Berner Fachhochschule

Wirtschaft

# Proof of Concept Microservices

Development of a Microservice-based Point-of-Sale (PoS) System

Assignment 1 SDA

**vorgelegt von** Elia Emanuel Aeby, Michael Etter, Lenny Ernst Hurni, Marc Steiner, Sebastian Winjroks

**Experte/Expertin** Sid Singh, Sebastian Höhn

**Datum des Einreichens** 11. November 2024

# Table of Contents

# Introduction

## Objective of the Project

The goal of this Project is to implement a microservice-based Point-of-Sale system, Transforming an already existing monolithic architecture. The system is designed to manage the core functions of a retail shop. The task at hand is to decompose a monolithic database into several independent microservices, each responsible for a specific business domain managing its own database. This decomposition allows each microservice to operate independently, providing scalability, flexibility, and fault isolation.

Additionally, the project involves the integration of a Function-as-a-Service component. This report will detail the approach taken to achieve this architecture as well as discuss the technical challenges and trade-offs involved.

# Architecture and Data Decompositon

## Microservice Architecture

Microservice architecture is an architectural style that structures an application as a collection of small services, each focused on a specific function in the system. Each microservice operates independently and can be deployed, updated, and scaled without affecting the other services in the system.

## Comparision with Other Architectures

### Monolithic Architecture

In a monolithic architecture the entire system is built as a single, large unit. All components are integrated and deployed together. While it is simpler to develop initially, monolithic architecture becomes very hard to scale, maintain and update properly. A single issue in the system could crash the entire system and applying new small changes requires to redeploy the entire system.

### Service-Oriented Architecture

SOA is like microservices, but they tend to be generally larger, and the services interact through an ESB. SOA services require more complex middleware whereas microservices favor simpler communication methods.

## Data Decomposition

In microservice architecture, the monolithic database is split into multiple smaller databases. Each one of these belongs to a specific microservice. This Process is known as data Decomposition, and it is a very important step in creating a microservice based system.

Through Domain-Driven Design we gain a new way to approach complex systems. DDD introduces the concept of Bounded Contexts, which are important for effectively decomposing data in microservice architecture.

### DDD's role in Data Decomposition

DDD helps define clear boundaries between different business domains, such as customer management, orders, or inventory. Each of these domains form a Bounded Context, within which the data and business logic are consistent.

DDD also provides a strategic approach to decomposing the data by organizing it around business domains. This makes sure that each microservice is aligned with the business logic it is bound to support.

DDD introduces tactical design patterns, such as aggregates, entities, and repositories which help manage the data within a bounded context.

# Project Description

## Use Case: PoS System

In this section, we will illustrate how a customer interacts with our five Services to complete a transaction. This example demonstrates the flow from adding an item to the cart to completing a transaction.

### Process Overview

1. **Login**: User log in through the User Management Service
2. **Add to Cart**: Cart & Order Management Service updates the cart with selected items, pulling product details from the Product Management Service.
3. **Checkout**: Cart & Order Management Service creates an order and hands it over to the Sales & Payment Service.
4. **Payment**: Sales & Payment Service processes the payment.
5. **Post-Purchase**: Return & Refund Service manages any return, coordinating with the Sales & Payment Service for refunds and updating the User Management Service.

## Illustration

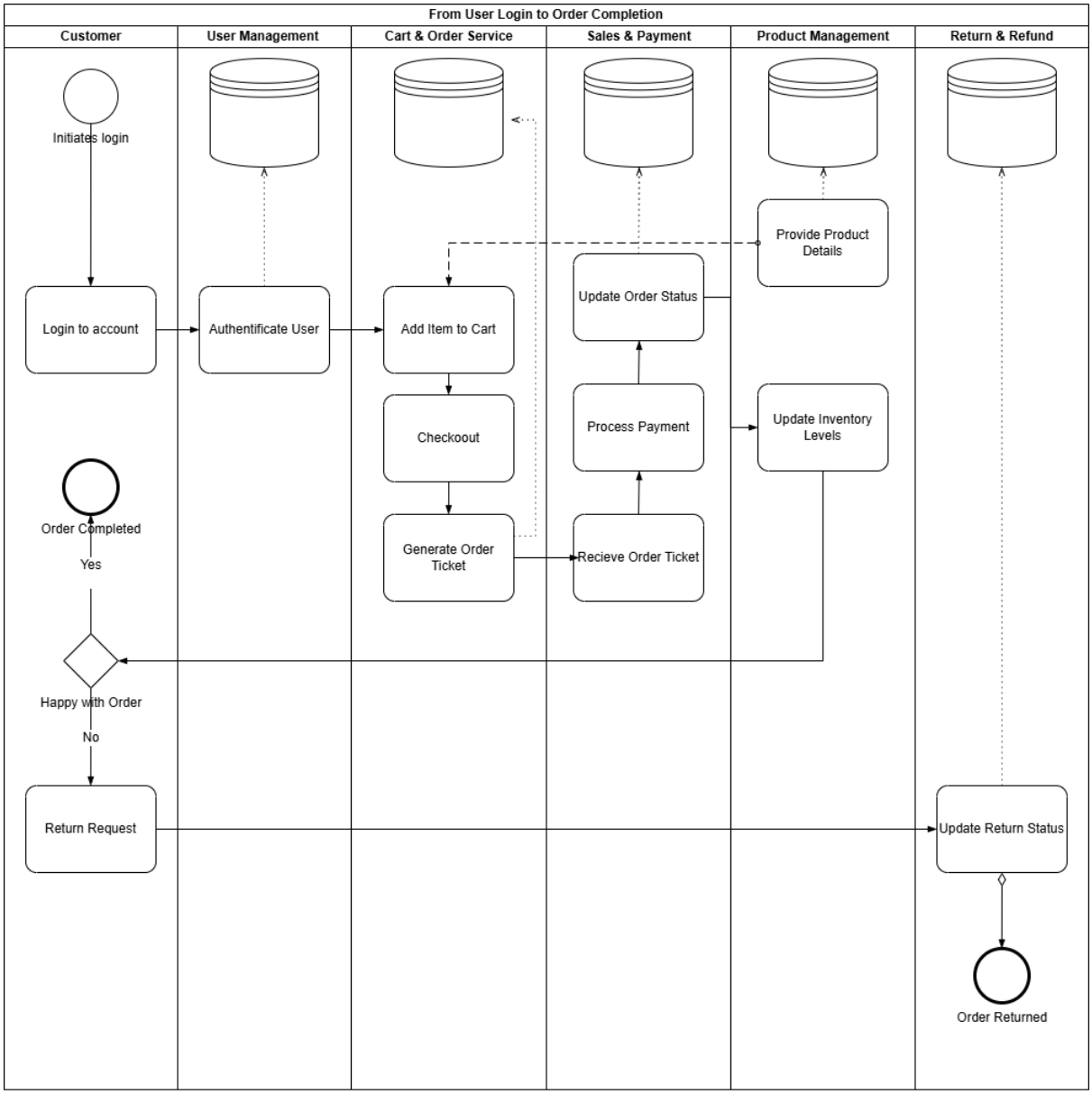


*Figure 1 BPMN From Login to Order Completion*

# Microservices and Data Decompostion

## Overview of the Microservices

### 1. User Management Service

- **Role:** Authentication, customer profile management.
- **Actions:**
  - Authenticate customer login.
  - Retrieve and manage customer data such as order history, rewards, and return status.

### 2. Cart & Order Management Service

- **Role:** Shopping cart operations, order creation.
- **Actions:**
  - Add items to the *cart* and manage cart state.
  - Generate an order ticket during checkout.
  - Update item quantities and manage the order lifecycle.

### 3. Sales & Payment Service

- **Role:** Handling payments and order processing.
- **Actions:**
  - Process payment based on the order ticket.
  - Update financial records and order status in the *ticket_system*.
  - Provide feedback on payment success or failure.

### 4. Return & Refund Service

- **Role:** Handling returns and refunds.
- **Actions:**
  - Process customer-initiated return or exchange requests.
  - Update the *return_table* and coordinate refunds or exchanges.
  - Inform the User Management Service to update the customer's account.

### 5. Product Management Service

- **Role:** Manage product inventory, details, and supplier information.
- **Actions:**
  - Provide product details for items added to the cart.
  - Adjust inventory levels based on purchases and returns.
  - Communicate with suppliers for stock replenishment if needed.

## Data Decomposition Explanation:

The original database structure was split into five microservices, this was done to improve scalability, maintainability and flexibility.

### 1. Product Management

**Tables**: *product_inventory, vendorinfo, orders, orders_ticket*

- **Reason**:
  - Product management involves inventory and supplier data.
  - This separation allows inventory updates, vendor management, and order tracking without affecting other services like sales or user management.

### 2. Cart and Order Management

**Tables**: *cart, cart_inprogress, item_list*

- **Reason**:
  - The cart system requires high performance to handle frequent updates as customers add or remove items.
  - Keeping the cart logic separate reduces load on the sales and payment database, ensuring a responsive user experience.

### 3. Return and Refund

**Table**: *return_table*

- **Reason**:
  - Return and refund operations are distinct processes involving post-sale customer service.
  - Isolating this ensures that return processing does not interfere with real-time sales transactions.

### 4. Sales and Payment

**Tables**: *gift_card, registers_table, tax_table, ticket_system*

- **Reason**:
  - Sales and payment data is highly transactional and requires a robust structure to handle financial operations securely.
  - By isolating this, the integrity of financial records is maintained, and sensitive data is better protected.

### 5. User Management

**Tables**: *customer_info, employee_info, stores*

- **Reason**:
  - User management involves sensitive personal data and authentication credentials.
  - Separating this service enhances security and allows specific optimizations for authentication and user data retrieval.

# Implementation Details

## Technologies and Tools Used:

### Adminer

Was used to gain access to our databases and our tables trough a Web interface

### Docker and MariaDB

Docker was utilized to run MariaDB databases as independent containers. This ensured that each microservice had its own separate database environment. This made it possible for us to set up and manage the databases without interference from the other services.

### Python

We chose Python as our programming language as we already have some experience with it.

### Flask

Flask is being used to build the APIs for each of our microservices. We decided to go for it because it is easy to integrate with python.

### Github

We decided to use Github for the sole purpose of having one repository where we can all contribute to. So, there is no need to send the file to everyone but instead we can just push and pull from that repository. It makes teamwork easier and more convenient.

## Implementation Process:

### Initial Discussion

As a group, we first reviewed all the materials provided. The goal was to fully understand the task at hand and to identify the core components of the system.
After discussing the scope and goals, we had a brainstorming session to figure out how we could split the system into microservices.

### Microservice Design

Initially, we considered implementing four microservices, each responsible for a major aspect of the system. However, as the process evolved and our understanding of the assignment got better, we decided to expand this to five microservices.

The five microservices we choose:

1. User & Employee Management Service
2. Product & Vendor Management Service
3. Cart & Order Management Service
4. Sales & Payment Service
5. Return & Refund Service

### Database Design and Decomposition

We then decided to split the monolithic database into smaller databases dedicated to the microservices we choose.
The goal is to make sure that every microservice has its own database, adjusted to the data it needs to manage.

# Trade-offs and Challenges

## Why Choose a Microservice Architecture:

Scalability: A microservice architecture allows different parts of the system to be scaled independently of each other. As an example, Product & Vendor Management Service module can be adapted to increased loads without affecting the database or other services.
Flexibility and modularity: As the services work independently of each other, they can be updated or changed without having to restart the entire system. Different technologies can also be used for each of the different services.
Isolation: If a microservice fails, the entire system is not affected. As an example, Sales & Payment Service could fail all the while the Cart & Order Management Service continues to function.
Collaboration: As the services work independently of each other and can also run on different technologies, different Teams can thus work on different services simultaneously, speeding up the development of the System as a whole.

## Trade-offs compared to other architecture

### Increased Complexity

A monolithic architecture is easier to manage because everything is in one codebase, and it usually interacts with one database. A microservice architecture on the other hand has many independent services, each with its own database and deployment process. This makes the system more complex because there is much more to manage at the same time.

### Data Consistency

Ensuring data consistency in microservices can be more of a challenge than in a monolithic system, where all the system components share the same database. In a system with separate databases for each service, it is hard to keep transactional consistency across the services.

## Challenges with Keeping a Single Database:

If the system were designed as a monolithic application, keeping a single shared database for all services, there would be significant challenges that could impact performance, scalability, and fault isolation.

### Performance and Scalability Bottlenecks:

A single database would become a bottleneck as more services start to interact with it. As different services grow in usage, the database would need to handle an increasing amount of traffic.
This results in performance degradation, as a single database can only handle a

specific amount of load. It also becomes difficult to scale the system as it is not always possible to scale specific components of the database without scaling the system.

### Failure Isolation:

In a monolithic architecture, failure in one part of the system can have devastating effects on other parts of the system. If the shared database becomes unavailable, it could bring down all services dependent on it.

## Conclusion

### Summary

In this project, we successfully transformed a monolithic PoS system into a microservice-based architecture. This was done so by decomposing a monolithic database into multiple independent services. Each of the microservices was designed to handle specific business domains, allowing independent development.

Using technologies such as Docker ensured isolated environments for development and testing, Flask provided us a framework for building our microservices. Github served as our collaborative platform which facilitated our teamwork.

This microservice transformation has set the stage for a more flexible, scalable and maintainable system.

# Appendices

Implemented Functions in the Architecture:

## Funktioniert

**USER**

1. GET: Alle Kunden abrufen

2. POST: Einen neuen Kunden erstellen

3. GET: Einen bestimmten Kunden abrufen

4. PUT: Einen bestimmten Kunden aktualisieren

5. DELETE: Einen bestimmten Kunden löschen

7. GET: Einen bestimmten Mitarbeiter abrufen

9. DELETE: Einen bestimmten Mitarbeiter löschen

10. POST: Einen neuen Store erstellen

**SALES**

2. POST: Ein neues Ticket erstellen

4. PUT: Ein bestimmtes Ticket aktualisieren

5. DELETE: Ein bestimmtes Ticket löschen

6. POST: Einen neuen Steuerdatensatz erstellen

7. GET: Alle Steuern abrufen

8. GET: Einen bestimmten Steuerdatensatz abrufen

9. PUT: Einen bestimmten Steuerdatensatz aktualisieren

10. DELETE: Einen bestimmten Steuerdatensatz löschen

1. GET: Alle Geschenkkarten abrufen

3. GET: Eine bestimmte Geschenkkarte abrufen

5. DELETE: Eine bestimmte Geschenkkarte löschen

1. GET: Alle Register abrufen

2. POST: Ein neues Register erstellen

3. GET: Ein bestimmtes Register abrufen

4. PUT: Ein bestimmtes Register aktualisieren

5. DELETE: Ein bestimmtes Register löschen

**RETURN REFUND**

2. POST: Eine neue Rückgabe erstellen

4. PUT: Eine bestimmte Rückgabe aktualisieren

5. DELETE: Eine bestimmte Rückgabe löschen

**PRODUCT MANAGEMENT**

1. GET: Alle Produkte abrufen

2. POST: Ein neues Produkt erstellen

3. GET: Ein bestimmtes Produkt abrufen

4. PUT: Ein bestimmtes Produkt aktualisieren

5. DELETE: Ein bestimmtes Produkt löschen

1. GET: Alle Lieferanten abrufen

3. GET: Einen bestimmten Lieferanten abrufen

1. GET: Alle Bestellungen abrufen

2. POST: Eine neue Bestellung erstellen

3. GET: Eine bestimmte Bestellung abrufen

4. PUT: Eine bestimmte Bestellung aktualisieren

5. DELETE: Eine bestimmte Bestellung löschen

**CART ORDER**

1. GET: Alle Einkaufswagen abrufen

2. POST: Einen neuen Einkaufswagen erstellen

4. PUT: Einen bestimmten Einkaufswagen aktualisieren

5. DELETE: Einen bestimmten Einkaufswagen löschen

1. GET: Alle Einkaufswagen in Bearbeitung abrufen

2. POST: Einen neuen Einkaufswagen in Bearbeitung erstellen

4. PUT: Einen bestimmten Einkaufswagen in Bearbeitung aktualisieren

5. DELETE: Einen bestimmten Einkaufswagen in Bearbeitung löschen

1. GET: Alle Artikel abrufen

2. POST: Einen neuen Artikel erstellen

# Funktioniert nicht:

**USER**

06. POST: Einen neuen Mitarbeiter erstellen

8. PUT: Einen bestimmten Mitarbeiter aktualisieren

11. GET: Alle Stores abrufen

12 GET: Einen bestimmten Store abrufen

13. PUT: Einen bestimmten Store aktualisieren

14. DELETE: Einen bestimmten Store löschen

**SALES PAYMENT**

1. GET: Alle Tickets abrufen

3. GET: Ein bestimmtes Ticket abrufen

2. POST: Eine neue Geschenkkarte erstellen

4. PUT: Eine bestimmte Geschenkkarte aktualisieren

**RETURN REFUND**

1. GET: Alle Rückgaben abrufen

3. GET: Eine bestimmte Rückgabe abrufen

**PRODUCT MANAGEMENT**

2. POST: Einen neuen Lieferanten erstellen

4. PUT: Einen bestimmten Lieferanten aktualisieren

5. DELETE: Einen bestimmten Lieferanten löschen

1. GET: Alle Bestelltickets abrufen

2. POST: Ein neues Bestellticket erstellen

3. GET: Ein bestimmtes Bestellticket abrufen

4. PUT: Ein bestimmtes Bestellticket aktualisieren

5. DELETE: Ein bestimmtes Bestellticket löschen

**CART ORDER**

3. GET: Einen bestimmten Einkaufswagen abrufen

3. GET: Einen bestimmten Einkaufswagen in Bearbeitung abrufen

3. GET: Einen bestimmten Artikel abrufen

4. PUT: Einen bestimmten Artikel aktualisieren

5. DELETE: Einen bestimmten Artikel löschen

## Glossary

| | |
|---|---|
| Bottleneck | A part of the system that limits overall performance due to overload |
| crash | A sudden failure of a system, causing it to stop functioning altogether |
| performance degradation | A slowdown in system performance under high load or over time |
| tight coupling | Components are dependent on each other |
| transactional consistency | Ensures all steps in a transaction are fully completed or rolled back |
| Bounded Context | A defined area within which specific business rules and data apply |

## List of abbreviations

| | |
|---|---|
| FaaS | Function-as-a-Service |
| PoS | Point-of-Sale |
| SOA | Service-Oriented Architecture |
| ESB | Enterprise Service Bus |
| DDD | Domain Driven Design |

# List of Illustrations