Coffee Order Processing Service

Microservices Engineer Assessment Project - Part 2 Implementation

Service Overview

The Process Order Service handles the complete lifecycle of coffee orders placed by customers, including:

Solution Overview

When a customer places an order, the system first validates that both the customer and shop are registered in the database. It then checks if the shop is currently open by verifying the current time against the shop's operating hours. The system confirms the requested queue number is valid for that location and ensures the queue isn't at full capacity. If all validations pass, it calculates the customer's position in the queue (current queue length + 1) and saves the order while incrementing the customer's loyalty score by 1 point. For status checks, the system retrieves the order details and calculates a 2-minute wait time per queue position. Order cancellation is only permitted for orders with a "WAITING" status. The queue status feature provides real-time queue length and wait time estimates (2 minutes per order) for each shop's queues. Throughout this process, the system enforces business rules including valid customer/shop registration, operating hour compliance, queue capacity limits, and status-based cancellation policies.

- Order creation and queue placement
- Real-time order status tracking
- Order cancellation
- Queue management and monitoring

> **Implementation Highlights:** Java Spring Boot (v3.x), PostgreSQL, Docker containerization, AWS deployment, Liquibase for database migrations, comprehensive error handling.

API Endpoints

POST `/api/v1/orders`

Creates and queues a new coffee order.

Request Example

```
{
    "customerId": 1,
    "shopId": 1,
    "itemName": "Cappuccino",
    "itemPrice": 4.50,
    "queueNumber": 1
}
```

Response Example (201 Created)

```
{
    "orderId": 123,
    "customerName": "John Doe",
    "shopName": "Downtown Coffee",
    "itemName": "Cappuccino",
```

```
    "itemPrice": 4.50,
    "queueNumber": 1,
    "queuePosition": 3,
    "status": "WAITING",
    "createdAt": "2023-07-15T09:30:45",
    "estimatedWaitMinutes": 6
}
```


GET `/api/v1/orders/{orderId}`

Retrieves the current status and queue position of an order.

Response Example (200 OK)


```
{
    "orderId": 123,
    "customerName": "John Doe",
    "shopName": "Downtown Coffee",
    "itemName": "Cappuccino",
    "itemPrice": 4.50,
    "queueNumber": 1,
    "queuePosition": 2,
    "status": "WAITING",
    "createdAt": "2023-07-15T09:30:45",
    "estimatedWaitMinutes": 4
}
```


DELETE `/api/v1/orders/{orderId}`

Cancels an existing order if it's still in the waiting queue.

Response

204 No Content on success

GET `/api/v1/orders/queue/{shopId}`

Retrieves queue status information for a specific shop.

Parameters

Parameter	Type	Description
queueNumber	integer	Optional. Specific queue number to check (default: 1)

Response Example (200 OK)


```
{
    "queueNumber": 1,
    "queueLength": 5,
    "estimatedWaitMinutes": 10
}
```

Error Handling

The service returns appropriate HTTP status codes with error details:

Error Code	Scenario	Response Example

400	Invalid request data	<pre>{
		  "timestamp": "2023-07-15T10:15:30",
		  "status": 400,
		  "error": "Bad Request",
		  "message": "Validation failed",
		  "details": ["itemPrice must be positive"]
		}</pre>

404	Order not found	<pre>{
		  "timestamp": "2023-07-15T10:15:30",
		  "status": 404,
		  "error": "Not Found",
		  "message": "Order not found"
		}</pre>

409	Business rule violation	<pre>{
		  "timestamp": "2023-07-15T10:15:30",
		  "status": 409,
		  "error": "Conflict",
		  "message": "Queue is full"
		}</pre>

Test Demonstration

Prerequisites

1. Docker installed
2. PostgreSQL (or use Docker container)
3. curl or Postman for API testing

Running the Service

Pull and run the Docker image:

```
docker pull uditha97/spring_order_process:v1.0
docker run -d -p 8080:8080 \
  -e SPRING_DATASOURCE_URL=jdbc:postgresql://host.docker.internal:5432/coffee_orders \
  -e SPRING_DATASOURCE_USERNAME=postgres \
  -e SPRING_DATASOURCE_PASSWORD=password \
  --name order_service uditha97/spring_order_process:v1.0
```

Test Scripts

Complete test sequence demonstrating all functionality:

```
#!/bin/bash
```

```
# 1. Create a new order
echo "Creating new order..."
ORDER_ID=$(curl -s -X POST http://localhost:8080/api/v1/orders \
  -H "Content-Type: application/json" \
  -d '{
    "customerId": 1,
    "shopId": 1,
    "itemName": "Latte",
    "itemPrice": 4.25,
    "queueNumber": 1
  }' | jq -r '.orderId')

echo "Created order ID: $ORDER_ID"

# 2. Check order status
echo -e "\nChecking order status..."
curl -s http://localhost:8080/api/v1/orders/$ORDER_ID | jq

# 3. Check queue status
echo -e "\nChecking queue status..."
curl -s "http://localhost:8080/api/v1/orders/queue/1" | jq

# 4. Cancel the order
echo -e "\nCanceling order..."
curl -s -X DELETE http://localhost:8080/api/v1/orders/$ORDER_ID

# 5. Verify cancellation
echo -e "\nVerifying cancellation..."
curl -s http://localhost:8080/api/v1/orders/$ORDER_ID | jq
```

> **Note:** The test script uses `jq` for JSON formatting. Install with `brew install jq` (Mac) or `sudo apt-get install jq` (Linux).

Docker & AWS Deployment

Why Docker?

- Consistent runtime environment across development, testing, and production
- Simplified deployment to AWS and other cloud platforms
- Easy scaling with container orchestration
- Isolation of dependencies

Deployment Architecture

The service is deployed as a Docker container on AWS with the following components:

- **Application Container**: Runs the Spring Boot service
- **PostgreSQL Container**: Database service (can be RDS in production)
- **Load Balancer**: Distributes traffic to multiple instances
- **Security Groups**: Restrict access to necessary ports

Production Deployment Steps

1. Push image to Docker Hub:

   ```
   docker tag spring_order_process uditha97/spring_order_process:v1.0
   docker push uditha97/spring_order_process:v1.0
   ```

2. Create AWS ECS cluster
3. Configure task definition with environment variables
4. Set up load balancer and auto-scaling

5. Configure monitoring and logging

Run Deployment Steps

Befor you run the deployment steps, ensure you have the following: Copy and pase the following file from the given repository to your ec2 instance:
`docker-compose.yml` file contains the necessary configuration.

To run the deployment steps, execute the following commands in your terminal:

1. sudo docker-compose down
2. sudo docker-compose up -d
3. sudo docker logs spring_boot_order_app -f

Repository Information

Complete source code and additional documentation:

- **Docker Hub Image:** `uditha97/spring_order_process:v1.0`
- **GitHub Repository:** https://github.com/udithanayanajith/coffe_Order_service.git

Conclusion

This Process Order Service provides a robust, scalable solution for managing coffee shop orders with:

- Clear RESTful API design following best practices
- Comprehensive error handling and validation
- Containerized deployment for easy scaling
- Complete test demonstration
- Production-ready configuration

The service meets all requirements specified in the assessment while following modern microservices architecture principles.