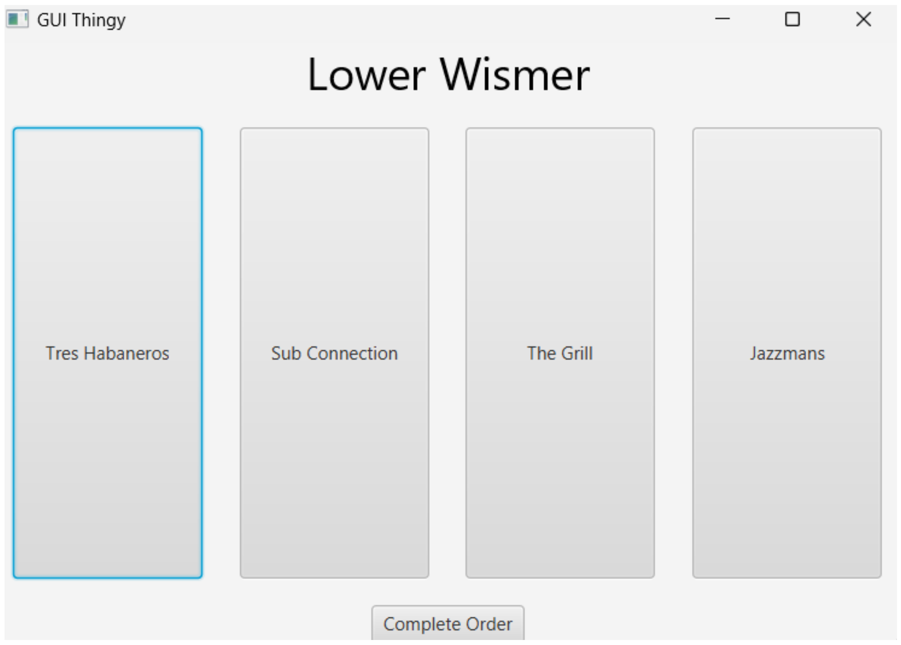
FWLK (AKAFLWK) Final Report

Front Ends

Customer App

Java Application used to create and send orders to the database. A sample of the main menu of the GUI is displayed below.



Installation:

- Required Files
    - Main.java
        - Contains the start() method
        - Sets up the database:
            - If using demo database, make sure getDatabase() is not commented out in start() and getItems() and getAddons() are commented out

- When running the program press enter in the console twice
- If using real database, make sure getItems() and getAddons() are not commented out in start() and getDatabase() is commented out
    - When running the program type the ip address and then the port of the database into the console
- MainController.java
- mainMenu.fxml
- TresController.java
- tresEntrees.fxml
- SubController.java
- subEntrees.fxml
- GrillController.java
- grillEntrees.fxml
- JazzController.java
- jazzFood.fxml
- ItemController.java
- menuitemtemplate.fxml
- ClientCook.java

- Required Libraries
    - JavaFX
        - Used for the graphic user interface (GUI)
    - simple.JSON
        - Used to parse the JSON Strings returned from the microservice/database

- okhttp3
  - Used to make http calls to the microservice/database

Methods:

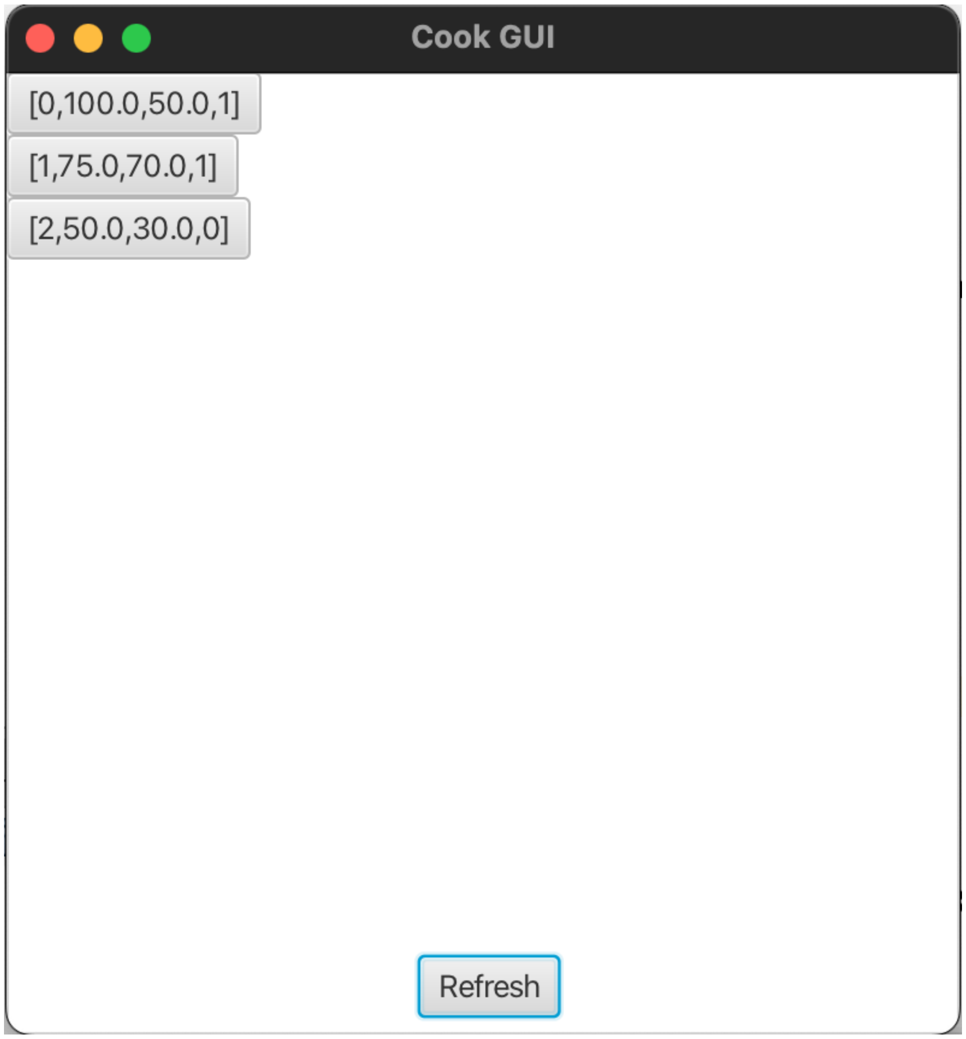
- Main.start()
  - Launches app, displaying the main menu
- Main.getItem()
  - Unused
- Main.getItems()
  - Queries the database for all the menu items and stores them
- Main.getAddons()
  - Queries the database for all the menu addons and stores them
- Main.getDatabase()
  - Used to create a demo database
- Main.makeHash(int, String, int, int, double, Boolean, List, Boolean)
  - Creates a menu item hashmap
- Main.makeAdd(int, String, int, double)
  - Creates a menu addonhashmap
- MainController.toTresEntrees(ActionEvent)
  - Populates tresEntrees.fxml and then updates the gui to show tresEntrees.fxml
- MainController.toSubEntrees(ActionEvent)
  - Populates subEntrees.fxml and then updates the gui to show subEntrees.fxml
- MainController.toGrillEntrees(ActionEvent)
  - Populates grillEntrees.fxml and then updates the gui to show grillEntrees.fxml

- MainController.toJazzfood(ActionEvent)
    - Populates jazzFood.fxml and then updates the gui to show jazzFood.fxml
- TresController.toMain(ActionEvent)
    - Sets gui to mainMenu.fxml
- TresController.toTresEntrees(ActionEvent)
    - Unused
- TresController.toItem(ActionEvent)
    - Populates menuitemiemplate.fxml and then updates the gui to show menuitemtemplate.fxml
- TresController.toMenuItem(ActionEvent)
    - Unused
- SubController.toMain(ActionEvent)
    - Sets gui to mainMenu.fxml
- SubController.toSubEntrees(ActionEvent)
    - Unused
- SubController.toItem(ActionEvent)
    - Populates menuitemiemplate.fxml and then updates the gui to show menuitemtemplate.fxml
- SubController.toMenuItem(ActionEvent)
    - Unused
- GrillController.toMain(ActionEvent)
    - Sets gui to mainMenu.fxml
- GrillController.toGrillEntrees(ActionEvent)

    - Unused
- GrillController.toItem(ActionEvent)
    - Populates menuitemiemplate.fxml and then updates the gui to show menuitemtemplate.fxml
- GrillController.toMenuItem(ActionEvent)
    - Unused
- JazzController.toMain(ActionEvent)
    - Sets gui to mainMenu.fxml
- JazzController.tojazzFood(ActionEvent)
    - Unused
- JazzController.toItem(ActionEvent)
    - Populates menuitemiemplate.fxml and then updates the gui to show menuitemtemplate.fxml
- JazzController.toMenuItem(ActionEvent)
    - Unused
- ItemController.toMain(ActionEvent)
    - Sets gui to mainMenu.fxml
- ItemController.addOrder(ActionEvent)
    - Unfinished, just calls ItemController.toMain(ActionEvent)

Cook App

Java Application used to view and remove active orders. A sample of the GUI of this app (running in macOS) is displayed below.



Installation:

- Required Files
    - CookUI.java
        - Contains the main() method
    - MenuNav.java
        - For current build using web calls:
            - Set isLive = true
            - Set ip to the ip running the microservice
            - Set mapVer = false

    - ClientCook.java
    - CookuiV0.1.fxml
- Required Libraries
    - JavaFX
        - Used for the graphic user interface (GUI)
    - simple.JSON
        - Used to parse the JSON Strings returned from the microservice/database
    - okhttp3
        - Used to make http calls to the microservice/database

Methods:

- CookUI.start()
    - Launches the app, displaying the GUI with a single “Refresh” button
- MenuNav.refesh()
    - Clears any item buttons present, then creates a new button for each active order in the database
        - Calls MenuNav.getOrders()
    - If isLive is set to true, queries the database for all active orders. Otherwise, uses the String sampleMap (if mapVer is true) or sampleArray (if mapVer is false)
    - Clicking on any of these new buttons will call remove() using that button
- MenuNav.getOrders()
    - Sends a request to the microservice for all active orders
    - If successful, returns a string containing all orders
    - If the call fails, returns "no return recieved"

- MenuNav.remove(button)
    - Gets the order id from the button’s label (as a String), prints the ID. Then if isLive is true, queries the microservice to set that order to inactive.
        - If isLive is set to false, this method will NOT update the sample data.
    - Removes the button from the GUI.
- ClientCook.sendRequest(iurl, json, command)
    - Sends a request to the microservice via the given url iurl. If the command is a POST command, sends with the request the information included in json
    - If command is “GET”, calls ClientCook.sendGet(iurl)
    - If command is “POST”, calls ClientCook.sendPost(iurl, json)
- ClientCook.sendGet(irul)
    - Sends a GET request to the microservice
    - Returns the json response string
- ClientCook.sendPost(irul, json)
    - Sends a POST request to the microservice, including the information in the string json
    - Returns the json response string

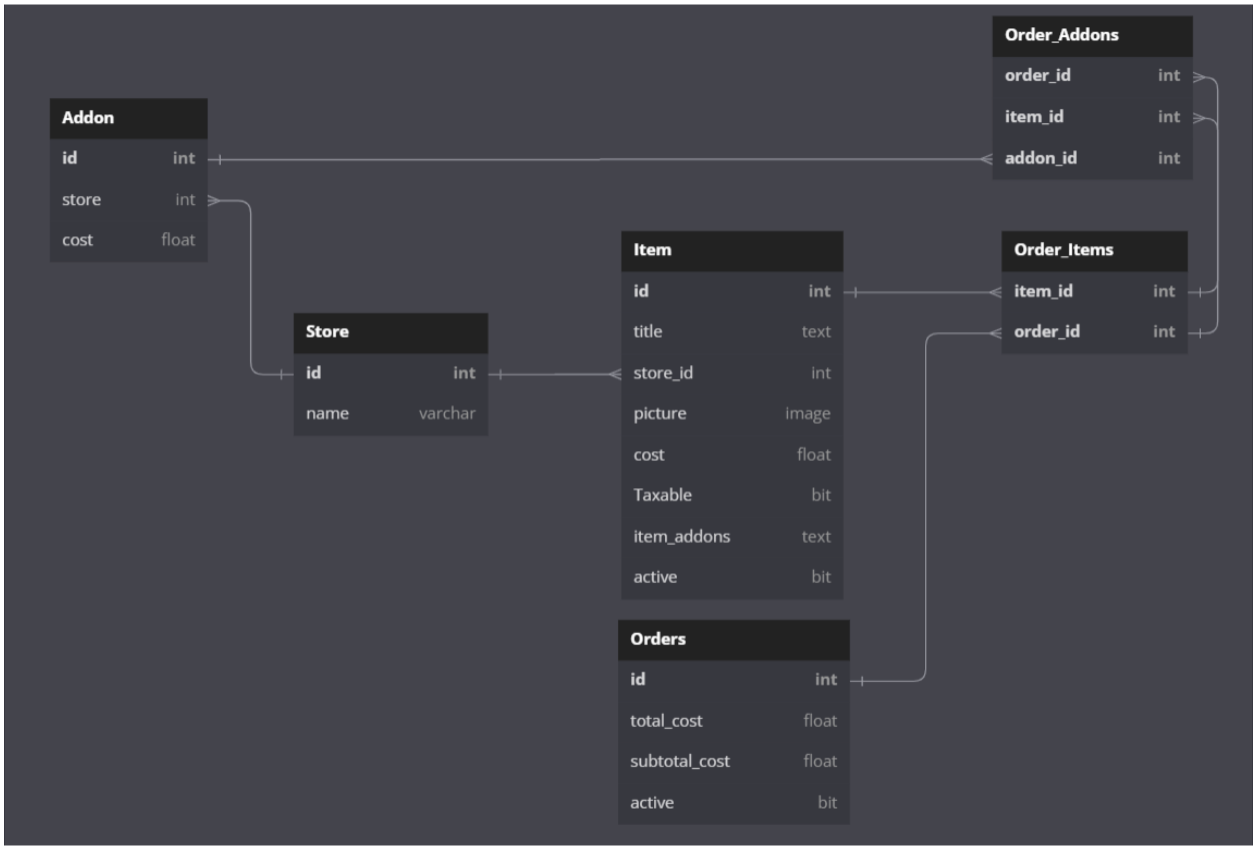
Back Ends

Database

Design:

- Sqlite design through python
- Interact with database through python or java microservice
- JSON returns and posts
- Stored procedures are not supported under sqlite, so microservice reliance is heavy

Structure Diagram:



Microservices

Server and client ends for wireless database access and interaction. Client ends can be run in java or in python and the server end runs in python using flask and waitress to deploy it. Client ends both function primarily by making HTTP messages to send to the server meaning clients in other languages could also be set up with ease if needed.

Server Installation/setup:

- Files: “microservices.py”, “database.db” (or any SQL database structured as mentioned previously)
- Libraries: datetime, sqlite3, flask, waitress, shutil, os

Client Installation/setup:

- Files: for java: “Client.java”, for python: “client.py”
- Libraries: for java: okhttp3 (and dependencies), for python: requests, json

Use Cases (by user):

- Customer UI: print_table (http://ip:port/database/methods/print_table/<table>, GET); used to print the Item and Orders tables from the SQL database as a means of fetching all entries that need to be displayed on the kiosk. add_entries (http://ip:port/database/methods/add_entries, POST); used to add orders to the database. Specific json structure is exemplified in the file.
- Cook UI: fetch_active_orders (http://ip:port/database/methods/fetch_active_orders, GET); retrieves all orders in the database marked as being active. rm_order (http://ip:port/database/methods/rm_order/<order id>, GET); sets an order in the

database as inactive so that it is no longer retrieved when fetching orders. Effectively "completes" an order

- Administrator: dump_db (http://ip:port/database/methods/dump_db, GET); archives the entire database to a database file labeled with the current date and restores an empty database (ideally would also readd item/addon entries). view_old_orders (http://ip:port/database/methods/view_old_orders/<date>, GET); Prints an archived copy of the database.
- Debug*: home_test (http://ip:port/, GET); prints a list of available microservices. create_db (http://ip:port/database/methods/create_db, GET); creates a database file if one does not exist or overwrites it with an empty one. query_database (http://ip:port/database/dev/methods/query_database/<code>, GET); queries the SQL database with the inputted SQL code. print_db (http://ip:port/database/methods/print_db, GET); prints the entire database

*In a final release, all debug methods would absolutely need to be removed or at least have access restricted.

Testing*

*We did not have enough time to implement testing, especially considering we didn't finish the main part of the project

User Report

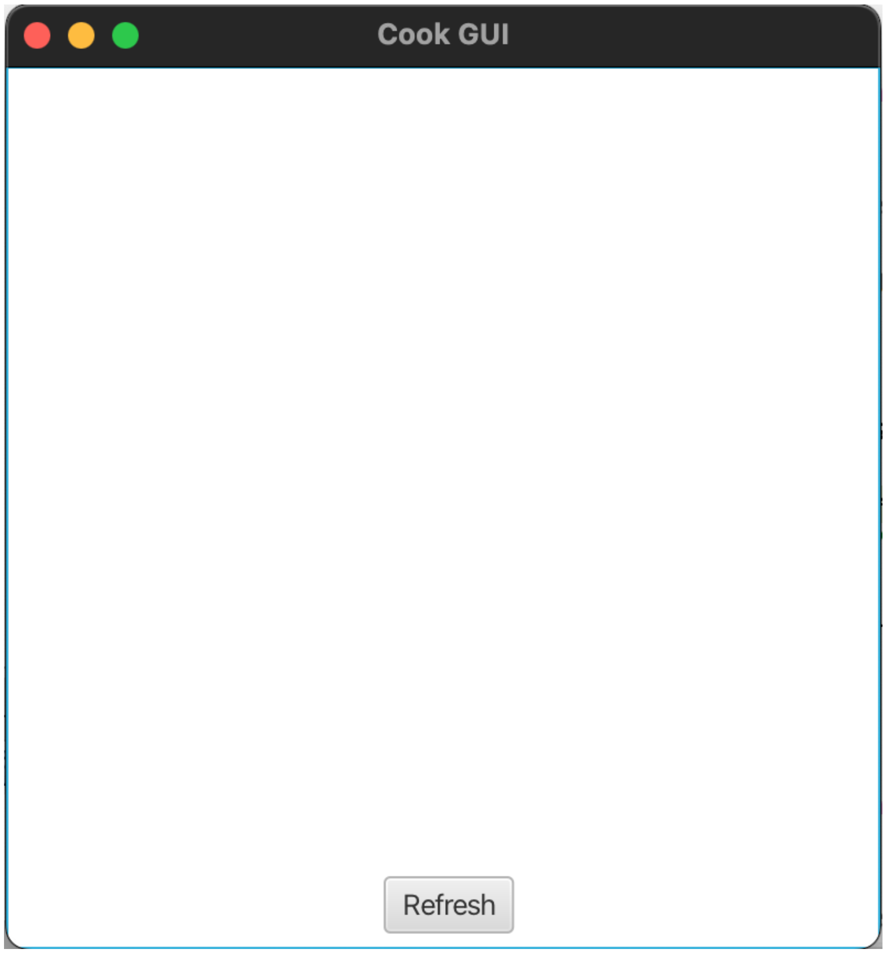
Customer App

1) Begin by following the all installation instructions.
2) Then, you will be greeted by the main menu. There will be four store options infront of you. Choose and click one of them.
3) You will now see the name of the store at the top of the screen, a list of all the menu items being offered at that store, and a button to go back to the main menu. Choose and click one of the menu items.
4) You will now see the name of the item at the top of the screen, a list of addons for that menu item, a back button, and an add to order button. Choose whatever options you want, and then click add to order.

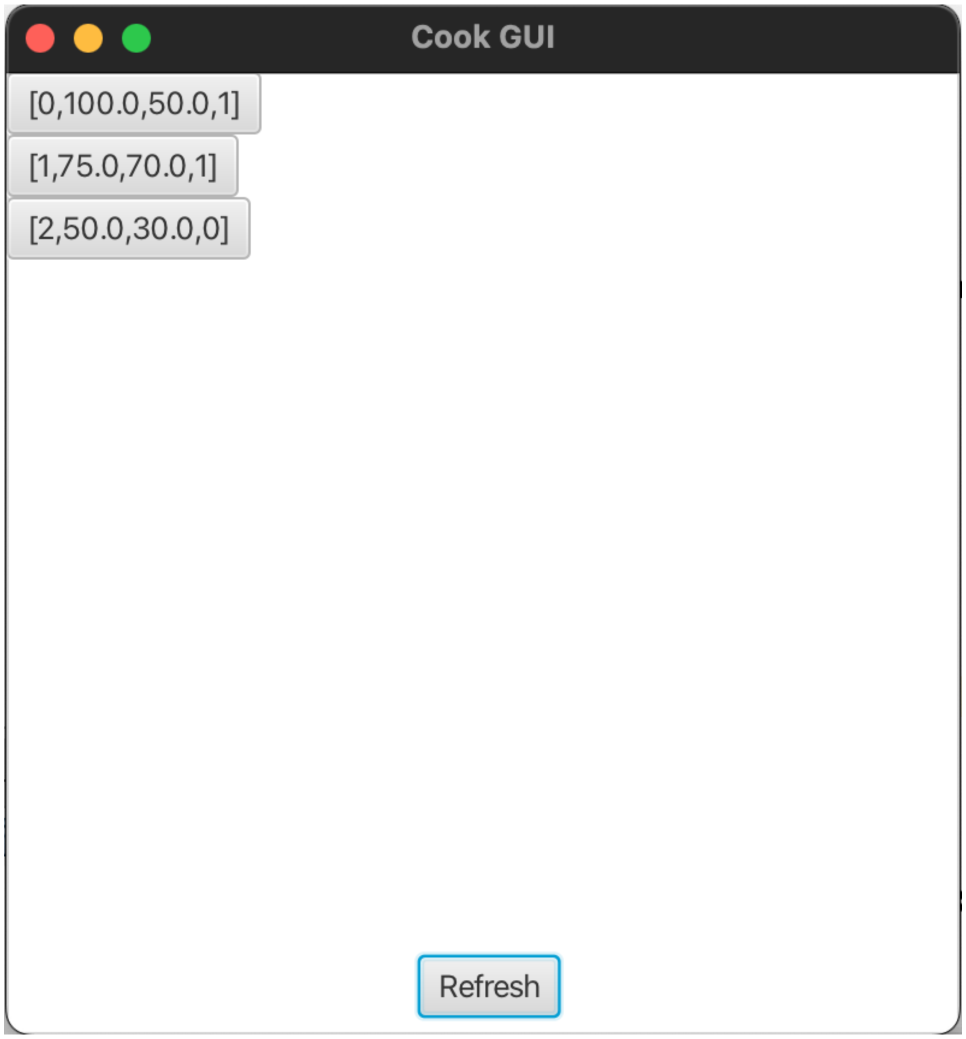
That is the full functionality of the customer ui.

Cook App

1. Upon starting the app, you will view the a panel containing a single button with the text “refresh” in the bottom middle.



2. Upon clicking the refresh button, any active orders will be shown in the form of new buttons starting at the top left of the screen. This must be done again any time new orders are received.



3. Clicking any of these buttons will mark the order as complete, removing the button from the screen. Clicking the refresh button will no longer cause this order to display.

Future Considerations

Were this project to continue, work would definitely have to continue on all fronts. For the customer UI, we would need to hook it up to the microservices and get the order compiling working. For the microservices, we'd want to ensure all the methods work fully and correctly (especially fetch_active_orders). Most notably, we'd need to fully set up unit testing and user acceptance for the project, an area in which we were lacking severely by the end of the project timeframe. For the unit testing, we'd implement this via github so the unit tests would be conducted automatically. For user acceptance, ideally we would get random people who were not involved with the project to follow a set of instructions. Some tests from our initial test plan remain relevant, though many would need to be updated for the current state of the project. Finally, we would generally strive to meet all of our requirements that we set out as goals at the start of the project. Unfortunately though, a single college semester doesn't amount to a lot of time and we had to make cuts for the final submission, but we did manage to lay the infrastructure for a full product to be made if only we had the time to connect it all together.