

# Food Delivery Project using Spring

The objective of this project is to develop a food delivery application (similar to Zomato, Swiggy). The application will be organized as a set of three microservices, each hosting a RESTful service. The high-level features of the application are as follows:

- There is a fixed set of [customers](#), [restaurants](#), [item IDs](#), and [delivery agents](#), specified initially in a given input file when the application starts. This input file is called `/initialData.txt`, and is explained more in a section titled Initialization later in this document.
- The three services are [Restaurant](#), [Delivery](#), and [Wallet](#).
- Delivery is the main service, with which customers interact and which invokes the other services in turn. The Delivery service keeps track of orders placed so far, the current statuses of the orders, and the current statuses of the delivery agents. The Delivery service is also aware of the price of each item in each restaurant (this is constant and specified upfront). This way, the Delivery service is able to calculate the total amount of an order when the order is received. An order can be for a single item only (but any quantity of it).
- The Restaurant service keeps track of the inventory of items available in all the restaurants. The inventory reduces when an order is received, and can be increased using a specified end-point.
- The Wallet service keeps track of the balance maintained by each customer, and supports end-points to decrease or increase wallet amounts.
- Any delivery agent can sign-in whenever they like. Whenever they are in signed-in state, they are either *available* (ready to deliver an order) or *unavailable* (i.e., currently delivering an order). They can sign-out whenever they like provided they are in *available* state. The Delivery service can assign an order to an agent only if the agent is in *available* state.
- Restaurants are assumed to be always open and serving.
- For any end-point below, if it is not mentioned who is to invoke it, then it is intended to be invoked by a human (delivery agent, customer, restaurant manager, etc). In our setting, a test-script will send requests to these end-points on behalf of humans.

## End-Points in [Restaurant](#) Service

1. POST `/acceptOrder`  
with JSON payload of the form `{"restId": num, "itemId": x, "qty": y}`

This end-point will be invoked by the Delivery service.

If the restaurant with ID *num* has at least *y* quantities of itemId *x* in its current inventory

then

reduce the inventory of item *x* in restaurant *num* by *y*

```
        return HTTP status code 201 (created)
    else
        return HTTP status code 410 (gone)
```

2. POST /refillItem  
with JSON payload of the form {"restId": *num*, "itemId": *x*, "qty": *y*}

Increases the inventory of itemId *x* in restaurant *num* by *y*. Should return status code 201 unconditionally.

3. POST /reInitialize

Set inventory of all items in all restaurants as given in the /initialData.txt file.  
Return HTTP status code 201.

## End-points in **Delivery** Service

Keeps track of the status of each order placed so far. The status of any order can be either *unassigned*, *assigned*, or *delivered* (*unassigned* means not yet given to a delivery agent). For any *assigned* or *delivered* order, the service also records the agentId assigned to this order. Also keeps track of the current status of each delivery agent (*signed-out*, *unavailable*, *available*). Maintains information about the price of each itemId in each restaurant (this is constant information).

1. POST /requestOrder  
with JSON payload of the form {"custId": *num*, "restId": *x*, "itemId": *y*, "qty": *z*}

This is the main end-point used by the customer to place an order. It first computes the total billing amount for this order (as it maintains the price of each itemId in each restaurant). It tries to deduct the amount from the custId *num*'s wallet. If deduction does not succeed, return HTTP status code 410. If the deduction succeeds, invoke the Restaurant Service's acceptOrder, passing to it *x*, *y*, *z*. If that returns 410, restore the wallet balance and return HTTP status code 410. Otherwise, first generate a fresh orderId *w* (let orderIds start from 1000 and let them be incremented by one each time). Initializes the status of this fresh order *w* as *unassigned*. Then see if any Delivery Agent is *available* right now. If yes, assign an agentId to this order *w* from among the available agentIds, update the status of the orderId *w* to *assigned*, mark the assigned agentId as *unavailable*, and record that *w* is assigned to this agentId. (If many agents are available, choose the lowest numbered agentId among them.) Then, whether or not the orderId was assigned an agent, return HTTP status code 201 and return response body {"orderId": *w*}.

2. POST /agentSignIn

with JSON payload of the form {"agentId": *num*}, where *num* is an agentId

If *num* is already signed-in, do nothing. Otherwise, If any orderIds are currently in *unassigned* state, find the least numbered orderId *y* that is unassigned, mark *num* as *unavailable*, mark the status of *y* as *assigned*, and record that *num* is assigned to *y*. Otherwise, mark the status of *num* as *available*. In all cases return HTTP status code 201.

3. POST /agentSignOut

with JSON payload of the form {"agentId": *num*}, where *num* is an agentId

If *num* is already *signed-out* or is *unavailable*, do nothing, else mark *num* as being in *signed-out* state. In both cases, return HTTP status code 201.

4. POST /orderDelivered

with JSON payload of the form {"orderId": *num*}, where *num* is an orderId

Ignore the request if *num* is not an orderId in *assigned* state. Otherwise, mark the status of orderId *num* as *delivered*, and mark the status of the agentId assigned to the order as *available*. Let *x* be this agentId. If any orderIds are currently in *unassigned* state, find the least numbered orderId *y* that is unassigned, mark *x* as *unavailable* again, mark *y* as *assigned*, and record that *y* is assigned to *x*. In all cases return 201.

5. GET /order/*num*

where *num* is an orderId.

If *num* is a non-existent orderId return HTTP status code 404. Otherwise return status code 200 along with response JSON of the form {"orderId": *num*, "status": *x*, "agentId": *y*}, where *x* is *unassigned*, or *assigned*, or *delivered*. *y* will be the agentId that is assigned the order *num* in case *num* is in *assigned* or *delivered* state, else *y* will be -1.

6. GET /agent/*num*

where *num* is an agentId

Return status code 200 and response JSON of the form {"agentId": *num*, "status": *y*}, where *y* is *signed-out*, *available*, or *unavailable*.

7. POST /reInitialize

Delete all orders from the records (no matter what their status is), and mark status of each agent as *signed-out*. Return HTTP status code 201.

## End-points in [Wallet](#) service

1. POST /addBalance  
with JSON payload of the form {"custId": *num*, "amount": *z*}

Increase the balance of custId *num* by *z*, and return HTTP status code 201.

2. POST /deductBalance  
with JSON payload of the form {"custId": *num*, "amount": *z*}

To be invoked by Delivery Service.

If current balance of custId *num* is less than *z*, return HTTP status code 410, else reduce custId *num*'s balance by *z* and return HTTP status code 201.

3. POST /reInitialize

Set balance of all customers to the initial value as given in the /initialData.txt file.  
Return HTTP status code 201.

## Requirements from implementation

You should develop each of the three services as a separate Spring project. Each project should be buildable using "maven package". Each project should contain a Dockerfile at the root folder. The Docker image from each project needs to be run (thus resulting in three microservice instances) in order to use the system.

In the end-points listed above, whenever we have not specified a certain form of validity checking on the given input, then you can assume that the input will be valid in that way and you need not check for it.

You need not use databases in Phase 1. Each microservice can represent its data using in-memory data structures. When the Restaurant microservice starts up it will initialize the inventory from the /initialData.txt file. The Delivery microservice will initialize all its data also from this file, will initialize the status of each agent mentioned in /initialData.txt to *signed-out* state, and will have an empty set of orders. The Wallet microservice will initialize the balances of all customers in the /initialData.txt file to the given initial balance in this file.

In Phase 1, you can assume that all requests will be sequential (i.e., no concurrent requests to any of the end-points). This will change in Phase 2.

We will be providing you a couple of "public" test cases soon. These will be somewhat informal in nature, and will be in plain English. You are to develop your own sufficient set of test cases. You will need to implement your test cases using shell scripts, python, Java, or any other suitable language. Each test case should finally print "Pass" or "Fail" exactly once. As a good practice, a single test case should not test multiple independent scenarios, as you should prefer to have separate test cases for separate scenarios. It would be a good practice to begin each test case with invocations to all three reInitialize end-points, so that the internal data of each microservice gets re-initialized. **You should aim to create a good set of test cases that test numerous interesting scenarios, corner cases, etc.**

## Initialization

An input file (the same input file) will be given to all three of your microservices. This will be mounted as file /initialData.txt using the "-v" option of "docker run" and available inside all your containers. A sample contents for this file is as follows:

```
101 2 //Restaurant_Id  number_of_item_available
1 180 10 //itemId price initial_quantity
2 230 20
102 3 //Restaurant_Id  number_of_item_available
1 50 10 //itemId price initial_quantity
3 60 20
4 45 15
****
201 // DeliveryAgent_Id
202
203
****
301 //Customer_Id
302
303
****
2000 //Initial wallet balance of all customers
```

[The comments will be not present in the data file.]

## Logistics

Each project is to be done by a team of students. Discussion across teams is not allowed. If you need any clarifications, please post in the “Project 1 (Spring)” channel on Teams. You can look at generic Java or Spring code fragments on the internet, but you should not look at any code projects online that resemble this given project.

You will implement this entire 1st project in two phases. **The deadline to complete the first phase is Feb. 8th 11.59 pm.** There will be no extensions granted under any circumstance. We will share with you instructions soon on how to upload your projects for evaluation. We will first evaluate your system by running it without your involvement. The correct performance of your system at this time on the public test cases, our private test cases, and your test cases, with exact expected output in each test case, will account for 75% of the marks of the 1st phase. The remaining 25% marks will be decided on the basis of a brief viva per team, where you will explain your code and the test cases you developed. The elegance, completeness, and comments in your code and the completeness of your test cases will be the factors here.

The first phase will have its own marks, which are assigned on the basis of your first phase submission. If the first phase demo shows any incompleteness/incorrectness, then you will not get full marks for this phase; however, you should still fix the first phase's issues after the first phase is over before you begin the work of the second phase as otherwise the 2nd phase performance will also suffer. (Phase 2 code will be based on Phase 1 code.)