ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/azrajabeen-mohamed-ali/379

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science
https://urfpublishers.com/journal/artificial-intelligence

Vol: 2 & Iss: 3

Research Article

Overview of Microservices Design Patterns' Problems, Solutions

AzraJabeen Mohamed Ali*

Citation: Ali AJM. Overview of Microservices Design Patterns' Problems, Solutions. *J Artif Intell Mach Learn & Data Sci 2024*, 2(3), 1745-1748. DOI: doi.org/10.51219/JAIMLD/azrajabeen-mohamed-ali/379

Received: 02 August, 2024; **Accepted:** 18 August, 2024; **Published:** 20 August, 2024

***Corresponding author:** AzraJabeen Mohamed Ali, Independent researcher, California, USA, E-mail: Azra.jbn@gmail.com

ABSTRACT

This paper discusses the thorough examination of the challenges and solutions associated with the Microservices design patterns. Microservices have transformed the software development sector by encouraging modularity, scalability and maintainability, which enables businesses to react to shifting consumer needs and technology breakthroughs faster. The study's main research question explores the major implementation challenges, including inter-service communication, data consistency and security and provides possible solutions to address these challenges. It also provides a thorough analysis of several microservices patterns, including Decomposition design patterns, Integration patterns and Database patterns. This paper is therefore meant to be more development-environment centered and infrastructure agnostic. Developers and architects who wish to concentrate on code, patterns and implementation specifics will find this part most interesting.

Keywords: Micro Services, Design patterns, API Gateway, Decomposition pattern, Integration pattern, Database patten, monolithic

1. Introduction

1.1. Microservice Architecture:

Microservices architecture, as the name suggests, is a method of developing a server application as a collection of discrete services. Thus, while the front end can also use a microservices design, the back end is the primary focus of this strategy. Every service operates in a separate process and engages in communication with other processes via protocols including AMQP, WebSocket's and HTTP/HTTPS.

To succeed in today's unstable, uncertain, complex and ambiguous world, business-critical enterprise applications must offer changes quickly, often and consistently. Consequently organizations are split up into small, loosely connected, cross-functional teams. Every team delivers software using DevOps techniques. It uses continuous deployment specifically. Before being deployed into production, the team's stream of frequent, minor changes is tested by an automated deployment pipeline. By enabling teams to deploy each microservice as needed, the goal is to enable developers to use microservices to accelerate application releases.

2. What Makes Businesses Use Microservices Architecture?

The majority of businesses begin by building their infrastructures as either a single monolith or a number of closely related monolithic applications. The monolith performs a variety of tasks. A single, coherent piece of application code contains all of the programming for those features. It's challenging to decipher the code for these functions because it's all tangled together. A single feature addition or modification in a monolith can cause the entire application's code to break. This turns any upgrade-no matter how basic-into a costly and time-consuming procedure. Programming gets increasingly complex as upgrades are made, until scaling and upgrades are nearly impossible.

As time goes on, businesses are unable to modify their coding further without beginning anew. The procedure quickly becomes too much to handle and businesses may end up stuck with outdated practices for a long time after they ought to have upgraded.

3. What are the fundamental ideas of Microservices architecture?

Autonomous: By operating independently, each service removes the issues brought on by interdependence. Increased deployment flexibility is also made possible by this.

3.1. Resilience: The other services remain unaffected in the event that one goes down. The capacity of a system to bounce back quickly from malfunctions and keep working.

3.2. Scalability: The ability of services to immediately increase or decrease in response to demand maximizes cost and resource allocation.

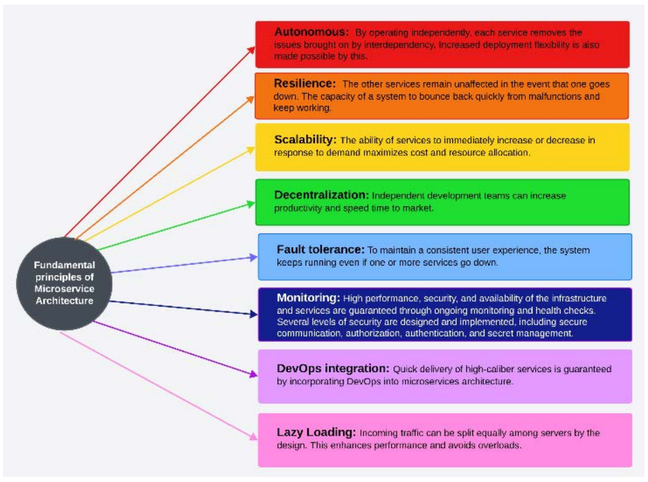
3.3. Decentralization: Independent development teams can increase productivity and speed time to market.

Fault tolerance: To maintain a consistent user experience, the system keeps running even if one or more services go down.

3.4. Monitoring: High performance, security and availability of the infrastructure and services are guaranteed through ongoing monitoring and health checks. Several levels of security are designed and implemented, including secure communication, authorization, authentication and secret management.

3.5. DevOps integration: Quick delivery of high-caliber services is guaranteed by incorporating DevOps into microservices architecture.

3.6. Lazy Loading: Incoming traffic can be split equally among servers by the design. This enhances performance and avoids overloads.


Fundamental principles of Microservice Architecture
Autonomous: By operating independently, each service removes the issues brought on by interdependency. Increased deployment flexibility is also made possible by this.
Resilience: The other services remain unaffected in the event that one goes down. The capacity of a system to bounce back quickly from malfunctions and keep working.
Scalability: The ability of services to immediately increase or decrease in response to demand maximizes cost and resource allocation.
Decentralization: Independent development teams can increase productivity and speed time to market.
Fault tolerance: To maintain a consistent user experience, the system keeps running even if one or more services go down.
Monitoring: High performance, security, and availability of the infrastructure and services are guaranteed through ongoing monitoring and health checks. Several levels of security are designed and implemented, including secure communication, authorization, authentication, and secret management.
DevOps integration: Quick delivery of high-caliber services is guaranteed by incorporating DevOps into microservices architecture.
Lazy Loading: Incoming traffic can be split equally among servers by the design. This enhances performance and avoids overloads.

The optimal pattern (or patterns) to apply will depend on business goals and other relevant criteria. There are many different design patterns for microservices, each with unique benefits and cons. The design patterns are categorized based on their purpose.

3.7. Decompose Patterns: Purpose: To assist in breaking down applications into smaller, easier-to-manage services. This pattern includes "Decompose by Business capability", "Decompose by Subdomain", "Decompose by Transactions", "Strangler pattern", "Bulkhead pattern", "Sidecar pattern".

3.8. Observability Patterns: Purpose: To continuously observe in real time in order to spot mistakes and improve performance. This pattern includes "Log Aggregation", "Performance Metrics", "Distributed Tracing", "Health Checks".

3.9. Integration Patterns: Purpose: To integrate the systems and applications to exchange information through API and their protocols. This pattern includes "API Gateway pattern", Aggregator pattern", "Proxy pattern", Gateway Routing pattern", "Chained Microservice pattern", "Branch pattern", "Client-Side Ui Composition pattern".

3.10. Database Patterns:

Purpose: To manage data effectively from one or many different services. This pattern includes "Database Per services", "Shared Database per Service", "CQRS", "Event Sourcing", "Saga pattern".

3.11. Cross-Cutting Concern Patterns:

Purpose: To uniformly handle specific capabilities throughout the system, such as logging, tracing and monitoring across several microservices. This pattern includes "External Configuration", "Service Discovery Pattern", "Circuit Breaker Pattern", "Blue-Green deployment Pattern".

3.12. Problems and fixes when choosing a design pattern:

Problem #1: An application must be divided into smaller components in order to transition to micro services. Which design pattern facilitates the division of an application into smaller services?

Fix: One method is to decompose by business capability. Using the business capability pattern to break down an application into smaller services can be advantageous. Because business capabilities are often stable, this pattern leads to stability. However, the capability to recognize each unique company's skill is essential to its success.

Sample: For instance, HealthCare company's business capabilities are mainly classified as Healthcare Clinical delivery, Patient Driven Management, HealthCare Research, Strategy, Finance & Accounting, Human Resources, Marketing & Sales, Enterprise Support and Governance & Compliance.

The phrase "code smell" refers to imperfections in design and coding techniques. "God class" is a type of code smell occurs when functionalities are distributed unevenly throughout large classes. A number of services will share these classes. To break down further small pieces, "Decompose by Subdomains" will be very helpful. By dividing the primary domain into smaller domains, Domain Driven Design helps achieve this "decompose by subdomains."

Sample: For instance, above in Health Care company's one of the business capabilities is "Patient Driven" which can be broken into further as "Patient Management", "Payer Management", "Partner Management", "Service Level Management"," Medical Billing Management", "Patient Lead Management". Each micro service would be bound independently with its own logic.

Problem #2: The current live legacy application follows the traditional monolithic design and it is expected to be divided into smaller components in order to transition to micro services. Which design pattern facilitates the division of an application into smaller services?

Fix: The best approach is Strangler Pattern alias Vine pattern as the name mentions, vine strangles around the tree, micro service wrapped around the existing application. Strangler Pattern allows to create a new service and functions along with the existing monolith application service next to each other. A microservice application expands whereas a monolith application shrinks with time. With the help of this design, it is possible to move features and data from the monolith to the new microservices without affecting user experience.

Problem #3: What kind of design pattern will work best for a company that deals with a large volume of customers at once?

Fix: To deal large volume of customers at once, Bulkhead pattern works best. Because it resembles the sectioned partitions of a ship's hull, the bulkhead design got its name. It functions by dividing an application's components into distinct sections. The others can still work even if one fails. A significant level of traffic from one service may use up all database connections, making it impossible for other services to access the database. Limiting the number of connections accessible to each service through the implementation of a bulkhead pattern ensures that no single service may create a bottleneck. This entails segmenting the system into distinct resource pools, such as database connections or network connections, in microservice architecture. By doing this, we can lessen the effect of any one failure, which makes it simpler to isolate and recover from disasters.

Problem #4: How can we manage failures graciously and prevent cascade service failures?

Fix: Additionally, for improved fault isolation, it can be used with the circuit breaker pattern. Circuit breaker pattern exactly works same as circuit breaker switch. A circuit breaker switch works by interrupting the electrical current flow when it exceeds a predetermined limit causing the contacts to separate and break the circuit, thus protecting the system from damage due to overload or short circuit. Likewise, the link to failed services is disconnected by this pattern. In order to keep protected calls from being made and left "hanging," any calls to the breaker are either redirected to another service or result in an error default message. For a predetermined "timeout" period, this occurs. This prevents service failures from cascading and allows for gentle handling of failures.

Three states are handled in circuit pattern like Open, closed, Half open. When the number of failures surpasses the threshold, a circuit breaker pattern is in open state. In this state, the microservice does not execute the intended function but instead returns errors for the calls. A circuit breaker is in its default condition when it is in closed state and all calls are handled normally. A very ideal state to be expected by the users. A circuit breaker stays in a half-open position as it looks for underlying issues. While some calls might receive a regular response, others might not. The reason for the circuit breaker's initial flip to this state will determine this.

Problem #5: What kind of design pattern should be used to call several microservices and manage numerous protocols?

Fix: In order to solve the aforementioned problem, the API gateway pattern acts as a reverse proxy between client apps and the services, offering a single-entry point for many microservice calls. The pattern also shields the client from having to understand how services have been divided, which is another benefit. This design pattern allows developers to separate client apps from internal microservices, allowing for the use of a partially unsuccessful request. This guarantees that a single unresponsive microservice won't cause an entire request to fail. The encoded API gateway does this by using the cache to either deliver a correct error code or an empty response. Furthermore, API gateway patterns can handle important functions like SSL termination, authentication and caching, which improves the security and usability of application. It may also convert one protocol request to another protocol and vice versa.

Problem #6: Which design pattern aids in formatting the data and responses from reusable microservices that can change based on the client?

Fix: Aggregator pattern helps in the process of combining data from several providers and sending the final response to the customer. Before returning the data, a composite microservice will call all necessary microservices, combine the data and change it. Additionally, an API gateway can aggregate the data and divide the request among several microservices before delivering it to the customer. Selecting a composite microservice is advised if any business logic is to be used. Otherwise, the tried-and-true method is the API gateway.

Problem #7: Which design pattern can be used to create a user interface (UI) page or screen that shows data from several services?

Fix: Client-Side UI composition type helps to resolve above mentioned issue. Every UI team has the ability to create a client-side UI element, that implements or correlates to a certain microservice. The UI team is in charge of creating page skeletons or skeleton user interfaces for different services by constructing pages that are made up of various service-specific UI elements. Many JS frameworks help to do the same. It becomes simpler and easier to maintain UI development.

Problem #8: When an organization handles different clients like Web application, mobile application and third-party applications. How are the individual services accessed by the different clients of a microservices-based application?

Fix: Design patterns like API gateway / Backend for frontends can be considered. When we choose API gateway, it will act as a single-entry point for all clients. The Backends for Frontends pattern is a variant of this design. For every type of client, a distinct API gateway is defined.

4. Conclusion

The key takeaway is that no specific technology, architecture pattern or style is appropriate in every circumstance. The majority of major websites are switching from monolithic to microservice architectures. It has a distinct set of problems that need to be solved, just like any other software. A successful microservice workflow can be achieved by putting in place the right architecture, process tools and design patterns.

5. References

1. https://microservices.io/patterns/decomposition/decompose-by-business-capability.html?source=post_page-----649cfec42dc5--------------------------------
2. https://microservices.io/patterns/decomposition/decompose-by-subdomain.html
3. https://microservices.io/patterns/refactoring/strangler-application.html

4. https://www.openlegacy.com/blog/microservices-architecture-patterns/
5. https://medium.com/geekculture/design-patterns-for-microservices-circuit-breaker-pattern-276249ffab33
6. https://medium.com/@parserdigital/resilience-in-microservices-bulkhead-vs-circuit-breaker-54364c1f9d53
7. https://microservices.io/patterns/microservices.html
8. https://dzone.com/articles/design-patterns-for-microservices