Dine & Dash

Engineering and Service Management

João Monteiro 102690
João Sousa 103415
João Gaspar 107708
Venâncio Almeida 123008

Department of Electronics, Telecommunications and Informatics

University of Aveiro

List of Figures

Contents

Abstract

This report describes the academic project *Dine & Dash*, a delivery platform that allows users to place orders, process payments, and track deliveries in real time, benefiting from optimized routes for couriers. The architecture is based on a set of autonomous microservices communicating via HTTP/REST and using asynchronous queues (RabbitMQ), with MongoDB databases. It integrates the external OpenRouteService API for route calculation. The document presents an overall view of the architecture, employed technologies, communication flows, security mechanisms, and suggested future improvements. Note: The Payment Service is fully implemented.

Keywords: microservices, Kubernetes, RabbitMQ, MongoDB, JWT, OpenRouteService, service architecture.

1 Introduction

Dine & Dash is an prototype developed within the scope of the course *Engenharia e Gestão de Serviços*, whose main objective is to demonstrate the application of service-oriented (SOA) and microservice architectures in a delivery system scenario. The platform allows clients and couriers to interact independently, ensuring scalability and flexibility.

2 System Overview

Dine & Dash targets two main user profiles:

1. **Customer**: Places pickup and delivery orders, selects a payment method, and tracks the delivery status in real time.
2. **Courier**: Receives notifications of new orders, accepts them, obtains an optimized route to the customer, and updates location during delivery.

Key functionalities include:

- **Customer Registration and Authentication**: Uses JWT to generate access tokens.
- **Order Selection and Payment**: The customer specifies order details, confirms the request through a simulated payment service, and awaits validation.
- **Courier Acceptance**: The courier, can view pending orders and accept one.
- **Optimized Route Planning**: Once accepted, the Route Optimization Service calls the external OpenRouteService API to compute the most efficient path between courier and customer coordinates.
- **Real-Time Tracking**: A geolocation sensor (simulated) publishes coordinates via RabbitMQ to the Tracking Service, which provides continuous updates to both customer and courier.
- **Delivery Confirmation**: The courier marks the delivery as completed, registering the event in the system.

3 Microservice Architecture

The overall architecture consists of four core microservices, one *API Gateway*, and two frontends (for customers and couriers). The diagram below summarizes the structure:

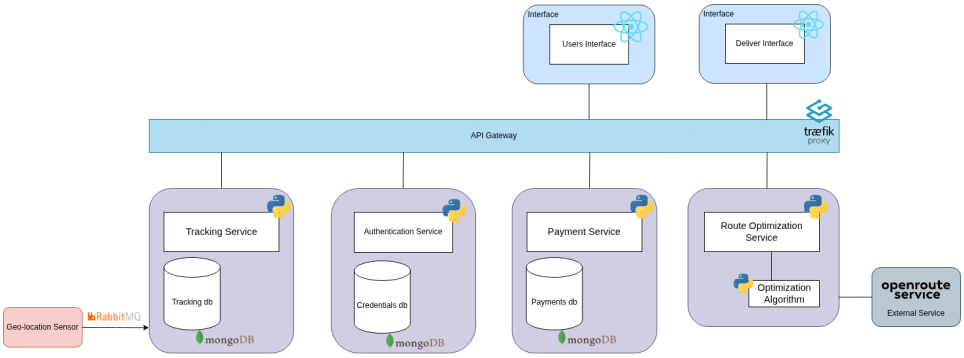

Figure 1: Microservice Architecture of "Dine & Dash"

3.1 Microservice Descriptions

Authentication Service Responsible for customer registration and login. Implemented in Python, FastAPI, it uses MongoDB (*Credentials DB*) to store credentials (email and password hashed with bcrypt via Passlib). It issues JWT tokens (HS256) without refresh tokens.

Payment Service Intended to manage payment transactions. Each payment record would contain `order_id`, `amount`, `method` (payment type), and `status` in the *Payments DB* (MongoDB). However, this service is not fully implemented and has design shortcomings: there is no real integration with external payment gateways, and it doesn't do nothing usefull.

Route Optimization Service Calculates routes between courier and customer. Calls the external OpenRouteService API via HTTP/REST. It returns a JSON object containing the optimized route.

Tracking Service Receives courier location data (via RabbitMQ) and persists history in the *Tracking DB* (MongoDB). It exposes HTTP endpoints for querying the current and historical location. It contains the order and delivery details on the database too.

3.2 API Gateway

The *API Gateway* serves as the single entry point for all external requests, exposing unified endpoints to the frontends and routing calls to each microservice.

- Simplifies access: both customer and courier call standardized REST endpoints, unaware of each service's internal port mapping.
- Acts as a passive routing layer; authentication via JWT is delegated to the *Authentication Service*.

- Facilitates deployment in Kubernetes by maintaining clear Ingress rules (service names defined as environment variables).

3.3 Databases (MongoDB)

Some microservices has its own MongoDB instance, with no direct data sharing:

- **Credentials DB (Authentication)**: Collection `users` with fields `id`, `username`, `email`, `hashed_password`, `created_at`, etc.
- **Payments DB (Payment)**: Collection `payments` containing `order_id`, `amount`, `method`, `status`, `timestamp`. Due to incomplete implementation, this service only stores and updates statuses manually.
- **Tracking DB (Tracking)**: Collection with the deliveries containing `tracking_id`, `order_id`, `customer_name`, `address`, `restaurant_address`, `delivery_date`, `status`. For the courier geolocation it have `latitude`, `longitude`, `city`, `country`, `ip`, `timestamp`.

This isolation respects the principle of *bounded context*, ensuring that each service only accesses its own data, thereby reducing tight coupling.

4 Communication and Integration

The architecture follows a hybrid approach for inter-service communication, combining synchronous HTTP and asynchronous messaging:

1. **Synchronous Communication via HTTP (REST):** Frontend interfaces interact with backend services through the *API Gateway* (Traefik), which routes HTTP requests to the appropriate microservices. These interactions are direct RESTful calls for operations such as authentication, tracking queries, or route requests. Examples include:
   - *Client → API Gateway → Authentication Service (POST /auth/login)*
   - *Client → API Gateway → Tracking Service (GET /tracking/deliveries)*
   - *Courier → API Gateway → Route Optimization Service (GET /routes?courier_ id=...)*

   Routing is statically defined in the Kubernetes `Ingress` resource. No service discovery mechanism (e.g. Consul, Eureka) is used; instead, service hostnames are configured manually through environment variables in deployment files.

2. **Asynchronous Communication via RabbitMQ:** A simulated or physical geolocation sensor publishes courier location updates to a RabbitMQ queue. The *Tracking Service* consumes these messages asynchronously, stores the data in MongoDB, and provides it via REST endpoints. Key implementation details:

    - Built using the `pika` Python library to interface with RabbitMQ.
    - Message acknowledgment is used to ensure that messages are only removed after successful processing.
    - The consumer can be horizontally scaled for higher throughput.

5 Payment Service

The *Payment Service* was initially planned to simulate the processing of transactions in a decoupled microservice. However, the implementation is incomplete and non-functional. The following aspects reflect its current state:

1. Although a route `POST /payments` is defined to receive basic payment data (`order_id`, `amount`, `method`), no request handling or validation logic is functional.

2. In fact, no documents are inserted into any MongoDB collection; the supposed `payments` database or schema is not integrated.

3. There is no status tracking mechanism — no "pending", "completed", or "failed" status transitions exist.

4. The frontend does not interface with the Payment Service in any capacity, nor does the service contribute to the flow of delivery or route allocation.

Consequently, the following payment-related concerns are entirely unaddressed:

- **No Gateway Simulation:** There is no mock integration with any external payment API (e.g., Stripe or PayPal), not even a dummy logic layer.
- **No Workflow:** Operations such as authorization, capture, refunds, or transaction validation are not present.
- **No Communication:** The Payment Service does not interact with any other service, frontend, or API Gateway.

At present, the Payment Service can be considered a placeholder without functionality. It was ultimately excluded from the system's active workflow.

6 Frontend Interfaces

Two separate React applications were developed, each built with Vite:

6.1 Users Interface

- **Features:**
  - Registration and login page (calls `/auth/login` and `/auth/register`).
  - New order form (collects origin, destination, and order details).
  - Real-time delivery status display with an interactive map showing the courier's current location retrieved from the *Tracking Service*.
  - Delivery confirmation and optional feedback form upon order completion.
- **Communication:** All HTTP requests are sent to the *API Gateway*, which routes them to the appropriate backend service.
- **Kubernetes ConfigMap:** Frontend environment variables are defined in the `.env` and injected using a `ConfigMap`. Examples include:
  - `VITE_AUTH_API = http://grupo8-egs.deti.ua.pt/auth`
  - `VITE_TRACKING_API = http://grupo8-egs.deti.ua.pt/tracking`
  - `VITE_ROUTE_API = http://grupo8-egs.deti.ua.pt/routes`

6.2 Courier Interface

- **Features:**
  - Displays a list of pending deliveries (via `GET /deliveries`).
  - Option to accept a delivery (`PATCH /deliveries/{id}` with status update).
  - Once accepted, it fetches the optimized route from the *Route Optimization Service* (`GET /route?courier_id=...`).
  - Shows the route and customer location on a map.
  - Publishes simulated GPS coordinates to RabbitMQ for real-time tracking.
  - Allows marking the delivery as completed.
- **Communication:** Uses the *API Gateway* for RESTful HTTP communication and RabbitMQ for location publishing.

7 Deployment and Infrastructure

The entire system was containerized using Docker and deployed to the university's Kubernetes cluster. The deployment involved several key steps and Kubernetes resources:

- **Containerization:**
  - All microservices were packaged as Docker images using `Dockerfile`s and `docker-compose`s.
- **Secrets and Configuration:**
  - `Secrets` store sensitive data like database URIs, JWT secret keys, and RabbitMQ credentials.
  - `ConfigMaps` define environment configuration (e.g., service URLs for frontends).
  - Both `Secrets` and `ConfigMaps` are mounted as environment variables in the deployments.
- **Deployments and Services:**
  - Each microservice (Authentication, Tracking, Route Optimization, Frontends) has its own `Deployment` and `ClusterIP Service`.
  - MongoDB is used in two instances (for Auth and Tracking), each backed by a `PersistentVolumeClaim`.
  - RabbitMQ was assumed to be externally available or simulated locally for Tracking Service integration.
- **Ingress Configuration:**
  - A `Traefik` Ingress Controller was used to route external traffic.
  - The domain `grupo8-egs.deti.ua.pt` was mapped to the cluster's public IP via DNS.
  - Ingress rules were defined to route specific paths:
    - `/auth` → Auth Service
    - `/tracking` → Tracking Service
    - `/routes` → Route Optimization
    - `/client` → User Frontend
    - `/courier` → Courier Frontend
- **Persistent Storage:**
  - MongoDB pods mount volumes using `PersistentVolumeClaim`s with `longhorn` storage class.

  - This ensures data is preserved across pod restarts.

- **Resource Management and Scaling:**
  - Each Deployment can be scaled by adjusting the `replicas` field.
  - Probes (readiness/liveness) were considered but not fully implemented in this version.

- **Limitations:**
  - Due to the lack of HTTPS support in the current deployment, several external APIs (e.g., Google Maps, Google Places) failed to load, as they require secure origins.
  - This restricted full integration of route visualization and map functionalities in the client interfaces.

8 Future Improvements

Based on the current state of the project, the following enhancements are recommended:

1. **Integrate a Real Payment Gateway:** Add integration with providers like Stripe or PayPal, including secure payment authorization, automatic capture, and refund workflows.

2. **Support Multiple Couriers:** Extend the backend logic to allow for multiple couriers to operate simultaneously, including load balancing of deliveries, route assignments, and real-time status tracking for each courier.

3. **Enhanced Courier Mapping:** Implement a real-time map interface that displays the full delivery route from courier to destination, with support for live position updates and estimated arrival times, using secure integration with Google Maps or OpenStreetMap.

9 Conclusion

This report presented a comprehensive overview of the project *Dine & Dash*, which adopts a microservice architecture to enable modularity, scalability, and independent service development. The selection of technologies — including Python, MongoDB, RabbitMQ, and Kubernetes — was guided by ease of use and alignment with the objectives.

Throughout the development, key challenges were encountered, particularly the lack of integration with a real payment gateway and with the

Google API. Nevertheless, the project successfully demonstrated the orchestration of multiple services, real-time communication via messaging queues, and containerized deployment on a distributed infrastructure.

This experience reinforced practical knowledge in service-based architecture, infrastructure provisioning, and the complexities of building resilient and maintainable systems in a Kubernetes-based environment. Future enhancements have been proposed to transform this project into a more complete and production-like system.