**Middleware Technologies Assignment**

Course Code: **CSIWZG524**

Assignment topic: **Middleware for Online Food Delivery Platform**

Batch: **SAT\_10**

Developed By:

1. **Abid Rafique Khan – BITS ID (2021WA86438)**
2. **Anuj Suryavansh – BITS ID (2021WA56448)**
3. **Siddesh Shejole – BITS ID (2021WA86442)**
4. **Ashish Bharti – BITS ID (2021WA84440)**
5. **Happy Pawha – BITS ID (2021WA86432)**

GitHub Repository Link: [**https://github.com/AbidKhan01ak/Middlleware-project---Online-Food-Delivery-System**](https://github.com/AbidKhan01ak/Middlleware-project---Online-Food-Delivery-System)

Tech Stack used:

|  |  |  |  |
| --- | --- | --- | --- |
| Backend | Frontend | Testing | Message Queue |
| Java 17 | **React + Vite** | **Postman** | **RabbitMQ** |
| Spring-boot | **Tailwind CSS** | **Browser console log** |
| Spring-boot dev tools | **Axios** | **React Dev tools** |
| Spring JPA | **Node** |  |
| Jackson | **Npm** |
| Hibernate | **TypeScript** |
| Maven |  |
| MySQL |

Date: **10-July-2025**

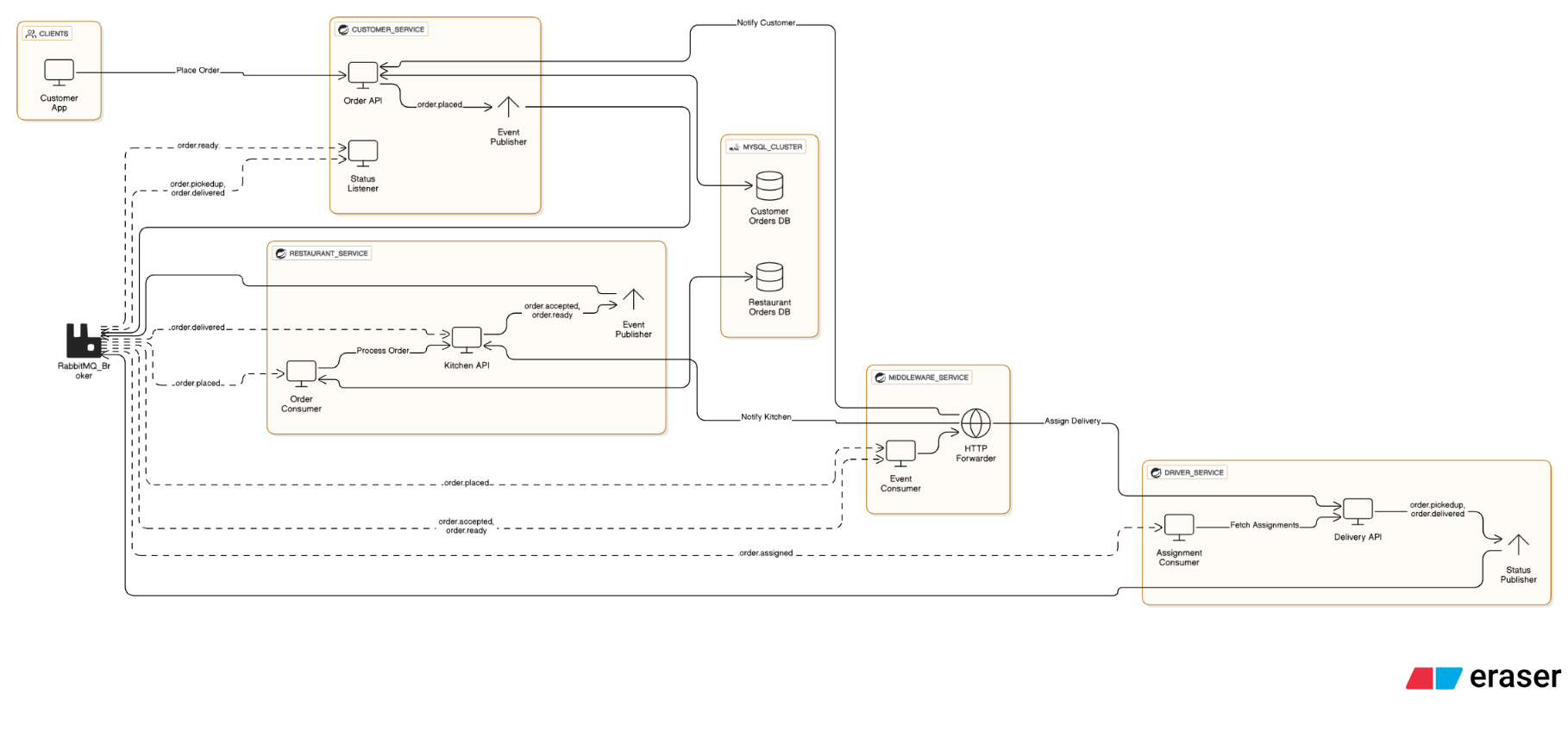
**Table of Contents**

|  |  |
| --- | --- |
| ***Index*** | ***Page No*** |
| **Introduction** | **3** |
| **Abstract** | **5** |
| **Literature Work** | **6** |
| **Implementation** | **7** |
| **Testing** | **9** |
| **Result and Discussion** | **11** |
| **Future Improvement** | **13** |
| **Conclusion** | **15** |
| **Screenshots** | **16** |
| **Code Snippets** | **31** |
| **References** | **52** |

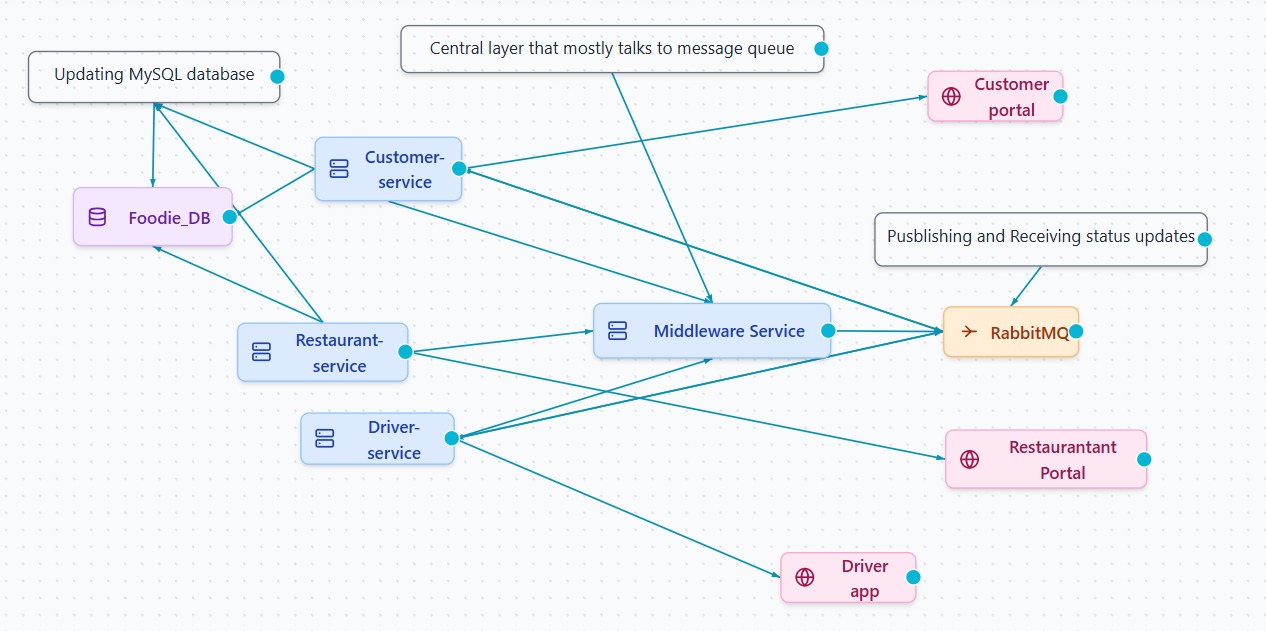
**Introduction**

This project deals with the design and development of a middleware-based online food delivery system which links seamlessly with customers, restaurants, and food delivery drivers. The primary aim is to deploy a scalable, efficient, and real-time communication system to handle order placement, dispatching, and delivery monitoring through the application of middleware layers. The middleware is the platform's central communication spine, which supports message routing, event processing, and data sharing among the frontend application, restaurant portal, and driver app. This facilitates a loosely coupled and modular architecture by microservices, with each component running independently but being connected with a stable messaging system.  
  
Customer-service, restaurant-service, and driver-service backend services  
These backend services are implemented with Java 17 and Spring Boot. These services exposes RESTful APIs and hides business logic behind Spring's ecosystem, including Spring Data JPA for persistence, Hibernate for ORM, and Jackson for serialization of JSON. Spring DevTools accelerates development with features such as hot reload. The tool Maven is used for dependency management and build automation, while Postman is utilized for backend API testing. Development and debugging are performed in IntelliJ IDEA.  
  
For event-driven, real-time communication, the project uses RabbitMQ as a middleware message broker. RabbitMQ provides asynchronous, reliable, and scalable delivery of messages between services. It manages queuing and routing of messages—like when a customer orders food, a restaurant posts a status message, or a driver updates delivery status. This decoupled approach enhances system responsiveness, lessens latency, and enables the system to efficiently manage high traffic and concurrent users.  
  
The frontend interfaces are developed using React with Vite for efficient builds and development. Addition of TypeScript gives type safety and improves maintainability, and Tailwind CSS provides a clean and responsive UI design. HTTP communication between frontend and backend is facilitated by Axios. The customer app, restaurant portal, and driver app all have a uniform UI/UX core for smooth user interaction. It is developed using Visual Studio Code with assistance from browser tools such as React Developer Tools for debugging.  
  
For persistent storage of data, MySQL is utilized by the platform, being managed via MySQL Workbench. This guarantees secure and efficient data operations with all services. Code quality is ensured through tools like Prettier and ESLint, which are set up to ensure consistent code styling and catch problems early during development.  
  
All in all, this project integrates contemporary technologies and software engineering best practices to create a strong, scalable, and maintainable food ordering system adequate for real-world implementation and future extension.

**Architecture:**



**High Level Design Diagram:**

****

**Abstract**

The accelerated growth of digital technology has revolutionized the way consumers get basic services, with the food delivery industry being highly transformed. This work introduces the design and development of an online food delivery system based on middleware that integrates customers, restaurants, and delivery drivers in a single system. The main aim is to develop a scalable, effective, and real-time order placement, dispatching, and delivery status system through a centralized middleware framework.  
  
Upon which sits the middleware layer, built with RabbitMQ, a trusted and extensively used message broker. This layer orchestrates event-driven, asynchronous communication between services so that data flows smoothly and updates in real-time. Loose coupling is encouraged through the architecture so that components such as frontend, restaurant portal, and driver application may run independently but remain synchronized by way of message queues.  
  
The backend is built in Java 17 and the Spring Boot framework, utilizing Spring Data JPA, Hibernate, and Jackson for ORM and JSON. Lombok is used to minimize boilerplate code, and Postman is used to test and debug APIs. A MySQL relational database is employed for persistent storage of data, managed through MySQL Workbench and accessed using the native MySQL connector.  
  
At the frontend, up-to-date web technologies have been utilized, such as React with TypeScript, packed with Vite for performance development improvement. Tailwind CSS styles the UI, and Axios is utilized for API calls. Development is done in Visual Studio Code, aided by React Developer Tools. Code consistency and quality are ensured through Prettier and ESLint.  
  
This project illustrates the successful integration of middleware technology with modern web frameworks to provide a responsive, modular, and user-focussed food ordering solution. The architecture's focus on real-time communication and scalability positions it well for real-world deployment and extension in the future.

**Literature Work**

Middleware in distributed systems has undergone tremendous change over the last two decades with increased focus on modularity, scalability, and real-time data transfer. Middleware acts as an intermediary communication bridge that hides the intricacies of direct client-server communications. In service-oriented and microservices architecture, research continues to emphasize the application of middleware to separate tightly coupled services, thus enhancing system maintainability, fault tolerance, and scalability.

RabbitMQ, an open-source message broker, is used extensively in both industrial practice and academic work because it supports asynchronous communication, message routing, load balancing, and event-driven architecture. RabbitMQ uses the Advanced Message Queuing Protocol (AMQP) optimized for high-throughput and low-latency communications—suitable for real-time platforms such as online food delivery systems where constant status updates and rapid response time are essential.

Further, real-time order tracking is another area of emphasis in contemporary middleware research. Middleware behaves as an intermediary that ensures synchronization among all parties—customers, restaurants, and drivers—without involving direct, tightly coupled communication. RabbitMQ's message buffering, routing, and retry features complement system reliability further, as validated in various benchmark-based studies.

From both frontend and user experience standpoints, Single Page Applications (SPAs) with frameworks such as React find great backing in literature. With inclusion of tools such as Vite for quick bundling and Tailwind CSS for utility-first styling, SPAs help with quick load times and better interactivity—pillars that are crucial to food ordering applications that are real-time.

The utilisation of TypeScript offers static type checking, which narrows down runtime errors and enhances long-term maintainability. Surveys among developers uniformly identify the advantages of TypeScript in developing large-scale applications. Likewise, Axios is well known for making HTTP communication easier, while ESLint and Prettier are commonly used to ensure code quality and consistency. All these technologies, when combined with a reactive backend and middleware system, facilitate the creation of end-to-end, high-performance, and user-centric food delivery platforms.

**Implementation**

The system is designed on a microservices architecture, where the backend is separated into three separate services: customer-service, restaurant-service, and driver-service. Every service handles its own domain logic, data storage, and API endpoints. All these services communicate with each other via a core middleware-service, that handles inter-service communication and order workflow orchestration.

Once the customer orders through the frontend portal, the request is first processed by the customer-service, where it is validated and stored. The order information is then passed on to the middleware-service, a message relay hub. Here, the message is relayed to the restaurant-service, which informs the respective restaurant through its respective frontend portal. This decoupled architecture guarantees that each service may be developed, deployed, and scaled separately, but business logic is isolated and maintainable.

To facilitate real-time communication and provide resilience, the middleware layer is closely coupled with RabbitMQ, a highly resilient message broker that provides asynchronous, event-driven messaging. Order placement, acceptance, preparation, pickup, en route, and delivery events are sent as messages to respective queues or exchanges in RabbitMQ. The messages are processed by the concerned services or frontend clients, which subsequently initiate updates in their respective domains.

For instance:

* When a restaurant accepts an order, the restaurant-service sends a message to RabbitMQ.
* The middleware-service receives this message and broadcasts it to the customer-service and driver-service.
* Customers get real-time updates on order status, and available drivers get dispatched.
* This asynchronous event-driven design breaks tight coupling among services, enhances fault tolerance, and promotes horizontal scalability, particularly during high-load scenarios.

On the client-side, each user group—customer, restaurant, and driver—is served by a distinct Single Page Application (SPA) coded with React and driven by Vite for improved build times and hot module replacement. Each portal has specific functionality:

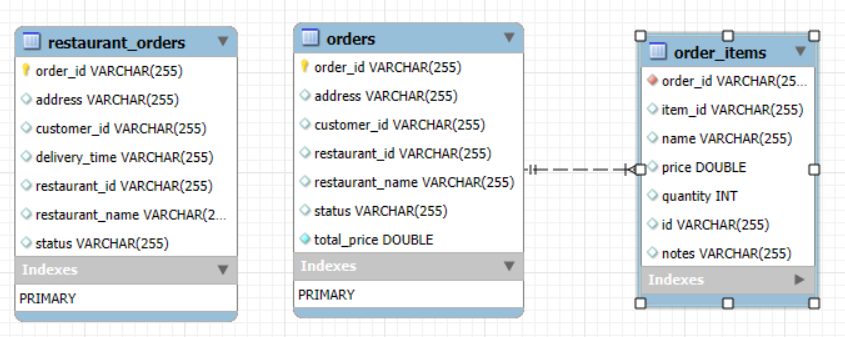
* Customers can search restaurants, order, and monitor live order status.
* Restaurants receive incoming orders and can accept or decline them.
* Drivers are assigned orders and mark delivery status (e.g., "Picked up", "En route", "Delivered").

Frontend-backend communication is handled using Axios for HTTP requests, and backend updates are reflected in near real-time through messages processed through RabbitMQ.

To improve code quality and user experience:

* TypeScript applies strict type checking and minimizes bugs.
* Tailwind CSS is employed for responsive and contemporary UIs.
* Utilities like Prettier and ESLint provide code consistency and maintainability.

**Database ER Diagram:**

****

**Testing**

Testing in this microservices-based food ordering system was performed at various levels to validate functionality, reliability, and smooth integration between all parts. A strong testing policy was employed throughout the backend services, middleware, and frontend interfaces.

Unit Testing (Backend Services)

Each of the backend modules—customer-service, restaurant-service, and driver-service—was unit-tested to ensure internal business logic in isolation. Utilizing Spring Boot's test framework along with JUnit, unit tests were implemented to test major operations including:

* Order placement and verification
* Status updates at different stages (accepted, picked up, delivered)
* Invalid input handling (e.g., non-existent order IDs or malformed payloads)

Edge cases, including parallel updates or missing fields in request bodies, were thoroughly tested as well to provide robustness and predictability under various usage patterns. These unit tests caught bugs early and made it possible to ensure that every service was able to run independently.

Integration Testing (Middleware & Inter-Service Communication)

Integration testing was done extensively using Postman to verify the end-to-end flow. These tests mimicked real-world API calls between services, including scenarios like:

* Customer order placement
* Restaurant order acceptance
* Driver pickup instructions receipt
* Final delivery status and confirmation

RabbitMQ queues and exchanges were monitored under integration tests to confirm message publishing and consumption were behaving as required. Test cases also included error handling cases, such as:

* Temporary service outage
* RabbitMQ delivery failure
* Duplicate or out-of-order message

These tests verified the fault-tolerance and resilience of the middleware-based communication, ensuring that the system was able to recover gracefully from message delays or partial failures.

Frontend Testing (Customer, Restaurant, and Driver Portals)

Frontend, both manual and automated testing techniques were used to validate user interface behavior and real-time data synchronization:

* React Developer Tools and browser console logs were utilized to examine state changes, props flow, and debug rendering issues.
* Axios request/response streams were traced for appropriate API integration and error feedback.
* Real-time update testing was performed by pretending multiple users (customer, restaurant, and driver) interacting at the same time on various browsers and devices.
* UI responsiveness, user feedback messages (such as loading spinners, status alerts), and design consistency were confirmed through cross-browser testing.

Also, code quality and maintainability were maintained by enforcing linting and formatting standards via ESLint and Prettier so that the frontend code is clean and readable in all modules.

**Result and Discussion**

The release of the middleware-based online food ordering system produced extremely satisfactory results in regard to performance, modularity, scalability, and user satisfaction. The microservices of the platform—customer-service, restaurant-service, and driver-service—were able to function autonomously but communicate perfectly through the central middleware-service. This loosely coupled architecture enabled by RabbitMQ tremendously enhanced responsiveness and system maintainability.

One of the standout achievements was the efficiency of the platform in providing real-time status updates with little latency. For instance, when a customer ordered, the target restaurant received and processed it virtually immediately, and the acknowledgement was sent back within milliseconds. These asynchronous message exchanges routed through RabbitMQ ensured that true real-time communication was possible with no service dependencies, the hallmark of good event-driven system design.

In stress testing and simulated real-world operations, the system effectively processed several orders concurrently without message loss, duplication, or delay. RabbitMQ efficiently delivered messages to their target consumers, and message acknowledgments provided guaranteed delivery, even in the presence of high traffic volumes. Users in all roles—customer, restaurant, and driver—were provided with timely and correct updates at every point in the order cycle.

Most importantly, failure modes—e.g., transient service unavailability or RabbitMQ restarts—were exercised and didn't lead to a platform-wide failure. This confirmed the fault tolerance of the system by showing asynchronous communication delivers not just improved performance but also distributed fault tolerance and graceful degradation in the event of service outages.

On the client-side, the React-driven SPAs for customers, restaurants, and drivers were incredibly responsive, with low load times and seamless navigation on different devices and browsers. Deployment was supported by fast development iterations and smooth bundling thanks to the use of Vite. Real-time status updates appeared in the UI accurately, confirming frontend components were properly in sync with backend events through Axios and RabbitMQ-based triggers.

User response was always very positive, particularly about the clean and intuitive UI of the platform built with Tailwind CSS. The union of a contemporary tech stack and a solid backend-middleware architecture ended up in a system that is not only technologically solid but also very usable and responsive.

In summary, the experiment confirmed the possibility and efficacy of employing middleware-based microservices architecture to drive an actual food delivery system in real time. The findings confirm the readiness of the platform for future development and possible real-world implementation, with both robust underpinnings in software design and user experience.

**Future Improvement**

Though the present system effectively facilitates real-time food ordering and delivery by virtue of a middleware-based microservices architecture, there are a few modifications that can be made to enhance functionality, scalability, user interaction, and production deployment readiness.  
  
**1.** **Notification System Integration**  
At present, all the notifications are in-app. A major upgrade would be to introduce a multi-channel notification system (via email, SMS, or push notifications) to notify customers, restaurants, and drivers when they are not on the platform actively using it. It will enhance reliability and user engagement, particularly in real-world deployments.  
  
**2. Payment Gateway Integration**  
The present implementation is presuming manual or external processing of payments. To make the platform production-ready, integration with secure and popular payment gateways like Razorpay, Stripe, or PayPal would be crucial. This would simplify the transaction flow and ensure a secure and seamless checkout experience for customers.  
  
**3. Smart Order Dispatching with AI**  
Currently, dispatching relies on simple parameters like driver availability and location. The integration of machine learning algorithms would make dispatching smarter by considering:  
 a. Real-time traffic conditions

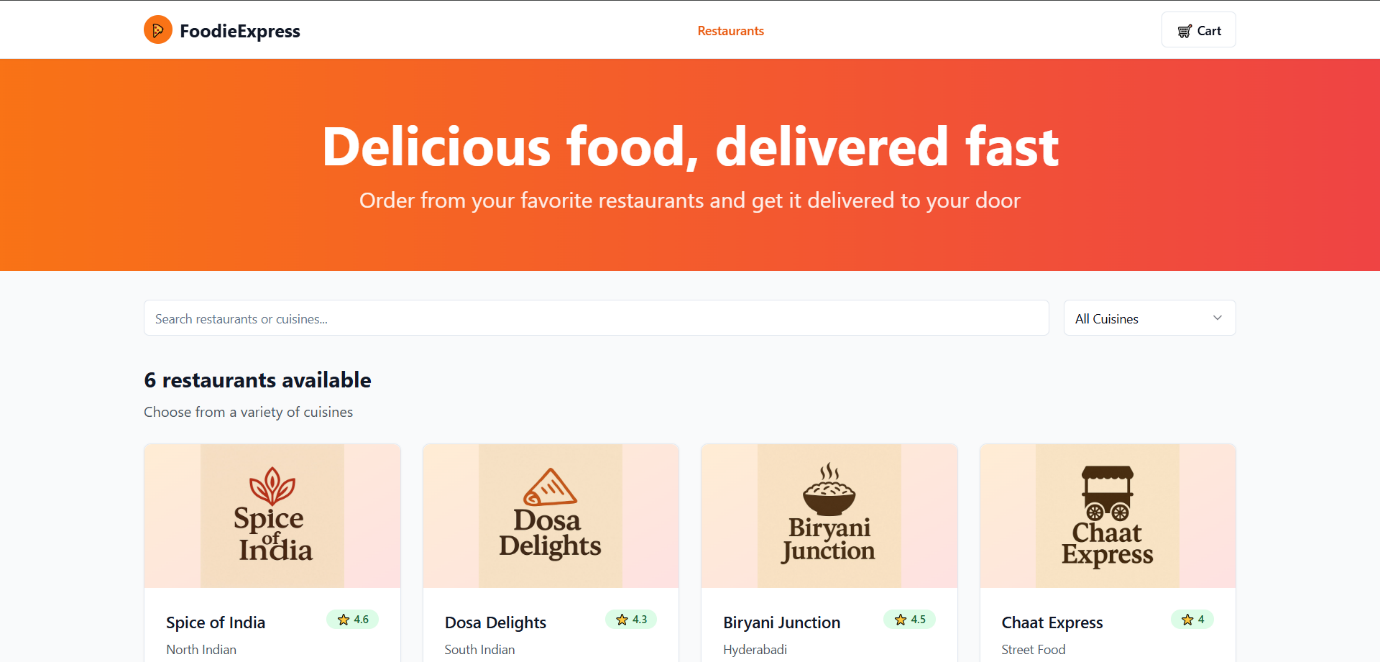
b. Historical delivery patterns  
 c. Driver performance and efficiency  
  
This smart dispatching would increase delivery speed as well as customer satisfaction.  
  
**4. Rating and Feedback System**  
Inclusion of a review and rating system for customers to rate restaurants and delivery drivers would:  
 a. Increase transparency and trust  
 b. Help ensure service quality  
 c. Motivate improved performance via user feedback  
  
This would also allow the system to filter and suggest top-rated restaurants or drivers in later iterations.  
  
**5. Restaurant Analytics Dashboard**  
Having an analytics dashboard for restaurants could provide insights into:  
 a. Peak order hours  
 b. Most popular dishes  
 c. Average delivery times  
 d. Trends of orders over time  
  
These insights would make data-informed decisions possible for restaurant owners to enhance efficiency and revenue.  
  
**6. Infrastructure Scaling and Deployment Improvements**  
To enable enterprise-scale scaling, the application can be containerized with Docker and orchestrated with Kubernetes. This would enable:  
 a. Automated scaling during traffic bursts  
b. Optimized resource utilization  
c. Simpler deployment across cloud providers such as AWS, Azure, or GCP  
  
In addition, having an API Gateway (e.g., Spring Cloud Gateway or Netflix Zuul) and Service Registry (e.g., Eureka) in place would improve request routing, load balancing, and service discovery—particularly in an expanding microservices environment.  
  
**7. Security and Access Control**  
Adding user authentication and authorization is essential for production use. Incorporating standards like OAuth2 and JWT (JSON Web Tokens) would enable:  
  
 a. Secure user session and login management  
 b. Role-based access control (customer, restaurant, driver)  
 c. Securing sensitive activities like placing orders, status tracking, and accessing data  
  
Making these enhancements would greatly improve the reliability, scalability, and performance of the platform, and would enable enterprise-grade deployment of the system in real-world food delivery situations.

**Conclusion**

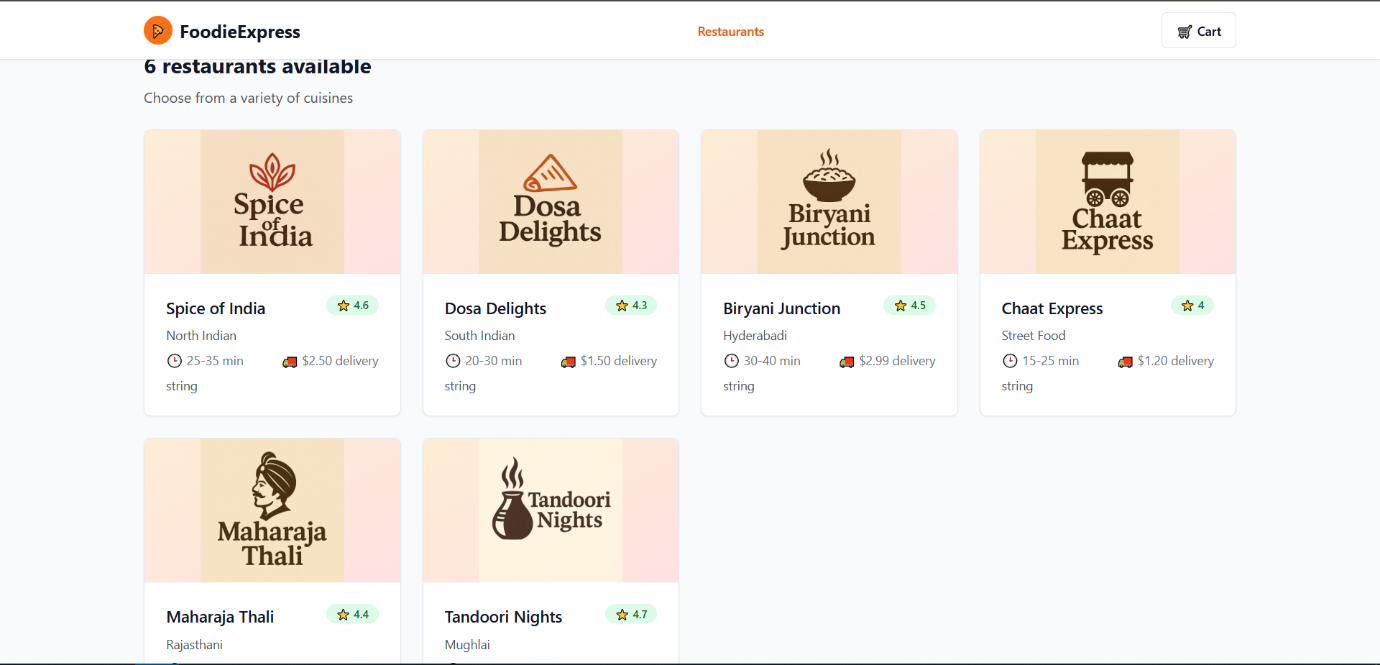
This project is able to effectively illustrate the design and implementation of an online food delivery system using a contemporary microservices-based architecture enabled by a middleware-based communication layer. Through the separation of the system into modular pieces—customer-service, restaurant-service, driver-service, and the central middleware-service—we were able to effectively provide a separation of concerns, leading to a system that is simpler to develop, keep up with, scale, and extend.  
  
The middleware, which was deployed with RabbitMQ, proved to be instrumental in facilitating asynchronous, event-driven communication between independently deployed services. This design choice decoupled the system components without losing real-time synchronization between them, which resulted in a more robust and agile solution compared to common monolithic platforms.  
  
The platform supports all primary features of a contemporary food delivery system:  
 a. Customers are able to place and monitor orders.  
 b. Restaurants can accept and process requests.  
c. Drivers are able to see assignments and mark delivery status as updated.  
  
Both user groups communicate via their own Single Page Application (SPA), built on React, Vite, Tailwind CSS, and TypeScript—providing a responsive and user-friendly interface. Real-time updates were obtained through a combination of RabbitMQ messaging and Axios-based API integration, while development tools like Postman, React Developer Tools, ESLint, and Prettier improved debugging, testing, and code quality.  
  
System-wide testing confirmed the fault tolerance, scalability, and support for concurrent operations of the platform without performance loss or message loss. The success of this project once again justifies the importance of incorporating middleware technologies in distributed systems to create scalable, modular, and responsive applications.  
  
In short, this project not only meets its operational needs but also lays a robust foundation for future development, such as intelligent dispatching, payment gateway, real-time analytics, and user authentication. The architecture and implementation are a realistic template for rolling out real-time, middleware-driven applications in the food delivery space—and maybe in other service-based industries as well.

**Live Application Screenshots**

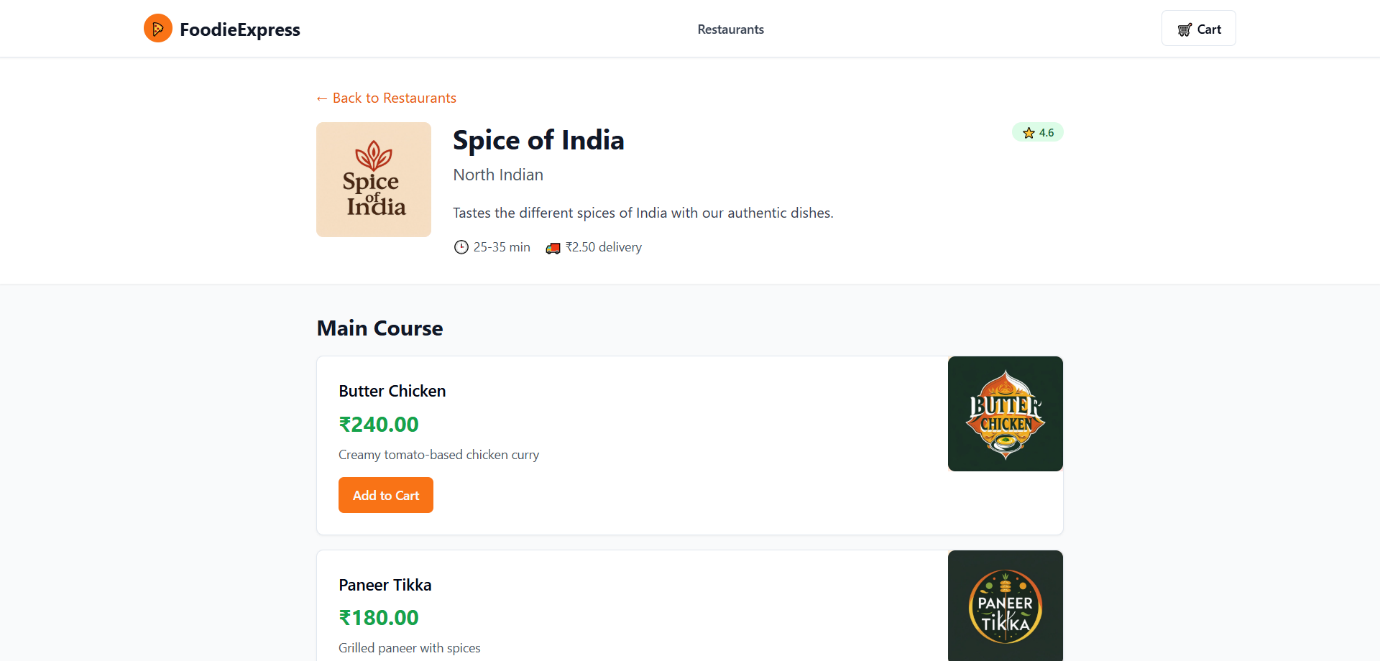
* Customer frontend Homepage

****

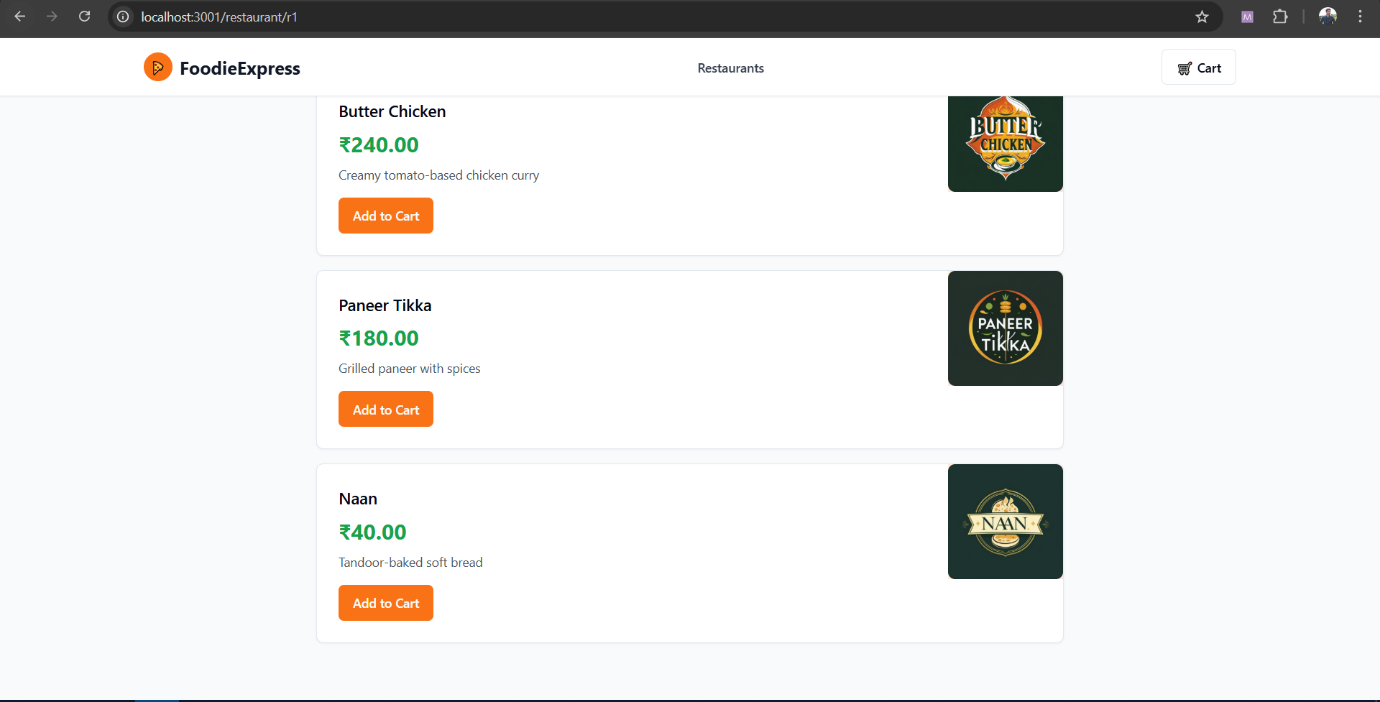
* Available restaurants in customer portal

****

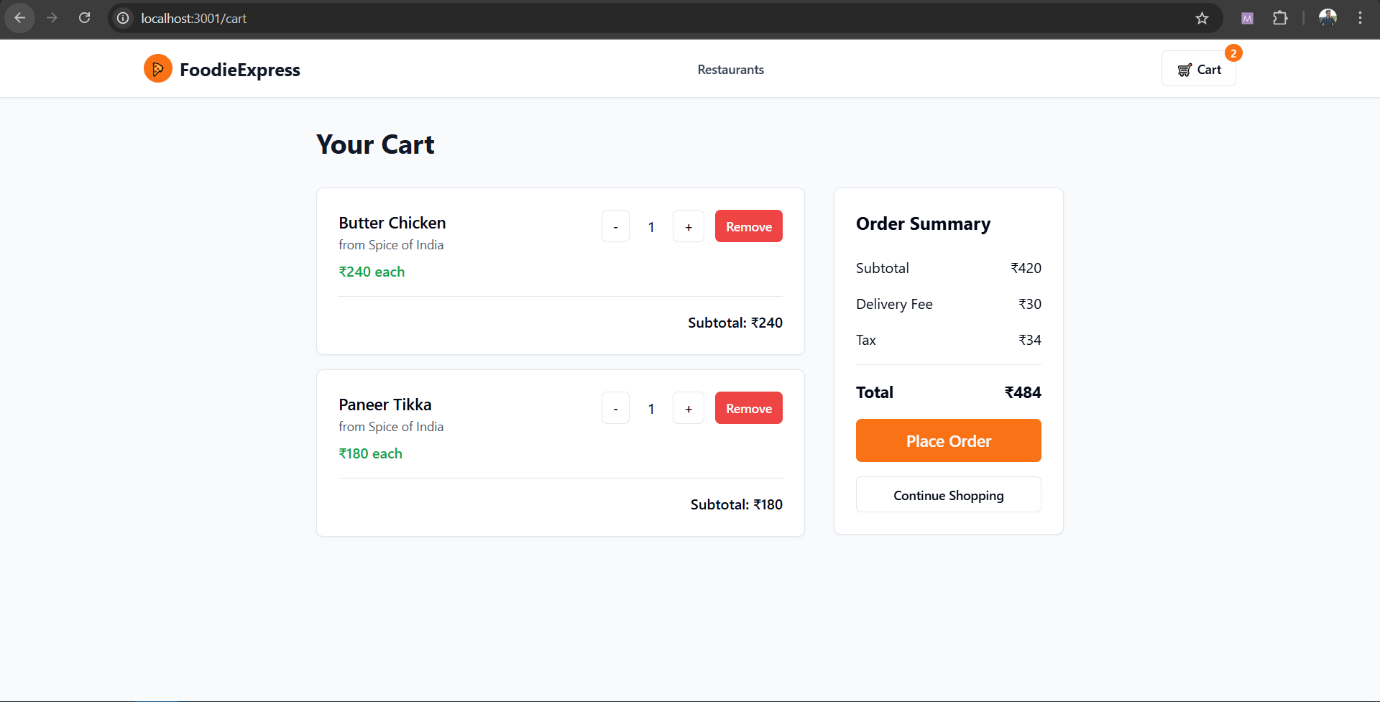
* Dishes available in a restaurant to order.

****

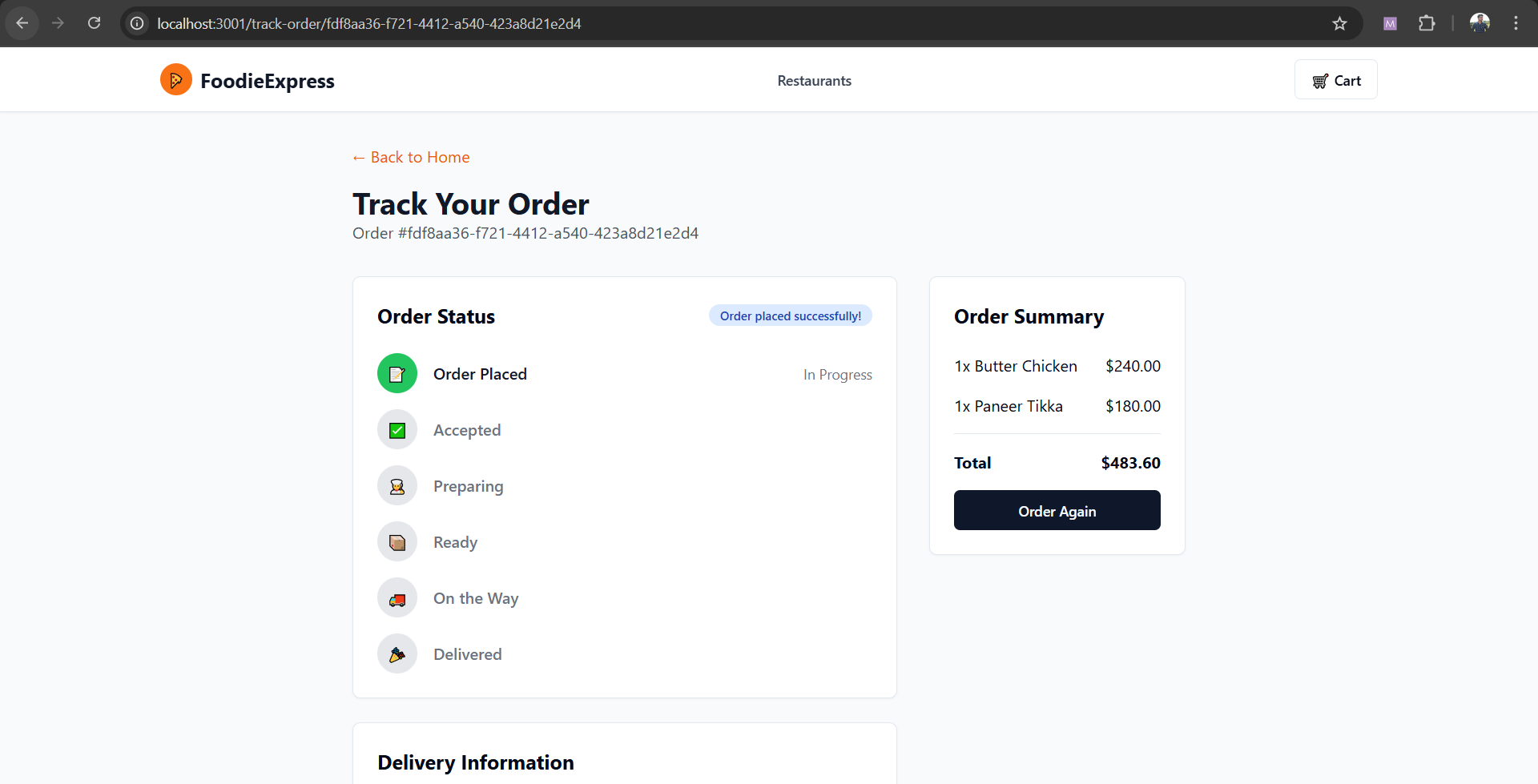
* List of Dishes with description and price

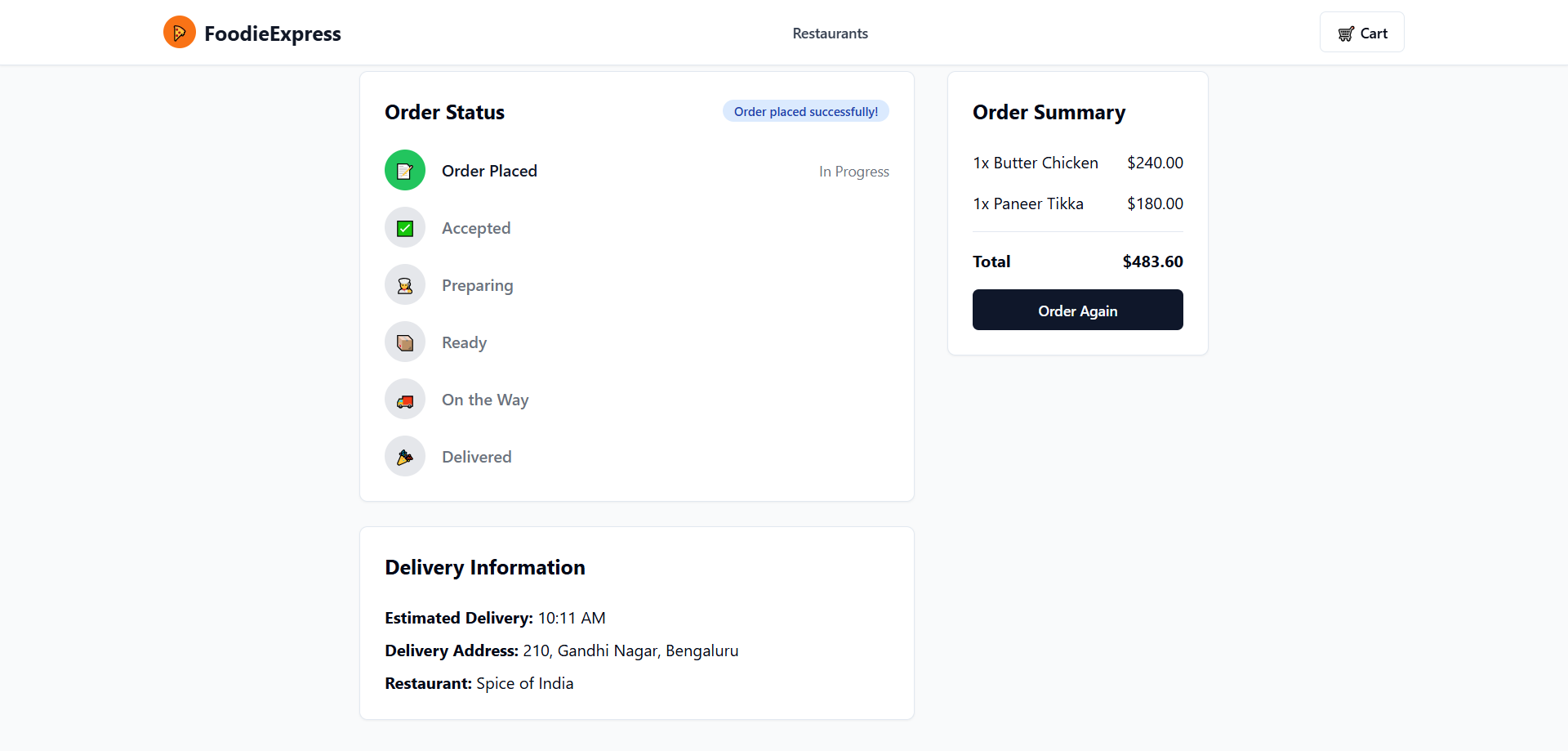
****

* Cart which add the items and show order summary along with Delivery fee and tax

****

* Order placed, shows order status.

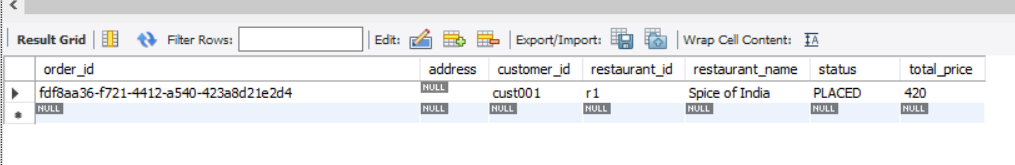
****

****

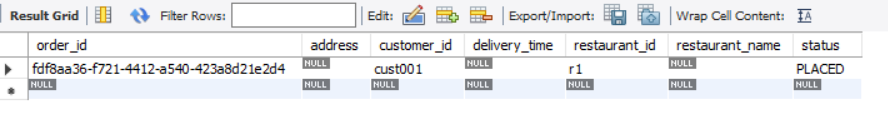
* Restaurant received order placed update via RabbitMQ.

****

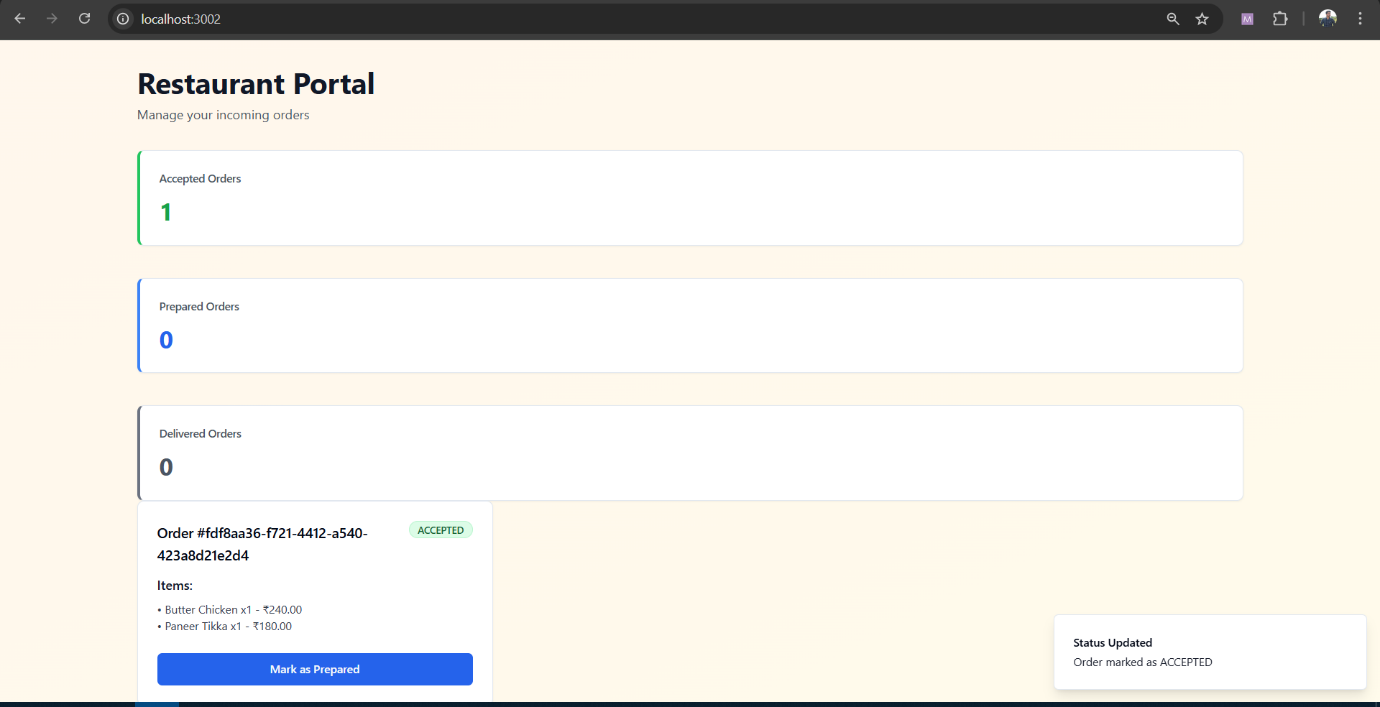
* Orders table updated with order\_id, status and rest of the details.

****

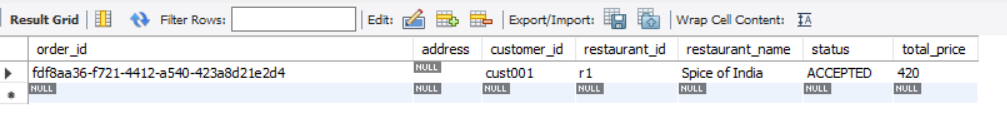
* Order with same order\_id populate in restaurant\_orders table with same status.

****

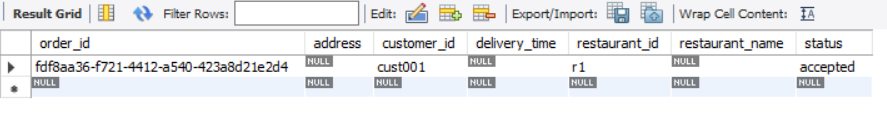
* Restaurant “accepts” the order. Status changed to “ACCEPTED”

****

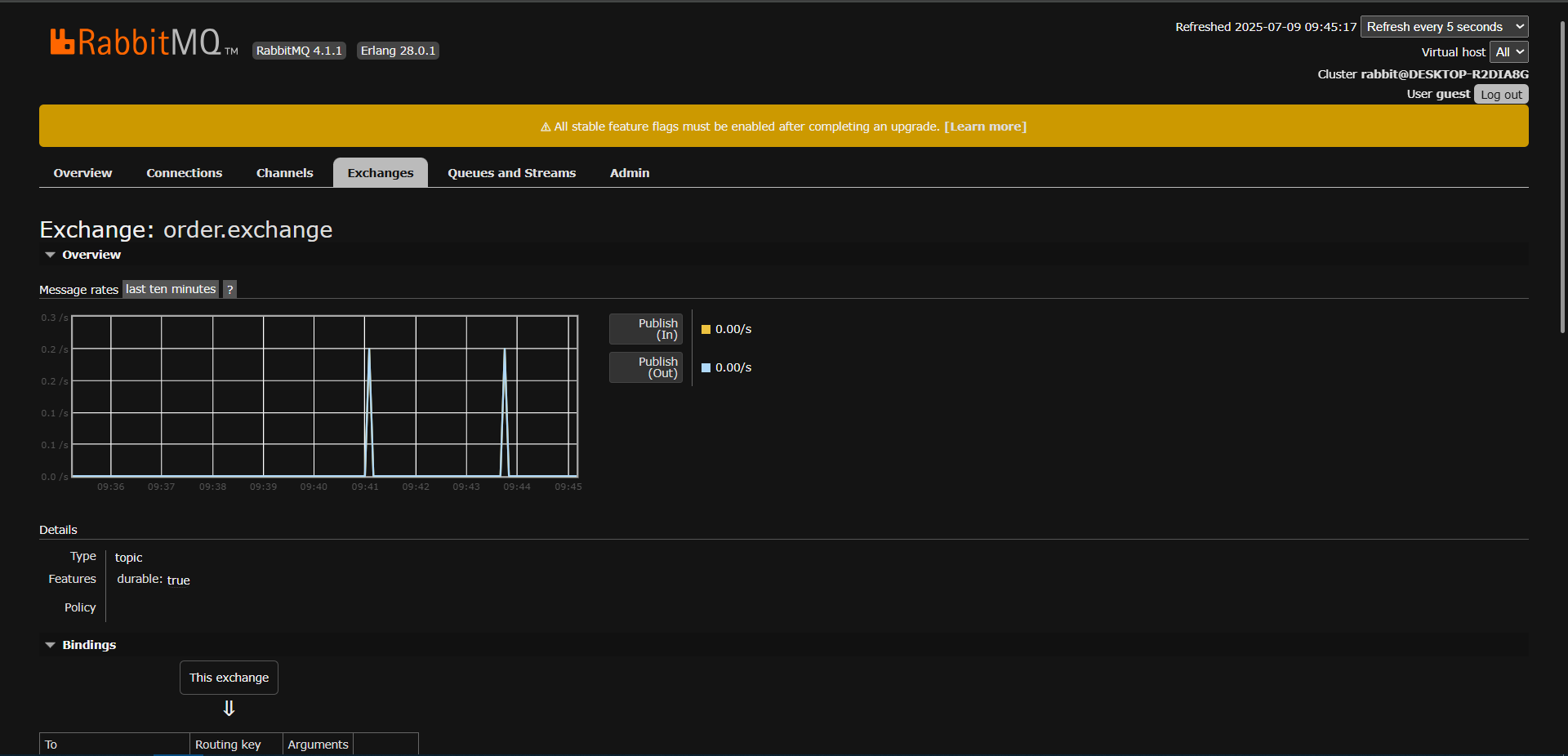
* Status changed in orders table to “ACCEPTED”.

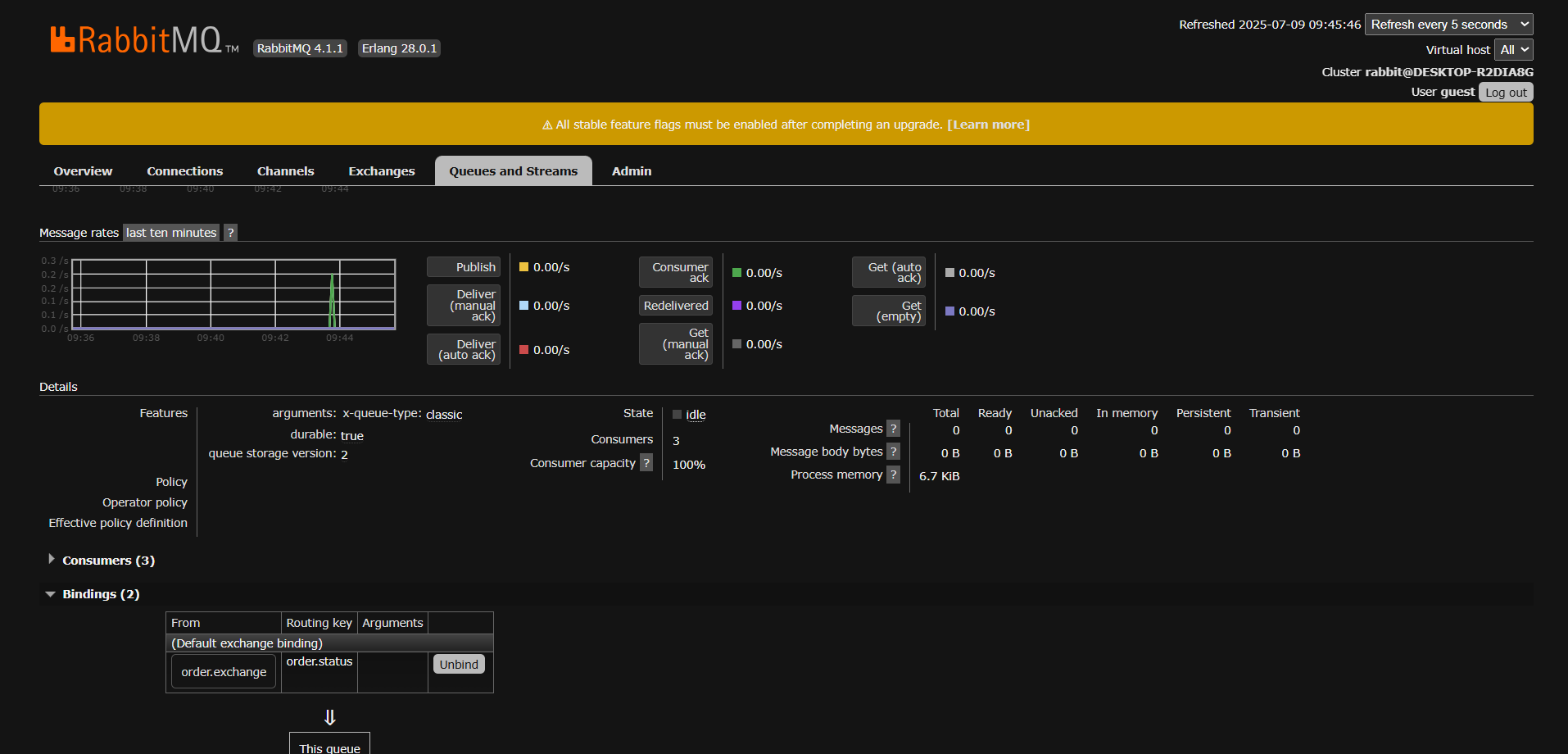
****

* Status changed in restaurant\_orders table to “ACCEPTED”.

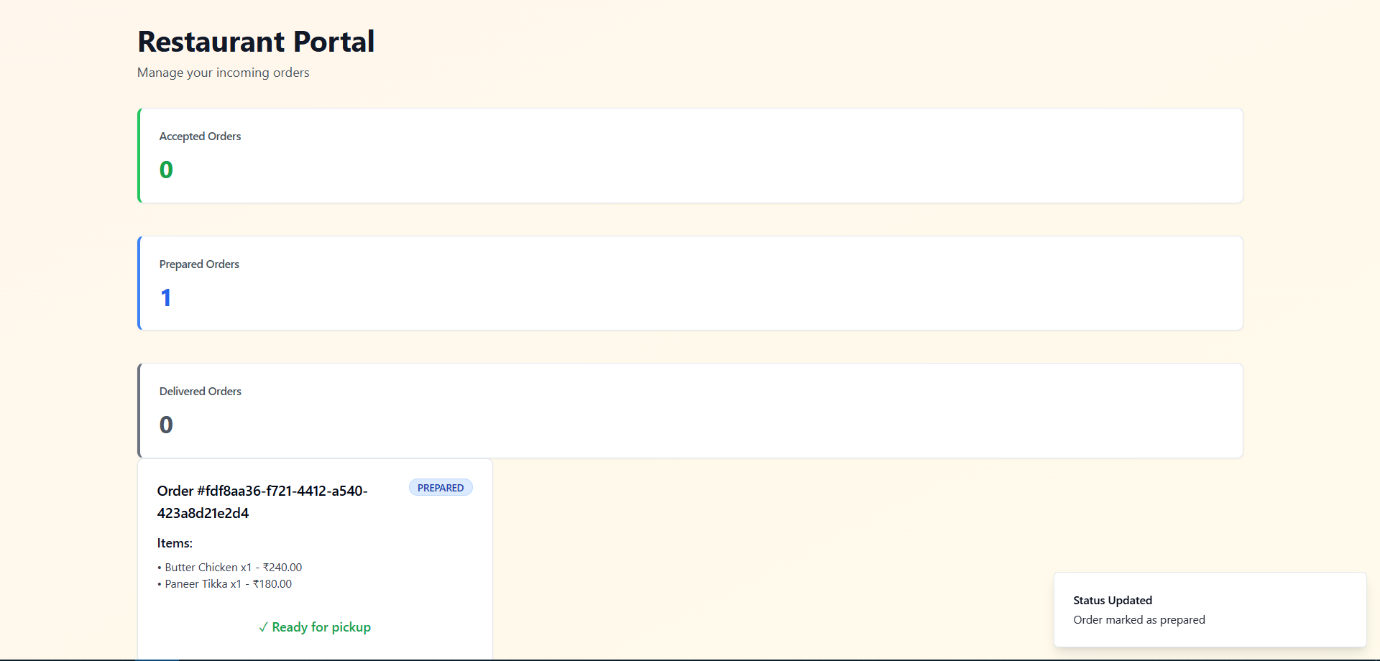
****

* RabbitMQ shows peaks when the messages are published or order “PLACED” and “ACCEPTED”.

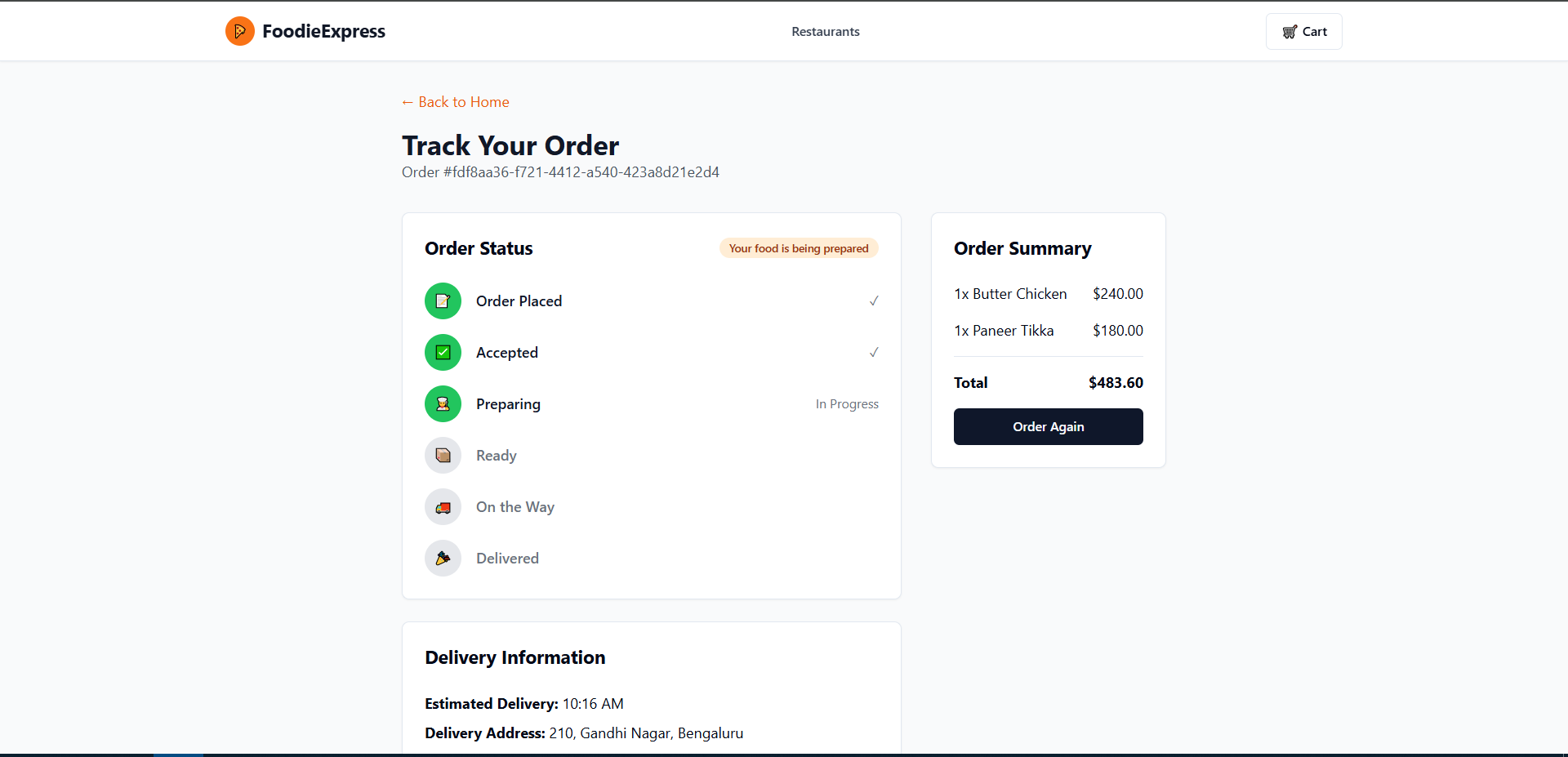
****

****

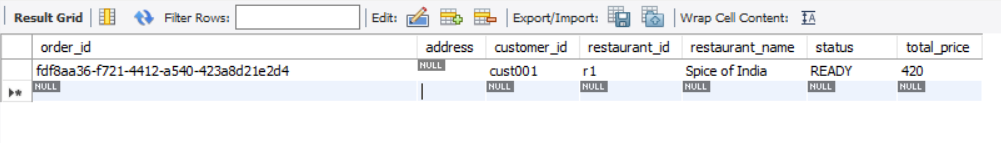
* Restaurant marked the order as prepared.

****

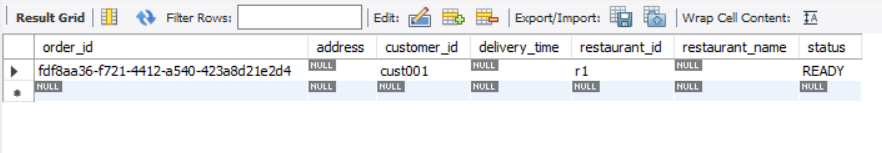
* Customer received the status update and updates the frontend.

****

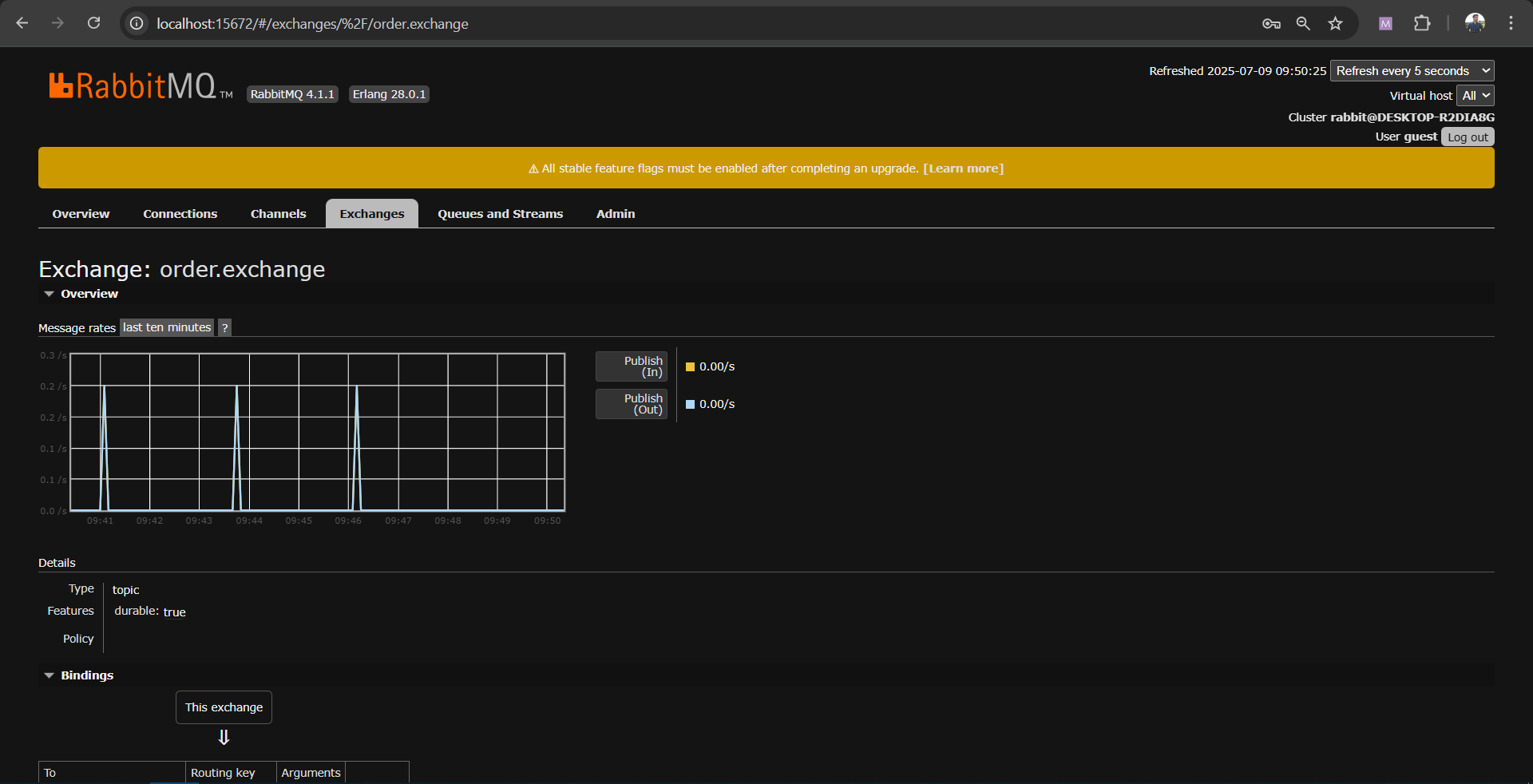
* Orders table updated the status to “READY”

****

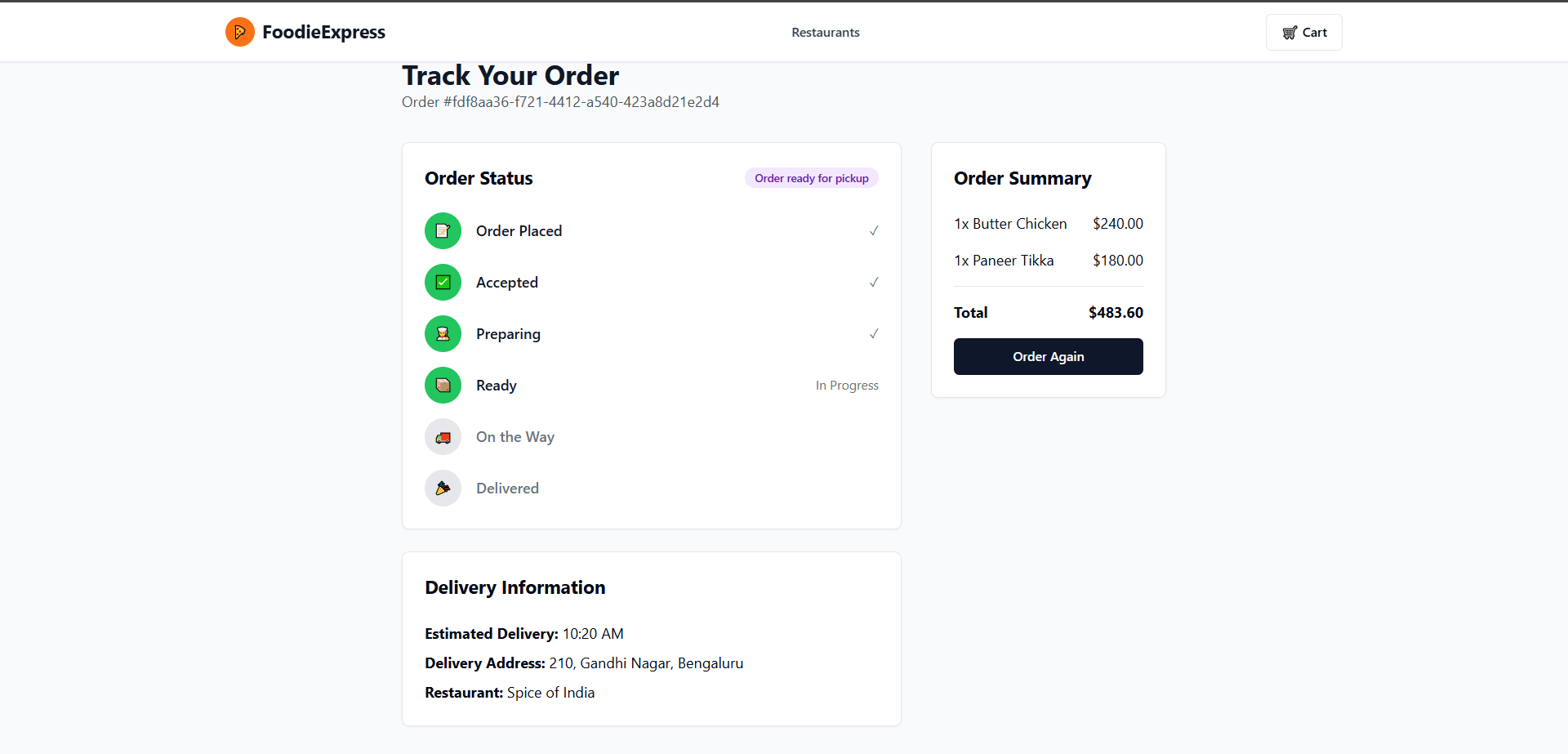
* Restaurant\_orders table updated the status to “READY”

****

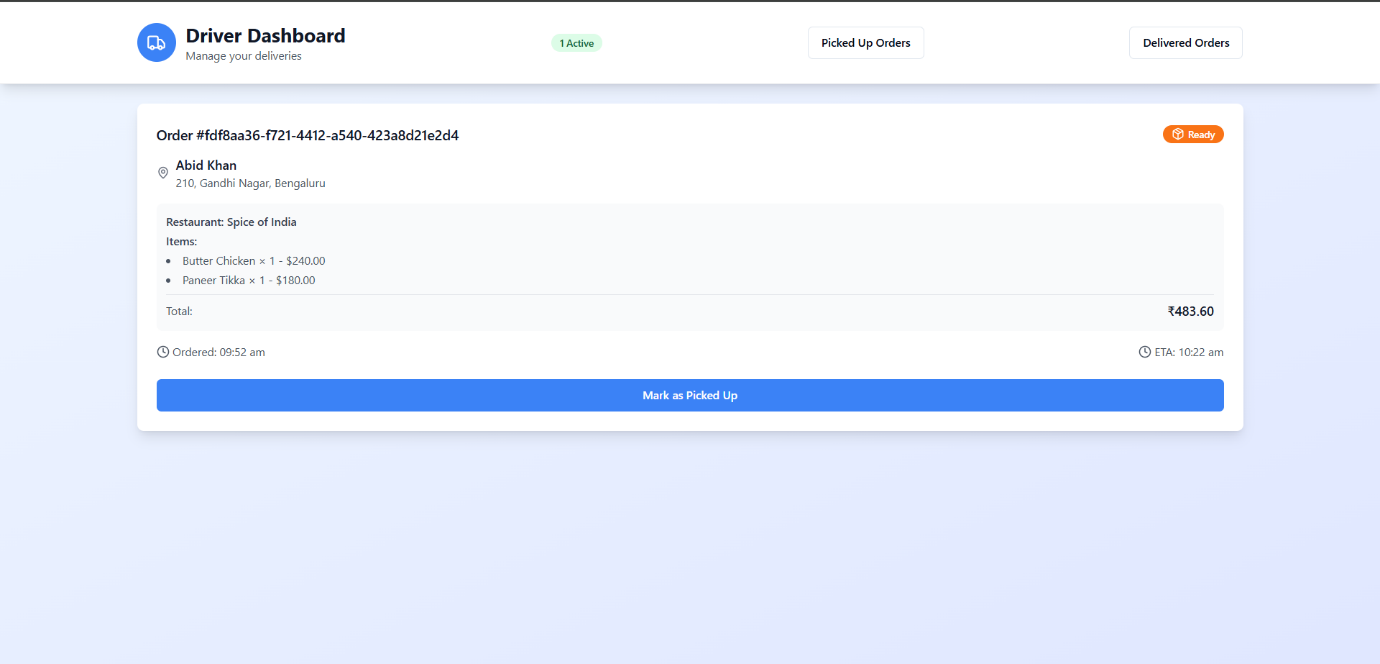
* RabbitMQ shows peak for order status updates.

****

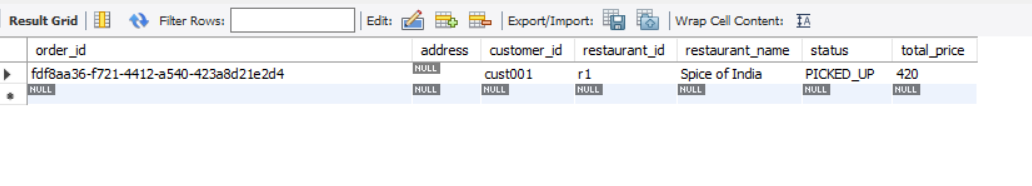
* Customer frontend updates to “READY” status.

****

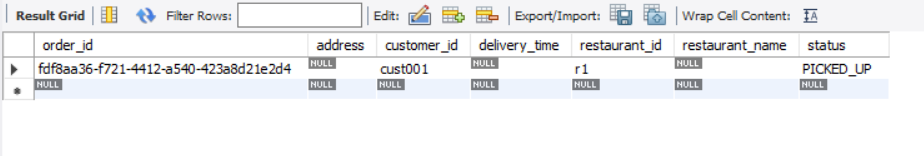
* Driver-service checks the status from the orders table and if the status is “READY”, the order is visible in the driver frontend.

****

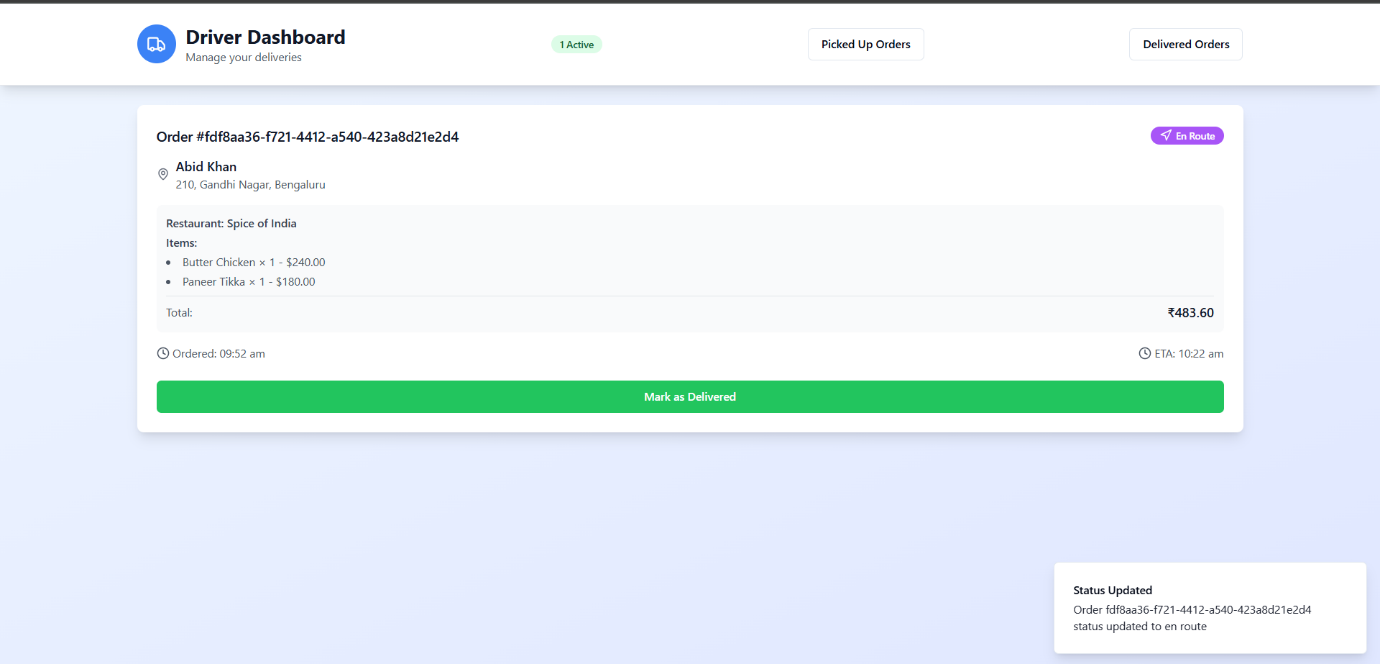
* Driver picks up the order, status changed in orders table as “PICKED\_UP”.

****

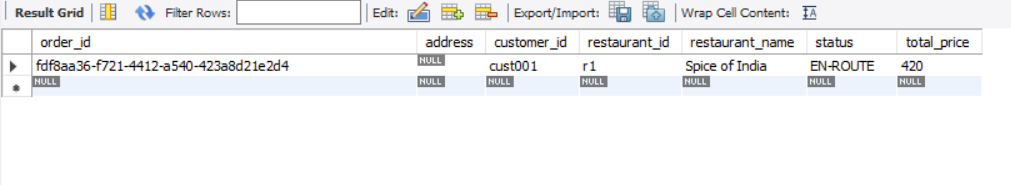
* Status also changes in restaurant\_orders as “PICKED\_UP”

****

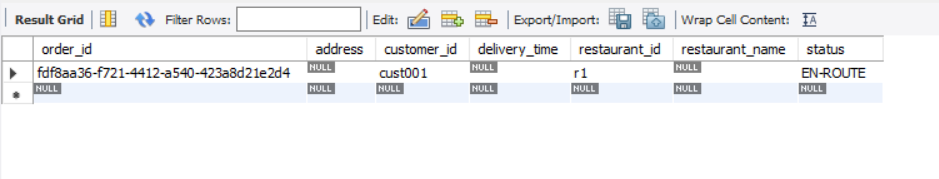
* Driver is on the way to deliver the order, marked order status as “EN-ROUTE”.

****

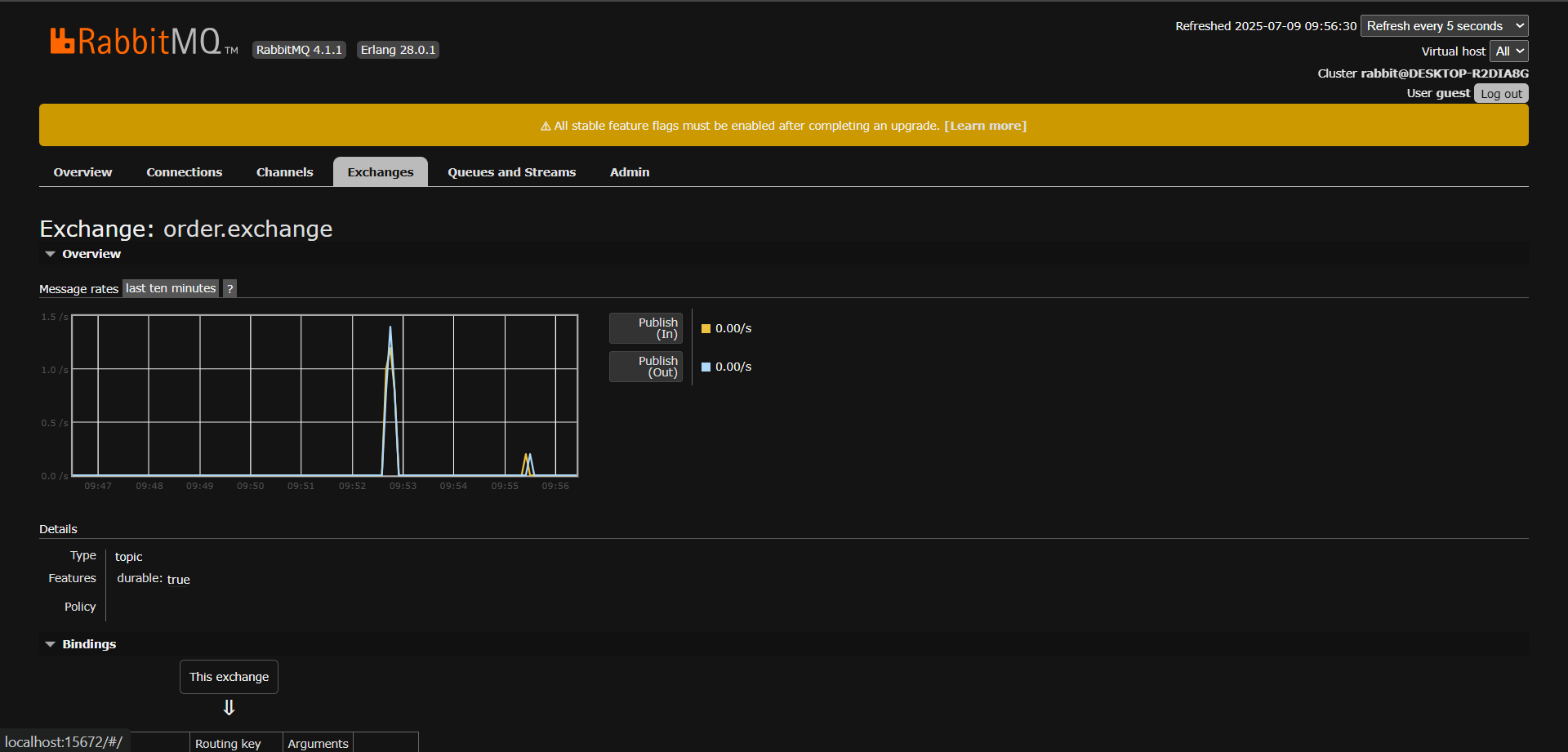
* Status changed in orders table as “EN-ROUTE”.

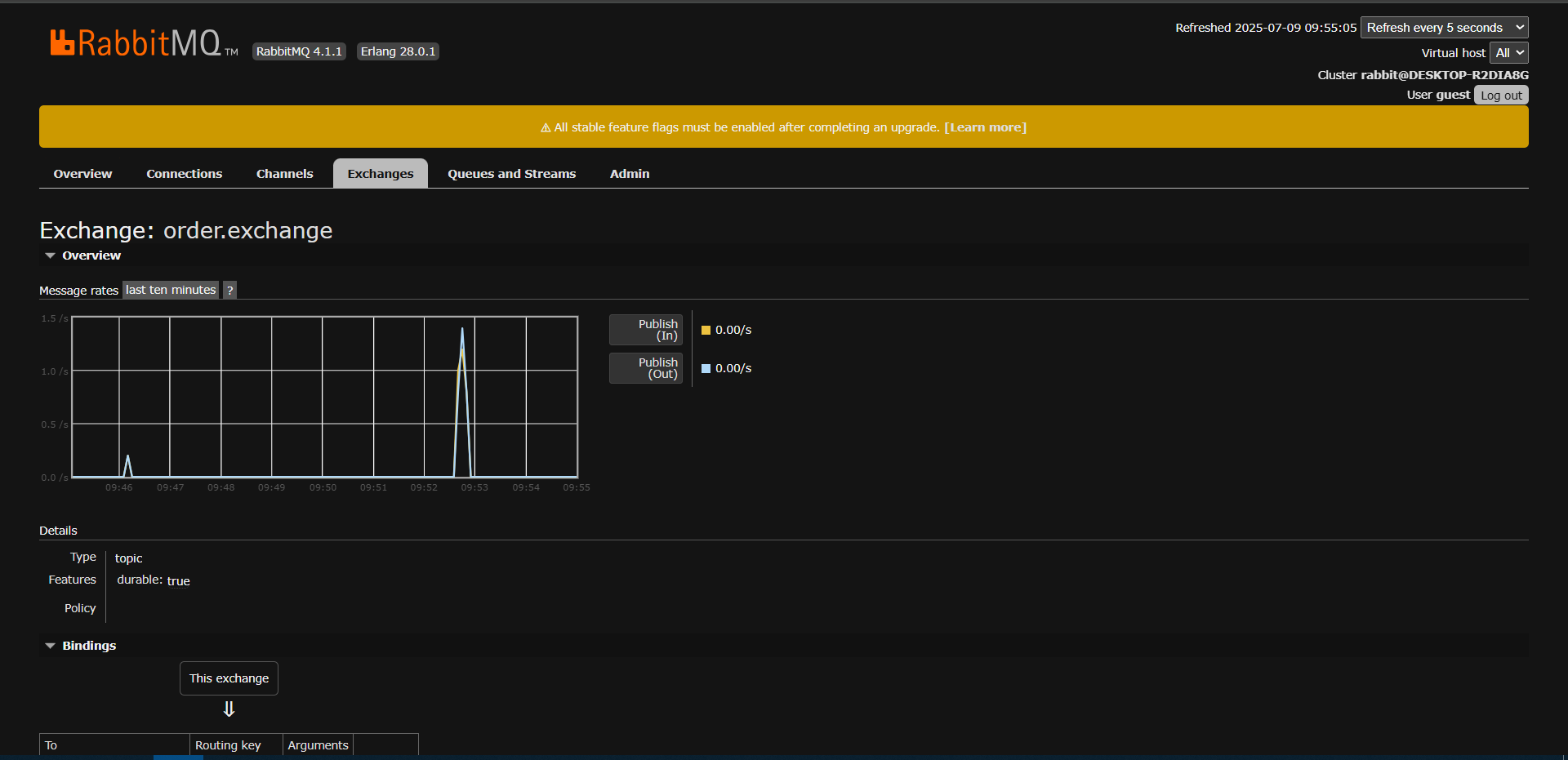
****

* Status changed in restaurant\_orders table as “EN-ROUTE”.

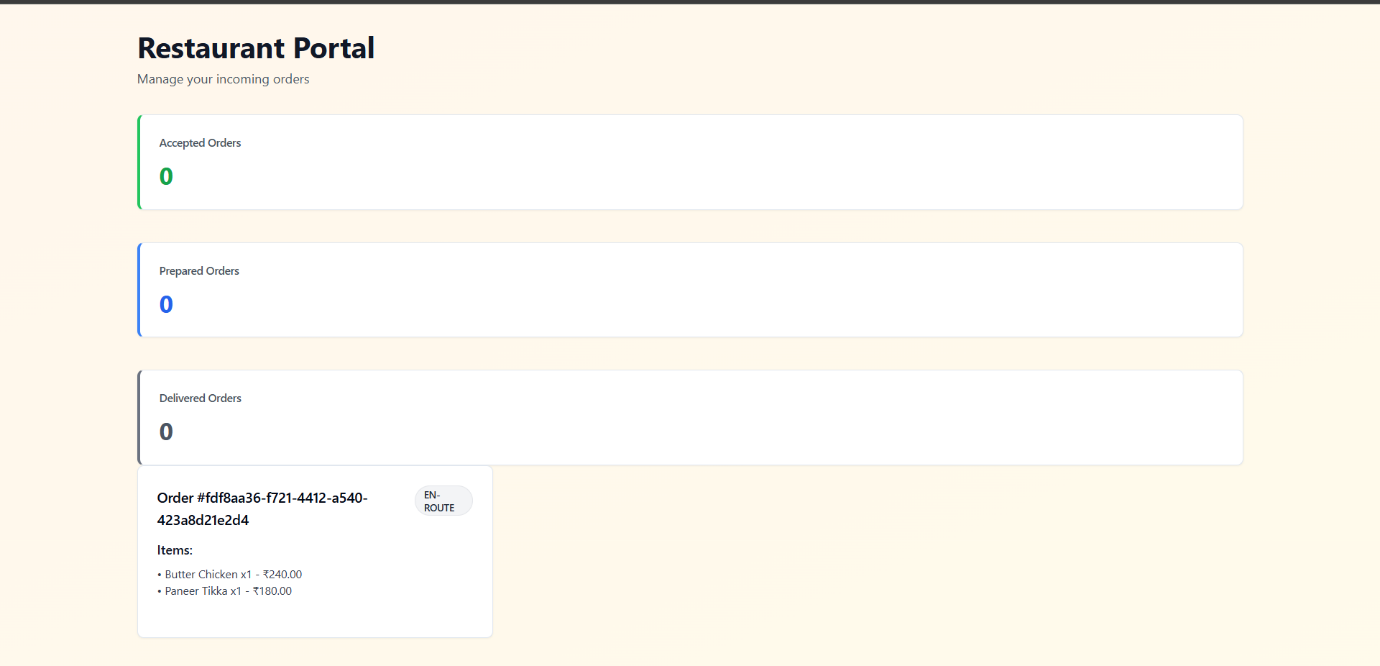
****

* RabbitMQ shows peaks for order status updates for “PICKED\_UP”, “EN-ROUTE” and publish out message for restaurant-service and customer-service to update the status.

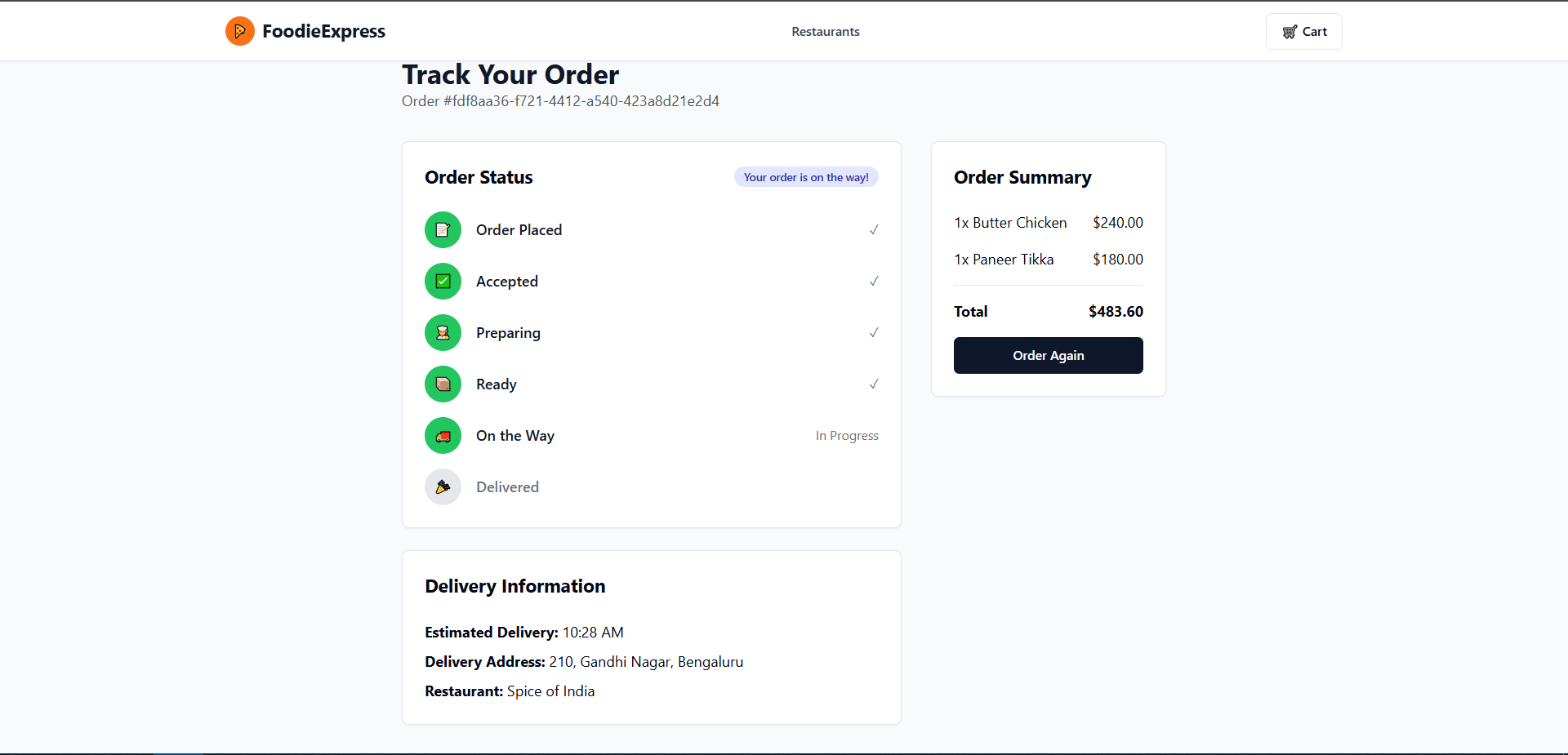
****

****

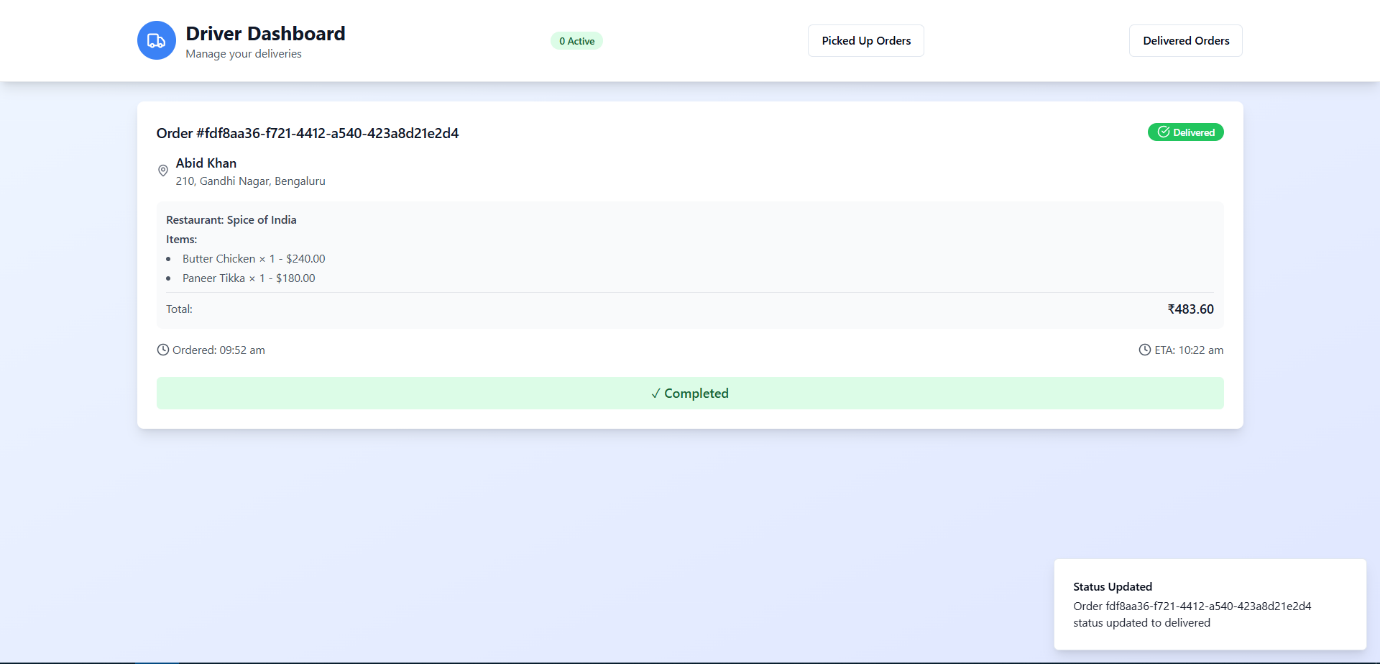
* Restaurant updates the frontend, and shows badge as “EN-ROUTE”.



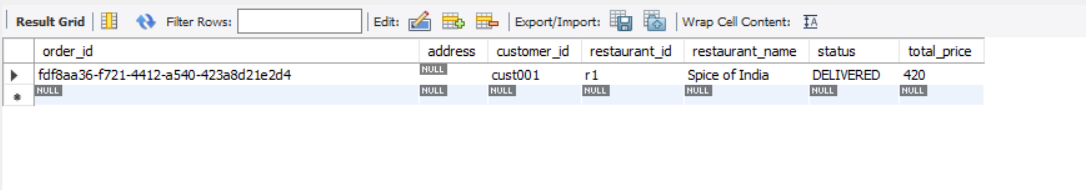
* Customer updates the track order to “On the way”



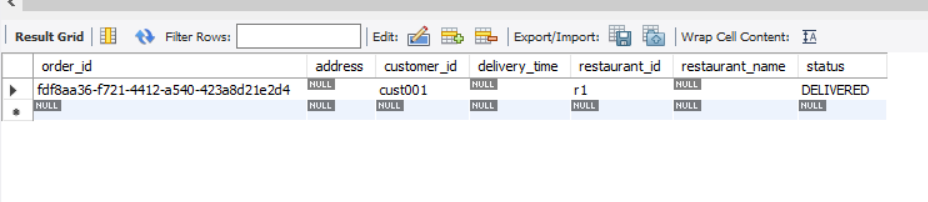
* Driver marked the order as delivered.



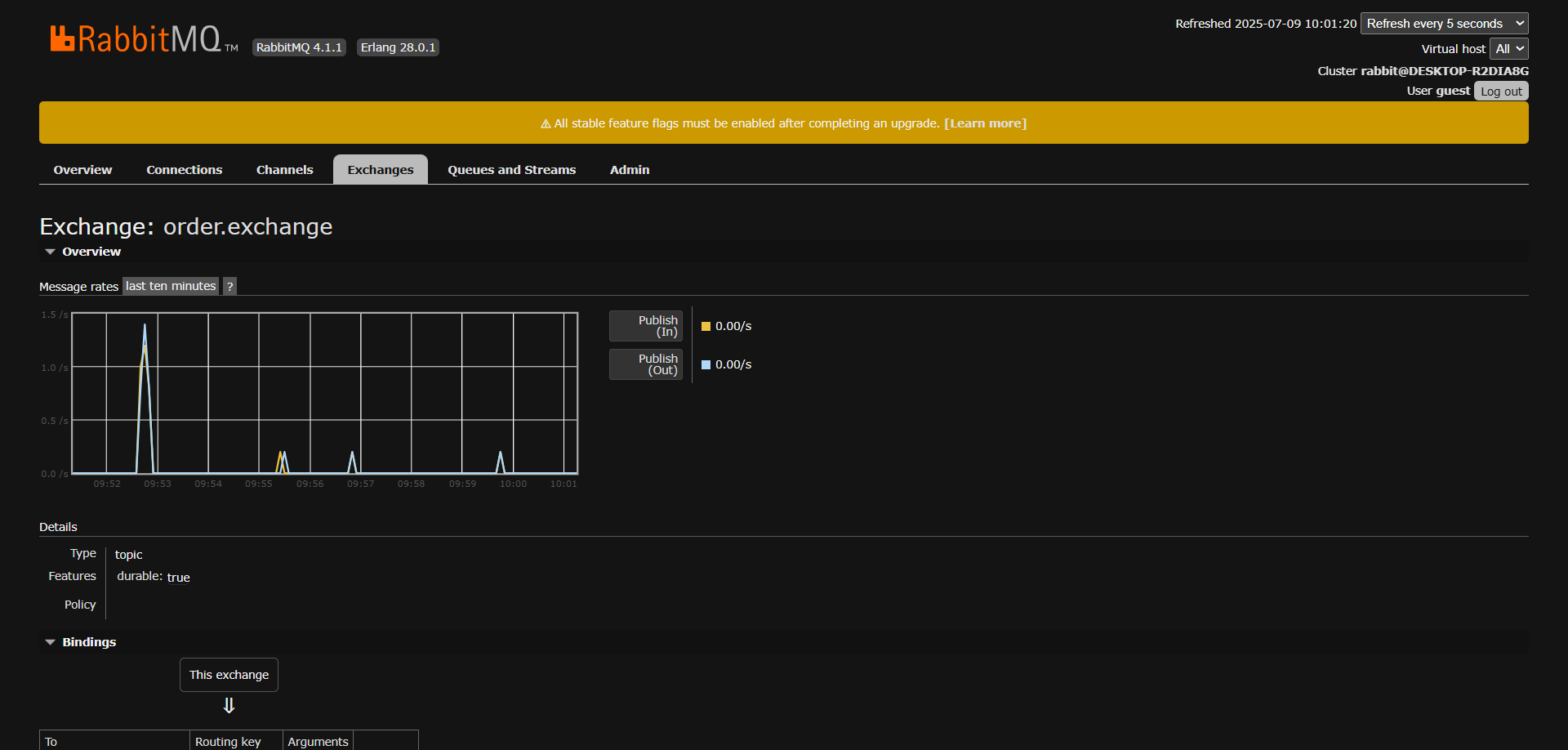
* Status updated in orders table as “DELIVERED”.



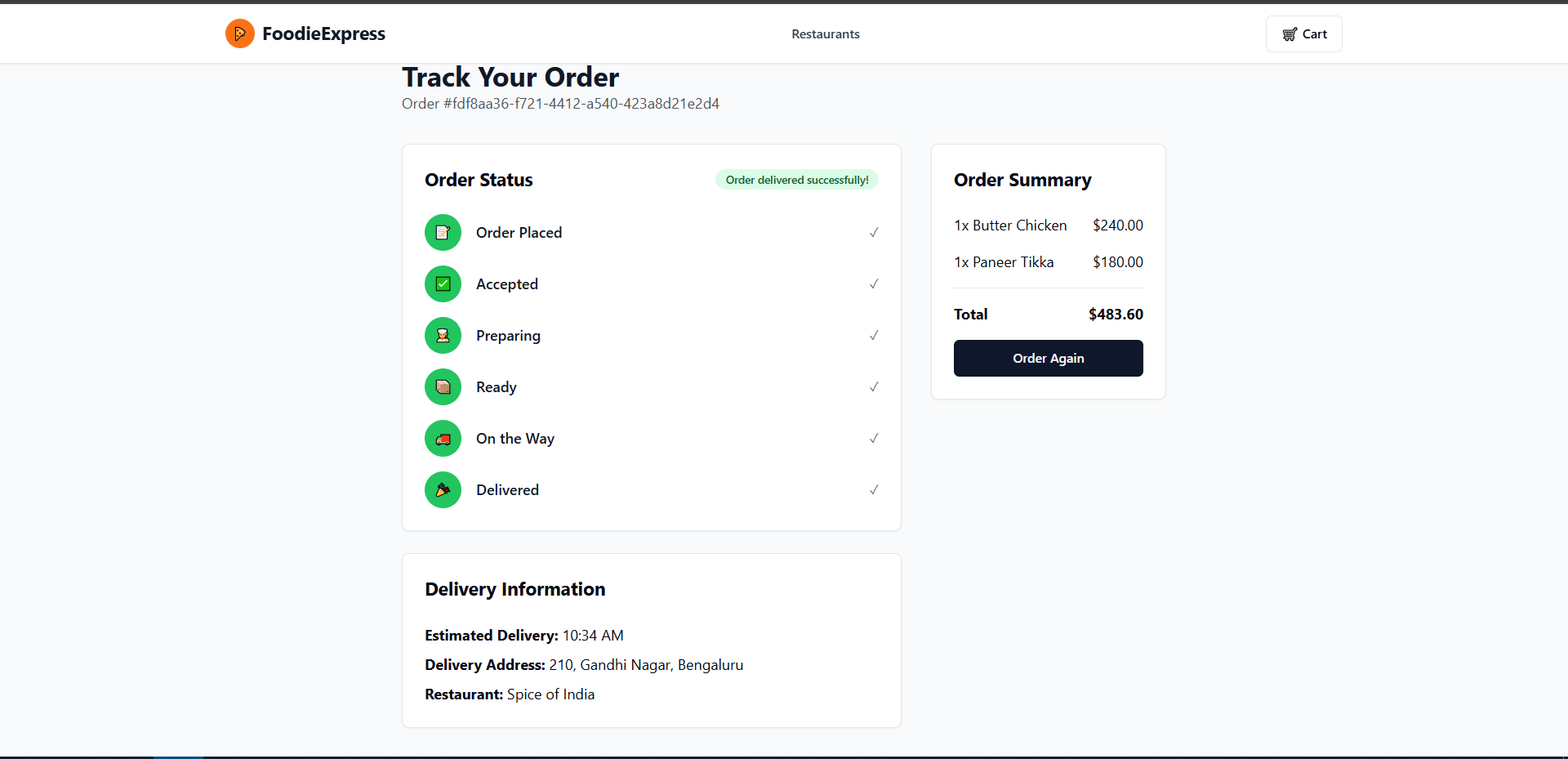
* Status updated in restaurant\_orders table as “DELIVERED”.



* RabbitMQ published out the status update as “DELIVERED”.



* Customer updates the order status as “DELIVERED”.



**Code Snippets**

**Customer-service**

* OrderController.java
  + REST endpoint in OrderController to place a new order, calculate pricing, and trigger downstream processing.

****

* + Endpoint to receive asynchronous status updates from the driver and notify the customer.

****

* + Endpoint to return minimal status info for lightweight order tracking.

****

* + Endpoint to retrieve full details of an order, including estimated delivery time for enhanced customer tracking.

****

* Order.java
  + This is JPA entity class Order defining the structure of customer orders and mapping to the relational database.



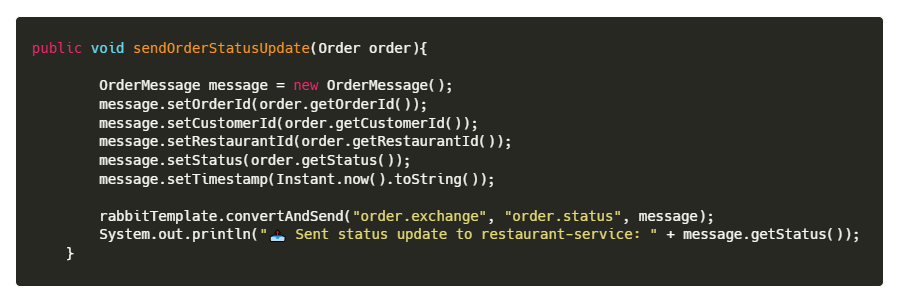
* OrderMessage.java (dto)
  + This is DTO class OrderMessage used for RabbitMQ communication between customer, restaurant, and driver services.



* MessageListener.java
  + Listener method in MessageListener.java that updates customer order status in response to messages from the middleware via RabbitMQ.



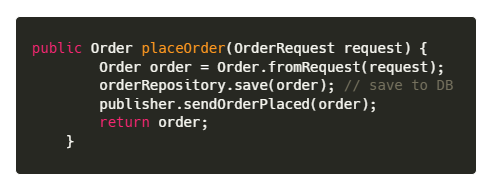
* RabbitMQPublisher.java
  + Publishes an order status update message to RabbitMQ so that downstream services (like restaurant or driver) receive the change in real time



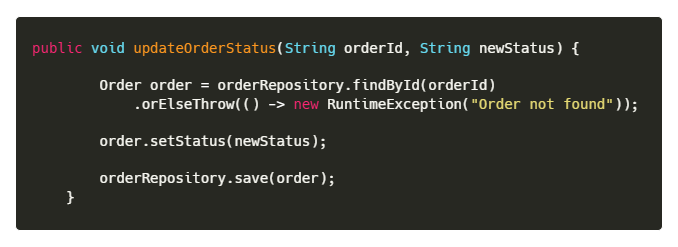
* Publishes a newly placed order to RabbitMQ so that the restaurant-service can receive and begin processing the order.



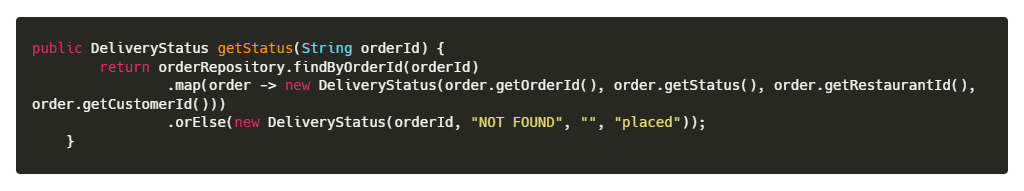
* OrderService.java
  + Handles order placement: saves order to DB and publishes it to RabbitMQ for restaurant processing.



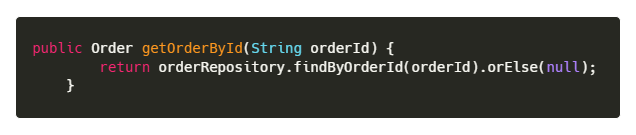
* + Updates the delivery status of an existing order and persists the changes in the database.



* + Retrieves order status for live tracking; provides fallback response if order is not found.



* + Fetches complete order information from the database based on a unique order ID.



**Restaurant-service**

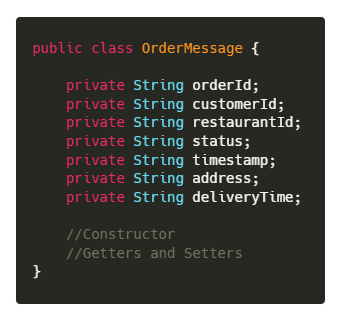
* OrderController.java
  + OrderController.java provides REST endpoints for restaurants to receive, track, and update order statuses. It integrates with RabbitMQ to notify other services in real-time.



* Order.java
  + Entity class Order defining the structure of restaurant-side orders and enabling persistent local processing using Spring JPA.



* OrderMessage.java(dto)
  + DTO class OrderMessage used for inter-service communication through RabbitMQ within the restaurant-service.



* MessageListener.java
  + Message listener that updates restaurant-side order status in response to real-time RabbitMQ events.



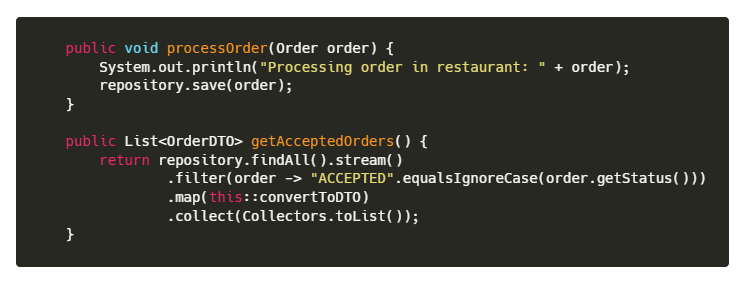
* RabbitMQPublisher.java
  + Publishes order status updates from restaurant-service to RabbitMQ for asynchronous communication with other services.



* OrderService.java
  + Updates the order status to "ACCEPTED", persists it, and publishes an order status update message to RabbitMQ for downstream services.



* + Processes and saves new orders locally; retrieves a list of orders currently marked as "ACCEPTED" for restaurant display.



* + Marks an order as accepted, saves the status update, and sends a notification message asynchronously to notify other services.



* + Changes the order status to "READY", persists the update, and publishes a corresponding status message for real-time updates.



* + Marks the order as "READY", updates the database, and sends a detailed RabbitMQ message containing order metadata for synchronization.

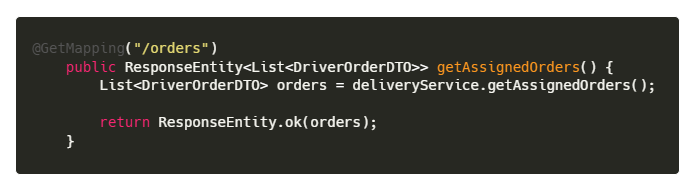


* + Sets the order status to "DELIVERED", persists the change, and publishes a delivery confirmation message to inform all relevant parties.



**Driver-service**

* DriverController.java
  + Handles HTTP GET requests to fetch all delivery orders assigned to the current driver, returning them as a list of DTOs for display in the driver app.



* + Accepts POST requests with delivery status updates (PICKED\_UP, EN\_ROUTE, DELIVERED) from the driver app, routing them to appropriate service methods and returning confirmation or error responses.



* Order.java
  + JPA entity Order storing delivery-relevant order information within the driver-service for managing assigned orders and statuses.



* OrderMessage.java (DTO)
  + DTO class OrderMessage for transporting delivery order status updates in the driver-service microservice.



* StatusUpdatePublisher.java
  + Publishes delivery status updates from driver-service to RabbitMQ to inform other services asynchronously.



* DriverService.java
  + Updates the order status to "PICKED\_UP" when the driver picks up the order. Saves the updated order in the database and sends a RabbitMQ message with the new status to notify other services like customer and restaurant.



* + Sets the order status to "EN\_ROUTE" when the driver is en route to deliver the order. The updated status is saved, and a message is sent via RabbitMQ to inform customer and restaurant services in real-time.

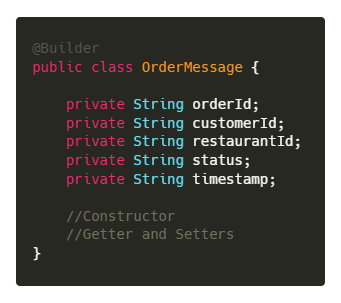


* + Marks the order as "DELIVERED" upon successful delivery by the driver. Persists the change and publishes a delivery confirmation message through RabbitMQ to synchronize all stakeholders.



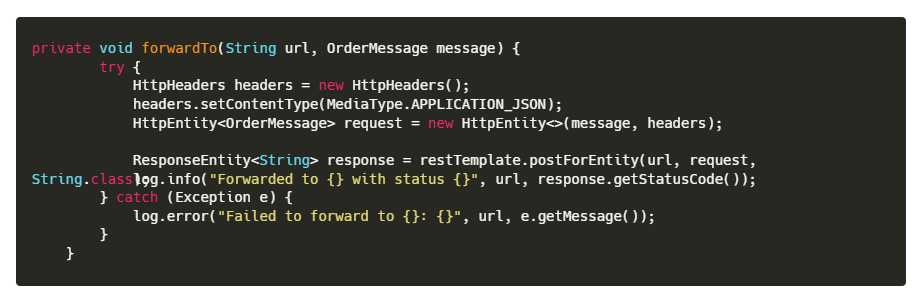
**Middleware-service**

* OrderMessage.java (DTO)
  + Middleware DTO class OrderMessage using Lombok builder for clean, structured message passing between services.

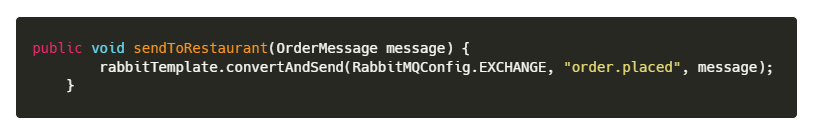


* MessageListener.java
  + Middleware service RabbitMQ listener for new orders that forwards messages to restaurant-service via HTTP POST.

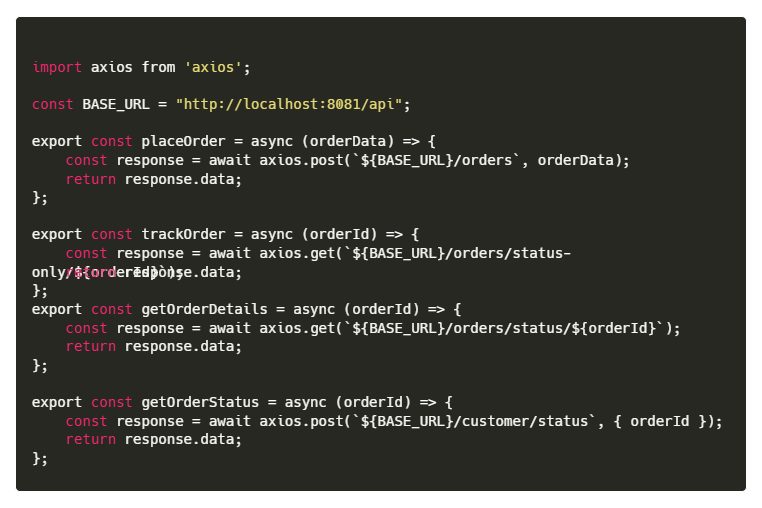




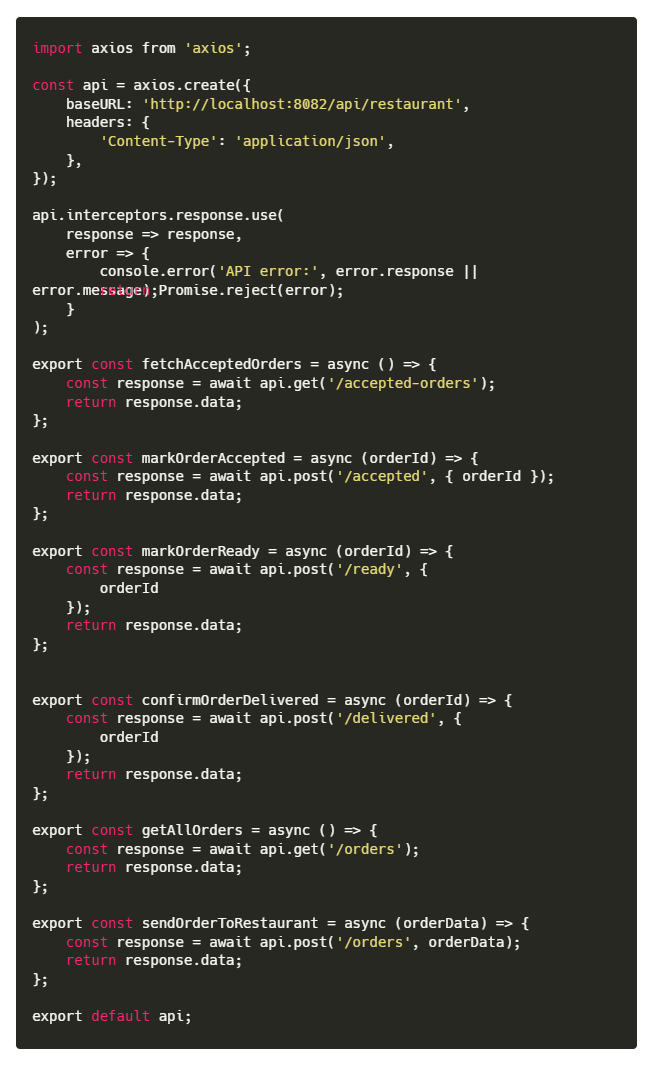
* RabbitMQPublisher.java
  + Publishes new order messages from customer-service to RabbitMQ exchange for restaurant processing.



* **Customer-frontend API code**
  + API utility functions in the customer frontend using Axios to interact with backend order services.

****

* **Restaurant-frontend API Code**
  + Restaurant frontend API utility module handling order status updates and retrieval via Axios with error interceptors.

****

* **Driver-frontend API Code**
  + Driver frontend API utilities managing assigned orders and delivery status updates using Axios.

****

**References**

* P. Ramesh and R. V. Kumar, Middleware: The Central Communication Layer for Distributed Applications. Smart and Emerging Technologies Journal (SET)
  + <https://setjournal.com/SET/article/view/19>
* K. S. Mahajan, B Mhetre, M. B. (2021). *Online Food Delivery System Using Microservices Architecture*. arXiv preprint arXiv:2108.03384.
  + <https://arxiv.org/abs/2108.03384>
* Gupta, S. (2020). *Scalable Microservice Architecture Using RabbitMQ RPC*. Medium.
  + <https://medium.com/swlh/scalable-microservice-architecture-using-rabbitmq-rpc-d07fa8faac32>
* Sharma, V., & Patil, S. (2022). *Event-Driven Architecture for Real-Time Order Delivery Platforms*. International Journal of Innovative Research in Computer Science & Technology, 10(1), 28–33.
  + <https://www.journals.acspublisher.com/index.php/ijircst/article/view/13380>
* Kamath, D. (2021). *Burgernaut: Online Food Ordering System (Microservices Architecture)*. GitHub.
  + <https://github.com/dhanushkamath/Burgernaut>
* Sharew, M. (2022). *Online Food Delivery Platform using Spring Boot and React*. GitHub.
  + <https://github.com/MahiSharew/onlineFoodDelivery>
* CloudAMQP. (2021). *How Relay Delivery Enhanced Food Delivery with RabbitMQ*. CloudAMQP Blog.
  + <https://www.cloudamqp.com/blog/how-relay-delivery-enhanced-food-delivery-with-rabbitmq.html>
* Oracle. (n.d.). *Java SE Documentation*. Oracle.
  + <https://docs.oracle.com/en/java/>
* Spring. (n.d.). *Spring Boot Documentation*. Spring.io.
  + <https://docs.spring.io/spring-boot/index.html>
* Apache Maven. (n.d.). *Maven Project Guides*. Apache Software Foundation.
  + <https://maven.apache.org/guides/index.html>
* Node.js Foundation. (n.d.). Node.js v20.0.0 Documentation.
  + <https://nodejs.org/docs/latest/api/>
* npm Documentation. (n.d.). *npm Docs – Getting Started*.
  + <https://docs.npmjs.com/>
* Mozilla Developer Network (MDN). (n.d.). *Getting Started with React*.
  + <https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries/React_getting_started>
* Vite. (n.d.). *Getting Started with Vite*
  + <https://vite.dev/guide/>
* Tailwind CSS. (n.d.). *Tailwind CSS with Vite*
  + <https://tailwindcss.com/docs/installation/using-vite>
* Oracle. (n.d.). *MySQL 8.0 Reference Manual*
  + <https://dev.mysql.com/doc/>
* Richardson, R. (n.d.). *RESTful Web Services – Introduction*. RESTfulAPI.net
  + <https://restfulapi.net/>
* Axios. (n.d.). *Axios Documentation – Introduction*.
  + <https://axios-http.com/docs/intro>
* Microsoft. (n.d.). *TypeScript Documentation*.
  + <https://www.typescriptlang.org/docs/>