

Stack Over Burger

TECHNICAL REPORT

Authors: Bevan Shaw, Dantong Huang, Aidan Brock, Carlos Arenas & Sarah Turner

1.0 Introduction

1.1 Business Objective

Stack Over Burger is a small burger business specializing in gourmet, custom burgers in a single store entrepreneurial venture. The business has been proving popular and requires a more modern solution to facilitate customer ordering and efficiently managing production in the kitchen to meet demand.

1.2 Solution Overview

The purpose of the designed system is to allow “Stack OverBurger” the ability to keep up with the increasing demand for their products. .

The web application allows customers to order custom burgers online, from any mobile, tablet or desktop device.

The production-line application (Java based application) was required by the kitchen staff to control incoming orders, process orders as they are completed, and the ability for certain staff to manage inventory levels.

A dedicated database supports these two applications by storing information relating to their customers, staff, order history and current inventory levels.

The Stack OverBurger system involves multiple components and real-time connections to a central database.

2.0 Technical Architecture

2.1 Database Architecture

For Stack Over Burger's system we decided to use the Firebase Realtime Database which is a cloud-hosted database. We had to learn about the way Firebase Realtime Databases format and store their data. They use a key and value pair to store data, which can then be retrieved as JSON objects.

Because our production-line application is a Java-based application we used the Firebase REST API, which provides a URL REST endpoint. This meant that we could append a .json to the end of the URL and send a request from our HTTP client. We decided to use the OkHttp library as our HTTP client. The OkHttp library has the inbuilt methods PUT, PATCH and GET to manipulate data from the .json URL in the Firebase database. The PUT method allowed us to write or replace the database data at a defined path, e.g. ingredient/avocado/quantity/<data>. The PATCH method was a convenient way to update some of the keys for a defined path without replacing all of the data. This made the structure of the data in the database very flexible and easy to change as we progressed through the project. The GET method allowed us to retrieve the data from our Firebase database by issuing a GET request to its URL endpoint. After we used the GET method to retrieve the data from the database, we used a Map to store the data in our java project.

For the customer web application we connected to the same Firebase Realtime Database this time using the JavaScript SDK (software development kit). Because the Firebase realtime database has a serverless architecture we were able to make changes relatively quickly. Although we were following the waterfall model of software development for this assignment, the serverless architecture provided an agile way to implement our application as we were able to quickly and continuously improve our implementation throughout the project. We implemented HTML, CSS and JavaScript in our customer web application. We also imported the Bootstrap

library and jQuery to help us to achieve the layout and interaction. We also used LocalStorage to store and access the data between different pages. Lastly, the Firebase Realtime Database provides security by having authentication rules for which users can read or write to the database. During the development we did not set up these authentication rules and instead just configured them to public access allowing anyone to read or write to the database. Making use of the security features of the Firebase Realtime Database would definitely be one of the first things we would implement in future releases.

Below is just a sample of the data in the JSON format, for simplicity purposes it has some entries removed.

```
{
  "customer" : {
    "customer1@gmail~com" : {
      "dob" : "10/10/1990",
      "email" : "customer1@gmail.com",
      "name" : "Bob Marley",
      "password" : "123456",
      "phone" : "19000000"
    },
    "customer2@gmail~com" : {
      "dob" : "1993-09-09",
      "email" : "customer2@gmail.com",
      "name" : "Callum Morris",
      "password" : "password123",
      "phone" : "0226483436"
    }
  },
  "ingredient" : {
    "avocado" : {
      "category" : "vege",
      "price" : 2,
      "quantity" : 100
    },
    "beef" : {
      "category" : "meat",
      "price" : 1,
```

```
    "quantity" : 0
  }
},
"order" : {
  "order1" : {
    "createTime" : 1541928225982,
    "customer" : "customer1@gmail.com",
    "details" : {
      "burger1" : {
        "beef" : 2,
        "mayo" : 1,
        "tomato" : 1
      }
    },
    "modifyTime" : 1541967537001,
    "status" : "complete"
  },
  "order2" : {
    "createTime" : 1542105309928,
    "customer" : "customer1@gmail.com",
    "details" : {
      "burger1" : {
        "lettuce" : 2,
        "sesame" : 1
      }
    },
    "modifyTime" : "1542156984456",
    "status" : "complete"
  }
},
"staff" : {
  "manager1" : {
    "email" : "manager1@gmail.com",
    "name" : "manager1",
    "password" : "managerpass1",
    "type" : "manager"
  },
  "worker1" : {
    "email" : "worker1@gmail.com",
    "name" : "worker1",
    "password" : "workerpass1",
    "type" : "worker"
  }
}
```

```
}  
}
```

2.2 Client Web Application

The client application is a simple website application which makes use of Hypertext Markup Language (HTML), Cascading Style Sheets (CSS), JavaScript (JS) and Bootstrap (Front End Component Library) to create the necessary User Interface for customers to order customised burger.

The order page contains collapsible sections so the complete list of options does not take up too much space on the screen. When one is opened, the previously selected one closes. Given the different operation of each category (only one bun at a time vs. multiple patties of multiple types vs. different sauces but no multiples) different functions had to be created to update the display of what the user had selected. This was also influenced by the fact that initially, the order form was created with only one option from each category able to be selected for each burger, except for the vege and sauce (although only multiple lettuce for example, could not be chosen). The configuration for the bread and sauce was kept the same, so the code for updating the display to reflect the user's choice remained in the loadOptions() function. The rest were changed and so they received their own separate functions.

An example of one of the new functions:

```
function updateVege(){  
    vegeFinal = "";  
    for(var i = 0; i < vegeChoice.length; i++){  
        if(vegeChoice[i] > 0){  
            switch (i){  
                case 0: vegeFinal = vegeFinal + "Lettuce x " + vegeChoice[i] + ", ";  
                    break;  
                case 1: vegeFinal = vegeFinal + "Tomato x " + vegeChoice[i] + ", ";  
                    break;  
                case 2: vegeFinal = vegeFinal + "Pickles x " + vegeChoice[i] + ", ";  
                    break;
```

```

        case 3: vegeFinal = vegeFinal + "Avocado x " + vegeChoice[i] + ", ";
        break;
    }
}
}
vegeFinal = vegeFinal.substring(0, vegeFinal.length-2);
document.getElementById("vegeButton").innerHTML = "Vege: " + vegeFinal + "
($" + vegeCost + ")";
addOptions();
}

```

The vegeChoice, vegeFinal and vegeCost variables were made global so they could be easily accessed by any function that required them without any issue.

2.3 Production Application

A system requirement was for the Production Line Application be coded in Java with a Graphical User Interface. To store the data and create object in our application, we created three data model which is Order, Burger and Ingredient. To adapt the key and value Firebase Realtime Database System, We used Map as our data structure. Connector class behaves like a connector to retrieve and update data from the database. For our system we used JavaFX Scene Builder to layout the four Graphical User Interfaces which include: a log-in user interface, a registration user interface, an inventory user interface and a worker user interface.

The log-in user interface allows valid staff (a worker or a manager) to log-in to the system. If the manger logs in they go directly to the inventory user interface while a worker will go directly to the worker user interface. If a user can't log in they can also click the link to be taken to the registration user interface.

The registration user interface allows a user to make a new valid email and password combination which will be stored in the database. Once they have completed registration they are directed back to the log-in user interface.

The inventory user interface allows the updating of a single burger ingredient or multiple burger ingredients in the database. Labels will change in response to

ingredients dropping below 20 and also if ingredients are out of stock. Any ingredient with less than 5 in stock will disappear from the customer web application burger order page. There is a refresh button to view the current quantities of ingredients in the database.

The worker user interface allows staff to get the burger orders with the status pending from the database. The individual orders can then be clicked to show the details of the order. Orders can then be completed which will update the status in the database and remove them from the pending orders list.

There is a navigation button in both the inventory user interface and the worker user interface to allow navigation between the pages. There is also a logout button in both the inventory user interface and the worker user interface which directs the user back to the log-in user interface.

Please see the section “5.0 Application Functionality” for screenshots and full explanation of the functionalities.

3.0 Changes Made From Original Planning & Design Stage

In the course of developing the three-component system the team found a few aspects of our original design would need to be altered to accommodate changes or roadblocks we came across in the development phase.

In our initial design we had chosen to design the system with a SQL relational table database. The reasoning behind this was due to the team’s comfort levels and prior experience in using SQL relational databases and the availability of PostgreSQL on the university servers.

The database the development group settled on utilising for the current system is a NoSQL database which utilises key-value pairing for data in a JSON format through Google Firebase.

This deviation was accepted by the group when issues surrounding the use of a member's private hosting of SQL server was not accessible, as well as complications surrounding the PostgreSQL database server supplied to us was not allowing access calls from outside the university's WiFi or from PHP scripts.

The team learnt to adapt their thinking quickly from SQL to a NoSQL environment, and to trust in other team members knowledge and guidance in an area where the general consensus was to stick with "what we know". The switch to Firebase required most of the team to re-learn basic concepts about NoSQL and the different connections used between the database and the web application.

Despite handling the learning curve that came with handling JSON objects to work with the Database and Web application, Firebase was found to be a more flexible platform in terms of saving data, and gave us the ability to adapt our data structure in the JSON to meet the application needs. There were several changes and additions to the data structure from Thursday through to Monday as we realised necessary details or changes were needed. This would then require changes to the code and time delays. For example, the java data model classes in our production line application would then change requiring additions to methods in the code. The team became more accurate at predicting the appropriate amount of time a change would require.

Despite changes in database, our expected notation for a unique username given to every customer would have been the first section of their email address up to the "@" symbol. During development it was detected that this will cause issues with uniqueness as as a potential customer with email address "*customer1@outlook.com*" and "*customer1@gmail.com*" will have identical usernames.

To overcome this the username key given to customers is adapted from their whole email. When storing the unique user email address as a key in the firebase realtime database, we found that we cannot use the original user's email as it contains the

char '.' symbol. This is because firebase uses the URL as the REST endpoint and Ref (reference). Instead we decided to replace the char '.' with the char '~' in order to store the unique email key of the user in the database. For example, *"customer1@gmail.com"* became *"customer1@gmail~com"*. This meant we can store the user email address key and have no issues with duplications of customer identity.

With Firebase Real-Time Database hosting the companies data, the need for server-side PHP scripting was no longer required, and the use of JavaScript to make the connections to the cloud-based database to send and retrieve information as required was used. This change was made easier by the Firebase platform.

A deviation from the planning report (where nothing was mentioned) was the decision to use JavaFx Scene Builder. The reasoning behind the decision was to save time during the design process of the Graphical User Interfaces that we required. JavaFX Scene Builder is a visual layout tool that lets users quickly design JavaFX application user interfaces, without coding (similar to Java Swing WindowBuilder that we had used in the SWEN 502 networking module). Before installing JavaFX Scene Builder via Gluon (<https://gluonhq.com/products/scene-builder/>), we had to make sure we also had the JavaFx library installed on every computer in the team and connected correctly in our java project build paths. Also, noone on the team had used JavaFX Scene Builder before and we had to discover how to find, drag and drop UI components into the work area, modify their properties and apply style sheets. The FXML code for each layout was automatically generated in the background which resulted in an FXML file that we then had to learn how to combine with our Java project by binding the UI to the application's logic (e.g. in the fxml files by using the `fx:id = "uniqueIdentifier"` and `onAction="#methodName"` properties, also matching the `fx:id`'s in the java code and including `@FXML` tags). All these things took time initially but once everything was working it was much faster making changes and overall we

saved significant time. We learned about considering the long-term efficiency and importance of rapidly learning a new tool even if it takes time initially to set up and understand the tool.

In the details of designing the web pages, there was nothing highly specific mentioned. On the order page, there was to be expandable sections for the categories and order buttons. These were implemented as planned. However, there were a number of changes made from what was initially put into the order page. Firstly, the HTML buttons at the head of each section were grey to begin with. The idea was then raised to put a picture on each of them relating to their respective categories to make them visually more interesting and to indicate more clearly what they were. The background was also a light to medium-dark purple gradient. Later, when the Bootstrap library was added to the web pages, this was changed to teal. The appearance of the elements within the page such as the buttons remained the same. Another change was in what options the customer was allowed to select. In the first configuration, they could only choose one bread, one patty, all the vegetables and sauces (although only one instance of each) and one cheese). After a group discussion, the patty, cheese and vegetable sections were changed to allow selection of up to three of every option. The bread and sauces were kept the same. This would allow for a more customisable burger, in accordance with the objective. To this end, the JavaScript code for updating the choice of the customer on-screen for these categories was moved out into separate functions.

Another deviation from our original planning documentation was the inclusion of unit testing our production and web applications. Given a longer production time we would include testing in subsequent planning. The team has learnt more about the waterfall development method and how the risk of over-promising and how scope creep can drag out development times as coding of the applications begins. IF given another chance we would have allocated specific time during development

timeline for just testing and not adding components and functionality in during this time.

4.0 Individual Team Member Contributions

Our team was lucky to have members who were multi-disciplined when handling and willing to handle different aspects of this project and to include others who may not be as skilled in an area to share their knowledge with.

Database

- Data Model - Sarah, Carlos, Bevan, Dantong
- Trial of Postgres Database - Sarah, Carlos, Dantong
- Trial of Private Server / SQL DB - Sarah, Dantong
- Initial Creation of Firebase - Dantong
- Firebase Server
 - Connector - Dantong, Sarah
 - loginType Method - Bevan

Web App

- Login Page HTML, CSS & JS - Sarah, Dantong
- Registration Pages HTML, CSS & JS - Sarah, Dantong
- FireBase connection code - Dantong
- Order Page HTML and CSS - Aidan, Bevan (Design)
- Order Page Javascript functionality - Aidan, Bevan, Dantong
- Payment Screen HTML - Sarah
- Payment Screen CSS and JS - Dantong
- Confirmation Page - Dantong

Java Production Application

- Data Model
 - Burger.java - Bevan
 - Ingredients.java - Bevan
 - Order.java - Bevan, Dantong
- Graphic GUI
 - Login
 - Fxml - Bevan, Carlos

- Java - Bevan, Carlos
- Manager
 - Fxml - Aidan, Bevan, Carlos
 - Java - Bevan
- Worker
 - Fxml - Carlos, Bevan, Dantong
 - Java - Dantong, Carlos, Bevan
- Main.java
 - Fxml - Carlos, Bevan
 - Java - Carlos, Bevan
- Server
 - Connector - Dantong, Bevan
 - OKLMP - Dantong

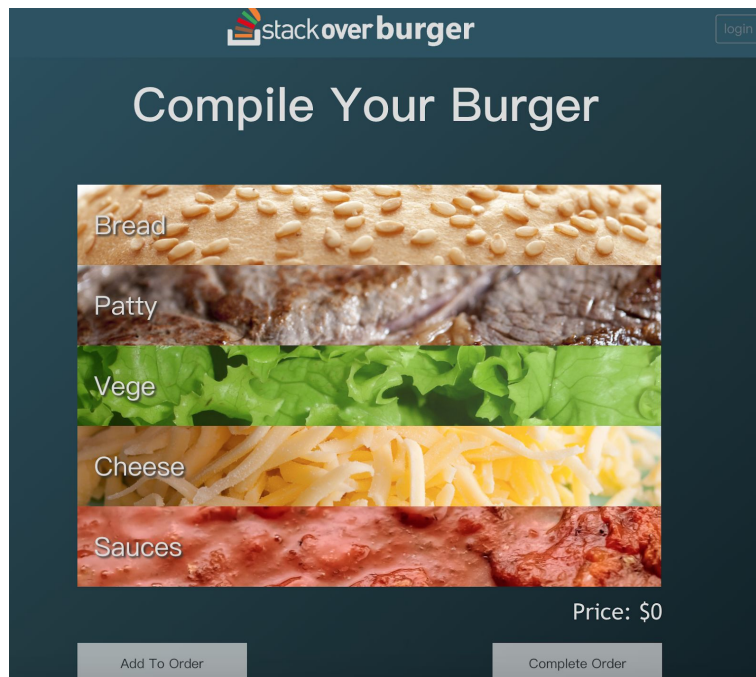
Planning Documentation

- Data ER Diagram - Sarah, Carlos, Dantong, Bevan
- Data Model - Sarah, Carlos, Dantong, Bevan, Aidan
- Use Case Diagram - Sarah, Carlos
- Activity Flow for Burger Ordering (Customer) - Bevan, Dantong, Sarah
- Sequence Diagram - Dantong, Bevan

5.0 Application Functionality

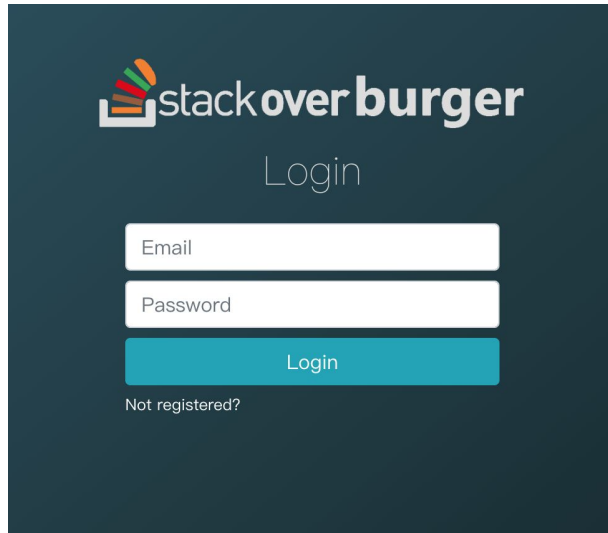
5.1 CUSTOMER WEB APPLICATION FUNCTIONALITY

5.1.1) Customer arrives at ORDER PAGE:

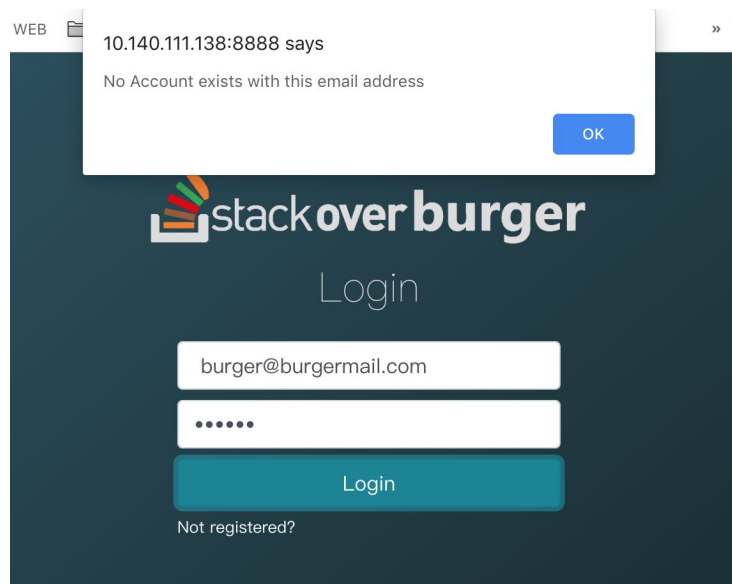


Commentary: The buttons "Add to Order" and "Complete Order" are disabled until the customer logs in to the webpage.

5.1.2) CLICK "LOGIN" BUTTON on ORDER PAGE and go to LOGIN PAGE:




5.1.3) Entering unregistered email address :



5.1.4) CLICK "Not registered" LINK to go to the REGISTRATION PAGE:

Commentary: on the REGISTRATION PAGE once all fields are successfully and the "REGISTER" BUTTON is clicked the user is taken back to the LOGIN PAGE.



stackoverburger

Register

First Name:

Last Name:

Email:

Date of Birth:


Phone:

Enter Password:

Repeat:

Current client?

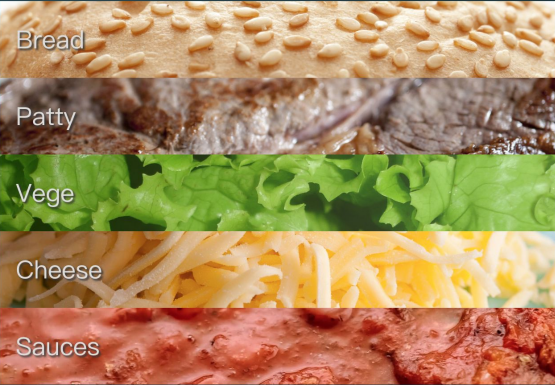
5.1.5) Enter correct user email and password on the LOGIN PAGE + CLICK "LOG IN" BUTTON to be taken to the ORDER PAGE:



stackoverburger

Sarah Turner

Compile Your Burger



Bread

Patty

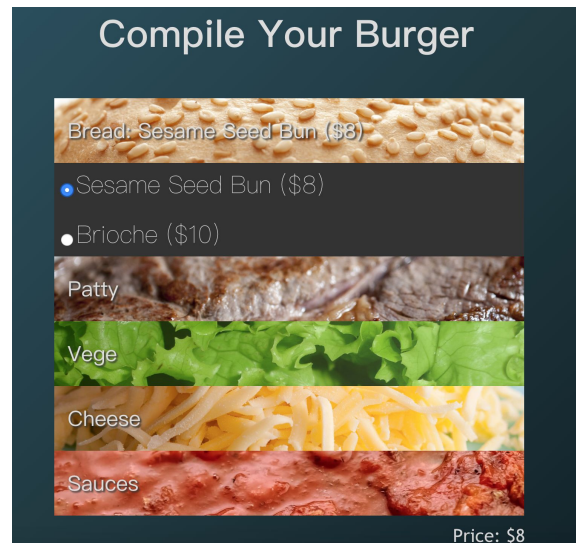
Vege

Cheese

Sauces

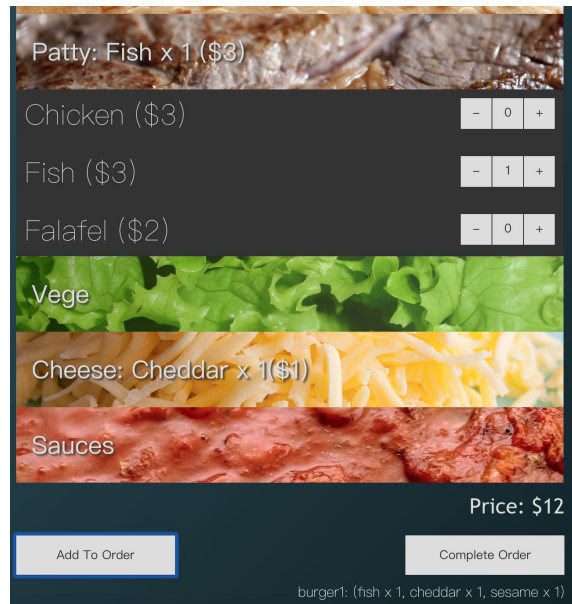
Price: \$0

5.1.6) COMPILE YOUR BURGER:



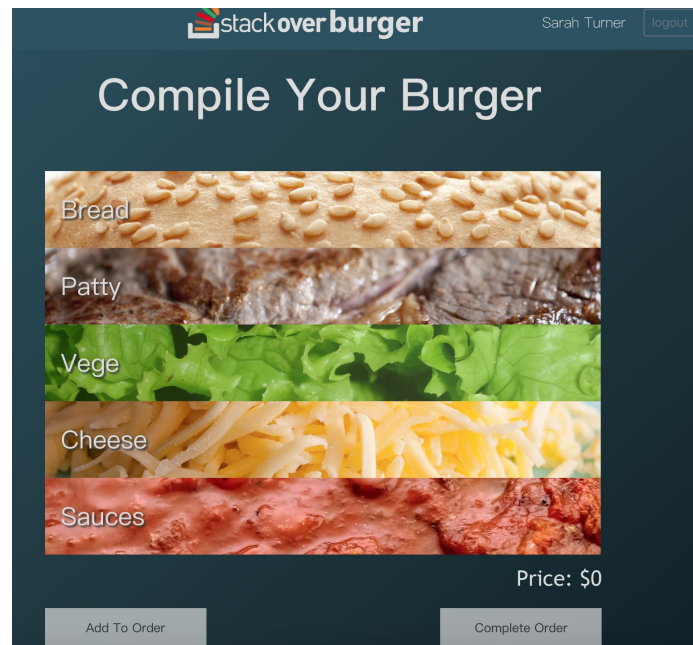
Commentary: At this point you can compile your burger as you want, just choose between Bread, Patty, Veggie, Cheese, or Sauce, the value of each ingredient will be projected.

5.1.7) CLICK “ADD TO ORDER” BUTTON:



Commentary: Once you click the button “Add to Order” the screen will project the list of the ingredients, once ready just click the button “Complete Order” .

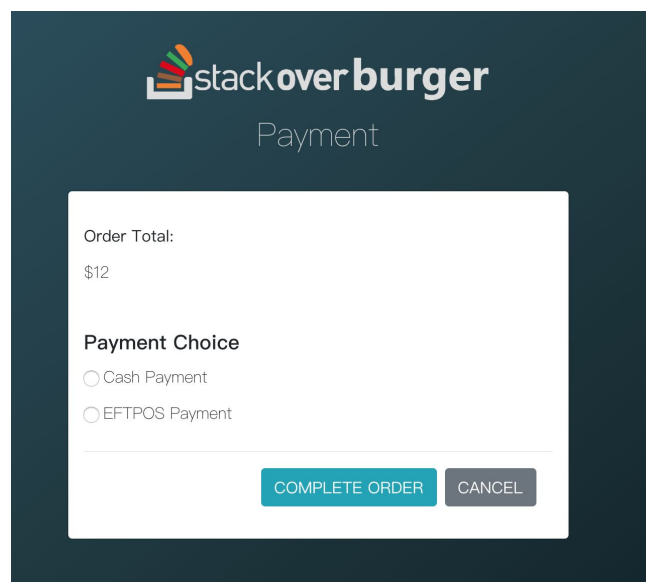
5.1.8) CANCEL THE ORDER:



The screenshot shows the 'Compile Your Burger' interface for 'stackoverburger'. At the top, the logo is on the left and 'Sarah Turner' with a 'logout' button is on the right. The main title 'Compile Your Burger' is centered. Below it, five horizontal sliders represent burger components: Bread (sesame seed bun), Patty (cooked meat), Vege (green lettuce), Cheese (shredded yellow cheese), and Sauces (ketchup and mustard). The 'Price: \$0' is displayed at the bottom right. At the bottom, there are two buttons: 'Add To Order' and 'Complete Order'.

Commentary: you can cancel the order and the system will take you to the “Compile your Burger” again.

5.1.9) COMPLETE THE ORDER:



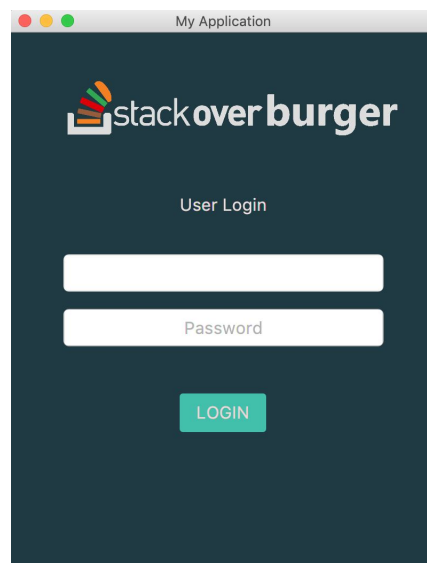
The screenshot shows the 'Payment' interface for 'stackoverburger'. The logo is at the top left. The title 'Payment' is centered. Below it, a white box contains the 'Order Total: \$12'. Underneath, the 'Payment Choice' section has two radio buttons: 'Cash Payment' and 'EFTPOS Payment'. At the bottom of the white box, there are two buttons: 'COMPLETE ORDER' (in teal) and 'CANCEL' (in grey).

5.2 PRODUCTION LINE APPLICATION

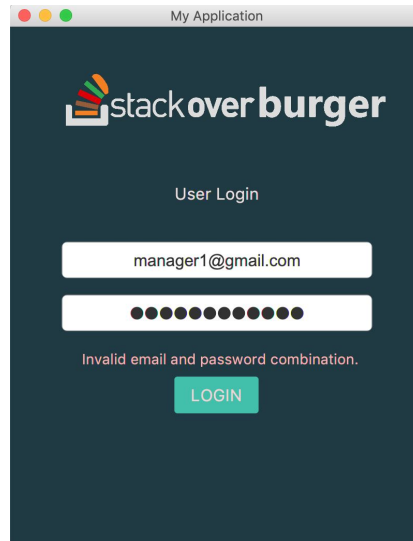
There are 2 types of staff currently available in our system, Managers and Workers.

5.2.1 MANAGER PRODUCTION LINE APPLICATION

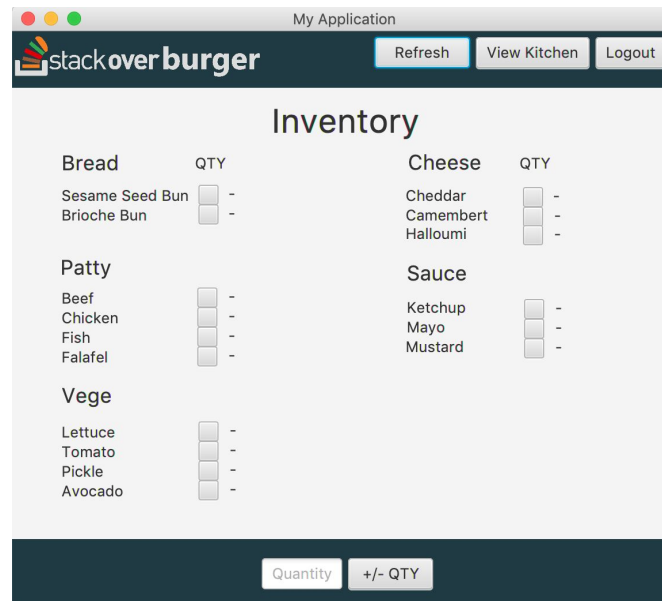
5.2.1.1) LOGIN PAGE:



5.2.1.2) ENTER WRONG EMAIL:



Commentary: error label appears: **“Invalid password and email combination”**
 5.2.1.3) ENTER CORRECT EMAIL + “LOG IN” BUTTON:



Commentary: We can see all the ingredients in inventory page.

5.2.1.4) CLICK “REFRESH” BUTTON:

My Application

stackoverburger Refresh View Kitchen Logout

Inventory

Bread	QTY	Cheese	QTY
Sesame Seed Bun	201	Cheddar	100
Brioche Bun	101	Camembert	100
		Halloumi	100
Patty		Sauce	
Beef	0 - OUT OF S...	Ketchup	100
Chicken	0 - OUT OF S...	Mayo	100
Fish	453	Mustard	100
Falafel	99		
Vege			
Lettuce	100		
Tomato	100		
Pickle	100		
Avocado	100		

Quantity +/- QTY

Commentary: Show the available quantities of every burger ingredient. The system does this by going into the database at the moment the button is clicked to give you the most up-to-date and current information possible.

5.2.1.5) CHECKBOX AN INGREDIENT + CHANGE VALUE BETWEEN 1-4 + CLICK "QTY" BTN:

My Application

stackoverburger Refresh View Kitchen Logout

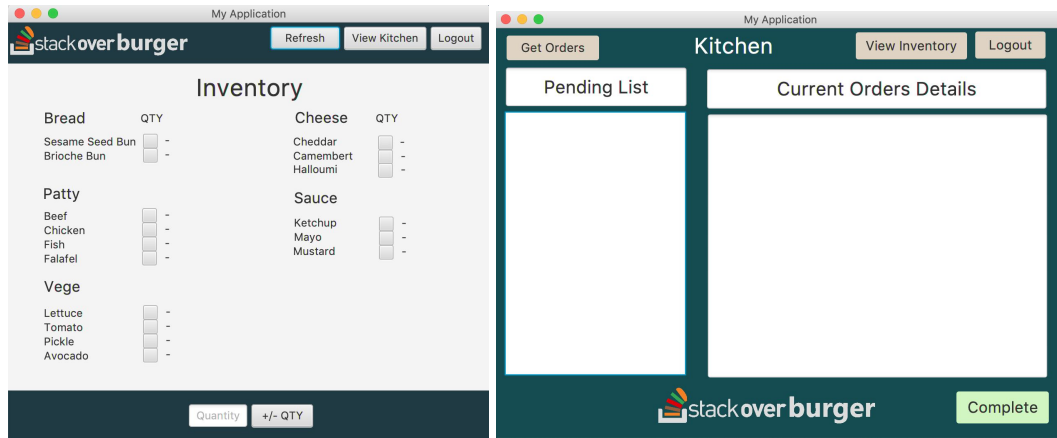
Inventory

Bread	QTY	Cheese	QTY
Sesame Seed Bun	201	Cheddar	100
Brioche Bun	101	Camembert	100
		Halloumi	100
Patty		Sauce	
Beef	200	Ketchup	100
Chicken	200	Mayo	100
Fish	453	Mustard	100
Falafel	2 - REORDER		
Vege			
Lettuce	100		
Tomato	100		
Pickle	100		
Avocado	100		

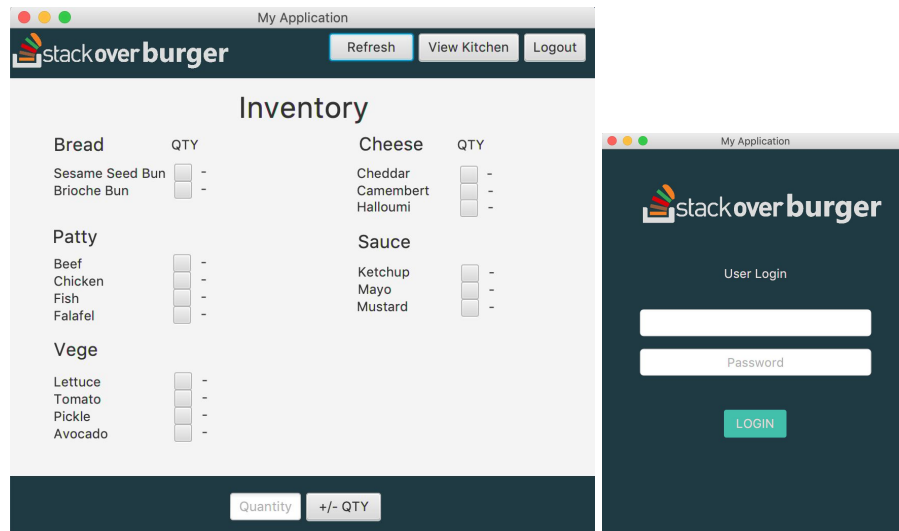
-200 +/- QTY

Commentary: in this case if the order of the orders go down to less than 20, an alert message is displayed to fill stock.

5.2.1.6) CLICK “View Kitchen” BUTTON:

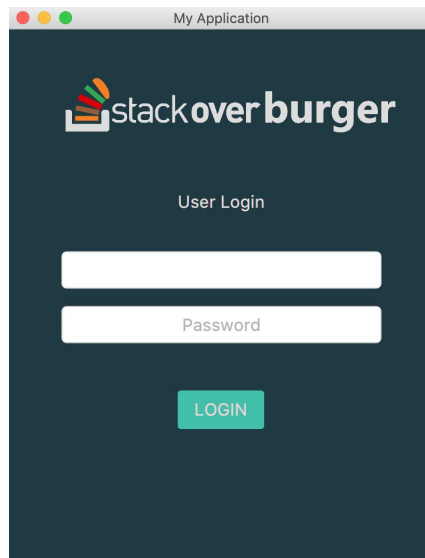


5.2.1.7) CLICK “LOGOUT” BUTTON:

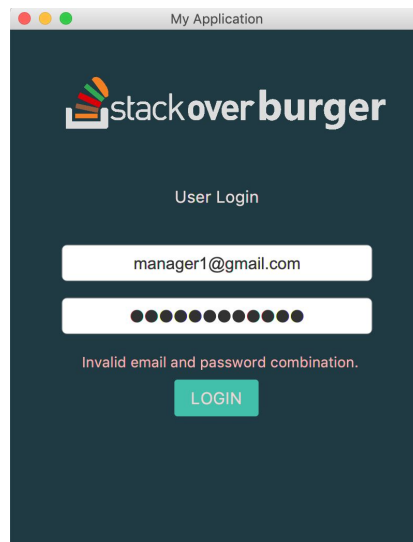


5.2.2 WORKER PRODUCTION LINE APPLICATION

5.2.2.1) LOGIN PAGE:



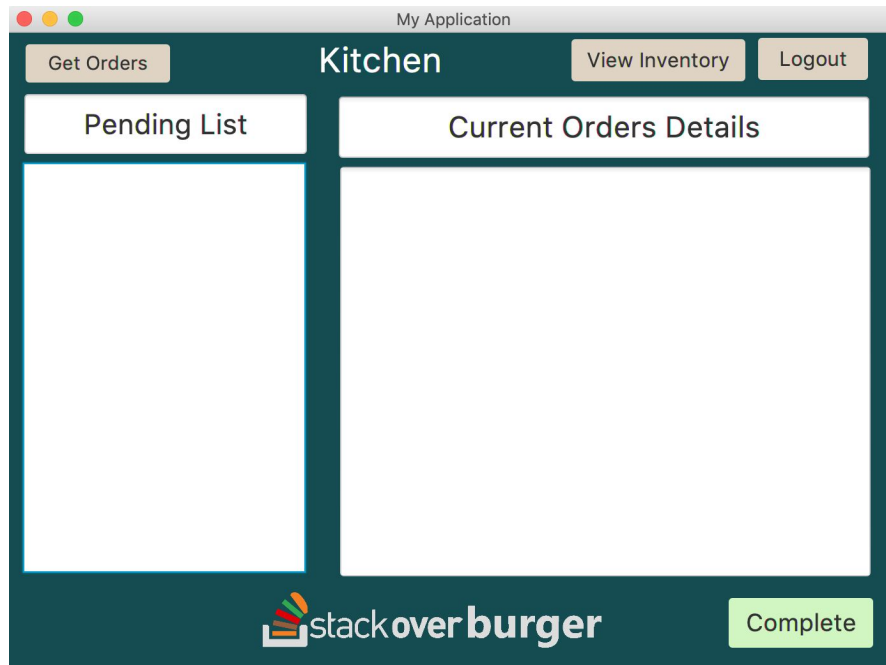
5.2.2.2) ENTER WRONG EMAIL:



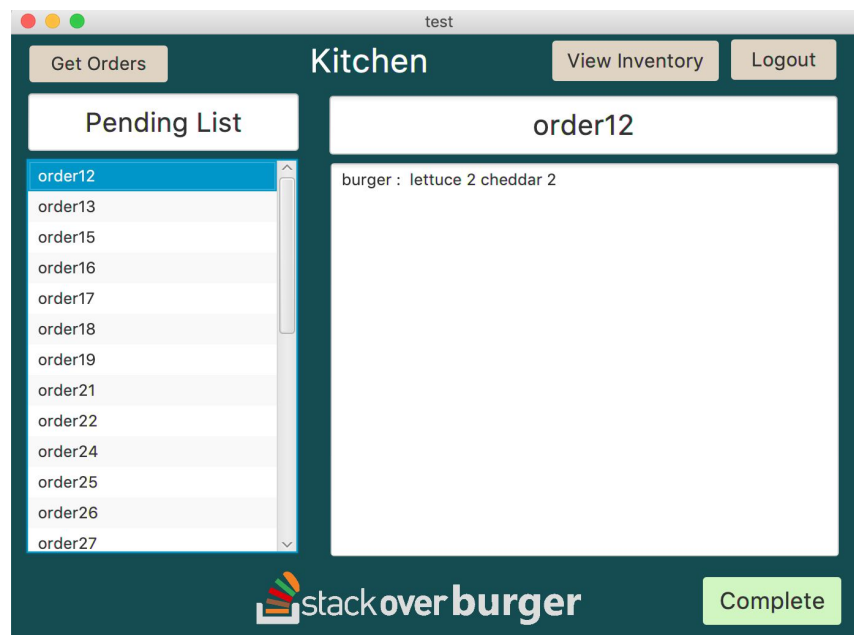
Commentary: error label appears: **“Invalid password and email combination”**

5.2.2.3) ENTER CORRECT WORKER EMAIL + “LOGIN” BUTTON:

Commentary: takes a user to the KITCHEN PAGE

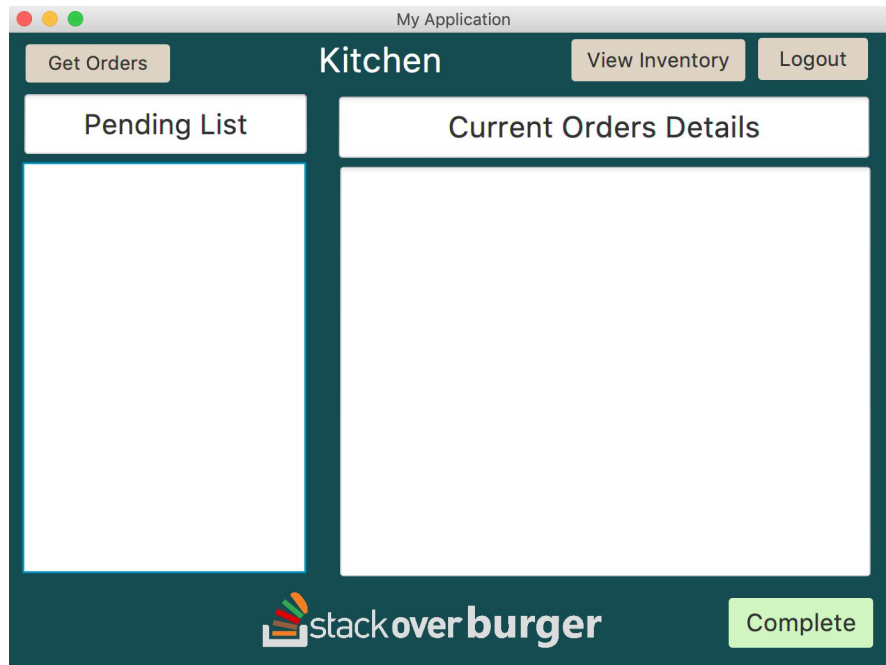


5.2.2.4) CLICK GET ORDERS" BUTTON + CLICK ON AN ORDER IN THE LIST:



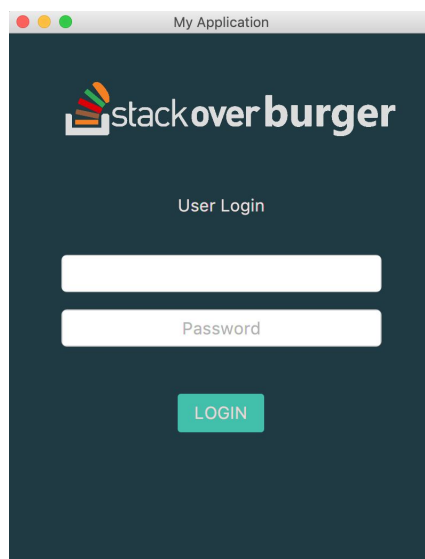
Commentary: when the customer compile the burger, the details of the order are immediately displayed in the Worker Scene.

5.2.2.5) CLICK "COMPLETE" BUTTON:



Commentary: when the worker finish the order, we make a click in complete and the order and the order is deleted from the list

5.2.2.6) CLICK "LOGOUT" BUTTON:

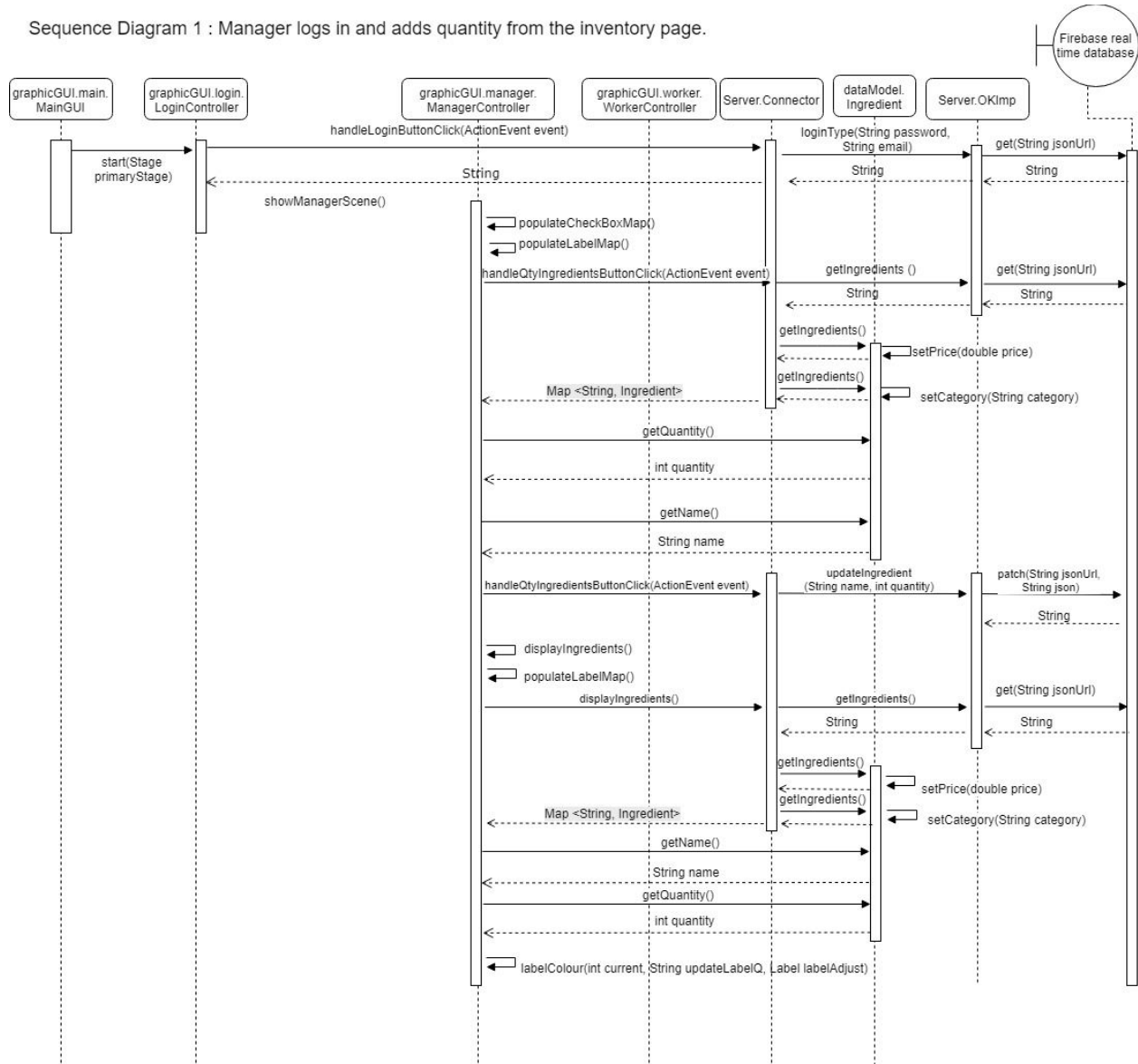


Commentary: When the worker press the button Logout, automatically go to the login page.

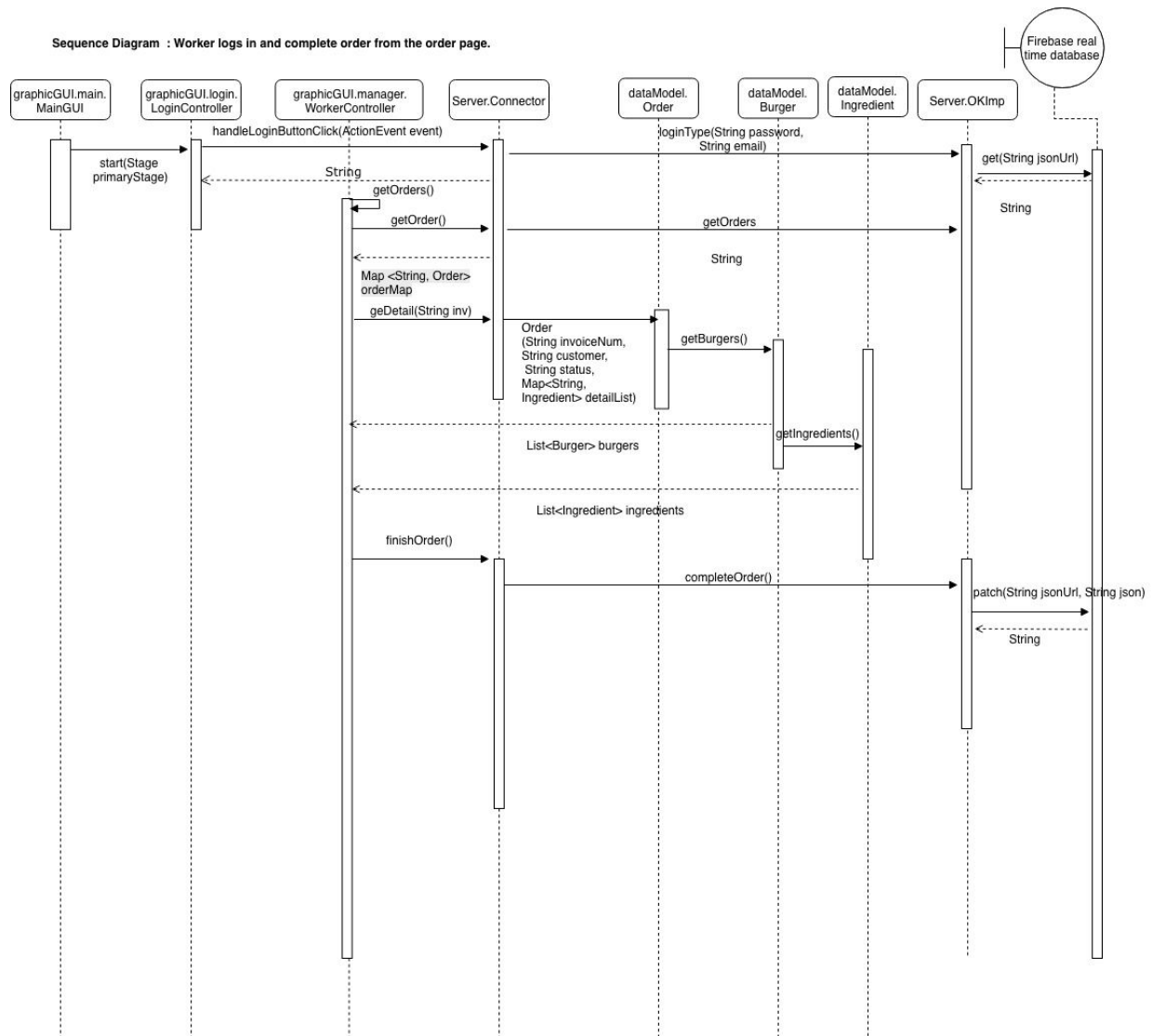
6.0 APPLICATION DIAGRAMS

6.1 Sequence Diagram of Production App

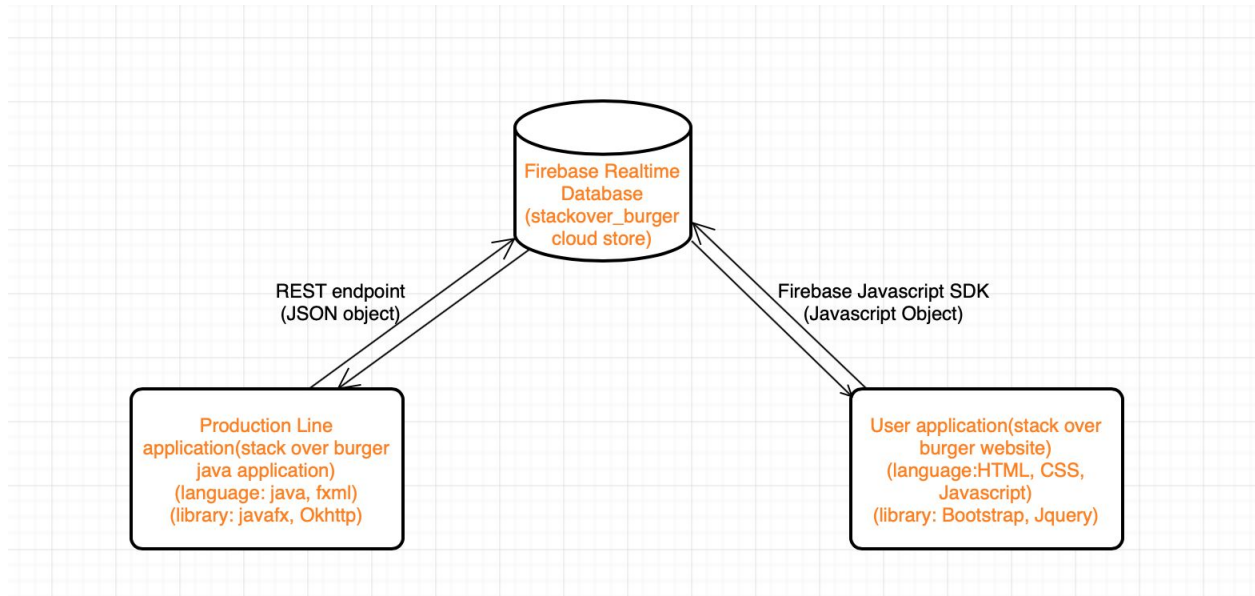
Sequence Diagram 1 : Manager logs in and adds quantity from the inventory page.



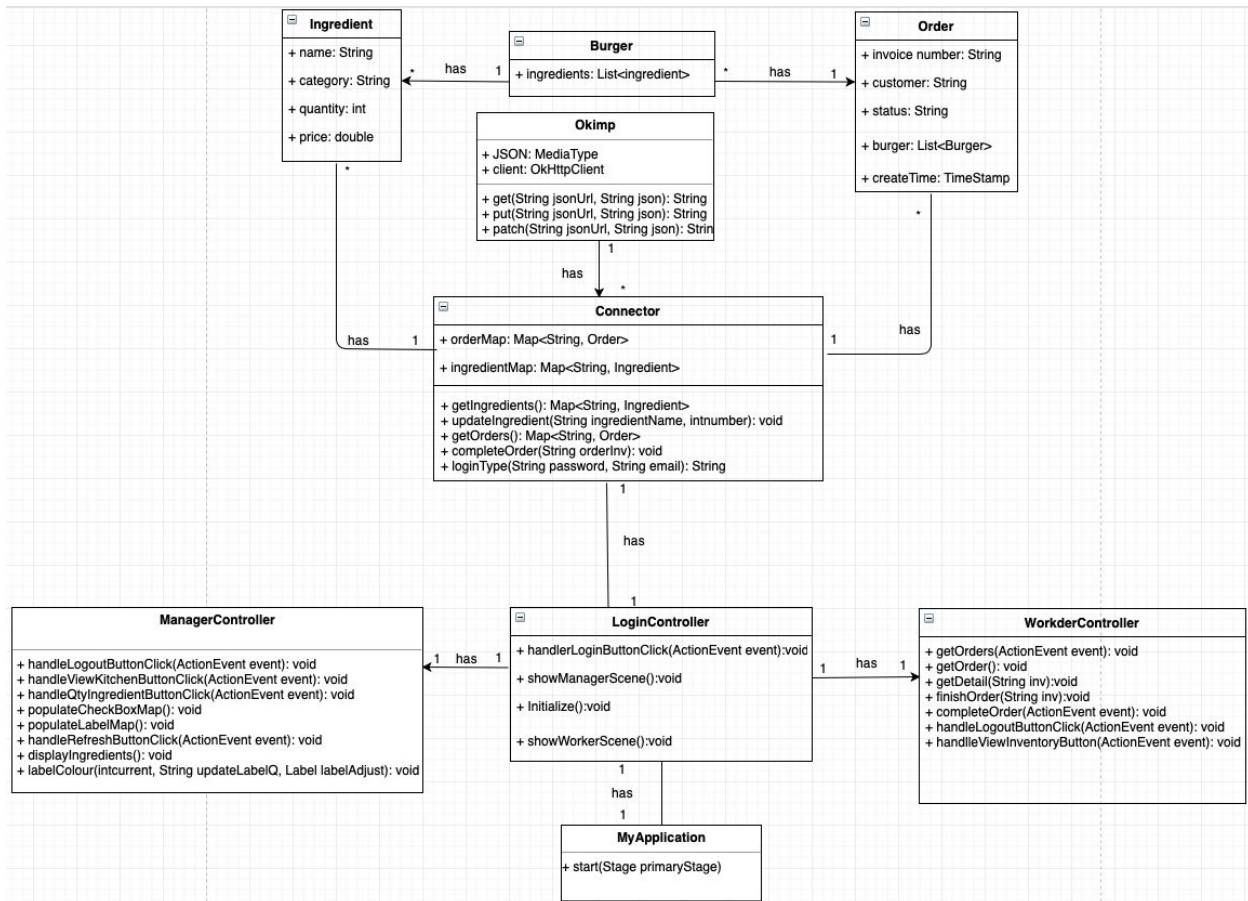
6.2 Sequence Diagram of Production App



6.3 High Level Architecture Diagram



6.4 Production Application Class Diagram



6.6 Ordering Application Flow Chart

