Brno University of Technology
Faculty of Information Technology

PDB

Night snack

System design and architecture

October 31, 2021

Svätopluk Hanzel

1 Project overview

The goal of this project is to implement a simple system that can act as a demo of a food delivery company specialized in late-night deliveries for programmers.

Since this is a demo application, the real-world usability is quite limited and this project mainly focuses on properly implementing CQRS pattern and using the correct DB systems for various types of data.

1.1 Application principle

Main principle of the application can be described as follows: there are restaurants, which have menu items organized into categories, which can be ordered by customers. Both customer and restaurant have a position in real-world-like map and the optimal delivery route is computed from these. Since the menu items' price, customer's location and delivery status must be freezed in each order, the orders are saved separately with all the data in a single object.

2 Data

As described in Application principle (1.1), the application works with `Restaurants`, `Customers`, `Addresses`, `Menu categories`, `Menu items`, `Orders`, and map `Nodes`. The first three of which can be saved in a single relational DB, the `Orders` would benefit from being saved as a single document in a NoSQL database. This is mainly because the ordered items must be saved in the order as they were at the time of ordering, thus creating a duplicates in the DB. Another benefit of saving non-normalized Order data in a NoSQL database is their support for quick and scalable analysis using MapReduce.

2.1 Relational data

The entities described above can be saved in a relational DB using a schema describe in 1. The missing `Order` entity is described in 2.2 and is used to bind restaurants and customers.

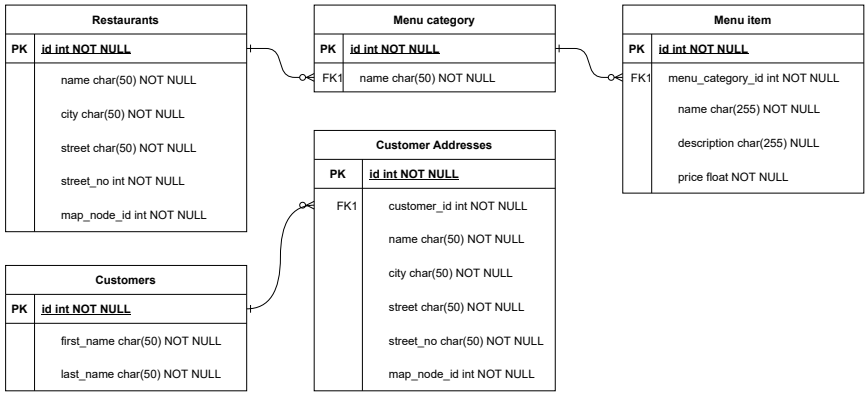

Figure 1: Relational data model: entity relationship diagram

2.2 Non-relational data

There are 2 parts to the non-relational data in this system: Map and Orders

Orders are saved in a document database and they usually have similar structure, which can be describe using the following schema bellow. Orders are the most commonly manipulated data in the entire system - they are used for keeping track of the orders, order history, and delivery status and as such must be always available and very responsive.

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Food delivery order",
  "description": "This document records the details of a food delivery order",
  "type": "object",
  "properties": {
    "id": {
      "description": "A unique identifier for an order",
      "type": "number"
    },
    "created_at": {
      "description": "Unix timestamp of order creation",
      "type": "number"
    },
    "restaurant": {
```

```
      "description": "Restaurant this was ordered from",
      "type": "object",
      "properties": {
        "id": {},
        "name": {},
        "map_node_id": {
          "description": "Restaurant location map node
             ↪ ID",
          "type": "number"
        }
      }
    },
    "customer_id": {
      "description": "ID of the customer in the
         ↪ relational DB",
      "type": "number"
    },
    "menu_items": {
      "description": "Ordered menu items",
      "type": "array",
      "items": {
        "description": "Menu item",
        "type": "object",
        "properties" {
          "id": {
            "description": "ID of the menu item",
            "type": "number"
          },
          "name": {
            "description": "Name of the menu item",
            "type": "string"
          },
          "price": {
            "description": "Price of the menu item at
               ↪ the time of the order",
            "type": "number"
          },
        }
      }
    },
    "delivery": {
      "status": {
        "description": "Status of this order",
        "type": "string",
      },
      "courier_position": {
        "map_node_id": {
          "description": "ID of a map node in neo4j",
          "type": "number"
```

```
        }
      },
      "destination": {
        "description": "Object describing the delivery
            ↪ location",
        "type": "object",
        "properties": {
          "address": {
            "description": "Formatted address to be
                ↪ displayed in order detail",
            "type": "string"
          },
          "map_node_id": {
            "description": "ID of a map node in neo4j",
            "type": "number"
          }
        }
      }
    }
  }
}
```

Map The map model is designed to be saved in a graph DB and closely copies the model of OpenStreetMap XML in the sense, that all junctions are graph nodes and the connections between them are graph edges. The map is more or less stable, will only be altered sporadically, but the readability of these data is crucial for navigation and delivery, making it very important part of the system. The shortest path queries also shouldn't take too long, but a latency of few seconds is acceptable.

3 System architecture

The application will be written in Rust using the Warp framework. The system will not have any frontend UI and will accept commands and queries using REST API.

3.1 DB systems

The chosen DB systems reflect the data types and needs describe in 2. For relational data I've chosen to use the CockroachDB as it is a scalable SQL database with good support and PostgreSQL syntax resulting in very good interoperability.

The map data will be stored in neo4j, which is a graph DB allowing us to use some of the shortest path algorithms in its library and the Orders data

will be stored in MongoDB which is very fast, scalable document NoSQL DB.

3.2 Operations and CQRS

The system will use REST API and will implement CQRS using separate endpoints and handlers (either command or query) for every operation.

Basic CRUD (create, read, update, delete) operations will be created for `Restaurants`, `Customers`, `Addresses`, `Menu categories`, `Menu items`, `Orders` entities. Furthermore there will be operations to calculate the best route for delivery, and find restaurants near customer. For each of these operations a new structure called either `<op>Command` or `<op>Query` will be created. This structure will then cover the whole operation execution resulting in a clear architecture.

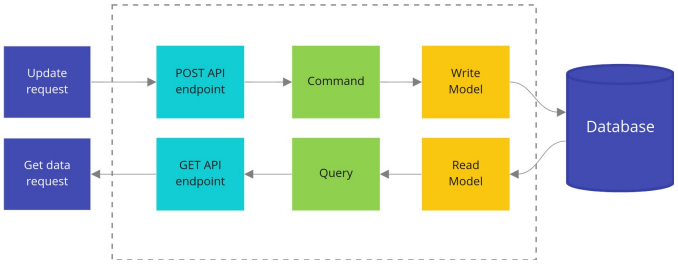

Figure 2: CQRS architecture (source)