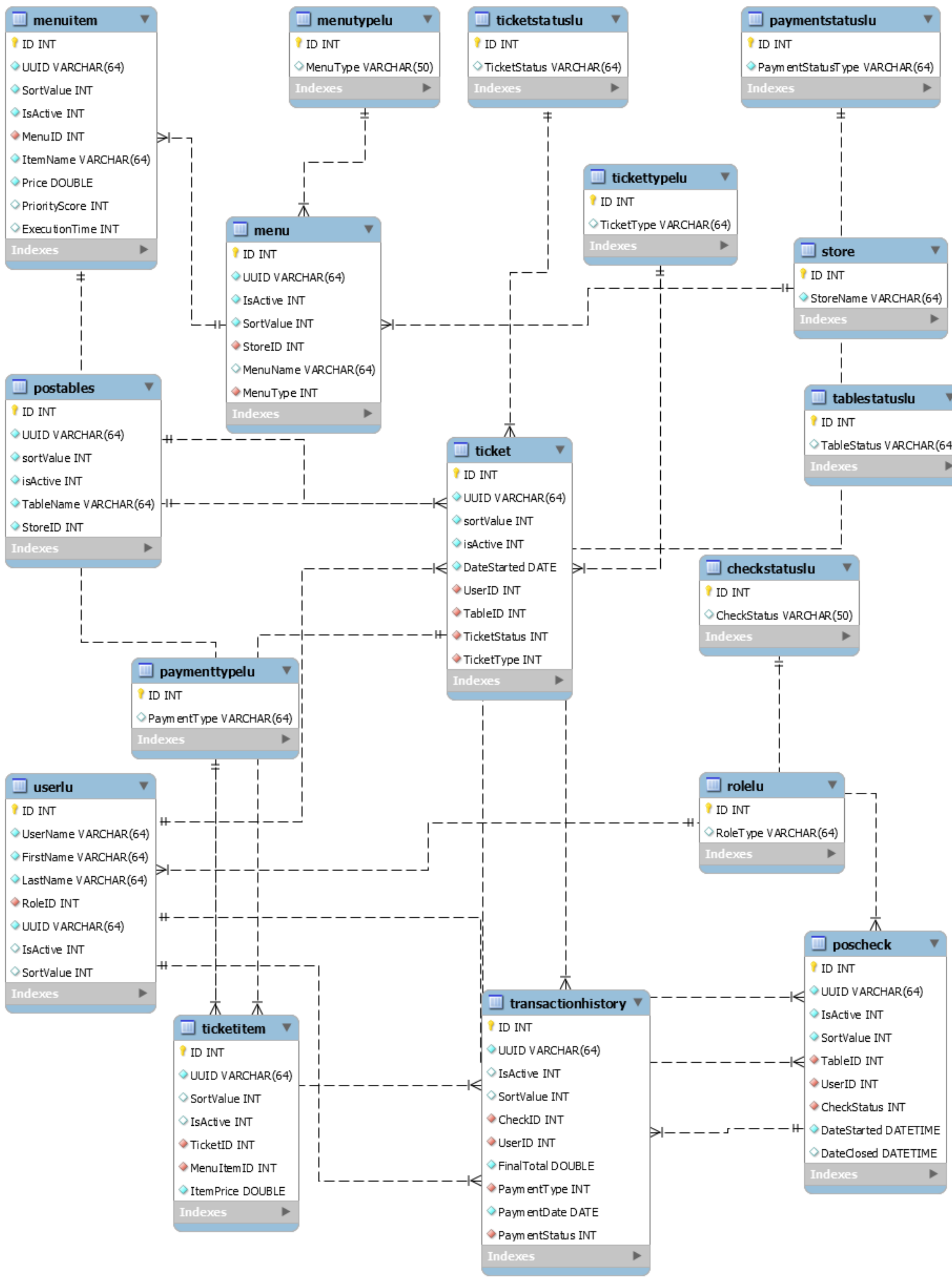
The above sample diagram shows the expected flow of data for user input in GetThisBread. It is a simple setup with the user inputting data into a view after which it passes it on the controller and then finally to the database for storage.



The above sample diagram shows the expected flow of data for a user request. A user makes a request and data is pulled up to the view from the database to display to the user.

Entity-Relationship Model (E-R Model)

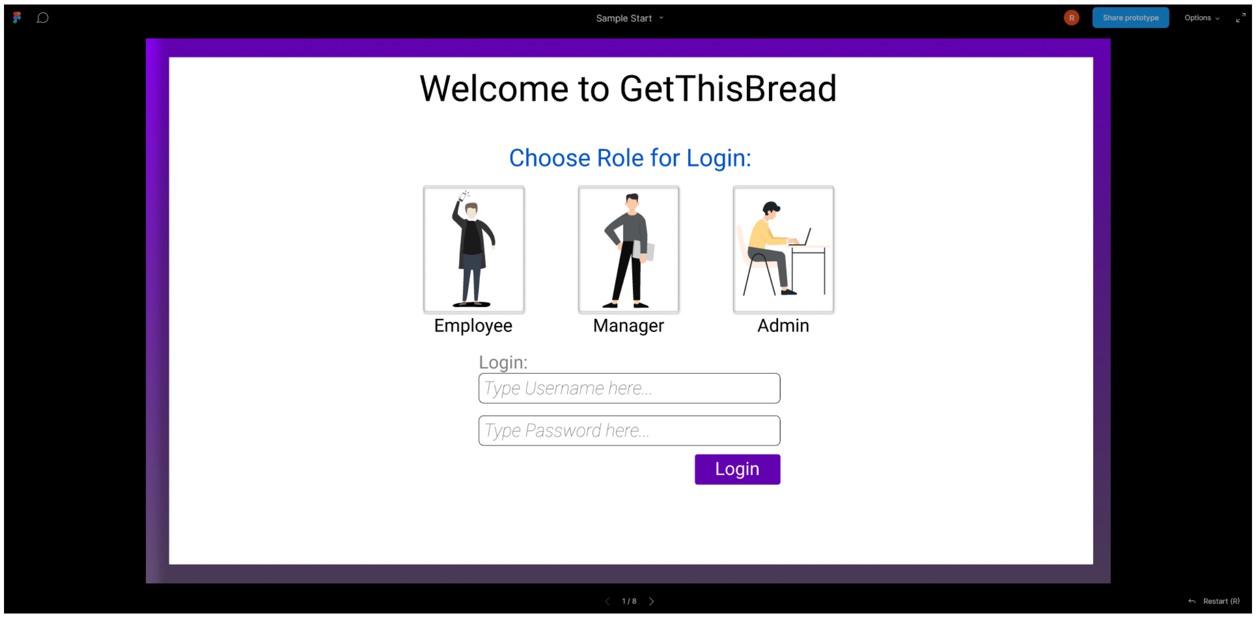
The initial ER diagram of our database.



The above entity-relationship diagram was created using the existing MySQL database. This diagram highlights the primary and foreign key relationships within the models in the POS subsystem.

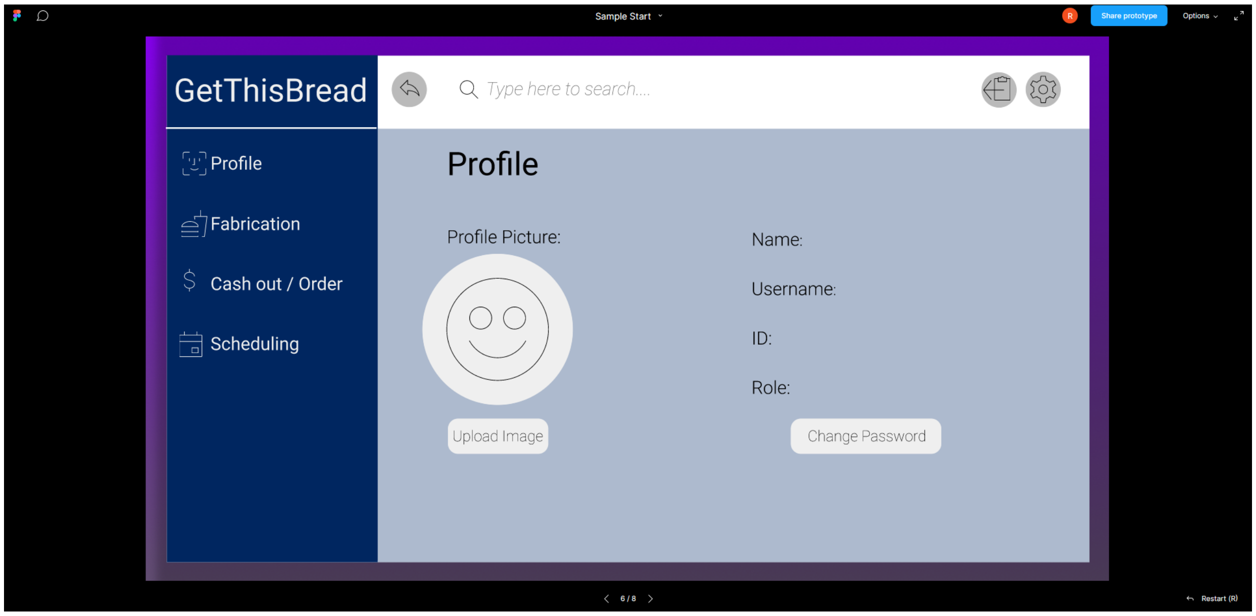
System Model

System Interaction Sample 1:



The above sample shows an example of the expected login page. It would tie into the login system.

System Interaction Sample 2:



The above sample is an example of the expected landing page. It is an early prototype and would serve as a connection to all the other views and subsystems in GetThisBread

5. System – Analysis Perspective

Identify subsystems – analysis point of view

System (Tables and Description)

Subsystem	Description
SSO	Google SSO that authenticates the user and allows them to be instantiated by the application.
Inventory	Tracks the incoming and outgoing material of a store.
Scheduling	Captures the scheduling of employees.
POS	POS captures customer selection of products.
Back-end	Keeps the state of all instances of other front-end models.

Process Model:

Our team has decided to develop code on an Agile basis. We made this choice based on several key factors. Our requirements document is a living document that is continuously updated. Items are being refactored and new discoveries are being made every day to make this application better. Constant discussion about the nature and spirit of the requirements is a constant topic at all our meetings.

Algorithm Analysis

Analysis of Subsystems and Subsystem Algorithm Analysis:
For purposes of this section, we are going to divide the subsystems into 3 families: data transforming front-end subsystems, back-end subsystems, and data displaying front-end subsystems. The back-end subsystems maintain state and provide data to the views and determine the time complexity of the other two families of the application. The login subsystem works in relative linear time, if not constant. The login subsystem fetches the authentication of a user and maintains session details. A large portion of the time complexity is derived from the database subsystem. The database will be organized such that extracting data will be stacked with one-to-many rows. An example being, extracting the items offered by one location could be as time-consuming as using a store ID to find all menu IDs of that store then menuItems of that menu. This process is polynomial-time but

by using a store-specific configuration file and creating a series of global variables the time could be shortened to nlog(n).
The other two families of subsystems, written mostly in angular, are initialized partly by the webserver in constant time. The web server serves up html for initial consumption but portions will later be altered as the page comes live from ajax calls made to the back-end subsystem. For instance, a display of all drinks offered at a bar will come up in constant time relative to the time the back-end subsystem brought up the view, models, and data for that particular page. However, some pages will require an ajax call to populate a table, fill out the forms or assemble lists. Since the bulk of the process is carried out on the web server via the ajax call we can express this time complexity as linear.

---

6. Project Scrum Report

Product Backlog

Task Name	Owners	EE	PE	Status
Database table creation	Ian	20	40	DONE
Log In Google SSO	Mike/Ronald	10	10	Done
Log in Chooses Role	Mike/Ronald	10	10	WIP
Log in Changes Navigation	Mike/Ronald	10	10	DONE
Log in API call for Roles	Mike/Ronald	10	10	DONE
Log in Persists	Mike/Ronald	10	10	DONE
Inventory Models	Group	10	20	Done
Schedule Models	Group	10	20	Done
Log in views	Mike/Ronald	10	10	Done
Create schedule view	Jason	20	40	DONE
Schedule API calls	Jason	40	80	WIP
Inventory View	Richard	40	80	Done
Inventory Controller	Richard	40	120	DONE

Schedule Controller	Jason	40	120	WIP
Database stored procedures	Ian	20	80	Done
Database Seed/LU/Funcs	Ian	20	80	Done
API file	Ian	10	10	Done
API Utility Class	Ian	10	10	Done
Database Classes	Ian	20	20	Done
Database scripts	Ian	5	5	Done
POS views	Emily	20	40	DONE
POS Controller	Emily/Ian	20	40	WIP
Report 1 Draft	Group	20	40	DONE
Report 1 Finalized	Emily/Ian	5	10	DONE
Report 2 Draft	Group	20	40	Done
Report 3 Draft	Group	20	40	Done
Report 2 Finalized	Emily/Ian	5	10	Done
Report 3 Finalized	Emily/Ian	5	10	Done
Slides 1	Group	10	10	Done
Slides 2	Group	10	10	Done
Slides 3	Group	10	10	Done
Poster Draft	Group	10	10	DONE
Poster Final	Emily/Ian	10	10	DONE

---

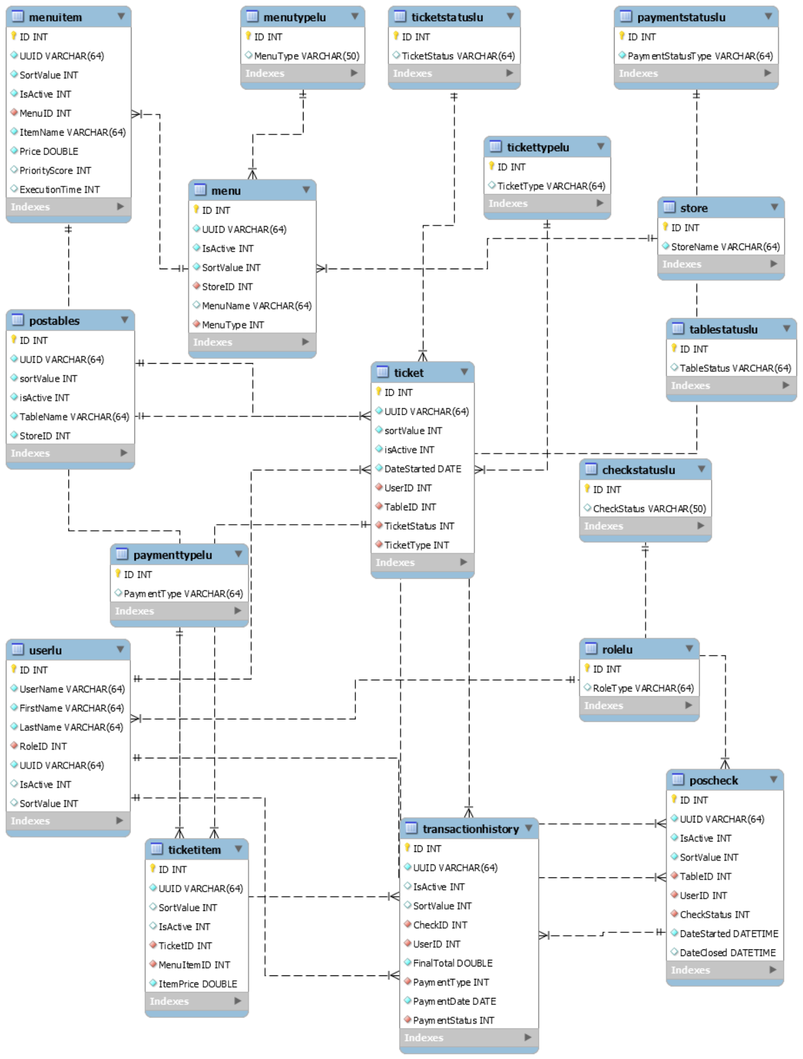
7. Subsystems

7.1 Database Subsystem – Ian Wilhelmsen

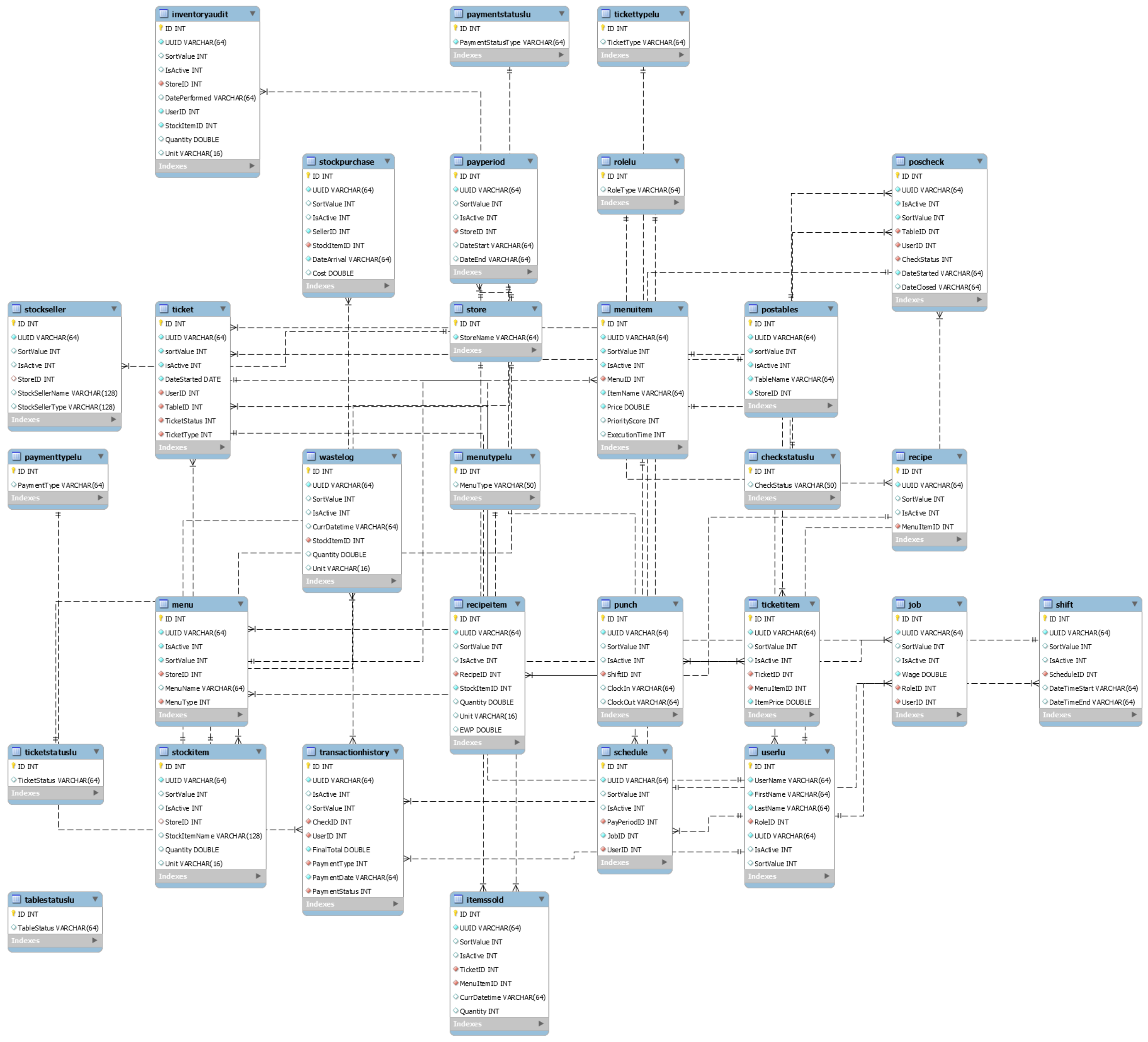
Design and model

The design of this subsystem is to emulate all other subsystems on a model basis and allow interaction on a CRUD level through an API.

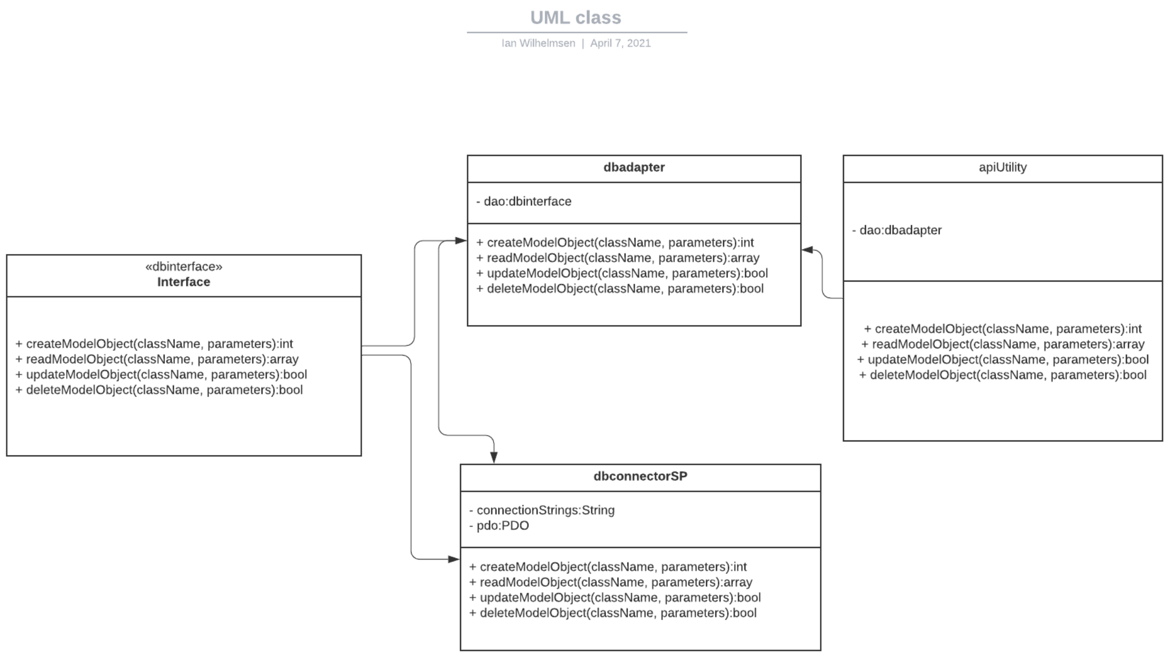
Previous Entity-Relationship Model (E-R Model):



Finalized Entity-Relationship Model (E-R Model):



UML Diagram


UML class
Ian Wilhelmsen | April 7, 2021
dbadapter
- dao:dbinterface
+ createModelObject(className, parameters):int
+ readModelObject(className, parameters):array
+ updateModelObject(className, parameters):bool
+ deleteModelObject(className, parameters):bool
apiUtility
- dao:dbadapter
+ createModelObject(className, parameters):int
+ readModelObject(className, parameters):array
+ updateModelObject(className, parameters):bool
+ deleteModelObject(className, parameters):bool
«dbinterface»
Interface
+ createModelObject(className, parameters):int
+ readModelObject(className, parameters):array
+ updateModelObject(className, parameters):bool
+ deleteModelObject(className, parameters):bool
dbconnectorSP
- connectionStrings:String
- pdo:PDO
+ createModelObject(className, parameters):int
+ readModelObject(className, parameters):array
+ updateModelObject(className, parameters):bool
+ deleteModelObject(className, parameters):bool

Design choices

I decided on a purely functional simplified API. Instead of supporting many pages and expanding skeleton architecture, all requests are funnelled into a few functions that reflectively handle all data. Furthering the design would include making more models that handle error checking reflectively and produce conditions to pull or push data into the database.

Data dictionary

As the back-end of the entire application and its subsystems, this subsystem supports all the models of all the other subsystems.

If refined (changed over the course of the project)

The design of this subsystem changed over the course of this project significantly. Models were added, subsystems were flushed out and the shape of the database was changed. Many of the subsystems did not finalize their versions of models or establish the parameters needed until very recently. This was expected as we are developing on an agile concept. The team is constantly making the design better and the back-end needed to support these changes.

The cost of this flexibility is error checking. In a perfect world, the back-end would handle error checking in addition to the ETL functions it needs to. This is still possible. Reflexive model based error checking can still be included. Another issue is that in future load balancing cannot be supported programmatically. At the most load balancing can only take place on the web server or at the physical level. The way to handle this would be to adopt a more distributed API file structure but that would mostly come after the application becomes fully established. If the final structure of the application flushes out, both of these features can be added and more security can be added to the application.

The model analysis has not changed for the back-end. The back-end API portion still operates on linear time. The database handles retrieving and all $O(n^2)$ operations. Due to our choice to use only CRUD operations, all of our operations are linear time except read operations. Reads are still returns of foriegn key restrained rows that are active for the application. The design can stand as it is for now. Later releases would need to completely change end points of API or significantly reroute traffic.

Coding

Approach (Functional, OOP)

The approach taken is a functional object oriented approach. The subsystem receives a request to the API which spins up an apiUtility class to arrange the hand-off of parameters and retrieval of data from the Database Access Object. That object directly talks with the database through prepared statements calling stored procedures with credentials to only execute things on the database. The return is evaluated and then json encoded before being sent back as a response to the caller.

Language

This subsystem used PHP, SQL, and a small amount of T-SQL.

User training

This portion of the application has no training or user manual for its use. Additions, updates or changes in this functionality should be handled by a development team or single software developer to gather new requirements and features from the stakeholders and then begin work. There is no reason for a lay person to explore this subsystem.

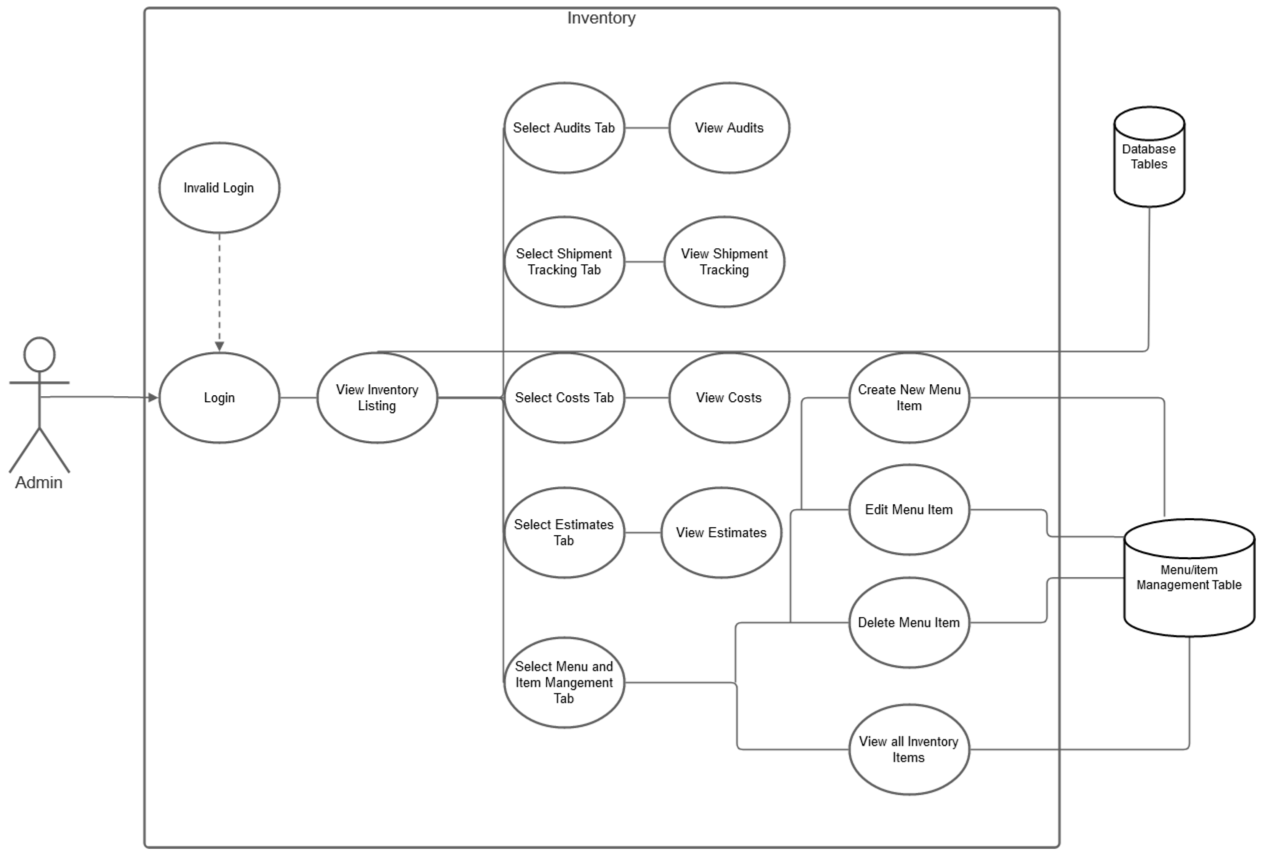
Testing

Testing can be accomplished by running a series of inputs in an api tester freely available. Preferably, Postman, a free to use application can be used for unit and integration testing. Postman can be programmed for a series of requests to supply and will check inputs.

7.2 Inventory Subsystem – Richard Abrams

Design and model

Use Case Diagram



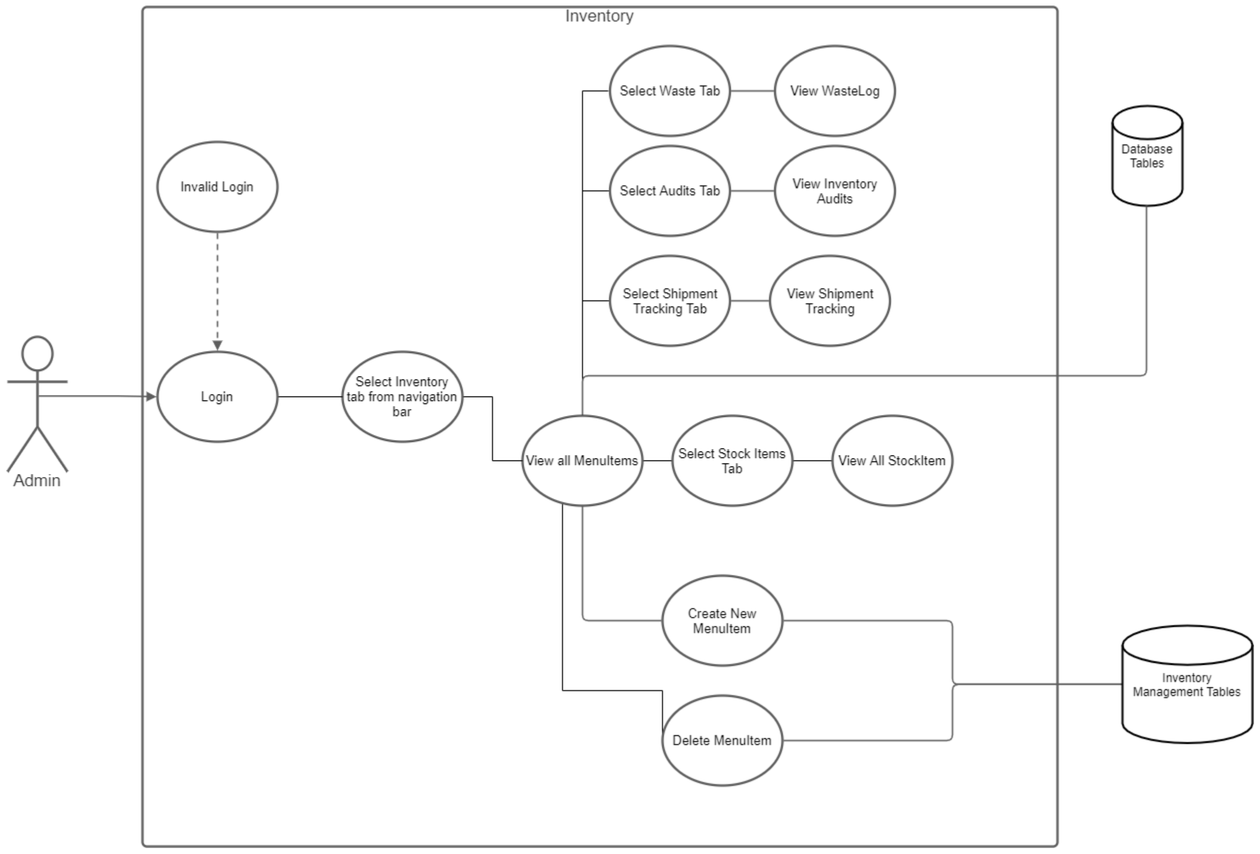
The above diagram was the original intended use case diagram for the inventory subsystem.

As the project has progressed, various changes have been made to how the subsystem functions. The most prominent change from the user perspective is that the navigation bar is much more of a focus removing the potential need for a dedicated inventory landing page. In addition, there have been some table modifications to handle the various use cases more efficiently. Lastly, one of our plans is to move away from vanilla HTML to something more robust such as JQWidgets or Bootstrap.

The pros of these changes include an easier navigation experience for the user in specific use cases, such as inventory listing. The table modifications provide all the data required to handle the potential use cases. Additionally, a move to JQWidgets or Bootstrap would allow us to upgrade the look of the HTML list and form to fit a stylized and simple look to further ease the creation of new pages.

The cons of these changes mainly rest with the navigation bar reliance and the switch from HTML. Utilizing the navigation bar more will possibly make it look more obtuse and harder to use. For the HTML switch, there is a training turnaround period to get it setup. In addition, the views overhead will slightly increase and make it more expensive to run.

The changes in the model from the initial design are focused on allowing for a better setup for the new target use cases, which include displaying stock and menu items, addition and deletion of menu items, showing potential wastage, viewing an inventory audit, and tracking shipments. The new use case diagram for this subsystem can be seen below:



In the redone use case, if the changes are made the user will have an easier time to access certain features and is more in tune with the angular component structure.

Design choices

From the initial design, it was intended from the login to an inventory landing page that would act as a portal to other parts of the inventory system such as viewing the entire inventory and audits. All relevant tables would be updated and pulled from. For the front-end part of the system, Angular was utilized using TypeScript, HTML, and SCSS to accomplish view construction. A REST API is utilized for external communication.

Data dictionary

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"Store"	"ID"	"Int"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"ItemsSold"	"TicketID"	"Int"
"ItemsSold"	"MenuItemID"	"Int"
"ItemsSold"	"CurrDatetime"	"Int"
"ItemsSold"	"Quantity"	"Float"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"WasteLog"	"ID"	"Int"
"WasteLog"	"CurrDatetime"	"Int"
"WasteLog"	"StockItemID"	"Int"
"WasteLog"	"Quantity"	"Float"
"WasteLog"	"Unit"	"String"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"Recipe"	"MenuItemID"	"Int"
"Recipe"	"ID"	"Int"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"RecipeItem"	"RecipeID"	"Int"
"RecipeItem"	"StockItemID"	"Int"
"RecipeItem"	"Quantity"	"Float"
"RecipeItem"	"Unit"	"String"
"RecipeItem"	"EWP"	"Float"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"StockItem"	"ID"	"Int"
"StockItem"	“StoreID"	"Int"
"StockItem"	"StockItemName"	"String"
"StockItem"	"Quantity"	"Float"
"StockItem"	"Unit"	"String"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"StockPurchase"	"ID"	"Int"
"StockPurchase"	"SellerID"	"Int"
"StockPurchase"	"StockItemID"	"Int"
"StockPurchase"	"DateArrival"	"Int"
"StockPurchase"	"Costs"	"Float"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"StockSeller"	"ID"	"Int"
"StockSeller"	"StoreID"	"Int"
"StockSeller"	"StockSellerName"	"String"
"StockSeller"	"StockSellerType"	"String"

“TABLE ASSOCIATION”	**“VARIABLE NAME”**	**“DATA TYPE”**
"InventoryAudit"	"ID"	"Int"
"InventoryAudit"	"StoreID"	"Int"
"InventoryAudit"	"DatePerformed"	"Int"
"InventoryAudit"	"UserID"	"Int"
"InventoryAudit"	"StockItemID"	"Int"
"InventoryAudit"	"Unit"	"Float"
"InventoryAudit"	"Quantity"	"String"

Coding

The approach for the front-end of the inventory system is utilizing Angular 11.2. By utilizing that version of Angular, it forces the use of TypeScript to support HTML and SCSS for view creation. In addition, the Angular framework is handled in discrete units called components. Each component represents a logical User Interface (UI) unit such as a navigation bar or an HTML list. Each component is defined by a TypeScript file, an HTML file, and in our case an SCSS file. In addition, each subsystem has a service that can make external calls utilizing the REST api.

Approach (Functional, OOP)

The project baseline uses the Angular base template for a blank project. It includes base files to allow the Angular framework to run in addition to an app which serves as the start single page for Angular to work off of. From there within the app each front-end part of each subsystem is separated out into their respective folders. Each logical unit in Angular like a component or service receives its own folder to group the necessary files together. Each subsystem interacts with the app files to allow for navigation maintenance.

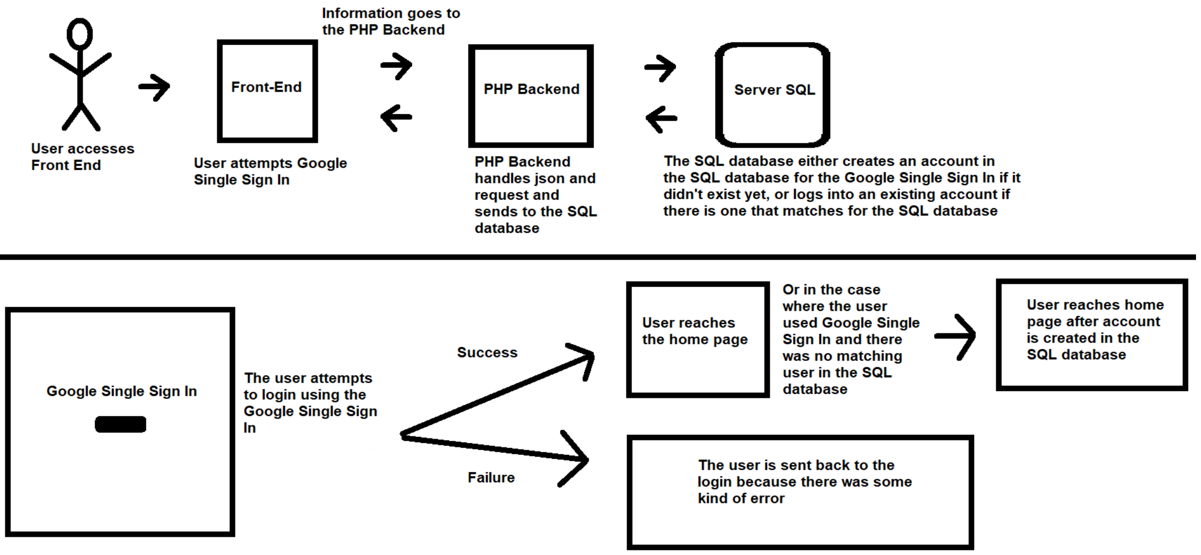
Language

For the inventory, a separate inventory page component was created to serve as a navigation link for the app router within the project. The inventory component page also serves as a formatting unit for the layout of the inventory display as well as providing inventory specific styles. What comprises the main body of the inventory is the inventory list component. The inventory list component's main purpose is to serve as the view of an HTML list. It possesses various TypeScript lists to handle the information it displays. In the future, that task might be moved to another per entry component to allow for better styling. Lastly, it also sends requests to the inventory services which handles external communication.

7.3 Login Subsystem – Mike Calvo and Ronald Abrams

Design and model

Use Case Diagram


User accesses Front End
Front-End
User attempts Google Single Sign In
Information goes to the PHP Backend
PHP Backend
PHP Backend handles json and request and sends to the SQL database
Server SQL
The SQL database either creates an account in the SQL database for the Google Single Sign In if it didn't exist yet, or logs into an existing account if there is one that matches for the SQL database
Google Single Sign In
The user attempts to login using the Google Single Sign In
Success
Failure
User reaches the home page
Or in the case where the user used Google Single Sign In and there was no matching user in the SQL database
User reaches home page after account is created in the SQL database
The user is sent back to the login because there was some kind of error

Design choices

The login subsystem has one way to login, being a sign in from the front-end to the back-end using Google's Single Sign In to login. The way this login subsystem is going to work is that the client can make the request to see if it has a Google Account through Google Single Sign In. If the client does, the front-end will send a post request or requests where it is required. This will be potentially because of whether the account is there in the SQL database or not. The front-end then uses the information the client gets back from the SQL database. The rest of how this is handled is done by the back-end, which takes the information provided from the client and sees if what is passed as a Username value matches an account in the SQL database.

Data dictionary

"TABLE ASSOCIATION"	"VARIABLE NAME"	"DATA TYPE"	"DESCRIPTION"
"RecordedUsers"	"IDUser"	"Int"	"This field is for "ID"

			that tracks each user." PK
“RecordedUsers”	“UUID”	“String”	“This field guarantees that a recognized “IDUser” also can be uniquely found no matter what.” PK
“RecordedUsers”	“FirstName”	“String”	“This field gives a given recognized “IDUser’s” “FirstName”.”
“RecordedUsers”	“LastName”	“String”	“This field gives a given recognized “IDUser’s” “LastName”.”
"RecordedUsers"	"UserName"	"String"	"This field gives the name for the given user."
"RecordedUsers"	"UserType"	"String"	"This field will determine if the user is a manager, worker, or admin."
"RecordedUsers"	"UserAuthorityLevel"	"Int"	"This is a value starting from 1 which goes up to show the level of access the user has."

Coding

Approach (Functional, OOP)

The idea behind how this will happen with the back-end, is when the user logs in using the Google Single Sign In, a post request is sent to the PHP back-end, which takes the Google Account’s email value parsed to just the name, and checks that against the SQL database, potentially creating an account in the SQL database if it does not have a matching value, or finding the account in the SQL database that is there with a matching Username.

Language

An important thing to understand is that the post from the front-end actually goes to the basic CRUD operations in the back-end, and this is determined by a specifier in the json that is transferred from the front-end to the back-end.

User training

Training / User manual (needed for final report)

Using the Login Subsystem relates front-end Angular with the back-end PHP system. The Login Subsystem itself uses Firebase with the front-end and makes a post or posts to the back-end by sending a matching Username to the database. A person should have a Firebase dedicated location in their code that returns a Google Account, and this should be stored and used to match a Username in the database. This should allow the front-end to have access to a specific AuthorityLevel, and this value would then have to be stored to persist across the front-end session. All points where the front-end needs to decide what pages are visible would need to call this AuthorityLevel, and from what it has, this will decide which pages can be accessed.

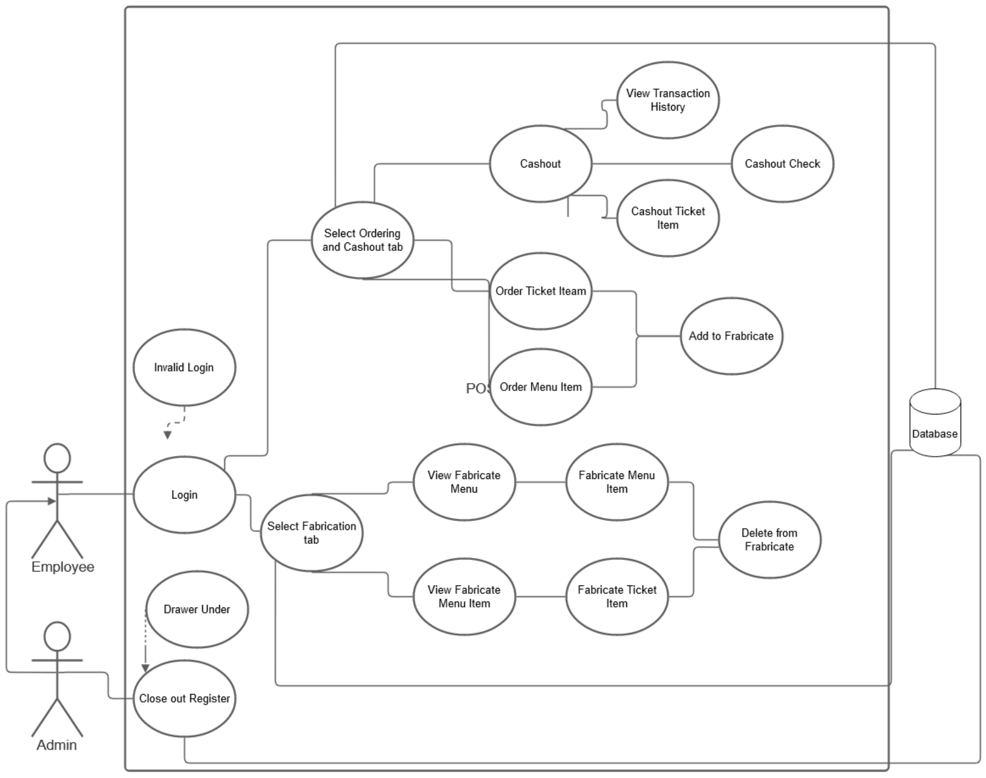
Testing

To confirm whether the login subsystem is working or not, the front-end would need to login using the Google Single Sign In, and then send a post request with a Username, going through the back-end to then potentially create an account in the SQL database if there isn't one, or getting back the account and its "UserAuthorityLevel" if there does exists one. After this, the front-end will display possible pages to click on that are based on the "UserAuthorityLevel", meaning theoretically if there are different pages visible on the front-end because a user is only an Employee rather than an Admin, these changes should be visible based on which user is logged in.

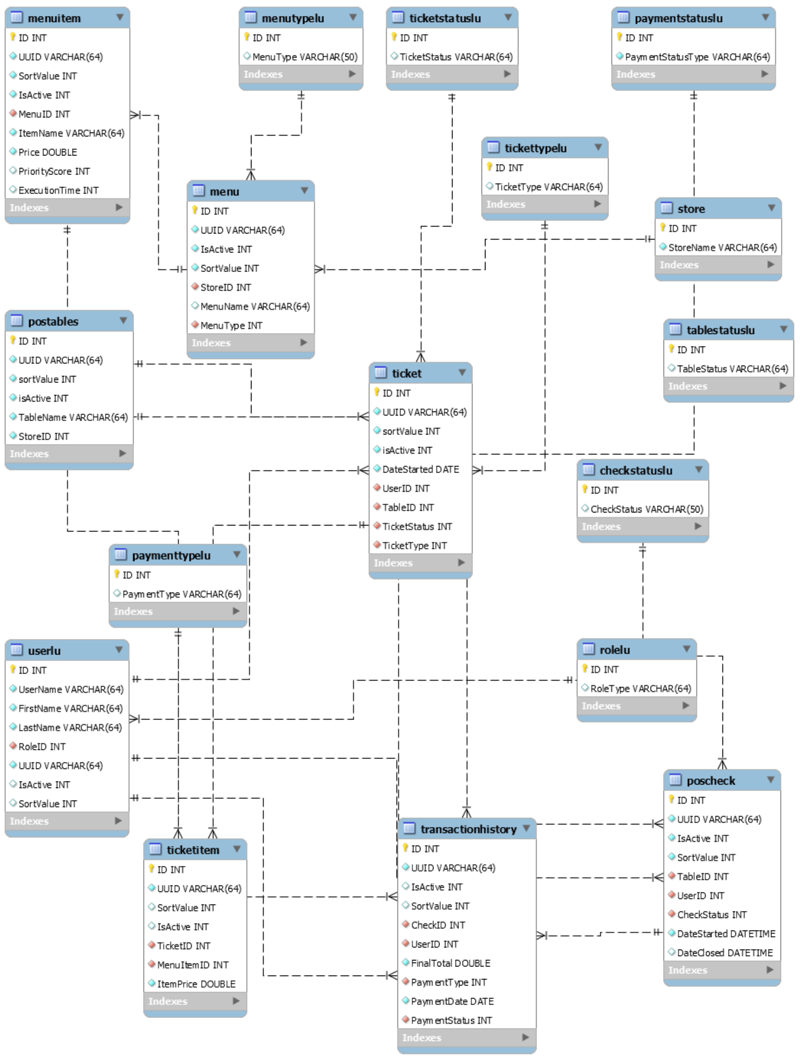
7.4 Point of Sales Subsystem – Emily Ashburn

Design and model

Use Case Diagram



ER Model



Design choices

Some of the design choices made in this subsystem include the use of Bootstrap, HTML, CSS, and JavaScript instead of Angular.
Additionally, this subsystem will utilize the MVC structure principle in order to efficiently communicate back and forth with the database.

Data dictionary

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
menu	ID	int	no	PK		

menu	UUID	varchar	no			
menu	IsActive	int	no			
menu	SortValue	int	no			
menu	StoreID	int	no	FK		
menu	MenuName	varchar	no			
menu	MenuType	int	no	FK		

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
menuitem	ID	int	NO	PRI	NULL	auto_increment
menuitem	UUID	varchar(64)	NO		NULL	
menuitem	SortValue	int	NO		0	
menuitem	IsActive	int	NO		1	
menuitem	MenuID	int	NO	MUL	NULL	
menuitem	ItemName	varchar(64)	NO		NULL	
menuitem	Price	double	NO		NULL	
menuitem	PriorityScore	int	YES		NULL	
menuitem	ExecutionTime	int	YES		NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
menutypelu	ID	int	no	PK		
menutypelu	MenuType	varchar	no			

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
paymentstatuslu	ID	int	no	PK		

paymentstatuslu	PaymentStatusType	varchar	no			

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
paymenttypelu	ID	int	no	PK		
paymenttypelu	PaymentType	varchar	no			

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
poscheck	ID	int	NO	PRI	NULL	auto_increment
poscheck	UUID	varchar(64)	NO		NULL	
poscheck	IsActive	int	NO		1	
poscheck	SortValue	int	NO		0	
poscheck	TableID	int	NO	MUL	NULL	
poscheck	UserID	int	NO	MUL	NULL	
poscheck	CheckStatus	int	NO	MUL	NULL	
poscheck	DateStarted	datetime	NO		NULL	
poscheck	DateClose	datetime	YES		NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
postables	ID	int	NO	PRI	NULL	auto_increment
postables	UUID	varchar(64)	NO		NULL	
postables	sortValue	int	NO		0	
postables	isActive	int	NO		1	
postables	TableName	varchar(64)	NO		NULL	

postables	StoreID	varchar(64)	NO		NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
recipe	ID	int	NO	PRI	NULL	auto_increment
recipe	UUID	varchar(64)	NO		NULL	
recipe	SortValue	int	YES		0	
recipe	isActive	int	YES		1	
recipe	MenuItemID	int	NO	MUL	NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
recipeitem	ID	int	NO	PRI	NULL	auto_increment
recipeitem	UUID	varchar(64)	NO		NULL	
recipeitem	SortValue	int	YES		0	
recipeitem	IsActive	int	YES		1	
recipeitem	RecipeID	int	NO	MUL	NULL	
recipeitem	StockItemID	int	NO		NULL	
recipeitem	Quantity	double	YES		NULL	
recipeitem	Unit	varchar(16)	YES		NULL	
recipeitem	EWP	double	YES		NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
rolelu	ID	int	NO	PRI	NULL	
rolelu	RoleType	varchar(64)	YES		NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
store	ID	int	NO	PRI	NULL	
store	StoreName	varchar(64)	NO		NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
ticket	ID	int	NO	PRI	NULL	auto_incremenet
ticket	UUID	varchar(64)	NO		NULL	
ticket	sortValue	int	NO		0	
ticket	isActive	int	NO		1	
ticket	DateStarted	date	NO		NULL	
ticket	UserID	int	NO	MUL	NULL	
ticket	TableID	int	NO	MUL	NULL	
ticket	TicketStatus	int	NO	MUL	NULL	
ticket	TicketType	int	NO	MUL	NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
ticketitem	ID	int	NO	PRI	NULL	auto_increment
ticketitem	UUID	varchar(64)	NO		NULL	
ticketitem	SortValue	int	YES		0	
ticketitem	IsActive	int	YES		1	
ticketitem	TicketID	int	NO	MuL	NULL	
ticketitem	MenuItemID	int	NO	MUL	NULL	
ticketitem	ItemPrice	double	NO		NULL	

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
ticketstatuslu	ID	int	no	PK		
ticketstatuslu	TableStatus	varchar	yes			

Table	**Field**	**Type**	**Nullable**	**Key**	**Default**	**Extra**
tickettypelu	ID	int	NO	PRI	NULL	
tickettypelu	TicketType	varchar(64)	YES		NULL	

Coding

Approach (Functional, OOP)

The approach for this system was fully OOP MVC structure. The original idea was to use models to create the DOM of the page. The models would also validate incoming data and house extracted for consumption. Controllers would be made to provide actions to save data to the database and to facilitate the validation process by the models.

Language

Originally, it was planned to write this subsystem in the Angular framework. Slow progress and setbacks in understanding the framework have put this subsystem on hold until recently. Instead, this subsystem utilizes Bootstrap, HTML, CSS, and JavaScript.

User training

Training / User manual

Training users for this subsystem is supposed to be semi-intuitive. This subsystem is designed to be extremely similar to Point of Sale systems used by many of the restaurant industry leaders.
A user will log into the subsystem and navigate to the point of sales system. The opening screen of the system will show a diagram of tables and their status. Each table will be a clickable button that moves the user to an order screen. The order will allow the addition of

menu items from a menu to be placed on a ticket for that table's customers. Clicking on menu items will add these items to the ticket and a running total will be displayed.
Tickets will not be held in session but will only remain with the table when they are instantiated by sending the ticket to be made using the SEND button.
Items can be added to a sent ticket but will need to be sent again to remain with the ticket.
Cashing out options enable the user to accept payment from the customer.

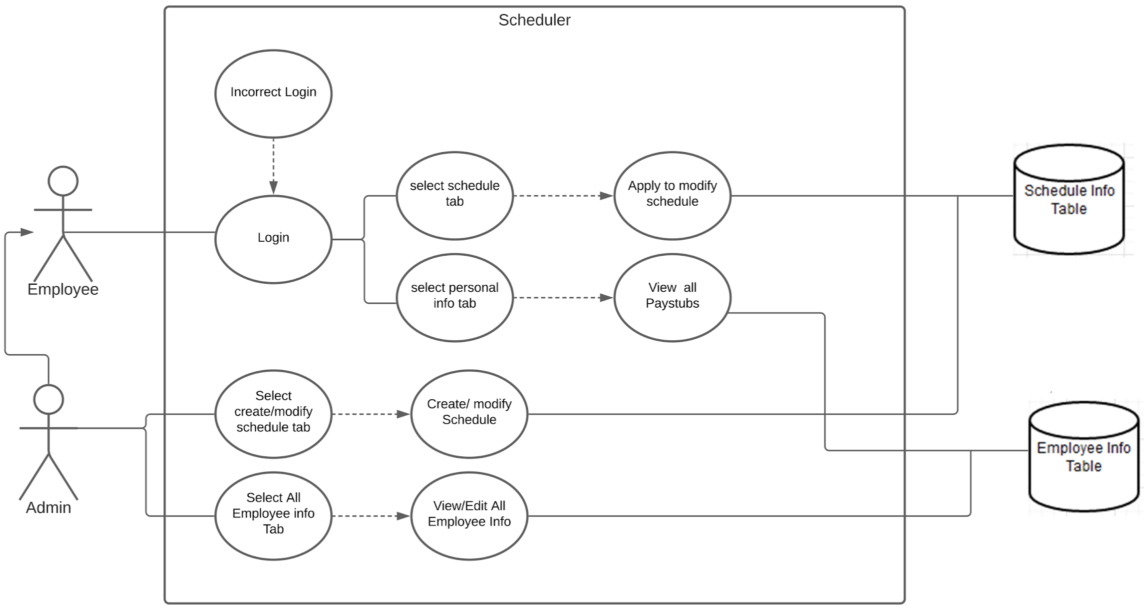
Testing

The testing solution of this subsystem uses PHPUnit to test the functionality of the models and the controllers.

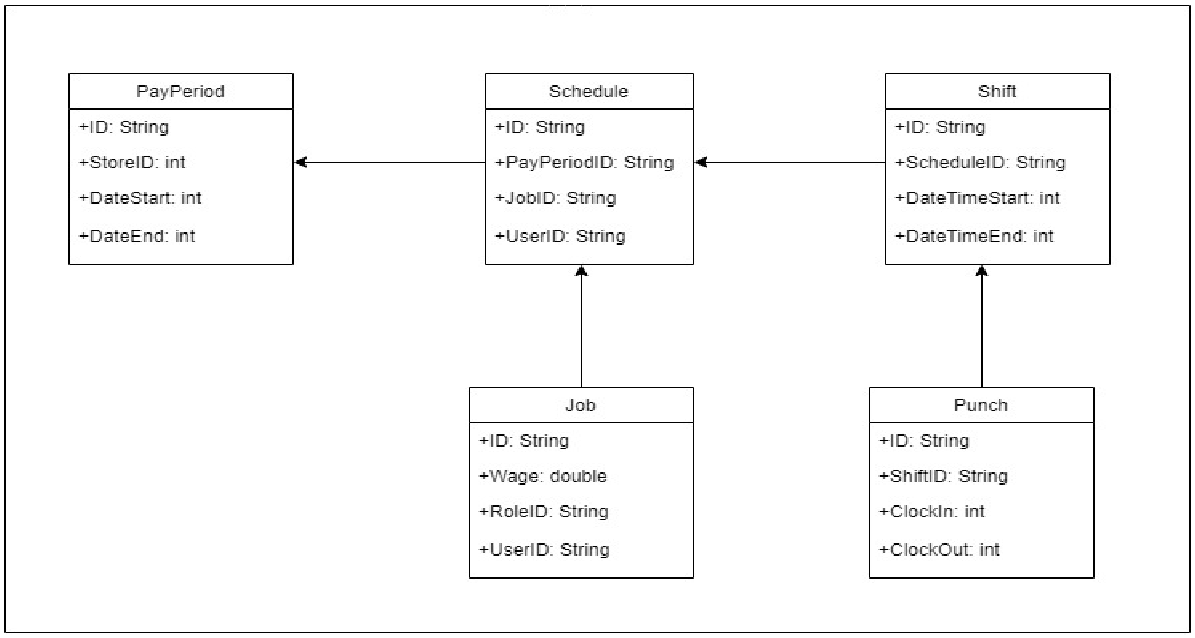
7.5 Scheduling Subsystem – Jason Manning

Design and model

Use Case Diagram



Class Diagram



Data dictionary

Table Association	Name	Type	Description
PayPeriod	ID	String	Identifies the particular pay period
PayPeriod	StoreID	Int	Identifies the store at which the employee is working
PayPeriod	DateStart	int	The time at which an employee starts the shift.
PayPeriod	DateEnd	int	The time at which an employee endst the shift.

Table Association	Name	Type	Description

Schedule	ID	String	Identifies the particular schedule being viewed or edited.
Schedule	PayPeriodID	String	Identifies the particular pay period.
Schedule	JobID	String	Identifies the job the employee is performing for a given shift.
Schedule	UserID	String	Identifies the specific employee on the schedule.

Table Association	Name	Type	Description
Shift	ID	String	Identifies the specific shift an employee is working .
Shift	ScheduleID	String	Identifies the particular schedule being viewed or edited.
Shift	DateTimeStart	int	The start time for a given shift displayed on the schedule.
Shift	DateTimeEnd	int	The end time for a given shift displayed on the schedule.

Table Association	Name	Type	Description
Job	ID	String	Identifies the job the employee is performing for a given shift.
Job	Wage	double	Amount paid for a given timeframe to an employee, such as per hour or per week for salary.
Job	RoleID	string	Verifies the role an employee is performing during a given shift.
Job	UserID	String	Identifies the specific employee on the schedule.

Table Association	Name	Type	Description
Punch	ID	String	Represents the timeclock used by the employee to verify time in and time out for a shift.
Punch	ShiftID	String	Identifies the specific shift an employee is working.
Punch	ClockIn	int	Time at which an employee clocks in for the shift.
Punch	ClockOut	int	Time at which an employee clocks out for the shift.

Coding

Approach (Functional, OOP)

Using Angular to create the front-end, there are aspects of functional programming, but Angular uses mostly object oriented programming, making it more appealing to users of C# and Java. The scheduling subsystem makes use of the object oriented aspect of Angular.

Language

The language of Angular is Typescript, a more robust version of Javascript. Typescript compiles to Javascript, but helps to eliminate errors during the process of writing the code. As we are creating the front-end of the application in Angular, the scheduling subsystem is written in Typescript.

User training

Training / User manual (needed for final report)

Upon logging in, the user is presented with a menu on the left side of the screen. One of the options is Scheduling. After choosing Scheduling, the user will be presented with a set of options dependent on his or her role.

Administrator (owner or manager):

The administrator page will have the following options:

View Schedule
Edit Schedule
View/Edit Payroll
Time Clock

User (employee):

The standard user page will have the following options:

View Schedule
View My Payroll
Time Clock

View Schedule:

The View Schedule option allows the user to view the current schedule.

Edit Schedule:

The Edit Schedule option allows an administrator to edit the current schedule and create a new schedule.

View/Edit Payroll:

The View/Edit Payroll option allows an administrator to see and edit as necessary the payroll records. Options for editing include adding new employees, deleting old employees, as well as adjusting wage information for each employee.

View My Payroll:

The View My Payroll option allows the user to view his payroll data, such as current wage information and paycheck totals.

Time Clock:

The Time Clock option used by any hourly employee to clock in and out of a particular shift.

---

8. Complete System

Final software/hardware product

The final product will be a development version of a Point of Sales Suite that includes a completed API and Database back-end, developmental editions of a Point of Sales subsystem, an Inventory Subsystem and a Scheduling subsystem.

Source code and user manual – screenshots as needed - Technical report

https://github.com/Michael-Calvo/Captsone-Spring2021

Evaluation by client and instructor

Team Member Descriptions

- Richard Abrams
  - The inventory services serve as the communicator to the back-end. It is where all REST API requests are made and any requests from external data are transferred from the service to the requestingcomponents.To help facilitate that, a transporter class was also created to serve as a generic container of data that can be utilized throughout the inventory to receive the data it needs. Utilizing both the inventory components will be to communicate with the back-end and receive and send data as needed.
- Ronald Abrams
  - For the login, the front-end accesses a Google Account using Google Login, and places it in a Firebase storage, and this relates to a user in the back-end, with an Authority Level known as a roleID, which is then used to determine which pages the user can see afterwards. The Google Email, before the @ symbol, is used to find a matching user in the back-end to make this match to the back-end. If the user has a sufficient Authority Level, the user can access a page to also create accounts for GetThisBread through the front-end.
- Emily Ashburn
  - The Point of Sale subsystem facilitates the front of the house sales of this restaurant-based system. The views to be created within this system are mainly for the cashier/waiter/waitress/bartender roles, allowing the order of food off of the business' menus, facilitating purchases, and displaying the view of the house and the available/taken/waiting-on-order seats.
- Mike Calvo
  - Users have been imported through google firebase. From there, their data is viewed on a profile landing page. If the person is not signed in, they cannot access the profiles tab. One there, the user can go back to the home page at any point using the profile tab. User is being implemented with the Database to find a matching user name and return their role and ID. This will be used to show which tabs users can access based on role.
- Jason Manning
  - The scheduling subsystem is used for the time clock function and viewing and creating an employee schedule. The time clock has basic functionality for employees to clock in and out and send the data to the database. The schedule views are separated by employment roles. Employees are able to

    see View Schedule whereas a manager can view Create Schedule and Edit Schedule as well.

- Ian Wilhelmsen
    - This subsystem is a no frills Database and api back-end with protections against sql injection attacks and api abuse. The subsystem supports ETL operations of the rest of the application. New instances of subsystems can be created using this functionality. This release of the subsystem is stable for immediate release and needs of the application.
