Design Document: Event Manager

Maximilian Keller; 5123018
David Heppenheimer; 5123026
Sebastian Nagles; 5123035
Max Tremel; 513117

15. Februar 2025

Inhaltsverzeichnis

1 Introduction

The event management system is based on hexagonal architecture. The system handles core functionalities like managing events, users, ticket purchases, and payments. For example a user can buy a ticket and therefore a purchase will be created. The persistence layer utilizes a PostgreSQL database hosted in a Docker container, providing a robust and scalable solution for data storage. The system is accessed via GraphQL, allowing efficient querying and mutation of data. During development, we used Postman for testing and validating the system's functionality.

2 GraphQL as API

GraphQL was chosen for this project due to its flexibility in querying data, allowing clients to request only the necessary fields, which reduces overfetching and improves performance. This is particularly beneficial for handling complex relationships, such as events, users, and tickets, where fetching nested data structures in a single request enhances efficiency.

The main advantage of GraphQL in this project is its ability to provide a more dynamic and client-driven API compared to REST, making it easier to retrieve specific data without multiple endpoints. Furthermore, strong typing ensures better validation and API documentation.

However, GraphQL also has some trade-offs. Its complexity increases on the server side, requiring resolvers to manage queries efficiently. Caching is more challenging compared to REST, as queries can be highly dynamic. Furthermore, performance overhead can arise from the resolution of deeply nested queries, potentially impacting response times.

3 Architecture Overview

3.1 Hexagonal Architecture

The hexagonal architecture in this project ensures a **clear separation of concerns**, making the system modular, maintainable and adaptable. The **domain layer** contains the core business logic, independent of external technologies. The **application layer** facilitates the interaction between the domain and external components, such as the **persistence layer**, which manages the data mapping for PostgreSQL. These infrastructure components enable easy testability and flexible integrations without affecting the business logic.

3.2 Project Structure

The basis for this simple separation lies in the ports and adapters. The ports define interfaces for interactions, while the adapters connect the domain layer with GraphQL, the database, and external services seen in the structure.

Here you can see how we have structured our project.

- **Domain**: Core of the hexagonal architecture and place where the business logic takes place.
  - **Models:** Representation of all Objects in the domain layer.
  - **Ports:** Define communication interfaces between domain and components.
  - **Services:** Handle the business operations and processes.
- **Infrastructure:** Layer provides technical capabilities and external system interaction.
  - **Persistence:** It handles data storage and retrieval by interacting with a database.
    - **Adapter:** Handlers manage data flow to the core.
    - **Entities:** Represent database entities.
    - **Mapper:** Translate between domain models and database entities.
  - **GraphQL**: API to interact with the running server.
    - **InputModels:** Define the structure for input data coming from the client.
    - **Models:** Define the input and output data structure that the resolvers work with.
    - **Resolvers:** Methods to fetch, modify, or transform data before returning it to the client.
      - **Mappers:** Translate between domain models and GraphQL models, including both output and input model mappers
      - **Mutations:** Handle data modification.
      - **Queries:** Handle incoming queries and resolve data.

3.3 Database

In Figure 1, the relational schema of our implemented database is shown.

4 Implementation

4.0.1 Database in Docker

The Docker Compose configuration in the `docker-compose.yml`-file defines the database service and an application service that interact within an isolated network. The PostgreSQL database serves as the persistence layer, storing structured data related to the application's domain.

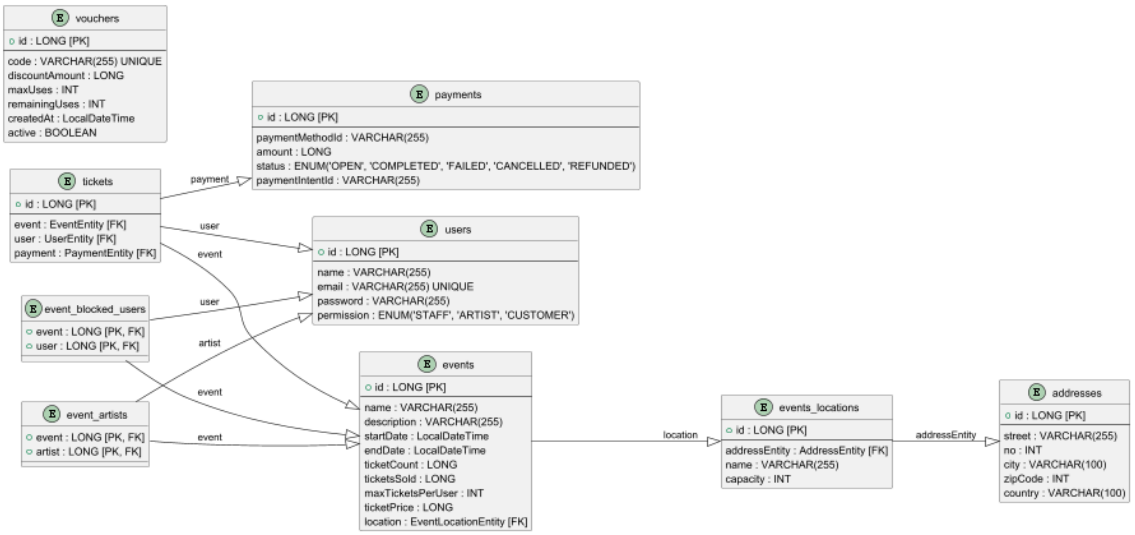

Abbildung 1: DatabaseUML

The database service is configured using the PostgreSQL 16 image, with predefined credentials and database settings. It listens on port 5432 and is configured to restart automatically in case of failure. The application container depends on this database, ensuring that the database service is available before the application starts.

Both services are connected through a dedicated Docker network ("backend"), which enables the application to communicate with the database using the service name "dbïnstead of an IP address. This setup abstracts the persistence layer, allowing the application to interact with the database seamlessly, whether to read, write, or update.

4.1 Component Frameworks

We selected PostgreSQL due to its robust support for complex queries, relational data, and its proven reliability. It integrates seamlessly with the backend, providing a solid foundation for data storage and retrieval while ensuring scalability.

GraphQL was chosen for its flexibility in querying data, enabling clients to request exactly what they need, which optimizes performance. Its schema-driven approach without a framework also provides strong validation, making the API more reliable and easier to implement.

4.2 Use-Cases

This section describes the actions that can be executed on the server via GraphQL. These actions can be performed, for example, by configuring Postman to use GraphQL and entering the URL "http://localhost:8080/graphql.Äfter refreshing the queries, the available actions can be selected and executed. The corresponding raw query text is generated automatically.

4.2.1 Users

The system provides functionality for managing accounts, enabling the creation, modification, and removal, as well as authentification of individual records. Each entry can be updated with new information or deleted when no longer needed. To facilitate efficient retrieval, records can be searched either by a unique identifier or filtered based on specific attributes such as name, email address, or assigned permissions. This ensures a flexible and structured approach to handling account information while maintaining accessibility and control over stored data.

4.2.2 Events

New events can be created with all necessary details, while existing ones can also be updated to reflect changes in name, description, location, availability, and pricing. If needed, they can also be removed permanently.

Individual events can be retrieved by their unique identifier, and the list of all can be filtered based on criteria such as description, name, location, ticket availability, pricing, and associated participants. Additional filters allow for searching by blocked users or artists. Popular events can also be accessed through a query that highlights trending entries, with pagination available for efficient navigation through results.

Event locations can also be managed and queried. New locations can be added, updated, or removed as needed. Individual event locations can be retrieved by their unique identifier. Additionally, a list of event locations can be filtered based on criteria such as address, name, or capacity, with pagination available to handle large datasets.

4.2.3 Tickets

Tickets can be purchased by users, specifying the event, the number of tickets, and the preferred payment method. If available, a discount code can be applied to reduce the total price. Payments are processed securely, and upon successful completion, the tickets are issued to the user. In case of cancellations or issues, a refund can be requested manually, updating the payment status accordingly. If an event is deleted, all associated ticket holders are automatically refunded.

Discount codes can be created with specific details, including the discount amount, maximum usage limits, and other criteria. These codes can be updated or deleted as needed to adjust promotional campaigns or remove expired offers. Additionally, codes can be activated or deactivated, controlling their availability for purchases. A deactivated code cannot be used, while an active one remains valid.

Both tickets and discount codes can be retrieved individually by their unique identifiers or searched using filtering criteria. For discount codes, parameters such as code, discount amount, maximum uses, remaining uses, and active status can be applied. Pagination ensures efficient navigation when handling large datasets.

5 Testing Strategy

For the testing strategy, both unit tests and integration tests were used. Through unit tests, we ensured that classes with complex or critical logic were tested without taking into account their integration with other components. This way, we ensured that said logic worked as expected, and it would also be easier to identify in which units errors occur.

On the other hand, integration tests ensure that the components work well with each other and achieve the expected communication. Tests were made that tested the communication from the input via GraphQL to the response, thus testing the complete functioning of the system.

6 Conclusion

6.1 Challenges during implementation

A key challenge was coordinating team schedules, which made it difficult to find time to work together. There were also differing interpretations of how to implement the hexagonal architecture, requiring alignment. Additionally, choosing the right framework took time to ensure compatibility with the system goals and project requirements.

6.2 Possible improvements

We noticed that, while the current queries work well, the search criteria could benefit from improvements, such as enabling more flexible search options instead of relying solely on exact string matching.

6.3 Learning

Nevertheless, we gained valuable insights into building a robust backend with GraphQL and PostgreSQL, while leveraging the power of hexagonal architecture for better separation of concerns.

Collaborating as a team also highlighted the importance of clear communication, especially in regard to architectural decisions. The use of hexagonal architecture helped us maintain modularity, allowing each component to evolve independently without disrupting the overall system, and made it easier to integrate and test different layers of the application efficiently.

7 Appendices

This article was drafted and refined using GPT-4 based on an outline containing related information. The GPT-4 output was reviewed, revised, and enhanced with additional content. It was then edited for improved readability and active tense, partially using Grammarly.