KEA CARS™

Group:
Oliver Roat Jørgensen
Marcus Klinke Thorsen

Date of delivery: 23/05-2025

Github:
https://github.com/Marcus-K-Thorsen/kea_car_microservices.git
Monolithic: https://github.com/niiicolai/kea_car_backend

1. Introduction

Our KEA Cars Microservices project is an evolution of our previous monolithic application, redesigned to embrace a modern microservices architecture. The system serves as an internal platform for KEA CARS™ employees, streamlining the management of car sales, customer registrations, and related business processes. By decomposing the original monolithic system into independently deployable microservices, we aim to achieve greater scalability, maintainability, and flexibility.

Each microservice in the new architecture is responsible for a distinct business domain: the admin service manages employees and their roles; the employee service handles the core business logic, such as car sales and registrations; the customer service exposes non-critical information to potential customers and unauthenticated users; and the auth service manages employee authentication for secure access to protected endpoints. These services communicate over well-defined APIs, enabling independent development, deployment, and scaling, while allowing each service to use the most suitable database technology for its needs.

The primary objective of this project is to deliver a robust, distributed system that supports the daily operations of KEA CARS™. Employees can securely log in, manage customer and car data, process sales, and track transactions through a unified web interface powered by the underlying microservices. The architecture is designed to facilitate future enhancements, such as adding new services or integrating with external systems, without disrupting existing functionality.

Transitioning to microservices has provided our team with valuable experience in distributed system design, API development, and cloud-native deployment practices. This project has fostered collaboration and problem-solving, mirroring the challenges faced in real-world software engineering. Ultimately, KEA Cars Microservices demonstrates our ability to architect, implement, and deploy a scalable enterprise application using contemporary software development methodologies.

1.1. Problem description

The main challenge addressed in this project was transitioning from a monolithic architecture, where all business logic and data were tightly coupled in a single application and database, to a distributed microservices architecture. In the new setup, each microservice operates independently with its own database, leading to challenges such as data synchronization, consistency across services, and managing multiple databases to ensure that information remains accurate and up-to-date throughout the system.

1.2. Functional and non-functional requirements

The following section describes the functional and non-functional requirements of the KEA CARS™ system.

1.2.1. Functional requirements

This chapter outlines the functional requirements of the KEA Cars system, structured as user stories. The user stories are divided into two main categories: employees and customers.

Within the employee category, there are three distinct roles: **admin**, **manager**, and **sales person**. These roles are organized hierarchically, where an admin has all the permissions of a manager and a sales person, and a manager has all the permissions of a sales person. In other words, any functionality available to a sales person is also available to a manager and an admin, and any functionality available to a manager is also available to an admin.

The customer category consists of unregistered users who interact with the system as potential customers.

1.2.1.1. Employee Stories

- As an employee with the role **admin**, I want to create a new employee, so that I can onboard new staff into the system.
- As an employee with the role **admin**, I want to update an existing employee's information, so that I can keep employee records accurate and up to date.
- As an employee with the role **admin**, I want to delete an employee, so that I can remove employees who are no longer part of the organization.
- As an employee with the role **admin**, I want to undelete a previously deleted employee, so that I can restore access for employees who return to the organization.
- As an employee with the role **sales person**, I want to view all models for a specific brand, so that I can help customers select the right car model.
- As an employee with the role **sales person**, I want to view all available insurances, so that I can register insurances for a purchase.
- As an employee with the role **sales person**, I want to view all customers, so that I can find and identify customers for a purchase.
- As an employee with the role **sales person**, I want to create a new customer, so that I can register a customer for a car purchase.
- As an employee with the role **sales person**, I want to create a car, so that I can register a car that is about to be purchased and assign it to myself.
- As an employee with the role **manager**, I want to create a car, so that I can register a car that is about to be purchased and assign it to any employee in the system.

- As an employee with the role **sales person**, I want to create a purchase for a car that is assigned to me, so that I can record a completed car sale for a customer.

- As an employee with the role **manager**, I want to create a purchase for a car that is assigned to any employee in the system, so that I can record a completed car sale.

- As an employee with the role **manager**, I want to create a new insurance, so that the available insurance options are up to date for all employees and customers.

- As an employee with the role **manager**, I want to update an insurance, so that insurance offerings reflect the latest terms and options.

- As an employee with the role **manager**, I want to delete a customer, so that customer records can be removed when necessary.

- As an employee, I want to log in to the system using my unique email and password, so that I am able to interact with the system.

1.2.1.2 Customer Stories

- As a customer, I want to view all available car brands, so that I can see which brands I can choose from when buying a car.

- As a customer, I want to view all available car models, or filter models by a specific brand, so that I can decide which model I might be interested in.

- As a customer, I want to view all available insurance options, so that I can consider which insurance I might want for my new car.

1.2.2. Non-Functional Requirements

The non-functional requirements for the KEA Cars Microservices project address a range of system qualities that ensure the platform's effectiveness and long-term viability. These include reliability, security, portability, and interoperability. The system is designed with containerization, modular APIs, and deployment flexibility in mind, supporting robust operations and ease of management.

1.2.2.1. Reliability

Reliability describes the system's ability to consistently perform its intended functions without failure, ensuring users can depend on the system to be available and accurate at all times. A reliable system minimizes downtime, prevents data loss or corruption, and gracefully handles unexpected conditions. The following requirements outline how reliability should be achieved in the KEA Cars Microservices project:

1. **High Availability**
   - The system should automatically redeploy containers if a service instance fails, ensuring minimal downtime.
   - The system should be designed to support integration with load balancers to route traffic to available service instances, although explicit load balancing may not be implemented in the current setup.

2. **Location Independence**
   - The system should behave consistently regardless of the deployment environment or geographic location.
   - All time-related data should be stored and processed using a standard timezone (e.g., UTC) to avoid discrepancies across regions.

3. **Idempotence**
   - The system should ensure that repeated requests to create the same resource do not result in duplicate entries.
   - All create operations should require the client to provide a unique identifier (ID), and if the resource already exists, the system should return the existing resource instead of creating a new one.

4. **Immutability**
   - The system should avoid destructive operations where possible.
   - Where full immutability is not feasible, the system should ensure that updates and deletions are performed in a controlled and auditable manner.

1.2.2.2. Security

Security defines the system's ability to protect data, ensure privacy, and control access to its resources.

1. **Authentication and Authorization**
   - The system should require employees to authenticate using secure JWT tokens.
   - Authorization should be enforced based on employee roles, ensuring that only users with the appropriate privileges can perform sensitive actions or access protected endpoints.

2. **Input Validation and Injection Protection**
   - The system should validate all user input to prevent injection attacks and ensure data integrity.

3. **Password Security**
   - The system should hash all passwords before storing them in the database and never return passwords to end users.
   - Passwords must meet minimum security requirements, such as length and resistance to known breaches.

4. **Access Control and Permissions**
   - For API endpoints that interact with a database on behalf of end users, the system should use a dedicated application user with only the minimum required privileges. These privileges should be limited according to the role and actions permitted for the authenticated user, ensuring that only authorized operations can be performed.
   - API endpoints that do not require authentication and are accessible by anyone should only provide read access and never allow write operations to the underlying data.

5. **Error Handling and Information Disclosure**
   - The system should only return prepared, generic error messages to end users, avoiding the exposure of sensitive internal details that could be exploited by malicious actors.

1.2.2.3. Portability

Portability refers to the system's ability to be easily deployed and run across different environments.

1. **Containerized Microservices**
   - Each microservice should be packaged as a Docker container, including all dependencies required for execution.
   - The system should support deployment and orchestration using both Docker Compose and Kubernetes.

2. **Independent Deployment**
   - Microservices should be independently deployable and manageable, allowing updates or scaling of one service without affecting others.

3. **Externalized Configuration**
   - Configuration should be managed through environment variables and configuration files, enabling adaptation to different deployment scenarios without code changes.

1.2.2.4. Interoperability

Interoperability describes the system's ability to enable communication and data exchange between microservices and clients through well-defined interfaces.

1. **Standardized APIs**
   - Microservices that are accessed by end users should expose RESTful APIs, following standard HTTP methods and status codes to ensure consistency and ease of integration.
   - APIs should use clear and consistent resource naming conventions and data formats (such as JSON).

2. **API Documentation**
   - All APIs must provide interactive documentation through Swagger (OpenAPI).
   - Every endpoint, request object, and response object must include descriptive documentation to assist developers and users in understanding the system's capabilities.

3. **Message-Based Communication**
   - Microservices should communicate with each other using message brokers (such as RabbitMQ) rather than direct API calls, supporting loose coupling and modularity.

1.3. Explanation of choices for the technology stack

The following describes the chosen technologies used in the project and the reasons behind them.

1.3.1. Programming Language

For the backend of the KEA Cars Microservices system, Python was chosen as the primary programming language. This decision was largely influenced by the fact that the previous monolithic version of the system was also implemented in Python, allowing the team to leverage existing code, patterns, and experience.

Python's versatility, readability, and extensive ecosystem of libraries make it well-suited for rapid development and maintenance of microservices. Since everyone on the team already knows Python well, it made working together easier and meant no one had to spend extra time learning a new language.

1.3.2. Frameworks

For the development of the backend microservices, FastAPI was chosen as the primary framework. This decision was influenced by the team's prior experience with FastAPI in the earlier monolithic project, as well as its reputation for simplicity and efficiency in building modern web APIs. FastAPI provides a straightforward and intuitive approach to defining endpoints, request validation, and dependency injection. One of its key advantages is the seamless integration with Swagger (OpenAPI), which enables automatic and interactive API documentation, making it easier for both developers and stakeholders to understand and test the available endpoints.

FastAPI is particularly beneficial in a microservices architecture because it is lightweight, fast, and designed to handle asynchronous operations efficiently. This allows each microservice to be independently developed, deployed, and scaled, while still maintaining high performance and responsiveness. Its modular design fits naturally with the principles of microservices, supporting clear separation of concerns and enabling teams to iterate quickly on individual services without impacting the entire system. Overall, FastAPI was selected for its developer-friendly design, robust feature set, and strong support for API documentation, all of which are essential in a distributed microservices environment.

1.3.3. Databases

The choice to use both MySQL and MongoDB in the KEA Cars Microservices project was influenced by our previous experience with these technologies in the earlier monolithic system. This familiarity allowed us to efficiently reuse existing logic and code for database interactions. MySQL was selected for its strengths in handling structured data and transactional operations, making it ideal for scenarios where data integrity and precise control over inserts are required. MongoDB, on the other hand, excels at fast read operations and flexible data retrieval, which is particularly useful for endpoints that primarily serve data to clients.

In our architecture, MySQL databases are used for core business operations and data creation, while MongoDB databases are optimized for read-heavy workloads and quick access to frequently requested information. Each microservice interacts with its designated database(s) according to its responsibilities, with some services writing to MySQL and others reading from or writing to MongoDB as needed. This hybrid approach leverages the strengths of both database systems to support the diverse requirements of the platform.

1.3.4. Message Broker

RabbitMQ was chosen as the message broker for the KEA Cars Microservices project. This decision was influenced by both educational guidance and prior team experience. RabbitMQ was introduced and recommended by the course instructor, making it a natural fit for the project's learning objectives. Additionally, one team member had hands-on experience with RabbitMQ from a previous internship, which helped accelerate the adoption and integration of the technology.

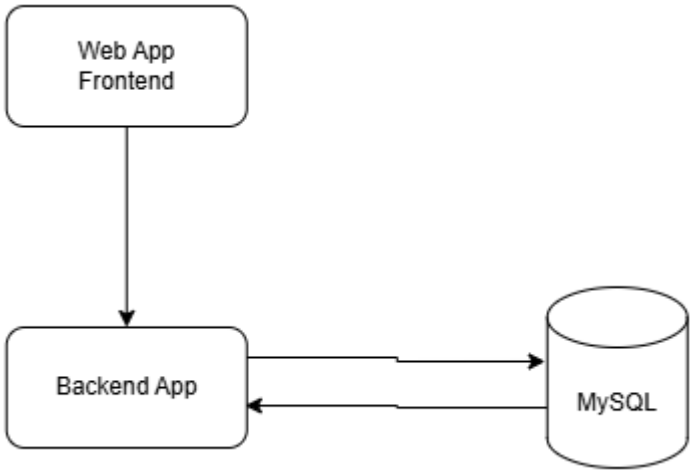
RabbitMQ is a widely used, reliable, and well-documented message broker that supports a variety of messaging patterns and protocols. Its robust feature set enables asynchronous communication between microservices, allowing the system to decouple service interactions and improve scalability and fault tolerance. By using RabbitMQ, the project benefits from reliable message delivery, flexible routing, and the ability to handle complex workflows without tightly coupling services. This aligns well with the microservices architecture, where independent services must coordinate and exchange information efficiently. Overall, RabbitMQ provides a proven and effective solution for managing inter-service communication in distributed systems.

2. System Architecture

System architecture refers to the high-level structure of a software system, describing how its components are organized and how they interact. It provides an overview of the system's main building blocks, their responsibilities, and the communication patterns between them. The following chapter outlines the architectural design of the project, illustrating the transition from a monolithic application to a microservices-based solution and detailing the roles and interactions of each service within the system.

2.1. Monolithic System Overview

The previous version of the project was built as a classic monolithic system, where all core functionality was contained within a single backend application. The web app frontend communicated directly with this backend, which handled all business logic and interacted with a single database, such as MySQL or MongoDB. This straightforward architecture meant that all components were tightly coupled and deployed together as one unit, making the system simple but less flexible and harder to scale or maintain. The simplicity of this approach is illustrated in the following diagram.



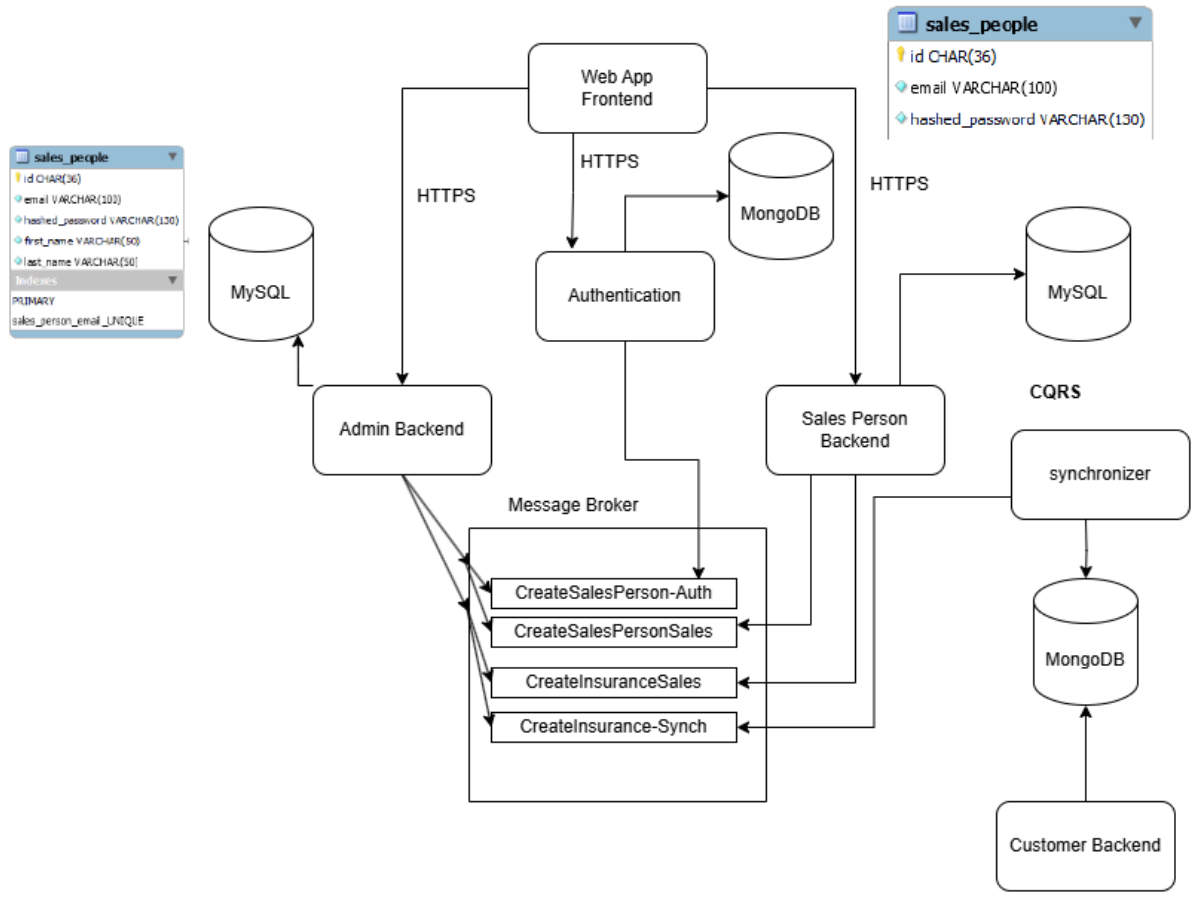
At the outset of transforming the monolithic system into a microservices-based architecture, we began by analyzing the existing application to identify areas that could be separated into independent services. While the majority of the original system's responsibilities—such as handling car sales, customer registrations, and related business logic—were retained within what is now the employee microservice (referred to as the "Sales Person Backend" in the diagram), we identified several areas that could benefit from further separation.

We recognized that tasks like employee management—creating, updating, and deleting employees—should be reserved for higher-ranked staff, leading us to plan for a dedicated admin microservice. Additionally, we saw that non-critical data, such as vehicle brands, models, colors, insurances, and accessories, was previously accessible to both employees and potential customers through the same backend, often without authentication. To address this, we decided to design a customer microservice that would provide public access to this non-sensitive data, backed by its own database. To keep this data consistent with the employee-facing system, we planned a synchronizer microservice to update the customer database with relevant changes, including the creation and updating of insurances so that the customer microservice could always serve the latest information.

We also determined that authentication and token management should be moved out of the main backend and into a dedicated authentication microservice, using its own MongoDB database for efficient read operations during login.

To facilitate communication and data synchronization between these planned services, we chose RabbitMQ as a message broker. These initial design decisions formed the foundation

for our transition to a microservices architecture, which is reflected in the first iteration of our system architecture diagram shown below.

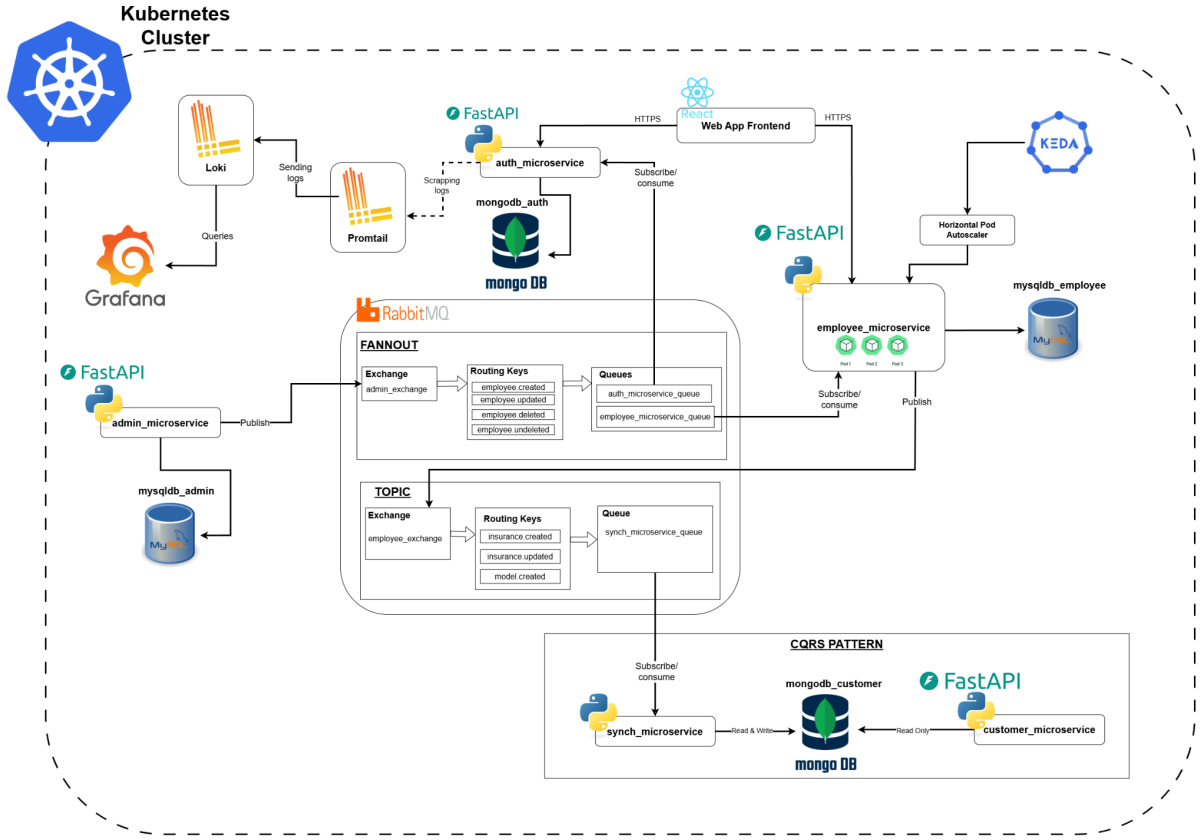


2.2. Introduction to the microservices architecture

The project includes five microservices:

- Admin Microservice: admin_microservice
- Authentication Microservice: auth_microservice
- Employee Microservice: employee_microservice
- Synchronizer Microservice: synch_microservice
- Customer Microservice: customer_microservice

The KEA Cars Microservices system consists of five main microservices: admin_microservice for managing employees and roles, auth_microservice for authentication and authorization, employee_microservice for handling car sales and business logic, customer_microservice for exposing public, read-only data to potential customers, and synch_microservice for synchronizing data between services. Each microservice is connected to its own dedicated database, either MySQL or MongoDB, and services communicate asynchronously using RabbitMQ as a message broker. Additionally, the auth_microservice has its logs collected by Promtail, sent to Loki, and visualized in Grafana for monitoring purposes. An illustration at the end of this section provides an overview of the system architecture and the relationships between these components.



2.3. Microservice Admin Description

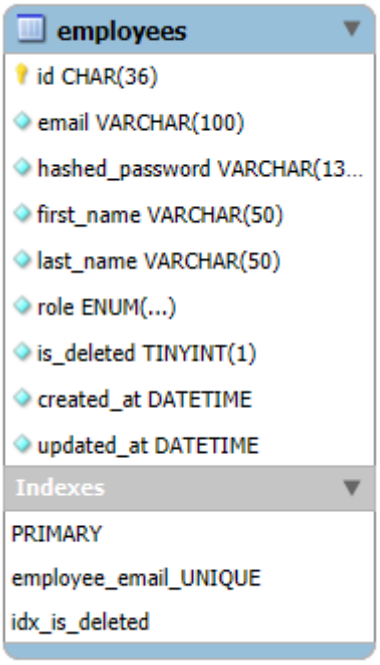
The admin_microservice is responsible for managing all employee records within the KEA Cars system. Its primary function is to allow users with the admin role to create, view, update, delete, and undelete employee accounts. Access to these operations is strictly limited to employees with administrative privileges, ensuring that only authorized personnel can manage sensitive employee data.

This microservice connects to a dedicated MySQL database that stores all employee information, including unique identifiers, email addresses, hashed passwords, names, roles (admin, manager, or sales person), and status flags indicating whether an employee is active or deleted. The admin_microservice is the only service with permission to read from and write to the employee table in this database, maintaining strict control over employee data integrity and security.

When employee records are created, updated, deleted, or undeleted, the admin_microservice publishes corresponding messages to the admin_exchange in RabbitMQ, using a fanout exchange type. These messages use routing keys such as employee.created, employee.updated, employee.deleted, and employee.undeleted, allowing other microservices to stay synchronized with changes to employee data.

Additionally, the admin_microservice is responsible for securely hashing passwords when new employees are registered or when existing employees update their passwords, ensuring that sensitive credentials are never stored in plain text.

The connected MySQL database, mysqldb_admin, is structured to store comprehensive employee data, including unique IDs, emails, hashed passwords, first and last names, roles, deletion status, and timestamps for creation and updates. Here is a EER Diagram of the database connected to the admin_microservice:



2.4. Microservice Authentication Description

The auth_microservice is responsible for handling authentication within the KEA Cars system by issuing JSON Web Tokens (JWT) to employees. This service enables secure access to protected endpoints in both the admin_microservice and employee_microservice by verifying user credentials and generating tokens upon successful login. It connects to a dedicated MongoDB database, mongodb_auth, which stores employee records including emails and hashed passwords. The data structure in this database mirrors that of the admin MySQL database, except that deleted employees are fully removed from mongodb_auth, ensuring that only active employees can authenticate and receive tokens.

The auth_microservice subscribes to the admin_exchange in RabbitMQ, consuming messages via the auth_microservice_queue. It listens for events such as employee creation, updates, deletions, and undeletions from the admin_microservice, and updates its MongoDB database accordingly to stay synchronized with the latest employee data. While the message consumer can both read from and write to the database, the API endpoints used for authentication only have read access, further enhancing security.

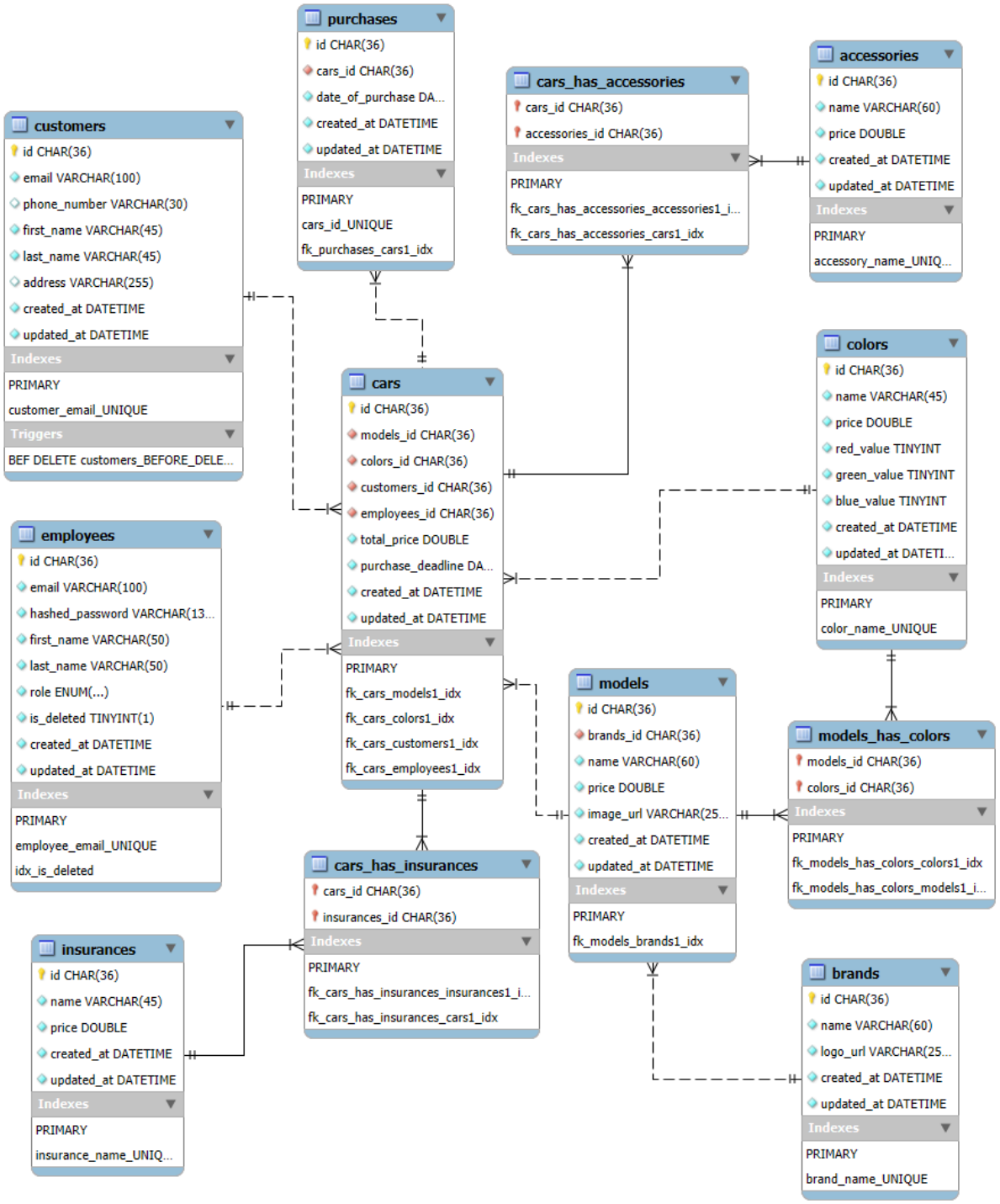
This microservice is the central point for any frontend or client application to obtain a JWT, which is required for accessing secured endpoints across the system. Additionally, the auth_microservice is configured for observability: Promtail scrapes its logs and forwards them to Loki, making them available for monitoring and analysis in Grafana. This setup ensures robust authentication, up-to-date employee data, and comprehensive logging for operational visibility.

2.5. Microservice Employee Description

The employee_microservice is the core service responsible for handling the majority of business logic within the KEA Cars system, carrying over much of the functionality from the original monolithic application. This microservice is designed to be used by all employees, including admins, managers, and sales people. It provides endpoints for managing and retrieving data related to brands, models, colors, accessories, insurances, customers, cars, and purchases. While all employees can read from the system, only specific roles are permitted to perform certain write operations, such as creating or updating insurances, cars, and customers. Notably, direct modifications to employee records in the mysqldb_employee database are only performed by the service when it consumes messages from the admin_microservice, ensuring that employee data remains consistent and centrally managed.

The employee_microservice acts as both a consumer and producer of messages. It consumes messages from the admin_exchange (fanout type) using the employee_microservice_queue, processing routing keys like employee.created, employee.updated, employee.deleted, and employee.undeleted to keep its employee data synchronized with changes made in the admin service. Additionally, it publishes messages to the employee_exchange (topic type) with routing keys such as insurance.created and insurance.updated, allowing other services to stay updated on changes to insurance data.

The mysqldb_employee database connected to this microservice is structured to support a wide range of business operations. It contains tables for employees, customers, cars, brands, models, colors, accessories, insurances, and purchases, as well as the necessary relationships between these entities. This comprehensive schema enables efficient management of all aspects of the car sales process. The structure and relationships within the mysqldb_employee database are visualized in the accompanying EER diagram.



2.6. Microservice Synchronizer Description

The synch_microservice is a background service dedicated to keeping non-critical data in sync between the employee and customer domains. It does not expose any API endpoints and operates solely as a message consumer. The synch_microservice subscribes to the employee_exchange (topic type) via the synch_microservice_queue, receiving messages from the employee_microservice about changes such as new or updated insurances, brands, models, colors, and accessories. Upon receiving these messages, it updates the mongodb_customer database, which is shared with the customer_microservice.

This design follows the CQRS (Command Query Responsibility Segregation) pattern, where the synch_microservice is the only service allowed to write to the mongodb_customer database, while the customer_microservice is responsible for reading from it. This ensures that customers always have access to up-to-date, non-sensitive information—such as available brands, models, colors, accessories, and insurances—without exposing any private or critical data related to employees, customers, or purchases. The synch_microservice thus plays a crucial role in maintaining data consistency and security across the system.

2.7. Microservice Customer Description

The customer_microservice is designed to provide public, read-only access to non-critical data within the KEA Cars system. It connects to a shared MongoDB database, mongodb_customer, which is maintained and updated exclusively by the synch_microservice. This microservice does not interact with RabbitMQ and does not require any form of authentication or authorization, allowing unauthenticated users—such as potential customers—to freely access its endpoints.

Through its API, the customer_microservice enables users to retrieve information about available insurances, accessories, colors, brands, and car models (including filtering models by brand). This allows customers to explore the options offered by the KEA Cars company and make informed decisions before engaging with a salesperson to initiate a purchase. The customer_microservice is strictly limited to reading data; it cannot modify or add any information in the database, ensuring the integrity and security of the underlying data. This approach makes it easy for customers to browse offerings while maintaining a clear separation between public and internal system operations.

2.8. Communication Between Microservices

The microservices in the KEA Cars system communicate primarily through asynchronous message queues and shared databases, rather than direct API calls. When changes occur in the Admin microservice—such as creating, updating, deleting, or undeleting an employee—messages are published to RabbitMQ, which are then consumed by the Authentication and Employee microservices to synchronize employee data across their respective databases. Similarly, when the Employee microservice creates or updates insurance information, it sends messages to the Synchronizer microservice, which then updates the customer MongoDB database. The Customer microservice does not communicate directly with other services, but instead reads the latest data from its dedicated

MongoDB database, which is kept up to date by the Synchronizer. Authentication is handled centrally by the Authentication microservice, which issues JWT tokens used by other services to authorize requests, but there is no direct API communication between these services for authentication purposes. This architecture ensures loose coupling, data consistency, and scalability across the system.

2.9. Description of the patterns and techniques used

The following describes additional patterns and techniques used to implement the microservices.

2.9.1. CQRS

The Command Query Responsibility Segregation (CQRS) pattern is a software architectural approach that separates the operations that modify data (commands) from those that read data (queries). By dividing these responsibilities, systems can optimize and scale read and write operations independently, often improving performance, security, and maintainability.

In the KEA Cars Microservices system, the CQRS pattern is clearly demonstrated in the interaction between the synch_microservice and the customer_microservice, both of which share access to the mongodb_customer database. The synch_microservice is solely responsible for handling all write operations to this database, updating non-critical data such as brands, models, colors, accessories, and insurances based on messages it receives from other services. In contrast, the customer_microservice is strictly limited to read-only access, providing unauthenticated users with up-to-date information about available options without ever modifying the underlying data. This clear separation of write and read responsibilities exemplifies the CQRS pattern in practice within the system.

2.9.2. Immutable Data (Tombstone and Snapshot pattern)

Immutable data refers to the practice of never physically deleting or altering existing records in a database. Instead, changes are tracked by marking records as inactive or deleted (the Tombstone pattern), or by creating new versions or snapshots of data over time (the Snapshot pattern). These patterns are often used to ensure data integrity, enable audit trails, and support historical queries.

In the KEA Cars Microservices system, we do not fully implement the Tombstone or Snapshot patterns, but we do incorporate some of their principles. Specifically, when an employee is deleted in either the admin_microservice or the employee_microservice, the record is not physically removed from their databases. Instead, a field called is_deleted is set to true, marking the employee as deleted while preserving the original data. This approach ensures that employee records remain available for auditing or recovery, even after deletion. While this does not provide full immutability or historical snapshots, it does capture the essence of immutable data by avoiding destructive operations and maintaining a traceable record of changes.

2.9.3. Idempotence

Idempotence refers to the property of certain operations in a system where performing the same action multiple times produces the same result as performing it once. This is important for ensuring reliability and consistency, especially in distributed systems where network issues or retries can cause duplicate requests.

In this project, idempotence is enforced for all create operations across the microservices. Each create endpoint requires the client to provide a unique UUID for the resource being created. When a create request is received, the service checks if a resource with the given UUID already exists in the database. If it does, the existing resource is returned and no new entry is created. This approach guarantees that repeated requests with the same UUID will not result in duplicate records, maintaining data integrity throughout the system.

2.9.4. Commutative Message Handlers

Commutative message handlers are designed so that the order in which messages are processed does not affect the final state of the system. In distributed systems, this property is valuable because message delivery order is not always guaranteed, especially when using asynchronous message brokers like RabbitMQ.

In this project, commutative message handling is addressed by incorporating timestamp checks in the message processing logic. When a message is received—such as for creating, updating, deleting, or undeleting an employee or insurance—the handler compares the timestamps (created_at or updated_at) of the incoming message with the current state in the database. If the message is older than the current state, it is either ignored or requeued for later processing. Only messages that represent a more recent state are applied.

By enforcing these timestamp checks, the system ensures that the final state is determined by the most recent message, regardless of the order in which messages arrive. This design reduces the risk of data inconsistency due to out-of-order message delivery and demonstrates a practical application of commutative message handling in a distributed microservices environment.

3. Deployment

3.1. Introduction to the cloud deployment

Our project is designed for deployment in a cloud-native environment using containerization and orchestration technologies. For this project, all microservices and supporting infrastructure are deployed to a single virtual machine running a self-hosted Kubernetes cluster. This approach provides a balance between real-world cloud deployment practices and the practical constraints of our development environment, allowing us to simulate production-like scalability, resilience, and service isolation without spending too much money on having it cloud deployed.

3.2. Description of used technologies

3.2.1. Docker

Each microservice in our KEA Cars Microservices project is packaged as a Docker container. Docker enables us to encapsulate application code, dependencies, and runtime environment, ensuring consistency across development, testing, and production. Dockerfiles are used to define the build process for each service, and images are built and pushed as part of our CI/CD pipeline.

3.2.2. Kubernetes

Kubernetes is used to orchestrate and manage the deployment of all microservices and supporting components. Our kubernetes folder contains YAML manifests for deploying each service, as well as for configuring persistent storage, networking, and service discovery. Kubernetes provides automated deployment, scaling, and recovery of services, ensuring high availability and efficient resource utilization. We also use Kubernetes to manage supporting infrastructure such as logging, with Loki and Promtail, which aggregate and store logs from all services in a centralized location.

3.2.3. Logging Stack: Loki & Promtail

For centralized logging, we deploy Loki as our log aggregation system and Promtail as the log collector. Promtail runs as a DaemonSet, collecting logs from all pods and forwarding them to Loki. To visualize and query these logs, we use Grafana, which connects to Loki and provides a powerful web interface for log analysis, monitoring, and troubleshooting. This logging stack enables us to efficiently collect, store, and analyze logs from all microservices in a single, centralized location.

3.3. CI/CD pipeline description

Our project uses GitHub Actions to implement a Continuous Integration and Continuous Deployment (CI/CD) pipeline for each microservice and database seeding script. The workflow files are located in the **workflows** directory and include:

- admin_microservice_ci_cd.yaml
- auth_microservice_ci_cd.yaml
- customer_microservice_ci_cd.yaml
- employee_microservice_ci_cd.yaml
- seed_auth_db_ci_cd.yaml
- seed_customer_db_ci_cd.yaml
- synch_microservice_ci_cd.yaml

Each workflow is triggered automatically when changes are pushed to the main branch or when the workflow file itself is updated. The pipeline performs the following steps for each microservice:

- Checkout the code using the actions/checkout action.
- Authenticate with Docker Hub using credentials stored in GitHub Secrets.
- Build the Docker image for the relevant microservice.
- Push the Docker image to Docker Hub with the latest tag.

For example, the **customer_microservice_ci_cd.yaml** workflow builds and pushes the **customer_microservice** image whenever changes are made to its source code or the workflow file. This ensures that the latest version of each service is always available for deployment.

This automated CI/CD process helps us maintain code quality, reduces manual deployment steps, and ensures that all microservices are consistently built and published as Docker images whenever updates are made.

3.3.1. Github Actions

Our project uses GitHub Actions to implement continuous integration and continuous deployment (CI/CD) for all microservices and supporting tasks. Each microservice and utility has its own workflow file located in the workflows directory, ensuring that each service can be built and deployed independently.

Workflows are triggered by events such as pushes to the main branch or pull requests. Each workflow consists of sequential jobs, including building Docker images and deploying the updated containers to the Kubernetes cluster. The deployment process involves pushing Docker images to a container registry and applying the latest Kubernetes manifests to update the running services.

This setup ensures that code changes are automatically validated and deployed, reducing manual intervention and enabling rapid, reliable delivery of updates across all microservices. By using GitHub Actions, the project maintains a consistent and automated pipeline for integration and deployment, supporting both development velocity and system stability.

3.4. Monitoring and logging of the deployed system

To ensure observability, reliability, and maintainability of our KEA Car Microservices system, we have implemented centralized logging and monitoring using the Loki and Promtail stack, as configured in our kubernetes/logging directory.

Loki is a log aggregation system designed for Kubernetes, which efficiently collects, stores, and queries logs from all microservices. Our deployment is defined in loki/loki-deployment.yaml, with persistent storage managed by loki/loki-pvc.yaml and service exposure via loki/loki-service.yaml. The configuration file loki/loki-config.yaml specifies how logs are ingested and stored.

Promtail acts as the agent that runs on each Kubernetes node, tailing the logs of all containers and forwarding them to Loki. Its deployment and configuration are managed by files such as **promtail/promtail-deployment.yaml**, **promtail/promtail-config.yaml**, and the necessary RBAC permissions are set up in **promtail/promtail-cluster-role.yaml** and **promtail/promtail-cluster-role-binding.yaml.**

With this setup, all logs from our microservices are collected centrally, making it easy to search, analyze, and visualize logs for debugging and monitoring purposes. This logging infrastructure is integrated with Grafana for advanced querying and dashboarding, providing a comprehensive view of system health and activity.

By leveraging Loki and Promtail in our Kubernetes environment, we ensure that our system is observable and that operational issues can be quickly detected and resolved, supporting both development and production needs.

4. Project management and team collaboration

Effective project management and team collaboration were essential to the success of our KEA Cars Microservices project. As a two-person team, we emphasized clear communication, shared responsibility, and agile practices to ensure steady progress and high code quality throughout the development process.

4.1. Description of the methods used during the project

We adopted Scrum as our primary project management methodology. We held daily stand-up meetings to discuss progress, identify blockers, and plan our next steps. This routine helped us stay aligned and adapt quickly to any changes or challenges.

Pair programming was a core part of our workflow. By working together, we were able to share knowledge, catch errors early, and make collaborative decisions on design and implementation. This approach also fostered continuous code review and improved the overall quality of our codebase.

For task management and planning, we used Trello to organize user stories, prioritize features, and track progress. User stories were created to capture requirements from the perspective of end users, and tasks were moved across the board as they progressed from planning to completion.

4.2. Documentation strategy

We prioritized clear and accessible documentation throughout the project. For API documentation, we used Swagger (OpenAPI), which automatically generates interactive documentation from our FastAPI. This allowed both us to easily explore and understand the available endpoints and their expected inputs and outputs. Additional project documentation, such as setup instructions and architectural decisions, was maintained in markdown files within the repository.

5. Conclusion

This project has been a big step forward for us in learning how to build modern backend systems. Moving from a simple monolithic setup to a microservices architecture helped us really understand the benefits and challenges of working with distributed systems. We didn't just build something that works — we designed something that's scalable, modular, and close to how systems are built in the real world.

5.1. Advantages and challenges of the distributed systems

One of the best things about going with microservices is how clean and organized everything becomes. Each service has one job, which has made it easier for us to think clearly about what each part of the system should do. For example, the **auth_microservice** only handles logins and tokens, while the **employee_microservice** takes care of the car sales logic. This separation helped us focus on small parts of the system without being overwhelmed.

That said, it wasn’t always easy. Microservices bring complexity too — like handling communication between services, syncing data across multiple databases, and making sure messages don’t get lost or processed out of order. Debugging also got trickier, since problems could happen in one service but show up in another. Still, these challenges pushed us to learn a lot about how real distributed systems work and how to deal with them in practice.

5.2. Pros and cons of used patterns like CQRS etc.

We used several patterns to keep our system stable and reliable. One of the most important ones was CQRS (Command Query Responsibility Segregation), which helped us split the system into parts that write data and parts that only read data. This made it easier to keep things fast and safe — like having the customer-facing service only read data, while another service updates it in the background.

We also used idempotence, meaning the system doesn't freak out if the same request is sent more than once. That's super useful when things go wrong with the network or RabbitMQ retries a message. Immutable data was another thing we looked at — instead of deleting stuff from the database, we just marked it as deleted so we don't lose important info. And commutative message handling helped us deal with out-of-order messages, making sure only the newest update was actually used.

These patterns worked well, but they also made the project more complex. It took extra time to implement and test them properly, and we had to think more carefully about how everything would behave over time. But overall, they made the system much more reliable and gave us a deeper understanding of how big systems are built in the real world.

5.3. Scalability

Autoscaling is a key feature in modern cloud-native systems, allowing applications to automatically adjust their resources in response to changing demand. Instead of manually increasing or decreasing the number of running service instances, autoscaling handles this dynamically, ensuring both efficiency and reliability.

In our project, we implemented autoscaling using KEDA (Kubernetes Event-Driven Autoscaling). As seen in our Kubernetes manifests, we defined ScaledObject resources for services like the employee-microservice. These configurations instruct KEDA to monitor CPU usage and automatically scale the number of pods between 1 and 3, depending on the current workload:

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: cpu-scaledobject-employee
  namespace: kea-car-microservices
spec:
  scaleTargetRef:
    name: employee-microservice
  minReplicaCount: 1
  maxReplicaCount: 3
  triggers:
    - type: cpu
      metricType: Utilization
      metadata:
        value: "1"
```

This approach is known as horizontal scaling. Rather than making a single instance more powerful (vertical scaling), horizontal scaling increases the number of instances (pods) running in parallel. This method is highly effective for microservices architectures, as it allows the system to handle increased traffic and workload by distributing it across multiple replicas. It also improves fault tolerance and system availability.

By leveraging KEDA for autoscaling, our system can efficiently respond to real-time demand, maintaining performance and optimizing resource usage without manual intervention. This makes the platform more robust and scalable as user activity fluctuates.

5.4. Possible improvements

Although our system is already deployed using Kubernetes in a self-hosted environment, a natural improvement would be to fully migrate to a managed cloud platform such as AWS, Azure, or Google Cloud. This would provide easier scalability, managed services, and better infrastructure resilience. However, due to the budget constraints of a student project, we chose to simulate a cloud-like environment on a virtual machine. In a real-world scenario, switching to a full cloud deployment would bring improved reliability and operational efficiency.

Another improvement could be scaling the system more dynamically based on events—not just CPU usage. While we used KEDA to scale on CPU metrics, using queue length or message rate in RabbitMQ as a scaling trigger would give more responsive horizontal scaling for message-heavy services like the **employee_microservice** or **synchronizer_microservice.**