Comparison between Monolithic and Microservices Architecture in Web Applications Built with Java

Ina Papadhopulli, Armada Alija
Faculty of Information Technology, Polytechnic University of Tirana, Albania
E-mail: ipapadhopulli@fti.edu.al, armada.alija@fti.edu.al

Abstract: *In today's times, people demand fast, stable, and reliable applications. Therefore, it is essential to choose the right software architecture before implementing functionalities adopted in the project. The monolithic architecture is implemented as a single unit, integrating all functionalities within a single codebase. In contrast, the microservice architecture decomposes the application into several independent services, each responsible for a specific business logic. Key factors such as scalability, fault tolerance, and data consistency are analyzed. Performance metrics, simplicity, flexibility, and the challenges of implementing and maintaining each architecture are gathered and compared. This study offers valuable insights into the practical implications of choosing between monolithic and microservice architectures, especially in the context of applications. It serves as a comprehensive guide for software developers to make informed decisions when designing scalable and reliable systems.*
Keywords*: ARCHITECTURE, MONOLITH, MICROSERVICE, APPLICATION, JAVA*

1. Introduction

Monolithic architecture has traditionally been used and continues to be utilized. Based on this, applications are built on a single database, and functionalities are closely interconnected, following a linear structure. However, while this model is simple, it can become very difficult and complex to manage as the application grows. Some issues that can be mentioned include the dependency between project modules, longer integration time for new developers, and longer deployment times.

Therefore, to address this issue, Microservice architecture promises to simplify project management and maintenance as things become more complex.[1] Many companies are interested in migrating their applications to microservices. However, this is not an easy path. This challenge will be explored through a case study approach, by implementing an application based on Java, utilizing both types of architectures. The problem can be formulated as:

What are the key challenges and benefits of transitioning from monolithic architecture to microservices architecture in developing and managing scalable applications?

2. Objectives and Research Methodologies

The main goal is to present the elements that demonstrate the connection between Monolithic and Microservices architectures, analyze or highlight their specific characteristics, and compare the performance and scalability of monolithic and microservice architectures in a web application. Additionally, the necessary mechanisms for transitioning from Monolithic to Microservices in the most practical way have been evaluated.

The defined objectives are listed as follows:

- Introduction of two architectures: monolithic and microservices and a practical case study comparing their characteristics.
- Quantitative evaluations of performance metrics such as responsivity, scalability, and fault tolerance.
- Insights into best practices
- Highlighting the challenges that arise during the use of each of them.

3. Case Study Implementation

Application**:** Sales Management Platform is a simple Java-based web application that is implemented twice and uses both architectures. It includes functionalities for user authentication, warehouse management, client data, and order processing.[19]

Technologies**:** Java, Spring Boot, Spring JPA, Hibernate

Java is a platform-independent, object-oriented programming language designed for high performance and versatility, enabling the development of secure, scalable, and portable applications.[3]

Spring Boot is a Java-based framework that simplifies application development by providing pre-configured setups.

Spring JPA is a module of the Spring Framework that provides a repository abstraction by enabling efficient and declarative interaction with relational databases.

Hibernate is a Java-based ORM framework that implements JPA, facilitating the mapping of Java objects to relational database tables while automating SQL generation and transaction management.

Database: PostgreSQL.

PostgreSQL is an advanced, open-source relational database management system that supports both SQL (relational) and JSON (non-relational) querying.

IDE: IntelliJ IDEA

IntelliJ IDEA is an integrated development environment (IDE) developed by JetBrains and used for developing computer software written in Java, Kotlin, Groovy, Scala, and more.

Testing**:** Postman, Gatling, Resilience4j

Postman is a tool for manual and functional API testing. It includes debugging, simple automation, and validation.

Gatling is a tool for performance and load testing. It is designed for high-concurrency simulations of API or web service performance.

Resilience4j is a library that evaluates fault tolerance and durability. It is used to implement fault-tolerance patterns like Circuit Breaker, Retry, Timeout, Bulkhead, and Rate Limiter, in applications. Also, it can be used to simulate fault scenarios for testing purposes when combined with tools like JUnit.

Device: The application is developed and tested on an HP Laptop with an Intel Core i7, 16GB RAM, x64-based processor, 2.7 GHz, and Windows 10 operating system.

3.1 Monolithic Architecture

Setup**:** Single codebase and database for all functionalities.[19]

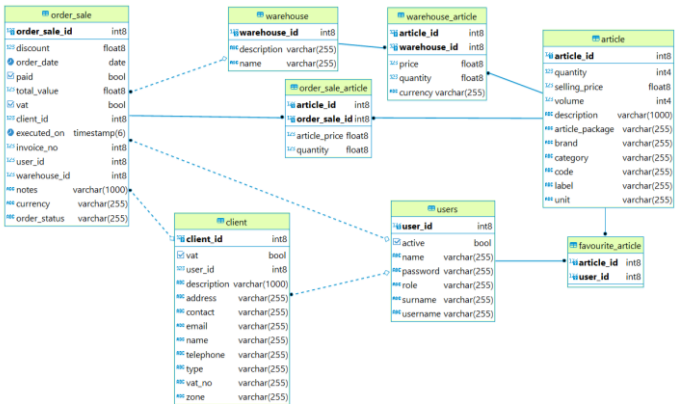

Fig. 1 *Entity Relationship Diagram (ERD) for the Sales Management Platform implemented in a Monolithic Architecture*

The monolithic system is implemented as a single WAR file, deployed on a Tomcat server. The PostgreSQL database contains tables for users, clients, orders, and warehouse inventory.

Performance**:** Using Gatling, the monolithic system handled up to 10,000 – 100,000 concurrent requests with acceptable latency. However, increasing requests beyond this threshold resulted in significant degradation, as all processes relied on a single deployment instance.[17]

Limitations and Fault Tolerance[18]:

Single Point of Failure: A monolithic application is typically a single, tightly integrated unit. A failure in any part of the application, e.g., a database connection failure or a critical module, can bring down the entire system.

Performance Bottlenecks: As all components share the same runtime environment and resources e.g. CPU, and memory, it can lead to performance issues. A bottleneck in one part of the application, such as excessive database queries, can slow down the entire system.

Deployment bottleneck: Re-deploying required restarting the entire system, causing downtime. Since monolithic applications are deployed as a single unit, failures due to incompatible changes in a module or server issues can bring down the entire system. This also limits the ability to perform isolated updates or rollbacks.

Resilience4j patterns that are used to handle these failures[12]:

Circuit Breakers and Timeouts: These are essential for ensuring that failure in one part of the application does not bring down the entire system. For example, a service call within the monolith might fail due to database issues or external dependencies, and a circuit breaker prevents this failure from affecting the rest of the application.

Bulkheads: These are used to isolate critical internal actions, e.g. order execution, from other less critical actions, ensuring that failure in one section doesn't disrupt the entire application

Retries: Retries help resolve temporary issues, such as brief network failures, by retrying operations before failing altogether.

3.2 Microservices Architecture

Setup**:** Individual services for authentication, warehouse, client, orders and favorite articles. Each service has its own database.[19]

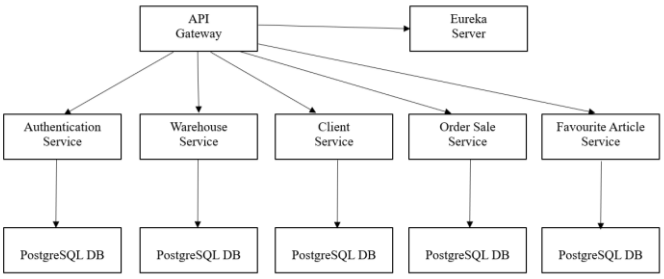

Fig. 2 *Microservices Architecture for the Sales Management Platform*

Eureka Server is a popular tool for implementing service discovery and registration in microservices architectures. Eureka acts as a centralized registry where all service instances register themselves. Other services can use Eureka to dynamically find the location of instances they want to communicate with. OpenFeign helps to simplify this communication by acting as a declarative HTTP client.[16]

API Gateway acts as a single entry point for all client requests, routing them to the appropriate microservices. It serves as a service aggregator and load balancer, enabling dynamic routing to various backend services that are registered with a service discovery tool such as Eureka. The API Gateway interacts with service registries to dynamically discover the locations of microservices, facilitating fault tolerance and scalability by rerouting requests to available instances in case of failure.

The Users table will be included in the Authentication Service and is related to system authentication. The Article, Warehouse, and WarehouseArticle tables will be included in the Warehouse Service. The Client table will be included in the Client Service. The OrderSale and OrderSaleArticle tables will be included in the Order Service. The FavouriteArticle table will be included in the Favourite Article Service.

Implementation**:** Each functionality was split into independent services, each with its own database. For instance, the Warehouse Service managed inventory-related operations using its specific PostgreSQL instance. [11]

Performance**:** Gatling tests revealed that microservices scaled effectively, handling over 100,000 concurrent users by distributing workloads across services.

Limitations and Fault Tolerance[20]:

Load Balancing Failure: Load balancing in microservices ensures that incoming traffic is distributed across multiple instances of services to avoid overloading a single instance, maintaining high availability and responsiveness. Resilience4j's Rate Limiter and Circuit Breaker help mitigate traffic overloading and prevent cascading failures in a load-balanced system.

Database Failure: Database failures can impact any service that depends on data persistence. These failures can be caused by issues with the database itself or the interactions between microservices and the database. Circuit Breakers and Retry Patterns help ensure service reliability by enforcing limited retry logic and ensuring that requests don't overload the database.

Service Unavailability: Service unavailability occurs when one or more microservices in the architecture fail, preventing requests from being processed. This failure can be due to network issues, crashes, or resource exhaustion. Circuit Breaker and Fallback mechanisms contribute to improving system availability and reducing the impact of service unavailability.

Throttling: Throttling is a method used in microservices architectures to control the flow of requests to a service to prevent overload and ensure fair usage. Rate Limiter and Bulkhead patterns enable fine-grained control over how requests are processed, preventing service overloads while ensuring fairness and optimal resource usage.

Authentication/Authorization Failure: Authentication and authorization failures occur when a service cannot verify the identity of a user or system, or when a user/system is not authorized to perform a certain action. By analyzing the security breach or token expiration processes, Retry and Circuit Breaker patterns can be used to minimize the risk of authentication failure affecting the system as a whole[14].

4. Comparative Analysis of Monolithic and Microservices Architecture

	Monolithic Architecture	**Microservice Architecture**
Component	Module	Microservice
Component size	Large	Limited
Elasticity	Single point of failure. Hard to handle new changes	Each service can be scaled independently
Deployment	The whole application is deployed as a single module	Each service is created and deployed independently
Storage mechanism	Shared database	Private database
Technology	The programming language and framework used must be the same from the start to the end of the application	Heterogeneous languages and frameworks
Resiliency	A single bug can compromise the entire system	Issues in one service do not affect the functionality of others
Agility	Adopting new technologies is difficult due to the tight-knit structure	Easily integrates new technologies to meet business needs
Performance	Performance can degrade as the system grows due to resource bottlenecks	Performance is optimized since each service can run independently, allowing specific tuning for performance gains
Durability	Single database and component structure may reduce long-term durability if updates are not frequent	Higher durability since individual services can be updated and maintained without disrupting the whole system
Fault-Tolerance	Lower fault tolerance; a critical failure in one module can affect the entire system	High fault tolerance, as a failure in one service, does not cascade to others, keeping most of the application functional
Scalability	Inability to scale when needed	Scalable (scale on demand)
Maintenance	A large code base is difficult to understand	A small code base is easier to be managed
Security	Secure, as a single system, data management is straightforward	Security is complex due to inter-service communication, requiring additional safeguards like API gateways

Table. 1 *Comparison between Monolithic and Microservices Architecture[4],[5],[6][7],[8],[13].*

5. Challenges of transition to Microservices

Despite its simplicity, monolithic-based applications can become very challenging and complex to manage as the application scales.[2] Some issues include:

- Dependencies between project modules
- Longer integration time for new developers
- Extended deployment times

To address these challenges[10], microservice architecture promises to simplify project management and maintenance as complexity increases. Many companies are interested in migrating their applications to microservices, but this is not an easy path.[15]

The microservices concept is to separate each business logic into distinct services. Each microservice is considered an independent API application with its lifecycle, allowing for independent development, deployment, and scalability according to customer needs, and management.[12]

Challenges observed during the migration to microservices include:

- Separating the database of the application built with monolithic architecture
- Managing transactions within microservices

- Ensuring the normal operation of the monolithic application until the microservices architecture is complete
- Adapting developers to the new architecture

6. Conclusions

Monolithic architecture, being a traditional and straightforward model, offers simplicity in deployment and development. It is an excellent choice for small-scale applications where modularity and scalability are not the primary concerns, as selecting a microservices architecture would not make a significant difference. So, for small applications, monolithic architecture allows for easier debugging, reduced latency in internal system communication, and simplified management due to a unified codebase.

However, as applications grow, the inherent limitations of monolithic structures become evident, particularly in terms of scalability and maintenance. The tight coupling of components makes the system more rigid, slowing down development processes and increasing the likelihood that updates or changes will introduce unwanted side effects throughout the system. Furthermore, deploying new versions requires the entire application to be restarted, which is inefficient for large and complex systems.

While monolithic architecture may be more suitable for smaller, less complex applications, microservices offer significant advantages in terms of scalability and fault isolation for larger and more dynamic systems.

7. References

1. Newman, Building Microservices., O'Reilly Media, Inc., (2021)
2. Momil Seedat, QaisarAbbas Qureshi, Alessia Amelio, Transition Strategies from Monolithic to Microservices Architectures: A Domain-Driven Approach and Case Study. (June 2024) https://www.researchgate.net/publication/381290696_Transition_Strategies_from_Monolithic_to_Microservices_Architectures_A_Domain-Driven_Approach_and_Case_Study
3. Desiree D. Martinez, Axl Heart P. Remegio, Darllaine R. Lincopinis, A Review on Java Programming Language, (May 2023) https://www.researchgate.net/publication/371166744_A_Review_on_Java_Programm ing_Language
4. J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, (March 5-6, 2018)
5. Nada Salaheddin, Nuredin Ali Salem Ahmed, Microservices vs. Monolithic Architectures: The differential structure between two architectures, (9 Sept 2022) https://www.minarjournal.com/dergi/microservices-vs-monolithic-architectures-thedifferential-structure-between-two-architectures20221202031410.pdfscal
6. E. Axelsson and E. Karlkvist, "Extracting microservices from a monolithic application," (2019)
7. Grzegorz Blinowski, Anna Ojdowska, Adam Przybylek, Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation, IEEE, (18 February 2022)
8. C. Richardson, "Microservices-Pattern: Microservice Architecture," (March 2014) http://microservices.io/patterns/microservices.html
9. Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, Zhang Li, Microservices: architecture, container, and challenges, (2020) IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C) https://qrs20.techconf.org/QRSC2020_FULL/pdfs/QRS-C2020-4QOuHkY3M10ZUl1MoEzYvg/891500a629/891500a629.pdf
10. Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, Microservices: The Journey So Far and Challenges Ahead, IEEE Software,Published by the IEEE Computer Society, (2018) https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8354433
11. A. Furda, C. Fidge, O. Zimmermann, W. Kelly and A. Barros, "Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency," in IEEE Software, vol. 35, no. 3, pp. 63-72, (May/June 2018)
12. Mohit Kumar; Christian Engelmann, Models for Resilience Design Patterns**,** (2020) https://ieeexplore.ieee.org/document/9306940
13. Taibi, K. Systä, From monolithic systems to microservices: A decomposition framework based on process mining, (2019), pp. 153–164
14. Architecting with microservices: A systematic mapping study J. Syst. Softw., 150 (2019), pp. 77-97
15. Victor Velepucha; Pamela Flores, Monoliths to microservices - Migration Problems and Challenges: A SMS, (23-25 March 2021)
16. Yalemisew Abgaz; Andrew McCarren; Peter Elger; David Solan; Neil Lapuz; Marin Bivol,Glenn Jackson, Murat Yilmaz, Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review, pp. 4213 – 4242, (June 2023) https://ieeexplore.ieee.org/document/10160171
17. Konrad Gos, Wojciech Zabierowski, The Comparison of Microservice and Monolithic Architecture, April 2020 https://www.researchgate.net/publication/341956559_The_Comparison_of_Microservice_and_Monolithic_Architecture
18. E Punithavathy, N Priya, Performance of Dynamic Retry Over Static Towards Resilience Nature of Microservice, (June 2024) https://www.researchgate.net/publication/381136184_Performance_of_Dynamic_Retry_Over_Static_Towards_Resilience_Nature_of_Microservice
19. Kapil Bakshi, Microservices-based software architecture and approaches, IEEE, (2017) https://ieeexplore.ieee.org/document/7943959
20. Muhammad Waseem; Peng Liang; Gastón Márquez; Amleto Di Salle, Testing Microservices Architecture-Based Applications: A Systematic Mapping Study, IEEE, (2020) https://ieeexplore.ieee.org/document/9359275