Title

Subtitle

Master Thesis

Submitted in partial fulfillment of the requirements for the degree of:

**Master of Science in Engineering**

to the University of Applied Sciences FH Campus Wien

Master Degree Program: Software Design and Engineering

Author:

First name surname

Student Identification Number:

Number

Supervisor:

Degree firstname name

Date:

dd. mm. yyyy

Declaration of authorship:

I declare that this thesis is my own work and that I did not use any aids other than those indicated or any other unauthorized help (e.g., ChatGPT or similar artificial intelligence-based programs). I certify that this work does not contain any personal data, and that I have clarified any copyright, license or image-law issues pertaining to the electronic publication of this thesis. Otherwise, I will indemnify and hold harmless the FH Campus Wien from any claims for compensation by third parties. I certify that I have not submitted this thesis (to an assessor for review) in Austria or abroad in any form as an examination paper. I further certify that the (printed and electronic) copies I have submitted are identical.

Date: Signature:

Abstract

This thesis investigates the impact of centralized configuration management using Spring Cloud Config Server on microservices-based architectures. It compares centralized and local configuration approaches with a focus on scalability, security, and maintainability.

List of Abbreviations

SOA Service-Oriented Architecture

URIs Uniform Resource Identifier

DEV Development Environment

PROD Production Environment

API Application Programming Interface

RBAC Role-Based Access Control

HTTP Hypertext Transfer Protocol

GRPC Google Remote Procedure Calls

DDD Domain-Driven Design

ETCD Distributed key-value store

Json JavaScript Objekt Notation

AWS Amazon Web Services

SSM Soft Systems Methodologie

WAF Web Application Firewall

TLS Transport Layer Security

MITM Man-in-the-Middle

JWTs JSON Web Tokens

JWKS JSON Web Key Sets

DDoS Distributed denial-of-service

RBAC Role-Based Access Control

SSO Single Sign-On

TSDB Time-Series Database

DVCS Distributed Version Control System

SQL Structured Query Language

NoSQL Not Only SQL

CI/CD Continuous Integration and Continuous Delivery

DVCS  Distributed Version Control System

MCP Model Context Protocol

JAR Compressed Archive Format

REST Representational State Transfer

DTOs Data Transfer Object

CRUD Create, Read, Update and Delete.

ORM Object-Relational Mapping

Contents

Table of Contents

[Abstract iii](#_Toc205119079)

[List of Abbreviations v](#_Toc205119080)

[Contents vii](#_Toc205119081)

[1. CHAPTER – INTRODUCTION 1](#_Toc205119082)

[1.1 Background and Motivation 1](#_Toc205119083)

[1.1.1 Introduce microservices 1](#_Toc205119084)

[1.1.2 why configuration management 3](#_Toc205119085)

[1.1.3 personal experience 3](#_Toc205119086)

[1.2 Problem Statement 4](#_Toc205119087)

[1.3 Research Objectives 5](#_Toc205119088)

[1.4 Research Question 5](#_Toc205119089)

[2. CHAPTER – LITERATURE REVIEW 6](#_Toc205119090)

[2.1 Microservices Overview 6](#_Toc205119091)

[2.1.1 Monolith to Microservices 7](#_Toc205119092)

[2.1.2 Monolith to Microservices: 5 Strategies 9](#_Toc205119093)

[2.1.3 Key characteristics of microservices 10](#_Toc205119094)

[2.1.4 Impacts of Migration to Microservice Architecture on Team 11](#_Toc205119095)

[2.2 Challenges in Microservices 12](#_Toc205119096)

[2.3 Centralized vs. Local Configuration 14](#_Toc205119097)

[2.4 Security and Observability Considerations 16](#_Toc205119098)

[2.4.1 Security 16](#_Toc205119099)

[2.4.2 Observability 17](#_Toc205119100)

[3. CHAPTER – RESEARCH METHODOLOGY 19](#_Toc205119101)

[3.1 Research Design 19](#_Toc205119102)

[3.2 Tools and Technologies 20](#_Toc205119103)

[3.3 Evaluation Metrics 22](#_Toc205119104)

[3.3.1 Scalability 22](#_Toc205119105)

[3.3.2 Security 22](#_Toc205119106)

[3.3.3 Maintainability 22](#_Toc205119107)

[4. Chapter - SYSTEM DESIGN 23](#_Toc205119108)

[4.1 Architecture Overview 23](#_Toc205119109)

[4.2 Implementation Approach 24](#_Toc205119110)

[4.3 Configuration Strategy 33](#_Toc205119111)

[4.3.1 API 33](#_Toc205119112)

[4.3.2 Docker 38](#_Toc205119113)

[4.3.3 Configserver 40](#_Toc205119114)

[4.3.4 Client-Side Service Discover 42](#_Toc205119115)

[4.3.5 RESILIENCY 44](#_Toc205119116)

[4.3.6 OBSERVABILITY AND MONITORING 46](#_Toc205119117)

[4.3.7 OpenID Connect, KeyCloak 48](#_Toc205119118)

[4.3.8 Event-driven microservices 50](#_Toc205119119)

[4.3.9 Kubernetes 52](#_Toc205119120)

[4.3.10 Helm 54](#_Toc205119121)

[5. CHAPTER – EXPERIMENTATION & RESULTS 56](#_Toc205119122)

[5.1 Performance and Maintainability 56](#_Toc205119123)

[5.1.1 Local Configuration 56](#_Toc205119124)

[5.1.2 Centralized Configuration 61](#_Toc205119125)

[5.1.3 Docker compose 63](#_Toc205119126)

[5.1.4 Spring Boot BOM and Shared Libraries 65](#_Toc205119127)

[5.1.5 Result 67](#_Toc205119128)

[5.2 Security 69](#_Toc205119129)

[5.2.1 KeyCloak 69](#_Toc205119130)

[5.2.2 Docker Compose 71](#_Toc205119131)

[5.2.3 Result 72](#_Toc205119132)

[6. CONCLUSION 75](#_Toc205119133)

[Bibliography 78](#_Toc205119134)

[List of Figures 79](#_Toc205119135)

[List of Tables 80](#_Toc205119136)

[Appendix 81](#_Toc205119137)

1. CHAPTER – INTRODUCTION

## Background and Motivation

### Introduce microservices

A decade ago, applications were typically deployed as a cohesive unit, with all functionalities integrated and hosted on a single server. This architectural style, referred to as Monolithic, presents advantages such as simplified development and deployment for smaller teams, enhanced performance, and reduced cross-cutting concerns. Nevertheless, it also has drawbacks, including restricted agility, challenges in adopting new technologies, and reliance on a single code base. Various types of Monolithic architectures exist, such as Single-Process Monolith, Modular Monolith, and Distributed Monolith. To tackle the issues associated with large, monolithic applications, Service-Oriented Architecture (SOA) was developed, which organizes software systems into interoperable services. SOA provides advantages like reusability, improved maintainability, increased reliability, and the ability for parallel development; however, it also entails disadvantages such as complex management, significant investment costs, and additional overhead. In contrast, Microservices represent a model of independently deployable services centred around a business domain, facilitating the construction of more intricate systems. [1].

Considering the increasing intricacy of software systems and a relentless quest for agility, monolithic architectures have begun to reveal their limitations. Microservices architectures, which emphasize independent and self-sufficient services, offer a compelling alternative by providing enhanced scalability, faster deployment cycles, and improved maintainability. However, transitioning to this fragmented model presents numerous challenges. One of the most significant obstacles is the disintegration of existing monoliths into cohesive microservices. Precisely identifying microservice boundaries and functional responsibilities within a monolithic software system is a vital task, yet it frequently proves to be challenging. Various approaches have been devised to assist in this endeavor, examining features, dependencies, and execution patterns to potentially outline well-defined microservices. Despite these advancements, a comprehensive understanding of the strengths, weaknesses, and ongoing challenges related to current decomposition strategies remains elusive. This systematic literature review aims to fill this knowledge gap. Through a rigorous methodology, we systematically compile, analyze, and synthesize research contributions on monolith decomposition, with a specific focus on techniques for identifying microservices. Our investigation explores the research objectives, evaluation methods, and enduring challenges that characterize this field, with the goal of establishing a robust classification of decomposition approaches and identifying avenues for further refinement. [2].

The microservices architecture started to attract attention following the publication of several success stories from companies such as Netflix, Gilt.com, and Amazon. However, all these companies, along with many other successful microservices implementations, had one commonality — they originated from web-based companies that were either creating new applications or did not possess a significant legacy code base to overhaul. When a traditional corporation transitions to microservices, one challenge they encounter after selecting the initial green-field applications to explore microservices is that certain principles of the microservices architecture, especially the "Decentralized Data Management" and "Decentralized Governance" principles, are challenging to implement when it is necessary to refactor a large monolithic application. Fortunately, a solution to this challenge has existed for several years in the form of a pattern that Martin Fowler first documented in 2004, several years before his work on microservices. This concept is known as the "strangler application pattern," which aims to tackle the reality that one rarely operates in a green field. The applications that require microservices the most are often the largest and most complex on the web; however, leveraging the architecture of the web can offer a strategy for managing the necessary refactoring. The strangler application is a straightforward concept based on the analogy of a vine that constricts the tree it envelops. The premise is to utilize the structure of a web application — the fact that it is composed of individual URIs that correspond functionally to various aspects of a business domain — to divide an application into distinct functional domains and replace those domains with a new microservices-based implementation one domain at a time [3].

The Strangler Fig Pattern is a software migration pattern used to gradually replace or refactor a legacy system with a new system, piece by piece, without disrupting the existing functionality. This pattern gets its name from the way a strangler fig plant grows around an existing tree, slowly replacing it until the original tree is no longer needed. When to Use the Strangler Fig Pattern: When you need to modernize a large or complex legacy system. When you want to avoid the risk associated with a complete system rewrite or "big bang" migration. When the legacy system needs to remain operational during the transition to the new system shown in Figure 1.

A group of trees with branches

AI-generated content may be incorrect.

Figure 1 .Strangler Fig Pattern

### why configuration management

Configuration management plays a vital role in the effective operation of microservices, encompassing elements such as environment variables, feature flags, service endpoints, and rate limits. Poorly managed configurations can result in downtime, erratic behavior, or security vulnerabilities. The difficulties associated with configuration management include decentralization, environment-specific settings, dynamic updates, and security issues. Decentralized storage may cause inconsistencies, while environment-specific configurations necessitate distinct environments for development, staging, and production. Safeguarding sensitive data, such as API keys or passwords, demands particular attention to prevent breaches. It facilitates dynamic updates without requiring redeployment or service restarts, thereby ensuring minimal disruption and a seamless rollout of settings across various microservice instances. Centralized configuration management systems, such as Spring Cloud Config, enable dynamic configuration updates to maintain service consistency, thereby preventing drift between environments and instances. Utilizing central repositories like Git or Consul in conjunction with Spring Cloud Config streamlines configuration management processes, making it easier to track changes and conduct audits. Environment-specific profiles (for instance, dev, staging, prod) are managed efficiently from a centralized location, and automatic configuration refresh (through Spring Cloud Bus, Kafka, or RabbitMQ) disseminates updates in real time to all clients, directly enhancing maintainability, operational efficiency, and system scalability. [4].

### personal experience

After working at Alliance for three and half years on a substantial application related to Insurance, I have come to understand how a large community application can encounter issues from various angles.

One significant challenge is managing a large team, which can be difficult, and finding someone who possesses comprehensive knowledge of the entire application is equally challenging. If you consider a single line of code, it can be problematic; even if there are just one or two lines, updating them can impact every aspect of the application. Consequently, extensive testing is required, which can lead to significant delays.

At one point, we decided to transition to microservice architecture, and while we implemented it, we were unable to convert the entire application. The business layer, which interfaces with the database, remained monolithic, while we transformed other components into microservices. Despite this, we observed improvements in performance and latency, and we were able to organize our services into smaller teams. Each team included roles such as a Scrum Master, designer, product owner, and testers, allowing us to work concurrently in different programming languages. However, we still face the limitation of having a single database. For instance, adding a new column to the database required a lot of changes, highlighting the challenges of analytics applications.

Additionally, microservices come with their own set of issues. In my view, if a project can be divided into three or four microservices, it may be better to avoid microservices altogether. However, if the project necessitates more than five microservices,

I recommend pursuing that route. Another aspect I wish to highlight is that when utilizing microservices, you have the flexibility to select the programming language for each service. This is one of the significant advantages of microservices, a freedom that is not available in monolithic applications.

## Problem Statement

In a distributed system, especially one built on microservices architecture, managing configurations can become complex and challenging. Each service typically requires its own set of configurations, including database connections, API keys, feature flags, and environment-specific settings (e.g., development, staging, production). As the number of services grows, keeping track of all these configurations across multiple environments can quickly become unmanageable. That’s why we should use a centralized configuration [3].

Without centralization, changes needed to be replicated manually, increasing maintenance overhead and reducing flexibility. A dedicated configuration service improved consistency and simplified the deployment pipeline.

In a typical microservices system, each service has its own configuration file. While this seems simple at first, it causes big problems in large, distributed systems:

Challenges in Configuration Management

1. Decentralization: With multiple services, storing and managing configurations locally for each service can lead to inconsistencies.
2. Environment-Specific Configurations: Development, staging, and production environments require different configurations.
3. Dynamic Updates: Certain configurations, like feature toggles or throttling limits, may require runtime updates.
4. Security Concerns: Storing sensitive information like API keys or passwords needs special care to avoid breaches.

These issues lead to higher operational effort, more downtime, and potential vulnerabilities. Centralized configuration — where all services pull their settings from one place — promises to solve these challenges, but it’s still unclear how much of an impact it really makes in practice.

This thesis will explore that question through a hands-on comparison.

## Research Objectives

The main goal of this thesis is to assess the effects of centralized configuration management on microservices architecture through the implementation of Spring Cloud Config Server. Specifically, this research aims to investigate how the centralization of configuration data influences three critical operational factors: scalability, security, and maintainability.

In microservices environments, decentralized or per-service configuration can lead to redundancy, inconsistency, and operational overhead—particularly when dealing with large-scale distributed systems. This study aims to illustrate how externalizing configurations to a centralized service can enhance scalability by facilitating easier deployment of changes, mitigate security risks through centralized secret management and access control, and improve maintainability by streamlining configuration updates and auditing.

The research will establish two configurations for comparison: one utilizing local per-service configurations and the other employing centralized configuration via Spring Cloud Config Server. These configurations will subsequently be assessed based on established metrics, including deployment complexity, response time during configuration updates, RBAC enforcement, and the operational effort required to maintain configurations.

## Research Question

In what ways does centralize configuration management through Spring Cloud Config Server enhance scalability, security, and maintainability within microservices architectures when compared to local configuration management? The sub-questions are as follows: To what degree does centralized configuration alleviate the operational challenges associated with configuration changes? How does it facilitate the secure management of environment-specific variables and sensitive information? What effect does it have on the capacity to scale services independently and uniformly across different environments? By exploring these inquiries, the thesis seeks to offer practical insights and an empirical assessment of configuration strategies in distributed systems.

Alongside these fundamental inquiries, the thesis investigates the wider ecosystem necessary for facilitating effective configuration management. It analyzes the ways in which Docker enhances configuration portability, how Kubernetes facilitates dynamic updates to configurations and manages secure secret handling, and how tools such as Spring Cloud Vault and Config Maps play a role in safeguarding sensitive information. Furthermore, the research delves into how centralized configuration bolsters Develop

and operation ops automation, influence’s fault tolerance and system resilience, and aids in the monitoring, auditability, and governance of configuration modifications across distributed microservices.

# CHAPTER – LITERATURE REVIEW

## Microservices Overview

Microservice architecture has increasingly emerged as the favored approach for creating distributed systems and large-scale applications in recent years. Unlike a monolithic structure, microservice architecture is defined by its loose coupling, with each service functioning independently of the others. Like a monolith, microservice includes business logic and local data storage, but it is specifically designed for single-purpose services only. This differentiation distinguishes microservices from monolithic architectures, which consolidate an entire application within a single service. As a result, a collection of microservices, each operating independently, functions as a suite of smaller services. The independent operation of these services facilitates maintenance due to their manageable size and organization, ensuring that if one microservice fails, the entire system remains operational. This design promotes high replaceability of components, allowing autonomous teams to perform maintenance and updates independently in most cases. Given the structural design of microservices and their independent operation, they depend significantly on internal communication among the services. A microservice is language-agnostic, meaning that components should not interact through language-specific functions or method calls (Microsoft, 2021). This poses a challenge when transitioning from a monolithic to a microservice architecture, as direct conversion via method calls is considered inefficient. Consequently, microservices employ inter-service communication protocols, such as HTTP or gRPC, for interaction. The architectural transformation is undergoing entails moving away from the legacy code base and modifying business areas to capitalize on the new opportunities offered by the updated digital architecture. Fowler observed that organizations with existing monolithic architecture would benefit from this transition before adopting microservice architecture, emphasizing that it would be challenging to develop applications from [6].

A diagram of a software company

AI-generated content may be incorrect.

Figure 2: Microservices Overview

### Monolith to Microservices

Monolithic, has pros such as simpler development and deployment for smaller teams, better performance due to no network latency, and a single code base. Cons include limited agility, difficulty in adopting new technologies, and a single code base.

SOA emerged as an approach to address the challenges of large, monolithic applications by organizing software systems as a collection of interoperable services. This approach offers benefits such as reusability, better maintainability, higher reliability, parallel development, and complex management due to communication protocols. However, it also has cons such as high investment costs and extra overload.

Microservices, independently releasable services modelled around a business domain, are easier to develop, test, and deploy, increase agility, and scale horizontally. They can represent various services, such as Accounts, Cards, and Loans, but together they could form an entire bank system.

A diagram of a computer service

AI-generated content may be incorrect.

Figure 3 : Monolithic vs. SOA vs. Microservices

|  |  |  |  |
| --- | --- | --- | --- |
| Aspect | Monolithic Architecture | Service-Oriented Architecture (SOA) | Microservices Architecture |
| Scope | Single, unified application | Broad architectural style with reusable enterprise services | Focused on independently deployable services |
| Size of Services | One large application or module | Larger, domain-specific services (e.g., business process layers) | Small, focused, single-purpose services |
| Data Management | Shared, centralized database | Often shared databases across services | Each service has its own database (Database-per-Service) |
| Communication | Internal method calls | Standard protocols (e.g., SOAP over ESB) | Lightweight RESTful APIs, messaging (Kafka, RabbitMQ) |
| Technology Diversity | Limited to one tech stack | Standardized stack with limited flexibility | Freedom to use different tech stacks per service |
| Deployment | Entire app is deployed together | Services deployed independently, but often tightly integrated | Fully independent deployments per service |
| Scalability | Entire application scales as a whole | Can scale services, but often at coarse level | Each microservice scales independently |
| Development Speed | Slower due to tight coupling | Moderate, depending on service size and dependencies | Faster due to small, independent components |
| Flexibility | Limited by monolithic nature | More flexible, but changes can affect multiple services | Highly flexible; services evolve independently |
| Team Structure | Large, centralized development team | Multiple teams, usually around domains or layers | Small, cross-functional teams per service |
| Maintenance | Complex and error-prone with tight coupling | Easier than monoliths but still involves coordination | Easier, isolated maintenance of smaller codebases |

Table 1: Monolithic vs. SOA vs. Microservices Comparison

### Monolith to Microservices: 5 Strategies

1. Incremental Refactoring

Incremental refactoring denotes the gradual conversion of a monolithic system into a microservices architecture. This methodology facilitates the stepwise breakdown of a monolith into microservices, thus reducing the likelihood of business interruption. In the process of incremental refactoring, the first step is to pinpoint the elements of the monolith that are most suitable for transformation into independent microservices. These elements may consist of functionalities that are relatively detached from the rest of the system or those that would benefit significantly from the advantages offered by microservices, including improved scalability and faster deployment. [6].

2. Strangler Pattern

The strangler pattern represents a strategy that entails the gradual replacement of segments of a monolithic application with microservices while the monolith continues to operate. This pattern draws inspiration from the strangler fig tree, which envelops other trees and gradually supplants them.

The strangler pattern facilitates the incremental introduction of microservices into your system without interrupting the operation of the monolith. This methodology mitigates risk and promotes a more seamless transition process [6].

3. Decomposing by Business Capability

This strategy focuses on dismantling a monolith into microservices based on business functionalities. This approach aligns the technical elements of your system with your business goals, simplifying the management and evolution of your system in response to business demands.

When decomposing by business capability, it is crucial to ensure that each microservice is accountable for a singular business capability. This practice helps preserve the independence of microservices and diminishes the complexity of the system [6].

4. Anticorruption Layer (ACL)

The anticorruption layer (ACL) is a strategy employed to guarantee that the transition from a monolith to microservices does not compromise the business logic of your system. The ACL serves as a protective barrier between the monolith and the microservices, facilitating the conversion of data and requests between the two systems.

Utilizing an ACL can assist in ensuring that the integrity of the business logic is maintained throughout the transition process [6].

5. Domain-Driven Design (DDD)

Domain-driven design (DDD) is a software development approach that focuses on understanding the business domain and using this understanding to guide the design and

development of software. In the context of transitioning from monolith to microservices, DDD can be used to identify the boundaries of microservices and to ensure that the transition process aligns with business goals. [6].

### Key characteristics of microservices

Microservices are an increasingly popular approach to building and deploying software applications. This architectural style involves breaking down an application into a set of independent services that can be developed, deployed, and maintained separately. The goal of microservices is to make software development more agile and scalable, allowing teams to release new features and updates quickly and efficiently [6].

1. Componentization via Services: Component is a unit of software that is independently replaceable and upgradeable.
2. Organized around Business Capabilities: The microservice approach to division is splitting up into services organized by business capability.
3. Products not Projects: This is Amazon’s notion of “you build, you run it” where a development team takes full responsibility for the software in production.
4. Smart endpoints and dumb pipes: Microservices aim to be as decoupled and as cohesive as possible, so they own their own domain logic and receive a request, applying logic and producing a response with using Restful APIs.
5. Decentralized Governance: Netflix is a good example of an organization that follows this philosophy. Sharing useful and all tested code as libraries encourages other developers to solve similar problems in similar ways.
6. Decentralized Data Management: That means Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems.
7. Infrastructure Automation: That means automate deployment to each new environment and for every microservice separately.
8. Design for failure, Resilience: Microservices design by dealing with failures and try to manage failures with managing errors with proper actions. Microservices are also designed to be resilient, meaning that they can continue to operate even if one or more services fail
9. Scalable: Each service operates independently, it is possible to scale individual services up or down as needed, without affecting the rest of the application. This allows teams to allocate resources more efficiently and ensure that the application can handle increased traffic or usage.
10. Technology Agnostic: Different services can be written in different programming languages or use different technology stacks.

### Impacts of Migration to Microservice Architecture on Team

Overall, the role of the Product Owner has been significantly influenced by the transition from a monolithic to a microservice architecture. As a representative of customer and product interests, the Product Owner has prioritized enhancing the frequency of software deliveries. The shift to microservice architecture has markedly improved the continuous delivery process. Unlike before, when new functionalities were deployed to customers in fixed deployment cycles, even at the conclusion of each Sprint, they are now delivered continuously throughout a Sprint. Delivery to the customer has been established as a criterion for the completion of User stories and has been integrated into the Definition of Done. This increased frequency of deployments has resulted in more regular interactions with customers, thereby enhancing agility [7].

In comparison to the Product Owner, the Scrum Master indicated that the migration from a monolithic to a microservice architecture resulted in a reduced number of changes. The Scrum Master primarily highlighted the importance of communication and coordination among teams, as well as the necessity for enhanced motivation among team members.

The Scrum Master underscored that decomposing the application into smaller, independent units facilitated the allocation of application responsibilities across teams and contributed to achieving cross-functionality within the team. This approach also mitigated issues related to inter-team dependencies, ultimately leading to improved efficiency.

Additionally, the Scrum Master noted that the ability to articulate Sprint goals with greater precision has bolstered team member motivation throughout the Sprint. However, the Scrum Master also acknowledged the complexities involved in transitioning to a microservice architecture.

Regarding Scrum ceremonies and artifacts, the Scrum Master did not perceive any significant changes from his perspective. Specifically, while the Sprint retrospective was discussed in greater detail, it remained fundamentally unchanged according to the Scrum Master [7].

The Impacts of Migration to Microservice Architecture on the Developer Role

Naturally, the technological dimensions of the migration from monolithic to microservice architecture were predominant in the interviews conducted with developers. They candidly expressed the challenges that accompanied the migration process, emphasizing that issues related to deployment, operation, and monitoring should not be underestimated. Moreover, the DevOps concept and the associated automation of processes were identified as critical. It became essential to fulfil the heightened demands for knowledge, experience, and technical expertise within the development team.

Conversely, microservices facilitated a more manageable workflow for development teams and reduced the risks associated with the creation of new functionalities. The process of adding or replacing individual microservices proved to be significantly simpler than redeploying the entire monolithic system [7].

## Challenges in Microservices

**Complexity**

Microservices offer flexibility and modularity. However, development teams tend to face many challenges, including service communication, data consistency, and distributed system management.

Developing, operating, and managing an application based on microservices requires specific expertise, tools, and sophisticated monitoring and orchestration functionalities. Organizations must allocate resources towards infrastructure, automation, and DevOps methodologies to effectively manage the intricacies linked to microservices. Distributed System Challenges [7].

In the microservice architecture, communication between services happens via a network, which leads to increased latency, networking overhead, and potential failure points.

Ensuring dependable communication, managing network disruptions, and preserving data consistency across distributed services can be challenging. Organizations need to establish robust communication strategies, including circuit breakers, retries, and protocols, to address these issues. [7].

**Operational Overhead**

The operational overhead associated with running many microservices in production environments is huge. Examples of tasks that become more complicated in a distributed system are monitoring, logging, debugging, and tracing.

Organizations necessitate dependable surveillance and insight into potential threats to acquire an understanding of the well-being and effectiveness of specific services and the overall system. Additionally, managing service dependencies, version control, and ensuring backward compatibility contribute to the operational intricacy of microservices.[7].

**Data Management**

In a microservices architecture, every service possesses its own data store, which may result in data duplication, consequently causing inconsistencies and synchronization challenges. Ensuring data consistency within distributed systems necessitates careful design and execution of data management strategies such as event sourcing, eventual consistency, and distributed transactions. Organizations must diligently oversee data access and uphold data integrity to avert data corruption and associated issues [7].

**Service Discovery and Communication**

Microservices are required to dynamically discover and communicate with one another. Consequently, an effective service discovery mechanism is essential. The management of service endpoints, load balancing, and failover among distributed services presents significant challenges. Organizations ought to implement service registry and discovery solutions, such as Consul or Eureka, to facilitate communication between services. Additionally, robust communication patterns, such as service mesh architectures, enhance reliability and fault tolerance [7].

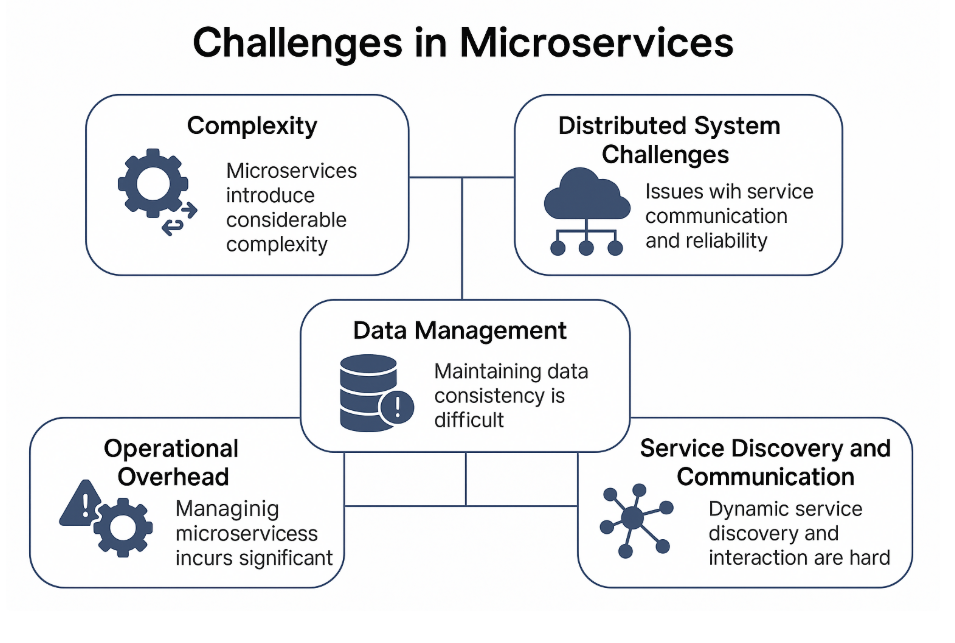


Figure 4: Challenges in Microservices

## Centralized vs. Local Configuration

1. Configurations play a crucial role in the operation of microservices. They include Environment Variables: Such as database URLs, API keys, and credentials. Feature Flags: Enabling or disabling features dynamically. Service Endpoints: The URLs for interaction with other services. Rate Limits: Regulating usage to avert system overload. Inadequately managed configurations can lead to downtime, unpredictable behaviour, or security vulnerabilities. Thus, it is essential to implement a robust strategy. Challenges in Configuration Management include Decentralization: With a multitude of services, the local storage and management of configurations for each can lead to inconsistencies. Environment-Specific Configurations: Distinct configurations are required for development, staging, and production environments. Dynamic Updates: Certain configurations, like feature toggles or throttling limits, may necessitate updates during runtime. Security Concerns: The handling of sensitive information such as API keys or passwords demands careful management to avoid breaches. Key Practices for Configuration Management include Externalize Configurations. Avoid embedding configurations directly within your application code. Instead, use configuration files, environment variables, or configuration management tools to externalize settings. This approach guarantees consistency across different deployments and environments [7].

2.Centralized Configuration Management

Implement centralized configuration management systems such as Consul, etcd, or Spring Cloud Config. These tools store configurations in a central repository, allowing services to dynamically access their settings [7].

Advantages:

* Consistency across services.
* Simplified updates without requiring service redeployment.
* Secure access control.

3. Utilize Environment-Specific Configurations

Maintain separate configuration files or entries for each environment (e.g., config.dev.json, config.prod.json). This approach aids in avoiding the inadvertent deployment of incorrect settings.

Do we genuinely need to externalize? It seems we are opening a Pandora's box in this situation. Let us evaluate the pros and cons of having my configuration file (e.g., config.json) in conjunction with my Docker image [7].

|  |  |  |
| --- | --- | --- |
| Configuration Type | Advantages / Disadvantages | Details |
| Embedded | Advantages | Easy to understand |
| Embedded | Advantages | Simplifies testing configuration for a specific state within the codebase |
| Embedded | Advantages | Local development is very convenient to initiate |
| Embedded | Advantages | Local changes to the configuration file do not impact other developers. |
| Embedded | Advantages | Deployment is uncomplicated. |
| Embedded | Disadvantages | Secrets are exposed in the Git repository, which is not ideal. (Mitigation: AWS SSM) |
| Externalized | Advantages | Changing values is quick. |
| Externalized | Advantages | Solutions exist to poll for changes and apply them without needing to restart the container. |
| Externalized | Advantages | Shared configuration among services can be established in a single location. |
| Externalized | Disadvantages | There is uncertainty regarding how local development functions. |
| Externalized | Disadvantages | What happens if I am modifying values during development? |
| Externalized | Disadvantages | Do other individuals or services notice this private change? |

Table 2: Centralized vs. Local Configuration

We will require a method to clone the configuration locally. More challenging to validate that a configuration change does not disrupt a service. Do we need to implement a rollback of configuration values? What if my configuration service is down, and my service cannot retrieve my external configuration? Taking a step back, what configuration values do we typically store, and what types of values would we modify at runtime? Let us consider, one could store the following information: Database Connection Information Timeout Values Service URLs https://service-a.com (yes, there is service discovery, but still, some URL needs to be stored) Feature Flags Other Constants? Which of these would we want to adjust post-deployment? Feature Flags, but we would want this to be persisted across deployments. Timeout values could be utilized for experimentation. However, they might be temporary [7].

## Security and Observability Considerations

### Security

Security in microservices goes far beyond a perimeter firewall. In a distributed system with multiple independent services, it’s critical to implement defense-in-depth strategies that address identity, data flow, access control, and observability across all layers.

A key element of securing microservices is the API Gateway. Since microservices expose multiple endpoints, a gateway acts as a centralized access control point. It manages authentication, applies authorization policies, and protects against common threats using WAF.

Gateways like Amazon API Gateway or Spring Cloud Gateway consolidate access, reduce the attack surface, and ensure that traffic is properly filtered before reaching any internal services [8].

Even within private networks, assuming that internal communication is secure by default is a mistake. Microservices systems should adopt a Zero Trust approach, where internal service-to-service communication is encrypted using TLS. For enhanced identity verification and resistance against MITM attacks, mTLS is recommended. In mTLS, both services authenticate each other before any data exchange occurs [8].

For access control, microservices typically rely on authentication (who you are) and authorization (what you’re allowed to do). Real-world implementations often combine several access control models:

* Role-based access control for grouping user/service permissions.
* Attribute-based access evaluates conditions at runtime.
* Policy-based access based on defined business logic.
* Relationship-based access considering hierarchies and ownership.

In many cases, no single model is sufficient. Secure microservices systems blend these approaches, assigning unique identities to each service and limiting permissions according to the principle of least privilege.

To reduce load on authentication servers and improve response time, many architectures use JWTs. JWTs encode user identity and permissions, allowing services to validate them locally using JWKS without needing round-trip validation on every request. While efficient, JWTs should be short-lived or revocable to avoid stale or overly permissive tokens.

Rate limiting and DDoS protection are also essential, especially for public APIs. Techniques such as IP throttling, API key restrictions, and behavioral analysis can prevent malicious or accidental service overloads. These protections help maintain uptime and performance underload [8].

Internally, many systems now use service meshes like Istio or Linked to enforce security policies and route traffic. These tools use sidecar proxies to manage service discovery, mTLS enforcement, and telemetry collection.

They also provide observability features like traffic shaping, tracing, and access control — all essential for secure operations.

Secrets management is another foundational layer. API keys, database credentials, and tokens must never be hardcoded. Instead, secrets should be stored in dedicated tools like Hashi Corp Vault, AWS Secrets Manager, or Doppler. Secrets should be rotated regularly and scoped to the smallest set of permissions needed.

Lastly, a security system must be observable. Distributed tracing tools like Open Telemetry allow you to assign a unique ID to each request and trace it across multiple services. When combined with log aggregation platforms (e.g., Datadog, Splunk), these traces help detect suspicious patterns, debug failures, and respond to incidents quickly.

Together, these practices create a resilient and secure microservices architecture. They not only prevent unauthorized access and breaches but also ensure that incidents are detected early and mitigated efficiently [8].

### Observability

In a microservices environment, observability is all about knowing what’s happening inside your system — even when it’s made up of dozens or hundreds of small, independent services. The goal is to gain visibility into the internal state, performance, and health of your distributed application. To do that effectively, developers and operations teams rely on a set of observability patterns that provide actionable insights into system behavior.

Logging is the most common and foundational observability practice. Every Micro Service typically generates its own logs, recording key events, errors, and informational messages. These logs are then collected by a centralized logging service (e.g., ELK Stack, Loki) and sent to a searchable analytics tool.

This setup makes it possible to trace how one event flows through multiple services. For instance, if one service logs an error, centralized logs let you quickly check whether that error was triggered by an upstream service or caused a downstream failure [9].

**Application Metrics Pattern**

In addition to raw logs, metrics offer elevated numerical insights regarding system performance, such as CPU utilization, memory usage, response durations, or error frequencies. Metrics can be gathered from both specific microservices and the underlying infrastructure on which they operate.

For instance, if a particular service begins to consume more CPU than usual, effective metrics collection allows for immediate identification of whether this is a singular problem or indicative of a larger trend. Tools such as Prometheus, particularly when combined with Kubernetes, furnish this level of visibility and can even initiate alerts in the event of anomalies. [9].

**Distributed Tracing Pattern**

Distributed tracing monitors an individual user request as it moves through various microservices. This technique is particularly beneficial for identifying performance bottlenecks and determining the locations of failures within intricate systems.

For instance, when a user encounters an error, logs may indicate which service generated the report, yet they often fail to explain the underlying cause. A trace provides a comprehensive view of the request's journey, emphasizing which service experienced delays or triggered the error. Tools such as Open Telemetry, Jaeger, and Zipkin are frequently employed for tracing purposes. [9].

**Exception Tracking**

While logs and metrics provide an overview of general behavior, exceptions are instrumental in pinpointing specific application-level errors. Exceptions arise when the code does not function as anticipated as in the case of a failed database call or a null pointer.

Monitoring exceptions enables you to differentiate between infrastructure issues (such as a full disk and genuine bugs within the code. After isolating the service and the method that triggers the exception, developers can more efficiently debug and resolve the problem [9].

**Health Check APIs**

Each microservice ought to provide a health check endpoint that indicates the operational status of the service. These APIs offer valuable information regarding uptime, latency, error rates, and additional metrics.

Health checks serve a purpose beyond human oversight orchestration tools such as Kubernetes depend on them to determine whether to restart malfunctioning services. In their absence, services may seem functional to users, even when they are not performing correctly behind the scenes [9].

**Auditing**

In industries subject to regulation, the process of auditing is of paramount importance. It guarantees that applications operate in accordance with compliance standards, for instance, confirming that sensitive actions are recorded or that data access is adequately monitored.

Audit logs can be produced by services in a manner like standard logs, which can then be scrutinized to identify unauthorized access, atypical behavior, or breaches of policy. Tools for observability facilitate the automation of this analysis, thereby simplifying the process of ensuring compliance and addressing incidents. [9].

# CHAPTER – RESEARCH METHODOLOGY

## Research Design

We will commence by constructing the system in a monolithic manner, subsequently transitioning to microservices, each possessing its own configuration, and ultimately progressing towards a centralized microservice architecture. Building Microservices with Spring Boot: In this pivotal section, we will delve into the creation of microservices utilizing Spring Boot, emphasizing configuration, RESTful APIs, and the fundamental concepts that are vital for Java developers. Service Discovery and Load Balancing: This segment will investigate how services register and discover one another, alongside load balancing strategies aimed at optimizing resource utilization and enhancing performance. API Communication: This part encompasses the various methods through which microservices can interact with each other, including REST calls and messaging solutions. Data Management: In this section, we will discuss the management of databases within microservices, focusing on Spring Data JPA and the oversight of data transactions and consistency. Security in Microservices: Here, we will explore the integration of security measures within microservices, utilizing JWT for API security and ensuring secure communication between services. Logging and Monitoring: This section is crucial for production environments, covering strategies for effective logging and monitoring of microservices to sustain operational health. Testing and Deployment: Concentrating on the deployment of microservices using Docker and Kubernetes, this section will address essential testing methodologies to guarantee quality.

To ensure a fair comparison, both systems will be deployed in controlled environments with identical workloads. Configuration changes will be introduced, and their effects will be monitored using standardized performance metrics, system logs, and user experience indicators.

This research design enables a focused and measurable comparison of local versus centralized configuration management strategies, providing empirical evidence to support architectural decision-making in microservices development.

To enhance the evaluation further, this research incorporates a focused experimental design that contrasts two methodologies for configuration management in microservices. The first methodology involves embedding configuration settings locally within each service through static files, such as application.properties, which are directly included in the service’s Docker image. Any changes to the configuration necessitate a rebuild and redeployment of the service container. Conversely, the second methodology externalizes configuration and manages it centrally via a configuration server, enabling services to dynamically retrieve and refresh their settings during runtime. Both systems will be deployed under the same conditions and workloads, and their performance will be evaluated using standardized metrics, system logs, and operational indicators. This comparison seeks to furnish empirical evidence concerning the influence of configuration strategies on maintainability, scalability, availability, and developer experience in microservices-based architectures.

## Tools and Technologies

Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code, you can significantly reduce the delay between writing code and running it in production [10].

Spring Boot

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can just run. You can get started with minimum configurations without the need for an entire Spring configuration setup.

Spring Boot offers the following advantages to its developers. Easy to understand and develop spring applications Increases productivity and reduces the development time [11].

Git

Git is an open-source DVCS that allows developers to track and manage changes to their codebase. You can easily manage small as well as large projects with high speed and efficiency by Git. Unlike traditional version control systems, Git allows multiple developers to work on a project simultaneously without interfering with each other's work. We can use Git privately as well as publicly.

Git offers numerous benefits to developers and development teams:

Version Control: Git helps with tracking changes, allowing you to go back to previous states if something goes wrong. Collaboration: It enables multiple developers to work on a project simultaneously without interfering with each other’s work [12].

Spring Cloud

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, short lived microservices and contract testing. Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand-up services and applications that implement those patterns [11].

Key cloak

Key cloak is an open-source Identity and Access Management tool. Being an Identity and Access Management tool, it streamlines the authentication process for applications and IT services.

The purpose of an IAM tool is to ensure that the right people in a company have appropriate access to resources. It usually enables the implementation of SSO, identity federation, and strong authentication [14].

Grafana

Grafana open-source software enables you to query, visualize, alert on, and explore your metrics, logs, and traces wherever they are stored. Grafana OSS provides you with tools to turn your TSDB data into insightful graphs and visualizations. The Grafana OSS plugin framework also enables you to connect other data sources like NoSQL/SQL databases, ticketing tools like Jira or ServiceNow, and CI/CD tooling like GitLab [13].

Prometheus

Prometheus is integrated for real-time monitoring and metrics collection. It provides visibility into application performance, resource usage, and system health, enabling objective measurement of scalability and maintainability impacts [13].

Apache Kafka

Kafka is a distributed event streaming platform used to build real-time data pipelines and messaging systems. It allows microservices to communicate asynchronously by publishing and subscribing to events messages in a fault-tolerant and scalable way [15].

RabbitMQ

RabbitMQ is a lightweight message broker that enables services to send and receive messages using queues. It supports various messaging protocols and ensures reliable delivery, routing, and acknowledgment of messages between microservices [16].

Kubernetes

Kubernetes is an open-source platform for automating deployment, scaling, and management of containerized applications. It manages clusters of containers and ensures applications run consistently, recover from failures, and scale as needed [17].

Helm

Helm is a package manager for Kubernetes that simplifies deployment by using charts configured with application definitions. It allows you to define, install, and upgrade Kubernetes applications in a repeatable and manageable way [18].

Postman

Postman serves as the collaborative platform for teams to develop APIs collectively. Featuring integrated support for the MCP, Postman facilitates the design, testing, and management of APIs that drive both human workflows and intelligent agents [19].

Java

Oracle Java stands as a prominent programming language and development platform. It effectively lowers expenses, accelerates development timelines, fosters innovation, and enhances application services. Java continues to be the preferred development platform for both enterprises and developers [20].

## Evaluation Metrics

### Scalability

* Deployment Time: How quickly services can be configured and restarted after a change.
* Autoscaling Behavior: How well services respond to increased load when horizontally scaled.
* Config Consistency Across Instances: Whether all replicas apply the same configuration during scale-out events.

### Security

* Secret Exposure Risk: Evaluates how securely sensitive data (e.g., credentials, tokens) is handled and stored.
* Access Control: Use of RBAC to limit who can change configurations and access secrets.
* Auditability: Ability to trace who made configuration changes and when, for compliance and forensics.

### Maintainability

* Change Propagation Time: Measures how long it takes for new configurations to be applied across services.
* Rollback Capability: How easily incorrect configurations can be reverted.
* Operational Effort: The number of manual steps or interventions required to apply or manage configuration changes.

# Chapter - SYSTEM DESIGN

## A diagram of a software system AI-generated content may be incorrect.Architecture Overview

Figure 5: Architecture Overview

1. The API Client refers to any external system, application, or tool (Postman) that initiates HTTP requests to engage with the backend system. It serves as the initial point for testing or utilizing APIs made available by microservices through a unified entry point.

2. (Spring Cloud Gateway) The API Gateway functions as the sole entry point for all requests from external clients. It undertakes essential cross-cutting tasks such as: Routing requests to the corresponding microservice (e.g./accounts, /loans) Enforcing security measures (authentication & authorization) Implementing rate limiting, retries, logging, and load balancing Serving as an abstraction layer that conceals internal service implementation details from the client.

3. (Service Registry) Eureka Server acts as the service discovery mechanism offered by Spring Cloud Netflix. Each microservice: Registers itself upon initialization, providing its name, IP address, and port number. Dynamically discovers other services using the registry (for instance, the Accounts service can find and invoke the Loans service without hardcoding its address). This facilitates dynamic scaling and fault tolerance in distributed environments.

4. Config Server (Spring Cloud Config Server) The Config Server offers centralized and externalized configuration management. Rather than hardcoding configurations within each microservice, they: Retrieve configurations at startup or during refresh from the Config Server. Obtain consistent values for aspects such as database credentials, service ports, feature toggles, and environment-specific properties. Configurations are generally stored in a Git repository or a local file system overseen by the config server.

5. (Accounts, Cards, Loans) Each microservice is crafted with the Single Responsibility Principle in mind, meaning it possesses a distinct business capability: Accounts Service – Oversees user accounts and profile information. Cards Service – Manages card-related data and operations. Loans Service – Manages loan processing, approval, and related tasks.

6.Event Broker (RabbitMQ / Kafka) The Event Broker facilitates asynchronous communication among microservices using events: Publishing Events: Services disseminate domain events (e.g., "Loan Approved", "Card Created") to the broker. Decoupled Processing: Other services can subscribe to pertinent topics without establishing direct dependencies. Scalability & Resilience: This approach diminishes tight coupling and improves responsiveness in environments with high load.

7.Message Consumer A message consumer is a backend component (or service) that listens to and processes incoming events from the broker: Asynchronous Handling: This enhances performance by delegating processing tasks. Loose Coupling: It allows microservices to respond to events without awareness of their source. Use Cases: Examples include sending notifications, updating projections, or initiating workflows.

8.Observability & Monitoring (Prometheus, Grafana) This layer is essential for sustaining operational visibility: Metrics Collection: Prometheus collects service metrics (CPU, memory, request latency). Visualization: Grafana offers dashboards for monitoring health, usage, and trends. Distributed Tracing: Open Telemetry facilitates tracing across services to pinpoint performance bottlenecks. Health Checks: Services provide /actuator/health and tracing endpoints to oversee uptime and readiness.

As the system evolves from a monolithic architecture to a microservices-oriented design, Docker assumes a crucial role in encapsulating and managing each phase of this transformation. Initially, the application was constructed as a conventional monolith utilizing Spring Boot. During this phase, Docker is employed to package the entire application into a singular image, encompassing all dependencies and configuration files. Typically, the configuration is integrated within the application itself, often represented as an application.properties file that is bundled directly within the JAR. This Docker image facilitates consistent execution across various environments; however, it lacks flexibility—any modification in configuration and requires the image to be rebuilt and redeployed.

As the architecture progresses towards microservices, the application is segmented into distinct business domains, such as Accounts, Cards, and Loans. Each microservice is crafted as an independent Spring Boot application, complete with its own Docker image. These images continue to include embedded configuration files, rendering them self-sufficient yet inflexible. A change in configuration for one service necessitates the rebuilding and redeployment of its Docker image, resulting in the duplication of environment specific values across services and prolonged feedback loops during development and testing.

The subsequent phase introduces a centralized configuration management system using Spring Cloud Config Server. This separate service, the Config Server, is also containerized using Docker. Configuration values are no longer embedded within each service image; instead, they are stored externally typically in a Git repository or a mounted local file system that is accessible to the Config Server. Microservices are reconfigured to dynamically retrieve their configuration at startup from the Config Server. This architecture allows the Docker images of the microservices to remain unchanged even when configuration values are modified. Environment-specific details, such as database URLs, ports, and credentials, are relocated outside the service container.

## Implementation Approach

### microservice

This project is an independent, self-contained application that handles specific business capability (such as accounts, cards, or loans). Each microservice has its own codebase, database, and can be developed, deployed, and scaled independently. Microservices communicate with each other over the network (usually via REST APIs), often using DTOs to exchange data. Service discovery (Eureka), centralized configuration (Config Server), and monitoring/tracing are used to manage and observe the microservices. This architecture improves modularity, scalability, and maintainability of the overall system.

1. Config server: A centralized configuration server designed for all microservices, providing configuration properties.
2. Eureka server: A service registry that facilitates service discovery, enabling microservices to locate and interact with one another.
3. accounts: Oversees business logic and data related to accounts (e.g., user accounts, balances).
4. loans: Manages operations associated with loans, including applications, approvals, and overall management.
5. cards: Oversees services related to cards, such as the issuance and management of credit and debit cards.
6. message: Presumably responsible for messaging or notifications between services or directed towards users.
7. Gateway server: An API gateway that directs external requests to the corresponding microservices, manages authentication, and serves as a unified entry point.

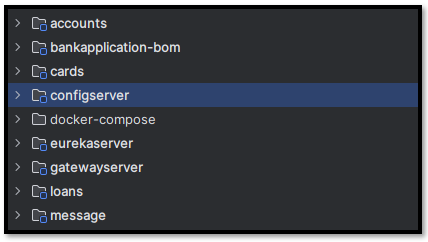


Figure 6: Microservices in Application

### ****Maven Dependance****

is a reference in your pom.xml file that tells Maven to download and include a specific library or module in your project. Dependences are defined inside the <dependencies> section. This will add the Spring Boot Web starter to your project. Maven will automatically download it and make it available for your build.

The following are examples of various Spring Boot starters and their functionalities: spring-boot-starter-actuator: This starter adds endpoints that facilitate the monitoring and management of the application. opentelemetry-javaagent: This component enables distributed tracing, enhancing observability. micrometer-registry-Prometheus: This starter exposes metrics that can be utilized by Prometheus. spring-boot-starter-data-jpa: It provides support for JPA and Hibernate, allowing for database access. Spring-boot-starter-validation: This starter enables bean validation capabilities. Spring-boot-starter-web: It serves as the core dependency for constructing RESTful APIs. Spring-boot-dev tools: This tool enhances the development experience by allowing hot reload during runtime. h2: This is an in-memory database designed for development and testing purposes, applicable at runtime only. Spring Boot Starter Test: It offers testing support specifically for Spring Boot applications. Lombok: This library minimizes boilerplate code using annotations.

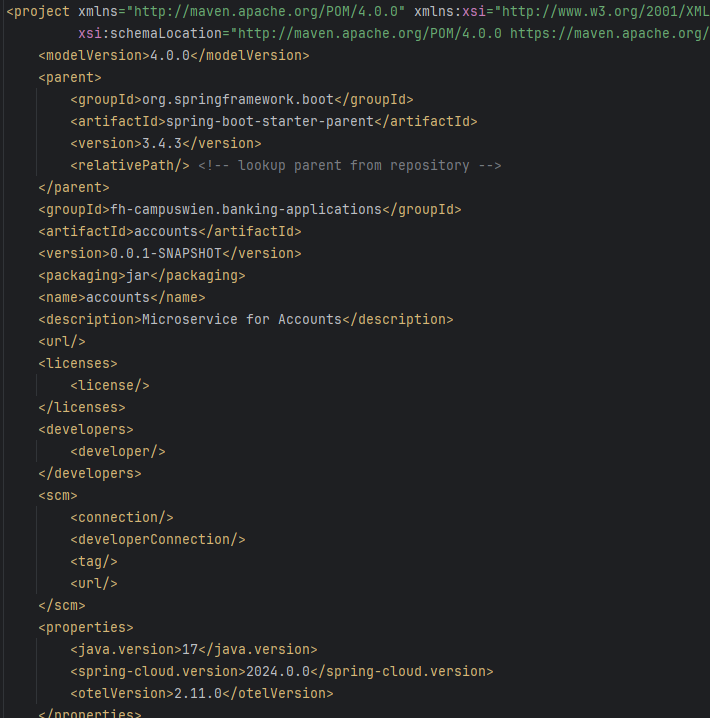


Figure 7: Account Maven Dependance Example

### Entity

In this project, an entity is defined as a Java class that corresponds to a table within the database. Entities are generally marked with the @Entity annotation and are integral to the persistence layer, overseen by Spring Data JPA. Each entity correlates its fields with the columns in the database, enabling the application to execute CRUD operations on the underlying data.

Entities serve to represent fundamental business objects, including accounts, customers, loans, or cards. They frequently incorporate annotations such as @Id for primary keys and may utilize relationships like @OneToMany or @ManyToOne to establish connections between tables.

By employing entities, the project capitalizes ORM, facilitating interaction with the database through Java objects rather than direct SQL queries. This methodology enhances maintainability, readability, and compatibility with Spring Boot’s data management capabilities.

The Customer class represents a customer in the database for the accounts service. It is a JPA entity mapped to a database table, with fields for customer ID, name, email, and mobile number. By extending Base Entity, it also inherits auditing fields (created/updated timestamps and user info). This class is used to persist, retrieve, and manage customer data within the application.

A screenshot of a computer

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.

Figure 8: Customer Class Entity Example

### Data Transfer Object

A DTO in a microservices architecture is used to transfer data between services, often over the network (e.g., via REST APIs). DTOs help decouple internal domain models from external representations, ensuring that only necessary data is shared between microservices. This improves security, versioning, and maintainability. For example, when the accounts service calls the cards service to fetch card details, it uses a DTO to structure the request and response payloads, making inter-service communication clear and consistent. Each microservice defines its own DTOs for both incoming and outgoing data.

A screenshot of a computer

AI-generated content may be incorrect.

Example: The CardsDto class serves as a Data Transfer Object (DTO) for card information within the application. Its main purpose is to encapsulate and transfer card-related data between different layers (such as controller and service) or between microservices, ensuring that only the necessary card data is exposed and validated. It also supports API documentation and input validation through annotations, promoting clear contracts and data integrity in the system.

A computer screen shot of a program code

AI-generated content may be incorrect.

Figure 9: Data Transfer Object Example

### Repository

In this project, a repository is defined as a Java interface that facilitates data access for entities, usually by extending either JpaRepository from Spring Data JPA.

Repositories serve to abstract the database layer, enabling the execution of CRUD operations (create, read, update, delete) on entities without the necessity of writing SQL queries.

The responsibilities of repositories include:

Interacting with the database to retrieve, save, update, or remove entity records.

Providing custom query methods through method naming conventions.

Supporting pagination, sorting, and complex queries.

This methodology fosters a clear distinction between business logic and data access, thereby enhancing the maintainability and testability of the codebase.

They define the contract for account-related operations in the application. It specifies methods for creating, fetching, updating, and deleting accounts, as well as updating account status. Implementations of this interface provide the business logic for managing account data, ensuring a clear separation between the service layer and other layers (such as controllers or repositories). This promotes modularity, testability, and maintainability in the codebase.

A screen shot of a computer

AI-generated content may be incorrect.A close up of a text

AI-generated content may be incorrect.

Figure 10: Account Repository Example

### Controller

In a Spring Boot application, the controller functions as a fundamental element of the web layer, tasked with managing incoming HTTP requests, processing them (often by delegating responsibilities to service classes), and delivering suitable HTTP responses. Controllers are marked with @RestController or @Controller and establish request mappings through annotations such as @GetMapping, @PostMapping, @PutMapping, and @DeleteMapping.

Controllers act as the initial point of contact for client interactions, providing RESTful APIs that enable clients to execute operations like creating, reading, updating, or deleting resources. They are responsible for validating input data, managing exceptions (frequently with the assistance of global exception handlers), and returning responses in a well-structured format, usually in JSON.

A screen shot of a computer program

AI-generated content may be incorrect.The AccountsController in fh\_campuswien.banking\_applications.accounts.controller exposes REST APIs for managing account-related operations. It provides endpoints to create, fetch, update, and delete accounts, as well as to retrieve build info, Java version, and contact information. The controller uses validation, integrates with the service layer, and leverages resilience patterns like retry and rate limiting. It ensures that all account operations are accessible via well-defined HTTP endpoints, returning clear and consistent responses.

Figure 11: Account Controller Class Example

### Exception Handler

The purpose of the is to centralize and manage exception handling for the accounts microservice. It contains custom exception classes and a global exception handler to: Define specific exceptions for business errors (e.g., customer already exists, resource not found).

Intercept and handle exceptions thrown during request processing.

A screenshot of a computer

AI-generated content may be incorrect.The GlobalExceptionHandler class, located in the fh\_campuswien.banking\_applications.accounts.exception package, serves as a centralized component for exception handling within your Spring Boot application. It is annotated with @ControllerAdvice, allowing it to intercept exceptions raised by controllers and deliver custom responses, thereby enhancing error reporting and the overall client experience. Key features: Validation Error Handling: This feature overrides the handleMethodArgumentNotValid method to capture validation errors (such as invalid request bodies) and returns a map detailing field error along with a 400 Bad Request status. Global Exception Handling: The handleGlobalException method is designed to catch all unhandled exceptions, providing a structured error response accompanied by a 500 Internal Server Error status. Custom Exception Handling: There are specific handlers for CustomerAlreadyExistsException and ResourceNotFoundException that return meaningful error messages along with the appropriate HTTP status codes (400 and 404).

Benefits: Ensures uniform error responses throughout the application. Enhances maintainability by consolidating exception logic. Improves API usability by offering clear and structured error messages to clients.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 12: Global Exception Class Example

### Application YML file

The configures your Spring Boot application. It defines settings for the server, database, logging, service discovery, messaging, monitoring, and resilience. This file centralizes environment-specific and service-specific properties, enabling features like:

* Setting server port and application name
* Configuring the H2 in-memory database and JPA/Hibernate
* Enabling the H2 console for development
* Importing external configuration from a config server
* Setting up Kafka messaging and (optionally) RabbitMQ
* Exposing and customizing Spring Boot Actuator endpoints for monitoring and health checks
* Registering with Eureka for service discovery
* Defining logging levels and patterns
* Configuring Resilience4j for circuit breaking, retrieving, and rate limiting
* This file ensures the application runs with the correct infrastructure, resilience, and monitoring settings.

A screen shot of a computer

AI-generated content may be incorrect.

Figure 13: Application.yml file Example

## Configuration Strategy

### API

The Swagger UI functions as a web-based interface that presents the technical specifications of the services within a software system. The accompanying screenshot illustrates the Accounts microservice, which is a part of a larger banking system tasked with functions such as opening accounts, updating customer information, and retrieving account data. Below is a clear explanation of what the Swagger UI reveals:

Purpose of the Page: It aids users (especially software developers or testers) in comprehending: What operations the system can perform (for example, creating a new account or retrieving account details) What information is required (such as name, email, and account number) What type of response will be generated (whether it is a success or an error, along with the relevant message)

Main Features of the Accounts Microservice: Create an Account: You can direct the system to create a new customer account by providing necessary information such as: Full name, Email address, Phone number, Account type (for instance, Savings), Branch address. Update Account Information: This feature allows for the alteration of a customer's contact or account details. Fetch Account Details: This function enables you to access all information related to a specific account. Delete an Account: This action removes a customer and their account information from the system. View Build and Version Info: This displays the current version of the service in use along with technical build information. Fetch Customer Details: This retrieves both customer and account information at once, which is advantageous for comprehensive profile views.

What Kind of Data Is Involved? The system expects and returns data in a structured format that includes Customer Info: name, email, mobile number; Account Info: account number, type, and branch; Other Info: error messages (in case of issues), success codes, or system responses.

Why It’s Useful (Even for Non-Technical Stakeholders): It ensures that all components of the system interact effectively. It supports developers, testers, and architects in their tasks.

**All Microservices API documents PDF will be in folder in GIT-Repository**

**This example for Account microservice**

A screenshot of a computer

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 14: API Example for Account micro service 1

Figure 15: API Example for Account micro service 2

Figure 16: API Example for Account micro service 3

A screenshot of a computer program

AI-generated content may be incorrect.

Figure 17: API Example for Account micro service 4

### Docker

In modern software development, Docker is used to package and run applications in isolated environments called containers. These containers bundle everything an application needs: the code, dependencies, and configuration. For microservices-based applications, Docker plays a critical role in ensuring portability, consistency, and scalability across development, testing, and production environments.

In this setup, we containerize Spring Boot applications (like Accounts, Loans, and Cards microservices) using three different methods, each suited for different developer needs and automation levels.

Three Ways to Containerize a Spring Boot Microservice

1. Docker file

2. Build packs

3. Google Jib

We decided to go Build packs in project as it is secure and easier way

This section defines the account microservice in Docker Compose setup. It specifies:

The Docker image to use (aimendocker/account:s14).

The container name (docuker-ms).

A computer screen shot of a program

AI-generated content may be incorrect.

Figure 18: Account Info for Docker Image

Docker Compose is utilized in this project to manage sophisticated microservices architecture, allowing developers to define, configure, and operate multiple interrelated services through a single YAML file. The Compose configuration encompasses essential business microservices (accounts, loans, cards, message, Gateway server), infrastructure elements (Kafka, Redis, RabbitMQ, Keycloak, Prometheus, Grafana, Loki, Tempo, Minio, Nginx), and auxiliary services (Config Server, Eureka Server). Each service is characterized by its image, ports, environment variables, health checks, dependencies, and a shared network. The implementation of extends and common configuration files (such as common-config.yml) adheres to DRY principles, centralizing resource limits, environment variables, and network configurations. This methodology streamlines scaling, testing, and deployment, facilitating the rapid initiation of the entire ecosystem for development, testing, or production. Docker Compose further oversees service dependencies and health checks, guaranteeing that services like Eureka and Config Server are operational before dependent microservices commence. Observability is incorporated through Prometheus, Grafana, Loki, and Tempo, delivering monitoring, logging, and tracing capabilities out of the box. The configuration promotes resilience, security (with Keycloak for authentication), and event-driven communication (via Kafka and RabbitMQ).

Three Distinct Environments:

Development Environment: Employs in-memory or lightweight databases (e.g., H2, local volumes). Opens all ports for straightforward access and debugging. Facilitates hot-reload and developer tools. Observability tools (Grafana, Prometheus, Loki) are entirely accessible. Environment variables are configured for local development (e.g., SPRING\_PROFILES\_ACTIVE=dev).

Testing/Staging and Production Environment: Replicates production but may utilize reduced resource limits. Health checks and dependencies are rigorously enforced. Data persistence is enabled through Docker volumes. Security and authentication (Keycloak) are operational. Environment variables direct to test endpoints and utilize SPRING\_PROFILES\_ACTIVE=staging.A screenshot of a computer

AI-generated content may be incorrect.

### Config Server

The configuration file delineates a Spring Boot application intended to function as a centralized configuration server within a microservices architecture. Its primary role is to externalize and oversee configuration properties for all related microservices from a singular, version-controlled source, thereby guaranteeing consistency, improving manageability, and enabling dynamic updates across diverse environments.

The application is given a logical name for identification when it registers with other components in a distributed system, such as a discovery server. This naming convention assists client applications in finding the configuration server and obtaining their specific configuration files based on their service identifiers.

The configuration profile activates the Git backend, signifying that the application will derive configuration properties from a remote Git repository. This selection permits version control of configuration files, ensuring that modifications are traceable over time, while also allowing rollback capabilities and promoting collaboration by treating configuration as code. An alternative native profile is available but commented out, which would facilitate loading configurations directly from the local file system or class path, beneficial in testing scenarios or when Git is not accessible.

Within the Git configuration section, various properties govern the application's interaction with the remote repository, including the repository's location, the default branch, and performance settings such as timeouts. The server is set to automatically clone the Git repository upon startup and to pull the latest changes with each refresh, ensuring that the configuration server consistently delivers the most up-to-date properties to client services without necessitating manual updates.

In summary, this setup creates a secure, scalable, and centralized system for overseeing externalized configurations across various microservices. It effectively tackles issues such as configuration drift, manual property overrides, and inconsistent deployments, thus enhancing the reliability, maintainability, and agility of the overall software architecture.

Developer Initiates Configuration Update:

A developer pushes new or modified configuration files to the centralized configuration repository (for instance, GitHub).

Webhook Activation:

A designated webhook on the Git repository identifies changes and automatically alerts the Config Server.

Broadcast of Configuration Change Event:

The Config Server transmits a configuration refresh event to a message broker (such as Kafka or RabbitMQ), which informs all subscribed microservices.

Dynamic Configuration Reload:

Subscribed microservices (like Accounts, Loans, and Cards) receive the event and update their configurations dynamically without necessitating a restart, thereby facilitating uninterrupted updates.

A screen shot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 19: Config Application YAML File

Figure 20: Config Server GitHub

### Client-Side Service Discover

1. Client-side service discovery is an architectural strategy utilized in distributed systems, where client applications dynamically locate and interact with services during runtime. Rather than depending on a central load balancer or gateway, each client queries a centralized service registry to gather information about the service instances it needs to connect with. Upon initialization, every microservice instance registers itself with the service registry, providing essential details such as its IP address, port number, and other relevant metadata. When a service instance is terminated or fails, it unregisters or is eventually removed from the registry, ensuring that the discovery data remains current.

A computer screen shot of a black background with orange text

AI-generated content may be incorrect.

Figure 21: Eureka Discovery

2.framework, when a client intends to communicate with another service, it queries the service registry to obtain a list of available instances. The client then selects one of these instances based on a load-balancing strategy defined within its logic, which may include methods like round-robin selection, least-connections, or more sophisticated latency-aware approaches. While this model enhances flexibility and efficiency by removing centralized bottlenecks, it also adds complexity to the client side, which must now incorporate service discovery and load balancing logic.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 22: Eureka Example

3. The Spring Cloud ecosystem provides strong support for this client-side discovery pattern. Eureka, a popular service registry created by Netflix, serves as the foundation for service registration and discovery. Clients utilize the Spring Cloud Load Balancer, which replaces the now-obsolete Netflix Ribbon, to manage load balancing among service instances. Additionally, feign clients are frequently employed in Spring applications to streamline HTTP communications between services and integrate smoothly with the discovery mechanism. Although Eureka is predominantly used in Spring-based applications, other service registries like Consul, etc., and Apache Zookeeper offer similar functionalities and are utilized in non-Spring or polyglot environments.

4. A heartbeat in a service discovery system like Eureka is a periodic signal sent by service instances to the service registry to indicate that they are active and healthy. These signals allow the registry to maintain an up-to-date list of available services. If a heartbeat is missed for a configured duration, the registry may assume the service is unavailable and remove it. However, to prevent false removals due to temporary issues, Eureka uses self-preservation mode, which delays eviction if too many heartbeats are missed system-wide, ensuring continued stability.

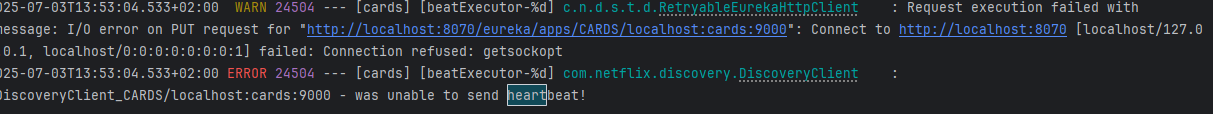
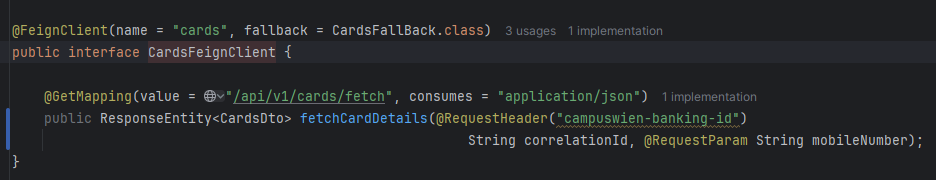


Figure 23: Eureka heartbeat Example

### RESILIENCY

Ensuring resilience and stability in microservices architecture requires careful handling of failures and network disruptions between services. A single slow or failed service should not cause cascading failures across the system. To mitigate this risk, fallback mechanisms are employed, allowing the system to return default values, fetch data from caches, or reroute to alternative services or databases when a dependent service is unavailable. This approach ensures continuity even in partial failure scenarios.

In addition to fallbacks, it is crucial for services to exhibit self-healing behavior. This involves configuring timeouts and automatic retries, allowing temporarily failing services to recover without manual intervention. Tools such as Resilience4J, which has become a popular alternative to the now-deprecated Hystrix library, provide robust support for such resilience patterns in the Java ecosystem.

A screen shot of a computer program

AI-generated content may be incorrect.The Circuit Breaker pattern, inspired by its counterpart in electrical systems, plays a key role in protecting services from being overwhelmed by repeated failed calls. It monitors remote calls and interrupts them if they become unresponsive or consistently fail, thereby preventing further strain on the system. Once the issue is resolved, the circuit breaker allows traffic to flow again, ensuring minimal downtime and improved fault tolerance.

Figure 24:Card Service Feign Client

Complementing this is the Retry pattern, which allows operations to be retried after transient failures, particularly useful in scenarios like brief network disruptions. It includes strategies such as exponential backoff to avoid overwhelming resources and integrates well with circuit breakers to halt retries after a certain threshold, maintaining system efficiency. To avoid unintended consequences, operations subject to retrieves must be idempotent, producing the same result regardless of how often they are executed.

Another crucial resilience strategy is the Rate Limiter pattern, which protects services from being overloaded by excessive or abusive requests. By limiting the number of allowed requests within a specific time frame—based on IP address, user, session, or subscription tier—the system ensures fair usage and maintains availability. When limits are exceeded, requests are rejected with a standard response, typically HTTP 429 (Too Many Requests). This not only protects system performance but also supports differentiated service levels for various user groups, such as basic and premium users.

Together, these patterns—fallbacks, circuit breakers, retries, and rate limiting—form the foundation for building resilient, self-healing, and stable microservices systems capable of withstanding failures and continuing to deliver reliable user experiences.

These settings help your service handle failures gracefully, avoid cascading errors, and protect downstream systems by controlling retries and request rates.

A screen shot of a computer program

AI-generated content may be incorrect.

Figure 25: Circuitbreaker Example

### OBSERVABILITY AND MONITORING

Observability refers to the ability to understand the internal state of a system by analyzing its external outputs. In the context of microservices, observability is achieved by collecting and evaluating data from various sources such as metrics, logs, and traces. Metrics provide quantitative insights into the health of a system, including CPU usage, memory usage, and response times. Logs serve as chronological records of events occurring within the system, helping to trace errors, exceptions, and other unexpected behaviors. Traces capture the journey of a request across the microservices architecture, helping to identify latency and performance bottlenecks. These three pillars together enable comprehensive visibility into a system's behavior and health.

Monitoring complements observability by continuously examining telemetry data and generating alerts for known failure states. It focuses on identifying and resolving problems, tracking the health of individual services, and optimizing system performance. Monitoring is reactive in nature, responding to issues as they arise, while observability takes a more proactive approach by offering insights that help us understand and resolve root causes in real time.

Logs are essential tools for diagnosing problems and understanding system behavior. Each log entry includes a timestamp and contextual information about specific events. Log levels such as trace, debug, info, warning, and error allow filtering of log data based on severity. While monolithic applications store all logs in a centralized manner due to their single-codebase nature, microservices architectures require centralized logging systems to consolidate logs from multiple services. This is crucial for tracking issues across services and reconstructing the sequence of events for individual requests.

A screenshot of a computer

AI-generated content may be incorrect.Grafana is a widely used open-source tool that provides visualization for metrics, logs, and traces. It enables interactive dashboards and alerting mechanisms by connecting to various data sources. Grafana Loki, designed for scalable and cost-effective log aggregation, works alongside Grafana Alloy, a lightweight log agent that ships logs from containers to Loki. Together, they offer a powerful and scalable logging solution, allowing centralized management and analysis of logs across distributed microservices environments.

A screenshot of a computer program

AI-generated content may be incorrect.

### OpenID Connect, KeyCloak

Securing the Spring Cloud Gateway using the OAuth2 Authorization Code Grant Type involves a multi-step process to ensure that only authenticated and authorized users can access protected microservices. When an end user attempts to access a secure page via a web or mobile application, the request is intercepted by the gateway server. If no access token is provided, the gateway denies access and instructs the client to obtain a token from the authentication server, in this case, Keycloak.

The client application redirects the user to the Keycloak login page, where the user enters their credentials. Upon successful authentication, Keycloak issues an authorization code to the client. The client then sends this code, along with its client ID and secret, back to Keycloak to request an access token. Once the authentication server validates the information, it issues an access token to the client.

With this access token, the client can now make authenticated API calls to the gateway. The gateway, acting as an edge server, forwards the token to the authentication server to verify its validity. If the token is valid, the gateway routes the request to the appropriate resource server, such as Accounts, Loans, or Cards microservices. These resource servers are typically deployed within a secure Docker or Kubernetes network and cannot be accessed directly from outside.

This process ensures secure communication between the end user and backend microservices, leveraging OAuth2 standards and the Authorization Code Grant flow to provide robust authentication and authorization mechanisms in microservices architecture.

In the OAuth2 framework, several key components work together to enable secure authorization. The Resource Owner is the end user who owns the data or resources, such as profile details or email. The Client is an application (e.g., a web or mobile app) that requests access to the user's resources on their behalf. The Authorization Server is responsible for authenticating the resource owner and issuing access tokens to the client. It maintains the identity of the user and handles authorization logic. The Resource Server hosts the protected resources and validates the access token before granting access. Scopes define specific levels of access that the client requests, such as reading user profile data or accessing email.

OpenID Connect (OIDC) extends OAuth2 by adding an authentication layer. While OAuth2 handles authorization through access tokens with scopes, OpenID Connect introduces an ID Token to convey user identity information. This ID Token is formatted as a JWT (JSON Web Token) and includes claims such as user ID, email, and other profile data. OIDC also introduces standardized scopes like openid, profile, email, and defines a /userinfo endpoint to fetch user data. By enabling secure and standard identity sharing between applications, OIDC completes the identity and access management (IAM) solution.

The Client Credentials Grant Type in OAuth2 is used when there is no end user involved. It is a simple and efficient flow used for server-to-server communication. In this flow, the client authenticates directly with the Authorization Server using its client\_id and client\_secret, along with the requested scope and the grant\_type set to client\_credentials.

If the credentials are valid, the Authorization Server responds with an access token. The client can then use this token to request protected resources from the Resource Server, which validates the token before granting access. This flow is suitable for backend services or APIs exchanging data without user interaction.

### Event-driven microservices

Event-driven architectures use two main models: Publish/Subscribe (Pub/Sub) and Event Streaming.

In the Pub/Sub model, producers generate events that are immediately sent to subscribers. Once an event is consumed, it cannot be replayed, so new subscribers joining later will not have access to past events. RabbitMQ commonly implements this model. When someone, for example, creates a new account, the accounts microservice publishes an event to the event broker. The broker forwards this event to queues subscribed by services like the message service. These services process the event and perform actions such as sending emails or SMS to the customer, while also updating database statuses accordingly. RabbitMQ uses AMQP (Advanced Message Queuing Protocol) and works through exchanges and queues, routing messages based on specific rules. Producers send messages to exchanges, which then route messages into queues for consumers to process. Exchanges support different routing mechanisms, and consumers listen to queues to receive and process messages asynchronously.

In contrast, the Event Streaming model is based on writing events sequentially to a log, allowing consumers to read events from any position in the stream. Apache Kafka is widely used in this approach. Events can be replayed, meaning new consumers can access historical data at any time. Kafka’s architecture organizes data into topics, which are divided into partitions. Producers write messages to topics, while brokers store and replicate messages to ensure fault tolerance. Each message within a partition has a unique offset for tracking. Consumers, grouped into consumer groups, read messages from these topics. Kafka also supports replaying events, message retention, and managing offsets for reliable consumption.

Spring Cloud Function is introduced to build the business logic using simple Java functions (Supplier, Function, Consumer). Functions are packaged and exposed via HTTP endpoints or integrated with event brokers through Spring Cloud Stream. Developers write functions, register them as beans, and compose them if necessary. The framework enables deploying these functions on serverless platforms (AWS Lambda, Azure Functions, etc.) or integrating them directly with messaging systems like RabbitMQ and Kafka through configuration without altering the application code.

Spring Cloud Stream acts as the abstraction layer that connects Spring functions with external brokers. It handles infrastructure concerns like channel creation and message routing. It uses destination binders to integrate with brokers and destination bindings to connect application code with these destinations. Input bindings receive messages for functions to process, while output bindings send data from functions to brokers. Each binding maps to exchanges or topics, depending on whether RabbitMQ or Kafka is used.

In practical implementation:

* In the accounts microservice, event production involves using StreamBridge to send events asynchronously to output bindings configured in the application. Events such as account creation trigger messages sent to a queue or topic.
* The message service listens for these events via input bindings, processes them (e.g., sending an email or SMS), and may publish further events to confirm processing completion.
* Functions consuming events update the database to reflect communication status.

Switching from RabbitMQ to Kafka primarily involves replacing the message broker dependencies and updating destination configurations in the application.yml files. Kafka handles data retention, allowing consumers to read past messages, whereas RabbitMQ focuses on direct message delivery with lower latency.

Kafka’s system scales horizontally by adding brokers and partitions, making it suitable for large-scale data handling. Messages in Kafka are written to partitions within topics and stored durably with offsets. Consumers fetch messages in a pull-based manner, keeping track of their progress using offsets, and are capable of replaying messages if needed.

Kafka producers configure broker addresses and serialization formats before publishing messages to specified topics. Messages are assigned to partitions using keys or a round-robin algorithm and replicated across brokers to ensure fault tolerance. Producers receive acknowledgments once the messages are written, and retries or error handling are managed accordingly.

Kafka consumers subscribe to topics, fetch messages from partitions, process them, and commit offsets to track progress. Consumers operate in a continuous polling loop to fetch and process new messages.

In summary, RabbitMQ supports event-driven systems with immediate message delivery and complex routing, while Kafka offers durable event storage and replay capability, ideal for large-scale, high-throughput streaming applications. Both are integrated into event-driven microservices architectures using Spring Cloud Stream and Spring Cloud Function, where the business logic is built using Java functions and broker configurations are handled declaratively.

### Kubernetes

Kubernetes is an open-source platform designed to automate the deployment, scaling, and management of containerized applications. Initially developed by Google and released as open-source in 2014, Kubernetes is cloud-neutral and integrates Google’s extensive experience with scalable production systems alongside community best practices. The term "Kubernetes" is derived from Greek, meaning helmsman or pilot, and is often abbreviated as K8s.

Kubernetes enables the resilient operation of distributed systems by offering functionalities such as service discovery, load balancing, container orchestration, storage orchestration, automated rollouts and rollbacks, self-healing, and secret and configuration management.

Its architecture consists of a Control Plane (Master Node) and multiple Worker Nodes. The Control Plane manages the cluster’s state and workload distribution, monitoring node health and reallocating tasks in case of node failures. Its key components include:

* API Server, the primary communication interface for the cluster.
* Scheduler, responsible for assigning Pods to worker nodes based on resource needs and constraints.
* Controller Manager, maintaining the system’s desired state by replicating components and handling failures.
* etcd, a distributed key-value store that holds configuration data and cluster state.

Worker Nodes are servers (physical or virtual) that run containerized applications. Each node includes:

* Kubelet, an agent that communicates with the Control Plane to manage Pods and containers.
* kube-proxy, which maintains network rules for Pod communication.
* Container Runtime (commonly Docker), responsible for running and managing containers.

Kubernetes uses Pods as its smallest deployment unit, which may contain one or more containers. Each Pod receives a unique IP within the cluster.

Configuration management in Kubernetes is handled using resources like ConfigMaps, which store key-value pairs externally from application code. These configuration values can be injected into Pods as environment variables, enabling dynamic configuration without altering container images.

Application deployment is defined declaratively using Deployment manifests. A Deployment ensures the desired number of replicas for an application are running, manages updates, and facilitates rollback if necessary. The Deployment manifest specifies details such as container images, environment variables (often sourced from ConfigMaps), ports to expose, and the number of replicas.

To expose applications, Kubernetes uses Services, which provide stable network endpoints and load balancing across Pods. A Service routes traffic to Pods based on label selectors. The most common Service type, ClusterIP, limits accessibility to within the cluster, while other types (like NodePort or LoadBalancer) expose services externally.

Overall, Kubernetes abstracts infrastructure complexity, providing a unified platform to deploy, manage, and scale containerized applications efficiently across varied environments.

### Helm

is widely recognized as the package manager for Kubernetes, created to simplify the management of Kubernetes applications by providing a structured and efficient approach to handling Kubernetes manifest files. In a typical Kubernetes environment without Helm, DevOps teams are required to manually maintain and manage numerous YAML files for every resource type, such as Deployments, Services, and ConfigMaps. Each microservice in a project would require its own set of manifest files, even though the content of these files is largely repetitive, differing only in certain dynamic values like service names or port numbers. This approach not only increases manual work but also makes the management of multiple microservices cumbersome and prone to error.

To address this challenge, Helm introduces a packaging system called charts. A chart in Helm is essentially a bundle of files that together define a related set of Kubernetes resources. Helm charts can be used to deploy anything from a single simple application to a complex multi-service infrastructure, such as an entire web application stack consisting of web servers, databases, and caching layers. Charts can also define dependencies on other charts, allowing entire project dependency trees to be installed with a single Helm command.

Helm simplifies Kubernetes resource management by using template files in place of static YAML files. Rather than maintaining individual manifest files for each service, Helm allows the creation of generic template files where placeholders are defined. The actual values for these placeholders are provided in a separate yaml file. When a Helm chart is deployed, these dynamic values are injected into the templates, resulting in customized manifests for each service without the need for duplicated code. This templating mechanism makes scaling, upgrading, and maintaining applications significantly easier.

In addition to templating, Helm offers several built-in capabilities to enhance the operational efficiency of Kubernetes environments. It enables packaging all necessary manifest files of an application into a single chart that can be stored in both public and private repositories. Helm charts can then be easily distributed and reused across teams and environments. Deploying, upgrading, rolling back, or removing entire applications is streamlined through Helm’s single-command operations, eliminating the need for repetitive manual kubectl commands.

Helm also manages the history of deployed applications automatically. Each installation or upgrade operation is tracked, allowing teams to roll back to any previous working version of their application with minimal effort. This version control feature ensures that if any deployment introduces issues, the system can be quickly reverted to a stable state.

A Helm chart is typically structured in a standardized way. The Chart.yaml file contains metadata about the chart itself, such as its name and version. The values.yaml file holds configuration data that defines dynamic values used in the templates. The charts directory contains other charts that the current chart depends on, supporting complex deployment scenarios. Finally, the templates directory houses the template YAML files that represent the Kubernetes resources to be deployed.

Using the Helm command-line tool, developers and DevOps teams can create, manage, and deploy charts efficiently. Commands like helm create generate a new chart scaffold, while helm install, helm upgrade, and helm rollback handle application lifecycle management. Helm also provides commands to list releases, view deployment histories, and render chart templates locally for inspection.

In summary, Helm brings automation, consistency, and simplicity to Kubernetes resource management, reducing manual effort and improving deployment reliability. It acts as a comprehensive tool that packages, installs, manages, and versions Kubernetes applications in a way like traditional package managers used in operating systems.

### Postman Collection for Microservices Testing

In order to facilitate systematic and automated testing of the microservices architecture, a Postman collection has been developed that encompasses all essential services: Accounts, Cards, Loans, Config Server, Eureka Server, and API Gateway. This collection offers a structured and reusable method for invoking API endpoints, validating service behavior, and testing updates to runtime configurations.

The Accounts service comprises endpoints for creating, updating, retrieving, and deleting account records, in addition to fetching customer details and checking system metadata such as build version and Java version. The Cards service permits similar operations tailored to card data, while the Loans service is responsible for the creation, updating, and deletion of loans. Each of these services features actuator endpoints like /actuator/refresh, which facilitate dynamic configuration updates without necessitating a service restart.

The Config Server endpoints are designed to support the encryption and decryption of sensitive configuration data, thereby validating the centralized configuration management process. The Eureka Server request confirms registered service discovery at runtime, ensuring that all services remain dynamically connected and accessible without the need for hardcoded service URLs.

The Gateway Server section includes requests that are routed through the API Gateway, validating endpoint mapping, security integration, and token-based authentication utilizing OAuth2. Numerous test cases encompass both Bearer tokens and client credential flows, effectively simulating real-world authentication scenarios.

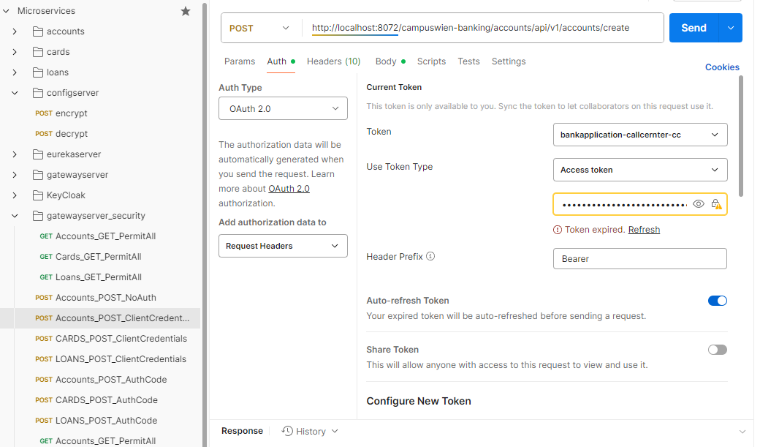
This Postman collection functions as both a validation tool for functional testing and a demonstration of the impacts of configuration strategies. By toggling between embedded and centralized configurations and invoking the same endpoints, one can observe the effects of configuration.

Figure 26:Postman

# CHAPTER – EXPERIMENTATION & RESULTS

## Performance and Maintainability

### Local Configuration

1. @Value Annotation – Local Configuration

This approach allows you to inject individual property values directly into specific fields within your application components. It is suitable for simple cases but can lead to hard-coded property keys scattered throughout the codebase.

This screenshot demonstrates how a Spring Boot application reads a configuration property and exposes it through a REST API.

The build. Version value is retrieved from the application's configuration file and injected into a variable using Spring's property injection mechanism. This allows the application to dynamically read values such as the current build version without hardcoding them.

An HTTP GET endpoint is provided at /build-info, which returns the value of the build. Version property. When a request is made to this endpoint, the application responds with an HTTP 200 status and includes the information in the response body.

This setup is useful for external systems or developers who need to check which version of the service is currently running. It supports better traceability, monitoring, and verification in environments where multiple deployments are involved.

A screen shot of a computer

AI-generated content may be incorrect.A screen shot of a computer

AI-generated content may be incorrect.

2. Using Environment Interface – Local Configuration

The Environment interface enables access to application properties programmatically. By auto wiring the Environment bean, developers can retrieve property values dynamically, offering greater flexibility. However, this method still involves manually specifying property keys in the code.

This screenshot demonstrates how a Spring Boot application uses the Environment interface to access configuration properties dynamically.

Instead of injecting individual values, the application declares an Environment object, which provides access to all properties within the application's environment. This approach allows the application to retrieve configuration values programmatically at runtime.

In this case, the environment is used to read a system or environment-specific variable and expose its value through a REST endpoint. When a request is made to the specified endpoint, the application responds with the value of the requested property.

This method is particularly useful when the property key is dynamic, or when conditional logic is needed to decide which property to access. It also provides more flexibility compared to direct injections but requires manual handling of property keys.



A screen shot of a computer

AI-generated content may be incorrect.

3. Using @ConfigurationProperties – Local Configuration

This is the most robust and scalable approach. The Configuration Properties annotation binds groups of related properties to a strongly typed bean, eliminating the need to hard-code property keys. This improves maintainability, readability, and allows validation of configuration values.

This example demonstrates how to use @ConfigurationProperties in a Spring Boot application to read grouped configuration values in a structured and scalable way.

Instead of injecting individual values or accessing properties directly through keys, this approach defines a dedicated class that represents a group of related configuration settings. The class is annotated with @ConfigurationProperties and linked to a specific prefix from the configuration file, allowing Spring Boot to automatically bind the corresponding values.

This strongly typed class is then made available in the application by enabling it through the @EnableConfigurationProperties annotation. The values can be returned or processed as needed, such as exposing them through a REST API.

A screenshot of a computer screen

AI-generated content may be incorrect.This method avoids hard-coding property keys, making the configuration more maintainable, readable, and easier to validate.

A screen shot of a computer code

AI-generated content may be incorrect.

4.Profile - Local Configuration

**Using Command-Line Arguments**

Spring Boot automatically maps command-line arguments into key-value pairs and makes them available through the Environment object. These arguments take the highest precedence, meaning they override values from any other source like property files or environment variables.

This is especially useful in production environments where configurations need to be injected dynamically at runtime. The naming convention matches the Spring property names and uses double hyphens (--) for passing values.

**Using JVM System Properties**

JVM system properties provide another way to externalize configuration. These are passed using the -D prefix and also override configuration from files.

This method allows properties to be injected during application startup without rebuilding the JAR file. When both JVM properties and command-line arguments are provided, command-line arguments take precedence.

This feature is useful for customizing builds or temporary overrides, especially in script-driven or automated deployment environments.

**Using Environment Variables**

Environment variables are a universal and platform-agnostic method for configuration. They work across operating systems and are often used in containerized or cloud environments.

To map environment variables to Spring Boot properties, variable names must be written in the uppercase and use underscores (\_) instead of dots (.). This is handled automatically by Spring Boot using relaxed binding.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

5. Local Configuration Problem

CLI arguments, JVM properties, and environment variables are effective ways to externalize configuration and maintain the immutability of the application build. However, using these approaches often involves executing separate commands and manually setting up the application, which can introduce potential errors during deployment.

Given that configuration data evolves and require changes, similar to application code, what strategies should be employed to store, track revisions and audit the configuration used in a release?

In scenarios where environment variables lack granular access control features, how can you effectively control access to configuration data?

When the number of application instances grows, handling configuration in a distributed manner for each instance becomes challenging. How can such challenges be overcome?

Considering that neither Spring Boot properties nor environment variables support configuration encryption, how should secrets be managed securely?

After modifying configuration data, how can you ensure that the application can read it at runtime without necessitating a complete restart?

### Centralized Configuration

In a distributed microservices architecture, managing configuration consistently across multiple services is crucial. Spring Boot, together with Spring Cloud Config, Spring Cloud Bus, and Spring Boot Actuator, provides a robust mechanism to externalize, manage, and dynamically refresh configuration without needing to restart applications.

The configuration values for each service are stored in a centralized Git repository. This repository includes multiple YAML files organized by service name and environment (e.g., accounts.yml, cards-prod.yml, etc.). These files are accessed by the Spring Cloud Config Server, which acts as a central source of truth for all configuration data. The Config Server is connected to the Git repository using a URI defined in its configuration. It pulls values from the specified branch (such as main) and serves them to client microservices like accounts, loans, and cards.

To allow applications to refresh their configuration at runtime, the Spring Boot Actuator module must be added to each microservice. This exposes endpoints such as /refresh or /bus refresh, which can be triggered to reload the configuration without restarting the application. The /refresh endpoint is a basic option that works for individual services, while /bus refresh enables broadcast-style refreshes across multiple services using a message broker like RabbitMQ or Kafka.

The basic refresh process begins when a developer commits changes to the configuration files in the Git repository. These updates are detected and fetched by the Config Server. A POST request to the /refresh endpoint of a running service triggers it to reload the updated configuration. However, in systems with many services or multiple instances per service, this manual process becomes inefficient.

To address this, Spring Cloud Bus comes into play. It connects the services via a messaging infrastructure, allowing configuration change events to be broadcast. When a POST request is sent to /actuator/bus refresh on any one instance, the event is distributed to all connected services, which then reload their configuration from the Config Server. This reduces the need to trigger refreshments manually for each instance.

For full automation, Spring Cloud Config Monitor can be used. This component exposes a /monitor endpoint on the Config Server. External systems like GitHub can be configured with a webhook that automatically sends a POST request to this endpoint after each push to the configuration repository. Once triggered, the monitor initiates a configuration refresh via Spring Cloud Bus, ensuring that all services reload their latest configuration in real time.

Additionally, to enhance visibility and security, management endpoints are enabled in the configuration to expose health checks and readiness/liveness probes. Sensitive properties can also be encrypted and stored securely in the configuration repo, with decryption handled at runtime using a shared encryption key.

In summary, the process begins with a change pushed to the Git configuration repository. This change is either manually or automatically propagated to the Config Server, which then refreshes connected services either individually or collectively using the message bus. This architecture supports seamless, scalable, and automated configuration management with minimal downtime and operational overhead.

A screenshot of a webhook

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.A computer screen shot of a code

AI-generated content may be incorrect.

### Docker compose

In modern microservice-based architectures, applications are often composed of multiple services that need to run together and communicate with one another. Managing these services individually using Docker CLI can become repetitive, error-prone, and difficult to maintain, especially when services depend on each other and require specific configurations. This is where Docker Compose becomes essential.

Docker Compose is a tool that simplifies the process of defining and running multi-container Docker applications. It allows developers to describe services, networks, volumes, environment variables, and dependencies in a single YAML configuration file. With one command, all services can be built, configured, and started together, eliminating the need to run and link each container manually.

Without Docker Compose, developers would need to manually run each container with the correct port mappings, volume mounts, network settings, and startup order. This creates a high risk of errors, increases deployment complexity, and makes automation difficult—especially when configuration needs to change across different environments like development, testing, and production.

In your current setup, Docker Compose is being used to orchestrate services such as RabbitMQ and the Config Server, along with shared configurations and environment-specific settings. Each environment (default, QA, production) has its own folder with a docker-compose.yml and common-config.yml, allowing for modular and reusable configuration.

The rabbit service includes health checks and port mappings, and it extends configuration from a shared file to define networking rules. The configserver depends on RabbitMQ, uses health checks to ensure it only starts when RabbitMQ is healthy, and imports properties from the centralized config server using Spring Cloud Config.

Shared configuration blocks such as microservice-base-config and microservice-configserver-config handle common settings like memory limits, environment variables, and Spring profiles. This promotes reuse, consistency, and easier management of resources.

Thanks to Docker Compose, all of these services can be started with a single command. Dependence is managed automatically, services wait for each other based on health checks, and all configuration details are version-controlled and environment-specific.

A screen shot of a computer

AI-generated content may be incorrect.A computer screen shot of a program

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

### Spring Boot BOM and Shared Libraries

In microservice architecture, managing dependencies and shared code across multiple services can quickly become complicated. Each service might use different versions of the same library, which can lead to version conflicts, bugs, and inconsistent behavior. To solve this, Spring Boot and Maven offer a solution called BOM, or Bill of Materials. A BOM helps define all dependency versions in one central place, so each microservice can automatically use the same versions without specifying them individually. This makes it much easier to keep everything consistent and simplifies upgrades when library versions change.

Another common challenge is code duplication. Many services might share the same utilities, configuration classes, or logging setup. Without a good structure, this shared code could be copied across projects or managed in a way that becomes difficult to maintain. One option is to create a shared library, but this can lead to large files with unused code. A better approach is to use a multi-module Maven project, where common code is placed in a separate module that other services can include. This avoids duplication and keeps the system modular.

To support this structure, the project includes helpful metadata such as developer information, organization URLs, and source code links, making it easy to track ownership and contribute across teams. It also uses automated Docker image naming based on the project name and version, helping keep deployments consistent.

Altogether, this setup improves the way microservices are developed and maintained. It ensures consistent dependency versions, avoids repeated code, and keeps configuration clean and reusable. This makes the development process more efficient and reduces the chances of issues during buildings, testing, and deployment.

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.A computer screen shot of a program

AI-generated content may be incorrect.

### Result

## Security

### KeyCloak

In a secure microservices environment, OAuth2 plays a crucial role in controlling and managing access to protected resources. The flow typically begins when a user initiates a request to access specific resources. This request is sent to the client application, which acts on behalf of the user. The client contacts the authorization server (e.g., Keycloak) to obtain an authorization code. This code proves the user has granted permission. Once received, the client uses this code along with its own credentials to request an access token from the authorization server.

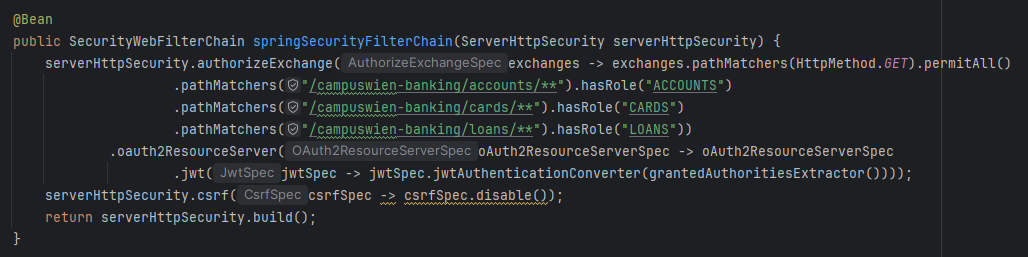
After receiving the access token, the client includes it in subsequent requests to the resource server (for example, a microservice behind Spring Cloud Gateway). Before granting access, the gateway validates the token by consulting the authorization server. If valid, the request proceeds, and the user gains access to the necessary data. This validation ensures that only authorized users can access protected endpoints.

On the resource server side, Spring Security is configured to control access to specific endpoints based on user roles. These roles are embedded within the access token as claims. To extract and interpret these roles, a custom converter is implemented. The KeycloakRoleConverter reads the realm\_access.roles claim from the JWT token and converts each role into a Spring GrantedAuthority, prefixed with ROLE\_. This allows fine-grained access control to be enforced through annotations or route-based rules.

To make this work, the application’s security configuration specifies which paths require which roles. For instance, requests to /campuswien-banking/accounts/\*\* might require the ACCOUNTS role. The Spring configuration also includes a URI pointing to the Keycloak public key set, which is used to verify the signature of JWT tokens.

On the client side, such as Postman, the access token is generated and attached to HTTP requests using the OAuth 2.0 mechanism. This token must be refreshed when it expired. Postman allows auto-refresh and easy inspection of the bearer token.

In Kafka-enabled services, security is equally critical. Messages may carry sensitive data or require role-based access. The token passed through HTTP headers may include roles that define what messages a consumer can read or produce. The extracted correlation ID in the custom logic helps trace and log requests across distributed services. It is injected into the request headers to maintain consistent context throughout the service flow.



A screenshot of a computer

AI-generated content may be incorrect.A computer screen with green and white text

AI-generated content may be incorrect.

### Docker Compose

In the Docker Compose configuration for the accounts microservice, you can observe that there is no port mapping defined. This means the container does not expose its internal ports directly to the host machine or the outside world. As a result, it is not possible to access the accounts service directly via browser or REST client by calling something like localhost:8080.

Instead, all access must go through the secure gateway, which acts as the single-entry point to all backend services. This architectural decision enforces centralized security, routing, and access control policies.

Additionally, in the environment configuration, the JWT Token verification is configured through the property:



"http://keycloak:8080/realms/master/protocol/openid-connect/certs"

This URI tells Spring Security where to fetch the public key needed to validate JWT tokens issued by Keycloak. It ensures that only requests carrying valid access tokens from a trusted source can reach the internal microservices through the gateway. This setup provides a strong security layer, combining network isolation with token-based authentication and authorization.

### Result

Ensures consistent identity/authentication configuration across services.

4. Service-to-Service Authorization

Local: 1. Managing Secrets (Passwords, Keys, Tokens)

Local Configuration:

Secrets (e.g., database passwords, API keys, OAuth credentials) are stored in each service’s application.yml or .properties file.

Risk of inconsistency or leaking secrets increases, especially if they are committed to version control by mistake.

Hard to rotate or revoke credentials across multiple services at once.

Centralized Configuration:

All secrets are stored securely in one place (e.g., Spring Cloud Config Server).

You can use encrypted values (with {cipher}) and an encryption key that the config server uses to decrypt at runtime.

Much easier to update or rotate secrets across the entire system, and access is centralized and auditable.

2. Protecting Configuration Endpoints

Local:

No central control over who accesses config files. If a service exposes actuator endpoints, they may leak info unless secured individually.

Each service must protect its own /actuator endpoints, leading to duplicated security config.

Centralized:

Config Server exposes /actuator/health, /encrypt, /decrypt, and possibly /monitor.

Only the Config Server needs to be secured tightly; clients simply consume config via secure internal communication.

Easier to implement role-based access control or authentication at a single point.

3. Gateway & OAuth2 Security Configuration

Local:

OAuth2 settings (client ID, secret, auth server URL) must be added manually to each service.

Risk of drift between services or incorrect settings.

If a secret changes, all services need to be redeployed manually.

Centralized:

Shared OAuth2 settings stored in Git or Vault, loaded by the Config Server.

One change in the Git config and a /refresh or /busrefresh will update all services at once.

Hardcoded or mismatched tokens and credentials may be used between services.

Manually updated, increasing the risk of broken authentication or leakage.

Centralized:

Shared service credentials can be managed centrally and rotated securely.

Policies and tokens can be enforced consistently across services.

# CONCLUSION

Bibliography

|  |  |
| --- | --- |
| [1] | Sam Neman, Building Microservices. |
| [2] | U. Alker and U. Weilenmann, *Sprachleitfaden, Gendergerechter Sprachgebrauch an der FH Campus Wien,* FH Campus Wien, 2006. |
| [3] | A. Petz and R. Oberpertinger, *Checkliste Wissenschafltiche Arbeiten,* FH Campus Wien, 2018. |
| [4] | M. Young, The Technical Writer's Handbook, University Science Books, 2002. |

List of Figures

[Figure 1 .Strangler Fig Pattern 2](#_Toc205326226)

[Figure 2: Microservices Overview 6](#_Toc205326227)

[Figure 3 : Monolithic vs. SOA vs. Microservices 7](#_Toc205326228)

[Figure 4: Challenges in Microservices 13](#_Toc205326229)

[Figure 5: Architecture Overview 23](#_Toc205326230)

[Figure 6: Microservices in Application 26](#_Toc205326231)

[Figure 7: Account Maven Dependance Example 27](#_Toc205326232)

[Figure 8: Customer Class Entity Example 28](#_Toc205326233)

[Figure 9: Data Transfer Object Example 29](#_Toc205326234)

[Figure 10: Account Repository Example 30](#_Toc205326235)

[Figure 11: Account Controller Class Example 31](#_Toc205326236)

[Figure 12: Global Exception Class Example 32](#_Toc205326237)

[Figure 13: Application.yml file Example 33](#_Toc205326238)

[Figure 14: API Example for Account micro service 1 35](#_Toc205326239)

[Figure 15: API Example for Account micro service 2 36](#_Toc205326240)

[Figure 16: API Example for Account micro service 3 37](#_Toc205326241)

[Figure 17: API Example for Account micro service 4 38](#_Toc205326242)

[Figure 18: Account Info for Docker Image 39](#_Toc205326243)

[Figure 19: Config Application YAML File 42](#_Toc205326244)

[Figure 20: Config Server GitHub 42](#_Toc205326245)

[Figure 21: Eureka Discovery 43](#_Toc205326246)

[Figure 22: Eureka Example 43](#_Toc205326247)

[Figure 23: Eureka heartbeat Example 44](#_Toc205326248)

[Figure 24:Card Service Feign Client 45](#_Toc205326249)

[Figure 25: Circuitbreaker Example 46](#_Toc205326250)

List of Tables

[Table 1: Monolithic vs. SOA vs. Microservices Comparison 8](#_Toc205322359)

[Table 2: Centralized vs. Local Configuration 15](#_Toc205322360)

Appendix

1. https://samnewman.io/books/building\_microservices\_2nd\_edition/<https://www.researchgate.net/publication/378111158_Evolution_of_Microservices_Identification_in_Monolith_Decomposition_A_Systematic_Review>
2. <https://ahrefs.com/writing-tools/paragraph-rewriter>
3. <https://article.arunangshudas.com/managing-configuration-in-a-microservice-environment-b2fddba31f28>
4. <https://www.geeksforgeeks.org/microservices/>
5. <https://www.duo.uio.no/bitstream/handle/10852/95663/1/Master-thesis--Eivind-Solberg.pdf?utm_source=chatgpt.com>
6. <https://miladezzat.medium.com/microservices-architecture-benefits-challenges-and-use-cases-9cec05adcb57>
7. <https://epub.jku.at/download/pdf/8073606.pdf>
8. <https://www.solo.io/topics/microservices/microservices-securitys>
9. <https://vfunction.com/blog/microservices-architecture-guide/>
10. <https://www.docker.com/>
11. <https://spring.io/projects/spring-boot>
12. <https://git-scm.com/>
13. <https://spring.io/projects/spring-cloud>
14. <https://www.keycloak.org/>
15. <https://kafka.apache.org/>
16. <https://www.rabbitmq.com/>
17. <https://kubernetes.io/>
18. <https://helm.sh/>
19. <https://www.postman.com/home>
20. https://www.java.com/de/