

ORDER MY FOOD APPLICATION

I have created majorly three microservices named

- Restaurant Search Service
- Order Management Service
- Inventory Service

Other supporting services are **Service discovery (Discovery Server Eureka)**, **API-Gateway**, **Notification- Service**.

Firing all microservices and verify whether all services are discovered by Discovery Server.

The screenshot shows two instances of the Spring Eureka web interface. Both instances are running on port 8080 at localhost. The top instance displays the 'System Status' section, which includes environment details (Environment: test, Data center: default), system metrics (Current time: 2022-11-08T13:49:32 +0530, Uptime: 00:01, Lease expiration enabled: false, Renews threshold: 10, Renews (last min): 0), and a 'DS Replicas' section showing three registered instances: API-GATEWAY, INVENTORY-SERVICE, and NOTIFICATION-SERVICE, each with one AMI and one availability zone. The bottom instance also shows the 'System Status' section with identical values and a larger 'Instances currently registered with Eureka' table that includes five additional services: ORDER-MANAGEMENT-SERVICE, RESTAURANT-SEARCH-SERVICE, and three others from the top instance. It also includes a 'General Info' table with a single entry for total available memory.

| Application | AMIs | Availability Zones | Status |
|----------------------|---------|--------------------|---|
| API-GATEWAY | n/a (1) | (1) | UP (1) - MT-2LJ15L3:api-gateway:8080 |
| INVENTORY-SERVICE | n/a (1) | (1) | UP (1) - MT-2LJ15L3:inventory-service:8082 |
| NOTIFICATION-SERVICE | n/a (1) | (1) | UP (1) - MT-2LJ15L3:notification-service:8084 |

| Name | Value |
|--------------------|-------|
| total-avail-memory | 82mb |

We can see that all services are upon running and can be seen in Eureka discovery server. In the next picture we can see all services are running in eclipse. Api-gateway configuration can be seen to avoid ports because in spring cloud dynamic ports exists such that port will be changing regularly so we need to adopt to it. It also helps us to interact with other microservices without the usage of port.

```

eureka.client.serviceUrl.defaultZone = http://localhost:8761/eureka
spring.application.name = api-gateway
server.port = 8080
#logging
logging.level.root = INFO
logging.level.org.springframework.cloud.gateway.route.RouteDefinitionLocator = INFO
logging.level.org.springframework.cloud.gateway = TRACE
#
#
#Order Service
spring.cloud.gateway.routes[0].id = order-service
spring.cloud.gateway.routes[0].uri = lb://order-management-service
spring.cloud.gateway.routes[0].predicates[0]=Path=/api/v1/order/**
#
#Restaurant Search Service
spring.cloud.gateway.routes[1].id = restaurant-search-service
spring.cloud.gateway.routes[1].uri = lb://restaurant-search-service
spring.cloud.gateway.routes[1].predicates[0]=Path=/api/v1/restaurant/**
#
#Order Management Service
spring.cloud.gateway.routes[2].id = order-management-service
spring.cloud.gateway.routes[2].uri = lb://order-management-service
spring.cloud.gateway.routes[2].predicates[0]=Path=/api/v1/inventory/**
#
#Inventory service
spring.cloud.gateway.routes[3].id = inventory-service
spring.cloud.gateway.routes[3].uri = lb://inventory-service
spring.cloud.gateway.routes[3].predicates[0]=Path=/api/v1/inventory/**
#
#Notification Service
spring.cloud.gateway.routes[4].id = notification-service
spring.cloud.gateway.routes[4].uri = http://localhost:8761
spring.cloud.gateway.routes[4].predicates[0]=Path=/api/v1/notification/**
#
#API Gateway
spring.cloud.gateway.routes[5].id = api-gateway
spring.cloud.gateway.routes[5].uri = http://localhost:8080
spring.cloud.gateway.routes[5].predicates[0]=Path=/api/v1/*

```

- RESTAURANT – SEARCH – SERVICES walkthrough:

Adding the restaurant details to the database using postman client. (Here we observe port as 8080 which is api-gateway port which can be used to all microservices)

The screenshot shows the Postman interface with the following details:

- Request URL:** `http://localhost:8080/api/v1/inventory`
- Method:** POST
- Body (JSON):**

```

1 {
2     "name": "Bhargav Paradise",
3     "location": "Benz circle, Vijayawada",
4     "distance": 20,
5     "items": [
6         {
7             "itemName": "Palak Paneer",
8             "price": 100,
9             "cuisine": "INDIAN"
10        },
11        {
12            "itemName": "Mushroom Noodle",
13            "price": 160,
14            "cuisine": "THAI"
15        }
      ]
    }
  
```
- Response Status:** 201 Created
- Response Body:** Restaurant Details Saved Successfully

The screenshot shows the H2 Console interface with the following details:

- SQL Statement:** `SELECT * FROM RESTAURANT_DETAILS`
- Result Set:**

| ID | DATE_OF_ESTABLISHED | DISTANCE_FROM_CITY_CENTER | LOCATION | RESTAURANT_NAME |
|----|---------------------|---------------------------|-------------------------|------------------|
| 1 | 2022-11-08 | 20 | Benz circle, Vijayawada | Bhargav Paradise |

Saved details can be seen in H2 console which is an in-memory database. The actual MySQL database is also configured for each microservices but using h2 for easier implementation.

The screenshot shows the H2 Console interface with the following details:

- Top Bar:** Shows tabs for "Subscription Details | Nuvepro", "a4d20bdd-a788-4ac3-a707-785c", "Eureka", and "H2 Console".
- Toolbar:** Includes buttons for "Run Selected", "Auto complete", and "Clear".
- Left Sidebar:** Displays the schema structure with tables: ITEM, RESTAURANT_DETAILS, INFORMATION_SCHEMA, Sequences, and Users. It also shows the version: H2 2.1.214 (2022-06-13).
- SQL Statement:** The query "SELECT * FROM ITEM;" is entered in the SQL statement input field.
- Result Table:** The result of the query is displayed in a table with the following data:

| ITEM_ID | TYPE_OF_CUSINE | ITEM_CODE | ITEM_NAME | COST_OF_ITEM | RESTAURANT_FK |
|---------|----------------|----------------------------------|-----------------|--------------|---------------|
| 2 | INDIAN | bc4534e017884208bbd90255ec53cc96 | Palak Paneer | 100 | 1 |
| 3 | THAI | 586e5222b5b8422cad4485fd8d9e2c2d | Mushroom Noodle | 160 | 1 |

- Bottom:** A note "(2 rows, 1 ms)" and an "Edit" button.

The screenshot shows the Postman application interface with the following details:

- Left Sidebar:** Collections, APIs, Environments, Mock Servers, Monitors, History.
- Request URL:** http://localhost:8080/api/v1/restaurant
- Method:** POST
- Body:** JSON (Pretty)
- Request Body:**

```

1   {
2     "name": "Hungry Birds",
3     "location": "Necklace Road, Hyderabad",
4     "distance": 40,
5     "items": [
6       {
7         "itemName": "Chicken Dum Biryani",
8         "price": 250,
9         "cuisine": "INDIAN"
10      }
11    ]
12  }

```
- Response Headers:** 201 Created, 29 ms, 158 B
- Response Body:** "Restaurant Details Saved Successfully"

Using a GET method to retrieve all restaurants from the database.

The screenshot shows the Postman application interface with the following details:

- Left Sidebar:** Collections, APIs, Environments, Mock Servers, Monitors, History.
- Request URL:** http://localhost:8080/api/v1/restaurant
- Method:** GET
- Params:** Query Params
- Body:** JSON (Pretty)
- Request Body:**

```

1   [
2     {
3       "id": 1,
4       "name": "Bhargav Paradise",
5       "location": "Benz circle, Vijayawada",
6       "distance": 20,
7       "items": [
8         {
9           "id": 2,
10          "itemCode": "bc4534e0-1788-4208-bbd9-0255ec53cc96",
11          "itemName": "Palak Paneer",
12          "price": 100,
13          "cuisine": "INDIAN"
14        },
15        {
16          "id": 3,
17          "itemCode": "586e5222-b5b8-422c-ad44-85fd8d9e2c2d",
18          "itemName": "Mushroom Noodle"
19        }
20      ]
21    }
22  ]

```
- Response Headers:** 200 OK, 353 ms, 444 B

Retrieving restaurants with given location (Here the provided location is case-insensitive such that we can provide location in any case). It returns list of restaurants.

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/api/v1/restaurant/location/VIJAYAWADA`. The response body is a JSON array of restaurant objects:

```

1  [
2    {
3      "id": 1,
4      "name": "Bhargav Paradise",
5      "location": "Benz circle, Vijayawada",
6      "distance": 20,
7      "items": [
8        {
9          "id": 2,
10         "itemCode": "bc4534e0-1788-4288-bbd9-0255ec53cc96",
11         "itemName": "Palak Paneer",
12         "price": 100,
13         "cusine": "INDIAN"
14       },
15       {
16         "id": 3,
17         "itemCode": "586e5222-b5bb-422c-ad44-85fd8d9e2c2d",
18         "itemName": "Mutton Masala"
19       }
20     ]
21   }
22 ]

```

Retrieving restaurant based on distance (Here we are proving the distance from the city you are located and the distance under you want the restaurants). It returns list of restaurants.

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/api/v1/restaurant/distance/38/4`. The response body is a JSON array of restaurant objects:

```

1  [
2    {
3      "id": 4,
4      "name": "Hunger Birds",
5      "location": "Necklace Road, Hyderabad",
6      "distance": 40,
7      "items": [
8        {
9          "id": 5,
10         "itemCode": "11bd193c-c4f3-49d8-a644-f9cc06efc649",
11         "itemName": "Chicken Dum Biryani",
12         "price": 250,
13         "cusine": "INDIAN"
14       }
15     ]
16   }
17 ]

```

The picture below shows retrieving the restaurant details based on Cusine they provided. Here we provided THAI as cusine then it retrieves all the restaurant that provided THAI food as cusine.

The screenshot shows the Postman application interface. In the top navigation bar, there are links for Home, Workspaces, Explore, and a search bar labeled "Search Postman". On the right side, there are buttons for Sign In, Create Account, and environment dropdowns. The main area is titled "Scratch Pad" and shows a collection named "Collections". A central panel displays a GET request to "http://localhost:8080/api/v1/restaurant/cuisine/THAI". The "Body" tab is selected, showing a JSON response with three items:

```
    "id": 1,
    "name": "Bhangav Paradise",
    "location": "Benz circle, Vijayawada",
    "distance": 20,
    "items": [
        {
            "id": 2,
            "itemCode": "bc4534e0-1788-4288-bbd9-0255ec53cc96",
            "itemName": "Palak Paneer",
            "price": 100,
            "cuisine": "INDIAN"
        },
        {
            "id": 3,
            "itemCode": "586e5222-b5b8-422c-ad44-85fd8d9e2c2d",
            "itemName": "Mushroom Noodle",
            "price": 160,
        }
    ]
```

The status bar at the bottom indicates a 200 OK response with 30 ms and 444 B.

Retrieving the restaurants based on your budget. We pass the budget and it checks all the restaurant for minimum price retrieves all the restaurant that are in our provided budget.

This screenshot is identical to the one above, showing the same GET request to "http://localhost:8080/api/v1/restaurant/cuisine/THAI". The response body is the same, listing three restaurants: Bhangav Paradise, Palak Paneer, and Mushroom Noodle.

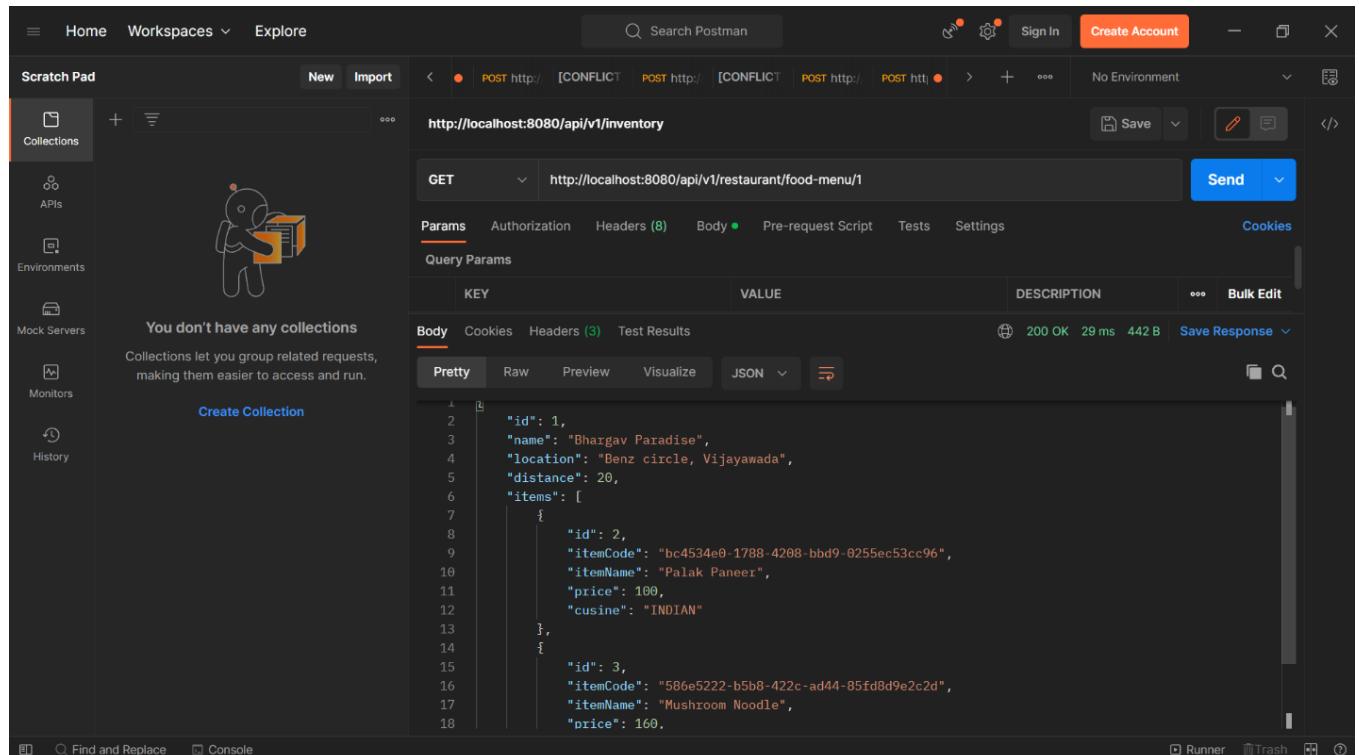
Retrieving the restaurant based on the name provided.

The screenshot shows the Postman application interface again. This time, the URL in the request bar is "http://localhost:8080/api/v1/restaurant/restaurant-name/Hunger Birds". The "Body" tab is selected, showing a JSON response with three items:

```
    "id": 4,
    "name": "Hunger Birds",
    "location": "Necklace Road, Hyderabad",
    "distance": 40,
    "items": [
        {
            "id": 5,
            "itemCode": "11bd193c-c4f3-49d8-a644-19cc06efc649",
            "itemName": "Chicken Dum Biryani",
            "price": 250,
            "cuisine": "INDIAN"
        }
    ]
```

The status bar at the bottom indicates a 200 OK response with 24 ms and 333 B.

Viewing the food menu based on the id provide for the restaurant

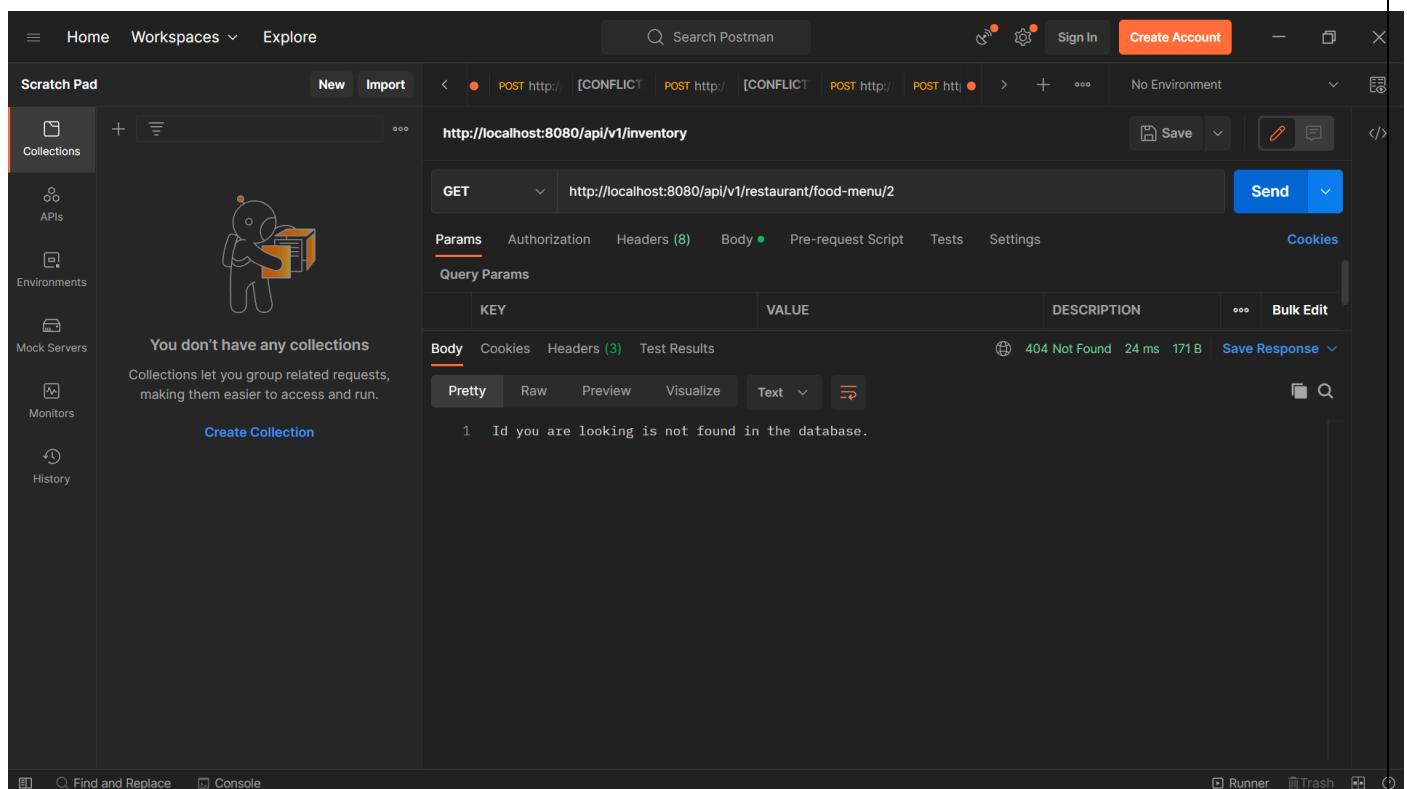


The screenshot shows the Postman application interface. On the left, there's a sidebar with options like Home, Workspaces, Explore, Scratch Pad, Collections, APIs, Environments, Mock Servers, Monitors, and History. The main area displays a request to `http://localhost:8080/api/v1/restaurant/food-menu/1`. The response body is a JSON object representing a restaurant's food menu:

```
2 "id": 1,
3   "name": "Bhargav Paradise",
4   "location": "Benz circle, Vijayawada",
5   "distance": 20,
6   "items": [
7     {
8       "id": 2,
9       "itemCode": "bc4534e0-1788-4208-bbd9-0255ec53cc96",
10      "itemName": "Palak Paneer",
11      "price": 100,
12      "cuisine": "INDIAN"
13    },
14    {
15      "id": 3,
16      "itemCode": "586e5222-b5b8-422c-ad44-85fd8d9e2c2d",
17      "itemName": "Mushroom Noodle",
18      "price": 160.

```

If we provided the id which is not in the database it throws 404 and return customized message.



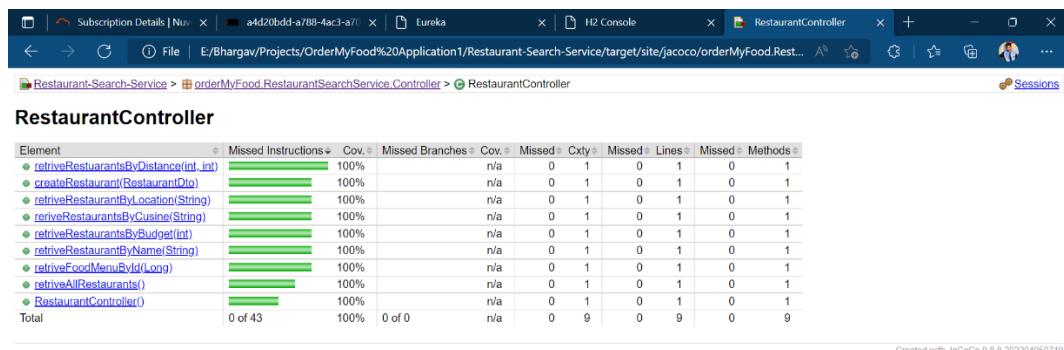
The screenshot shows the Postman application interface. The request URL is `http://localhost:8080/api/v1/restaurant/food-menu/2`. The response body is a simple text message:

```
1 Id you are looking is not found in the database.
```

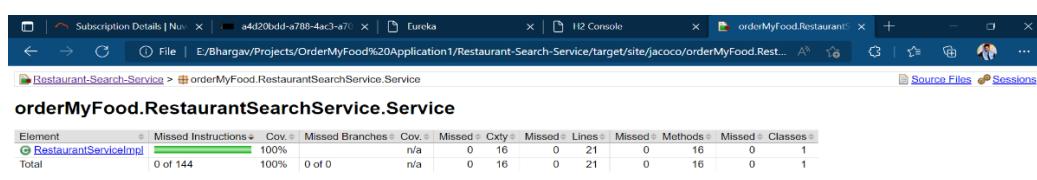
Running the maven-test command to generate the Jacoco code coverage report.

```
[INFO] Tests run: 9, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.144 s - in Project.Resta...
[2022-11-08 14:26:13.085 INFO [restaurant-search-service,] 9096 --- [ Thread-10] o.s.c.s.a.z.ZipkinAutoConfiguration
[2022-11-08 14:26:13.088 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean
[2022-11-08 14:26:13.090 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] o.s.c.n.e.DiscoveryServiceRegis...
[2022-11-08 14:26:13.090 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] com.netflix.discovery.DiscoveryClient
[2022-11-08 14:26:13.090 INFO [restaurant-search-service,] 9096 --- [infoReplicator-0] com.netflix.discovery.DistributedClient
[2022-11-08 14:26:13.095 INFO [restaurant-search-service,] 9096 --- [infoReplicator-0] com.netflix.discovery.DiscoveryClient
[2022-11-08 14:26:13.095 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] j.LocalContainerEntityManagerFactoryBean
[2022-11-08 14:26:14.100 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] com.zaxxer.hikari.HikariDataSource
[2022-11-08 14:26:14.100 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] com.netflix.discovery.DiscoveryClient
[2022-11-08 14:26:17.138 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] com.netflix.discovery.DiscoveryClient
[2022-11-08 14:26:17.148 INFO [restaurant-search-service,] 9096 --- [ionShutdownHook] com.netflix.discovery.DiscoveryClient
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 26, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] jacoco-maven-plugin:0.8.0:report (report) @ Restaurant-Search-Service ...
[INFO] Analyzing execution data file E:\Bhargav\Projects\OrderMyFood Application1\Restaurant-Search-Service\target\jacoco.exec
[INFO] Analyzing bundle 'Restaurant-Search-Service' with 13 classes
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time: 54.315 s
[INFO] Finished at: 2022-11-08T14:26:17+05:30
[INFO]
```

Jacoco code coverage report for Restaurant – service - Controller



Jacoco Service layer report



- INVENTORY – SERVICE – Walkthrough:
Adding inventory into the database

The image contains two screenshots of the Postman application interface, demonstrating the process of adding inventory items.

Screenshot 1: The left screenshot shows the Postman interface with a collection named "November 3". A POST request is being prepared to "http://localhost:8080/api/v1/inventory". The Body tab contains the following JSON payload:

```

1 ... "itemCode": "8b511aa9-bb87-4ab6-ac14-73740d0ea703",
2 ...
3 ...
4 ...

```

The response status is 201 Created, and the message is "Inventory Added Successfully".

Screenshot 2: The right screenshot shows the Postman interface with a collection named "November 3". A POST request is being prepared to "http://localhost:8080/api/v1/inventory". The Body tab contains the following JSON payload:

```

1 ...
2 ...
3 ...
4 ...

```

The response status is 201 Created, and the message is "Inventory Added Successfully".

Checking the inserted data in H2 console

The image shows the H2 Console interface with the following details:

- URL: <http://localhost:8082/h2-console/login.do?sessionid=faf028a08f7d0fc4b7eba53f420d79a9>
- Database: jdbc:h2:mem:dicapp
- Schema: INVENTORY
- Table: INVENTORY
- Version: H2 2.1.214 (2022-06-13)

The SQL statement entered is:

```
SELECT * FROM INVENTORY;
```

The resulting table is:

| INVENTORY_ID | ITEM_CODE | STOCK |
|--------------|--------------------------------------|-------|
| 1 | 8b511aa9-bb87-4ab6-ac14-73740d0ea703 | 10 |
| 2 | ce0f22eeeebe434da5bb7a1675ed1206 | 50 |

(2 rows, 2 ms)

Checking the Spans and traces in Zipkin for Inventory service. We observe a post request that is recorded.

The screenshot shows the Zipkin interface with a single trace for an INVENTORY-SERVICE POST request. The trace duration is 3.741ms. The trace tree shows a single span for the POST request. The annotations section is empty. The tags section includes:

- http.method: POST
- http.path: /h2-console/query.do
- Client Address: [:1]:56757

Retrieving the inventory using id

The screenshot shows a Postman request to `http://localhost:8080/api/v1/inventory{id}`. The response status is 200 OK, with a duration of 41 ms and a size of 185 B. The response body is:

```
1 [ {  
2   "id": 1,  
3   "itemCode": "8b511aa9-bb87-4ab6-ac14-73740d0ea703",  
4   "stock": 10  
5 } ]
```

Retrieving the inventory using item-code

The screenshot shows a Postman request to `http://localhost:8080/api/v1/inventory?itemCode=8b511aa9-bb87-4ab6-ac14-73740d0ea703`. The response status is 200 OK, with a duration of 242 ms and a size of 185 B. The response body is:

```
1 [ {  
2   "id": 1,  
3   "itemCode": "8b511aa9-bb87-4ab6-ac14-73740d0ea703",  
4   "stock": 10  
5 } ]
```

M1093477

Checking the item is in stock for the provided quantity if yes returns true else false.

The screenshot shows the Postman interface with a successful GET request to `http://localhost:8082/api/v1/inventory`. The response status is 200 OK, and the body contains the value `true`.

If item quantity required is more than the items have then it returns false.

The screenshot shows the Postman interface with a successful GET request to `http://localhost:8082/api/v1/inventory`. The response status is 200 OK, and the body contains the value `false`.

If item is in stock it updates the stock (here from 10 to 6)

The screenshot shows the Postman interface with a successful GET request to `http://localhost:8082/api/v1/inventory/item-code/8b51aa9-bb87-4ab6-ac14-73740d0ea703`. The response status is 200 OK, and the body shows the updated item details with `stock: 6`.

If you entered an invalid item code then it throws 404 and returns customized error message

The screenshot shows the Postman interface. In the top navigation bar, 'Explore' is selected. Below it, the 'Scratch Pad' section lists several recent API requests. A specific GET request is highlighted with the URL `http://localhost:8080/api/v1/inventory/item-code/ab51aa9-bb87-4ab6-ac14-73740d0ea703`. The response status is 404 Not Found, and the body of the response contains the message: "Item Code you are looking for is not found in the database."

While searching based on id, if we provide wrong id then it throws 404 and returns customized error message

This screenshot of the Postman interface shows a similar setup to the previous one. A GET request is made to `http://localhost:8080/api/v1/inventory/id/213`, resulting in a 404 Not Found error. The response body states: "Id you are looking for is not found in the database."

We can see the tracing of api in zipkin in the below picture.

The screenshot displays the Zipkin UI at `localhost:9411/zipkin/?serviceName=inventory-service&lookback=15m&endTs=1667899457691&limit=10`. It shows three traces for the 'api-gateway' service, each involving the 'inventory-service'. The first trace has a Trace ID of 529ebadaa82cdba7, the second has 42bb8be8b59862715, and the third has b56a8d3b41a4a3fd. Each trace is composed of two spans: one for the 'api-gateway' and one for the 'inventory-service'. The spans are color-coded in red and green boxes.

- ORDER - MANAGEMENT – SERVICE walkthrough:

Placing the order which saved the data to database. It calls inventory service and checks for the stocks using rest template.

```

POST http://localhost:8080/api/v1/order
{
  "customerName": "Bhargav Siddineni",
  "phone": "6303778542",
  "address": "6-375 Rtc colony Hyderabad",
  "orderItems": [
    {
      "itemCode": "ce0f2ee-ebe-434d-a5bb-7a1675ed1206",
      "price": 20,
      "quantity": 1
    }
  ]
}
  
```

Body Cookies Headers Test Results
Pretty Raw Preview Visualize Text
1 Order Saved successfully

We can observe the tracing of post request which involves two other microservices.

API-GATEWAY: post
Duration: 174.415ms Services: 3 Depth: 3 Total Spans: 3 Trace ID: 79f1c2f09829b383

Annotations

Tags

| |
|----------------|
| http.method |
| POST |
| http.path |
| /api/v1/order |
| Client Address |
| ::1:60499 |

We can see the order details in the database.

| ORDER_ID | LAST_UPDATED_ON | CUSTOMER_ADDRESS | CUSTOMER_NAME | DATE_OF_ORDER_PLACED | PHONE_NUMBER | STATUS | TOTAL_PRICE |
|----------|------------------------|----------------------------|-------------------|------------------------|--------------|--------|-------------|
| 1 | 2022-11-08 15:26:05.53 | 6-375 Rtc colony Hyderabad | Bhargav Siddineni | 2022-11-08 15:26:05.53 | 6303778542 | PLACED | 20 |

After order is placed using the rabbit Template from Rabbit MQ broker we send a notification along with order details to Notification service through PLACE_ORDER_QUEUE.

```

package orderMyFood.NotificationService.consumer;
import org.springframework.amqp.rabbit.annotation.RabbitListener;

@Component
public class Consumer {
    @RabbitListener(queues = "PLACE_ORDER_QUEUE")
    public void placeOrderStatusConsumer(OrderDto orderDto) {
        System.out.println("Order Placed! Order details provided below");
        System.out.println(orderDto);
    }
}

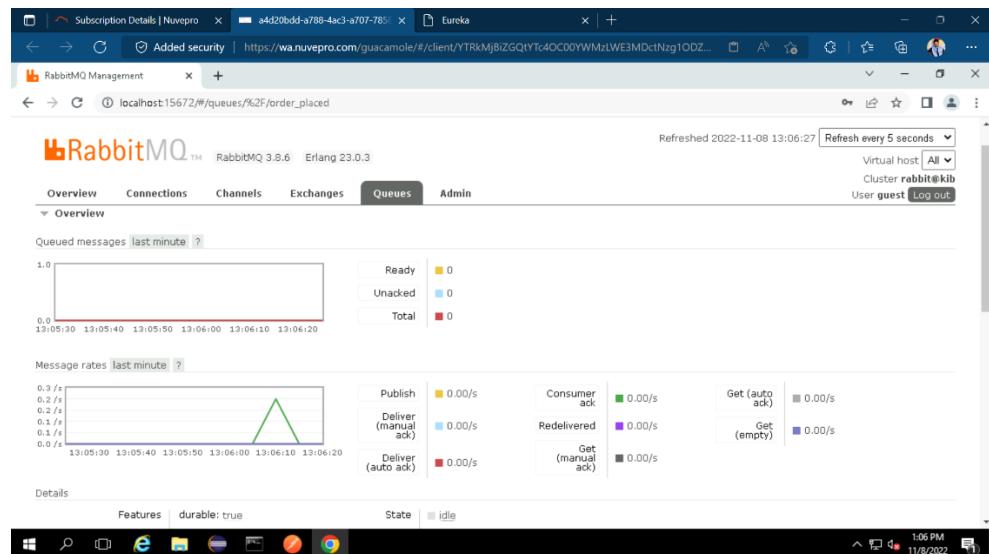
NotificationServiceApplication [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\java.exe (Nov 8, 2022 1:03:45 PM)
2022-11-08 13:03:49.751 INFO 6620 --- [ restartedMain] com.netflix.discovery.DiscoveryClient : Saw local status change event StatusChangeEvent [state:UP]
2022-11-08 13:03:49.752 INFO 6620 --- [InfoReplicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_NOTIFICATION-SERVICE/kib.pikodwyls : DiscoveryClient_NOTIFICATION-SERVICE/kib.pikodwyls
2022-11-08 13:03:49.775 INFO 6620 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8084 (http) with context root /
2022-11-08 13:03:49.775 INFO 6620 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8084 (http) with context root /
2022-11-08 13:03:49.793 INFO 6620 --- [ restartedMain] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2022-11-08 13:03:49.804 INFO 6620 --- [InfoReplicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_NOTIFICATION-SERVICE/kib.pikodwyls : DiscoveryClient_NOTIFICATION-SERVICE/kib.pikodwyls
2022-11-08 13:03:49.875 INFO 6620 --- [ restartedMain] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory@34
2022-11-08 13:03:50.064 INFO 6620 --- [ restartedMain] o.N.NotificationServiceApplication : Started NotificationServiceApplication in 3.923 seconds (JVM running for 4.002)
Order Placed! Order details provided below
OrderDto{id=2, orderItems=[orderItemsDto{id=2, itemCode=0b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=1}], totalPrice=20, status=PLACED, customerName=John Doe}

```

All queues that are created can be seen in Rabbit Mq UI at port 15672.

| Name | Type | Features | State | Ready | Unacked | Total | incoming | deliver / get | ack |
|---------------|---------|----------|-------|-------|---------|-------|----------|---------------|--------|
| cancel_order | classic | D | idle | 0 | 0 | 0 | | | |
| cancel_placed | classic | D | idle | 1 | 0 | 1 | | | |
| message_queue | classic | D | idle | 0 | 0 | 0 | | | |
| order_placed | classic | D | idle | 0 | 0 | 0 | 0.00/s | 0.00/s | 0.00/s |
| update_order | classic | D | idle | 0 | 0 | 0 | | | |

We can see graphical the message transferred from order service to notification service when an order is placed.



Updating the data that is present in the database.

POST http://localhost:8080/api/v1/order

PUT http://localhost:8080/api/v1/order

Body

```
1 "customerName": "Likitha Siddineni",
2 "phone": "6303778541",
3 "address": "6-375 Rtc colony Hyderabad",
4 "orderItems": [
5   {
6     "itemCode": "ce0f22ee-ebe0-434d-a5bb-7a1675ed1206",
7     "price": 20,
8     "quantity": 5
9   }
10 ]
11
12
13
```

Body Cookies Headers (3) Test Results

Pretty Raw Preview Visualize Text

200 OK 126 ms 142 B Save Response

Checking the updated data in the database

Auto commit: Off Auto complete: Off Auto select: On

Max rows: 1000

Run Run Selected Auto complete Clear SQL statement:

SELECT * FROM ORDERS_TABLE;

| ORDER_ID | LAST_UPDATED_ON | CUSTOMER_ADDRESS | CUSTOMER_NAME | DATE_OF_ORDER_PLACED | PHONE_NUMBER | STATUS | TOTAL_PRICE |
|----------|-------------------------|----------------------------|-------------------|-------------------------|--------------|--------|-------------|
| 1 | 2022-11-08 15:30:35.618 | 6-375 Rtc colony Hyderabad | Likitha Siddineni | 2022-11-08 15:30:12.448 | 6303778541 | PLACED | 100 |

(1 row, 1 ms)

Edit

Checking the trace after executing put request in zipkin.

The screenshot shows the Zipkin UI interface. At the top, there's a search bar with 'serviceName' and 'order-management...' selected, and a 'RUN QUERY' button. Below that, it says '10 Results'. The main table lists the trace details:

| | Root | Start Time | Spans | Duration | |
|---|--|--|-------|----------|-----------------------|
| ^ | api-gateway: put | a few seconds ago (11/08 15:30:35.586) | 5 | 40.101ms | <button>SHOW</button> |
| | Trace ID: 64a49478df51d061 | | | | |
| | api-gateway (2) order-management-service (2) inventory-service (1) | | | | |
| ^ | order-management-service: post | a few seconds ago (11/08 15:30:22.209) | 1 | 2.769ms | <button>SHOW</button> |
| ^ | order-management-service: post | a few seconds ago (11/08 15:30:42.856) | 1 | 2.190ms | <button>SHOW</button> |
| ^ | order-management-service: get | a minute ago (11/08 15:30:18.302) | 1 | 1.525ms | <button>SHOW</button> |

A notification is sent to Notification service from order Service regarding the update of order along with the updated order details using rabbit MQ broker. We can see it in the console.

The screenshot shows an Eclipse IDE environment. On the left, there's a code editor for 'InventoryServiceApplication.java' with the following code:

```
1 package orderMyFood.NotificationService.consumer;
2
3 import org.springframework.amqp.rabbit.annotation.RabbitListener;
4
5 @Component
6 public class Consumer {
7
8     @RabbitListener(queues = RabbitMQConfig.PLACE_ORDER_QUEUE)
9     public void placeOrderStatusConsumer(OrderDto orderDto) {
10         System.out.println("Order Placed! Order details provided below");
11         System.out.println(orderDto);
12     }
13 }
```

On the right, there's a terminal window showing log output:

```
2022-11-08 13:03:49.793 INFO 6620 --- [ restartedmain] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2022-11-08 13:03:49.804 INFO 6620 --- [infoReplicator-0] com.netflix.discovery.DiscoveryClient : DiscoveryClient_NOTIFICATION-SERVICE/kib:p1kcdwyis: discovered service instance with ip:kib and port:5672
2022-11-08 13:03:49.875 INFO 6620 --- [ restartedmain] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#34002
2022-11-08 13:03:50.064 INFO 6620 --- [ restartedmain] o.N.NotificationServiceApplication : Started NotificationServiceApplication in 3.923 ms
Order Placed! Order details provided below
OrderDto{id=2, orderItems=[orderItem(id=2, itemCode=8b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=1)], totalPrice=20, status=PLACED, customerName=John Doe}
Order Placed! Order details provided below
OrderDto{id=3, orderItems=[orderItem(id=3, itemCode=8b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=1)], totalPrice=20, status=PLACED, customerName=Jane Doe}
Order Updated! Delivery details provided below
OrderDto{id=1, orderItems=[orderItem(id=null, itemCode=8b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=2)], totalPrice=40, status=PLACED, deliveryDetails={id=1, address=123 Main St, city=Anytown, state=CA, zip=90210}}
```

We can see graphical the message and time of transfer in Rabbit MQ UI regarding update order.

The screenshot shows the RabbitMQ Management UI. At the top, it says 'RabbitMQ Management' and 'RabbitMQ 3.8.6 Erlang 23.0.3'. The main area is titled 'Queue update_order' under the 'Overview' tab. It displays the following information:

- Queued messages (last minute):** A chart showing 0 messages.
- Ready:** 0
- Unacked:** 0
- Total:** 0
- Message rates (last minute):** A chart showing 0.3/s.
- Publish:** 0.00/s
- Deliver (manual ack):** 0.00/s
- Consumer ack:** 0.00/s
- Redelivered:** 0.00/s
- Get (auto ack):** 0.00/s
- Get (empty):** 0.00/s
- Get (manual ack):** 0.00/s

Retrieving the order based on id

The screenshot shows the Postman interface. The left sidebar has a 'History' section with several recent requests. The main area shows a GET request to `http://localhost:8080/api/v1/order/1`. The response body is displayed in JSON format:

```
1  {
2      "id": 1,
3      "orderItems": [
4          {
5              "id": 2,
6              "itemCode": "ce0f22ee-cebe-434d-a5bb-7a1675ed1206",
7              "price": 20,
8              "quantity": 5
9          },
10         {
11             "totalPrice": 100,
12             "status": "PLACED",
13             "address": "6-375 Rtc colony Hyderabad",
14             "phone": "6303778541",
15             "customerName": "Likitha Siddineni"
16         }
17     ]
18 }
```

The status bar at the bottom indicates a 200 OK response with 32 ms and 352 B.

Cancelling an order

The screenshot shows the Postman interface. The left sidebar has a 'History' section with several recent requests. The main area shows a DELETE request to `http://localhost:8080/api/v1/order/1`. The response body is displayed in Text format:

```
1 Order with id: 1 Canceled successfully
```

The status bar at the bottom indicates a 200 OK response with 22 ms and 154 B.

Checking the modified status. It is changed to Canceled.

The screenshot shows the H2 Console interface. The left sidebar lists tables: ORDERS_TABLE, ORDER_ITEMS, INFORMATION_SCHEMA, Sequences, and Users. The right side shows the results of a SQL query:

```
SELECT * FROM ORDERS_TABLE;
```

| ORDER_ID | LAST_UPDATED_ON | CUSTOMER_ADDRESS | CUSTOMER_NAME | DATE_OF_ORDER_PLACED | PHONE_NUMBER | STATUS | TOTAL_PRICE |
|----------|-------------------------|----------------------------|-------------------|-------------------------|--------------|----------|-------------|
| 1 | 2022-11-08 15:33:03.442 | 6-375 Rtc colony Hyderabad | Likitha Siddineni | 2022-11-08 15:32:35.692 | 6303778541 | CANCELED | 100 |

(1 row, 1 ms)

A notification is sent to Notification service from order service regarding the cancellation of the order along with the order details using rabbit mq. We can see the message in the console

```

1 package orderMyFood.NotificationService.consumer;
2
3 import org.springframework.amqp.rabbit.annotation.RabbitListener;
4
5 @Component
6 public class Consumer {
7
8     @RabbitListener(queues = RabbitMqConfig.PLACE_ORDER_QUEUE)
9     public void placeOrderStatusConsumer(OrderDto orderDto) {
10         System.out.println("Order Placed! Order details provided below");
11         System.out.println(orderDto);
12     }
13 }

```

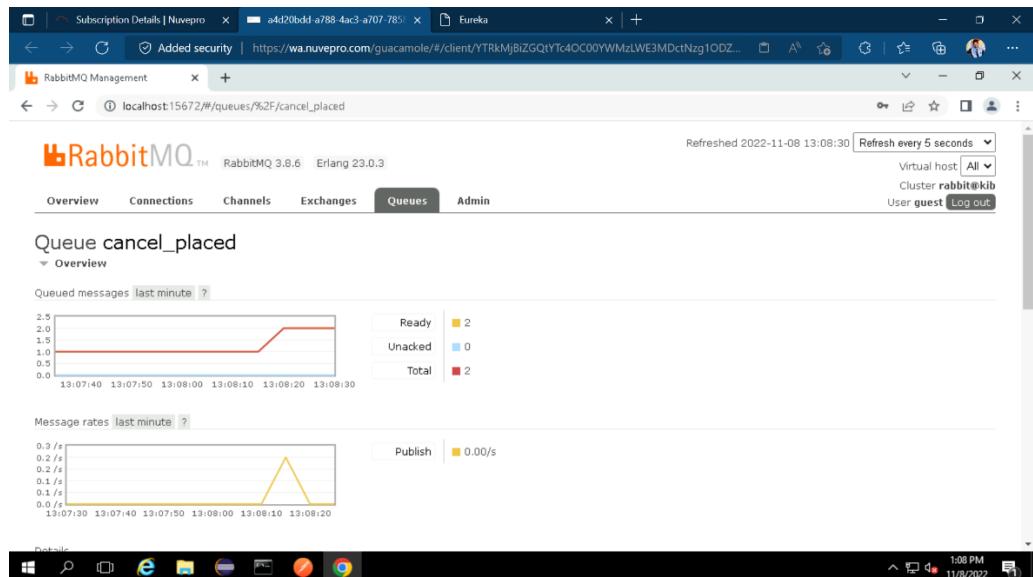
Console output:

```

NotificationServiceApplication [Java Application] C:\Program Files\Java\jre1.8.0_261\bin\javaw.exe (Nov 8, 2022 1:03:45 PM)
2022-11-08 13:03:49.875 INFO 6620 --- [ restartedMain] o.s.a.r.c.CachingConnectionFactory      : Created new connection: rabbitConnectionFactory34
2022-11-08 13:03:50.064 INFO 6620 --- [ restartedMain] o.N.NotificationServiceApplication      : Started NotificationServiceApplication in 3.923 seconds (JVM running for 4.002)
Order Placed! Order details provided below
orderDto[id=2, orderItems=[orderItemDto{id=2, itemCode=0b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=1}], totalPrice=20, status=PLACED, customerName=Bhargav Siddineni, address=6-375 Rtc colony Hyderabad, phone=6303778542]
Order Placed! Order details provided below
orderDto[id=3, orderItems=[orderItemDto{id=3, itemCode=0b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=1}], totalPrice=20, status=PLACED, customerName=Bhargav Siddineni, address=6-375 Rtc colony Hyderabad, phone=6303778542]
Order Updated! Delivery details provided below
orderDto[id=1, orderItems=[orderItemDto{id=null, itemCode=0b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=2}], totalPrice=40, status=PLACED, customerName=Bhargav Siddineni, address=6-375 Rtc colony Hyderabad, phone=6303778542]
Order Canceled! Order details provided below
orderDto[id=1, orderItems=[orderItemDto{id=4, itemCode=0b511aa9-bb87-4ab6-ac14-73740d0ea703, price=20, quantity=2}], totalPrice=40, status=CANCELED, customerName=Bhargav Siddineni, address=6-375 Rtc colony Hyderabad, phone=6303778542]

```

Graphical analysis of the time rate of the message that is sent to the notification service in rabbitMq UI.



If we are posting order having item quantity more than the actual quantity present it shows error as below.

POST http://localhost:8080/api/v1/order

Body

```

1 {
2     "customerName": "Bhargav Siddineni",
3     "phone": "6303778542",
4     "address": "6-375 Rtc colony Hyderabad",
5     "orderItems": [
6         {
7             "itemCode": "ce0f22ee-ebe4-434d-a5bb-7a1675ed1206",
8             "price": 20,
9             "quantity": 1000
10        }
11    ]
12 }
13

```

Body Cookies Headers (3) Test Results

404 Not Found 34 ms 184 B Save Response

The Items you are looking are not in stock, Please try later.

Same we observe in PUT operation

The screenshot shows the Postman interface with a PUT request to `http://localhost:8080/api/v1/order/1`. The request body is a JSON object:

```
[{"customerName": "Likitha Siddineni", "phone": "+9193778541", "address": "5-375 Rtc colony Hyderabad", "orderItems": [{"itemCode": "ce0f22ee-ebe-434d-a5bb-7a1675ed1206", "price": 20, "quantity": 50}]}]
```

The response status is 404 Not Found.

We can check the price of the item by calling to this end point before actual order is placed.

The screenshot shows the Postman interface with a GET request to `http://localhost:8080/api/v1/order/price`. The response body is a JSON object:

```
{"itemCode": "ce0f22ee-ebe-434d-a5bb-7a1675ed1206", "price": 20, "quantity": 5}
```

The response status is 200 OK.

Using maven -test for order management service such that jacoco report is created. We can see unit tests passed

The screenshot shows the Eclipse IDE interface with the Order Management Service project open. A terminal window at the bottom displays the output of the `maven -test` command, which generates a Jacoco report. The output includes logs from the OrderManagementServiceApplication class and the jacoco-maven-plugin.

```
[INFO] Tests run: 14, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:report (report) @ Order-Management-Service ---
[INFO] Loading execution data file E:\Bhargav\Projects\OrderMyFood Application\Order-Management-Service\target\jacoco.exec
[INFO] Analyzed bundle 'Order-Management-Service' with 18 classes
[INFO]
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 31.037 s
[INFO] Finished at: 2022-11-08T15:42:39+05:30
[INFO] -----
```

We can observe the order Controller report

The screenshot shows a browser window with multiple tabs open, including 'OrderManagementService' and 'orderMyFood.OrderManagementController'. The main content is a 'OrderManagementController' report from Jacoco. It displays a table of methods and their coverage metrics:

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed Cxt | Cov. | Missed Lines | Cov. | Missed Methods |
|-----------------------------|---------------------|------|-----------------|------|------------|------|--------------|------|----------------|
| updateOrder(OrderDto, Long) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1 | |
| getActualPrice(List) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1 | |
| placeOrder(OrderDto) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1 | |
| getOrderById(Long) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1 | |
| cancelOrder(Long) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1 | |
| OrderManagementController | 100% | n/a | 0 | 1 | 0 | 1 | 0 | 1 | |
| Total | 0 of 29 | 100% | 0 of 0 | n/a | 0 | 6 | 0 | 6 | 0 |

Created with Jacoco 0.8.8 2022-04-05 07:19

Order Management service layer jacoco report

The screenshot shows a browser window with multiple tabs open, including 'OrderManagementService' and 'orderMyFood.OrderManagementService_Service'. The main content is an 'OrderServiceImpl' report from Jacoco. It displays a table of methods and their coverage metrics:

| Element | Missed Instructions | Cov. | Missed Branches | Cov. | Missed Cxt | Cov. | Missed Lines | Cov. | Missed Methods |
|---|---------------------|------|-----------------|------|------------|------|--------------|------|----------------|
| cancelOrder(Log) | 0% | n/a | 1 | 1 | 3 | 3 | 3 | 1 | |
| placeOrder(OrderDto) | 77% | 50% | 1 | 2 | 3 | 10 | 0 | | |
| updateOrder(OrderDto, Long) | 89% | 50% | 1 | 2 | 2 | 14 | 0 | | |
| lambda\$updateOrder\$1\$1() | 0% | n/a | 1 | 1 | 1 | 1 | 1 | 1 | |
| checkItemStock(List) | 100% | 75% | 1 | 3 | 0 | 6 | 0 | | |
| getActualPrice(List) | 100% | 100% | 0 | 2 | 0 | 5 | 0 | | |
| OrderServiceImpl\$OrderRepository_RestTemplate_OrderManagementMapper_RabbitTemplate | 100% | n/a | 0 | 1 | 0 | 1 | 0 | | |
| getOrderById(Long) | 100% | n/a | 0 | 1 | 0 | 2 | 0 | | |
| lambda\$updateOrder\$2\$OrderItemsDto | 100% | n/a | 0 | 1 | 0 | 1 | 0 | | |
| lambda\$placeOrder\$0\$OrderItems | 100% | n/a | 0 | 1 | 0 | 1 | 0 | | |
| lambda\$cancelOrder\$4\$1 | 100% | n/a | 0 | 1 | 0 | 1 | 0 | | |
| lambda\$getOrderById\$3\$0 | 100% | n/a | 0 | 1 | 0 | 1 | 0 | | |
| static(...) | 100% | n/a | 0 | 1 | 0 | 1 | 0 | | |
| Total | 43 of 258 | 83% | 3 of 10 | 70% | 5 | 18 | 7 | 42 | 2 |

Created with Jacoco 0.8.8 2022-04-05 07:19

Inventory Service maven-test report, all unit tests are passed

The screenshot shows the Eclipse IDE interface with the 'Inventory Service' project selected in the Project Explorer. The terminal window at the bottom displays the output of a Maven test run:

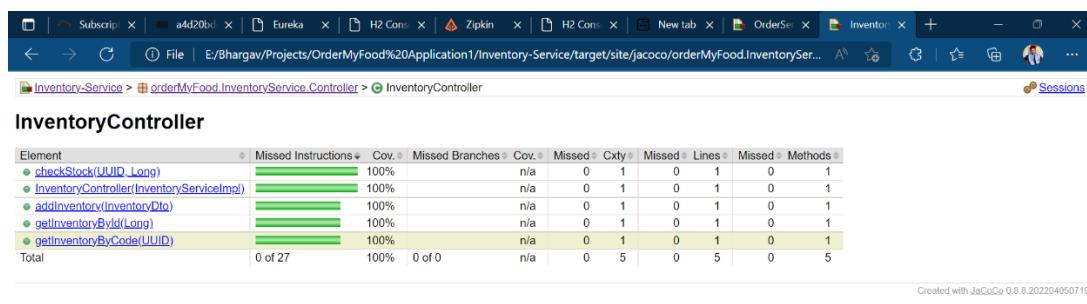
```

eclipse-workspace - Inventory Service/src/main/java/orderMyFood/InventoryService/InventoryServiceApplication.java - Eclipse IDE

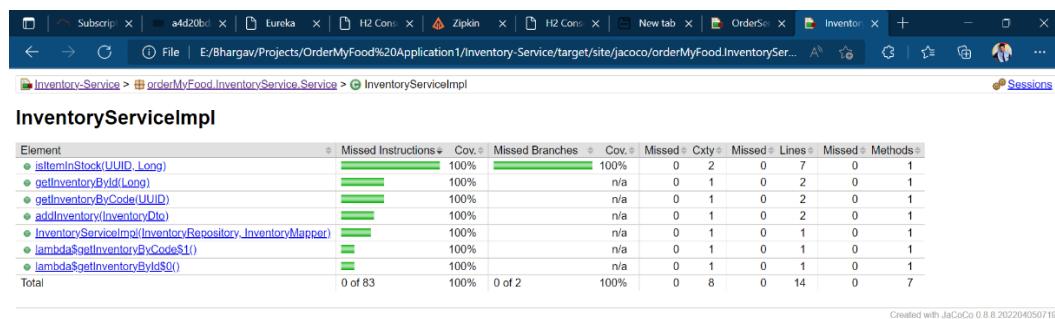
[...]
[INFO] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:report (report) @ Inventory-Service ---
[INFO] Loading execution data file E:\Bhargav\Projects\OrderMyFood\Application1\Inventory-Service\target\jacoco.exec
[INFO] Analyzed bundle 'Inventory-Service' with 13 classes
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 30.456 s
[INFO] Finished at: 2022-11-08T15:44:44+05:30
[INFO] -----

```

Inventory Controller Jacoco code coverage report



Inventory Service layer jacoco code coverage report



Here while placing the order if the inventory service is down then that exception is handled successfully using circuit breaker.

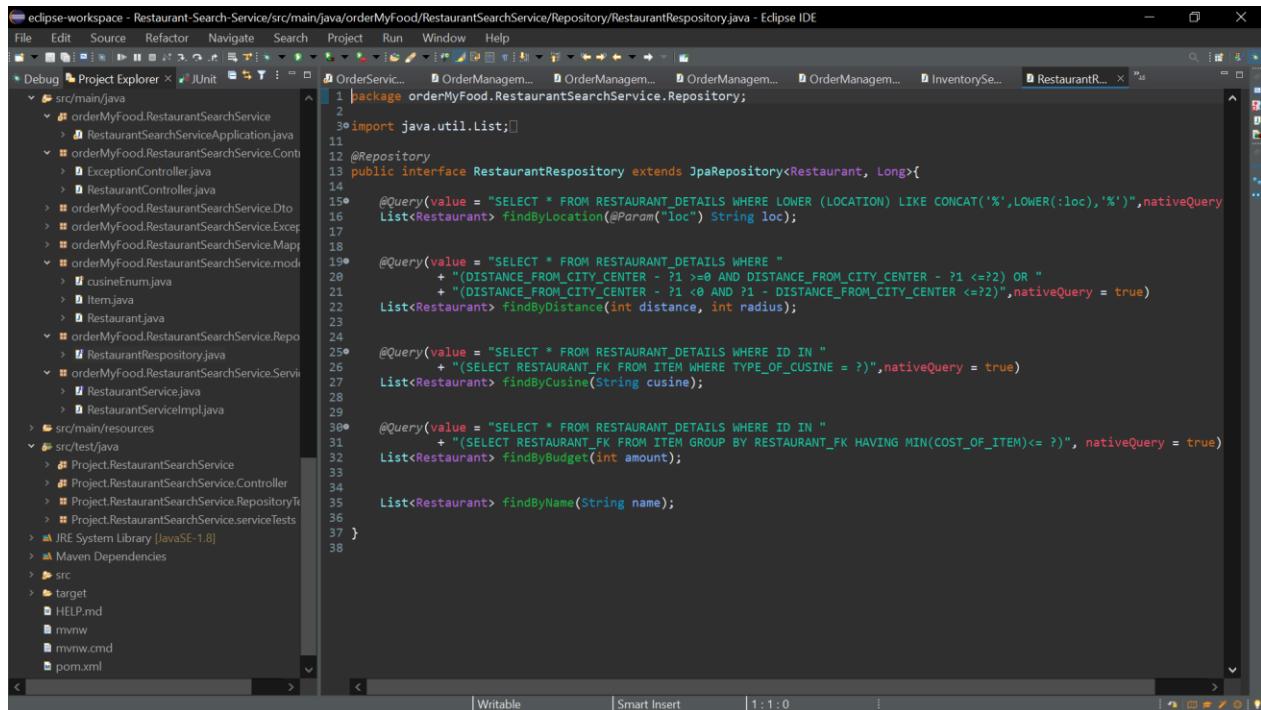
The screenshot shows a Postman interface with a collection named 'Today'. A POST request is being made to 'http://localhost:8080/api/v1/order'. The request body is:

```
1 "customerName": "Bhagav Siddineni",
2 "phone": "+9198778542",
3 "address": "6-375 Rtc colony Hyderabad",
4 "orderItems": [
5   {
6     "itemCode": "ce@f22ee-ebe-434d-a5bb-7a1675ed1206",
7     "price": 20,
8     "quantity": 1
9   }
10 ]
11 }
```

The response status is 201 Created, with a duration of 672 ms and a size of 170 B. The response body is:

```
1 Oops something went wrong please try again later!
```

Query Management for Restaurant -search service is done using native query technique



The screenshot shows the Eclipse IDE interface with the following details:

- Project Explorer:** Shows the project structure under "src/main/java".
- Code Editor:** Displays the `RestaurantRepository.java` file.
- Content:** The code implements a native query interface for a restaurant search service. It includes methods for finding restaurants by location, distance, cuisine, budget, and name.

```
1 package orderMyFood.RestaurantSearchService.Repository;
2
3 import java.util.List;
4
5 @Repository
6 public interface RestaurantRepository extends JpaRepository<Restaurant, Long>{
7
8     @Query(value = "SELECT * FROM RESTAURANT_DETAILS WHERE LOWER (LOCATION) LIKE CONCAT('%',LOWER(:loc),'%')",nativeQuery = true)
9     List<Restaurant> findByLocation(@Param("loc") String loc);
10
11     @Query(value = "SELECT * FROM RESTAURANT_DETAILS WHERE "
12             + "(DISTANCE_FROM_CITY_CENTER - ?1 >=0 AND DISTANCE_FROM_CITY_CENTER - ?1 <=2) OR "
13             + "(DISTANCE_FROM_CITY_CENTER - ?1 <0 AND ?1 - DISTANCE_FROM_CITY_CENTER <=2)",nativeQuery = true)
14     List<Restaurant> findByDistance(int distance, int radius);
15
16     @Query(value = "SELECT * FROM RESTAURANT_DETAILS WHERE ID IN "
17             + "(SELECT RESTAURANT_FK FROM ITEM WHERE TYPE_OF_CUSINE = ?)",nativeQuery = true)
18     List<Restaurant> findByCuisine(String cusine);
19
20     @Query(value = "SELECT * FROM RESTAURANT_DETAILS WHERE ID IN "
21             + "(SELECT RESTAURANT_FK FROM ITEM GROUP BY RESTAURANT_FK HAVING MIN(COST_OF_ITEM)<= ?)", nativeQuery = true)
22     List<Restaurant> findByBudget(int amount);
23
24     List<Restaurant> findByName(String name);
25
26 }
27
28
29
30
31
32
33
34
35
36
37 }
38
```