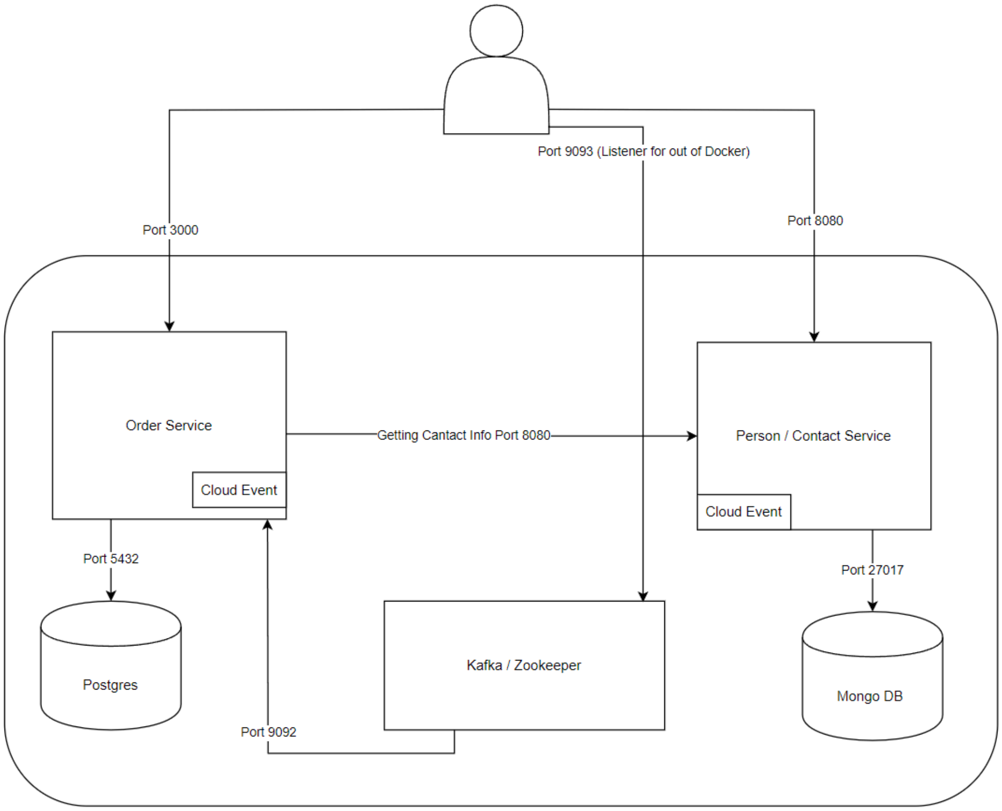
Order Service with Kafka Integration

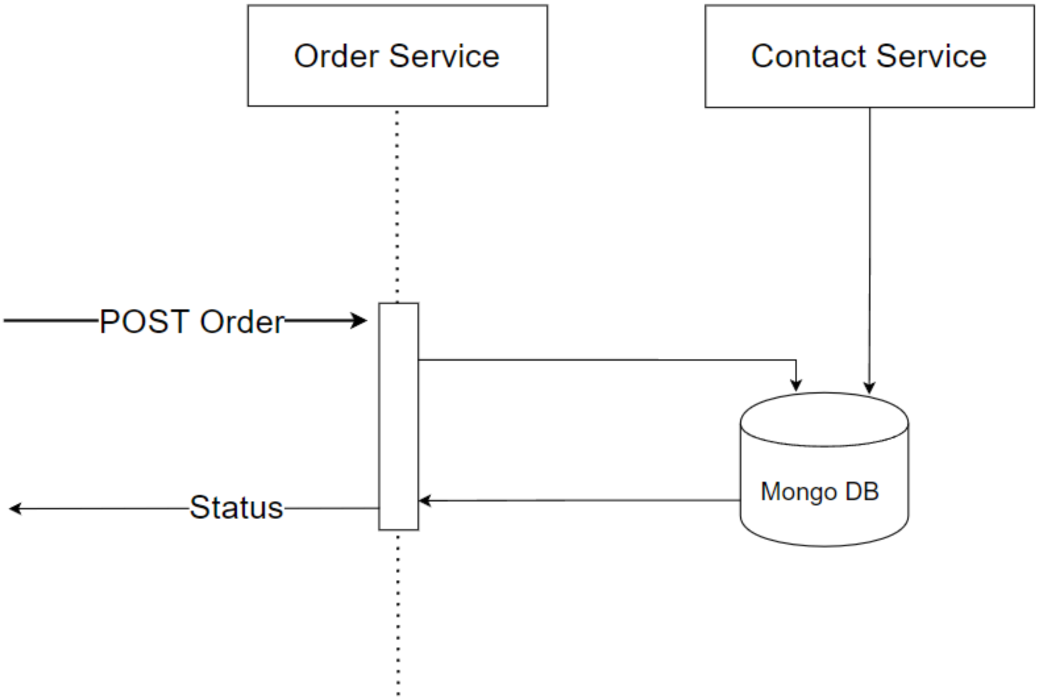
Introduction

This project involves creating an Order Service using Node.js and Express.js, integrating it with Kafka for event-driven communication, and ensuring synchronization with the Contact Service. The primary goal is to handle order operations (create, read, update, delete) while maintaining consistency with contact information through Kafka events.



Project Walkthrough

Initially, there was some confusion regarding Task 3, where the idea of creating the Order Service separate from the Contact Service and using a different database. I was thinking of creating the order service on the same contact service and using the same database, this way there will be no need for listening for kafka contact events. However I found out that using a separate database will make order service independent from the contact service. This approach ensures that if the Contact Service encounters any issues, the Order Service can still operate. Additionally, the Contact Service could contain more data than just names and addresses, making linking person data to orders more complex. Using microservices in this scenario allows for independence and prevents a single point of failure.



Technologies Used Discussion

- **NodeJS and ExpressJS:** I already had experience using NodeJS with ExpressJS, so I wanted to use it for the backend architecture and focus more on the kafka communication and error handling. NodeJs was chosen for their non-blocking I/O architecture, which allows for handling concurrent requests effectively, making it a good choice for building the Order Service. Express.js provides a streamlined framework for building RESTful APIs quickly.
- **Using Java and spring boot:** As the contact service was already built in java, it will be faster and better to integrate the order service on top of the contact service.
- **Rust:** If I had time to learn it, I would have used it to develop the order service. Rust's combination of performance, reliability, and safety makes it a strong contender for building order service.

DataBase Discussion

- **MongoDB vs SQL:** As I said in the beginning, I started with using the same database and it was very easy to finish the order service and sync it with the person data. However to create a separate microservice, a different database was used. MongoDB is a non-relational document database, and the order service relies on relation between tables, so creating an SQL database will be convenient for the service. I used PostgreSQL and added it to the docker-compose file to start the project.

- **Prisma:** I like Prisma, It saves a lot of time for me when working with databases. One big benefit of using ORM like Prisma is that you can migrate to another database easily without rewriting the code. Like moving from SQL to MongoDB.

Model/Router/Controller structure

This architecture allows for clean and maintainable code. Each folder has a single responsibility, resulting in better code design.

- **Routes Folder:** Defines routes in a simple form, delegating functionality to controller functions.
- **Controller Folder:** Contains the logic for handling routes, including user input validation, error handling, and model calls.
- **Model Folder:** Handles database CRUD operations, with errors propagated to the controller and then to the user with appropriate status codes.
- **Kafka Folder:** Kafka Folder: Contains Kafka configuration code, including producer and consumer configurations.
- **Prisma Folder:** Contains Prisma schema and configuration.

User Input Validation

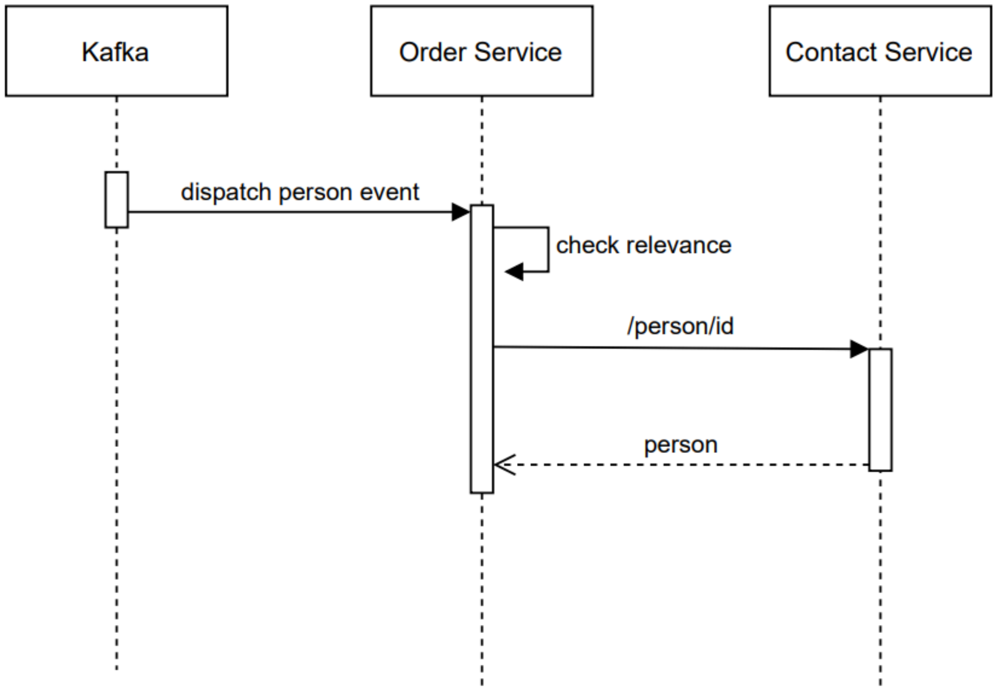
For creating and updating orders, user input should be validated before entering the database. Validations include checking for valid dates, positive order values, and ensuring soldToID corresponds to an actual user. These validations are crucial for detecting errors and informing the user. For example, the PUT request to update an order includes input validation to prevent errors and ensure any missing order details result in an error response.

Kafka Configuration

- **Publishing Order Events:** When creating, updating, or deleting an order, an event is sent to Kafka with details such as order ID, item IDs, timestamp, and source. The entire order details are not sent, as they can be retrieved from the database using the ID.
  Example:
  ```
  {
    "specversion": "1.0",
    "id": "42cdff2b-f7cc-4380-b67a-213d6b2aa274",
    "source": "http://localhost:3000/api/v1/order",
    "type": "order.created",
    "time": "2024-08-04T12:17:35.281Z",
  ```

```
  "datacontenttype": "application/json",
  "data": {
    "orderId": "f0a46ae8-5769-4de4-9678-770dfbad990a",
    "itemIds": ["c172fc92-073f-49fc-89fb-ea4a332505d8"]
  }
}
```

- **Consuming Contact Events:** By subscribing to *personevents-changed* and *personevents-deleted*, the Order Service can update or delete user data in the PostgreSQL database accordingly. When a person changes, the service checks if that person exists in the database, fetches the updated information from the Contact Service, and saves the changes. If a person is deleted, all related orders and order items are also deleted.



Docker Compose

- **Postgres Image:** Created with a username, password, and database called "orders", listening on port 5432.
- **Order Service:** The Dockerfile includes all dependencies, and the service listens on port 3000. It depends on the Postgres database for data storage, the Contact Service for fetching information, and Kafka for publishing and consuming events.

Postman Testing

After each code change:

- Tested every API endpoint for expected values.
- Tested user input validation and error messages.
- Verified if user data is updated in the Order Service after changes in the Contact Service.
- Checked if orders are deleted when a person is deleted in the Contact Service.

Improvements should be made to the service

- **TypeScript:** Using TypeScript instead of JavaScript for better type checking and error handling.
- **NestJs Framework instead of ExpressJS:** Nestjs is designed to provide a structured and modular approach, making it easier to build and maintain large-scale applications
- **Automated Tests:** Creating automated tests for each API endpoint to ensure they work correctly. This includes unit tests, integration tests, and end-to-end tests to verify the functionality of the service.
- **Monitoring and Logging:** Implementing monitoring and logging tools to track the performance and health of the service.
- **Scalability**: Enhancing the service to handle increased load by implementing load balancing and auto-scaling. Using tools like Kubernetes can help in managing containerized applications and scaling them based on demand.
- **Caching:** Implementing caching mechanisms to improve the performance of read operations. Using tools like Redis or Memcached can help in reducing the load on the database by storing frequently accessed data in memory.
- **Advanced Error Handling:** Improving error handling to provide more descriptive error messages and handling edge cases. This helps in diagnosing issues more effectively and provides a better user experience.
- **Security:** Adding middleware for security, such as JWT for authorization.
- **Soft Delete:** Implementing soft delete for orders to retain deleted data for business analytics.
- **Person Event Deletion:** If a person was deleted from the contact service, an update to the order database should be made. As deleting a person might cause deleting several orders, marking the user as deleted and keeping his orders up could be a better option.

Security

- **JWT Refresh Token:** Adding JWT refresh tokens to enhance security. This ensures that even if a user's access token is compromised, the attacker cannot maintain prolonged access without the refresh token.

- **Role-Based Access Control (RBAC):** Implementing role-based access control to restrict access to certain endpoints based on the user's role. This ensures that only authorized users with the appropriate permissions can access sensitive operations.

- **Input Validation and Sanitization**: Implementing thorough input validation and sanitization to prevent injection attacks, such as SQL injection or NoSQL injection. This ensures that the data entering the system is clean and safe to process.

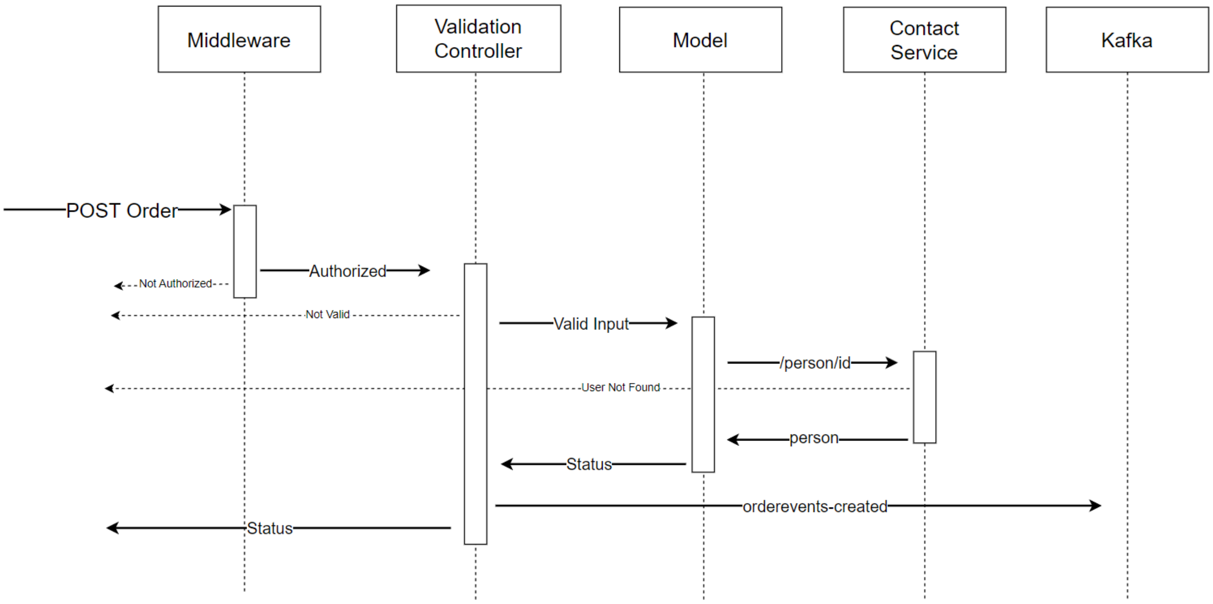
- **Rate Limiting and Throttling:** Implementing rate limiting and throttling to prevent denial-of-service (DoS) attacks. This ensures that the service can handle legitimate traffic without being overwhelmed by excessive requests.

- **Environment Variables Management**: Storing sensitive configuration details like database credentials, API keys, and other secrets in environment variables instead of hardcoding them into the codebase. Using tools like Docker secrets or AWS Secrets Manager can help in managing these securely.

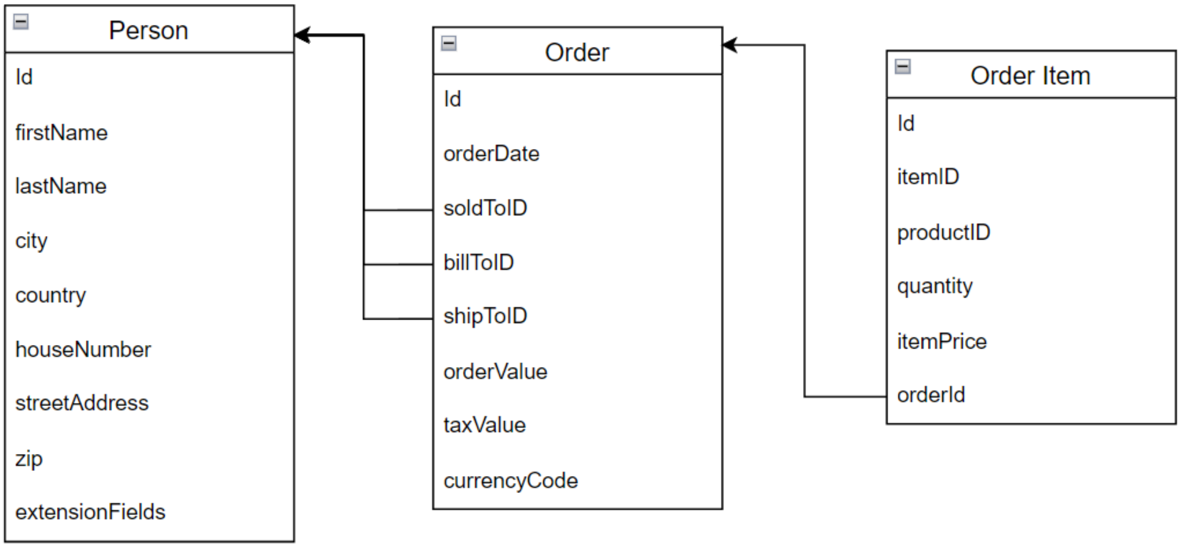
- **HTTPS**: Ensuring that all communications between clients and the server are encrypted using HTTPS. This protects data from being intercepted or tampered with during transmission.

By implementing these security measures, the Order Service can be safeguarded against various security threats, ensuring the integrity, confidentiality, and availability of the data and services provided

Order Post Request with JWT



Database Diagram


Person
Id
firstName
lastName
city
country
houseNumber
streetAddress
zip
extensionFields
Order
Id
orderDate
soldToID
billToID
shipToID
orderValue
taxValue
currencyCode
Order Item
Id
itemID
productID
quantity
itemPrice
orderId