**Microservices:** Microservices is an architectural style where a software application is broken down into **small, independent services** that:

* Run **in their own process**
* Communicate via **lightweight protocols** (like HTTP/REST or messaging queues)
* Are **independently deployable**
* Focus on **a single business capability**

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Decentralized** | Each service has its **own database**, logic, and responsibility. |
| **Independently Deployable** | One service can be deployed or scaled without affecting others. |
| **Lightweight Communication** | Services talk via **REST APIs**, **gRPC**, **RabbitMQ**, **Kafka**, etc. |
| **Fault Isolation** | Failure in one service does **not crash** the entire system. |
| **Technology Diversity** | Different services can be written in **different languages/frameworks**. |
| **DevOps Friendly** | Suited for **CI/CD**, **containers**, and **cloud** deployments. |

**Disadvantages**:

|  |  |
| --- | --- |
| **Challenge** | **Explanation** |
| **Increased Complexity** | Managing multiple services, each with its own codebase, deployment, and database, can be complex. |
| **Difficult Distributed Debugging** | Tracing bugs across multiple services is harder than in a single monolith. |
| **Network Latency** | Communication between services over the network can introduce **latency and failure points**. |
| **Data Consistency** | Maintaining **ACID** transactions across services is difficult. You often rely on **eventual consistency**. |
| **Deployment Overhead** | Managing the CI/CD pipelines, containers, or Kubernetes for each service adds overhead. |
| **Higher Resource Usage** | Each service might run in a separate container or VM, leading to **higher memory and CPU** consumption. |
| **Complex Testing** | Integration and end-to-end testing become more difficult due to service dependencies. |
| **Service Coordination** | Orchestrating workflows across multiple services requires tools like **Saga Pattern**, orchestration engines, etc. |
| **Monitoring and Logging** | Requires advanced **observability tools** (like Prometheus, ELK stack, or Jaeger) for centralized monitoring and tracing. |
| **Team Skillset** | Teams need strong skills in **DevOps, containerization, monitoring, networking, resilience patterns**. |

**Microservices vs Monolithic Architecture**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Monolithic Architecture** | **Microservices Architecture** |
| **Definition** | A single unified application where all components are combined. | A collection of small, independently deployable services. |
| **Codebase** | Single codebase, shared database | Multiple codebases, database (one per service) |
| **Deployment** | Deploy the entire application at once | Deploy services independently |
| **Scalability** | Difficult to scale parts individually | Each service can be scaled independently |
| **Technology Stack** | Usually uses one tech stack | Can use different tech stacks per service |
| **Development Speed** | Faster to build initially | Slower start but better long-term agility |
| **Team Structure** | One large team works on entire app | Small, independent teams per service |
| **Fault Isolation** | Failure in one module may affect the whole app | Failure in one service doesn’t crash others |
| **Testing** | Easier to perform end-to-end testing | Complex integration testing required |
| **Performance** | Often faster due to fewer network calls | Slightly slower due to inter-service communication (e.g., REST, gRPC) |
| **Maintainability** | Harder to maintain as app grows | Easier to maintain due to modularity |

**Challenges of Microservices Architecture:**

|  |  |
| --- | --- |
| **Challenge** | **Explanation** |
| **Distributed System Complexity** | Microservices interact over the network, requiring robust handling of latency, failures, and retries. |
| **Service Coordination** | Deploying multiple services together requires orchestration tools (e.g., Docker, Kubernetes). |
| **Managing Distributed Transactions** | ACID transactions are harder to implement across services. Requires patterns like Saga or eventual consistency. |
| **Difficult to Trace Errors** | Tracing issues across multiple services needs centralized logging and monitoring tools. |
| **Inter-Service Communication Overhead** | More HTTP/gRPC calls mean increased latency and potential for message failure. |
| **API Compatibility** | Changes in one service’s API may break dependent services. Backward compatibility becomes crucial. |
| **Distributed Security Concerns** | Need to secure APIs, manage authentication (OAuth2/JWT), authorization, and data access at service level. |
| **Database per Service** | Ensuring proper data separation and handling joins/reports becomes challenging. |
| **Infrastructure Overhead** | Needs CI/CD, containers, load balancing, service discovery, and monitoring for each service. |
| **Complex Integration Testing** | Unit tests are easier, but end-to-end and contract testing between services is more complex. |
| **Service Ownership & Boundaries** | Requires strong domain modeling and clearly defined service boundaries to avoid overlap or dependency hell. |

**Microservices (MSA) vs Service-Oriented Architecture (SOA)**

|  |  |
| --- | --- |
| **Microservice Based Architecture (MSA)** | **Service-Oriented Architecture (SOA)** |
| Microservices uses **lightweight protocols** such as **REST**, and **HTTP**, etc. | SOA supports **multi-message protocols**. |
| It focuses on **decoupling**. | It focuses on application service **reusability**. |
| It uses a **simple messaging system** for communication. | It uses **Enterprise Service Bus** (ESB) for communication. |
| Microservices follows "**share as little as possible**" architecture approach. | SOA follows "**share as much as possible architecture**" approach. |
| Microservices are much better in **fault tolerance** in comparison to SOA. | SOA is not better in fault tolerance in comparison to MSA. |
| Each microservice have an **independent** database. | SOA services share the **whole** data storage. |
| used **modern** relational DB.(NoSQL, RDBMS) | SOA used **traditional** relational databases. |
| **MSA minimizes sharing by using bounded contexts**—each service owns its logic and data, reducing dependencies. | Shared components and data |
| It is better suited for the **smaller** and **well portioned**, web-based system. | It is better for a **large** and **complex** business application environment. |

**Microservices Monitoring**

Monitoring is the control system of the microservices. There are **five** principles of monitoring microservices, as follows:

* Monitor the container and what's inside it.
* Alert on service performance.
* Monitor services that are elastic and multi-location.
* Monitor APIs.
* Monitor the organizational structure.
* **Configure Eureka Server, same for Spring-boot-admin dashboard**

1. **Create the Eureka Server**(default port:8761):

**Add dependency:** spring-cloud-starter-netflix-eureka-server

**@EnableEurekaServer**

@EnableAdminServer for better UI.

Properties: eureka.client.register-with-eureka=false, eureka.client.fetch-registry=false

1. **Configure All Microservices as Eureka Clients:**

**Add dependency:** spring-cloud-starter-netflix-eureka-client

**@EnableEurekaClient or** @EnableDiscoveryClient

1. Properties: **eureka.client.register-with-eureka=true, eureka.client.fetch-registry=true**

**Service Registry:** A **Service Registry** is a **centralized directory** where microservices **register themselves with (such as IP, port, service name)** and **discover each other** dynamically without hardcoding network locations. This is a key pattern in **Service Discovery**, enabling **scalable**, **resilient**, and **loosely coupled** microservice architectures.

**Service Discovery: Service Discovery** is the mechanism that allows a **service or client** to **locate a service instance** from the **registry**, instead of using hardcoded addresses.

* It asks **Eureka**: “Where is service?”
* Eureka replies with IPs/Ports of healthy instances.

**Without Service Registry?**

* You must **hardcode IPs and ports**.
* You **lose flexibility** for scaling or deployment.
* Difficult to manage **failovers** and **load balancing**.

**1. Client-Side Service Discovery**

**🔧 How it works:**

In client-side discovery, **the client is responsible for:**

* Querying the **Service Registry** to get available service instances.
* **Load balancing** the requests (selecting one of the instances).
* Calling the selected instance directly.

**🔄 Example Flow:**

1. OrderService wants to call UserService.
2. OrderService queries Eureka for UserService instances.
3. Eureka responds with a list of healthy instances:

- http://10.0.0.12:8080

- http://10.0.0.15:8080

1. OrderService (via Ribbon or LoadBalancerClient) picks one and makes the call directly.

**2. Server-Side Service Discovery**

**🔧 How it works:**

In server-side discovery, the **client doesn’t query the registry directly**. Instead:

* The client sends the request to a **gateway or load balancer** (like Zuul or Spring Cloud Gateway).
* The **gateway queries the Service Registry** to find the target service instance.
* The **gateway forwards the request** to the selected service.

|  |  |
| --- | --- |
| **Situation** | **Recommended Approach** |
| Internal service-to-service calls | **Client-side discovery** |
| Front-end or external client calls | **Server-side discovery** |
| Centralized control/routing needed | **Server-side discovery** |
| Simpler microservices | **Client-side discovery** |

**In microservices:**

* Services are **ephemeral** (they start, stop, scale dynamically).
* IPs and ports change frequently (e.g., in Docker/Kubernetes).
* You **cannot hardcode** IP addresses.

A **Service Registry** helps solve this by maintaining a **real-time map** of available service instances.

**How to call the API by service name except IP: port**

Because we call the API using RestTemplate.RestTemplate is not auto configured so We need to create a bean for this. Using @Bean annotation.

For calling the API by Service name, we apply the @**LoadBalanced** annotation on the RestTemplate Bean.

**Feign Client:** The Feign client is a declarative HTTP Web client.

* **Add dependency**: spring-cloud-starter-openfeign
* Apply **@EnableFeignClients** in the spring main class
* Create an interface for the service. Apply @**FeignClient(name=”your eureka service/app name”** on this interface
* Create the same controller mapping/methods as the micro client service in this service.
* Call these interface method using auto wired in your serviceimpl instead

**API Gateway:**

**dependency:** spring-cloud-starter-**gateway** & spring-boot-starter-**webflux, eurekaclient**

**Note**: plz don’t add **web** dependency

**Add @EnableEurekaClient in main class**

**Add properties**: id, uri, predicate

spring.cloud.gateway**.routes[0].id= USERSERVICE**

spring.cloud.gateway.**routes[0].uri=lb:// USERSERVICE**

spring.cloud.gateway.**routes[0].predicates[0]=Path=/users/\*\*, /hotelStaff/\*\***

spring.cloud.gateway**.routes[1].id=HOTELSERVICE**

spring.cloud.gateway.**routes[1].uri=lb://HOTELSERVICE**

spring.cloud.gateway.**routes[1].predicates[0]=Path=/rating/\*\***

**If both services must share the same base path (e.g., /users/\*\*), you can:**

* Use additional predicates (like **Header, Method**, etc.).
* Add a custom filter to distinguish them.

**WebFlux:**  WebFlux is **non-blocking** and **reactive**, designed for handling high concurrency with fewer resources

**Setting up Spring Cloud Config Server:**

* Create a Spring Boot Project (Config Server)
* Add dependency: **spring-cloud-config-server**
* @**EnableConfigServer** enable config server in main class
* Add properties: **spring.cloud.config.server.git.uri=your git repo URL**

**spring.cloud.config.server.git.clone-on-start=true**

* Git Repo Structure (e.g., on GitHub)
  + your-config-repo(microservice-spring-config)/
  + ├── application.yml
  + ├── application-dev.yml
  + └── application-prod.yml
* Start the config server **mvn spring-boot:run**
* **For testing from browser: http://localhost:9092/application/profile(default/dev/prod)**
* Read server config in Client Microservice
* Add dependency **spring-cloud-config-client**
* **Add property: spring.config.import=optional:configserver:http://localhost:9092**
* For reding **application-dev.yml,** add property**: profiles.active=dev. We can also activate the profile while running the spring jar.**

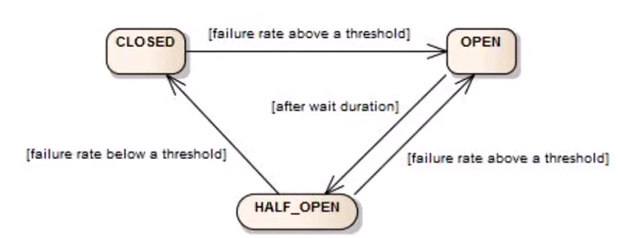
We can add bootstrap.yml or bootstrap.properties. because bootstrap.yml is loaded before application.yml, which is important for configuration loading.

Optional: Add Security

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| **Config Server** | Hosts and serves configurations |
| **Git/File backend** | Stores externalized config files |
| **Client Microservices** | Fetch config from server at startup |

**Fault Tolerance:**

**Circuit Breaker(Resilemce4j)**



* 1. Add dependency: spring-boot-starter-actuator, -aop, resilence4j-spring-boot2
  2. Apply annotation in controller method: @**CircuitBreaker**(name="ratingHotelBreaker/ same as in below property ",**fallbackMethod** = "ratingHotelFallback/customMethod")
  3. **Note**: controller API method and fallback method return type and parameter should be the same
  4. **Add properties**:
* management.**health.circuitbreakers.enabled**=true
* management.**endpoints.web.exposure.include**=health
* management.**endpoint.health.show-details**=always
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**registerHealthIndicator**=true
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**slidingWindowSize**=10
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**minimumNumberOfCalls**=5
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**failureRateThreshold**=50
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**waitDurationInOpenState**=10s
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**permittedNumberOfCallsInHalfOpenState**=3
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**automaticTransitionFromOpenToHalfOpenEnabled**=true
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**slidingWindowType**=COUNT\_BASED
* resilience4j.circuitbreaker.instances.ratingHotelBreaker.**eventConsumerBufferSize**=10

**Retry.:**

* apply on controller method: @**Retry**(name="ratingHotelService/ same as in below property ",**fallbackMethod** = "ratingHotelFallback")
* Add property:

resilience4j.retry.instances. ratingHotelService.**maxAttempts**=3

resilience4j.retry.instances. ratingHotelService.**waitDuration**=2s

**RateLimiter:**

* Apply @**RateLimiter**(name = "userRateLimiter/same as in below property", **fallbackMethod** = "ratingHotelFallback")
* Add property:

resilience4j.ratelimiter.instances.userRateLimiter.**limitForPeriod**=5

resilience4j.ratelimiter.instances.userRateLimiter.**limitRefreshPeriod**=1s

resilience4j.ratelimiter.instances.userRateLimiter.**timeoutDuration**=500ms

**To enable OAuth 2.0 authentication in a Spring Boot application**

1. Add Required Dependencies
2. Configure application.properties with **clientid, secretid, issuer-uri, and jwt-set-uri**

**spring.security.oauth2.resourceserver.jwt.issuer-uri**=https://<your-okta-domain>/oauth2/default

**spring.security.oauth2.resourceserver.jwt.jwk-set-uri**=https://<your-okta-domain>/oauth2/default/v1/keys

**spring.security.oauth2.client.registration.okta.client-id**=<client-id>

**spring.security.oauth2.client.registration.okta.client-secret**=<client-secret>

1. Set Up **Security Configuration** (Optional if defaults are okay). Add **@EnableWebSecurity**

http.**authorizeHttpRequests**(auth -> auth.**requestMatchers**("/public/\*\*").**permitAll**()

.anyRequest().**authenticated**() ).**oauth2ResourceServer**().**jwt**();return http.build();

1. (Optional) Add Role-Based Access with Scope

On Method() **@PreAuthorize("hasAuthority('SCOPE**\_admin')")

**LDAP**

I integrated LDAP in Spring Boot using Spring Security by configuring **LdapContextSource**, applying **userSearchFilter**, and enabling login through an LDAP directory like OpenLDAP. Spring handles authentication by validating users against the LDAP tree structure.

1. **Add Required Dependencies**
2. Configure application.properties

spring.ldap.urls=ldap://localhost:8389

spring.ldap.base=dc=springframework,dc=org

spring.ldap.username=cn=admin,dc=springframework,dc=org

spring.ldap.password=admin

# Optional: user search filter

spring.security.ldap.user-search-filter=uid={0}

spring.security.ldap.user-dn-patterns=uid={0},ou=people

1. Create a Security Configuration Class

@Bean

public LdapContextSource contextSource() {

LdapContextSource contextSource = new LdapContextSource();

contextSource.setUrl("ldap://localhost:8389");

contextSource.setBase("dc=springframework,dc=org");

contextSource.setUserDn("cn=admin,dc=springframework,dc=org");

contextSource.setPassword("admin");

return contextSource;}

1. **Use Spring Security Default Login Page**