ascatu

Arbeitsmedizin I Arbeitssicherheit I Arbeitspsychologie

ascatu GmbH

Programming Challenge

REST API For CRUD Operations On Orders

# A Microservice For Persistency

Prepared By

Emre Aygener

## Approach

Upon receiving the programming challenge my first steps were scanning the document for twice to estimate the time and roadmap necessary to implement the tasks. After a brief scanning, I started studying each task for the necessities and the requirements. I found 1st and the 4th tasks rather simple and easy to complete. The 5th task seemed more time consuming then the 1st and 4th but other than that it was easy from the difficulty perspective. The two tasks left were about the implementation of the project and required most of the attention at first sight.

After reviewing the docs a few times more, I decided to checkout if the tech stack in my knowledge would do the job or would I compromise a lot just to feel the comfort of the known. After searching on google, scanning the documentations and watching some YouTube videos, I decided, moving on with .NET framework won't impact the performance or capabilities with a significance. The Confluent Kafka library in .Net was managed by the team that also released Apache Kafka in the first place. Even though the technology was written with Java thus more compatible with it, .Net seemed to have great support and community that was active on the topic.

Now that I decided on the framework, I needed to decide on the database that I will use with the application. My initial instinct was using MS SQL Server as it was the one that I had most experience. But it would be harder to use with docker and I started venturing my other options. PostgreSQL and MySQL were other strong options that needed me to drop the Entity Framework which would make the process a little more inconvenient. I have opted out MongoDB for the same reason. While searching my options I came across SQLite which was fully supported with Entity Framework and was a directory-based database (it creates an order.db in the project directory and work as a SQL database). Because of the scope of the programming challenge, I have decided on continuing with SQLite.

The Microservices Architecture was already decided, so I didn't have to think about it. Next step was the structural decisions. Since the release of .Net 6 the minimal APIs were gaining more popularity with each update. It brings simplicity and less overhead (less boilerplate codes and less configurations) to the table. Therefore, I have been looking for a chance to build a project using minimal APIs. Due to the small scale of this project, I thought it would be a great match. Dependency Injection was also aligning well with this approach (keeping simplicity in mind) and some design patterns like repository and CQRS was only going to over complicate the project.

That was the thought process that I had starting the project.

# General Decisions

## Framework / Language

### .Net 8 / C#

- Framework and language that I am most fluent with.
- No significant negative impact.
- ( For choosing .NET8 ) Latest stable build even though out of support.

## Database

### SQLite

- Light and easy to configure.
- Directory based approach seemed logical.
- Supports Entity Framework.

## Object Relational Mapping (ORM)

### Entity Framework

- Most I am fluent with.
- Supports extensive LINQ queries.
- Easy to write & read.

## Libraries

### Confluent.Kafka

- Maintained by same team that maintain Apache Kafka
- Top-choice among other developers.

### CloudNative.CloudEvents

- Compliance with CloudEvents 1.0 specs.
- CloudEvents C# SDK page refers to this library.

### Newtonsoft.Json

- Industry standard until the release of .NET9 (will be included inside System.Text.Json)

### Swashbuckle/OpenApi AspNetCore Libraries

- Provides Swagger UI with a few lines of code.

# Endpoints

## GetOrders (GET, localhost:8081/api/v1/order)

- **Description**: Sends a GET request with no additional parameters.
- **Returns**: All the order data from the database.
- **Negatives**:
    - A pagination feature would improve usability and performance in a real-world use case.
    - The endpoint is reachable for everyone. An authorization and authentication structure should be implemented to limit access for both unauthenticated and unauthorized clients.

## GetOrderById (GET, localhost:8081/api/v1/order/{id})

- **Description**: Sends a GET request with the order `id` as a path parameter.
- **Returns**: The order data from the database if the order is present.
- **Negatives**:
    - A pagination feature would improve usability and performance in a real-world use case.
    - The endpoint is reachable for everyone. An authorization and authentication structure should be implemented to limit access for both unauthenticated and unauthorized clients.
    - Prone to SQL Injection.

## CreateOrder ( POST , localhost:8081/api/v1/order )

- **Description**: Sends a POST request with order data in the request body sends a GET request for the received person id from body to verify if person exists. If person id is valid saves the order to the database.
- **Returns**: The created order from database.
- **Negatives**:
    - Endpoint allows any client to create orders without any form of authentication or authorization, which can lead to unauthorized access and manipulation of data.
    - No rate limiting to prevent abuse by repeated order creation.
    - Prone to SQL Injection.

## UpdateOrder (PUT, localhost:8081/api/v1/order/{id})

- **Description**: Sends a PUT request with the order `id` as a path parameter and updated order data in the request body. Checks the person id provided if it is valid, saves the order modification to database.
- **Returns**: The updated order data.
- **Negatives**:
    - Endpoint allows any client to update orders without any form of authentication or authorization, which can lead to unauthorized access and manipulation of data.
    - No rate limiting to prevent abuse by repeated order updates.
    - Prone to SQL Injection.

### DeleteOrder (DELETE, localhost:8081/api/v1/order/{id})

- **Description**: Sends a DELETE request with the order `id` as a path parameter.
- **Returns**: No content if the order is deleted successfully.
- **Negatives**:
  - Endpoint allows any client to delete orders without any form of authentication or authorization, which can lead to unauthorized access and manipulation of data.
  - No soft delete functionality to allow recovery of deleted orders if needed.
  - No rate limiting to prevent abuse by repeated order deletions.

### DeleteAllOrders (DELETE, localhost:8081/api/v1/order)

- **Description**: Sends a DELETE request to delete all orders in the database.
- **Returns**: No content if all orders are deleted successfully.
- **Negatives**:
  - Endpoint allows any client to delete all orders without any form of authentication or authorization, which can lead to unauthorized access and manipulation of data.
  - No soft delete functionality to allow recovery of deleted orders if needed.
  - Extremely dangerous operation without any form of confirmation or protection against accidental or malicious use.
  - No rate limiting to prevent abuse by repeated deletion of all orders.
  - Prone to SQL Injection.

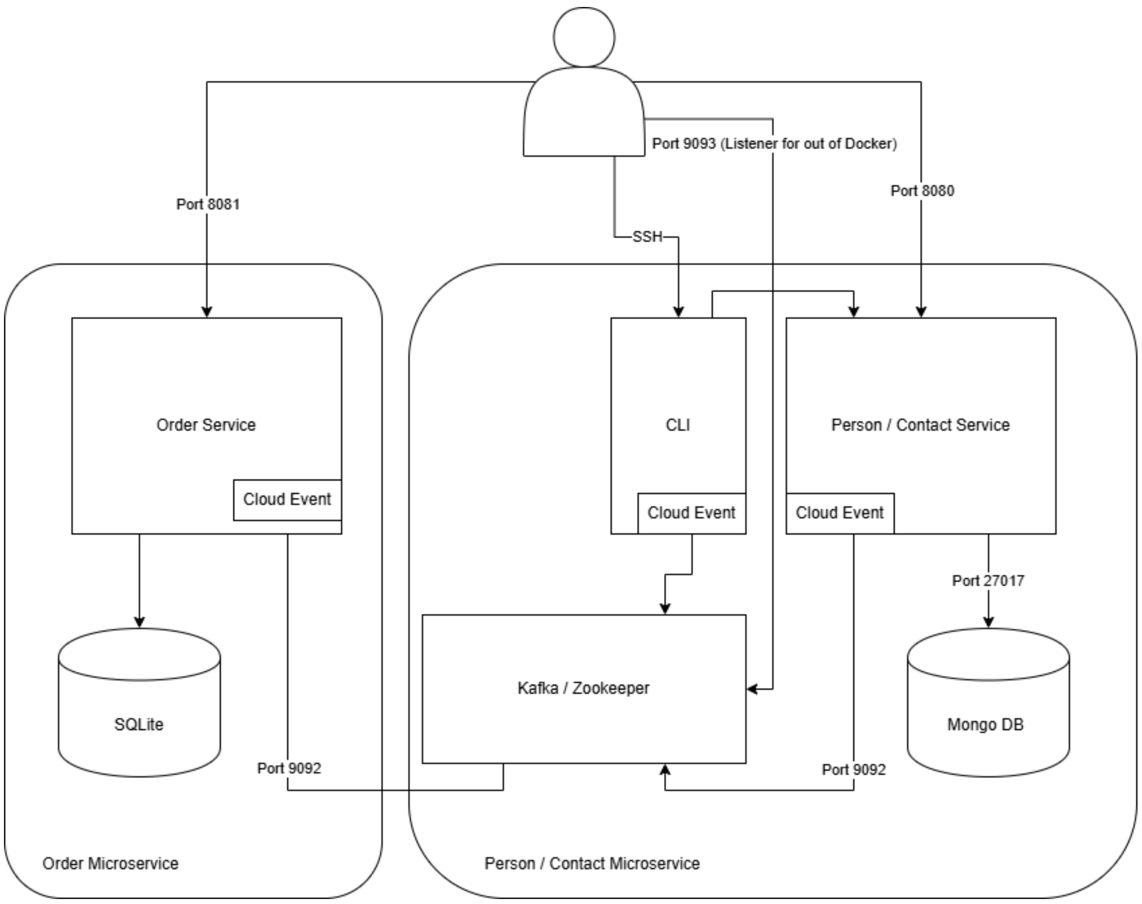
## Exception Handling

A custom middleware was added to the pipeline to catch any thrown exception and log the exception to the console. Provides a simple solution instead of using try, catch blocks everywhere. Does not prevent the system from crushing if there was a fatal error.

### Proposed Improvements

- More detailed Order object would reflect more real-life scenarios more efficiently.
  - For example, if there was an address field or if there were any fields related to the person that is placing the order like his personal information, subscribing to Kafka server would make much more sense because the modified field could also be related with the order (like address or customer name) and any changes could also be reflected to the order service as well. The subscription was only made to showcase the ability. Implementation details would enhance the experience.
- A JWT Bearer or OAuth 2.0 Token implementation could be done for authentication then could be used for authorization.
- Some code repetition could be prevented with more utility methods.

## Architecture Schema



*Thanks for giving me the opportunity.  This was a pleasant task to work on. Hope to hear from you soon.*