Software Architecture & Platforms — Amadeus Beckmann

Assignment #03 – Kubernetes-based EBike application

1. Introduction

In the third and final assignment, the EBike application is to be implemented based on an event-driven microservices architecture and deployed on a distributed infrastructure using Kubernetes. In addition to the existing e-bike, there will be another vehicle, the a-bike, which will act as the digital twin of an autonomous bicycle.

This report presents the architectural design and implementation of an event-driven EBike application based on a microservices-based architecture as specified in the assignment description.

2. Analysis and Design

As this work builds upon many of the findings of the Domain-Driven Design (DDD) conducted for the previous assignment, I will not repeat them here, but instead refer to the according report.

3. Architecture

This chapter outlines the key patterns and decisions that shape the system's robust and flexible design.

3.1. Kubernetes

Kubernetes is a container orchestration platform that automates the deployment, scaling, and management of containerized applications. It ensures high availability and fault tolerance by continuously maintaining the desired state of the system.

In the EBike application, Kubernetes is used to manage both the application's microservices and its supporting infrastructure. The following components were used to deploy and operate the system:

- **Pod**: This is the smallest deployable unit in Kubernetes. Each pod wraps one or more containers and includes shared storage and network resources. Every service in the application runs inside a pod.
- **Deployment**: Manages stateless application pods. It ensures that a specified number of replicas are running and replaces failed pods automatically.
- **StatefulSet**: Used for components that require persistent storage and stable network identities, such as databases and the Kafka broker.
- **Service**: Exposes pods on a stable network address. Internal services enable communication between microservices, while load balancer services expose the frontend and gateway to the outside.
- **Volume**: Provides persistent storage to pods, especially important for databases and message queues to retain data across restarts.
- **Secret**: Securely manages sensitive data like JWT signing keys, which are mounted into the gateway for authentication purposes.

Kubernetes enables the system to scale services independently, recover from failures automatically, and maintain clear separation of concerns. Secrets and environment variables are managed centrally, and persistent volumes ensure data durability across restarts. This makes Kubernetes an ideal foundation for running a distributed, event-driven microservices architecture.

3.2. Orchestration vs. Choreography

As in the second assignment, the individual microservices are organized around a gateway that accepts all requests from the client, authorizes them and forwards them to the corresponding service. Fundamentally, there are two approaches to coordinating client requests: orchestration and choreography.

Orchestration involves a central coordinator that directs interactions among services and manages the workflow explicitly. In contrast, choreography lets each service process user calls independently, with no central controller which makes the system more flexible and scalable.

For the third assignment, I opted for choreography. After testing orchestration in the second assignment, I wanted to explore a different approach that emphasizes decentralized communication and adaptability.

3.3. Message Queue

Probably the most important piece of technology for an event-driven application is a scalable and reliable message queue.

Event-driven architectures allow applications to scale individual services arbitrarily, as there is virtually no coupling between the services. In the case of the application presented in this report, the microservices communicate with each other almost exclusively via asynchronous events. However, this means that events must first be sent to the message broker, which in turn distributes them to the corresponding consumers so that they can be processed there.

There are many different options available these days like RabbitMQ, ActiveMQ, NATS, and many more. In the case of this assignment, I chose Apache Kafka as seen in the lecture.

Apache Kafka organizes messages into topics and partitions, allowing scalable and fault-tolerant data processing. Using consumer groups, Kafka assigns each partition to a single worker, which ensures that each microservice processes a unique subset of events in order. This message grouping maintains consistency and prevents duplicate processing, which is crucial for robust microservice architectures.

3.4. Background Tasks

In addition to distributing events to multiple consumers via a message queue, it is often useful in distributed systems to schedule asynchronous tasks to a single executor. This is the case, for example, if an active rental is to be charged every 60 seconds. For this purpose, the developed application relies on Redis to store scheduled tasks and distribute them to workers later.

3.5. CQRS

Each microservice is based on the CQRS principle, which separates operations into two distinct sides: write and read.

The write side, which handles commands, is responsible for processing business logic and enforcing rules, e.g., that a vehicle can only be rented if there is no active rental yet. It is solely dedicated to updating the system's state based on actions or commands received.

The read side remains isolated from business rules and is updated exclusively via events. These events are transmitted using Apache Kafka and ensure that the read model reflects the latest changes. In most cases, in addition to the basic data of a model, the read database also stores auxiliary data, such as the active rental of a customer or the total cost of a rental.

During the transit and processing of these events, the read database may contain stale data, however, this is a known trade-off in CQRS-based systems and is also known as eventual consistency. Figure 1 visualizes the general approach of CQRS.

3.6. Databases

To support the CQRS pattern, each microservice utilizes separate databases for command and query operations.

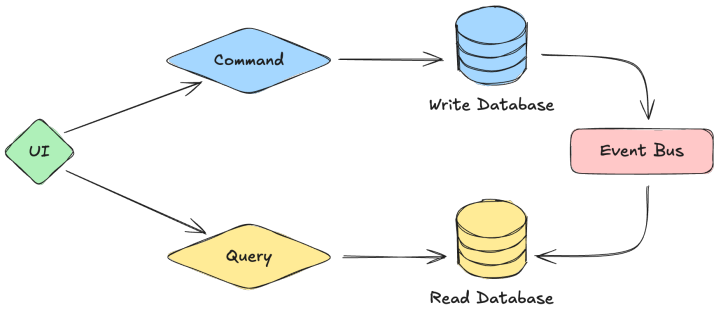

Figure 1: The CQRS design pattern. The strict differentiation between commands and queries is realized by even querying different databases for writing and reading. Changes to the write database are distributed via an event bus to reach eventual consistency.

PostgreSQL is used as the primary write database due to its robustness, transactional guarantees, and strong consistency. For read operations, MongoDB is employed to efficiently store and serve materialized views, offering flexible schemas, and fast querying capabilities.

3.7. Event Sourcing

Event sourcing is an architectural pattern where every change to an application's state is captured as an immutable event. Instead of storing just the current state, the system records a sequence of events that can be replayed to reconstruct any past state. This approach enhances auditability, traceability, and resilience, as every action is stored as a distinct, chronological record.

In the accounting service, event sourcing is used to compute each customer's credit balance. The service aggregates all confirmed payments, recorded expenses, and preliminary expenses that have not yet expired. By replaying these events, the system continuously updates and accurately reflects each customer's current balance.

4. Microservices

In this chapter, we will get a better overview over the microservice architecture composing the distributed EBike application.

The microservices-based architecture is structured around the bounded contexts identified during the analysis and design phase. Each context corresponds to an independent, loosely coupled service that can be developed, deployed, and scaled individually. To maintain clean boundaries and accommodate future changes, the microservices are implemented following the hexagonal architecture style, which ensures that domain logic remains independent of implementation details such as databases, transport protocols, or external frameworks.

All internal communication is based on HTTP, which promotes loose coupling between services. While this technically introduces some overhead compared to an gRPC-based approach, reducing the coupling between services allows each service to evolve with more flexibility without breaking the protocol that gRPC relies on.

As the design of the application follows the Command Query Responsibility Segregation (CQRS) principle, each microservice stores a local copy of relevant information rather than querying other services to reduce latency. However, this data may be stale for a short period of time while the corresponding event is still being processed. In these types of applications, data is eventually consistent.

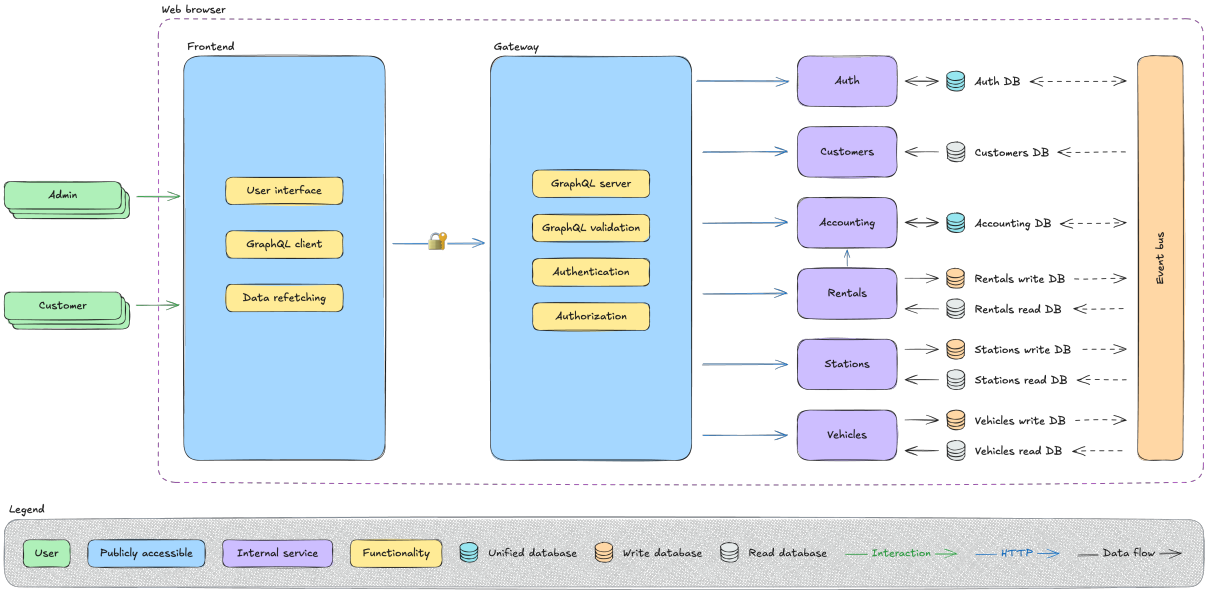

Figure 2: Component & Connector (C&C) view of the overall architecture of the application. The frontend communicates with the gateway via HTTPS, which in turn interacts with microservices over HTTP. The auth and accounting services use a unified database for both reads and writes, while the customer service uses only a read database. Other services follow the CQRS pattern with separate read and write databases. All services dispatch events to a shared event bus.

4.1. Frontend PUBLIC

The frontend is the user-facing interface that both admins and customers interact with. It provides all functionality from authentication related features like registration and login, to managing stations, vehicles, and payments for admins, and starting and stopping rentals, viewing past rentals, and managing payments and expenses for customers.

The frontend communicates exclusively with the gateway using GraphQL. The use of GraphQL enables a good developer experience, as the API is fully typed, and in many cases the requests sent by the frontend can be adapted without much effort and without having to make changes to the gateway.

4.2. Gateway PUBLIC

The gateway is the external entry point to the application. It validates and processes incoming GraphQL requests and forwards them to the according microservice. It also performs the necessary authentication and role-based authorization to limit the amount of work and traffic for the microservices. For this purpose, it manages user sessions via JWT cookies.

The gateway communicates with internal microservices using an HTTP API exposed by each microservice. Each of these interfaces is CRUD-based, allowing easy and loosely coupled interaction between the gateway and the microservices, as well as between the microservices themselves.

As opposed to the second assignment, the gateway no longer orchestrates the requests and instead forwards the requests immediately to the according microservice.

4.3. Auth INTERNAL

The first internal microservice is the authentication service. This service is responsible for registration, login, and logout, for validating the current user's session based on a session id, and providing a list of all registered users.

The auth service does not have different databases for read and write operations, as user authentication requires a strong data consistency. Instead, it always operates on the same database for any kind of operation.

Finally, this service does not react to any domain events, but emits events on registration and login so that other services can react to them and manage their resources accordingly.

4.4. Customers INTERNAL

The next service is the customer service. It stores and manages all relevant customer related data, i.e., their name, position, and credit balance, any active rental, and their last login. This service primarily reacts to events sent via the message queue and only provides endpoints to fetch all customers, a single customer, and an endpoint to update a customer's position.

In contrast to the auth service, the customer service only has a read database, as it operates exclusively on materialized views.

4.5. Accounting INTERNAL

The accounting service is another very important service for the Kubernetes-based EBike application as it manages customer payments, expenses due to rentals, and aggregating all records for a customer to compute their actual credit balance using event sourcing. For this purpose, it provides a lot of endpoints to create, read, update, and delete payments, create immediate expenses, for creating and finalizing preliminary expenses that expire if not finalized, and for querying payments, expenses, and the credit balance for a single customer.

As for the authentication service, for accounting a strong consistency is crucial, and waiting for events to be transmitted and processed could cause in unwanted problems. Therefore, all commands and queries operate on the same database.

The accounting service does not react to any domain events, but dispatches events for payments, expenses, and preliminary expenses.

4.6. Rentals INTERNAL

While the rental service only exposes a few endpoints for starting and stopping rentals, querying a customer's active rental, and querying all of a customer's past rentals, it has probably the most complex interaction with other services.

This complexity is necessary as for starting a rental it is necessary to check whether the customer's credit balance is sufficient to pay at least the unlocking fee. This fee is blocked until the rental started successfully and the preliminary expense was finalized or the preliminary expense expired. For this procedure, the rental services communicates with the accounting service.

In addition to the unlocking fee a customer has to pay to rent a vehicle, there is a recurring fee per minute depending on the vehicle type. This repeating task is managed as a background task and runs as long as the rental is still active. As soon as the rental is stopped, the background task is not executed again.

The rental service has both a write and a read database for rentals, fully implementing the CQRS pattern. In addition to the strictly necessary data such as customer id, vehicle id, start, and end date,

the rental service also stores the total cost of a rental. It also stores a materialized view for vehicles so that it does not have to query the vehicle service directly.

To store the materialized views for rentals and vehicles, the rental service reacts to rental events, vehicles events, and accounting events. Naturally, it also emits events when a rental started, when it was stopped, and when the total cost of a rental changed.

4.7. Stations INTERNAL

The station service is a quite simple service that provides an interface to create, read, update, and delete stations. It has both a regular database tables for write operations and a document database for read operations.

It emits events after creating, updating, or deleting stations, and also reacts to these events to update its own materialized views.

4.8. Vehicles INTERNAL

The final service is responsible for creating and deleting vehicles. Additionally, it exposes an endpoint to query all vehicles and a different endpoint to query only available endpoints.

As for the station service, vehicles are stored immediately in the write database, and after the according events have been processed, also in the read database. However, in addition to the base data of vehicles, i.e., their type, position, and battery level, the read database also stores the active rental of a vehicle for improved query performance.

For this purpose, the vehicle service not only processes vehicle related events, but also rental events in order to update its read databases accordingly.

5. Implementation

The implementation of the Kubernetes-based EBike application builds upon modern languages, libraries, and frameworks. By utilizing state-of-the-art tools and technologies, each service provides both a smooth user and developer experience.

5.1. Frontend

The frontend is a web-based application developed with Typescript, React[1], Next.js[2], Tailwind CSS[3], and urql[4]. These technologies help ensure a smooth developer experience and offer broad community support. The browser-based frontend allows developers to deploy application updates with zero downtime. The use of Typescript and strong typing helps prevent errors, and the combination of Next.js and React leverages well-known patterns to build interactive interfaces.

5.2. Gateway

The gateway is implemented using Typescript, Yoga[5]. Typescript provides consistent typing across the codebase and Yoga offers a straightforward GraphQL server interface. User authentication is implemented using JWT. This JWT has to be sent along all requests to the gateway to properly authenticate users and check their permissions.

[1] https://react.dev/ – The library for web and native user interfaces
[2] https://nextjs.org/ – Create high-quality web applications with the power of React components
[3] https://tailwindcss.com/ – A utility-first CSS framework
[4] https://commerce.nearform.com/open-source/urql/ – A highly customizable and versatile GraphQL client
[5] https://the-guild.dev/graphql/yoga-server/ – Fully-featured GraphQL server designed for optimal DX

5.3. Internal Services

The remaining, internal microservices *auth*, *stations*, *vehicles*, *customers*, *accounting*, and *rentals* are all implemented in Go. All implementations follow the hexagonal architecture style and follow SOLID principles.

For handling and validating HTTP-requests, I implemented a custom thin layer on top of the existing standard library. This layer enables request parsing and validation, error handling, and response formatting.

To enable the services to run simultaneously in any quantity, all ids generated by the services are based on the Snowflake algorithm developed by Twitter. That makes every ID a 64-bit unsigned integer. These ids are encoded as Base62 each time they *leave* a service for better legibility.

The services interact with their local PostgreSQL and MongoDB databases using sqlx[6] and the official MongoDB driver[7] respectively.

As the Kubernetes-based EBike application relies on Kafka for asynchronous, event-based communication, each microservice interacts with the Kafka message broker using IBM's Kafka library[8]. It allows us to easily dispatch events to the Kafka producer and consume and process specific events in our application.

Finally, for delayed task execution, the *rentals* service uses asynq[9], which uses Redis as a distributed task queue. This allows us to dispatch and execute tasks in the future, in our case to continuously charge active rentals every minute.

Over all, the amount of dependencies for this quite sophisticated event-driven application is quite low despite the fact that it is highly scalable and implements many aspects of a cloud-native architecture.

6. Deployment

The entire application is designed to be run on distributed hardware from the outset. Each service and its dependencies comes with a Kubernetes manifest that can be executed in your existing Kubernetes setup. The only thing that has to be prepared before you can deploy the application is a key pair for signing the JWT session cookie. Apart from that, the application is self-contained.

In addition to publishing in Kubernetes-based systems, it should also be possible to run the application in Docker (with or without Docker Swarm) without much effort. The necessary `docker-compose.yaml` configurations would still have to be prepared for this, but the existing Kubernetes manifests are a good starting point and should be able to be converted without further difficulties.

7. Limitations

Although the system works without any issues, there are some limitations, ranging from problems that are easy to solve to those that require more time and development to resolve.

1. **Not tested in production**: So far, the application has only been tested in development mode using Minikube[10] and Devspace[11]. However, the application has not yet been deployed on actual distributed infrastructure.
2. **Lots of repetition**: Many services have very similar functionality. For example, both the customer service and the vehicle service save the active rental for the customer and the vehicle respectively.

[6] https://github.com/jmoiron/sqlx/ – General purpose extensions to golang's database/sql
[7] https://github.com/mongodb/mongo-go-driver/ – The Official Golang driver for MongoDB
[8] github.com/IBM/sarama/ – Go library for Apache Kafka
[9] github.com/hibiken/asynq/ – Simple, reliable, and efficient distributed task queue in Go
[10] https://minikube.sigs.k8s.io/ – Set up a local Kubernetes cluster on macOS, Linux, and Windows
[11] https://www.devspace.sh/ – Open-source tool for Kubernetes to develop and deploy cloud-native software faster

And there are many other examples where models, service functions, repositories, or even event processors are the same or at least very similar. This means a considerable amount of extra work if changes need to be made to these areas in the future.

3. **Snowflake ID**: Although the Snowflake ID algorithm is built for high scalability, in its adjusted implementation for this assignment reserving only 4 bits for the worker id might limit the application's ability to scale arbitrarily. As of the time writing it is not exactly clear to me how collision resistant the id generation algorithm is under high load and with a lot of services running at the same time.
4. **Event sourcing without snapshots**: The accounting service computes a customer's credit balance using all confirmed payments, all expenses, and all preliminary expenses that are not yet expired. This can become slow over time as the amount of records to consider can grow enormously. A snapshot-based approach would solve this issue.
5. **Frontend repeatedly fetches data**: The frontend currently re-queries all data every second which creates unnecessary load. Introducing asynchronous mechanisms like websockets could enable an even more event-driven approach. This would reduce overhead and improve overall responsiveness.

8. Conclusion

This assignment demonstrated the design and implementation of a distributed, event-driven EBike application using microservices and Kubernetes. By adopting the findings from the Domain-Driven Design in the previous assignment and implementing CQRS and event sourcing, we can build a scalable and resilient system suitable for modern cloud environments. While the prototype performs well in development, future improvements such as creating snapshots for the customer's credit balance, code de-duplication, and real-time data updates via websockets will further enhance performance and maintainability.