# ASSIGNMENT COVER SHEET
# TAUGHT POSTGRADUATE STUDENTS

# EXAMINATION NUMBER: Y3854628

MODULE NAME: Service Oriented Architectures (SOAR)

MODULE TUTOR: Prof. Dimitris Kolovos

ESSAY TITLE (if appropriate): Open Individual Assessment

WORD COUNT: 4193

**Academic Misconduct Declaration**

In writing my examination number at the top of this page, I declare:

- that the work that I am submitting for assessment contains no material copied in whole or in part from any other source unless it is explicitly identified by means of quotation marks;

- that I have acknowledged any quotations and content based on other sources by providing detailed references in an approved format;

- that I understand that unidentified and/or unreferenced copying constitutes plagiarism, which is one of a number of very serious offences in the University's Academic Misconduct Policies, Guidelines and Procedures (http://www.york.ac.uk/about/departments/support-and-admin/registry-services/academic-misconduct/) as outlined in the Language & Linguistic Science Postgraduate Handbook.

- That in preparing this assessment any assistance with proofreading and editing has been in accordance with the University's Guidance on Proofreading and Editing (for details see the Language & Linguistic Science Postgraduate Handbook). Any proofreading or editing assistance is acknowledged here:

In the interests of giving feedback, it is sometimes best to share writing samples among the students. Any written work that is shared will be make completely anonymous and all personal information, including the exam number, will be removed. By checking the following box, I indicate that I do not wish any part of my written work to be shared:

☑

## Table of Contents



# Notice

This solution uses Java 8 Lambdas, therefore Java 8 or above must be installed. Maven is also used for dependency management, and the project should be imported as a maven project.

While the solution works in Eclipse, I noticed that Eclipse does not always terminate a running service properly. In that case I had to resort to terminating the process from the task manager.

I've named this application *Time to Eat.*

# System Design

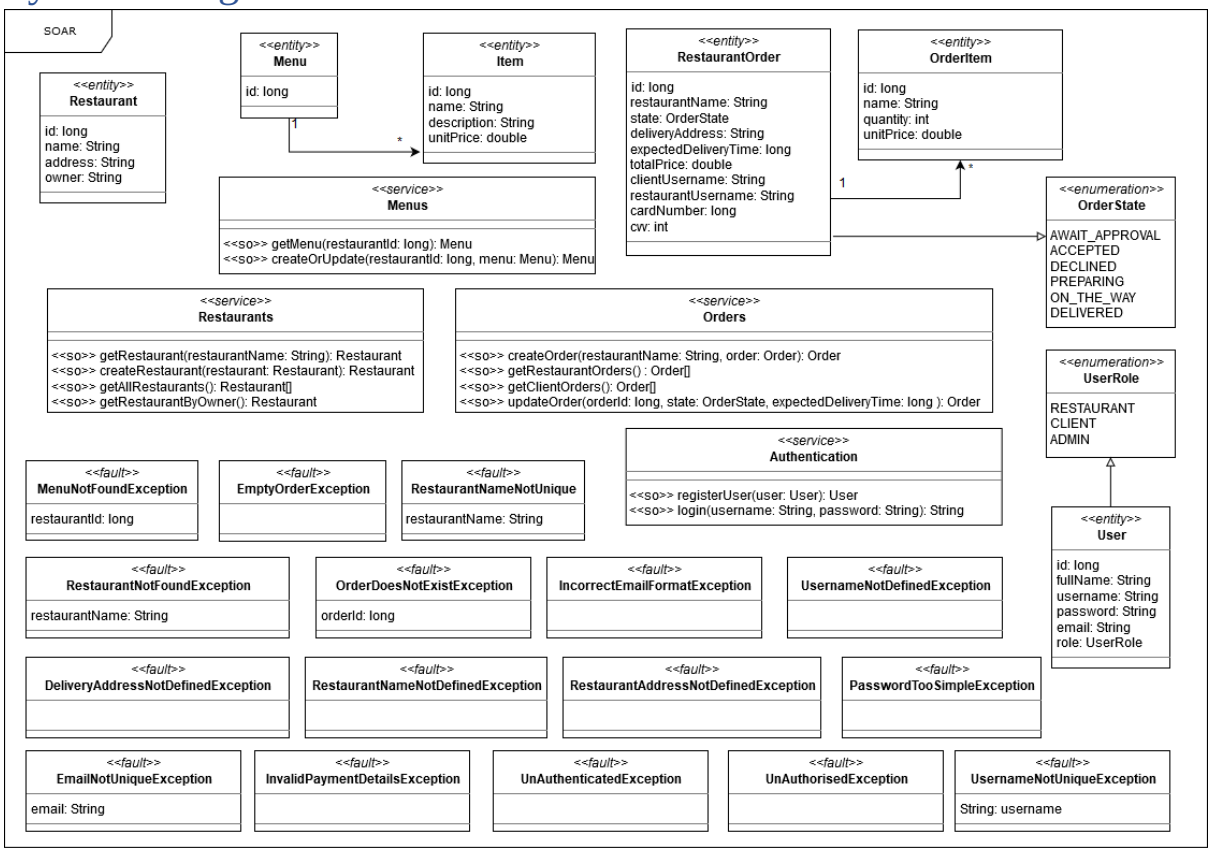


*Figure 1 - The UML diagram*

Figure 1 depicts the UML diagram for the *Time to Eat* food ordering and delivery management system.

## Entities

Each Service houses its own set of entities. They represent how models are persisted into the in-memory database. Notice that the restaurant entity is not directly linked to the menu entity in the UML diagram. This was intentionally drawn like this. It's true that the Menu entity consists of a *restaurantId*, hence in a way linking the two entities, but these entities are housed in two separate, standalone services. In the next sub section, I will talk a bit more about why I decided to separate services in this manner.

One thing hidden from this UML diagram is that there exist parameter classes for CRUD operations which are almost identical to their respective entities. All entity IDs are auto-generated, therefore are not included in parameter objects. In another case, *updateOrder()* in the Orders service in the implementation encapsulates the *state* and *expectedDeliveryTime* fields into a parameter object called *updateOrderParams*. In some cases, operation parameters are extracted from the method URL. To keep the UML diagram simple, I have omitted from defining every single parameter object and in some operations labelled the whole entity as the parameter.

## Services and Operations

There are four core services: Menus, Restaurants, Orders and Authentication. All these services are stateless, and do not contain any attributes. Each service contains basic CRUD operations. A design decision was made to make these services standalone services, each running on their own embedded Tomcat instance and database. They could even be housed in separate repositories or hosted on different continents and still work. I could have, say, merged the Restaurant and Menu services into one service, but in this way updating a menu for example doesn't require updating a restaurant entity and, more

specifically, doesn't add traffic to the restaurant service. One can say that a micro-service approach was taken to solve the given problem.

It is worth mentioning that some get methods don't have any parameters, such as *getRestaurantOrders()* and *getClientOrders()* in the Orders service. In these cases, the username is extracted from the current logged-in user and used to filter out orders by *clientUsername* or *restaurantUsername*. The same occurs for *getRestaurantByOwner()*, where the owner can only retrieve his or her own registered restaurant. More on this in the Authentication section.

All operations work on a specific HTTP method, which in this project is either GET, POST or PUT.

## Faults

Several faults have been defined. Rather than using SOAP, I decided to use REST, therefore the equivalent of Faults are Exceptions. Each Exception is being handled and mapped to a *ResponseEntity* with a specific status code (for example, *EmailNotUniqueException* maps to error 409, which is CONFLICT). Since these are Exceptions, all of them have a *message* attribute. Since I did not define the *message* attribute myself, I did not include it in the UML diagram. In some cases, such as *EmailNotUniqueException*, an additional attribute has been added, which in this case is the username that caused this fault.

## Other components that are not shown in the UML diagram

- **The portals**: There are two portals, one for restaurant owners, and the other for clients. These portals are what will be consuming these services. More about these in the Client Implementation section.

- **The Broker Service:** This service is another standalone service that contains the broker server. In this system, both message queues and topics are being used. More on this in the Push Notification section.

- **The Eureka Service:** Eureka is a tool developed by Netflix for service discovery. In the Implementation section I will give my motivations for choosing to use Eureka. It is important however that both the Eureka Service and the Broker Service are initialised first before the other services.

# Implementation

As mentioned in the previous section, a micro-service architecture has been adopted for this system. Maven has been used for dependency management, thus the Eclipse project should be imported as a maven project. Each service resides in its own maven module under the root module called *TimeToEat*.

## The Util Module

One of these modules is a utilities module that is used by all the other services and components. In this Util module, one can find:

- parameter classes
- some domain classes (that are used by all the other services)
- all the faults in the system
- some security constants and helper functions

## The Main Services

The core services are Orders, Menus, Restaurants and Authentication. They all follow the same structure and use the same technologies. I have decided to use Spring-Boot and several tools from Spring-Cloud-Netflix to implement my services. As previously mentioned, I have opted to use REST instead of SOAP.

A service contains the following packages:

### Dao

Short for *data access object*. This typically consists of the repository. I am using an in-memory repository called *hsqldb*, which is known for how fast and lightweight it is. One of the benefits of using Spring-Boot is how effortless it is to create a database and its DAO. If you were to look at *OrderRepository* for example, you would find that it is just an interface that extends a *CrudRepository*. The real magic happens later, where this interface is implemented at runtime. It automatically implements the basic CRUD operations for the given Entity. Mind you, this is not code generation! The implementation only appears at runtime and is not saved locally. Spring Boot is smart too and can implement methods that you add to it's interface. So, for example, in *OrderRepository* are two methods which I have defined. Since it knows that *RestaurantUsername* and *ClientUsername* are attributes of the *RestaurantOrder* entity, it knows that it needs to implement a filter by operation.

### Domain

Typically contains the entities that are persisted in a database. Hibernate annotations are being used here, which go hand in hand with the runtime implementation previously mentioned. One detail is that I've used the builder pattern a lot for my entities. There is no real motivation, other than personal preference.

### Handler

This package contains a handler which handles all the Exceptions (faults) that are being thrown. All Exceptions are placed in a *ResponseEntity* and mapped with an appropriate Error code. An *ExceptionResponse* is being used to standardise the way exception information are kept and send as part of the *ResponseEntity* body.

### Security

As the name implies, here lies all the security related classes. More on that in the Security section.

### Utils

This typically just consists of conversion methods that are used to convert between creation parameters and Entities.

## Application Class

This is the main class for the service. It launches the services as a Spring boot application. Notice that there are Eureka annotations. That is almost all that is required to register a service for Service Discovery. Starting a Spring boot application creates an embedded Tomcat in the application. There is no need to do any manual configurations, or place WAR/JARs in the tomcat folder. All you need to do is run ***mvn spring-boot:run*** in the service root folder.

## Controller

These are where all the service operations are defined and mapped to URLs. Technically for some service operations I did not need to define them in the controller because all the basic CRUD operations are implemented and mapped by the repository on runtime (so it is possible to do a POST with an Order Entity on /orders out of the box). However, I prefer to control the sort of parameters that will lead to the creation of entities and wrap some logic around them, such as validation. In some cases, you shouldn't pre-determine what an attribute should be. As an example, *RestaurantOrders* contain a total attribute. Rather than sending this attribute as part of the API call, this should be computed by the service, based on the prices of the order items, otherwise anyone could just send an incorrect total price. By default, all received calls are accepted as JSON. Another good reason for using Spring Boot is that JSON is mapped automatically to the objects that I annotate with @RequestBody (as long as they're serializable). Path parameters are also automatically extracted with the @PathVariable annotation. This was primarily the main reason why I stuck to JSON. I find this much quicker and easier to use than using XStream.

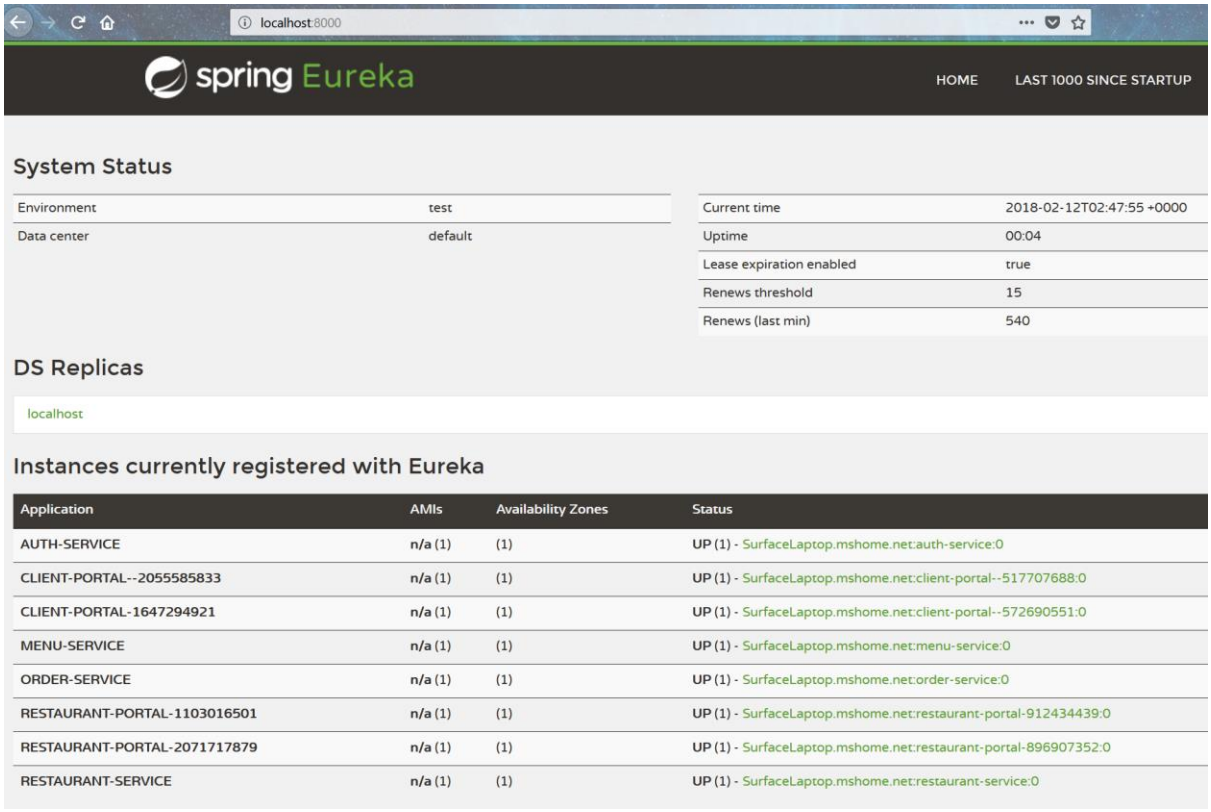


*Figure 2 - The Eureka portal with all the registered services*

## Resources

Here one would find an *application.yml* file. This is a configuration file that sets up properties such as port number, service registered name and eureka server URL. Since I'm using Eureka for service discovery, I decided to use random port numbers for services. The only services with fixed port numbers

are the Broker service and the Eureka Service. There was no real motivation behind randomising ports. I did it because I could.

## Netflix's Eureka, Feign and Ribbon

These three technologies are subtle at first, and require little configuration, but are very important to this solution. Eureka is a tool that provides service discovery. On start-up, each service registers itself with the Eureka server and make themselves discoverable to other services. The only service that does not register itself with Eureka is the Broker service, because it is not a Spring Boot application (I was running out of time). Once the Eureka service is started, you can navigate to http://localhost:8000 and see all the registered services. Clicking on the green link next to each service will open a new tab to the actual service address and port number.

Service discovery is cool, because I don't need to remember and hardcode port numbers for other services. However, on its own it is quite unnecessary for this assignment. The main motivation for using Eureka was that it integrates well with Netflix Feign. Before explaining what Netflix Feign is, I want to talk about client stubs.

In SOAP we can generate client stubs from WSDL. Here, I could have used Swagger to generate the equivalent stubs. Client stubs are convenient because they allow you to interact with the operations from other services, without having to define all the endpoints and looking up service operation URLs. However, you'd still have to remember port numbers, and most of the time generating stubs requires manual intervention (re-generate and move to desired code base etc. Also, if a service has a lot of operations, that means that you will have a lot of generated files to add to your repository, even you only require one of those operations.

Netflix Feign puts aside the idea of having client stubs. Instead it takes a declarative approach where one is only required to define an interface with the service operations that he or she needs for his client service. As an example, *RestaurantClient* in the *RestaurantPortal* is a Feign client. There are only two service operations defined here. These are almost identical to how they are defined in the *RestaurantController* class, except without the implementation. At the top of the interface is a *@FeignClient* annotation with the text "RESTAURANT-SERVICE". That is the service registered name of the Restaurants service in Eureka. That is all that is required to integrate with another service. Similar to what happens in the repository, the interface is implemented at runtime. At runtime it also searches the Eureka server for the service by its name and implements the interface using their IP address and port number. One can appreciate that such practices reduces our code base significantly.

Another Advantage of using Netflix Feign is that you can better control what attributes you want to receive from service operation calls. For example, the *getMenu()* service operation in the *MenuClient* interface in the *RestaurantService* does not use the same Menu Entity as that defined in the *MenuController*. In the restaurant service I did not need to work with the Menu ID, so I decided to omit it from the response. As mentioned before, Netflix Feign uses a declarative approach, and you only need to define what you really need.

While Netflix Feign provided so many benefits, there was one annoying thing about it. Initially I had mentioned how faults were being handled and mapped to a HTTP status code. The problem with Feign is that by default, a Feign Exception is thrown for any call that returns with an error code, even if it is wrapped in a *ResponseEntity*. According to forums, the reason for this behaviour is that an error is an error and must be treated and handled just like an exception. The work around for this is to add what is called a *FeignErrorDecoder* which will intercept errors and allow you to wrap them in an exception of your choice. Since most of the feign calls are done from the portals, I had to create a new kind of Exception called *ClientException* that contains the *ExceptionResponse* class.

The last piece of technology from Netflix is Ribbon. Ribbon is a piece of technology that provides client-side load balancing. I did not really need it, but by default, this is enabled when using Netflix Feign. I just didn't bother turning it off. In theory, if you're using Eureka, you can have several instances of the same service up and running (doesn't really make sense here if in-memory databases are being used). With Ribbon, Netflix Feign would decide which instance of the required service to call.

# Authentication

For authentication, I decided to use JSON Web Tokens (JWT). The main idea is that when one logs in for the first time, you are given a token, which you would need to add to the header of an API call that requires authentication. In this system I used Spring Security to secure my services.

Each core service has a similar security package structure, but only the Authentication service generates tokens. One thing to mention is that the service operation for login is not defined in the *UserController* but is an endpoint that is created and implemented by Spring Security at runtime. The only part of login that I had to implement was how it finds a user by username. This implementation can be found the *UserService* class. Technically, all the core services can perform login and generate a token. In fact, they all have a *UserService* class. However, I decided to keep the generation of tokens to the Authentication service only, since it contains the user repository.

Each core service has two security filters: *JWTAuthenticationFilter* and the *JWTAuthorizationFilter*. These do not differ between services and what they do is extract, verify and authenticate tokens from any incoming requests. Tokens are signed and verified with SHA-512 and a secret. This secret can be found in the Security constants in the Utils Module. I admit this isn't ideal in the real world. The best approach would have been encrypting the token using public and private key encryption.

The last class to cover is the *WebSecurity* class. This class consists of configurations, such as which URLs need to be authenticated. Notice that session creation is stateless. Each service has its own Web Security configuration for their own URLs. All create (except for register user) or update calls require authentication. Some lookup operations such as get Restaurant/s or menu seemed reasonable to be unsecured because both restaurants and clients would need to access this information. It made more sense securing the creation of entities, such as orders, restaurants and menus.

One feature that I did not have time to implement was the concept of user roles, where only restaurant users can call a certain service operation for example. I did however add some level of authorisation to my api calls. As previously mentioned, some of my service operations check the current logged in user. In the *RestaurantController*, the operation *getRestaurantByOwner* retrieves the restaurant that matches with the logged in user. It would have been cumbersome to first fetch a restaurant by id, and then add checks that it is called by the owner each time. The same happens for clients and restaurants for retrieving their orders.

While Authentication works well in this system, I can point out a few improvements. First, I would have used HTTPS, because as things are, any hacker can listen in to requests and extract tokens from headers. Another improvement that I would do is to add Gateways. I was considering using another tool from Netflix's arsenal called Zuul, however I felt that I would have been over-engineering my solution. If a gateway was present, authentication and authorization would happen there. On a production environment, all the available ports would be closed except for the gateway, which means that applications can only access services through this gateway (hence where all the authentication and verification is being done). With a gateway you can also proxy or hide internal service operations to external applications. Another improvement would be to have different gateways for restaurants and clients, such that a gateway only allows certain types users, and offers service operations tailored for that user.

# Client Implementation

As previously mentioned, there are two portals: one for restaurants, and the other for clients. These portals were implemented using Java Swing. Error handling has been included in most places, such as trying to create a user with an invalid email or password. Since these portals make use of Netflix Feign, they are booted up as SpringBoot applications.

Login and Register user are identical in both portals.

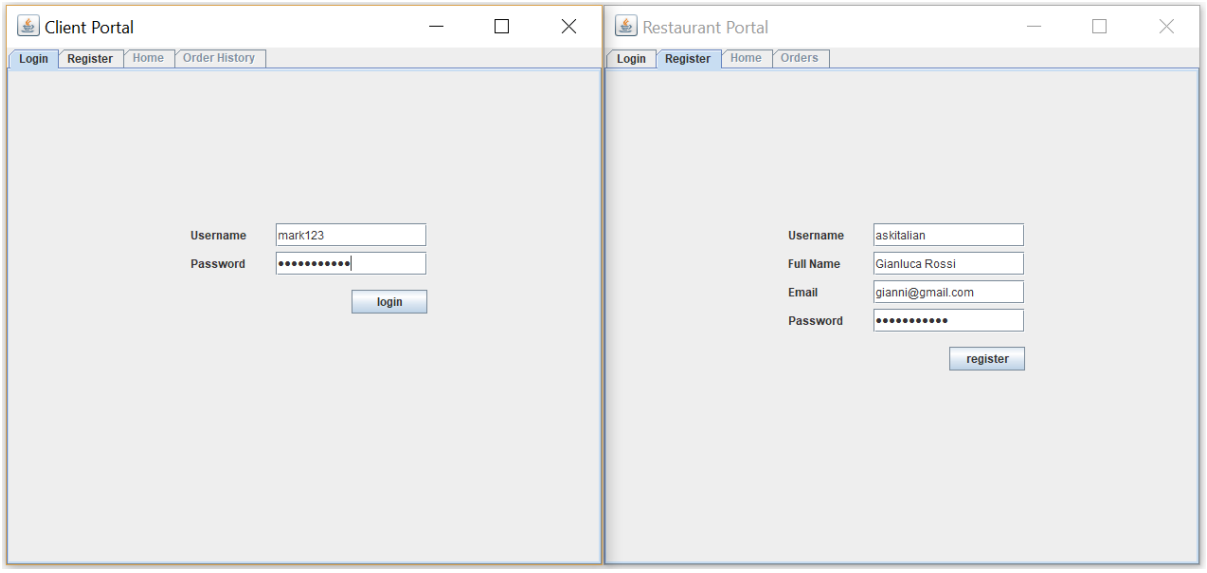


*Figure 3 - Left: Client Portal, Right: Restaurant Portal*

Once logged in the restaurant portal, the restaurant can create a restaurant and menu. After successfully creating a restaurant, the menu can be updated further. The menu table is editable and the create restaurant button is replaced with an update menu button. Selecting an item on the menu and clicking the remove selected menu item will remove the selected item from the table.

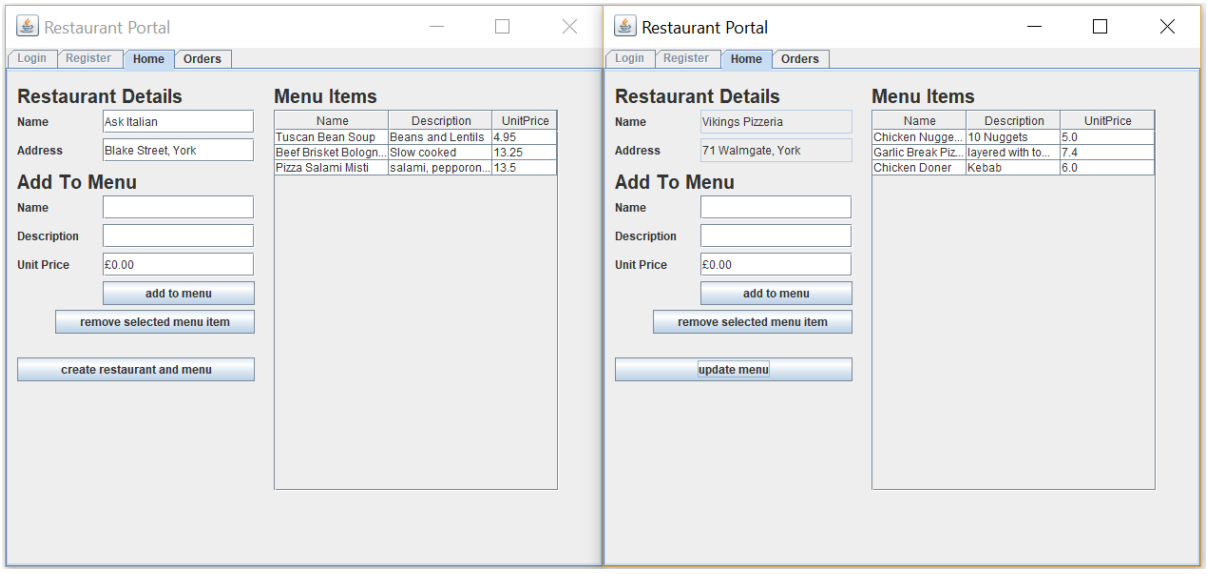


*Figure 4 - Creating a restaurant and menu*

Once restaurants and menus have been created, a client can log in and browse different restaurants and menus. The quantity column is editable but does not handle accepting integers only.

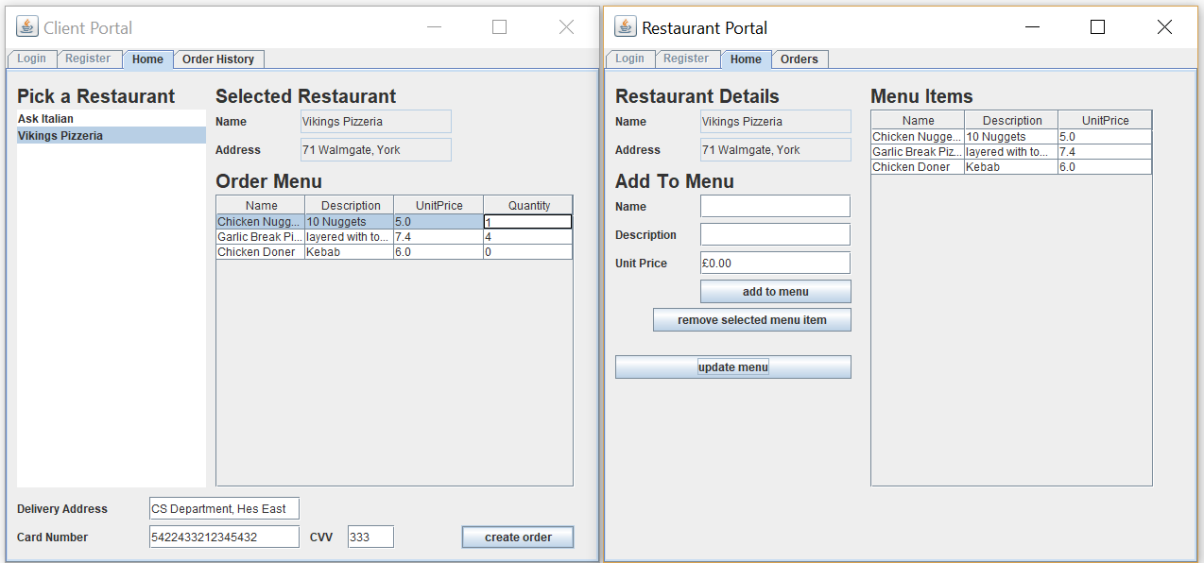


*Figure 5 - Creating an order*

Once an order is created, the restaurant receives a notification of a new order. He can either accept or decline the order. If he accepts, another dialogue box is show with the option of setting the expected delivery time.

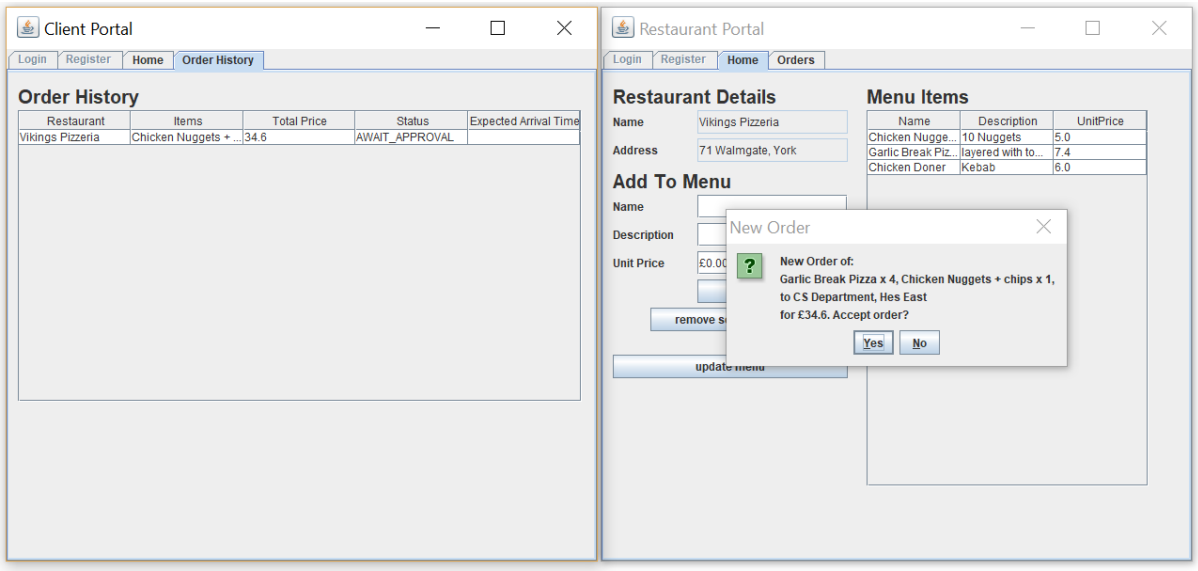


*Figure 6 - Accepting/Declining an order*

Once an order is accepted, the client receives a notification that the order status has changed. The restaurant can keep on updating the status to the next stage until it is delivered. Both the restaurant and client can see a history of their orders. After declining an order, you will notice that declined orders are not shown in the restaurant portal. This was intentional. Whenever a new order is created, the logged in username (a client) would be added to that order. When that order is accepted by the restaurant, the username of the restaurant owner (that is logged in his portal) is added to that order as well. If he declines, then the restaurant username is not added. Recall how the service operations getRestaurantOrders() and getClientOrders() extracted the current username to filter by. Since the restaurant username is never added to the order when declining, then they are not retrieved.

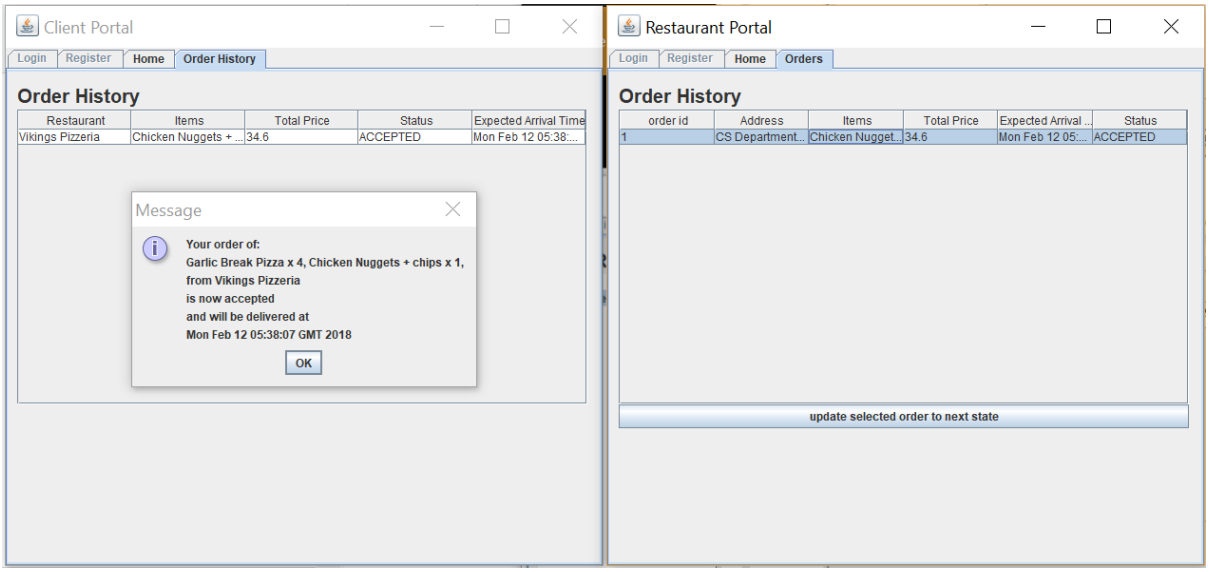


*Figure 7 - Receiving an order update*

## Push Notifications

As previously mentioned, message queues are being used for orders that need to be accepted or declined by a restaurant. Here I am using ActiveMQ. For orders, the Orders Service acts as the producer, while the Restaurant Portal acts as the consumer. As what can be observed in the *OrderController*, each time a new order is created, a new queue is created for the intended restaurant (if it is not already created yet). The name of this queue is a prefix concatenated with the restaurant name (which is unique). The same thing happens in the restaurant Portal. When a restaurant logs into the restaurant portal, a queue is created with the same prefix and restaurant name. Text messages are sent to the queues, with a preformatted message containing the order details.

A similar approach is used for creating topics. Topics are created and named for every client username, which are also unique. The order controller acts as a producer, while the client portal acts as the consumer. Once a client logs in the client portal, a topic is created for that user.

A different approach was initially considered, where both the client and restaurant portals acted as producers and consumers. This would have worked, with the possibility of the Orders service being a consumer too for updated orders. This approach was not adopted however because it felt better to first persist the new or updated order in the Orders service before sending the notification.

For new orders, it was assumed that the restaurant owner will only use one device to accept/decline orders. It also does not make sense to broadcast a new order to multiple consumers. What if two consumers respond differently to an order simultaneously? With message queues, restaurant owners are guaranteed to be notified of new orders, even if logged out. Upon login, restaurant owners will immediately be notified of pending orders. It did not make sense to use topics for creating new orders.

On the other hand, the main reason behind using topics for updating orders is that a client is likely going to have more than one device (phone, tablet, laptop…), and might be logged in all these devices. In this case, all connected devices will receive the notification.

Figure 6 and 7 show the push notification system in action. To fully test the push notification system, open several restaurant and client portals as possible and start creating orders. Also log into several portals with the same credentials. In the case of newly created orders, only one restaurant portal will receive the notification. In the case of updated orders, all client portals of the same logged in user will receive the notification.

## Running the System

I apologise from beforehand, but there are at least nine things that need to be executed, and they need to be executed in a certain order. However, I have added a folder with some eclipse launch configurations and they are numbered in the order that they should be run. I would also suggest freeing up some memory and increasing the Eclipse heap size in the vm-options for better performance.

This is the order of the build configurations:

1. Compile Everything (mvn clean install)
2. Run Eureka (mvn spring-boot:run)
3. Run Broker (run as normal Java class)
4. Run Auth Service (mvn spring-boot:run)
5. Run Menu Service (mvn spring-boot:run)
6. Run Order Service (mvn spring-boot:run)
7. Run Restaurant Service (mvn spring-boot:run)
8. Run Restaurant Portal (mvn spring-boot:run)
9. Run Client Portal (mvn spring-boot:run)

The order doesn't matter from 4-7. 8 and 9 and be run multiple times.