https://orcid.org/0000-0002-9374-2833
https://orcid.org/0000-0001-6559-1508

UDC 004.273

H.T. SAMOYLENKO*, A.V. SELIVANOVA*

FEATURES OF MICROSERVICES ARCHITECTURE IN E-COMMERCE SYSTEMS

*State University of Trade and Economics, Kyiv, Ukraine

Анотація. *Розробка архітектури інформаційних систем для електронної торгівлі є важливим завданням для підприємств у сучасному цифровому середовищі. У статті проводиться порівняння різних типів архітектур інформаційних систем, що використовуються в електронній торгівлі. У статті розглянуто особливості електронної торгівлі та специфіку веб-додатків, що підтримують ведення електронного бізнесу. Охарактеризовано традиційну монолітну архітектуру, мікросервісну архітектуру та сервер-безсерверну (serverless) архітектуру, визначено їх переваги і недоліки. Досліджено технологічні рішення в галузі розробки електронної комерції з метою подальшого створення ефективної та зручної платформи для користувачів. У роботі визначено такі ключові аспекти архітектур, як масштабованість, гнучкість, швидкість розгортання та управління, надійність та вартість розробки й експлуатації систем. Проаналізовано вплив кожного типу архітектури на продуктивність і відповідність бізнес-потребам у галузі електронної торгівлі. Проведений аналіз допоможе створити або модернізувати інформаційну систему для електронної торгівлі, допомагаючи зробити обґрунтований вибір архітектурного підходу. У статті визначено переваги й недоліки мікросервісної архітектури побудови веб-застосунків електронної торгівлі, зокрема, на базі фреймворку Spring Boot. У статті досліджено, як мікросервіси можуть забезпечити модульність та полегшити розробку, розгортання й підтримку веб-застосунків електронної торгівлі. Проаналізовано недоліки мікросервісної архітектури, зокрема, складність управління та взаємодії між сервісами, а також необхідність вирішення проблеми консистентності даних та керування транзакціями в розподілених середовищах. Розглянуто можливі підходи та технічні засоби для подолання цих проблем у контексті веб-застосунків електронної торгівлі на базі Spring Boot.*

Ключові слова: *архітектура інформаційних систем, мікросервісна архітектура, фреймворки.*

Abstract. *The development of information system architectures for e-commerce is a crucial task for businesses in today's digital environment. This article presents a comparative analysis of different types of information system architectures used in e-commerce. The paper examines the specific characteristics of e-commerce and the requirements of web applications that support electronic business operations. Traditional monolithic architecture, microservices architecture, and serverless architecture have been characterized, and their advantages and disadvantages have been identified. Technological solutions in the field of e-commerce development have been studied to create an efficient and user-friendly platform. The article highlights such key architectural aspects as scalability, flexibility, deployment speed and management, reliability, and development and operational costs. The impact of each architecture type on performance and alignment with e-commerce business needs has been analyzed. This analysis will assist in the creation or modernization of information systems for e-commerce, enabling an informed choice of architectural approach. The article specifically examines the advantages and disadvantages of microservices architecture for developing e-commerce web applications, particularly using the Spring Boot framework. It explores how microservices can provide modularity and facilitate the development, deployment, and maintenance of e-commerce web applications. Such challenges of microservices architecture as managing and coordinating services, as well as addressing data consistency and transaction management in distributed environments have been discussed. Some possible approaches and technical tools to overcome these challenges in the context of e-commerce web applications based on Spring Boot have been considered.*

Keywords: *information system architecture, microservices architecture, frameworks.*

DOI: 10.34121/1028-9763-2023-3-51-58

1. Introduction

Electronic commerce (e-commerce) has rapidly emerged as a prominent phenomenon in business and overall economic activity within a relatively short period of time. It can be conducted across various networks, not necessarily limited to the Internet. E-business, on the other hand, refers to business operations that utilize global information systems. In other words, it represents a form of conducting business where a significant portion of activities is performed using information technology. The key components of e-business include internal organization within a company based on a unified information network, as well as external interactions with partners, suppliers, and customers facilitated through the Internet.

Elements of e-business began to appear in the activities of companies starting from the 1960s. These included automated business systems such as Electronic Data Interchange (EDI), Electronic Fund Transfer (EFT), and Enterprise Resource Planning (ERP). Thus, e-business encompasses all forms of electronic business activities within production and organizational relationships among employees within a single company, between different companies, government bodies, scientific and educational institutions, cultural organizations, non-profit entities, and public organizations.

The Internet, in particular, has played a crucial role in driving the further development of e-commerce. It has expanded the scope of e-commerce not only for large corporations and small-to-medium-sized enterprises but also for individual users. As a result, it has opened up opportunities to engage a significantly broader audience of suppliers and consumers. However, in many cases, the definitions and metrics of electronic commerce have been formulated for marketing purposes rather than for conducting quantitative measurement or scientifically grounded analysis of this phenomenon.

The aim of the article is to analyze and justify the application of microservices architecture in electronic commerce systems.

2. Results of the research

Over the past few years, web application architecture has undergone significant changes, and new approaches and technologies have emerged. One of the most important choices that developers have to make is the choice of architecture, as it has a decisive impact on the application's performance, scalability, and ease of maintenance. Currently, the three most popular architectural options are monolithic architecture, microservices architecture, and serverless architecture. Each of these architectures has its own advantages and disadvantages, and the choice of a specific architecture depends on the needs and constraints of the specific project [1, 2].

Monolithic architecture is a traditional approach where the entire application is deployed as a single unit. It is a straightforward approach that is easy to develop and deploy, but large monolithic applications can become complex to understand and maintain over time. The entire application runs within a single process and uses a shared database. The codebase contains all the components such as the user interface, business logic, and database access configuration, which are executed within a single executable file. The main advantages of monolithic architecture lie in the simplicity of development and deployment as all the components are deployed together. It also simplifies debugging and scaling of the application as all parts interact without calling external services. Additionally, developing within a single codebase reduces overhead costs associated with communication and integration between components. However, monolithic architecture may have certain drawbacks. Making changes to one part can affect the rest of the system, which can be problematic. Moreover, scaling individual components may be limited as the entire application functions as a single unit.

Microservices architecture involves breaking down an application into small, independent services that interact with each other. This allows for flexible scaling of individual components and quick changes to specific services. However, it can lead to complexity in managing multiple services and their interactions. Microservices architecture offers numerous advantages, including scalability and flexibility. Since each microservice is a self-contained unit, it can be scaled independently of other microservices, making it easier to handle increased traffic or load. Additionally, microservices can be written in different programming languages and utilize different technologies, making them a flexible choice for developers. Another advantage of microservices architecture is the simplification of deployment and testing processes. Each microservice can be deployed and tested independently, reducing the risk of errors. Furthermore, updates or changes to a microservice only require the compilation and deployment of that specific microservice rather than the entire application.

«Serverless» architecture is based on the use of cloud services to deploy and execute code without the need to manage infrastructure. This allows developers to focus on building functionality without the overhead of server configuration and maintenance. However, this approach can be limited in terms of supporting certain technologies and can be complex when it comes to measuring and controlling costs. Serverless architecture is a methodology for creating and deploying web applications without directly using servers. Instead of traditional server management, developers can create and launch their programs on serverless platforms such as AWS Lambda, Azure Functions, or Google Cloud Functions. These platforms offer pay-per-use pricing for computational resources, allowing for cost optimization. One of the key advantages of serverless architecture is its cost-effectiveness. Additionally, these platforms take responsibility for scaling, managing, and securing the infrastructure, potentially reducing operational costs. Another advantage of serverless solutions is their ability to accelerate the development process. Developers can focus on writing application code without spending time on server management. Scalability is also an important advantage of serverless architecture. The platform automatically scales the program based on request volume, making it easier to handle increased traffic or load. However, there are some drawbacks to serverless solutions. The increased latency can occur due to the need to launch new instances of the program with each user request. Additionally, detecting and resolving errors can be more challenging as the application is distributed across multiple independent services. The choice of web application architecture depends on the specific needs of your project, its size, and the constraints imposed by project requirements.

3. Materials and methods

The simplest way of communication between microservices is through synchronous HTTP requests. In this case, one service simply calls another service, often using REST API. The first service initiates the call to the second service, waits for the second service to process the request, receives a response, and performs further application logic based on that response. In this way, services communicate with each other by passing data and requesting processing results. There is another method of communication between microservices that involves asynchronous communication through specialized software called message brokers. Some of the well-known brokers include Kafka, RabbitMQ, and ActiveMQ. The role of a message broker is to receive a message from a producer and deliver it to a consumer. There are many different implementations of this concept, allowing developers to choose a specific broker and configure it based on the requirements of their tasks. For example, RabbitMQ is an open-source platform written in Erlang. It uses a queue as a data structure where messages are processed following the FIFO (First-In-First-Out) discipline, implementing a non-priority queue. On the other hand, Kafka has a more complex structure known as a topic, which is divided into several partitions where data can be replicated. A Kafka topic resembles a table in a database. Messages in Kafka can be retained even after being received by a consumer for a certain period defined by the developer. The more complex structure of Kafka provides more configuration options, but not all tasks require message replication or retention, giving developers a wide choice of brokers to use in their systems.

An application with a microservices architecture often utilizes multiple services to execute specific business logic, similar to how a monolith can invoke different modules when receiving a request from a user, involving various parts of the program and data stored in different tables. For instance, let’s consider an application representing a hypothetical online computer hardware store. A user submits a request to purchase 10 laptops for a total of $20,000. During the processing of this request, the system needs to verify that these 10 laptops are available in stock and that the user has the necessary funds to complete the payment. If both checks succeed, the system should reserve the laptops and block the funds in the user’s account. This task can be easily accomplished within a single transaction in the database. Relational databases are known to have ACID properties – atomicity, consistency, isolation, and durability. This means that at any given moment, the system will be in a consistent state. Therefore, if the system reserves 10 laptops and the subsequent database query indicates that the user does not have the corresponding amount in their account, the transaction will be rejected, and the system will return to its initial state. Thus, such a user request does not pose problems when there is a monolithic architecture utilizing a relational database.

However, microservices architecture does not inherently provide automatic consistency. Each service is a separate isolated program that has access only to its own database. The principle of single responsibility requires introducing three services in the system: an order service, a warehouse service, and a user service. They need to synchronize their data to achieve «eventual consistency». This means that at a certain point in time, the system may be in an inconsistent state but will eventually reach consistency. For example, if the warehouse service successfully reserves 10 laptops but the user service returns a response different from positive (insufficient funds in the user's account), the reservation will be canceled. This problem can be addressed using the Saga pattern which has two implementations [2].

The first implementation involves having an orchestrator service that manages the execution of transactional steps with the corresponding services. The user's request is initially sent to the orchestrator which ensures that all the services have positive results. If any of the services has an error, the orchestrator sends a command to all other services to undo the changes made during the transaction. Thus, the system may be in an inconsistent state for a period of time, but thanks to the orchestrator's work, the transaction will either be completed or all the changes will be rolled back. Communication between services is typically carried out using HTTP requests. The second implementation utilizes a compensation mechanism where each action that modifies the system's state is accompanied by a corresponding compensating action that reverses the changes if something goes wrong. Both implementations of the Saga pattern allow for avoiding situations where the system remains in an inconsistent state. Either all changes made during the transaction are rolled back, or the transaction is successfully completed, bringing the system to a consistent state. The compensation mechanism or orchestrator helps to achieve this behavior in microservices architecture. The orchestrator is a central service that manages the execution of transactional steps. Each service communicates with the orchestrator before executing a step to obtain its approval. If an error occurs at any step, the orchestrator sends a command to other services to undo the changes that have already been made. This approach ensures a consistent system state but requires more communication with the orchestrator. The compensation mechanism is used to undo changes that have already been made in case of an error. Each transaction step is accompanied by a corresponding compensating step that restores the system to its previous state. When an error occurs, compensating actions are performed, ensuring the consistency of the system. This approach may be simpler to implement but requires additional code for compensating actions.

4. Designing

Developing an e-commerce application using a microservices architecture involves the following steps:

Requirements analysis. Identify the functional requirements of the system. Determine the types of products or services that will be available for purchase. Define the required payment and delivery processes. Identify the types of users who will interact with the system. This will help you determine the core components of the application.

Microservices decomposition. Identify the key functional components of the application and break them down into separate microservices. For example, this could include microservices for product catalogs, purchasing and payment, user management, and more. Each microservice should perform a specific function and have its own database.

Microservice development. Develop each microservice separately using appropriate technologies and frameworks such as Spring Boot or Node.js. Each microservice should have its own API for communication with other services and external clients.

Transaction management. In e-commerce, it is important to have mechanisms for managing transactions to ensure data consistency during purchase and payment operations. To ensure the atomicity of operations, a transaction coordinator such as Spring Cloud Netflix or Apache Kafka can be used.

Authentication and authorization. Implement mechanisms for authentication and authorization to secure access to your application and restrict user privileges. For example, you can use token-based authentication (JWT) and authorization mechanisms such as roles or permissions.

Scalability. Plan for scaling your application, especially during peak loads when the number of users and transactions increases. Consider using techniques like load balancing and horizontal scaling to handle increased traffic.

Monitoring and logging. Configure a monitoring and logging system to track the performance and behavior of microservices, detect errors, and ensure system reliability. Tools such as the ELK Stack (Elasticsearch, Logstash, and Kibana) or Prometheus with Grafana can be used as examples.

Building a microservices architecture using frameworks. Spring Framework (Fig. 1) is one of the most popular tools for developing web applications in Java [3]. It consists of various modules that handle different aspects of an application. One of its key principles is the inversion of control, which simplifies the development process.

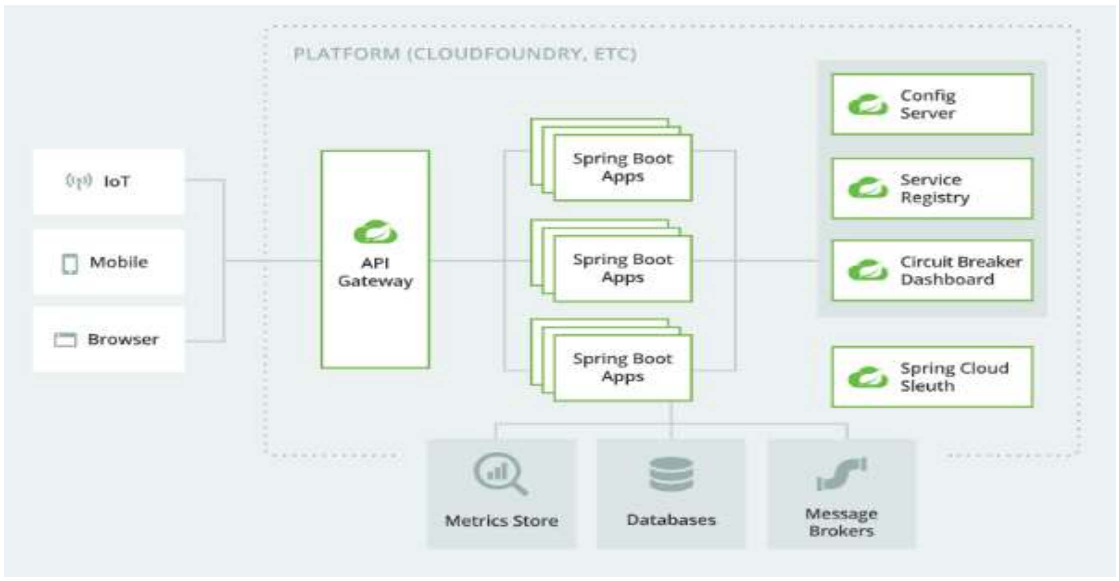

Figure 1 – Spring Framework

In Spring Framework, developers declare beans which are Java classes that implement specific logic, and the bean container manages their lifecycle. Beans are initialized at the start of the program, and the framework automatically creates and satisfies the dependencies defined by the developer. If certain beans cannot be initialized, such as due to circular dependencies, the program will fail to start, and the developer will see an appropriate error message. When using Spring Framework, developers are required to manually configure components required by the modules, often through the use of XML files. For example, to use the Spring MVC module, which is responsible for creating server-side rendering (SSR) web applications, the path to the folder containing HTML pages needs to be specified. While Spring Framework already solves the dependency injection (DI) problem, Spring Boot addresses the issue of autoconfiguration. Spring Boot includes all the modules available in Spring but introduces the concept of «starters» that contain necessary configurations, allowing developers to write code without creating unnecessary XML files. Additionally, an embedded web server is automatically integrated into the application when using a web starter. Thus, Spring Boot provides a convenient toolkit for developing web applications of any type whether it is REST API or MVC. In particular, the Spring Data JPA module allows for executing database queries by translating the Java method name into the corresponding SQL query.

Spring Cloud is a set of tools and libraries that extends the functionality of the Spring Framework for developing microservices architectures. It provides developers with convenient means for deploying, managing, and interacting between microservices [3, 4]. The main components of Spring Cloud include Eureka, Ribbon, Hystrix, Feign, Zuul, Config, and Sleuth.

Eureka is one of the components of Spring Cloud and serves as a service registry that allows microservices to register and discover each other in a distributed system [5].

The key concepts associated with Eureka include:

• Eureka Server. This is a central component that acts as a service registry. Each microservice that wants to be registered connects to the Eureka Server. It stores information about registered microservices, including their identifiers, URLs, and metadata.

• Eureka Client. This component is used by microservices to register and interact with the Eureka Server. Each microservice that wants to be discovered by other microservices needs to be configured as a Eureka Client.

• Registered Instances. These are the microservices that have successfully registered with the Eureka Server. Each instance of a microservice provides information about its availability, status, and other metadata.

• Replication Servers. If there is a need to enhance the availability and reliability of the Eureka Server, replication can be configured. It allows for having multiple Eureka Servers that synchronize their state and information about registered microservices.

The advantages of using Eureka:

• Service Discovery. Eureka enables microservices to discover each other using the service name, simplifying instance communication.

• Automatic Registration. Eureka Clients can automatically register their instance with the Eureka Server, simplifying configuration and scalability support.

• Fault Detection. The Eureka Server can detect the absence of registered microservices and inform other services about changes in service status.

• Load Balancing. With Eureka, load balancing can be utilized to distribute traffic among different instances of a service.

Ribbon is a load-balancing library that allows for traffic distribution among instances of a microservice to enhance availability and scalability. It is one of the libraries developed by Netflix and used for load balancing across microservice instances. This helps to ensure high availability, backup, and system scalability. With Ribbon, various load balancing algorithms such as Round Robin, Random, Weighted distribution, and others can be configured. It integrates with other libraries like Spring Cloud and Netflix Eureka, making it easier to work with microservices in a Java-based environment. Ribbon provides capabilities for dynamic instance discovery, health monitoring, and automatic retry. This enables the system to operate efficiently even when the number and state of microservice instances change. By intelligently distributing requests among available instances, Ribbon helps to optimize resource utilization and provides fault tolerance in distributed systems.

Hystrix is a library for handling failures and ensuring system resilience. It allows developers to add error handlers, rate limiting, and fallback mechanisms to ensure the reliability and stability of microservices. Hystrix provides several mechanisms to achieve this goal. One of them is the circuit breaker mechanism which allows requests to be intercepted to a dependent service if it becomes unavailable or returns a large number of errors. This helps to prevent system overload and failures. Additionally, Hystrix allows the addition of error handlers (fallbacks) that are executed in case of an error in the dependent service. This enables alternative logic or returning default data to avoid system failures. Hystrix also provides monitoring capabilities to track metrics and performance statistics of requests such as response time, number of successful and unsuccessful requests, and circuit status. This helps developers identify issues and analyze system performance.

Feign is a declarative HTTP client that simplifies interaction with other microservices [6]. It allows developers to describe the interaction with other services using annotations and automatically generates the necessary code. Developers can define interfaces with annotations specifying the URL, HTTP methods, and request parameters. Feign automatically generates the required code for interacting with the specified services. With Feign, developers can avoid writing repetitive code related to HTTP requests and response handling. Feign handles object serialization and deserialization, sets necessary request headers, and provides error handling capabilities. It integrates with other libraries like Ribbon and Eureka for automatic load balancing and service discovery.

Zuul is a router and load balancer that allows for managing incoming traffic to microservices [7]. It enables the configuration of routing rules, filters, and security policies. Zuul performs routing functions, meaning it receives incoming traffic and forwards it to the appropriate microservice. It can have routing rules that specify which request should be directed to which service. This allows developers to flexibly distribute traffic among different microservices based on URL patterns or other parameters. Zuul can also act as a load balancer, distributing traffic among available instances of a single microservice. This helps to ensure high availability and scalability of the system. Additionally, Zuul has filter functionality that allows developers to apply various operations to incoming traffic such as authentication checking, authorization, logging, data transformation, etc. This enables the implementation of security policies and additional traffic processing before it is passed on to microservices.

Config is a module for centralized management of microservices configuration. Config allows for storing the configuration of microservices in a remote repository such as Git and automatically loading it during the startup of the microservice. This allows for keeping the configuration separate from the microservices themselves, providing centralized management and a convenient ability to change configuration without restarting the microservices. Config supports various configuration file formats such as properties, YAML, or JSON and can automatically track changes in the configuration repository and provide updated values to microservices without their restart. This enables dynamically configuring microservice parameters without the need for code modification or restarts. Config also supports features like configuration versioning, access control to configuration files, encryption of sensitive data, and more, to ensure security and configuration management.

Sleuth is a library developed by the Spring company for tracing and analyzing logs in a distributed service architecture such as microservices. It provides the ability to generate and deploy unique request identifiers (trace IDs) that allow for tracking the path of a request as it passes through different microservices in the system.

The main functions of Sleuth include:

• Unique request identifiers. Sleuth generates a unique identifier (trace ID) for each request entering the system. This identifier is assigned to every log associated with that request, regardless of which microservices it has traversed. This enables tracing the path of a request and analyzing its journey through different components of the system.

• Span context. Sleuth adds a span context to each log, allowing for understanding the relationship between logs related to a single request. Span context contains information about the start and end of individual stages of request processing, which may occur in different microservices.

• Integration with other tools. Sleuth integrates with other analysis and monitoring tools such as Zipkin or ELK (Elasticsearch, Logstash, and Kibana) for centralized log collection, tracing, and analysis of distributed services.

With Sleuth, developers can trace request paths, analyze the execution time and throughput of individual system components, detect issues and bottlenecks in the distributed architecture, and improve monitoring and debugging processes in the system.

5. Conclusions

The article analyzes tools that assist developers in building stable, scalable, and easily manageable microservices systems, providing functions such as registration, load balancing, fault tolerance, interaction, and centralized configuration management. Spring Cloud integrates well with other Spring components and other technologies for developing distributed systems. The application of microservices technologies can ensure modularity and facilitate the development, deployment, and subsequent technical support of e-commerce systems.

REFERENCES

1. Pattern: Monolithic Architecture. URL: https://microservices.io/patterns/monolithic.html.
2. Pattern: Saga. URL: https://microservices.io/patterns/data/saga.html.
3. Spring Framework. URL: https://spring.io/projects/spring-framework.
4. Spring Cloud. URL: https://spring.io/projects/spring-cloud.
5. Service Registration and Discovery. URL: https://spring.io/guides/gs/service-registration-and-discovery/.
6. Spring Cloud OpenFeign. URL: https://spring.io/projects/spring-cloud-openfeign.
7. Spring REST with a Zuul Proxy. URL: https://www.baeldung.com/spring-rest-with-zuul-proxy.

Стаття надійшла до редакції 30.06.2023