10th International Scientific Conference
Technics, Informatics, and Education – TIE 2024
20-22 September 2024

Session: *Computer Sciences and Information Technology*
Professional paper
DOI: 10.46793/TIE24.042S

Enhancing Software Development with Microservice Architecture: Application to an Online Sales System

Biljana Savić[1*] [0000-0002-2544-6186] and Uroš Milačić[1] [0009-0003-9198-4104]
[1] University of Kragujevac, Faculty of Technical Sciences Čačak, Serbia
[*] biljana.savic@ftn.kg.ac.rs

Abstract: *This paper explores the critical role of microservice architecture in modern software development, illustrating its benefits through the creation of an online store as an example. Microservice architecture is highlighted for its capacity to improve scalability, maintainability, and deployment efficiency by decomposing applications into modular, independently deployable services. The approach facilitates a more robust and flexible system design, allowing for easier updates and better resource management. This study underscores the theoretical advantages of microservices, such as enhanced fault isolation and continuous delivery, while providing practical insights into its implementation. The online store example serves as a practical demonstration of these concepts, showcasing how microservice architecture can lead to more efficient and reliable software solutions.*

Keywords: *microservice architecture, scalability, software solution, continuous delivery*

1. INTRODUCTION

Microservice architectural style is a software development approach in which an application is divided into small, compact services that can be developed, deployed and scaled independently. Each microservice performs a specific business function and communicates with other microservices through well-defined API [1]. As it is well-known, enterprise applications are often built in three main parts: a client-side user interface (consisting of HTML, CSS and JS), a relational database management system (program used to create, update and manage relational databases) and a server-side application. Mostly, the purpose of this application is to handle HTTP requests coming from client-side, retrieve and update data from database and prepare response model in a format that is acceptable by client-side, which will later select and populate its HTML views. In this example, the server-side application acts as a single local executable, or in other words – a *monolith*.

For better understanding of what microservice architecture is, it is useful to compare it to the monolithic style, where all logic for handling a request runs in a single process and can be tested on a locale and deployed into production. As monolithic application puts all its functionality into a single process, the conclusion of deployment issue is very simple: any changes to the system involve building and deploying a new version of the server-side application, and over time, it is often hard to keep a good modular structure. Scaling requires scaling of the entire application rather than parts of it that require greater resource [2].

These adversities led to the microservice architectural style: building applications as suites of services, and the objective of this paper is to highlight the importance and benefits of using this architectural style through practical example in a case study of an online sales system application.

2. MICROSERVICES OVERVIEW

Throughout the years, the microservice architectural style was defined based on common patterns observed across a number of pioneering organizations, where they did not consciously implement microservice architecture [3]. While they evolved to it in pursuit of specific goals, each of them was tied back to the "balancing speed and safety at scale."

Microservice architectural style is even more and more used due to its various benefits, and some of them include:

- Scalability: microservice architecture allows applications to scale up or down quickly based on demand. Developers can add or remove microservices as needed, without affecting the rest of the application.
- Resilience: Microservices are designed to be resilient to failures, meaning if one microservice fails, it does not bring down the entire application.

- Fault isolation: Corresponding to the previous benefit, developers can isolate failures, preventing them from spreading and causing widespread issues.
- Technology independence: Microservices architecture allows developers to be flexible when choosing technologies, meaning it does not need to be tied to a single technology stack.
- Easy maintenance: Since microservices are concise and domain focused, they are easier to maintain than monolithic applications.

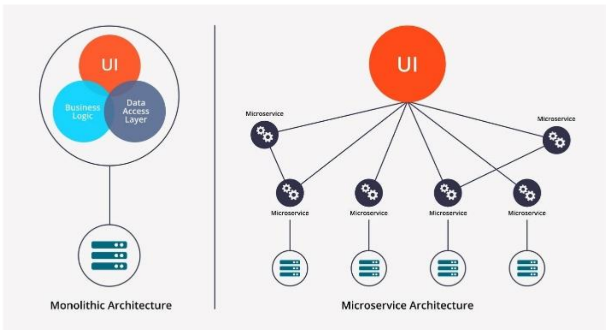

Figure 1. *Monolithic and Microservice architectural style*

Theoretically, we could apply microservice architecture to a washing machine program. However, the question remains whether there is a need to go that far, considering the challenges associated with building software solutions using microservice architecture:

- Development complexity: Each service is a separate entity with its own codebase, making the development process more complex.
- Communication overhead: Microservices communicate over a network which can increase latency and potentially communication failures.
- Data management issues: Each service will mostly have its own database, leading to challenges in ensuring data consistency and managing distributed data stores.
- Deployment complexity: Implementing continuous deployment practices becomes more complex with microservice architecture.

While microservice architecture offers variety of benefits, in some use cases it could bring unnecessary complexities. As it is important to weigh these disadvantages against the benefits to determine which architectural style is the right fit for specific system, it is also worth mentioning that the key in choosing between architectures when designing a program is modularity – the quality of an application being composed of distinct, self-contained parts that, when combined, form a complete whole. In simpler terms, an application is considered modular if each of its components is functional and self-sufficient on its own, eliminating the need for further expansion to maintain or enhance its effectiveness.

3. MICROSERVICE COMMUNICATION

Microservice architecture relies on the division of a large application into smaller, loosely coupled services, each focusing on a specific business function. Effective communication between these services is crucial for the system's overall functionality and performance. In the microservice architecture, all components of the applications run on several machines as a process or service, and they use inter-service communication to interact with each other. Microservices frameworks usually execute a consumer grouping mechanism whereby different instances of a single application have been placed in a competing consumer relationship in which only one instance is expected to handle an incoming message [4]. There are two primary modes of communication in microservice architecture:

- Synchronous communication style
- Asynchronous communication style

Synchronous communication is often regarded as request/response interaction style and pattern. One microservice makes a request to another service and waits for the services to process the result and send a response back. This method involves a service making a request to another service and waiting for an immediate response before continuing its execution. This pattern ensures direct and instant data exchange, which is essential for operations requiring immediate feedback or coordination between services.

Synchronous communication is typically implemented using protocols like HTTP/HTTPS, often through RESTful APIs, which are favored for their simplicity and wide adoption. Another popular protocol for synchronous communication is gRPC, which uses HTTP/2 and Protocol Buffers to achieve low-latency, high-performance communication.

The synchronous approach allows for straightforward error handling and straightforward service-to-service communication patterns, but it also introduces potential challenges, such as increased latency and tighter coupling between services. The decision to use synchronous communication in a microservice architecture should consider these trade-offs, focusing on scenarios where immediate response times are critical and the system can manage the dependencies and potential bottlenecks introduced by this communication style.

The asynchronous form of communication can be implemented in microservices when services exchange messages with each other through a message broker.

This method allows a service to send a request and continue its execution without blocking for a reply, making it well-suited for tasks that do not require immediate feedback. Asynchronous communication is commonly implemented using message brokers like RabbitMQ, Kafka or AWS SQS, which handle message queuing and delivery between services.

This approach can significantly improve system scalability and fault tolerance, as services can operate independently and handle peak loads by processing messages at their own pace. Moreover, it enables more flexible interaction patterns, such as event-driven architectures, where services react to events asynchronously.

However, this method also introduces complexities in terms of ensuring message delivery, handling message ordering, and managing eventual consistency. Despite these challenges, asynchronous communication is a powerful strategy for building robust, scalable, and decoupled microservice architectures that can efficiently manage varying workloads and improve overall system performance.

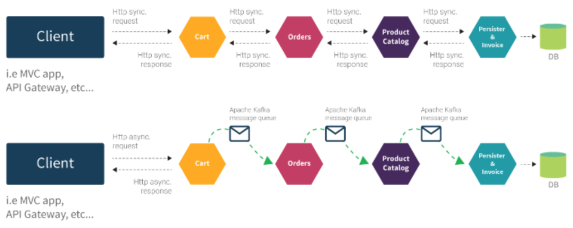

Figure 2. *Synchronous and asynchronous communication between microservices*

Choosing between synchronous and asynchronous communication in a microservice architecture depends on various factors, including system requirements, performance considerations, and complexity trade-offs [5]:

- Latency: If the application or a system requires immediate responses and low latency, synchronous communication via REST API or gRPC may be more suitable.
- Scalability: Asynchronous communication is often preferred for its scalability benefits. It allows services to handle varying workloads by decoupling the sender from the receiver, enabling better scalability compared to synchronous communication, where services may become blocked due to high request volumes.
- Resilience: Asynchronous communication promotes resilience by allowing services to continue processing requests even if a downstream service is temporarily unavailable.
- Consistency: If strong consistency is necessary, synchronous communication may be more suitable, since asynchronous communication can introduce eventual consistency challenges, where data may be temporarily inconsistent across services.

4. CASE STUDY

This case study focuses on the practical aspects of adopting microservices to improve scalability, resilience, and flexibility in the online sales domain. It involves breaking down functionalities into separate, independently deployable services, each serving specific business functions.

Through this case study, the practical advantages and challenges of employing microservice architecture in online sales systems will become apparent, providing valuable insights for businesses in the digital commerce sphere.

We will discuss an example case study of an online sales system, which highlights the usage and importance of microservice architecture with particular attention to repository and service design patterns which utilize and showcase the benefits of microservice architectural style.

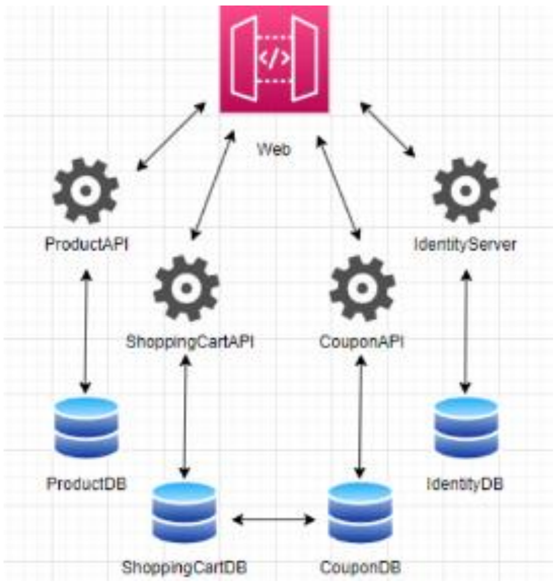

Figure 3. *Microservice overview on the practical example of the case study*

4.1. Repository pattern

The Repository pattern is a structural design pattern that isolates data and the data access layer from the rest of the application and its logic. This design pattern has two main objectives: to provide a separate space for communicating with the database, which is separated from the rest of the application, thus establishing a level of access for other layers of the application when it comes to data communication, and to handle the implementation logic of persistent storage of the application, which is necessary to retrieve data from the database [6]. This pattern can also be combined with other design patterns to significantly increase its functionality and to efficiently expose consistent APIs. In other words, the purpose of this pattern is to unify all methods for accessing database tables in one place, and such methods would be implemented through specific classes using the repository interface as the type of object access for the desired domain. The advantage of this pattern lies in having all the essential database-related logic in one place, and if changes need to be made to the code, they can be done in one place. Repository pattern's benefits extend well into microservice architecture, aiding in its scalability, maintainability, and overall robustness:

- Isolation of Data Access: The Repository pattern allows each microservice to encapsulate its data access logic within its own repository implementation. This isolation ensures that changes to the underlying data store or database schema only affect the repository implementation within that microservice, reducing the risk of unintended side effects on other services.
- Consistency: By providing a standardized interface for accessing data, the Repository pattern promotes consistency across microservices. This consistency makes it easier for developers to understand and interact with different services, as they can rely on familiar patterns and APIs for data access.
- Improved testing and debugging: With data access logic encapsulated within repositories, testing and debugging become more manageable. Developers can easily mock repository implementations for unit testing, and issues related to data access can be isolated within the boundaries of individual microservices.
- Modularity: The Repository pattern enhances the modularity of microservices by decoupling data access logic from the rest of the application. This decoupling allows individual services to scale independently, as changes to one service's data access logic do not affect others.

4.2. Unit of work pattern

It is important to mention that the Repository pattern is often used in conjunction with the Unit of Work pattern to facilitate efficient data access and persistence. It's primarily concerned with managing transactions and ensuring data consistency within an application. It encapsulates the transaction management logic, coordinating transactions across multiple repository operations. This ensures that a group of related database operations either succeed or fail together, maintaining data consistency. When changes are made to objects retrieved from repositories, the Unit of Work pattern tracks these changes and ensures that they are persisted to the database when the transaction is committed. This involves coordinating with the respective repositories to save or update the modified objects.

The Unit of Work typically has a scoped lifetime within a unit of work session, where multiple repository operations are performed within the same transactional context. Once the unit of work session is completed, the changes are either committed or rolled back, ensuring transactional consistency.

4.3. Service layer pattern

The Service Layer pattern is a structural, service-oriented design pattern aimed at organizing services and their functionalities by separating them into a service layer [7]. Services categorized in this manner have the ability to share functionalities, thereby reducing code redundancy. This approach also enhances efficiency and maintainability, as any changes to services are confined to the service layer. The purpose of this encapsulation is to minimize the impact on other parts of the code outside this layer. In other words, if there is an error in a service, the programmer will most likely find the error within the mentioned layer.

Services within this layer need to be defined generically enough to be reusable, with their functions tied to solving domain-specific problems of that service. The Service Layer pattern is an essential step in designing software and systems based on microservice architecture. Service layer sits between the presentation layer and the data access layer, usually in line with Repository pattern, or it can be above Repository pattern but below the presentation layer, providing a clear separation of concerns [8].

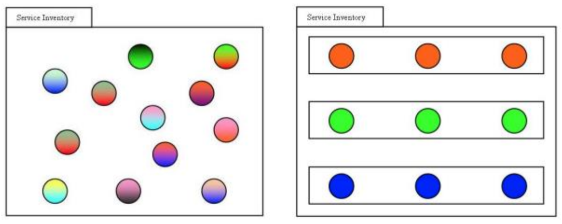

Figure 4. *Simple view of how service layer pattern applies its function to sort domain logic*

The Service Layer provides several essential characteristics that are crucial for the microservice architectural style:

- Scalability: Microservices can be scaled independently based on their specific needs. The service layer ensures that each microservice handles its business logic efficiently, aiding in overall system scalability.
- Reusability: Business logic encapsulated in services can be reused across different parts of the application or even across different applications.
- Autonomy: Microservices are designed to be autonomous. With the Service Layer pattern, each microservice independently manages its business logic, reducing dependencies and improving fault isolation.

- Interoperability: The service layer can expose a well-defined API, facilitating communication between microservices. This standardization simplifies integration and interaction between various services within the microservice architecture.

4.4. Base service implementation

Considering that there are four microservices in the practical example mentioned above, alongside these microservices, there exists a central project responsible for rendering pages, or to put it briefly, it is a client-side of the application. However, this project also has implementation of base service, acting as a gateway and facilitating communication between the client-side and the backend microservices, popularly called API Gateway, with a small footnote that the authorization and authentication strategy is solved by a separate microservice. By employing this architecture, the system achieves a separation of concerns, enabling flexibility, resilience, and the ability to independently scale and evolve its components.

This base service acts as a single-entry point for clients to access these microservices within the server-side application. It essentially functions as a reverse proxy that routes incoming requests from client-side to the appropriate microservices while abstracting the complexities of the underlying architecture. One of the primary purposes of this service is to address the challenges associated with microservices, such as service discovery, load balancing, authentication, authorization, and request routing. By consolidating these concerns into a centralized component, the API gateway simplifies the interaction between client-side application and the microservices.

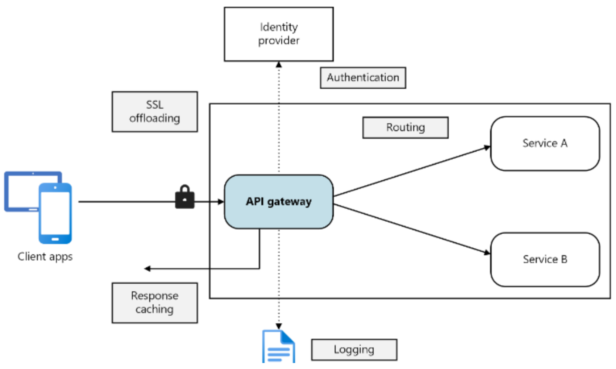

Figure 5. *Implementation of API Gateway (in our case, base service)*

Furthermore, this base service performs additional tasks such as request/response transformation, protocol translation, caching, and logging. These capabilities enhance the performance, security, and resilience of the microservices by offloading common functionalities from individual services to the gateway. From a client perspective, the base service represents a unified interface, shielding them from the complexities of the underlying microservices architecture. Client-side application interacts with the base service using standard protocols and conventions, while the service handles communication with the three specific domain microservices, and the identity server microservice which is responsible for authentication and authorization (this microservice also facilitates the implementation of cross-cutting concerns such as security policies and monitoring, offering a centralized point for enforcing policies and collecting metrics across the entire microservices ecosystem).

However, there is a potential risk to designing this base service, since it could become a single point of failure or a performance bottleneck [9]. To avoid such situations, it's essential to design this concept carefully, applying several strategies such as horizontal scaling, fault tolerance mechanisms, and intelligent routing, who can help mitigate these risks while ensuring the reliability and scalability of the microservices architecture.

Overall, this base service plays a pivotal role in orchestrating communication between client-side application and microservices on server-side application, simplifying development, enhancing security, and improving the overall performance and manageability of distributed systems.

5. CONCLUSION

Microservice architecture leverages server development expertise to enhance the flexibility, scalability, and maintainability of applications. By decomposing complex business system into smaller, independently deployable services and applying this architecture to a smaller system, such as an online sales platform, allows for independent development and deployment of features like user management, product catalog, and payments. This approach, while more complex, provides significant benefits in terms of modularity and scalability. Considering that, the intention of this work was to present an evolutionary perspective to help the reader understand the main motivations that lead to the distinguishing characteristics of microservices [10].

The modular nature of microservices fosters faster development cycles, enabling teams to iterate and innovate more rapidly. This agility is particularly advantageous in today's fast-paced, competitive market, where the ability to deliver value quickly is paramount. Moreover, microservices facilitate scalability, both in terms of technology and team structure. Teams can focus on developing and maintaining smaller, specialized services, reducing the cognitive load and enabling them to make independent decisions regarding technology choices, deployment strategies, and scaling requirements. Additionally, microservices enhance resilience by isolating failures and minimizing the blast radius of issues. If one service experiences a failure, it doesn't necessarily impact the entire system, allowing other services to continue functioning independently. This fault isolation is crucial for maintaining system stability and ensuring high availability in distributed environments.

However, adopting microservices is not without its challenges. The increased complexity of managing distributed systems requires careful consideration of issues such as service discovery, inter-service communication, data consistency, and deployment orchestration. Furthermore, organizations must invest in robust monitoring, logging, and debugging tools to effectively manage and troubleshoot microservices-based architectures.

Despite these challenges, the benefits of microservices architecture are compelling, driving widespread adoption across industries. As organizations strive to innovate and stay competitive in today's digital landscape, microservices offer a flexible, scalable, and resilient foundation for building modern software systems. With careful planning, thoughtful design, and continuous refinement, microservices architecture can unlock new opportunities for organizations to deliver value to their customers and adapt to evolving business requirements.

REFERENCES

[1] Bozic, V. (2023). Microservices architecture, Research Proposal, DOI:10.13140/RG.2.2.21902.84802.

[2] Fowler, M. (2014). Microservices: a definition of this new architectural term, https://martinfowler.com/articles/microservices.html.

[3] Nadareishvili, I., Ronnie, M., McLarty, M., Amundsen, M. (2016). Microservice architecture: aligning principles, practices and culture, Published by O'Reilly Media, Sebastopol, 978-1-491-95979-4 [LSI].

[4] Pachikkal, C. (2021). Interservice Communication in Microservices, IJARSCT, ISSN 2581-9429. DOI: 10.48175/568

[5] Weerasinghe, S., Perera, I. (2023). Optimized Strategy for Inter-Service Communication in Microservices, International Journal of Advanced Computer Science and Applications, Vol. 14, No. 2.

[6] Nzekwe, E. (2022). Demystifying the Repository Pattern in ASP.Net Core Web API

[7] Erl T. (2009), SOA Design Patterns, Prentice Hall, ISBN 978-0-13-613516-6.

[8] Mudassar Ali Khan, S. (2022). Onion Architecture in Asp.net Core 6 Web API.

[9] Gadge, S., Kotwani, V. (2017). Microservice Architecture: API Gateway Considerations.

[10] Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L. (2017). Microservices: Yesterday, Today, and Tomorrow. DOI 10.1007/978-3-319-67425-4_12