**Table of Contents**

1. Overview
2. Linux Server
3. SQL Server Database
   1. Tables
4. Restful Web API
   1. Controllers
   2. Models
   3. Interface Class Repository
   4. Class Repository
   5. Context Model
   6. Swagger Endpoints
5. API Helper Files
   1. Customers
   2. Employees
   3. Orders
   4. Products
   5. Sales
6. Login and Logout System
7. Ordering Systems
   1. Main Ordering system
      1. Menu Options
      2. Cart
         1. Adding Items
         2. Removing Single Items
         3. Removing All Items
   2. Payment Systems
      1. View Order Details
      2. Cash Payment
      3. Credit Payment
      4. Edit Order
   3. Receipt
      1. Email
      2. Print
      3. None
   4. Submitting To API
      1. Create Customer
      2. Create Order
      3. Create Sale
8. Cooking Systems
   1. Assembly
   2. Line Cook
   3. Fryer
   4. Complete Orders
9. Pick Up System
10. Admin Tools
    1. Main Menu
    2. Sales
    3. Products
    4. User
11. Mobile Application
    1. Customer Information
    2. Get Customer Order
    3. Get Customer Payment
       1. Cash
       2. Credit card
    4. Display Receipt
    5. Submitting To API
       1. Create Customer
       2. Create Order
       3. Create Sale
12. Finished Product vs Original Idea
13. References
14. Credits

**Overview**

Our point of sale system was design for a fancy burger shop similar to Five Guys or Red Robin. The system can be broken down into tens parts. A linux web server hosts a SQL database for which data is stored in multiple tables. The server also hosts a Restful Web API that allows for sending and receiving of data to the database. A set of API helper files were designed to allow the streamline of interaction between the API and exe programs, as well as the mobile application. The main exe systems consist of the main ordering system which includes the payment system and receipt system. The secondary systems consist of the line cook system, frier cook system, and pickup system. Finally a mobile application that allows user to order and pay for food on their own time. The entire project is built upon the C# and the .Net framework. This project has been designed and implemented by Team Tripod + 1 for their senior project.

**Linux Server**

We chose to host the linux server on a cloud computing website site called Linode.com. Linode is very similar to Amazon's AWS service EC2. The nice thing about Linode is that it cost very little for a decent server for most applications. It cost 10 dollars a month for a single instance at the following specifications. We’re hosting both the SQL server and the Restful Web API on one server instance. The specifications of the server for our needs are as follows. 1 CPU and 2 GB's of RAM as SQL server requires 2 GB at minimum. The specs can be scaled higher if needed.

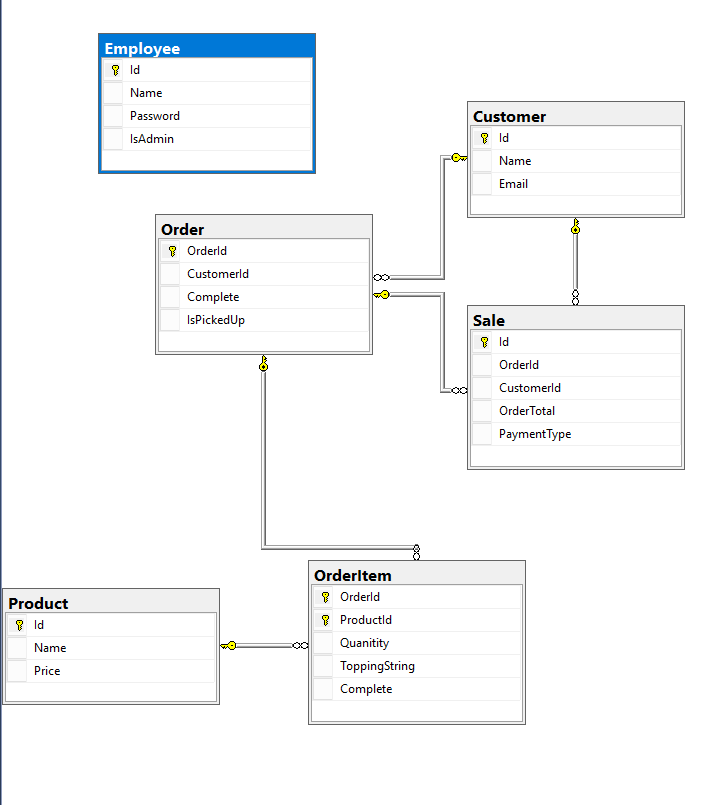
The linux version we are using for the cloud server is Ubuntu 16.04. It is a stable release of Ubuntu which has been out for a long enough time that it works great to fit our needs[9].

**Backups**

The cloud server instance is configured to have a daily backup and a weekly backup every Wednesday to ensure stability and reliability. This enables us if something was to go wrong on the API to spin up a working version of API / SQL server with very little data loss. There is also a manual snapshot of the instance working in the case that the others would fail as well.

**SQL Server Database**

The database was designed and implemented in SQL Server 17. The database consists of sixs tables to store information needed by the software[5].



The **Employee table** stores information for employees of two groups. The first group are employees that take orders or make the food. The second group is for management that has admin privileges to make changes to certain areas of the database. Each employee consists of Unique ID, a Name, a Password and whether or not the employee has admin privileges.

The **Customer table** stores information for customers who had made an order either with an employee or through the mobile application. Each customer consists of Unique ID, Name, and email. The Customer table shares a

relationship with Sale table and Order table.

The **Sale table** stores information related to all sales at the restaurant or the mobile application. Each sale consists of Unique ID, an Order ID which connect to Order table, a Customer ID which connects to the Customer Table, an Order Total, and a Payment Type. The Sale table shares a relationship with the Order table and the Customer table.

The **Order table** stores information for all orders created. Each order consists of an Unique ID, a Customer ID which connects to the Customer Table, a Complete status which keeps track of whether the whole order is complete, and a IsPickedUp status which keeps track of whether an order has been picked up by the customer yet. The order table also shares a relationship with the OrderItem table, the Sale table, and Customer table.

The **OrderItem table** stores information related to an item in an specific order table. Each order consists of an OrderID which connects to the order table, a ProductID which connects to the product table, a Quantity of how many of the certain item has been order, a ToppingString which stores topping information that the customer has requested, and a Complete status which keeps track of if the item has been completed.

The **Product table** stores information about each product that restaurant

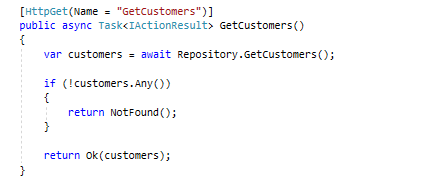
makes and sells. Each product consists of an Unique ID, a product name, and a price for the product. The Product table shares a relationship with OrderItem table.

**Restful Web API**

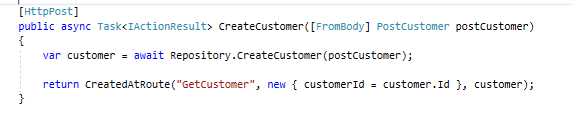
The **Restful Web API** allows accessing our database and making changes easy and eliminates a lot of code that would need to be written multiple times. It also defines what our entity framework database object look like and how we can interact with them. The API is made up of four main parts the **Controllers**, the **Models**, **Interface Class Repository**, and **Class Repository**. We also created a **DLL** extension library, so that moving our objects and functional to another program like API Helper Files could be done with ease. The API makes use of modern **HTTP Requests** and implements **Get**, **GetAll**, **Put** or **Update**, **Create** or **Post**, and **Delete.** Finally we used Swagger to make a

physical location or an endpoint on the on web found at <http://45.79.193.33:5000/swagger/index.html> to view the HTTP Requests that we make to the database[3][8].

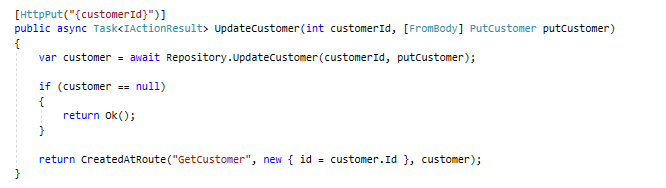
**Controllers** class of the different class objects allows for the definition of the HTTP Route paths and define the naming convention and format on the Swagger Endpoints. The current controllers available to be used are **Customer Controller**, **Employee Controller**, **Orders Controllers**, **Product Controller**, and **Sale Controller**. Each controller also handles all of the HTTP Messages that might occur whether they are successful or error messages.

Customer Controller Class Get code in image below

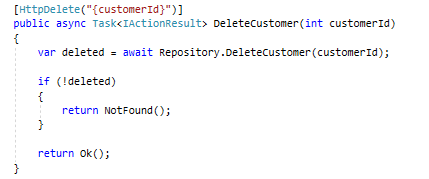
Customer Controller Class GetAll code in image below

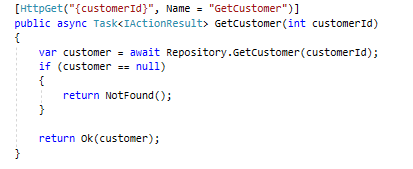


Customer Controller Class Post code in image below

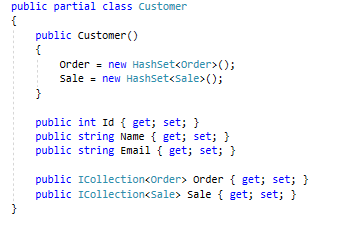


Customer Controller Class Put code in image below

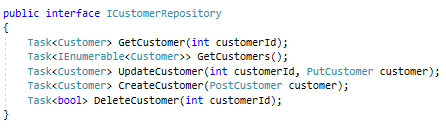


Customer Controller Class Delete code in image below

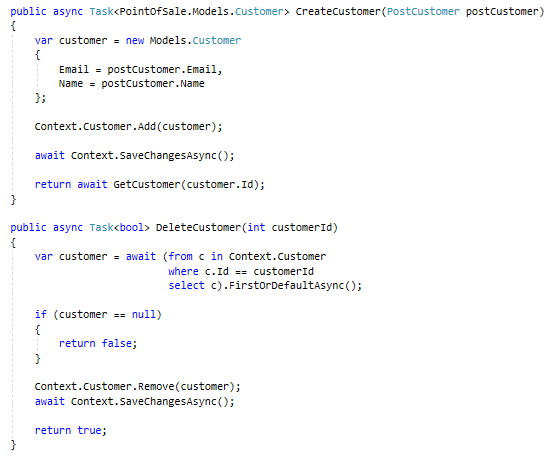
**Models** class define how each object in the database should be represented in code. The models class also defines the relationship that the object will have with other object much like how the tables share relationship with one another. This is done by making use of .Nets [1] Collections allowing us to define related object as an ICollection. In the the image below you can see this in action. As well as the properties that are defined for the Customer object which are the same as the database table Customer.



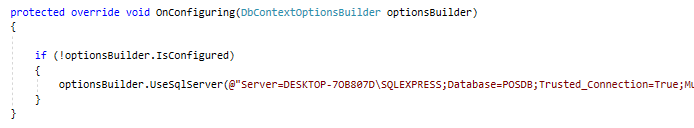
**Interface Class Repository** are a set of interface classes that define the basic structure of how the data will be able to be used with HTTP Requests. In the image below you can the see the basic structure of how the customer class is going to be implemented and defines what data types each HTTP Request will need.



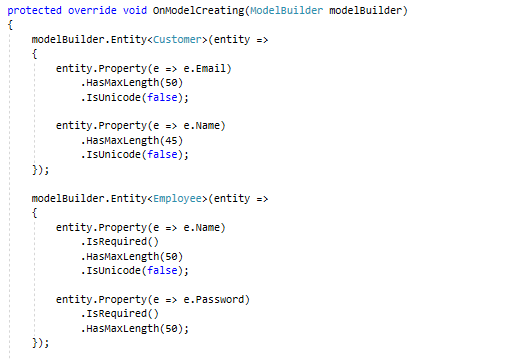
**Class Repository** are the implementation of the Interface classes. Each function in the class is defined as async functions allowing for the program to send data back regardless of the orders in which the requests came. This allows for a quick response time since it doesn't have to wait and returns the first item asked that it found first. The image below show the functions CreateCustomer which creates an object and stores it the database and DeleteCustomer which deletes a customer object from the data based on the ID given from the user. The DeleteCustomer statement uses a Linq statement to find look for the customer with ID given.



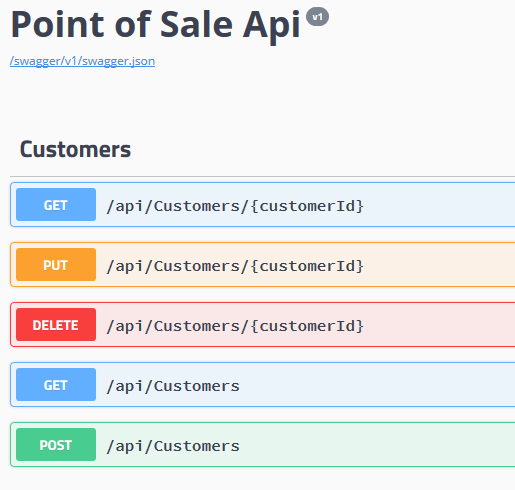
**Context Model** is the main model that defines the model of the database and its tables which then connects and creates the object models that were talk about above. It also store the address and how to connect to the database including passwords and settings. The image below show the implementation of this.



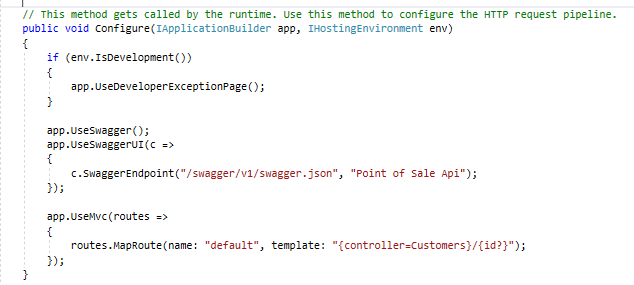
The image below shows the creation of models customer and employee. The statements below show the properties that the database has defined for each column.



**Swagger** is a third party tool that allows users to have a physical place or endpoint to interact with the API. It allows for testing and changing of data to make sure that the data is sent correctly to the database and the objects are defined correctly and won't break. The image below shows the customer HTTP Request and route that we can use to send messages to update the database with[8].

****

The image below is how swagger services are implemented in the API code.



**API Helper Files**

The API Helper files are made up of five different classes, the **Customers** class, the **Employees** class, the **Orders** class, the **Products** class, and the **Sales** class.

Each class streamlines the format and data needed to be sent or received from the Restful Web API. They call all user Newton.Json [4] to deserialize the information from the Web API and creates an object related to the class. These objects can then be used in the different parts of the exe programs or the mobile application allowing for easy manipulation and formatting of data that needs to be stored in the database. Each class checks to see if the information is in the correct format to be sent to Web API making sure not to send wrong data types or objects types.

The classes all have implantation of the different http request that the Web API offers.

**Get(ID)** returns a single object; based on class used , if the ID sent is found.

**GetAll()** returns all objects found based on class used.

**Delete(ID)** allows the user delete an object based on the ID that the user provides.

**Create(object information)** allows the user to create a new object to be sent to the Web API to be stored in the database. The object information varies depending on which class the user is wanting to create.

**Update(ID and object information)** allows the user to update an object based on the ID that the user gives. The object information varies depending on which class the user is wanting to change. The user also has option to change all parts of the object except information that relates to object ID that share relationship with other objects.

The **Employees** class has a unique feature that allows during the creation of new employee object that the password are hashed before being sent to the web. Wireshark can be used to see the hash and not the password that user created. The employee class also dehash passwords that the user request to check against store information from the exe programming that want to allow a user login.

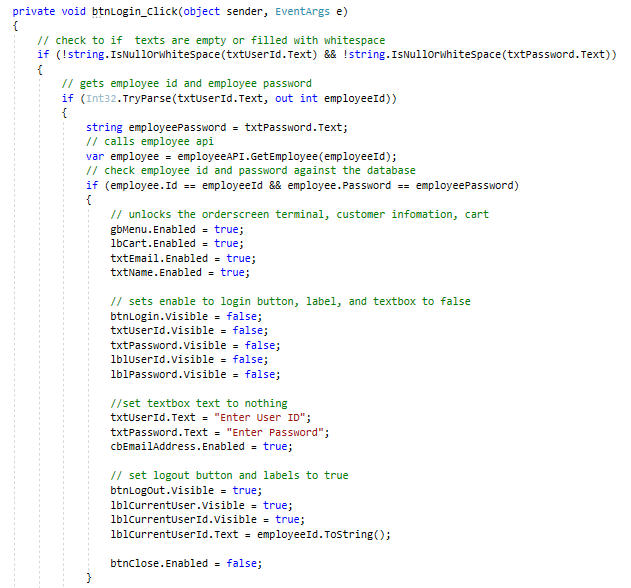
**Login and Logout System**

The **login and logout system** consists of a four part system to ensure that only user that is allowed to be on the system has access to the different terminals.

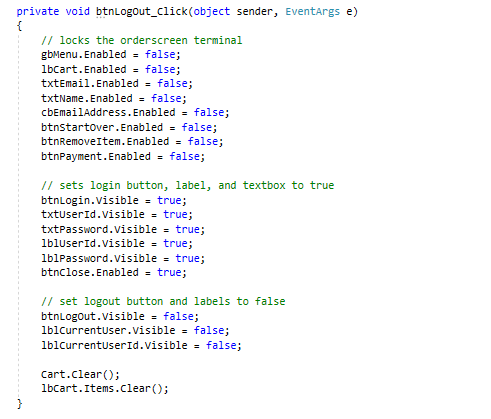
The **first part** takes in user ID and password, making sure that information that the user has entered is in the correct formatted. It validates the user ID is not null or has blanks and only consists of whole numbers. It also validates that the password is not empty or null. If any of the information that the user inputs is in the wrong format, the system will alert the user with a message box stating that information has been entered incorrectly and to please try again. Then the information is stored in variables to be sent out to the API helper file called employeeAPI.

The **second part** takes the stored variables that were sent to employeeAPI helper file and asks the API if a user ID exists and if the password matches. The password that is sent over the web is always hashed before being sent and unhashed in the API helper files. If the password and user ID are not a match then the system will halt and send an error message to the user stating that they have entered incorrect information.

The **third part** only works when the user successfully logs in. This part allows the user to click buttons on the screen whether to take orders or to complete orders along the food making process at different terminals. The login button and text boxes disappear from the screen and logout button appears to allow the user to log out of the current system. The system also displays who the current user that is logged in and working from the terminal.



The **fourth part** allows the user to logout and close the terminal that they were working or leave the terminal at the login screen for the next user to login. At logout, the system clears any orders that were being taken or being worked on by the user. It also disables all buttons on the screen, hides the logout button, and redisplays the login button and text boxes for the next user.



**Ordering Systems**

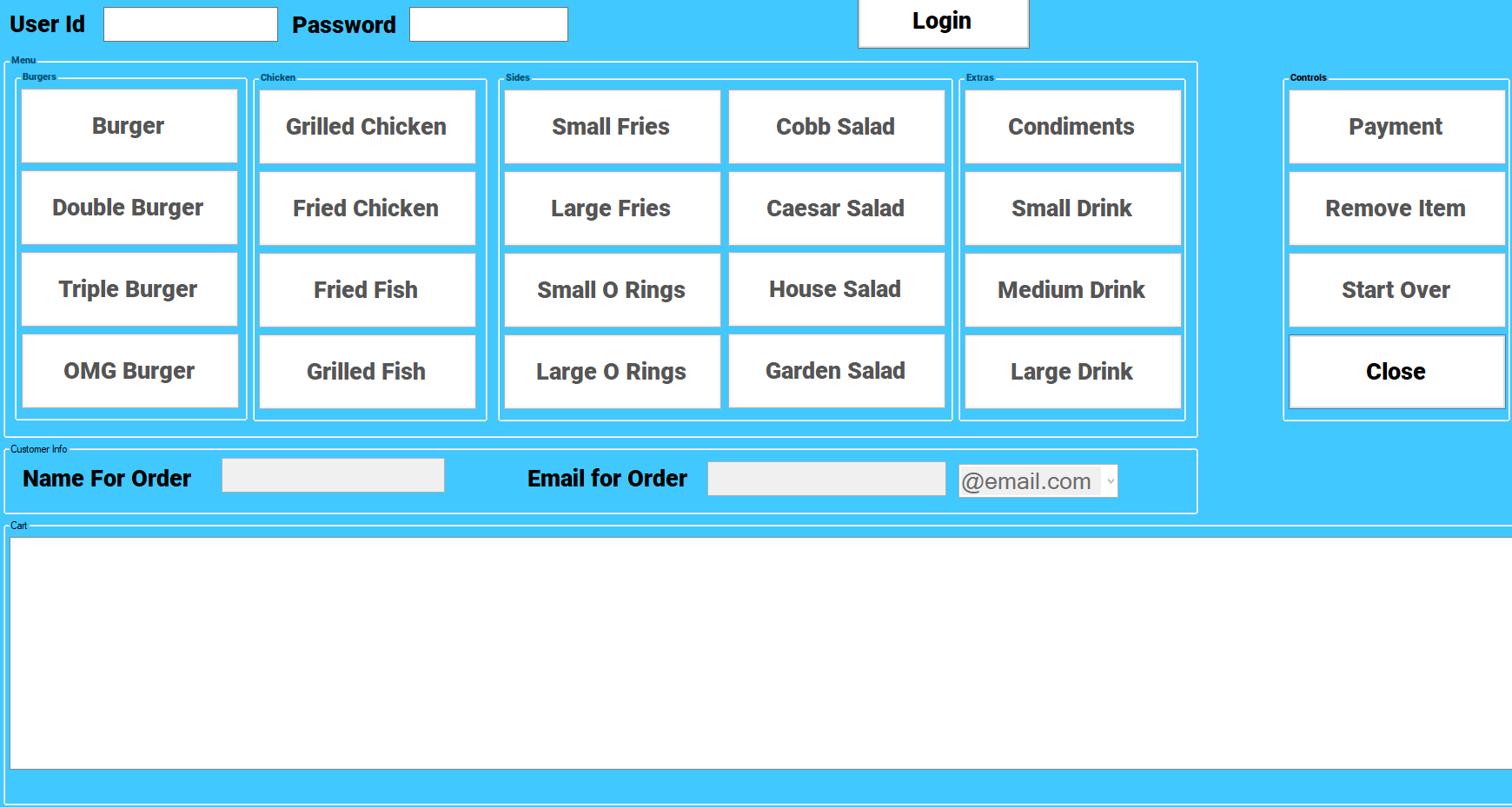
The **Ordering System** consists of four main parts the **Main Ordering System**, the **Payment System**, the **Receipt System**, and **Submitting to the API**. This system was designed to be used by employees working the counter taking orders from customers. It was design to be simple and easy to use by anyone after brief training. The system first requires the user to log in and then gives the user different option to choose from for taking the order. The next step is taking a payment from the user. The system is setup to accept cash or credit cards. No personal checks please. The third step is getting the customer their receipt. The final step in the system is to submit the order to the API which stores it in the database.

**Main Ordering System**

The main ordering system consists of two major parts Menu Options and the Cart.

The Menu Option which can be seen in the image consists of many buttons to choose from depending on what type and size of the food item that customer is wanting to order. After clicking the options button the user is then displayed a submenu that has different option and extras that can be added to the product, as well as the quantity of the product. The user then must either add the item or hit the back button before the system will let them interact with any other menu options.

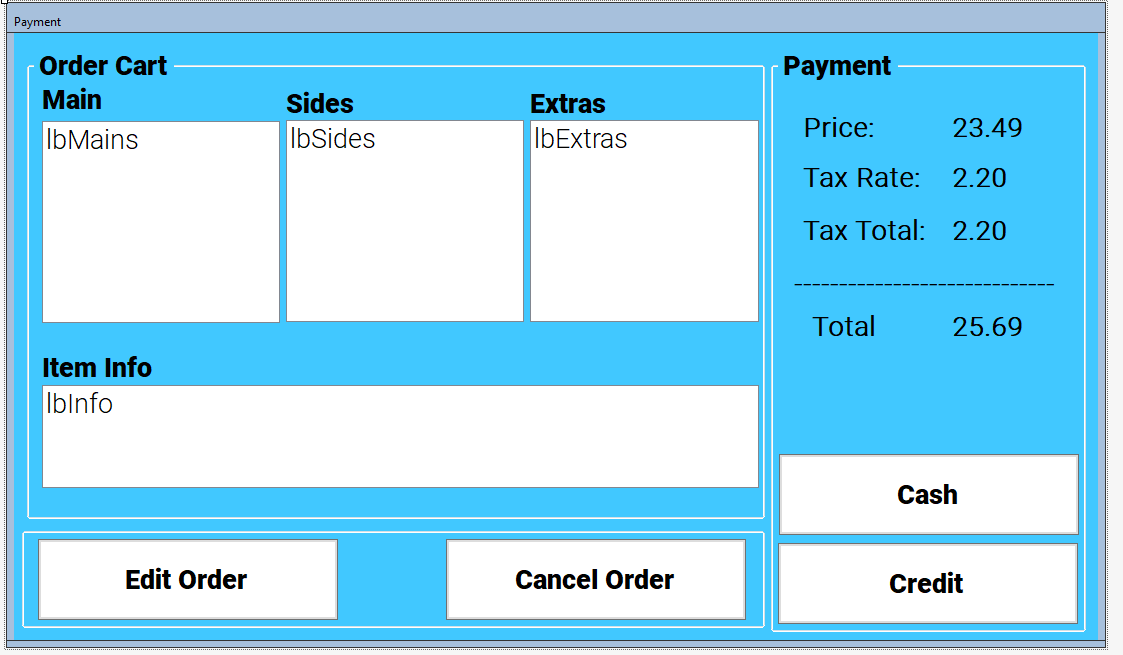
The Cart is made up of two parts. The first part is visual representation that is a display of the item that have been selected by the user. The second part is a few functions that take care of adding and removing items to the object that will be sent to the API when finished. The main features of the cart is the ability to add items, remove a single item or remove all items and start over. Theses abilities are the same for the code cart and the visual cart. The system will not move forward until the user has added at least one item to the cart and has taken in a name and email for the order to created under. This customer information is stored and will later be sent to the API.



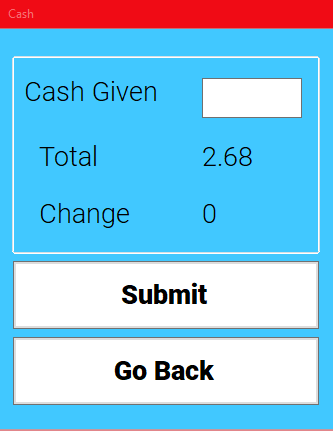
**Payment System**

The Payment System consists of four parts the **View Order Details**, the **Cash payment screen**, the **Credit payment screen**, and the **Edit Order**. The system will display the total cost of the order before taxes, after taxes, and the tax rate which will be charged to the customer.

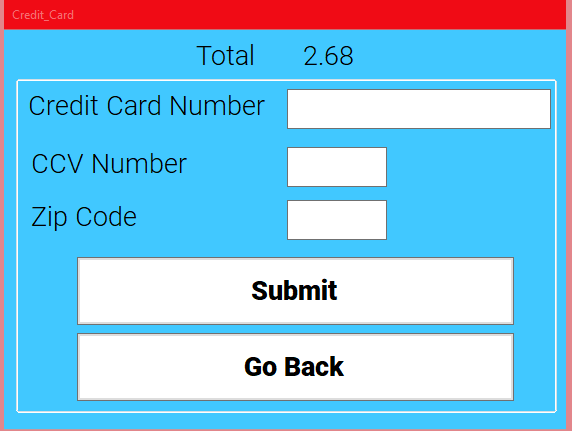
The **Order Details** are displayed across three different list boxes depending on what type of product is in the cart. Any main items like burgers or chicken sandwiches will be display in the Main listbox. Any items like drinks, salads, and fries will be displayed in the Sides listbox. Any items like sauces or condiments will be displayed in the Extras listbox upon clicking any of the items in the three listbox listed above the Item Info listbox will display information about that certain. The details could be the toppings the customer want or the size of the product or the different meat types.



The **Cash Payment** screen is displayed when the user clicks the Cash button. In the cash payment screen the user can enter the amount that was given by the customer. The system will then display how much change to give back to the user. From this the point the user will either submit the payment or click the go back button to select a different payment option.The payment information is stored to be sent to the API later.



The **Credit Payment** screen is displayed when the user clicks the Credit button. In the credit payment screen the user can input the customer credit card number, the CCV number, and zip code. Each text entry has error checking to make sure that the information entered is the correct format and data type. The system will check the credit card number to determine which type of card it could be like mastercard, discover, american express, and visa. Then the user can submit the payment or go back to select a different payment type. The payment information is stored to be sent to the API later.



The **Edit Order** has two options to either cancel order or edit order. If cancel order is select the user is returned back to the main ordering screen and the cart is clear and customer information is erased. If edit order has been selected then the user return to the main ordering screen and is able to change edit the order once again.

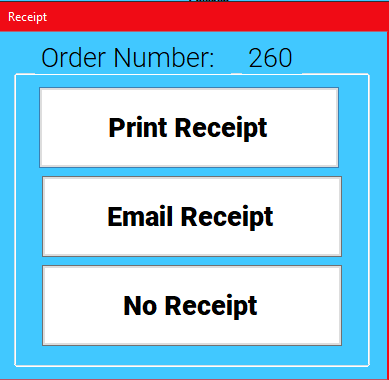
After completing the customer payment the system will go to next part which is the receipt system.

**Receipt System**

The receipt system is made up three options to either email, print, or not get a receipt. Once one of the options are selected the user is then returned to the main ordering screen. Selecting none will skip over any other processes in the receipt screen.

The email option, if selected by the user, will send an email custom to the customer’s email address give earlier when ordering. In the email the customer will receive the information from order and the total. The email system is setup through gmail [12] as the user [orderingsystem.usf@gmail.com](mailto:orderingsystem.usf@gmail.com). If the customer has blocked this email address then they will not receive an email.

The print option, if selected by the user, will currently not do anything other close the receipt screen and continue on. In the code there is a spot to connect to the external receipt printers. However since we do not have a receipt printer the code was not written, but a dummy code for opening windows print menu is there instead.



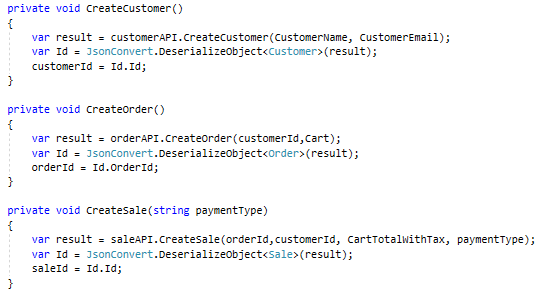
**Submitting to the API**

The final step in the ordering system is submitting to the API. At this setup all the information that has been stored along is packaged and sent off to the API and then submitted to the database, allowing for the other exe programs to get the orders and start working on them right way. The API can handle many orders coming from different sources at the same time since it was design to be an async system, meaning that it does matter what order the API receive data but will work on anything that is available to be submitted. If larger orders slow down the system, the API would submit orders in between submitting parts of the large order.

The **first step** of submitting is using the CreateCustomer function to create a customer object to be sent to the API. To create a customer object, a customer name and email must be given to the CreateCustomer function. The CreateCustomer function is a part of the API helper files and does all the work in the background, actually sending it to the API and checking to see its the correct format. After submitting the customer object a customer ID is return so that it can used later.

The **second step** of submitting is using the CreateOrder function to create an Order object to be sent to the API. To create an Order object we use a customer ID which we received when creating the customer object and List<ProductInfo> called Cart. The ProductInfo is the class that the API helper files gives to the user. ProductInfo objects are created when the a product is select and stored in the cart. It consists of a product ID, Toppings that customer wants, and the quantity of that certain type. The CreateOrder function is a part of the API helper files and does all the work in the background of actually sending it to the API and checking to see its the correct format. After submitting the order object an order ID is return so that it can used later.

The **third and final step** is using the CreateSale function to create a Sale object to be sent to the API. To create a Sale Order we use an order ID which we received from creating the order object, a customer ID which we received from creating the customer object, the total of order with tax, and how the customer paid for the order.The CreateSale function is a part of the API helper files and does all the work in the background of actually sending it to the API and checking to see its the correct format. After submitting the order object, an order ID is return so that it can used later.



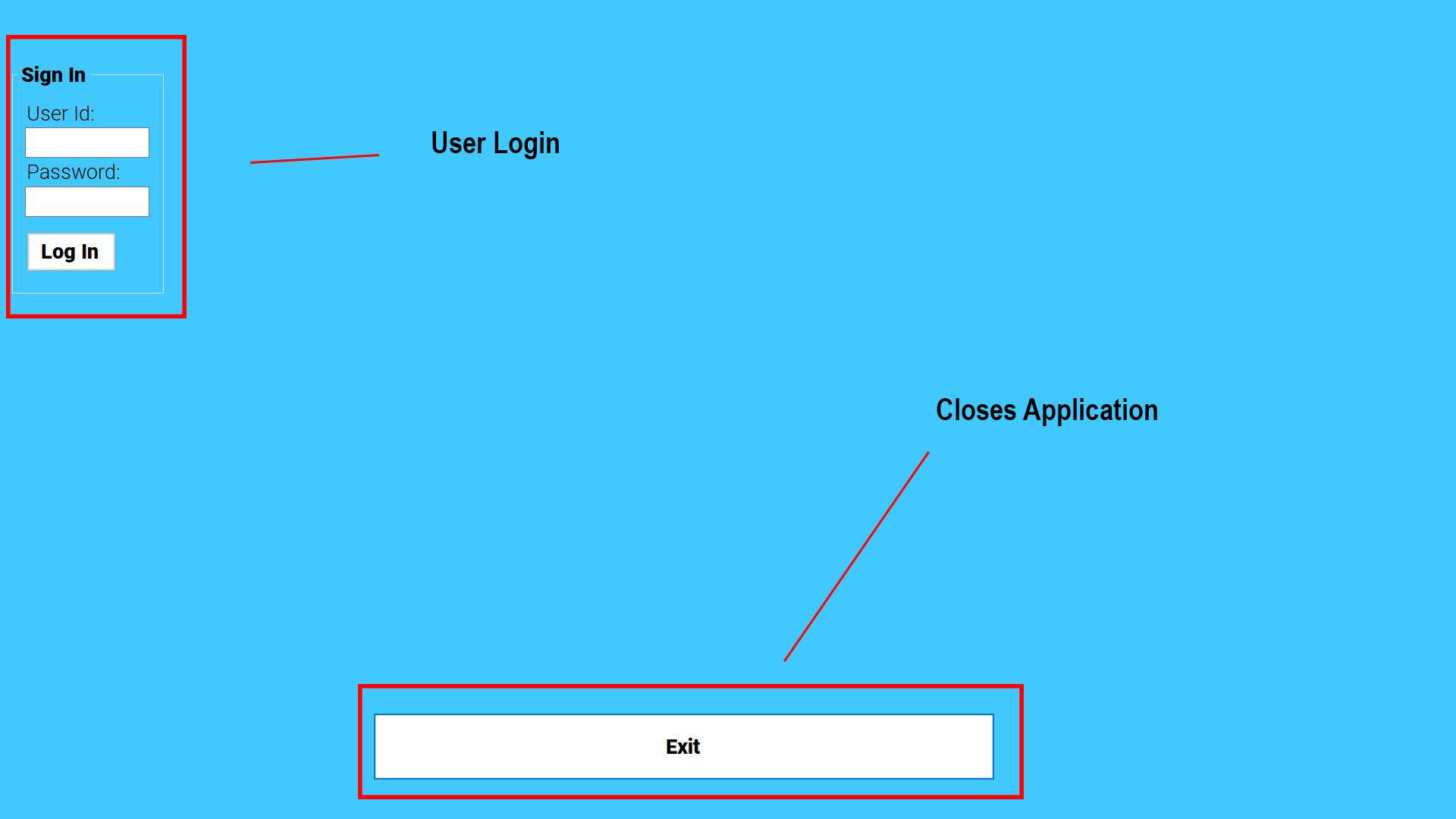
**Cooking Systems**

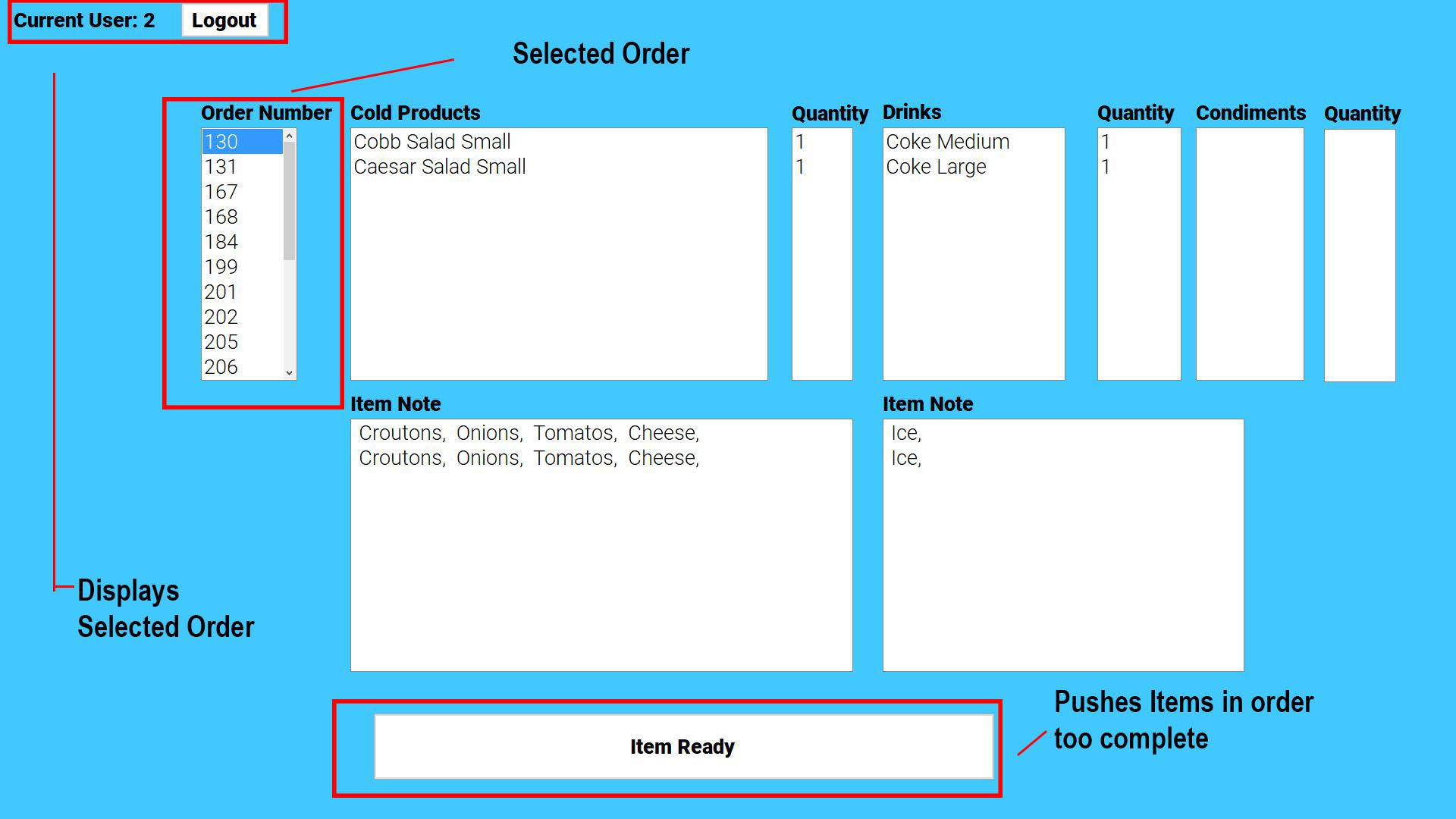
The **Cooking Systems** are divided up into four main parts as you will see in the following paragraphs. The cooking systems divide specific items into different screens for specific employees to fill those specific parts of the order. Each screen will show the order number and the employee can click on the order number for the screen to display what needs to be filled for that order once an order has been completed in lets say the Assembly Screen (cold foods) The employee presses order ready. This flags the API that those specific items in that order related to cold foods in it are flagged as true.

Note\* all cook systems use the API helper files in order to streamline getting the formation from the API specifically to server information.

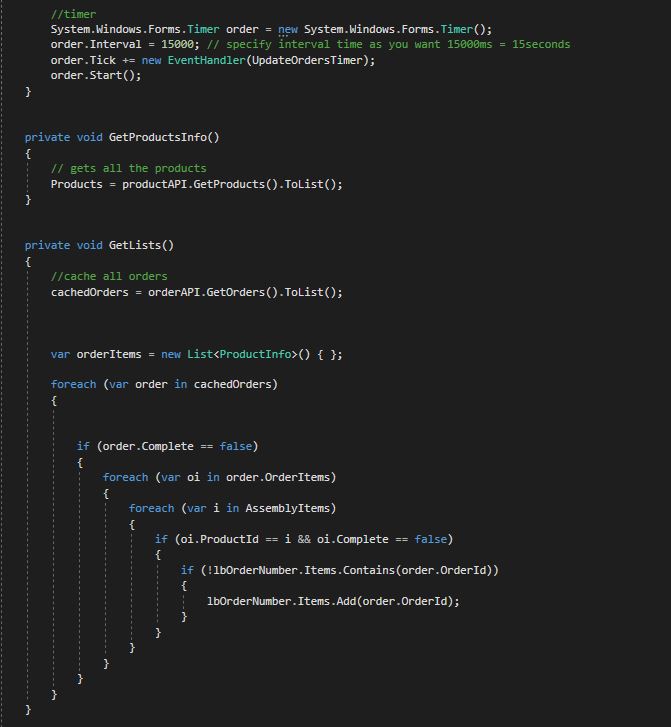
**Assembly Screen**

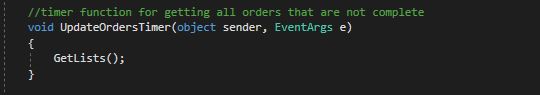
The assembly screen shows all orders that are not complete and have items related to cold products. For instance, salads, drinks, condiments, etc. Once these these items have been filled for that specific order they are then marked as complete when the employee presses order ready for that specific order number. It is then cleared from the Assembly screen.





**Assembly Screen Sample code:**

****



**GetLists()** is designed pull all orders from the Database where they are not complete

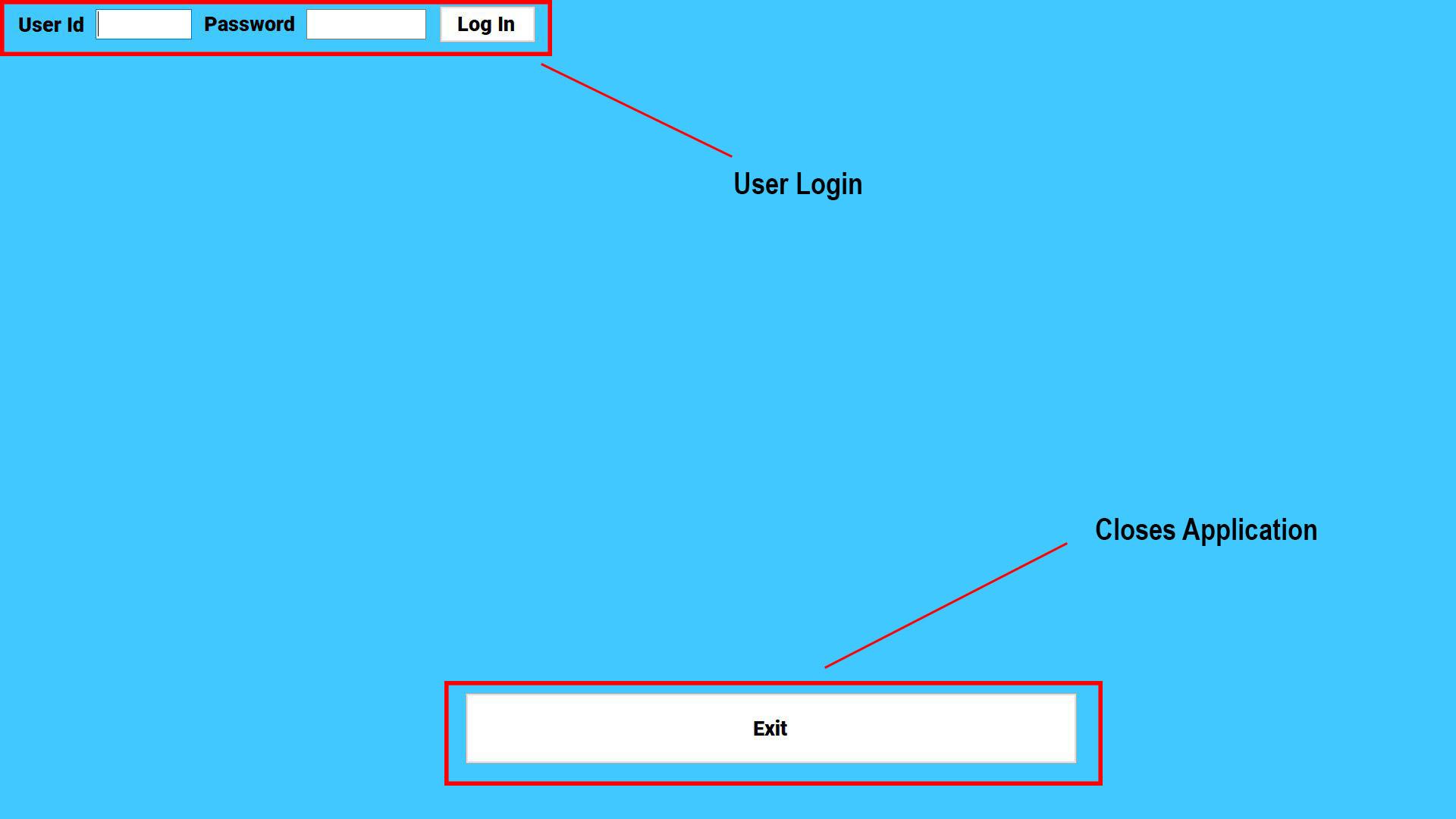
**UpdateOrdersTimer()** is used to refresh the list of orders from the database as they are updated. This function is called every 15 secs by a timer.

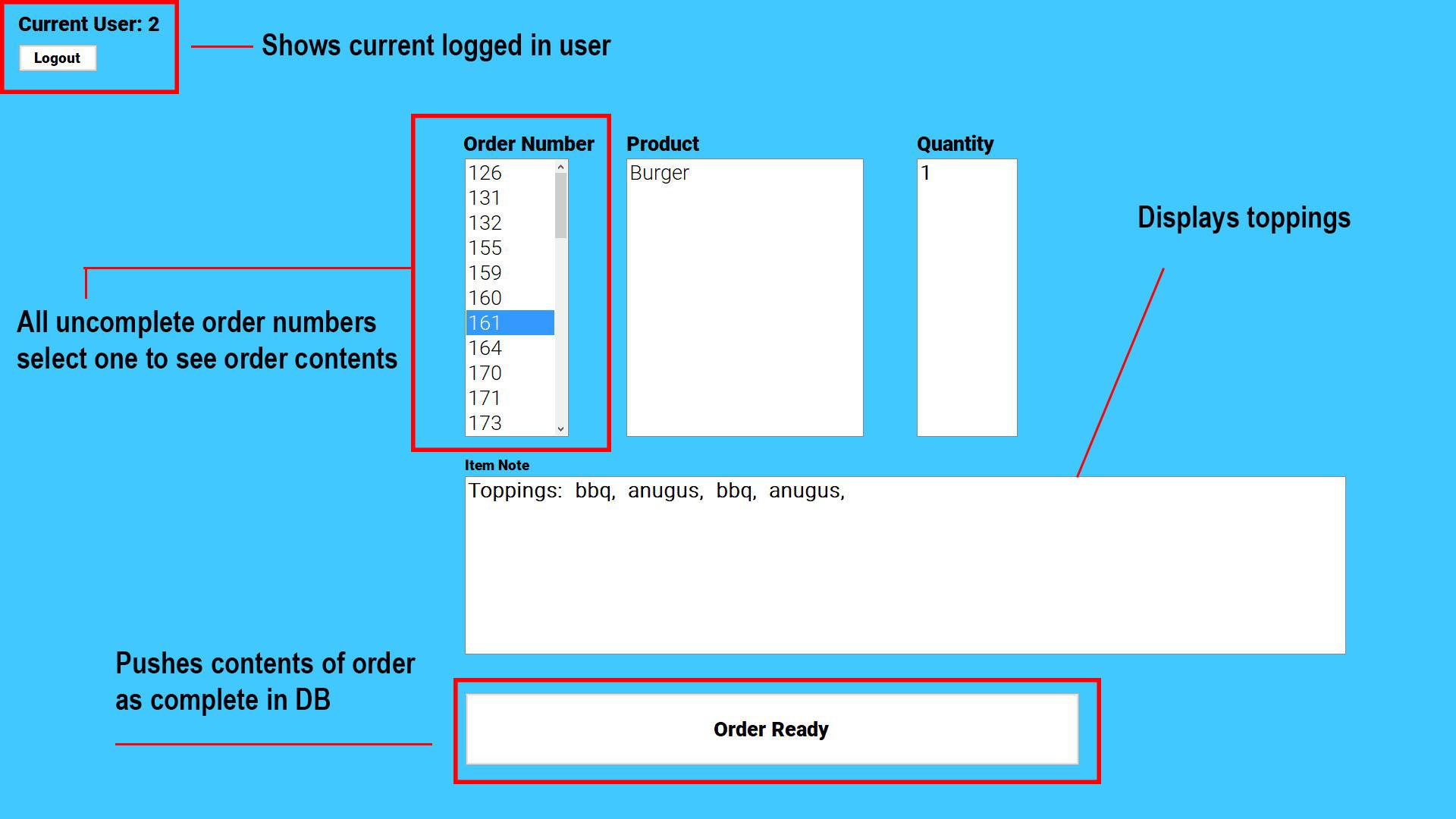
**System Timer** was put in play to update orders every so many seconds. Its set to 15000 ms which is 15 secs. This timer is used in all 3 of the Cook systems and also in the complete orders screen.

**\*Refer to login logout system** for information on how it works. It’s used in all screens

**Line Cook (**Grill Cook)

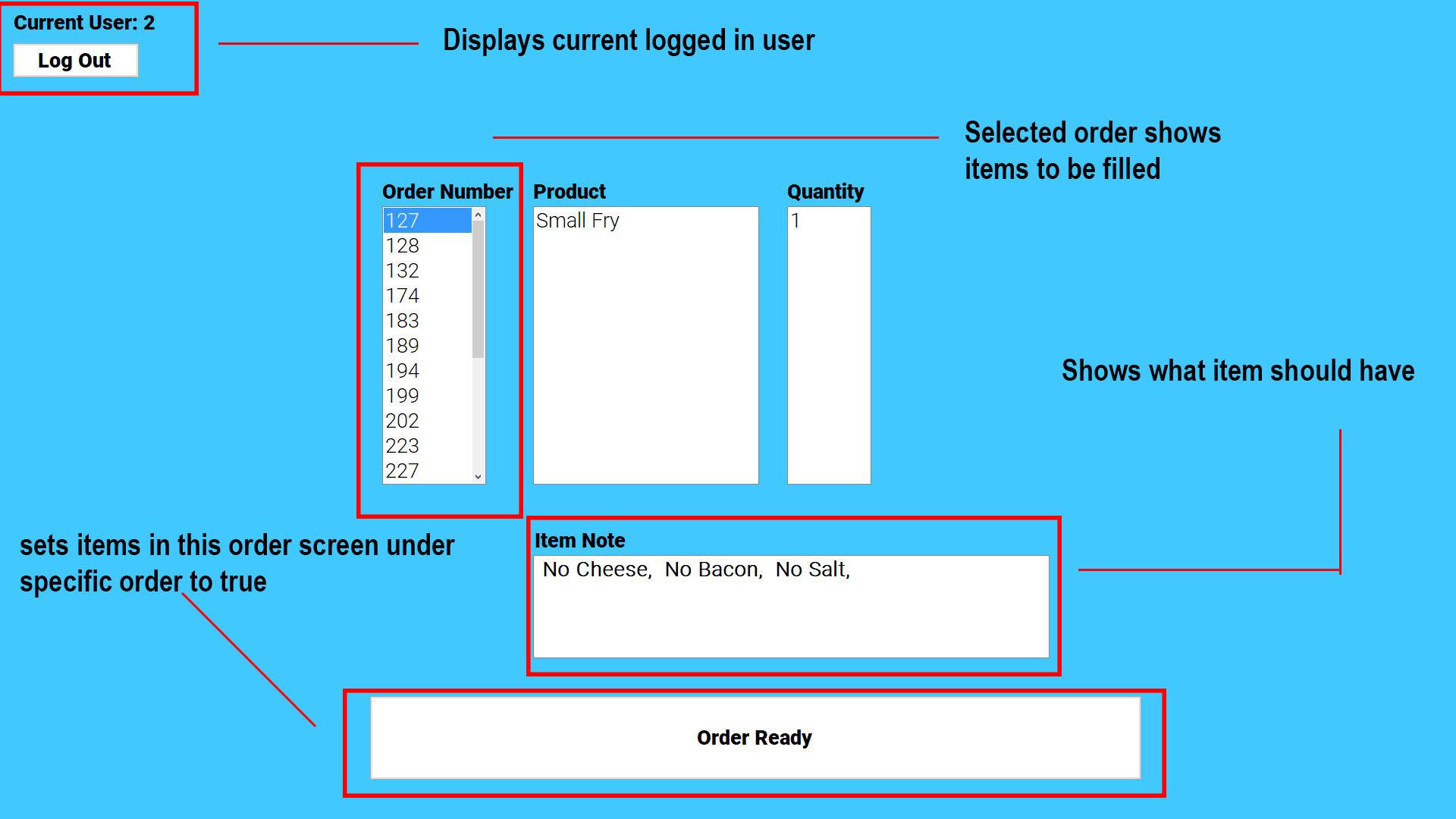
The line cook screen shows all burgers and items related to the grill. As these items in each order are complete they are then marked ready by pressing the order ready button.





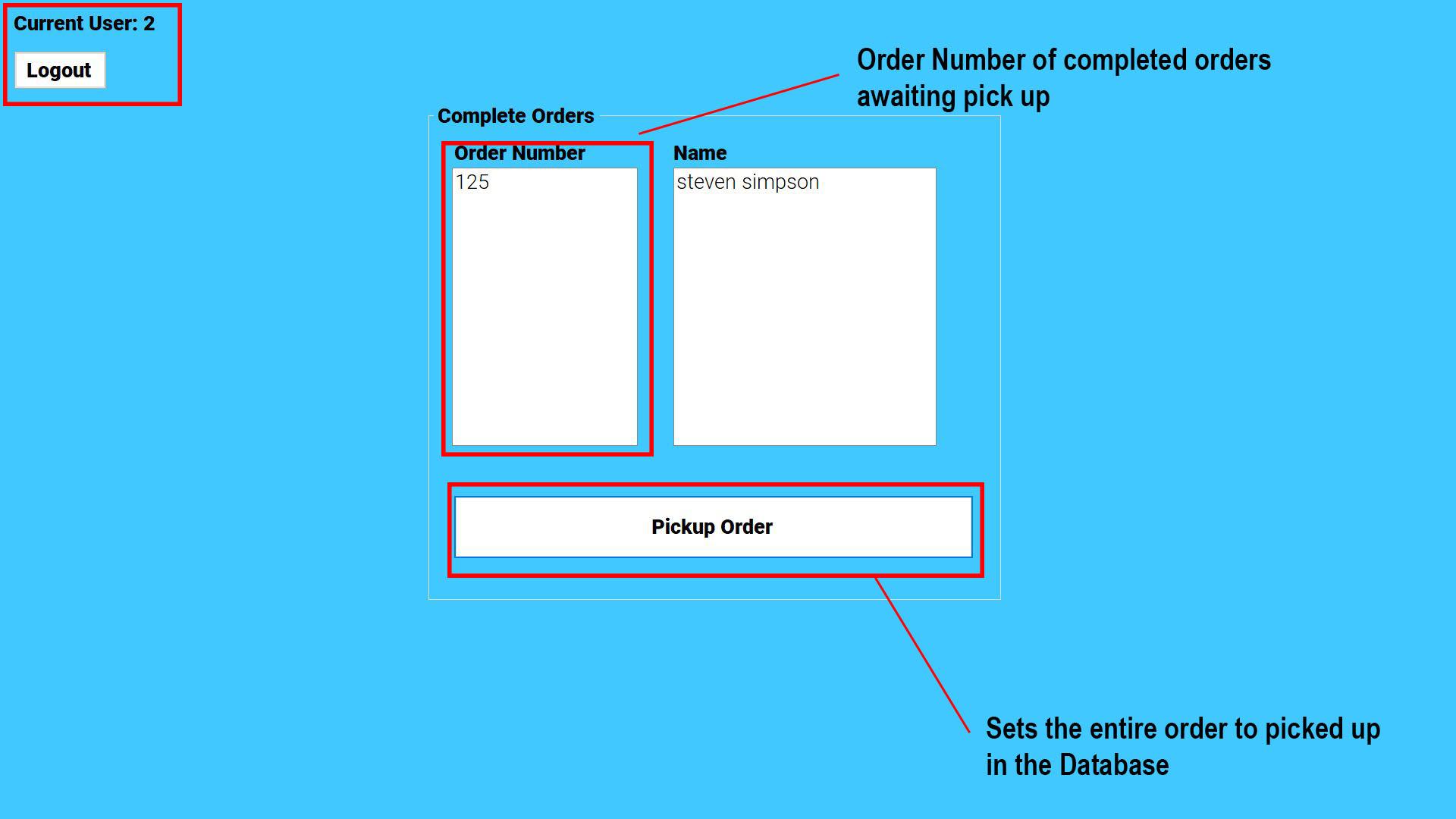
**Fryer**

All fried foods go through this system. Same thing as the other two screens once an order number that has fried foods is complete. The order ready button is pressed flagging all items that are in the menu for that order as complete in the database.



**Complete Orders**

Once all items in a order are complete they show up here. This is where the order is pushed as picked up each time a customer receives their order letting the system know that this order is done and removed from the complete order screen after it has been picked up.

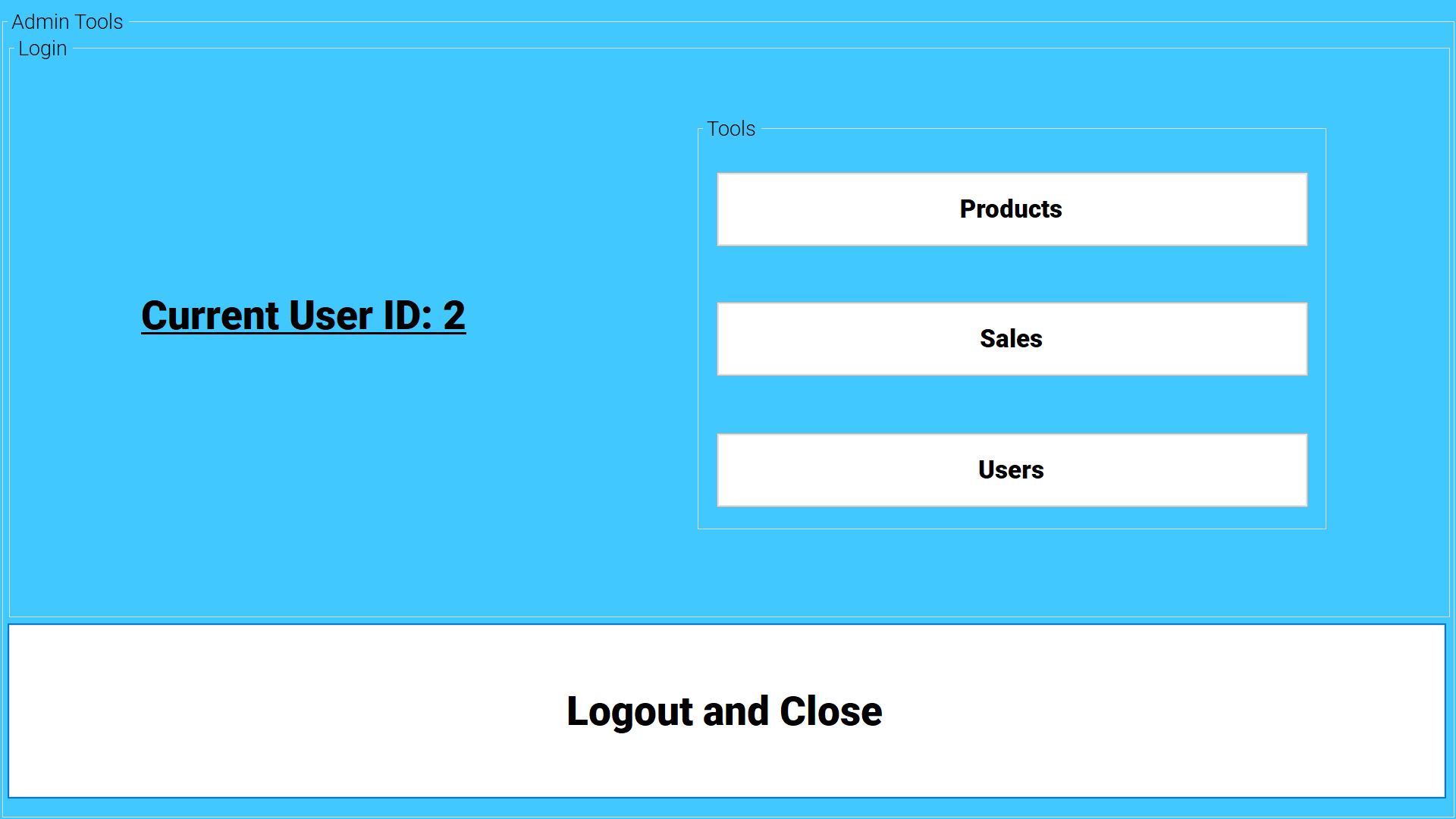


**Pick Up System**

Once all parts are done in each cooking system the order is then flagged as complete in the database. At this point the order number shows up in the complete order screen to be picked up. Once the order is picked up the employee or customer clicks order picked up and the database gets updated for that order making the order IsPickedUp from false to true clearing it from the cooking screens and the pickup screen all together

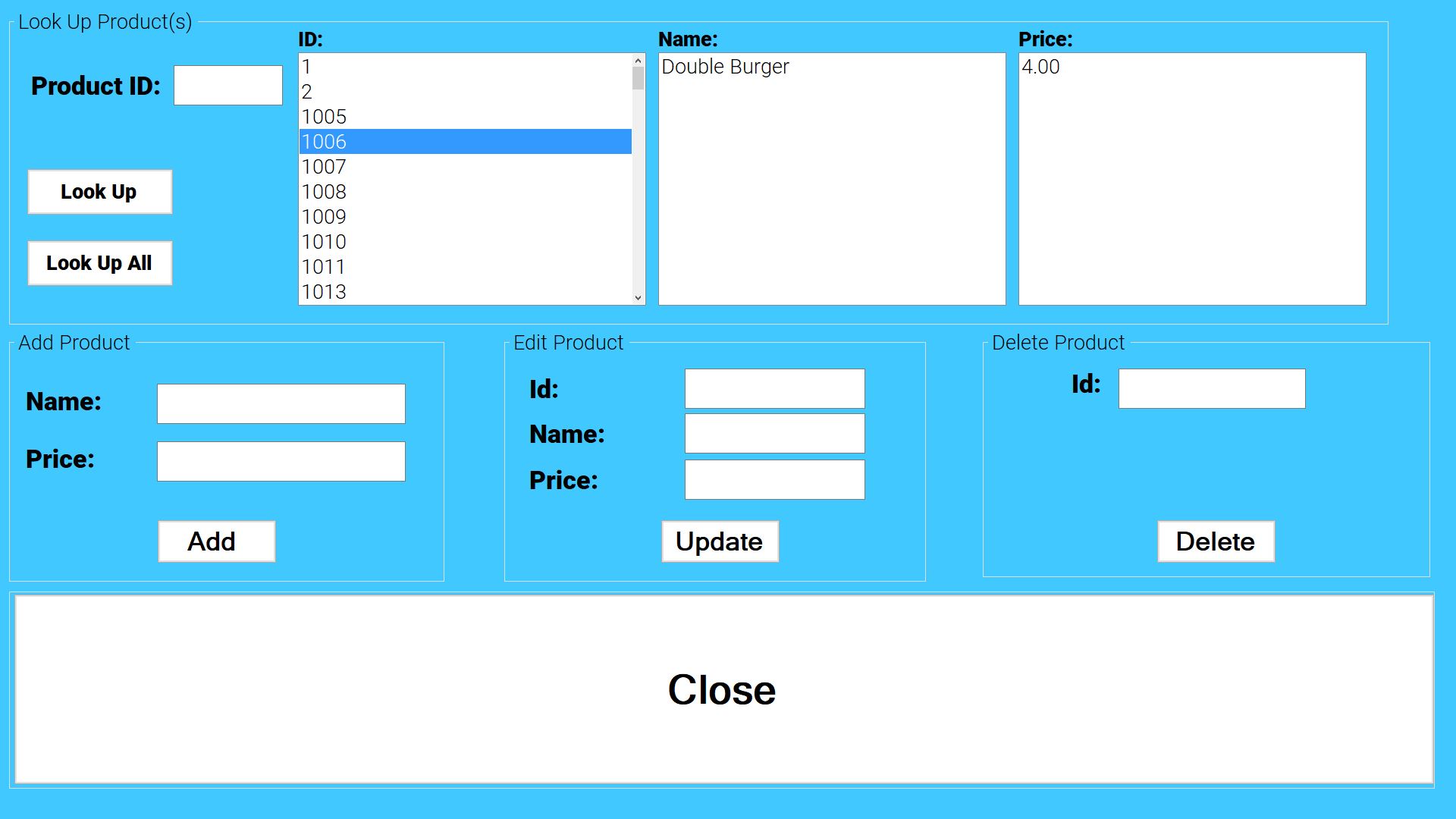
**Admin Tools**

The **Admin Tools** starts with you having to login before you can a have access to three sub menus. If you are marked as an admin in the **Employee Table** then you will have access to these sub menus.



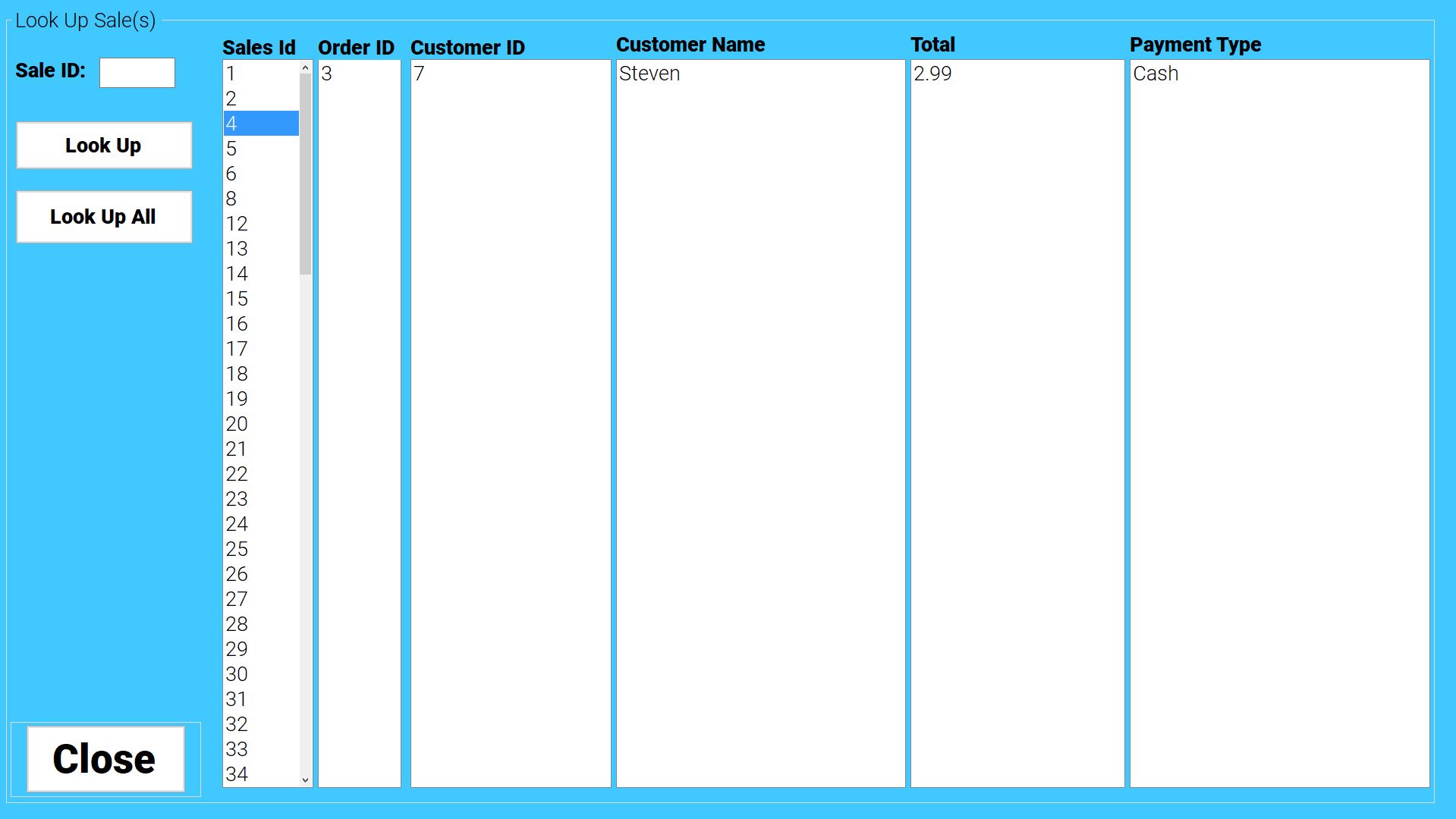
**Products**

In this menu you can look up all products in the **Products table** or you can look up a specific product with its corresponding product ID. In this menu you can add a new Product and edit an existing product



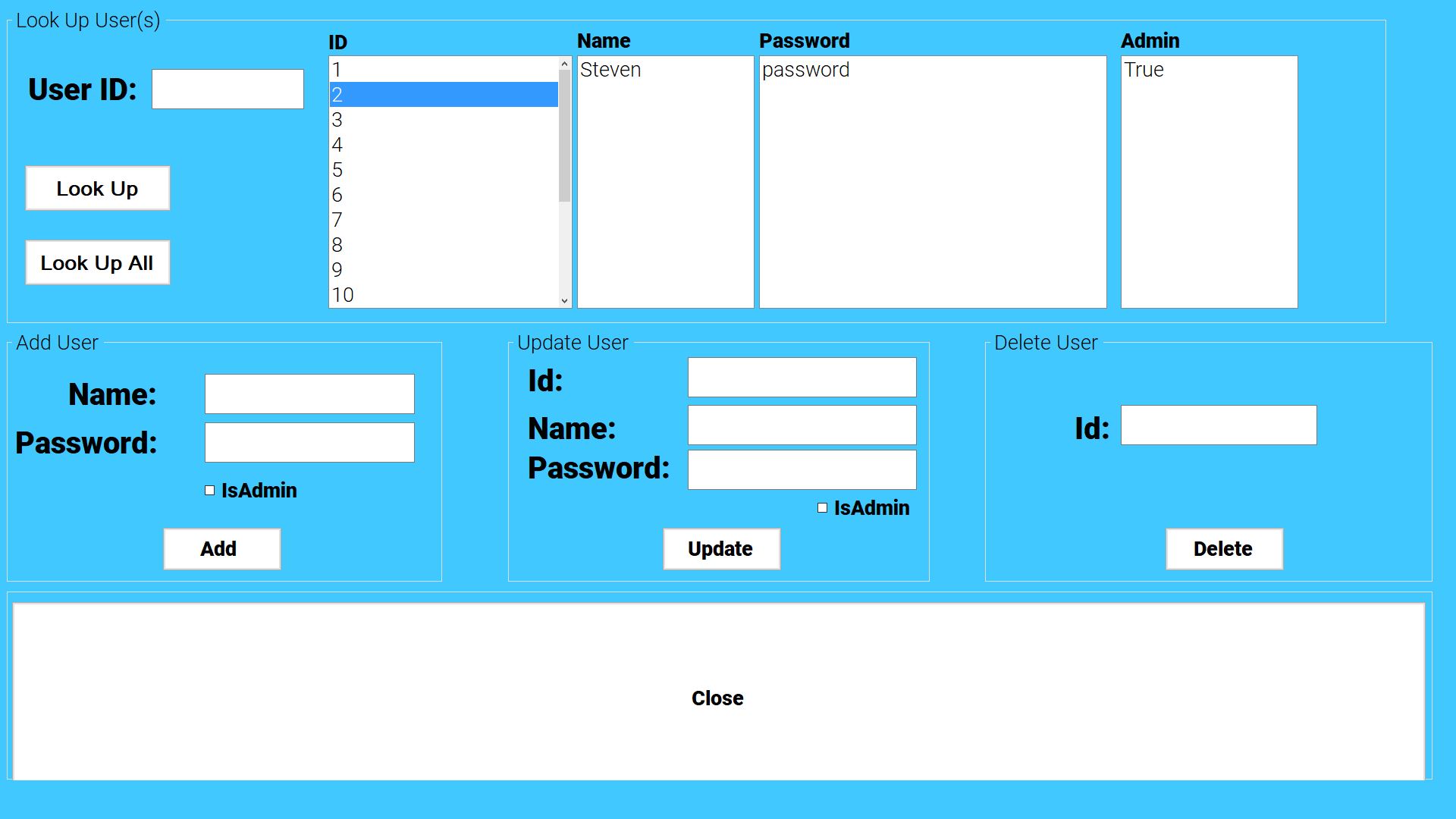
**Sales**

In this menu you can look up a specific sale based on the “Sale ID” or you can lookup all sales. The information is displayed individually based on the “Sales ID”. It will display Order ID, Customer ID, Customer Name, Total of order, and payment type.



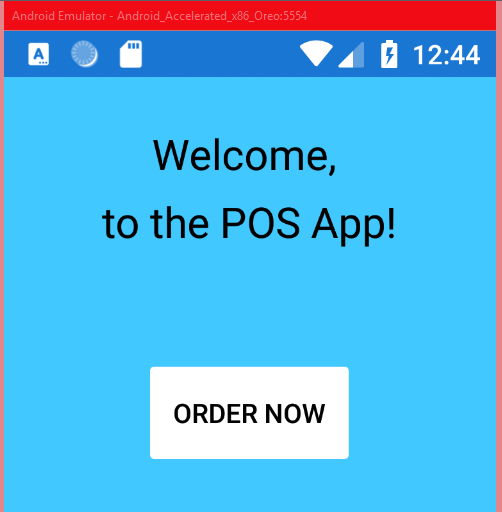
**Users**

This menu has many options for the **Employee Table.** You can look up all employees and all items related to each employee. It shows their ID, Name, Password, and **True** or **False** based on whether the user has admin privileges or not. You can also look up a specific employee by their **User ID.** There are three other options available to the admin user in this menu. You can Add User, Update User, and Delete User.

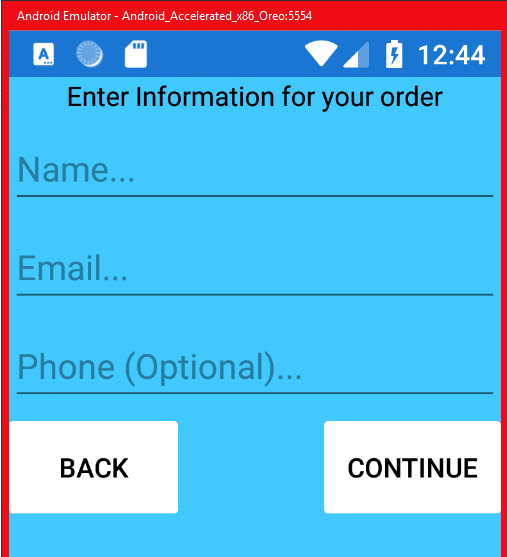


**Mobile Application**

The **Mobile Application** was design to allow customer to create and make there own order without having to go to the restaurant to place an order. The mobile app consists of five parts getting **Customer Information**, **Get Customer** **Order**, **Get Customer Payment**, **Displaying Receipt**, and **Submitting To API**. The mobile application was designed using Xamarin [2] which is a .Net [1] framework that allows for the production of allow mobile devices at once. We mainly developed the mobile app for Android and Windows. IOS was hard to work with since we did not have access to Apple devices to try it out on.



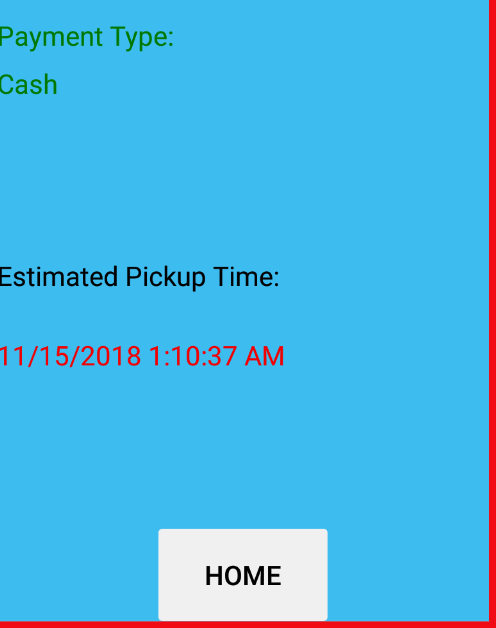
**Customer Information** is the first thing the mobile app gathers from the customer. The customer is asked to give a name and email. These will be stored for later use when submitting to the API. The customer can also give a phone number if they wish.

****

**Get Customer Order** is the second step the mobile app does to get the order that the customer wants to place. It has all the same options that are present in the main ordering system and uses the same cart system as described before. The application will also display to the user what they put in the cart and what the price will be.

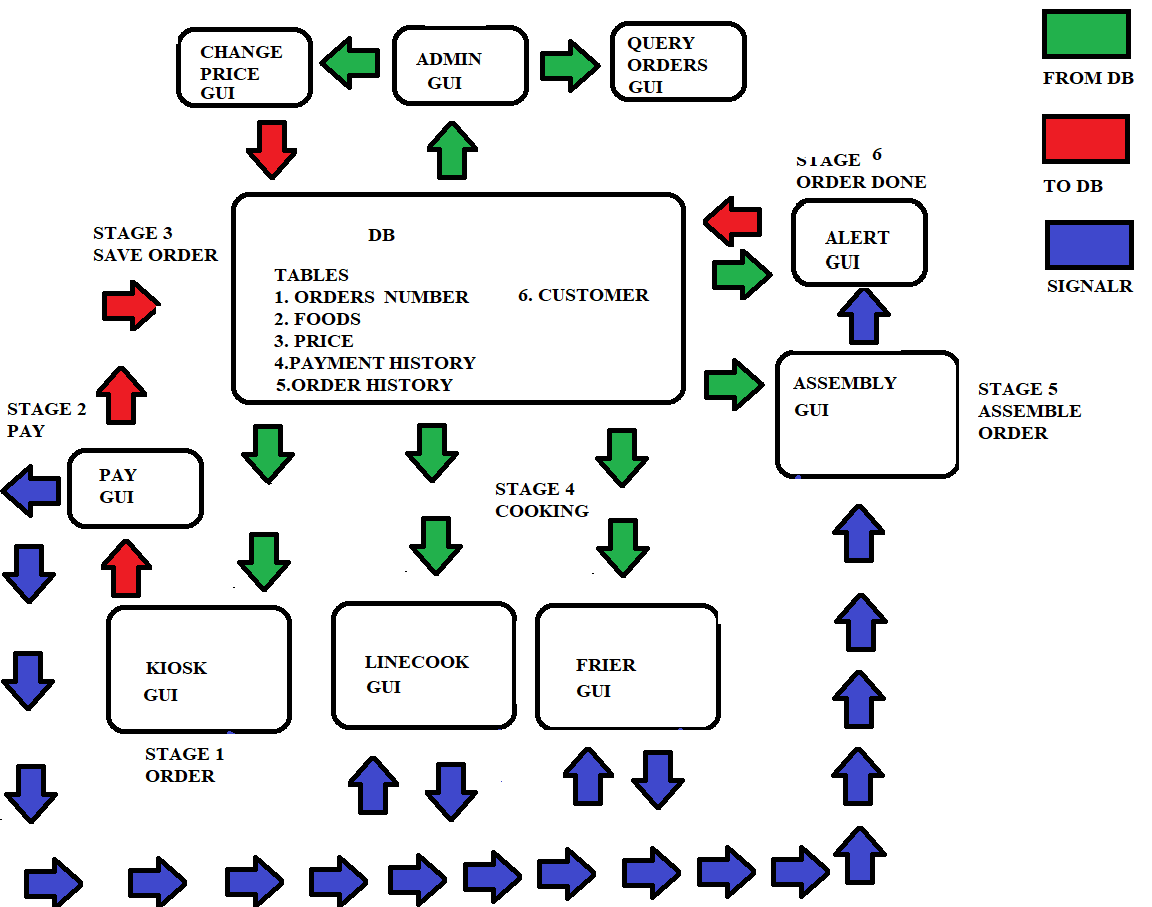
**Get Customer Payment**  is the third step the mobile app does. It shows the customer their total for the order and asks displays either to choose cash or credit for payment. Upon selection of cash the customer is prompted that they can pay at the restaurant. If credit is asked for, the system will bring them to a credit card screen to enter credit card information like in the main ordering system.

**Displaying Receipt** is the last step that user is involved with. It displays the order that has been received and has a pick up time and date. If anything goes wrong along the order instead of the receipt screen the user will be brought back to the main page and alert that order has not been placed and that something went wrong.

****

**Submitting To API** is the same as above in the Main Ordering System. Please reference for an depth detail there.

**Finished Product vs Original Idea**

****

Our original idea looked very different compared to how the finished system works.The above image is our final flow chart before speaking to the professor and before the mobile application idea was presented to us. We thought that we would be using signalR to communicate between the different exe programs and that we would keep it closed network with no outside communication. Mobile application presented new problems that we had to handle differently like having to connect to a network of some type to receive the orders. Our agreed upon plan was to create an Restful Web API that we could interface with to send and receive data with mobile and exe programs. It allowed for us to make our software with more versititly than originally intended. Along with Helper API files written in the .Net framework we could create anything to work for us. With API, we dropped the idea to use signalR to communicate with different programs since API could do this for us and would save us time learn a new technology. A lot of research went to developing the API as none of us had ever created one before [1][3][5][8][10].

The login and logout system with employees. This system was one that we hadn't planned on creating but was rather easy to make. We had to add another table called employees to the database and add it functionally to the API and API Helper files. These systems overall made our project more secure and stop anyone without credential from messing with anything.

The sales table was another item after original flow chart was created. Not much though had went into logging order history other than the order themselves but we decided that keeping track of sale information was a good idea after meeting with professor. We added the new table to the database sales and the functionality to the API and API helper files.

Another item that had changed from the above flowchart is the Admin Tools, more options were added to the tool set then we had planned. We integrated a sales query instead of an order query so that all the information about a sale could be displayed. A more in depth product tool was created not just to change the price of items but also to change name and add new one or delete product that we didn't need anymore. As well as add a new tool for editing, creating and deleting employes. In this tool admin rights can be given or taken away.

We also added an additional step after Stage 5 in the flowchart. We dropped the alert system since we were not using signalR anymore. The new Stage we called pick up which was a new system that we added to keep track of whether orders had been picked up and would log it in the database when it had been picked up.

Overall we are very happy with how everything turned out and how great our functionally actually works. We also learned a lot of great new technologies like Entity Framework [3], Swagger [8], Xamarin [2],and a more in depth view of the .Net Framework [1]. The best take away from our project is being able to build a project that is large enough to work as team and solve a problem that could be real world problem. We had a great team!

**References**

[1] .Net Framework:

Microsoft. (2018). *.NET | Free. Cross-platform. Open Source.*. [online] Available at: https://www.microsoft.com/net [Accessed 5 Nov. 2018].

[2] Xamarin:

Khairunj, “Xamarin Documentation - Xamarin,” *Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/xamarin/. [Accessed: 05-Nov-2018].

[3] Entity Framework Core:

Rowanmiller, “Overview - EF Core,” *Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/ef/core/. [Accessed: 05-Nov-2018].

[4] JSON.Net Framework:

“JSON,” *JSON DOC*. [Online]. Available: https://www.newtonsoft.com/json/help/html/Introduction.htm. [Accessed: 05-Nov-2018].

[5] SQL Server Database:

“SQL Server 2017,” *Software Asset Management – Microsoft SAM*. [Online]. Available: https://www.microsoft.com/en-us/sql-server/sql-server-2017-linux. [Accessed: 05-Nov-2018].

[6] SQL Server Management Studio 17:

“SQL Server Management Studio (SSMS),” *Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/sql/ssms/sql-server-management-studio-ssms?view=sql-server-2017. [Accessed: 05-Nov-2018].

[7] Visual Studios 2017 Community Edition:

“Visual Studio IDE, Code Editor, VSTS, & App Center,” *Visual Studio*. [Online]. Available: https://visualstudio.microsoft.com/. [Accessed: 05-Nov-2018].

[8] Swagger Endpoints:

“Swagger,” *What is Swagger | Swagger*. [Online]. Available: https://swagger.io/docs/specification/about/. [Accessed: 05-Nov-2018].

[9] Linode Linux Server:

“Get a Server Running in Seconds,” *Linode Guides & Tutorials*. [Online]. Available: https://www.linode.com/linodes. [Accessed: 05-Nov-2018].

[10] MSDN:

“developer.microsoft.com,” *Kinect for Windows Sensor Components and Specifications*. [Online]. Available: https://msdn.microsoft.com/en-us/dn308572.aspx. [Accessed: 05-Nov-2018].

[11] Dotnet:

dotnet-bot, “.NET Documentation,” *Microsoft Docs*. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/. [Accessed: 05-Nov-2018].

[12] Gmail:

“One account. All of Google.,” *Gmail*. [Online]. Available: https://www.gmail.com/. [Accessed: 05-Nov-2018].

**Credits**

Project Manager - Koltin

Database - Steven, Koltin, Erik, David

API - Steven, Koltin, Erik, David

API Helper Files - Steven

Main Ordering System - Steven

Cooking Systems - Koltin

Pick Up System - Koltin

Admin Tools - David

Mobile Application - Steven, Erik

Mobile Application GUI - Steven, Erik

Color Scheme - Koltin

EXE GUI Design - Steven, Koltin, David

Linode Server - Koltin