SWEN 502 Group Assignment

Final Report

Group 3 - Bespoke Burgers

Callum, Danielle, Guy, Julian, Lian

Table of Contents

Introduction

The Bespoke Burgers platform is a professional online burger ordering system that provides functionality to facilitate every step of the process. Firstly, it allows customers to order custom burgers via a website. Those orders are processed, received, and stored by the business in real-time, all while automatically managing inventory levels.

The system works in real-time, updating the store-side UI with new orders and then updating ingredient quantities as orders are submitted. A database is used to store a list of ingredients (with properties relating to them e.g. price, quantity, category) and orders. The database is updated and queried by the server whenever information is passed either from a connection to the website or from the store. In this way we can ensure that only orders that are valid (i.e. only contain available ingredients) are accepted, and that both the store and website have up-to-date and accurate information at all times.

System Description

Basic Functionalities:

- Allow a customer to order a burger through the Customer Web UI .
- Display the details of the order and the order number to the user upon successful order completion.
- Send the order to the Store UI with a status of pending.
- Update the ingredients stock in the database when an order comes through.
- Allow the store worker to update the status of an order (in-progress/completed/collected).
- Allow the store worker to add/update the ingredients list.

The application contains four systems:

- Customer UI - allowing the user to order a custom burger in a user friendly way.
- Production UI - receives the customer burger orders and updates ingredients.
- A server - forwards all information between store clients and web clients appropriately, checking with the database to see if requests are valid.
- A database - to store the stock of ingredients and user orders.

Customer UI:

The customer UI runs on an Apache Server with PHP doing some back-end processing before serving up an HTML page. Styling is done using CSS, and JavaScript is used to control the form functionality.

When a customer loads the order page, PHP opens a socket and connects to the Java server, and a request is sent for all of the ingredients currently stored in the database. This returns the category to which the ingredient belongs, its current stock level, price, and name. The order page is then generated with PHP, displaying only those ingredients for which the quantity is greater than zero (i.e. only ingredients which are in stock).
The ingredients are passed through to JavaScript as a JSON object to ensure that customers cannot select a number of any ingredient greater than the number which is in stock. It is also used to update the running totals as ingredients are selected as well as for verification of the order once it has been completed.

Once an order has been completed, and the customer presses the Submit Order button, a script verifies that a name has been entered and a bun selected (i.e. the bare minimum constraints are met for a valid order). It then sends the order through a post request to another PHP page which opens a new socket connection with the Java server to both verify that it is valid and (if it is) to retrieve the order number to display to the customer. It returns the server's response back to the script.
If the order is deemed to be valid, the customer is sent to a new page which displays their order, the total cost, and the order number for retrieval from the store. If it was invalid, a message is displayed to the user that some of the ingredients they selected are no longer available, and they are prompted to refresh the order page. Refreshing the page starts the process again; another request is sent to the server for the current ingredients. If stock levels for the ingredient had dropped to zero, that ingredient will no longer be displayed. Otherwise, the maximum quantity the customer can choose will be updated to the current stock level.

Production UI:
The production UI is built in Java with JavaFX. The majority of styling is achieved with CSS.

Multiple copies of the store client application can be running at once, either on the same computer with separate displays or on separate computers. They must all connect to the same server in order to receive updates in real-time. This is achieved through a network connection. At this time the IP address and port number to which the client connects is hardcoded in the ClientConnection class; this would need to be changed for each computer running the software. A decision was made not to add this to the UI as we would not want employees who do not understand networking to alter it and thereby preclude the client from establishing a connection with the server. In hindsight a better design would be to include this information as part of a config file (one for the client and one for the server) so that the software package could be deployed as an executable file, and the address and port number could be configured as required without needing to access the source code. Another option could have the client broadcast its intent to connect to the server, to which the server could respond by establishing a connection.

Information is shared between all clients that are connected to the same server. This means that all incoming orders, changes to order status, alterations to ingredients, etc are updated on all clients in real-time.

When the store client first loads (after the UI has been constructed), it connects to the server (connecting to the same port that the website connects to) and registers itself as a store client so that the server knows what information to send it. It then requests the lists of categories, ingredients, and all orders currently stored in the database. These arrive as comma separated strings, which are then parsed and used to instantiate Category, Ingredient, and Order objects. The ingredients and orders tabs are then populated (with each Order object being wrapped with an OrderPane object and appended to the OrderUI's layout, and each Ingredient object wrapped with an IngredientRow object and appended to the IngredientUI's gridpane), at which point the UI is ready to be used by an employee.

The orders tab was designed to be usable either with a mouse or as a touch interface, and to be easily seen and read from a distance. Orders are displayed in a horizontal scrolling layout, with the oldest order on the left. Orders can have one of four statuses ("pending" when they have been submitted but not yet made, "in-progress" when a cook is making the order, "complete" when the order has been made, and "collected" once it has been given to the customer). Status of an order is cycled simply by clicking (or tapping) on it.

The orders tab also has several "filters" through which it can be viewed. There is a drop-down menu in the top right corner of the screen containing the different filters.
When filter is set to Cook, only orders that are pending or in-progress are displayed. Furthermore, of the "in-progress" orders, only those selected by a given cook are displayed to that cook. Orders selected by cooks are hidden from view for other cooks. A cook can only toggle an order between "pending", "in-progress", and back again by tapping or clicking on it.
Once the order is complete, they can click the "Done" button to set its status to "complete" at which time it vanishes from their view.

The Cashier filter displays all orders which are not collected. They cannot toggle status by tapping or clicking the order. Instead, when they have delivered a completed order to a customer, they can click the “Collected” button, after which that order is hidden from their view.
The Manager filter can see all orders regardless of status from oldest to newest. A manager can cycle through all order status simply by tapping or clicking on them, or by clicking the appropriate button (“Done” or “Collected”) on the order.

The ingredients tab is intended to always be used with a mouse and keyboard, as there are many fields where data can be entered. The idea is that a store manager can use it to update information on what ingredients are in stock (including adding new ingredients or removing ingredients entirely).
Ingredients are listed vertically, with one row per ingredient. They are ordered first by category, and then alphabetical order. The name, current stock, and minimum threshold (the minimum quantity allowed in stock; below this level, the ingredient will display as red on the orders tab and the quantity in red on the ingredients tab) for each ingredient is displayed. To the right of each ingredient is a convenient way to alter the current stock levels in the system. An employee can choose whether to increase or decrease stock levels, and then input the amount by which to adjust them. The text field here will only ever accept integer values, and only up to four characters. Clicking the “ok” button will update stock levels for that ingredient on all clients and in the database (assuming the employee did not attempt to reduce stock below 0), and any customers loading the orders page after that moment will have up-to-date stock levels as well.

When an ingredient is selected, an order button and a settings button appear next to it. The order button displays a modal window that lists the ingredient name and wholesale price per unit. When used in a real-world scenario, this could be connected to the business’ existing ordering service allowing a quantity of the ingredient to be entered and ordered directly from the wholesaler. The settings button displays a modal window which allows the user to edit the ingredient name, category, threshold, and selling price, or to remove the ingredient from the database completely. Note that currently changing the ingredient name and category are not functional options; they are displayed in the UI purely to show that the intent for that functionality was there.

Next to the ingredient list is a drop-down box containing the options “New Ingredient” and “Edit Category”. Each of these options open a modal window, the former allowing the user to add a new ingredient to the database, while the latter allows ingredient categories to be added or removed from the system. Categories are used to define in what order ingredients are displayed, and the “bread”, “patty”, and “sauce” categories are populated on the website differently (i.e. in dropdown menus) than all other categories. Categories are also used to group related ingredients in the Orders UI to improve readability.

All alterations to orders, ingredients, and categories are sent through the ClientConnection (which assembles the appropriate String according to the requirements detailed in the Protocol class) to the server, which then processes them as appropriate.

Server:
The server is written in Java and uses JDBC to connect to the database.

When the server first starts up, it requests the timestamp and order number of the most recent order. This is used to ensure that each new order has a unique order number for that day (i.e. if the last order was in a previous day, the order number resets to 1; otherwise it continues where it left off).

There is a class of final static fields called Protocol which holds both Javadocs regarding how instructions sent through the server should be formatted, as well as fields to facilitate the identification of various instructions.

When a request is sent to the server, it checks which protocol has been used, and attempts to parse the request. If the request has not been correctly constructed (e.g. the correct delimiter has not been used, or an incorrect amount of data has been sent, or data is out of order) this will normally trigger an IndexOutOfBoundsException or NumberFormattingException which is caught by the server and registered as an error.
If the request is correctly constructed, depending on the protocol the database may be polled to check if it is valid (e.g. if requesting a decrease in ingredient stock levels, it will check if

there is enough stock to decrease it by the requested amount). Invalid requests are registered as failures.
If information has been requested (e.g. requesting a list of ingredients in the database), the information is requested from the database and sent back to only the client that requested it. Otherwise if the request is an instruction to alter something, and the request is deemed valid, the alteration is made in the database and then forwarded to all store clients other than the one which made the request.

Any requests which result in failure or error are sent back to the client that requested them indicating either failure or error as appropriate.

Successful requests are sent through to any store-side clients that are connected, but the appropriate changes are also made in the database. This means that even if no store clients are running, successful orders will be stored in the database to be displayed on the store client when it first loads up (with appropriate ingredient levels having been decreased).

Clients have a similar method to check which protocol has been sent, and call methods on either the IngredientsUI or OrdersUI as appropriate depending on the protocol.

Database:
The database is a PostgreSQL database.

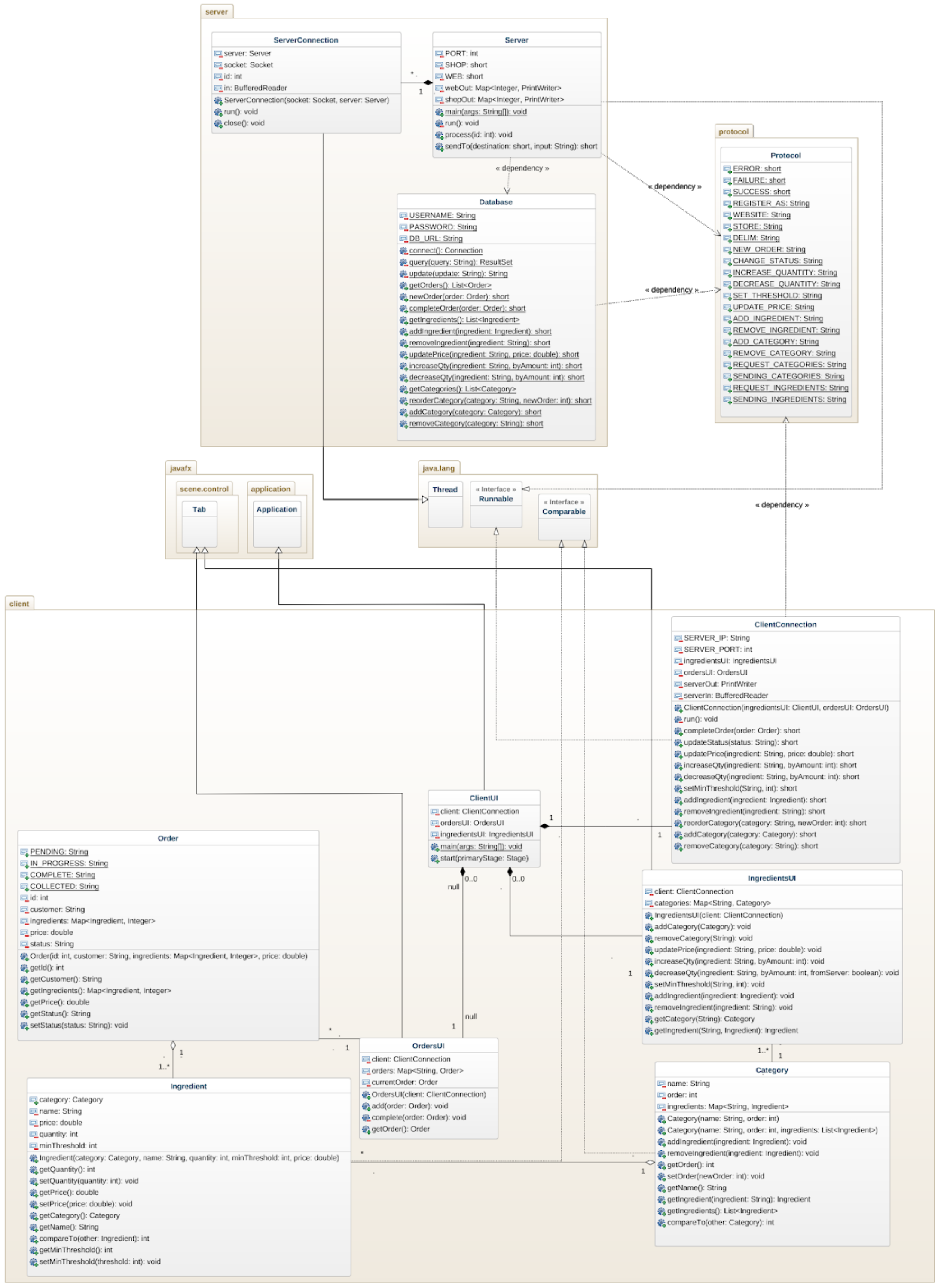
When an updating or querying request is sent from the customer client or store client, it will be received by the server. Static methods in the database class will be called to process the request.

In the database class, there are three main utility methods: connect, update, and query. The connect method initializes the connection between Java and the database and returns that connection; the update method deals with all the update requests (e.g. adding or removing ingredients, adding orders, changing order status); and the query method deals with queries (e.g. getting ingredients and orders). The update and query methods handle exceptions by returning a null (or ERROR) result and printing the stack trace.

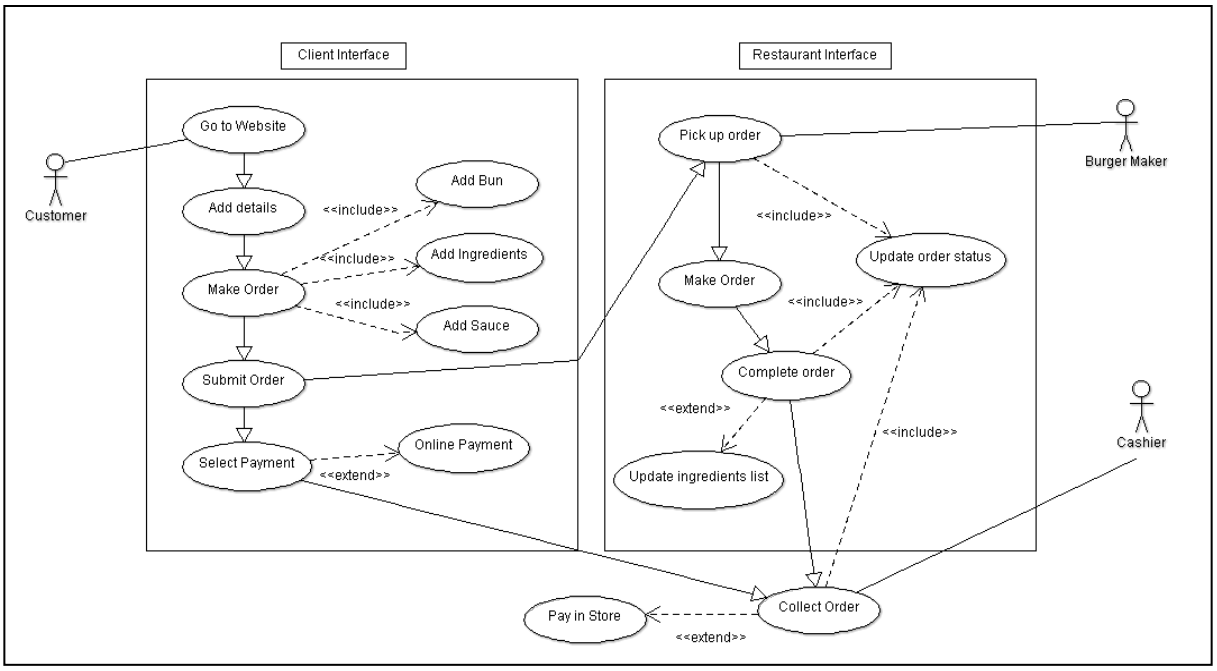
Then there are a series of specific update and query methods. For example "getOrders()" which queries and returns all the orders in database; "newOrder()" which updates the order table by adding a newly placed order. Each specific method will call either update or query method according to their types, and deals with query results separately. Get methods return either null or a string representation of the requested data. All other methods return a short value corresponding to the appropriate Protocol field (SUCCESS, FAILURE, or ERROR) back to the server indicating if the request is processed successfully or not.

All the results returned from database class will follow a standard format which is defined by protocol class, so that the customer client and store client will be easy to understand and display the results.

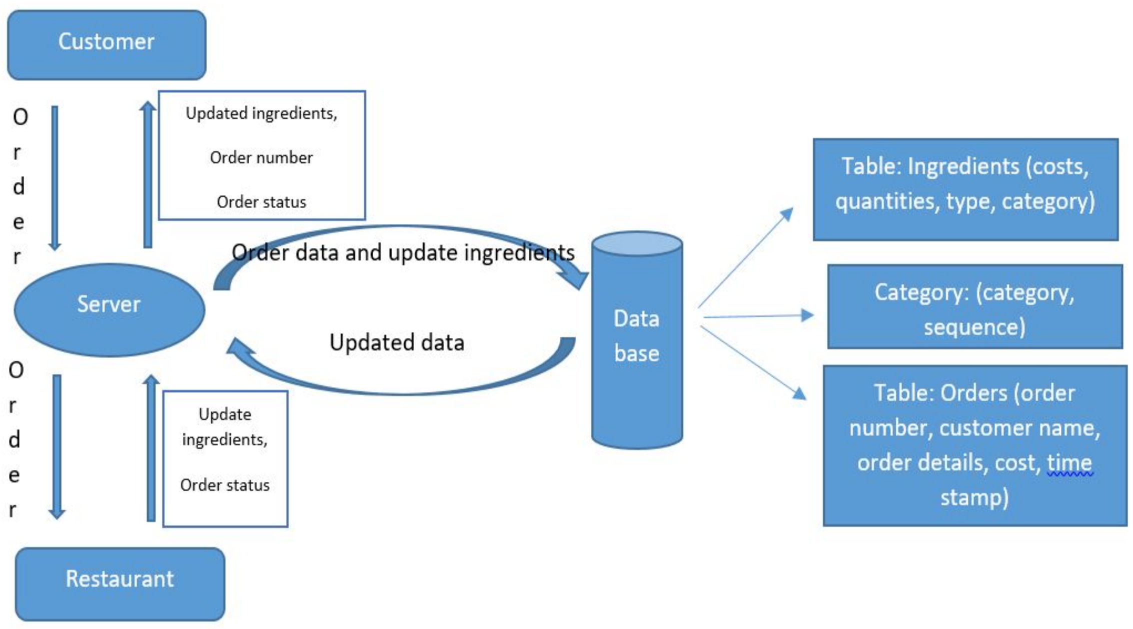
Class Diagram


server
ServerConnection
server: Server
socket: Socket
id: int
in: BufferedReader
ServerConnection(socket: Socket, server: Server)
run(): void
close(): void
*
1
Server
PORT: int
SHOP: short
WEB: short
webOut: Map<Integer, PrintWriter>
shopOut: Map<Integer, PrintWriter>
main(args: String[]): void
run(): void
process(id: int): void
sendTo(destination: short, input: String): short
« dependency »
Database
USERNAME: String
PASSWORD: String
DB_URL: String
connect(): Connection
query(query: String): ResultSet
update(update: String): String
getOrders(): List<Order>
newOrder(order: Order): short
completeOrder(order: Order): short
getIngredients(): List<Ingredient>
addIngredient(ingredient: Ingredient): short
removeIngredient(ingredient: String): short
updatePrice(ingredient: String, price: double): short
increaseQty(ingredient: String, byAmount: int): short
decreaseQty(ingredient: String, byAmount: int): short
getCategories(): List<Category>
reorderCategory(category: String, newOrder: int): short
addCategory(category: Category): short
removeCategory(category: String): short
« dependency »
« dependency »
protocol
Protocol
ERROR: short
FAILURE: short
SUCCESS: short
REGISTER_AS: String
WEBSITE: String
STORE: String
DELIM: String
NEW_ORDER: String
CHANGE_STATUS: String
INCREASE_QUANTITY: String
DECREASE_QUANTITY: String
SET_THRESHOLD: String
UPDATE_PRICE: String
ADD_INGREDIENT: String
REMOVE_INGREDIENT: String
ADD_CATEGORY: String
REMOVE_CATEGORY: String
REQUEST_CATEGORIES: String
SENDING_CATEGORIES: String
REQUEST_INGREDIENTS: String
SENDING_INGREDIENTS: String
javafx
scene.control
Tab
application
Application
java.lang
Thread
« Interface »
Runnable
« Interface »
Comparable
« dependency »
client
ClientConnection
SERVER_IP: String
SERVER_PORT: int
ingredientsUI: IngredientsUI
ordersUI: OrdersUI
serverOut: PrintWriter
serverIn: BufferedReader
ClientConnection(ingredientsUI: ClientUI, ordersUI: OrdersUI)
run(): void
completeOrder(order: Order): short
updateStatus(status: String): short
updatePrice(ingredient: String, price: double): short
increaseQty(ingredient: String, byAmount: int): short
decreaseQty(ingredient: String, byAmount: int): short
setMinThreshold(String, int): short
addIngredient(ingredient: Ingredient): short
removeIngredient(ingredient: String): short
reorderCategory(category: String, newOrder: int): short
addCategory(category: Category): short
removeCategory(category: String): short
ClientUI
client: ClientConnection
ordersUI: OrdersUI
ingredientsUI: IngredientsUI
main(args: String[]): void
start(primaryStage: Stage)
1
1
0..0
null
0..0
null
1
Order
PENDING: String
IN_PROGRESS: String
COMPLETE: String
COLLECTED: String
id: int
customer: String
ingredients: Map<Ingredient, Integer>
price: double
status: String
Order(id: int, customer: String, ingredients: Map<Ingredient, Integer>, price: double)
getId(): int
getCustomer(): String
getIngredients(): Map<Ingredient, Integer>
getPrice(): double
getStatus(): String
setStatus(status: String): void
*
1
IngredientsUI
client: ClientConnection
categories: Map<String, Category>
IngredientsUI(client: ClientConnection)
addCategory(Category): void
removeCategory(String): void
updatePrice(ingredient: String, price: double): void
increaseQty(ingredient: String, byAmount: int): void
decreaseQty(ingredient: String, byAmount: int, fromServer: boolean): void
setMinThreshold(String, int): void
addIngredient(ingredient: Ingredient): void
removeIngredient(ingredient: String): void
getCategory(String): Category
getIngredient(String, Ingredient): Ingredient
1
1..*
1
1
1..*
OrdersUI
client: ClientConnection
orders: Map<String, Order>
currentOrder: Order
OrdersUI(client: ClientConnection)
add(order: Order): void
complete(order: Order): void
getOrder(): Order
Ingredient
category: Category
name: String
price: double
quantity: int
minThreshold: int
Ingredient(category: Category, name: String, quantity: int, minThreshold: int, price: double)
getQuantity(): int
setQuantity(quantity: int): void
getPrice(): double
setPrice(price: double): void
getCategory(): Category
getName(): String
compareTo(other: Ingredient): int
getMinThreshold(): int
setMinThreshold(threshold: int): void
*
1
Category
name: String
order: int
ingredients: Map<String, Ingredient>
Category(name: String, order: int)
Category(name: String, order: int, ingredients: List<Ingredient>)
addIngredient(ingredient: Ingredient): void
removeIngredient(ingredient: Ingredient): void
getOrder(): int
setOrder(newOrder: int): void
getName(): String
getIngredient(ingredient: String): Ingredient
getIngredients(): List<Ingredient>
compareTo(other: Category): int

Use Case Diagram:



Server Architecture:


Customer
Order
Updated ingredients,
Order number
Order status
Server
Order data and update ingredients
Updated data
Data base
Order
Update ingredients,
Order status
Restaurant
Table: Ingredients (costs, quantities, type, category)
Category: (category, sequence)
Table: Orders (order number, customer name, order details, cost, time stamp)

Design Diversions (and additional problems we faced)

Overall we met all but one of our System Requirements for the Web UI and the Production UI. We didn't add UR6 which was functionality for the customer to pay for an order online (or choose to pay in store). With more time we would build this functionality into the system.

Technically the system is NOT robust against SQL injection attacks which was SR6 for the DataBase and server. Although the WebUI does not allow non-alphanumeric characters, that could be circumvented at the client-side quite easily. The server is not checking for invalid characters.

To format the style and layout of the orders and ingredients UI, it was decided that it would be best to create and link a CSS file to the appropriate classes, as opposed to using JavaFX methods or in-line CSS styling. This was done not only to keep it clean and distinct from the logic, but also because some styling could only be achieved using an external CSS file. At first this worked well, but once the CSS file grew larger, Eclipse began to cache the file at certain intervals to reduce overhead reading through it at every compilation time. This meant that a change could be made in the CSS file which you wouldn't see for up to half an hour, which meant you often didn't know whether the code had worked or not. We searched for a way to stop this from happening, but after several hours with no solution we decided to cut our losses and just accept it. A method where we would make some changes along with one obvious colour change, write that down, and then continue with some other task. Once the colour changed, we would check if the other changes resulted in the desired effect. This worked, but closer to the deadline when changes needed to be made quickly, some in-line styling was used instead, which resulted in less-than ideal code organisation. In future projects, we would make sure that this wouldn't become a problem by either finding a solution for Eclipse or by using a different IDE.

We found some interesting issues connecting to a Java server from PHP over a socket connection which were very unexpected. On one computer, reading multiple successive values through the connection from PHP (using the socket_read() function) worked fine, but

on other computers it did not. We ended up spending about an hour trying to track down why this difference occurred before finally choosing to establish a new connection for each piece of information we required. Not ideal, but it worked.
We also had issues due to new lines at the end of socket reads; socket_read requires either a pre-specified number of bytes to be sent, or a new line (“\n”) to indicate that the message is complete. Unfortunately it includes the new line symbol at the end of the final string, which took a while to realise. Once we figured that out we added a rtrim() to the socket_read() to remove that new line symbol.

The code for the server-client connectivity was written fairly hastily at the beginning of the project based on the class diagram, which had been partially based on the previous networking assignment. This code was naively not fully tested until it came time to integrate the UI with the ClientConnection object and thereby with the server.
The result was a series of small errors (e.g. attempting to compare strings using the == operator) that popped up late into the project, and were made more difficult to identify due to the increased complexity (i.e. addition of UI) that would have been much more easily resolved if tested as the code was written.

When creating the initial design for the store application, it did not take into account orders that had been made in previous days. Currently the client will retrieve all orders from the database when it first loads up, however order numbers can be repeated over various days (they reset to 1 at the commencement of each day) and are stored in the application in maps with a key being the order number. Orders with the same number on different days will likely override one another and could cause problems.

We had multiple problems with Git keeping track of the build path; the Mac computer in our group required a different Java library being included than the other computers, and as a result any time we merged between the Mac and other computers, everyone had to change the build path. We ended up adding the build path file to the .gitignore file but that only made issues worse. We’re still not sure what caused this issue to arise.

Lessons Learnt/reasoning for design diversions

- Testing should be done early and frequently. A good rule of thumb might be if you've written a couple hundred lines of code, you should already have tested by that point.
- Integrating system components early (and verifying integration with new features often) would have been hugely beneficial. Merging later in the design meant a lot of time spent trying to get things to work, and a lot of time tracking down relatively small mistakes that could have been more easily caught if the system were less complex at the time of integration.
- Git. There were knowledge gaps around the correct processes of pushing to the Master branch from multiple individual branches. The group had multiple merge conflicts that would take time to problem solve. We learnt to write the instructions for successful merging up on the whiteboard for everyone to see and to follow them closely.
- The socket issues with PHP taught us that just because two languages have a similar feature which we can use to connect the languages together, we shouldn't assume that those features work the same way on both languages and should do additional research to find out what differences exist and whether they may cause problems.
- We learnt to isolate which part of the program will be responsible for what piece of information. Sometimes there was confusion around what PHP should look after and what should JavaScript look after.
- We learnt that it is much easier to design a UI with a UI mockup already done. We stuck very closely to our UI mockups as a result and it helped keep us on track.
- General JavaScript and PHP programming. Since we hadn't covered a lot of this content in class, it was good to develop using these two languages and learn how they interact to build Web Apps.
- In future projects, to avoid issues like the duplicate order numbers between days issue, taking more time to think through and discuss how the backend would behave in a real-world scenario over time would help. Additionally, testing the server-client connectivity early and regularly would help mitigate errors when integrating with the UI components.

Additional functionality we would like to have added with more time:

- Ability to order multiple different burgers per order.
- User authentication and privilege levels based on job title e.g. manager has authentication to adjust stock levels.
- Ingredients on customer UI to update every 1 minute or so instead of just on page refresh.
- Currently, if a customer's order is not valid then an alert window will appear. The user then needs to refresh the page. It would be better that when the customer clicks the alert window that the page is programmed to reload.
- Suggested burger combinations (e.g. cheeseburger default, hamburger default).
- Ability for customer to save current order combination for future use.
- Opening hours displayed on the website.
- Ability to select method of collection (delivery or pick-up).
- Customer option to specify pick up time which notifies cooks at the correct time.
- Time bound progress for burger cooking - if burger is in-progress for too long then employee is notified.
- Web display (and live update) of status of order and the expected waiting time.
- Order of ingredients in web should reflect order suggested by categories ordering
- Currently listeners on the Ingredients objects are retained even after the objects that created them are removed. This could result in a stack overflow. With more time we would tidy that up (e.g. by clearing listeners from IngredientRows before refreshing the IngredientsUI.
- When an order is complete, it is retained in the UI (for the managers filter) until the client is closed. This could result in a stack overflow. Additionally, when the client first loads up it requests all the orders that exist in the database regardless of their status or when they were made. We would impose some time limit after which completed orders would be purged from the UI, and additionally some constraints on which orders could be requested at load time (e.g. all non-completed orders, as well as completed orders made in the last two days).

- The design of the orders tab in the store UI was intended to be usable as a touch interface, but would also easily lend itself to the sort of interface that McDonald's uses (i.e. a limited set of buttons on the bottom of the monitor). For this to work seamlessly we would need to make the orders neatly focus traversable, so that a button could be mapped to the tab key to cycle through the orders.
- Currently if a request from the ClientConnection is refused by the server, a fail message is sent back to the client and the "undoAction" method is called. This method however has not been implemented. The intention was that the action which was refused would be undone on the local client, thus putting the client back in conformity with the server and other clients (as the server will not have forwarded the request to any other clients).
- If the server goes down or the connection is disestablished unexpectedly, the client should attempt to reconnect automatically.
- It would be useful (for debugging but also in the real world) for both the store client and the Java server to keep a log of errors that occur, what message was sent to cause them, and the time they occured at the very least.
- Currently if no Java server is running, the store client won't load up at all. It would be better for the client to load, but display a user-friendly error indicating that no connection to the server has been established.

Individual Contributions

Although the project was a group effort, we worked to play to our individual strengths therefore specific areas of the project were assigned to specific people as listed below. We also did a lot of work in pairs when working on similar aspects of the project.

Callum - Store UI

- Initial design/mockups for entire store-side UI (the ingredients tab, orders tab, and modal windows).
- Implemented the orders UI page using JavaFX and CSS.
- Implemented and tested order status changing and filtering across multiple order UI clients.
- Created the modal windows controls for ordering more ingredients, settings, adding new ingredients, and adding/removing categories.
- Helped integrate the orders UI with the backend.

Guy

- Initial class diagram, method signatures, and Javadocs for store-side application (UML, Java).
- Server-Client Model implementation (i.e. Server, ServerConnection, Protocol, and ClientConnection classes) (Java).
- Helped integrate store-side UI with ClientConnection class (Java).
- Helped integrate web server with Java server (PHP, Java).
- Implemented CurrencyTextField and IntegerTextField classes to limit user input to only allow integers or currency data. (Java).
- Initial implementation of IngredientsUI (excluding modals) and IngredientRow classes (Java).
- Helped implement dynamic loading of ingredients on web app (PHP).
- Ensured price updates as ingredients are selected (JavaScript).
- Assisted with debugging and testing on most parts of the system (Java, PHP, JavaScript).

Lian - Database.

Responsible for creating and maintaining databases which stores data needed for the two clients, as well as all the methods involves database.

- Designing database structure mockup.
- Creating Postgres JDBC database and maintaining tables (ingredients, orders, category) in database.
- Implementing all the update and query functions in server/database class.
- Setting up buttons and menu in ingredient UI to display modal windows.

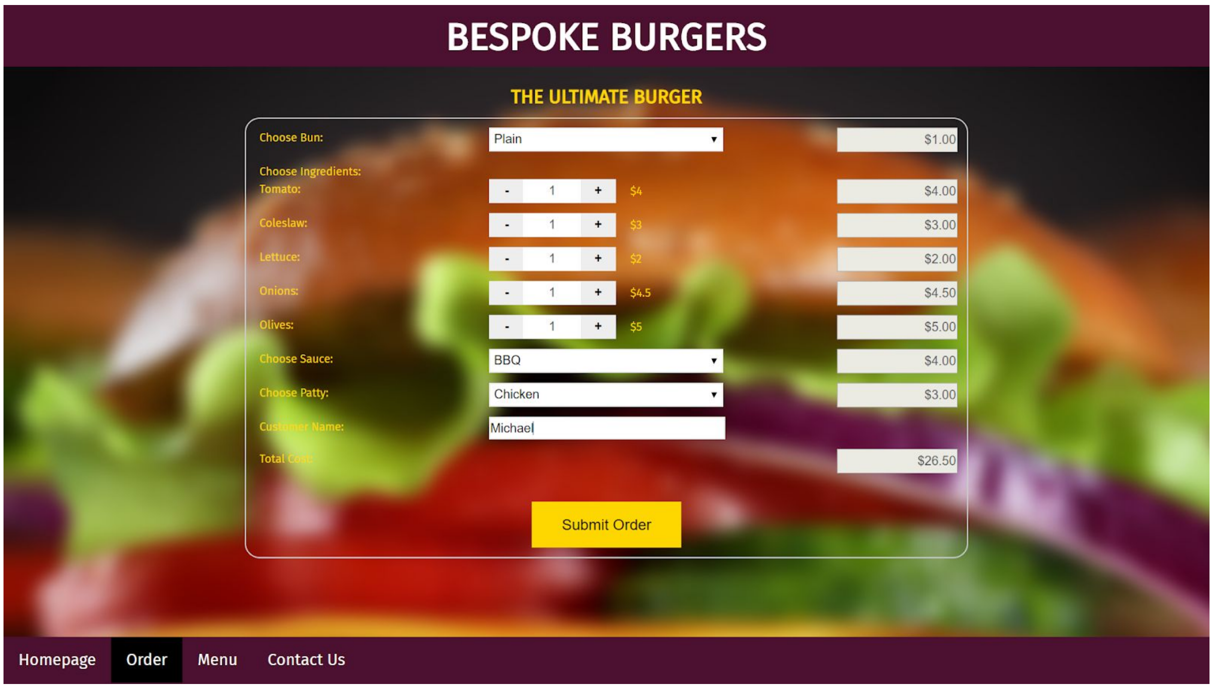
Danielle/Julian - Customer UI

Responsible for the customer website and ensuring the website was intuitive, easy to use, connected to the server and requested the right information at the right time. The tasks to ensure this was achieved were:

- Initial design mockup (Julian).
- Setting up the design and layout of the webpage using HTML/CSS.
- Creating a form layout for the ordering page, containing controls to enable a user to create a new order, achieved using HTML, CSS and JavaScript (Danielle/Julian).
- Ensuring the ingredients displayed to the user were based on real-time stock information, achieved using PHP, JavaScript (Julian).
- Creating a successful order page displaying the users order information, order number, name, back to them, achieved using PHP, JavaScript, HTML, CSS (Julian/Danielle).
- Testing to ensure the webpage was responsive and able to be viewed on multiple devices (Danielle).

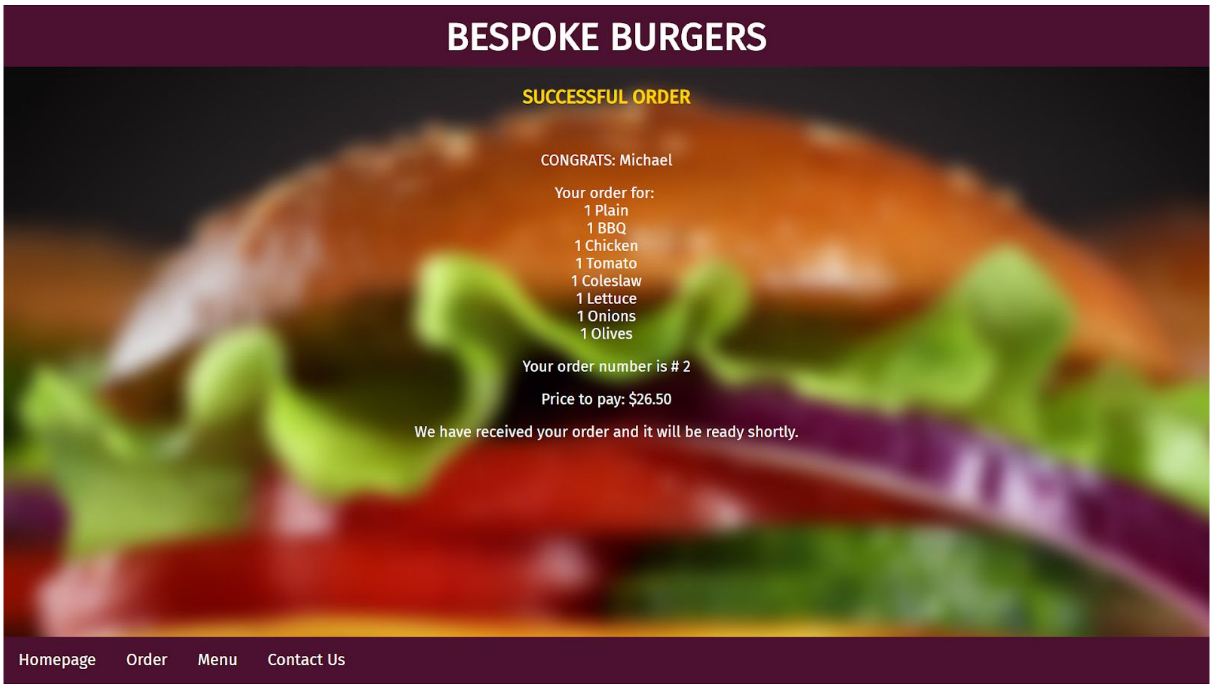
Appendix

A - Ordering Page.



Choose Bun:	Plain		$1.00
Choose Ingredients:			
Tomato:	- 1 +	$4	$4.00
Coleslaw:	- 1 +	$3	$3.00
Lettuce:	- 1 +	$2	$2.00
Onions:	- 1 +	$4.5	$4.50
Olives:	- 1 +	$5	$5.00
Choose Sauce:	BBQ		$4.00
Choose Patty:	Chicken		$3.00
Customer Name:	Michael		
Total Cost:			$26.50

B - Successful Order Page.



SUCCESSFUL ORDER

CONGRATS: Michael

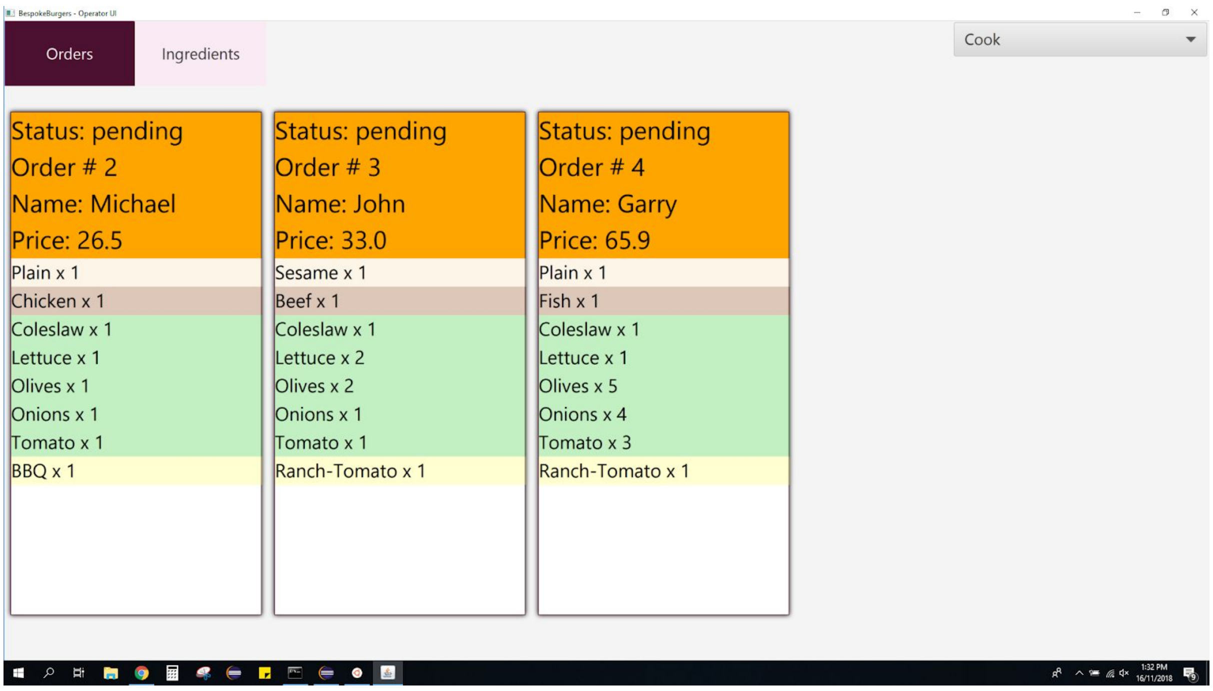
Your order for:
1 Plain
1 BBQ
1 Chicken
1 Tomato
1 Coleslaw
1 Lettuce
1 Onions
1 Olives

Your order number is # 2

Price to pay: $26.50

We have received your order and it will be ready shortly.

C - Cooks view.

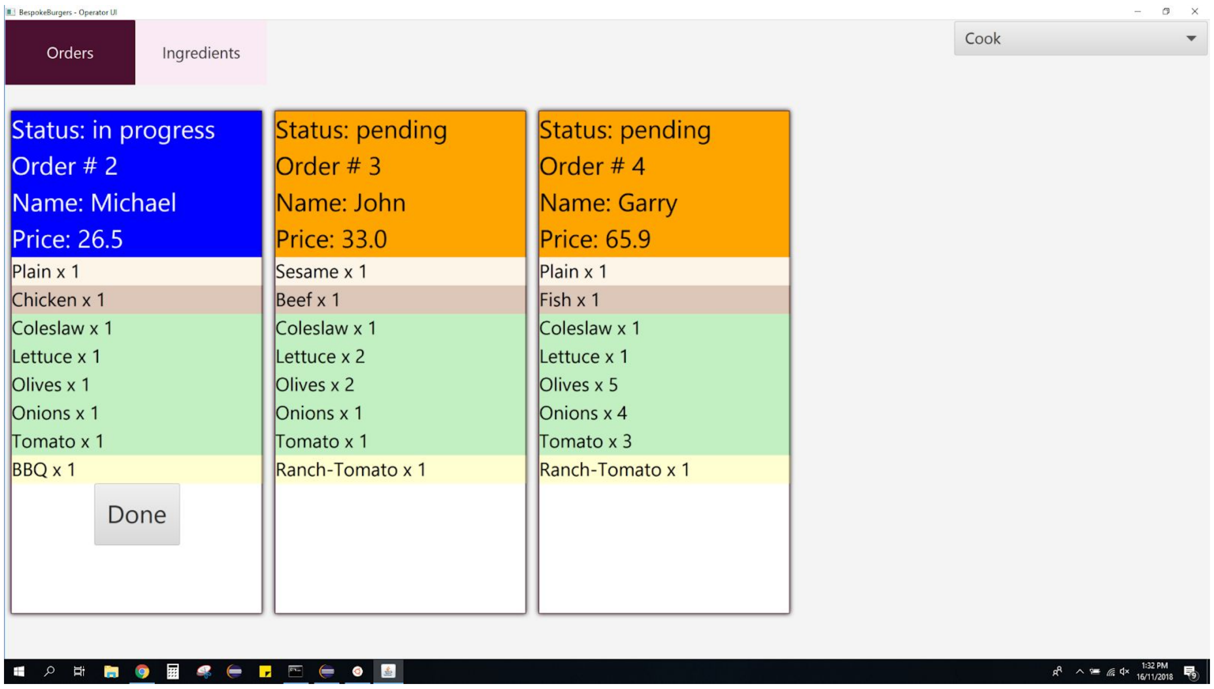


Status: pending
Order # 2
Name: Michael
Price: 26.5
Plain x 1
Chicken x 1
Coleslaw x 1
Lettuce x 1
Olives x 1
Onions x 1
Tomato x 1
BBQ x 1

Status: pending
Order # 3
Name: John
Price: 33.0
Sesame x 1
Beef x 1
Coleslaw x 1
Lettuce x 2
Olives x 2
Onions x 1
Tomato x 1
Ranch-Tomato x 1

Status: pending
Order # 4
Name: Garry
Price: 65.9
Plain x 1
Fish x 1
Coleslaw x 1
Lettuce x 1
Olives x 5
Onions x 4
Tomato x 3
Ranch-Tomato x 1

D - Cook changes Status, done button displays.

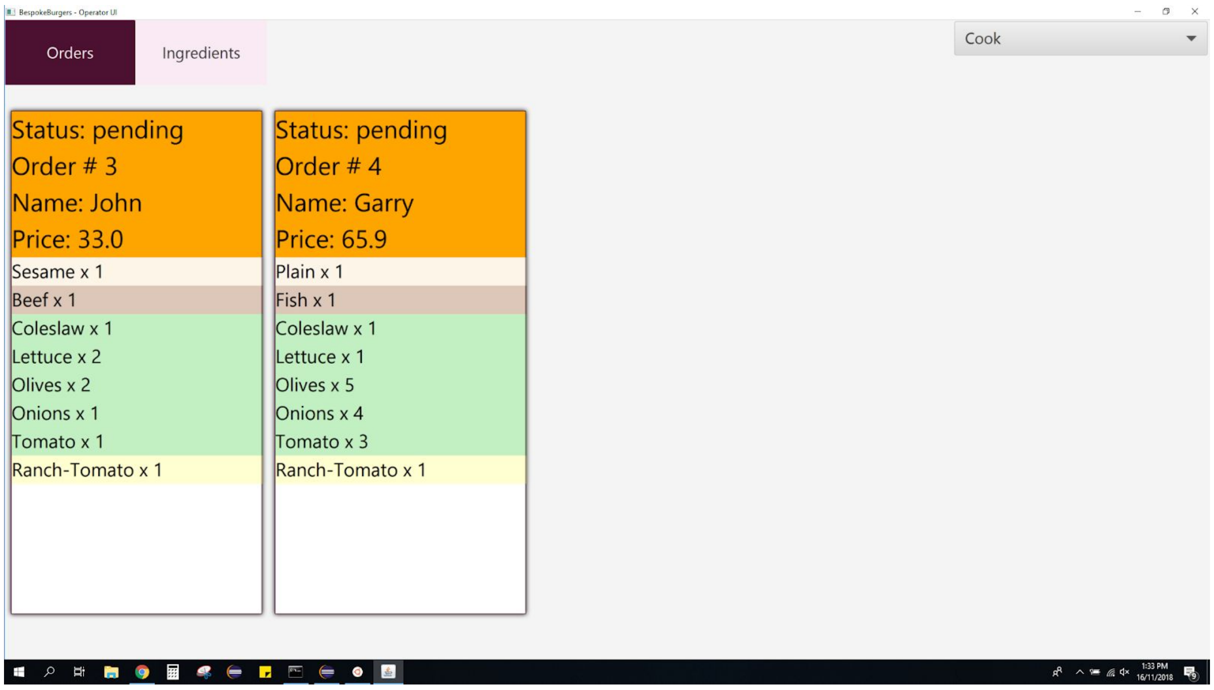


Status: in progress
Order # 2
Name: Michael
Price: 26.5
Plain x 1
Chicken x 1
Coleslaw x 1
Lettuce x 1
Olives x 1
Onions x 1
Tomato x 1
BBQ x 1

Status: pending
Order # 3
Name: John
Price: 33.0
Sesame x 1
Beef x 1
Coleslaw x 1
Lettuce x 2
Olives x 2
Onions x 1
Tomato x 1
Ranch-Tomato x 1

Status: pending
Order # 4
Name: Garry
Price: 65.9
Plain x 1
Fish x 1
Coleslaw x 1
Lettuce x 1
Olives x 5
Onions x 4
Tomato x 3
Ranch-Tomato x 1

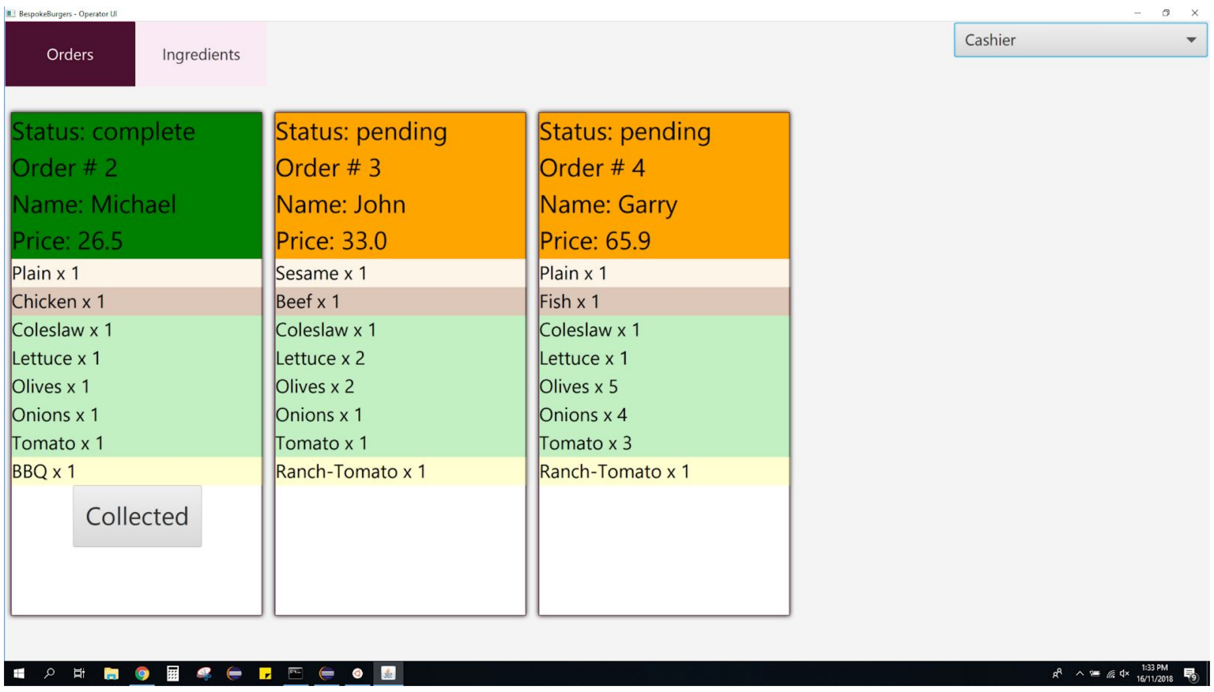
E - When done is clicked, order leaves cook screen.



Status: pending
Order # 3
Name: John
Price: 33.0
Sesame x 1
Beef x 1
Coleslaw x 1
Lettuce x 2
Olives x 2
Onions x 1
Tomato x 1
Ranch-Tomato x 1

Status: pending
Order # 4
Name: Garry
Price: 65.9
Plain x 1
Fish x 1
Coleslaw x 1
Lettuce x 1
Olives x 5
Onions x 4
Tomato x 3
Ranch-Tomato x 1

F - Cashier view can see complete orders ready for collection.


BespokeBurgers - Operator UI

Orders | Ingredients

Cashier

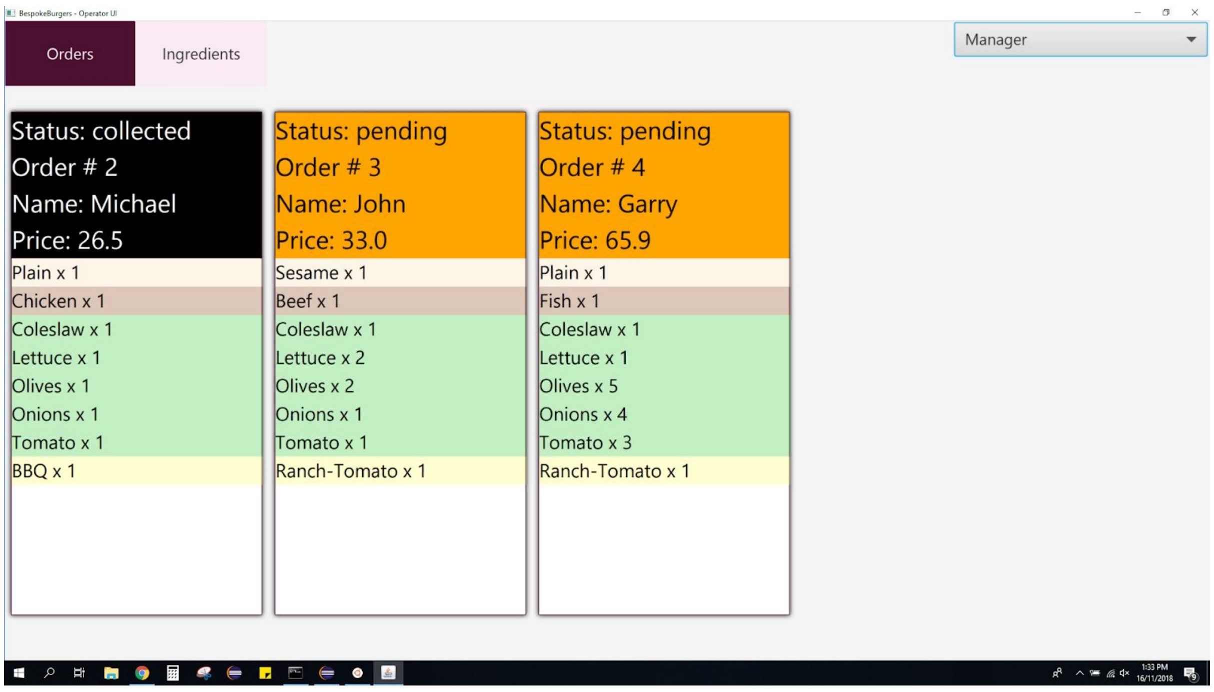
Status: complete
Order # 2
Name: Michael
Price: 26.5
Plain x 1
Chicken x 1
Coleslaw x 1
Lettuce x 1
Olives x 1
Onions x 1
Tomato x 1
BBQ x 1
Collected

Status: pending
Order # 3
Name: John
Price: 33.0
Sesame x 1
Beef x 1
Coleslaw x 1
Lettuce x 2
Olives x 2
Onions x 1
Tomato x 1
Ranch-Tomato x 1

Status: pending
Order # 4
Name: Garry
Price: 65.9
Plain x 1
Fish x 1
Coleslaw x 1
Lettuce x 1
Olives x 5
Onions x 4
Tomato x 3
Ranch-Tomato x 1

1:33 PM 16/11/2018

G - Manager view can see status of all orders for that day.


BespokeBurgers - Operator UI

Orders | Ingredients

Manager

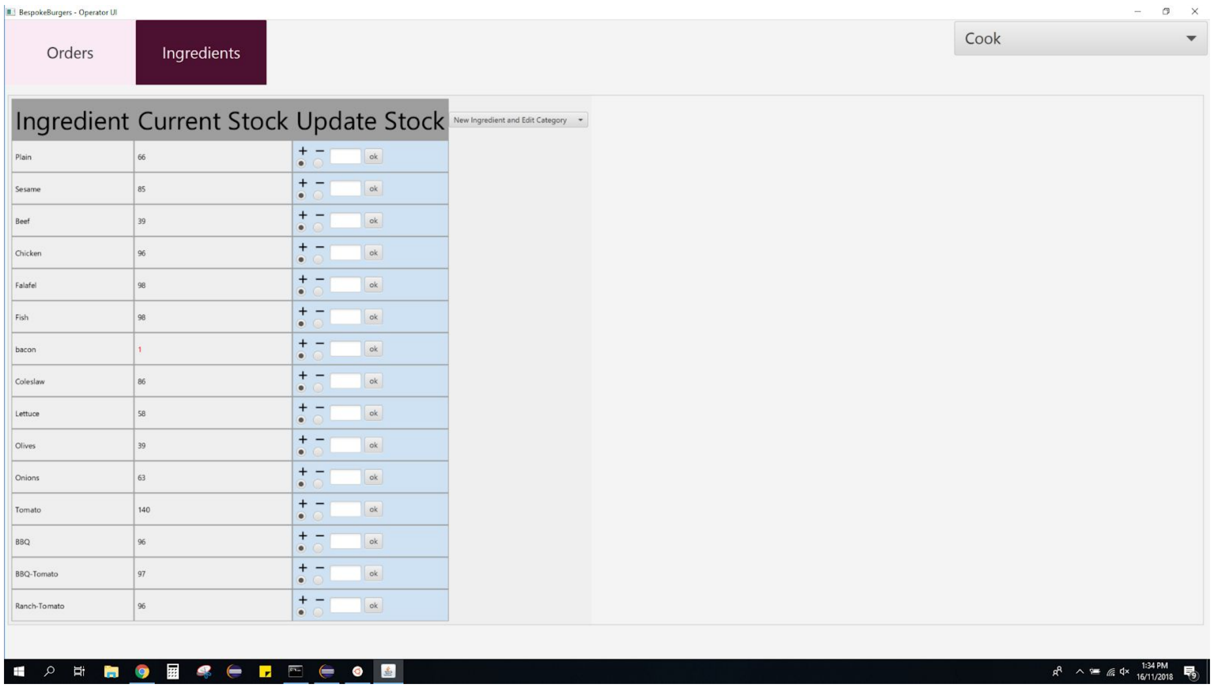
Status: collected
Order # 2
Name: Michael
Price: 26.5
Plain x 1
Chicken x 1
Coleslaw x 1
Lettuce x 1
Olives x 1
Onions x 1
Tomato x 1
BBQ x 1

Status: pending
Order # 3
Name: John
Price: 33.0
Sesame x 1
Beef x 1
Coleslaw x 1
Lettuce x 2
Olives x 2
Onions x 1
Tomato x 1
Ranch-Tomato x 1

Status: pending
Order # 4
Name: Garry
Price: 65.9
Plain x 1
Fish x 1
Coleslaw x 1
Lettuce x 1
Olives x 5
Onions x 4
Tomato x 3
Ranch-Tomato x 1

1:33 PM 16/11/2018

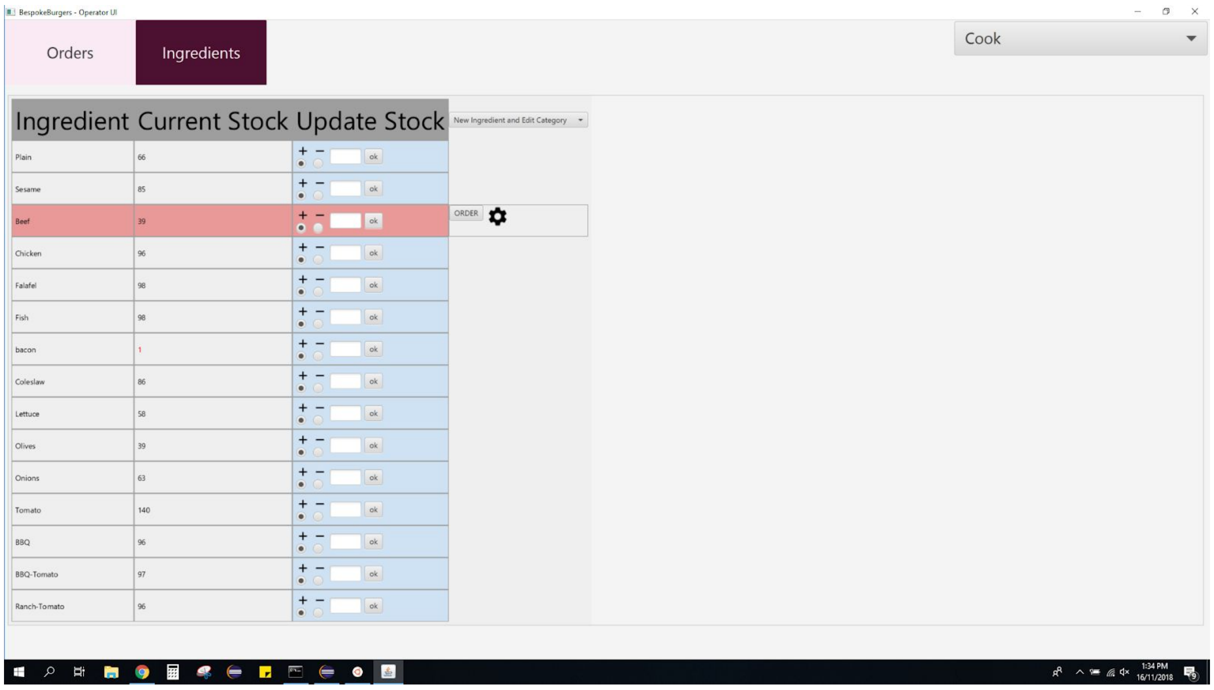
H - Ingredients tab displays current stock holdings.


BespokeBurgers - Operator UI

Ingredient	Current Stock	Update Stock
Plain	66	+ − ok
Sesame	85	+ − ok
Beef	39	+ − ok
Chicken	96	+ − ok
Falafel	98	+ − ok
Fish	98	+ − ok
bacon	1	+ − ok
Coleslaw	86	+ − ok
Lettuce	58	+ − ok
Olives	39	+ − ok
Onions	63	+ − ok
Tomato	140	+ − ok
BBQ	96	+ − ok
BBQ-Tomato	97	+ − ok
Ranch-Tomato	96	+ − ok

1:34 PM 16/11/2018

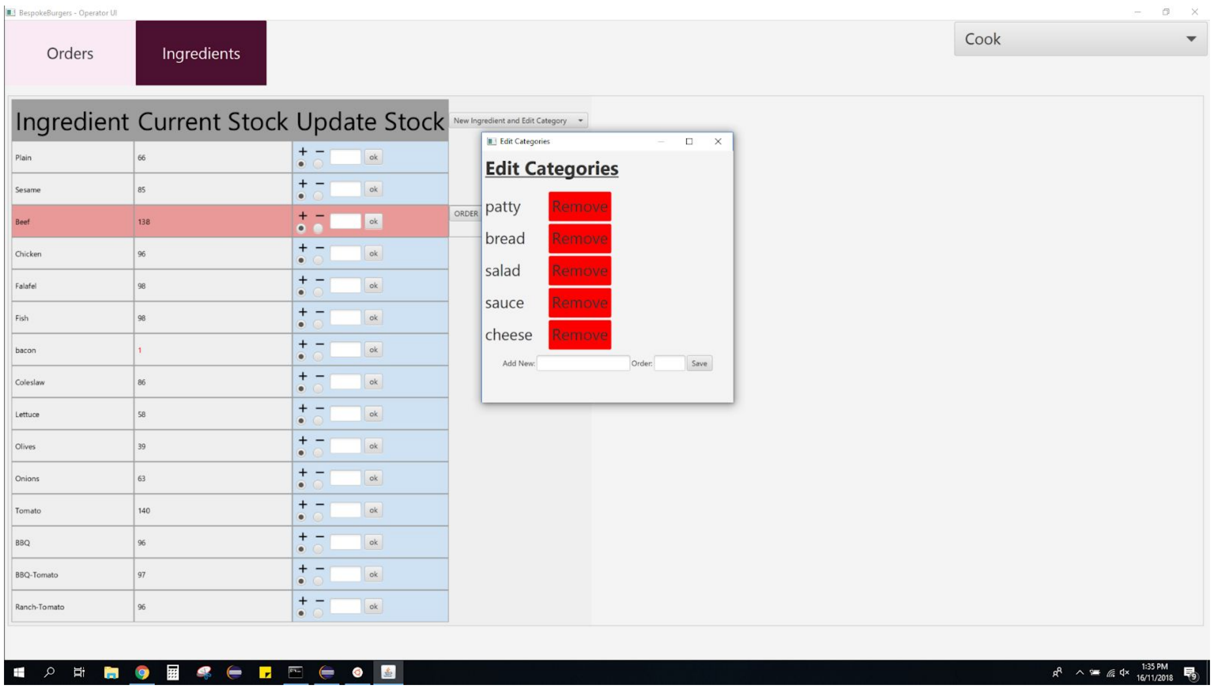
I - Selected row displays in red and gives options to the right of the column. Stock can be increased or decreased from this window.


BespokeBurgers - Operator UI

Ingredient	Current Stock	Update Stock
Plain	66	+ − ok
Sesame	85	+ − ok
Beef	39	+ − ok
Chicken	96	+ − ok
Falafel	98	+ − ok
Fish	98	+ − ok
bacon	1	+ − ok
Coleslaw	86	+ − ok
Lettuce	58	+ − ok
Olives	39	+ − ok
Onions	63	+ − ok
Tomato	140	+ − ok
BBQ	96	+ − ok
BBQ-Tomato	97	+ − ok
Ranch-Tomato	96	+ − ok

1:34 PM 16/11/2018

J - Categories can be added to or removed if Restaurant wants to serve new types of food or remove old types.


BespokeBurgers - Operator UI
Orders
Ingredients
Cook
New Ingredient and Edit Category
ORDER
ok

Ingredient	Current Stock	Update Stock
Plain	66	
Sesame	85	
Beef	138	
Chicken	96	
Falafel	98	
Fish	98	
bacon	1	
Coleslaw	86	
Lettuce	58	
Olives	39	
Onions	63	
Tomato	140	
BBQ	96	
BBQ-Tomato	97	
Ranch-Tomato	96	

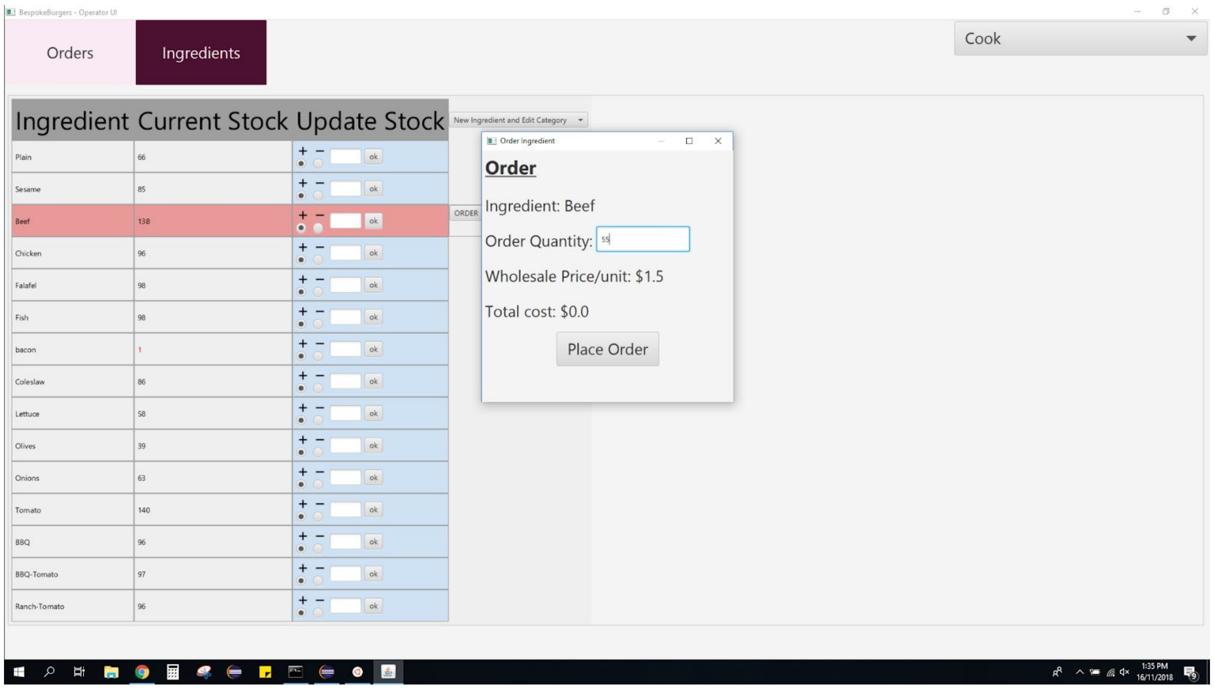
Edit Categories

Edit Categories

patty	Remove
bread	Remove
salad	Remove
sauce	Remove
cheese	Remove

Add New: Order: Save

K - Stock can be re-ordered manually. If stock reaches a minimum level, then ingredient will show as Red to all user types. This will re-prompt an order.


BespokeBurgers - Operator UI
Orders
Ingredients
Cook
New Ingredient and Edit Category
ORDER
ok

Ingredient	Current Stock	Update Stock
Plain	66	
Sesame	85	
Beef	138	
Chicken	96	
Falafel	98	
Fish	98	
bacon	1	
Coleslaw	86	
Lettuce	58	
Olives	39	
Onions	63	
Tomato	140	
BBQ	96	
BBQ-Tomato	97	
Ranch-Tomato	96	

Order Ingredient

Order

Ingredient: Beef

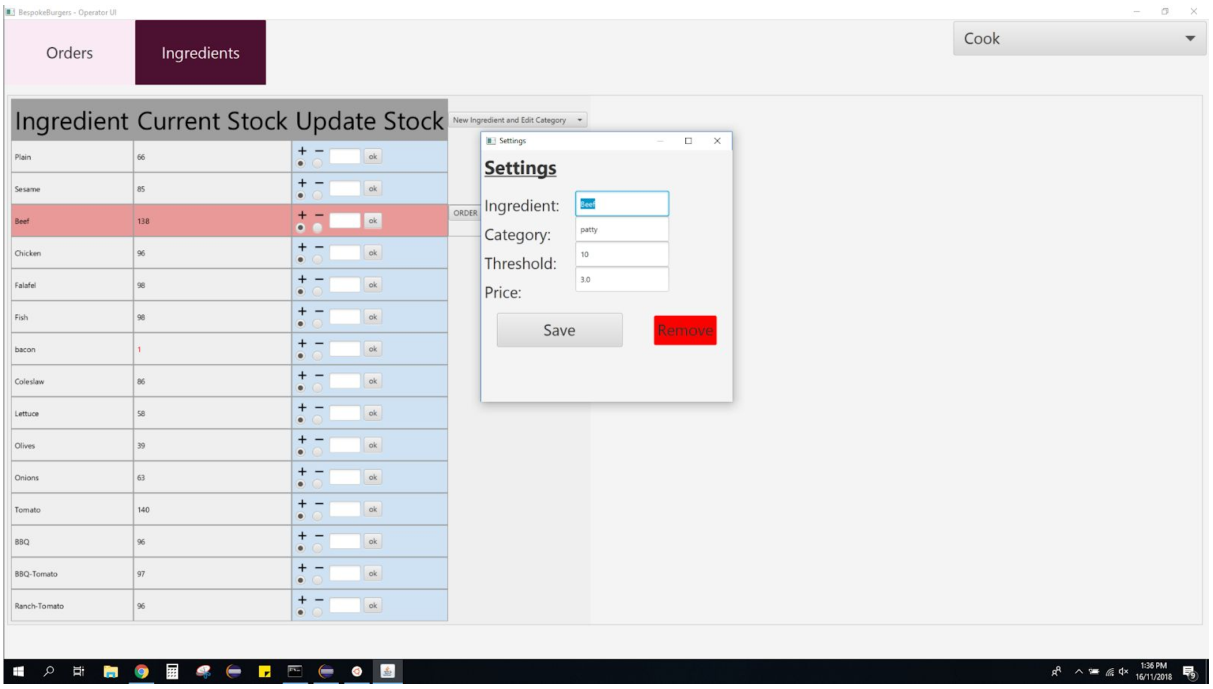
Order Quantity: 55

Wholesale Price/unit: $1.5

Total cost: $0.0

Place Order

L - Settings allows user to change prices, thresholds of categories/ingredients.

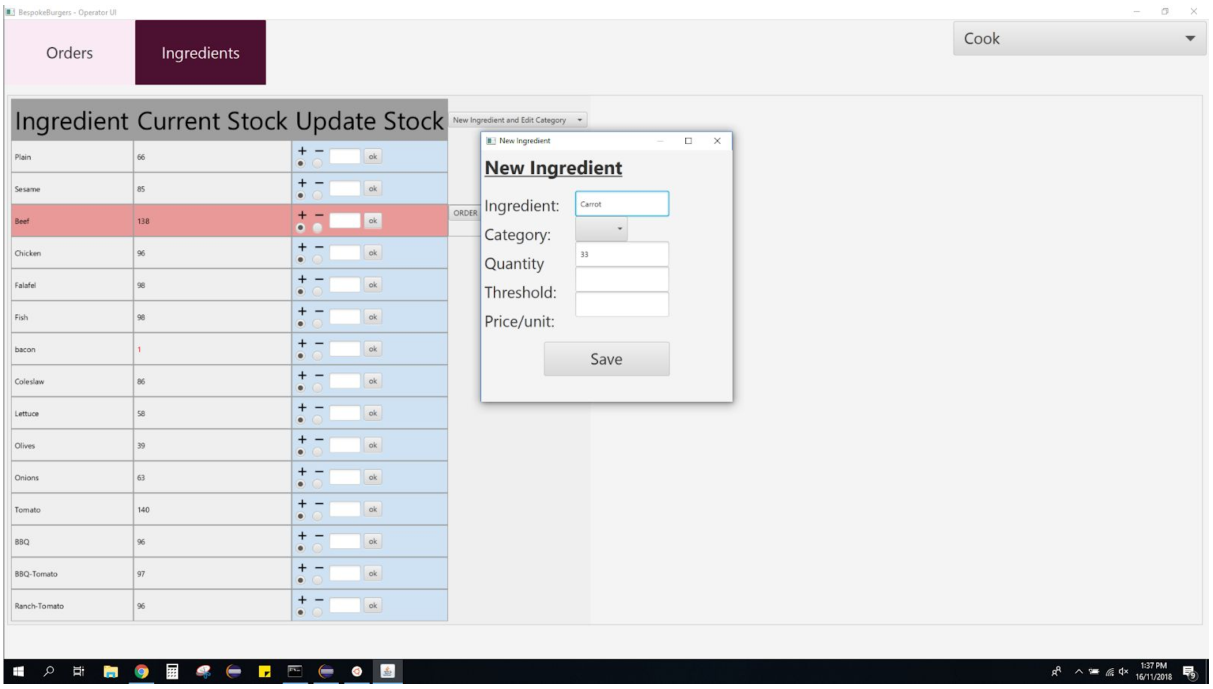

BespokeBurgers - Operator UI

Ingredient	Current Stock	Update Stock
Plain	66	+ − ok
Sesame	85	+ − ok
Beef	138	+ − ok
Chicken	96	+ − ok
Falafel	98	+ − ok
Fish	98	+ − ok
bacon	1	+ − ok
Coleslaw	86	+ − ok
Lettuce	58	+ − ok
Olives	39	+ − ok
Onions	63	+ − ok
Tomato	140	+ − ok
BBQ	96	+ − ok
BBQ-Tomato	97	+ − ok
Ranch-Tomato	96	+ − ok

Settings

Ingredient:	Beef
Category:	patty
Threshold:	10
Price:	3.0

M - New ingredients/categories can be added.

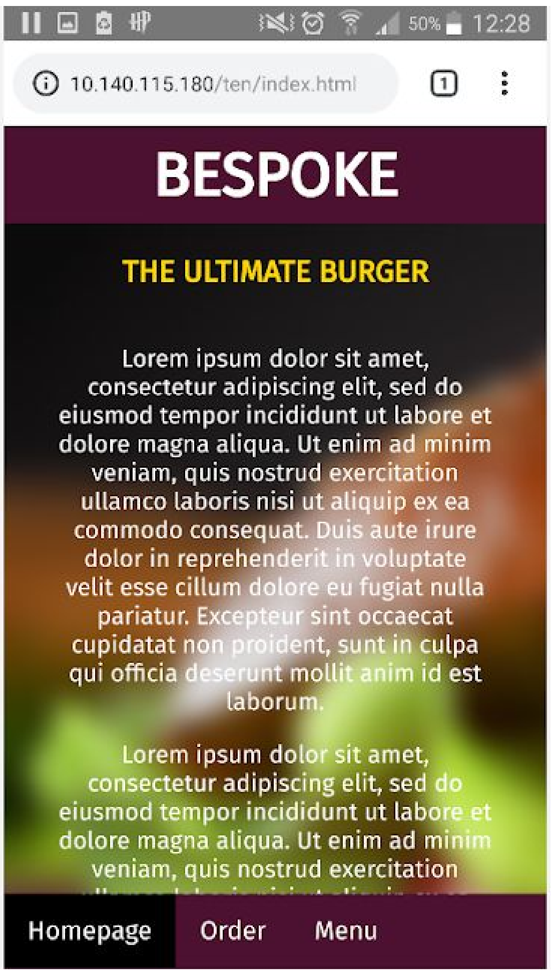

BespokeBurgers - Operator UI

Ingredient	Current Stock	Update Stock
Plain	66	+ − ok
Sesame	85	+ − ok
Beef	138	+ − ok
Chicken	96	+ − ok
Falafel	98	+ − ok
Fish	98	+ − ok
bacon	1	+ − ok
Coleslaw	86	+ − ok
Lettuce	58	+ − ok
Olives	39	+ − ok
Onions	63	+ − ok
Tomato	140	+ − ok
BBQ	96	+ − ok
BBQ-Tomato	97	+ − ok
Ranch-Tomato	96	+ − ok

New Ingredient

Ingredient:	Carrot
Category:	
Quantity	33
Threshold:	
Price/unit:	

N - Mobile Compatible.

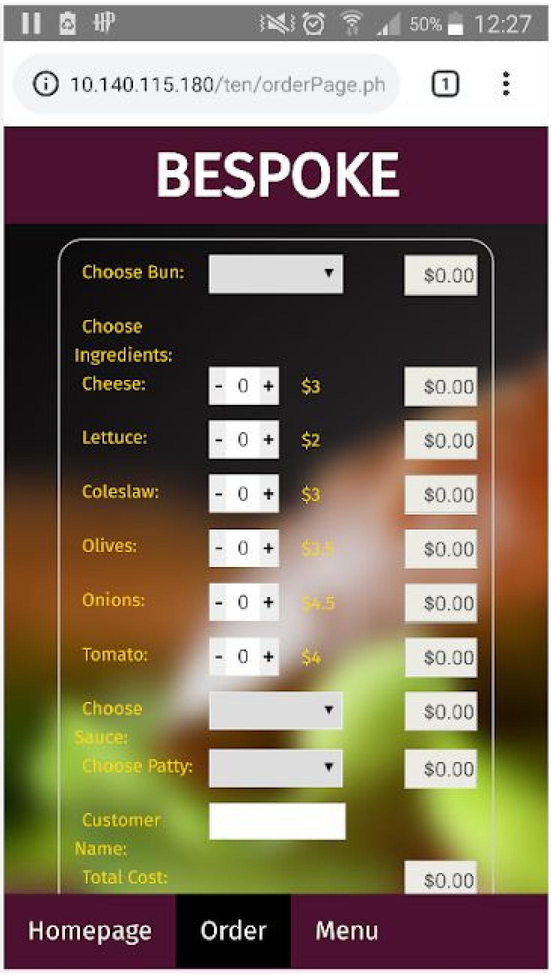


BESPOKE

THE ULTIMATE BURGER

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation



BESPOKE

Choose Bun:			$0.00
Choose Ingredients:			
Cheese:	- 0 +	$3	$0.00
Lettuce:	- 0 +	$2	$0.00
Coleslaw:	- 0 +	$3	$0.00
Olives:	- 0 +	$3.5	$0.00
Onions:	- 0 +	$4.5	$0.00
Tomato:	- 0 +	$4	$0.00
Choose Sauce:			$0.00
Choose Patty:			$0.00
Customer Name:			
Total Cost:			$0.00