Microservices Design Pattern

Amit Sengupta (Independent Research)
Email – amits2913@gmail.com
Independent Researcher

Abstract: -

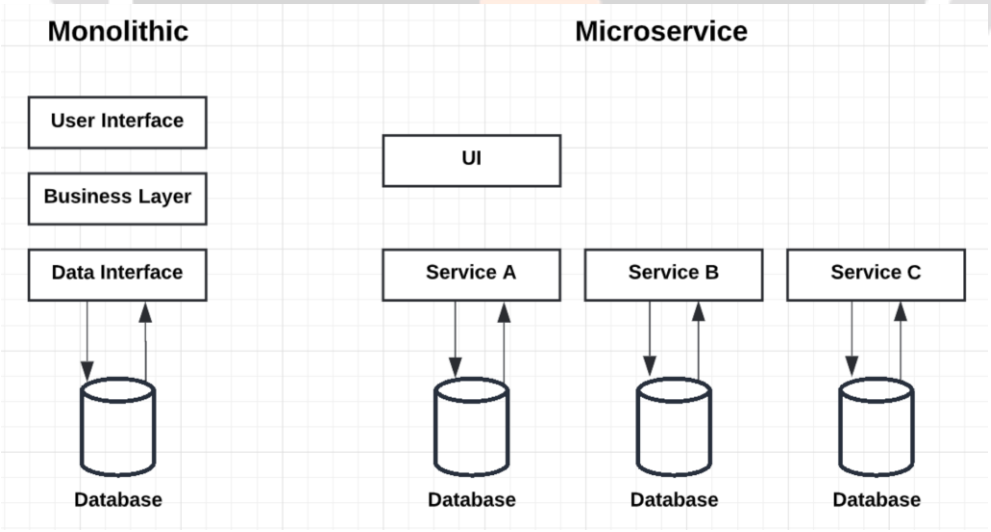
Microservices, a modern architectural paradigm, breaks down complex applications into smaller, independently deployable services, promoting agility, scalability, and resilience. However, deploying them successfully requires a good understanding of the challenges in distributed systems. Design patterns for microservices provide concise and effective solutions by leveraging established principles of software design.

This series will consist of 5 parts, covering a total of ten design patterns crucial for constructing resilient and scalable microservices architectures. In the first part, we'll explore the concept of microservices, the significance of design patterns, as well as discuss the Service Registry Pattern and the Service Mesh Pattern.

Keywords: - Microservices Pattern, Resiliency Patterns, API Gateway Pattern, Service Registry Pattern, Circuit Breaker Pattern, Service Mesh Pattern, Observer Pattern, Aggregator Pattern.

Introduction: -

Microservices are a software architectural style that structures an application as a collection of loosely coupled services, each focused on performing a specific business function. These services are independently deployable, scalable, and can be developed using different programming languages and technologies.



Microservices communicate with each other via APIs and often utilize lightweight protocols such as HTTP or messaging queues. This approach enables greater flexibility, agility, and resilience in building and maintaining complex software systems, as each service can be developed, deployed, and scaled independently, allowing for easier maintenance, faster iteration, and better fault isolation.

Need for Resiliency Design Patterns: -

Design patterns are crucial in software development for below notable reasons:

Reusability and Scalability: By encapsulating solutions to recurring design problems, design patterns enable developers to reuse proven solutions and organize code for scalable architectures. For instance, patterns like Microservices facilitate horizontal scaling by breaking down applications into smaller, independently deployable services.

Maintainability and Flexibility: Well-defined design patterns establish common conventions and structures, enhancing the maintainability of software systems and promoting modular, adaptable designs. This streamlines collaboration among developers and ensures changes can be implemented without unintended side effects, facilitated by patterns like Observer or Strategy.

Performance Optimization and Common Vocabulary: Certain design patterns are specifically geared towards optimizing performance, minimizing memory usage, and enhancing security. Additionally, design patterns establish a shared language and best practices within the development community, facilitating effective communication and understanding of complex systems.

Microservices Design Patterns: -

Below are listed some of the most effective microservices design patterns with corresponding design components: -

1. ***Service Registry Pattern:*** The Service Registry Pattern is a design pattern used for service discovery and communication. In this pattern, microservices register themselves with a service registry, which acts as a central repository for service metadata. This allows for dynamic updates, health checking, and seamless communication between services within the distributed system. The main components of the Service Registry Pattern are:

   Registration: Microservices provide information like name, network location, and relevant metadata to the service registry upon startup. This keeps an updated inventory of available services.

   Discovery: Services query the registry for needed information about other services by specifying their name or type. The registry responds with the required network location or endpoints, enabling communication.

   Health Checking: Registries monitor service health and availability, updating records as needed to avoid routing requests to unhealthy services.

   Dynamic Updates: Services continuously update their registry information as they start, stop, or change locations, ensuring accurate service discovery in dynamic environments.

2. ***The Service Mesh Pattern*****:** Service Mesh is a microservices architectural design pattern used to manage communication between microservices in a distributed system. It involves deploying a dedicated infrastructure layer, known as the service mesh, which intercepts and manages both the east west and north south communication between services. This enables features such as traffic routing, load balancing, service discovery, and security enforcement to be implemented consistently across the entire microservices environment. The main components of the Service Mesh Pattern are:

   Sidecar Proxies: Each microservice is coupled with a sidecar proxy, intercepting all inbound and outbound traffic. These proxies are deployed alongside microservice instances, forming the data plane of the service mesh.

Control Plane: The central management component of the service mesh, it comprises various services handling configuration, traffic routing, load balancing, service discovery, and policy enforcement. These services interact with sidecar proxies to enforce desired behaviors across the mesh.

Traffic Management: Service mesh facilitates advanced traffic management functionalities like routing rules, load balancing algorithms, traffic shifting, and fault injection. Operators can control traffic distribution and implement deployment strategies such as canary releases and A/B testing.

Observability: Built-in observability features allow operators to effectively monitor and troubleshoot the microservices environment. This includes metrics collection, distributed tracing, and logging capabilities, providing insights into service performance, latency, error rates, and dependencies.

Security: Service mesh prioritizes security with features like mutual TLS (mTLS) encryption, authentication, authorization, and access control policies. These mechanisms ensure secure communication between microservices and protect sensitive data from unauthorized access.

3. ***Circuit Breaker Pattern:*** Circuit Breaker is a design pattern used in microservices architecture where different services interacting with each other over a network and the circuit breaker protects them from cascading failures to enhance the resiliency and fault tolerance of a distributed system. Typically, once the count of consecutive failures surpasses a specific threshold, the circuit breaker is triggered, halting any further connections to the remote server for a predetermined period. During this interval, the remote service has the opportunity to recover or restart itself. Following this timeout, the circuit breaker conducts tests to determine if requests can successfully pass through. If the test succeeds, it resumes directing requests to the remote service. However, if the test fails, the circuit breaker waits for the specified duration once more before reattempting. *The circuit breaker has three states:*

   Closed: In this state, the circuit breaker allows normal service communication as long as there are no failures.

   Open: When the number of failures reaches a threshold, the circuit breaker switches to the open state, preventing requests from reaching the service and providing a fallback response.

   Half-Open: Once the timeout or reset interval passes, the circuit breaker goes to the "Half-Open" state. It allows a limited number of test requests to pass through to the service to see if the service has recovered or not. If the test requests succeed, it means the service has recovered and the circuit breaker goes back to "Closed" state. If any of the test requests fails, it means the service has still issues and the circuit breaker goes to "Open" state to block further requests.

   Below are the primary stages in the circuit breaker pattern:

   Monitoring: The Circuit Breaker consistently monitors the health and performance of the services it safeguards.

   ***Setting Thresholds*:** It establishes thresholds for various metrics like response times, error rates, or resource utilization.

   Tripping: If the monitored metrics surpass predefined thresholds, the Circuit Breaker "trips," effectively halting further requests from reaching the failing service.

   Fallback: Instead of instantly rejecting requests, the Circuit Breaker might offer a fallback response or utilize cached data to sustain some level of service availability.

   Retry Mechanism: Following a specified time period or when the failed service recuperates, the Circuit Breaker may endeavor to redirect traffic back to the service.

4. **Event Sourcing Pattern:** The Event Sourcing is an architectural approach for synchronously receiving and subsequently asynchronously distributing events within an architecture. Events, once received, are persisted

in a data store unique to the receiving service. Each event represents a state change or action, providing a comprehensive history of system behavior over time. By storing events rather than current state, developers can reconstruct the system's state at any point in time, enabling scalability, auditability, and resilience in distributed environments. Below are the main Components of Event Sourcing Pattern:

Event: Represents a discrete system occurrence or state change, containing essential details such as event type and associated data.

Event Store: Central storage for all system-generated events, maintaining an immutable log of activities in sequential order.

Event Handler: Processes events and updates system state accordingly, performing tasks like database updates or triggering further actions.

Aggregate: Domain-specific grouping of related events, encapsulating business logic and ensuring consistent event application to maintain entity state.

Event Stream**:** Sequence of events for a particular aggregate instance, representing its history.

Event Publishers: Components responsible for disseminating events to interested parts of the system, such as other bounded contexts or external systems.

Event Subscribers: Components that subscribe to events they are interested in, including other aggregates or event processors that take action upon event occurrence.

Command: Client-initiated requests to execute actions within the system, resulting in event generation reflecting the command outcome.

5. **API Gateway Pattern:** The API gateway pattern is a common architectural pattern in which an API gateway sits between the client and a collection of microservices. The API gateway acts as a single-entry point for clients to access the microservices, allowing clients to interact with the microservices as if they were a single service. Acting as a reverse proxy, it routes incoming requests to the appropriate services, handling tasks such as authentication, authorization, rate limiting, and logging. By centralizing these concerns, the API Gateway simplifies client interaction and enhances security. Additionally, it enables efficient monitoring and management of service communication, contributing to improved scalability and resilience in distributed architectures. The main characteristics of API Gateway architecture include:

Routing and Load Balancing**:** API gateways route incoming requests to appropriate microservices based on predefined rules, ensuring reliability and scalability through load balancing across service instances.

Protocol Translation**:** They facilitate the translation of diverse protocols and data formats. For instance, they can convert HTTP requests into formats compatible with backend services, such as gRPC.

Request Transformation: API Gateways modify outbound requests and inbound responses according to backend service specifications, including parameter changes, body transformations, and header manipulation.

Caching: Implementing caching mechanisms within API Gateways reduces request and response latency, enhancing overall performance by serving stored data directly to clients.

6. **Bulkhead Pattern:** The Bulkhead pattern draws inspiration from ship design, where compartments (bulkheads) are used to prevent a breach in one area from sinking the entire vessel. This is a design pattern wherein; you compartmentalize your application's resources so that failure of any one of your dependencies is

not cascaded into the entire system. The Bulkhead pattern is a type of application design that is tolerant of failure. In a bulkhead architecture, elements of an application are isolated into pools so that if one fails, the others will continue to function. It's named after the sectioned partitions (bulkheads) of a ship's hull. This pattern is employed to enhance the resilience and performance of distributed systems, particularly in scenarios where resources need to be partitioned to isolate failures and prevent cascading issues.
Consider an example in a banking application, there are various critical services like transaction processing, user authentication, and customer support. Each of these services requires different resources and can experience failures or performance issues independently.

7. **Database per Service Pattern:** The Database per Service pattern is a design approach where each microservice in a system has its dedicated database. In this pattern, each service is responsible for managing its own data storage, which is decoupled from the databases of other services. This pattern contrasts with the more traditional approach of sharing a single, centralized database among multiple services.
Consider an example of a web application for an online bookstore. In this scenario, we can apply a "database per service" pattern, where each microservice has its own dedicated database tailored to its specific needs.

Conclusion: -

The journey from monolithic application architectures to the intricate world of microservices has been challenging, enlightening, and full of innovations. Microservices Design Patterns stand as a testament to the IT industry's relentless pursuit of solutions that are robust, scalable, resilient and efficient. As they continue to evolve and adapt to new architectures, integrating with emerging technologies, and offering ever more advanced features, one thing remains clear, the microservices design patterns are here to stay, and their role in shaping the future of cloud-native computing is undeniable, understanding and harnessing their potential will be key to building the digital landscapes of tomorrow.

Reference: -

1. O. Zimmermann, "Microservices tenets", *Comput. Sci.-Res. Develop.*, vol. 32, no. 3/4, pp. 301-310, 2017.
2. I. MuleSoft, "Whitepaper: The top six microservices patterns", Oct. 2018, [online] Available: https://www.mulesoft.com/lp/whitepaper/api/top-microservices-patterns.
3. A. Gupta, "Microservice design patterns", Oct. 2015, [online] Available: http://blog.arungupta.me/microservice-design-patterns/.
4. I. Nadareishvili, R. Mitra, M. McLarty and M. Amundsen, Microservice Architecture: Aligning Principles Practices and Culture, Newton, MA, USA:O'Reilly Media, 2016.
5. S. Newman, Building Microservices: Designing Fine-Grained Systems, Newton, MA, USA:O'Reilly Media, 2015.
6. C. Richardson, Microservice patterns, Shelter Island:Manning Publications, 2018.
7. H. E. Schaffer, S. F. Averitt, M. I. Hoit, A. Peeler, E. D. Sills and M. A. Vouk, "NCSU's virtual computing lab: A cloud computing solution", *Computer*, vol. 42, no. 7, pp. 94-97, Jul. 2009.
8. V. F. Pacheco, Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable Maintainable and Testable Microservices, Birmingham, U.K.:Packt Publishing, 2018.
9. H. Khazaei, C. Barna, N. Beigi-Mohammadi and M. Litoiu, "Efficiency analysis of provisioning microservices", *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*, pp. 261-268, 2016.