**Spring Java Vishal Rathod**

SpringBoot

Java 17

React / Angular?

Microservices Design Approach

DevOps practices

CI CD CT

PostgreSQL

Restful API Security

1. Microservice Design pattern:
   1. Circuit Breaker

The Circuit Breaker is a design pattern commonly used in software development to improve system resilience and fault tolerance. Circuit breaker pattern can prevent cascading failures particularly in distributed systems In distributed systems, the Circuit Breaker pattern can be used to monitor service health and can detect failures dynamically. Unlike timeout-based methods, which can lead to delayed error responses or the premature failure of healthy requests, the Circuit Breaker pattern can proactively identify unresponsive services and can prevent repeated attempts. This approach can enhance the user experience.

**States:**

Closed State: When everything is normal, the circuit breakers remain closed, and all the requests pass through to the services. If the number of failures increases beyond the threshold, the circuit breaker trips and goes into an open state.

Open State: In this state circuit breaker returns an error immediately without even invoking the services. The Circuit breakers move into the half-open state after a timeout period elapses. Usually, it will have a monitoring system where the timeout will be specified.

Half Open State: In this state, the circuit breaker allows a limited number of requests from the service to pass through and invoke the operation. If the requests are successful, then the circuit breaker will go to the closed state. However, if the requests continue to fail, then it goes back to open state.

A diagram of a circuit breaker

Description automatically generated

* 1. Bounded Context

Focuses on modeling software to match a [domain](https://en.wikipedia.org/wiki/Domain_(software_engineering)) according to input from that domain's experts. DDD is against the idea of having a single unified model; instead, it divides a large system into bounded contexts, each of which have their own model.

Under domain-driven design, the structure and language of software code (class names, class methods, class variables) should match the business domain. For example: if software processes loan applications, it might have classes like "loan application", "customers", and methods such as "accept offer" and "withdraw".

Domain-driven design is predicated on the following goals:

* placing the project's primary focus on the core domain and domain logic layer;
* basing complex designs on a model of the domain;
* initiating a creative collaboration between technical and domain experts to iteratively refine a conceptual model that addresses particular domain problems.

Critics of domain-driven design argue that developers must typically implement a great deal of isolation and encapsulation to maintain the model as a pure and helpful construct. While domain-driven design provides benefits such as maintainability, Microsoft recommends it only for complex domains where the model provides clear benefits in formulating a common understanding of the domain.

Domain and Integration events.

Key Principles of DDD:

* **Ubiquitous Language:**

A shared language between developers and domain experts, used in both code and business conversations to ensure everyone understands the same concepts.

* **Bounded Contexts:**

Dividing the system into different areas of the domain, each with its own model and vocabulary, to manage complexity and avoid conflicts.

* **Domain Model:**

A conceptual representation of the domain, capturing the essential concepts, rules, and behaviors.

* **Strategic and Tactical Patterns:**

DDD provides both strategic (high-level organization) and tactical (detailed design) patterns to structure the application.

* **Entities and Value Objects:**

Entities have unique IDs and are mutable, while value objects are immutable and represent data aggregates.

* **Aggregates:**

Sets of related entities that are treated as a single unit to ensure consistency.

* 1. Saga pattern

The Saga pattern is a design pattern used in distributed systems, particularly in microservices architectures, to manage transactions that span multiple services. It achieves this by breaking down a large transaction into a sequence of smaller, local transactions, each with its own compensating action to undo changes if a step fails. This pattern helps maintain data consistency in distributed systems and provides flexibility in handling complex, long-lived transactions.

Key Concepts:

* **Local Transactions:**

Each step in the saga is a local transaction, managed by a specific service.

* **Compensating Actions:**

If a local transaction fails, the saga executes a compensating transaction to undo the effects of previous successful steps.

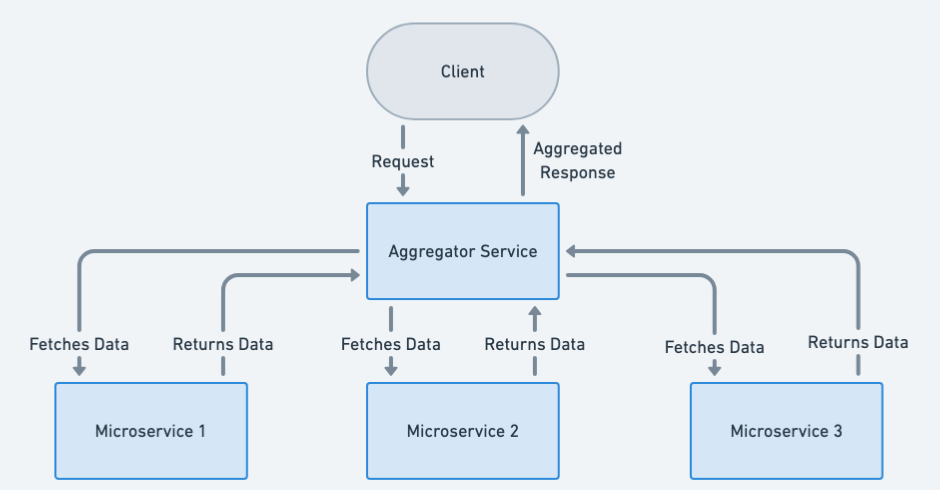
* **Eventual Consistency:**

Unlike traditional transactional systems, sagas provide eventual consistency, meaning that the system will eventually be consistent, but there might be temporary inconsistencies during the process.

* **Two Main Approaches:**
  + **Choreography:** Each service publishes an event after completing its local transaction, which triggers the next step in the saga in another service.
  + **Orchestration:** A central orchestrator manages the entire saga, sending messages to services to execute steps and handling compensating actions.
* **Benefits:**
  + **Loose Coupling:** Choreography-based sagas reduce tight coupling between services.
  + **Flexibility:** Sagas are more flexible than traditional two-phase commit (2PC) transactions, especially for complex, long-lived transactions.
  + **Resilience:** Each step can be independently executed, allowing for resilience to network partitions and **partial** failures.
  1. Aggregator microservices

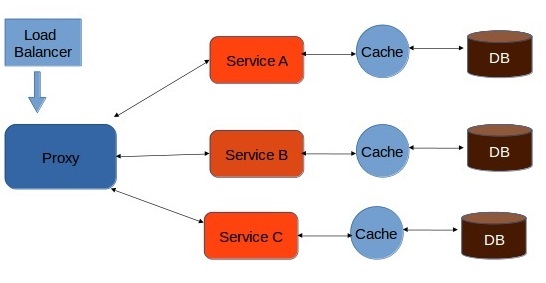
The Microservices Aggregator pattern helps aggregate responses from multiple microservices into a single unified response, optimizing client-server interactions in scalable systems.

Microservices Aggregator collects pieces of data from various microservices and returns an aggregate for processing.



* 1. Composite microservices
  2. Proxy microservice

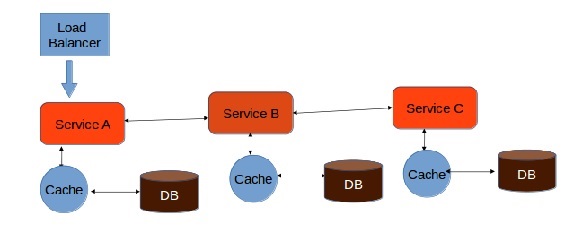
Proxy microservice pattern is a variation of the aggregator model. In this model we will use proxy module instead of the aggregation module. Proxy service may call different services individually.



In Proxy pattern, we can build one level of extra security by providing a dump proxy layer. This layer acts similar to the interface.

* 1. Chained microservice

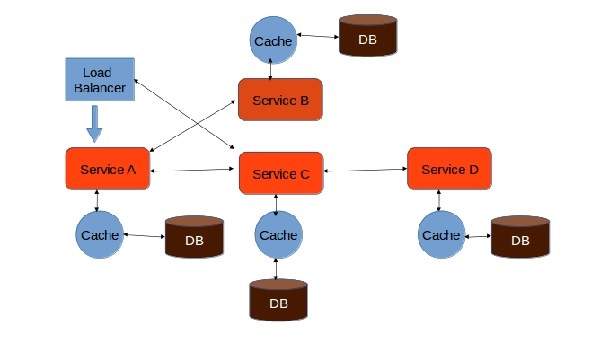
This type of composition pattern follows a chain structure. There is nothing between the client and service layer. Instead, the client is allowed to communicate directly with the services and all the services will be chained up in a such a manner that the output of one service will be the input of the next service.



One major drawback of this architecture is, the client will be blocked until the entire process is complete. Thus, it is highly recommendable to keep the length of the chain as short as possible.

* 1. Branch microservice

Branch microservice is the extended version of aggregator pattern and chain pattern. In this design pattern, the client can directly communicate with the service. Also, one service can communicate with more than one services at a time.



Branch microservice pattern allows the developer to configure service calls dynamically. All service calls will happen in a concurrent manner, which means service A can call Service B and C simultaneously.

**Spring Boot Questions**

1. What are the different SpringBoot Annotations
   1. @Component vs @Service @Repository
   2. @ComponentScan(base = “package name”)
   3. @SpringBootApplication contains what other annotations?
      1. @Configuration, @EnableAutoconfiguration , @ComponentScan
2. What is dependency injection and inversion of control?
   1. What are the different ways to inject the dependency
      1. Responsibility is to the Spring Container to handle the lifecycle of the components and their dependencies.
3. What are the different scopes of the bean?
   1. What is the default scope of Spring Bean?
   2. **SpringBoot Singleton Bean difference vs Singleton Pattern?**
   3. How does the Transaction Handling works in the Spring framework?
   4. What if in the same Transaction, we call the other method, will it be in the same transaction or not???
      1. Isolation or Propagation level?
4. @Configuration

@EnableAutoconfiguration

@ComponentScan

Scope of the bean by default is singleton

Three ways to inject dependency

1. Method level
2. Constructor level
3. Property level

IOC is basically by which the control of objects is transferred to a container.

1. **What is @Cacheable? How to use it?**
2. What are Predicates? What is the purpose of Predicates?
3. **Spring Boot @Async - how does it work?**
   1. **RestTemplate?**
   2. **TransactionTemplate?**
   3. **Non Blocking / Callback?**

Java Stream Api?

Java Collectors:

1. How to partition list into 2 different collections based on the Employee Age, and Department.
   1. How to Use Collectors.groupBy
   2. How to use Collectors.partitionBy
2. What are the benefits of Stream Api?
   1. Iteration, predicates, filtering, conversion, sorting.
   2. Functional inheritance?
3. How to collect the Map of Employee objects with ID and the value?
   1. Collections.toMap()
4. What is the Stream filter type?
5. How do you collect to List?

**Java Questions ((7+ years’ experience)**1. Difference between Final, Finally and Finalize  
 2. What are Threads and Process  
 3. Could you explain Public, Private and Proctected access specifiers.  
 4. Could you explain about Collections in Java and different collections classes used?  
 5. Difference between Array and ArrayList  
 6. What are the different ways to create Thread and which way do you prefer and why?  
 7. Time complexity for ArrayList, LinkedList, HashMap.  
 8. Internal implementation of Hashmap.  
 9. Does hashmap follow the insertion order if Not which collection class will follow the insertion order.  
 10. How does the arrayList size increases and how does it internally implement.  
 11. How does the Map size increase and how does it internally implement.  
 12. Program: Move all zero's to end of Array without using Temp array.

2. What is the initiated point of microservices in spring boot.

3. How can we technically define the streams in java.|  
 Caching in java  
 Circuit Breaker Design pattern in java

* SpringBoot

Spring Boot is a Java based framework which can create a stand-alone production grade Spring Application which can run independently. Spring Boot application required minimal Spring Configuration. Spring Boot comes great features like

* + Creating a stand-alone application
  + Comes with embed application server line Tomcat/Jetty/Undertow directly which removes the requirement of making .war files and directly runs a .jar file which can internally initiate the server.
  + It provides opinionated or pre configured ‘starter’ dependencies to simplify the build configuration.
  + It automatically configures Spring or third party libraries whenever possible.
  + Provides production-ready features such as metrics, health checks and externalized Configuration
  + No code generation and no requirement for XML configuration
* Java 17
* React / Angular?
* Microservices Design Approach
* DevOps practices
* CI CD CT
* PostgreSQL
* Restful API Security