

MicroServices: Microservices is an architectural style where an application is built as a collection of small, independent services that communicate with each other via APIs. Each microservice focuses on a specific functionality, making the system more scalable, flexible, and maintainable.

We can refer full code what we are discussing here and you can get this document in git also.

Link : https://github.com/Prasannagk67/Blog_Website_Microservices_Project

Key Characteristics of Microservices

- Independence → Each service can be developed, deployed, and scaled independently.
- Decentralized Data Management → Each microservice has its own database (or storage).
- API Communication → Services interact via REST APIs, gRPC, or messaging (RabbitMQ, Kafka).
- Technology Flexibility → Different services can be written in different programming languages.
- Resilience → If one microservice fails, the rest of the system continues to work.

Microservices vs Monolithic Architecture

Feature	Monolithic Architecture	Microservices Architecture
Development	Single codebase, tightly coupled	Multiple small services, loosely coupled
Deployment	Deploy everything together	Deploy services independently
Scalability	Hard to scale individual components	Easily scalable per service
Failure Handling	Single failure affects the whole app	Failure in one service does not crash others
Technology	Uses one tech stack	Can use different technologies per service

Benefits of Microservices :

- Scalability : Scale only the needed service (e.g., Product Service)
- Faster Deployment : Update one service without redeploying everything
- Fault Tolerance : If one service fails, others keep running
- Technology Diversity : Use different programming languages per service.

Challenges of Microservices :

- Complexity : Managing many small services is harder
- Networking Issues : More API calls increase latency
- Data Consistency : Requires strategies like Event Sourcing
- Security : Each service needs authentication & authorization

Example: Blog Website Microservices Architecture

A Blog Website consists of three microservices: User Service, Blog Service, and Comment Service. These services are registered using Eureka Server (Service Registry). Synchronous communication is handled via RestTemplate and Feign Client. An API Gateway manages requests, while a Configuration Server is used for exporting and importing Eureka configurations. To enhance resilience, we implement a Circuit Breaker, Retry Mechanism, and Rate Limiting, with JMeter.

Entities : The User entity stores user details with fields like id, name, email (unique), password, and signUpDate. The Blog entity contains id, name, and link to represent blog posts. The Comment entity links users and blogs using userId and blogId, with additional fields for comment and commentedDate.

Service Registry : A Service Registry is a centralized system that keeps track of all microservices and their locations (IP addresses and ports). It helps microservices dynamically discover and communicate with each other without hardcoding service locations. Service Registry also a one separate Service in Microservices

When working with individual services, we access them using their specific port numbers in URLs. However, in a microservices architecture with multiple services, managing different port numbers becomes complex. Additionally, when one service needs to call another, tracking these dynamic ports manually is difficult. To solve this, we use a Service Registry, which registers all services and manages their locations, making service discovery and communication seamless.

Service Discovery is a mechanism that enables microservices to dynamically locate and communicate with each other without hardcoding service URLs. It helps microservices automatically register, discover, and interact with other services, even when instances scale or change dynamically.

Service Registry is like a phone directory where all services register their contact details (IP & port). Service Discovery is the process of looking up that directory to find and communicate with a specific service.

Eureka is a service registry and discovery tool developed by Netflix for microservices architecture. It allows services to register themselves and enables other services to discover and communicate dynamically without hardcoded URLs.

Eureka Server: Acts as a service registry where microservices register themselves. Role: Stores service details (IP, port, status) and enables discovery.	<code><dependency></code> <code><groupId>org.springframework.cloud</groupId></code> <code><artifactId>spring-cloud-starter-netflix-eureka-server</artifactId></code> <code></dependency></code>
Eureka Client: Microservices that register with Eureka Server and discover other services. Role: Connects to Eureka Server to register itself and find other services.	<code><dependency></code> <code><groupId>org.springframework.cloud</groupId></code> <code><artifactId>spring-cloud-starter-netflix-eureka-client</artifactId></code> <code></dependency></code>

Simple Explanation: Eureka Server is like a phone directory where services register their addresses And Eureka Client is like a person using the directory to register itself and find others.

Some Alternatives for Eureka are Consul, Zookeeper, Kubernetes Service Discovery etc..

Below class and application.properties initializes a Spring Boot Eureka Server, enabling microservices to register and discover each other dynamically. It eliminates manual URL management, enhancing scalability and fault tolerance. The configuration sets up a Service Registry on port 9994, disabling client registration and discovery since it acts as the central registry, with services connecting locally.

<code>spring.application.name=ServiceRegistry</code> <code>server.port=9994</code> <code>eureka.instance.hostname=localhost</code> <code>eureka.client.register-with-eureka=false</code> <code>eureka.client.fetch-registry=false</code>	<code>@SpringBootApplication</code> <code>@EnableEurekaServer</code> <code>public class ServiceRegistryApplication {</code> <code> public static void main(String[] args) {</code> <code> SpringApplication.run(ServiceRegistryApplication.class, args);</code> <code> }</code> <code>}</code>
--	---

This Eureka Client configuration ensures the service registers using its IP address instead of the hostname. The defaultZone specifies the Eureka Server URL (http://localhost:9994/eureka/) where the client will register. This enables dynamic service discovery in the microservices architecture.

<code>//@EurekaDiscoveryClient registers UserService with the Service Registry, enables service discovery, same for all services.</code> <code>@SpringBootApplication</code> <code>@EnableDiscoveryClient</code> <code>public class UserServiceApplication {</code> <code> public static void main(String[] args) {</code> <code> SpringApplication.run(UserServiceApplication.class, args);</code> <code> }</code> <code>}</code>
<code># Eureka Client Configuration</code> <code>eureka.instance.prefer-ip-address=true</code>

```
eureka.client.service-url.defaultZone=http://localhost:9994/eureka/
```

If we search `http://localhost:9994/`, we will see the list of services registered in the Eureka Server.

The screenshot shows the Spring Eureka Server web interface. The top navigation bar includes 'HOME' and 'LAST 1000 SINCE STARTUP'. The 'System Status' section displays the following information:

Environment	test	Current time	2025-02-28T15:21:49 +0530
Data center	default	Uptime	05:17
		Lease expiration enabled	true
		Renews threshold	6
		Renews (last min)	10

The 'DS Replicas' section shows 'Instances currently registered with Eureka'.

Application	AMIs	Availability Zones	Status
BLOG-SERVICE	n/a (1)	(1)	UP (1) - IMSS-JK-LAP-111.imsspl.com:BLOG-SERVICE-9992
COMMENT-SERVICE	n/a (1)	(1)	UP (1) - IMSS-JK-LAP-111.imsspl.com:COMMENT-SERVICE-9993
USER-SERVICE	n/a (1)	(1)	UP (1) - IMSS-JK-LAP-111.imsspl.com:USER-SERVICE-9991

Sometimes, one service needs to call another service. Now, let's see how to do this using RestTemplate and Feign Client.

RestTemplate : RestTemplate is a Spring Boot HTTP client used to make synchronous REST API calls between microservices. It simplifies GET, POST, PUT, DELETE requests.

Key Features:

- Calls Other Services : Used for inter-service communication.
- Handles HTTP Requests : Supports GET, POST, PUT, DELETE, etc.
- Integrates with Eureka : When annotated with `@LoadBalanced`, it resolves service names dynamically.
- Supports Customization : Allows setting headers, authentication, and error handling.

`@LoadBalanced` is a Spring annotation that enables client-side load balancing for RestTemplate. It allows microservices to call others using their service names instead of hardcoded URLs.

How It Works: When a service is registered in Eureka, it may have multiple instances. `@LoadBalanced` automatically distributes requests among these instances using Ribbon (deprecated) or Spring Cloud LoadBalancer.

Example Code:

```
@SpringBootApplication
@EnableDiscoveryClient
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

```
@RestController
@RequestMapping("/user")
public class UserController {
    @Autowired
    UserServiceImpl userServiceImpl;
    @Autowired
    RestTemplate restTemplate;
    @GetMapping("/blogs")
    public ResponseEntity<Object> getAllBlogs(){
        String blogServiceUrl = "http://BLOG-SERVICE/blog";
        Object blogs =
            restTemplate.getForObject(blogServiceUrl,Object.class);
        return ResponseEntity.ok(blogs);
    }
}
```

Feign Client: Feign Client is a declarative HTTP client in Spring Boot that simplifies inter-service communication in a microservices architecture. Instead of manually using `RestTemplate`, Feign provides a cleaner, interface-based approach to calling REST APIs.

Key Features:

- Declarative API Calls : No need for RestTemplate, just define an interface.
- Service Name Resolution : Uses Eureka service names instead of hardcoded URLs.
- Load Balancing Support : Works with Spring Cloud LoadBalancer for distributing requests.
- Built-in Support for Headers & Authentication : Easily add headers like JWT tokens.

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class CommentServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(CommentServiceApplication.class, args);
    }
}
```

```
package com.blog.feignclint;
@FeignClient(name = "BLOG-SERVICE")
public interface BlogFeignClient {
    @GetMapping("/blog")
    List<BlogDTO> getAllBlogs();
    // we can get some implementations in github
}
```

```
@RestController
@RequestMapping("/comment")
public class CommentController {
    @Autowired
    CommentServiceImpl commentServiceImpl;
    @Autowired
    BlogFeignClient blogFeignClient;
    @GetMapping("/blogs")
    public ResponseEntity<List<BlogDTO>> getAllBlogsFromBlogService() {
        List<BlogDTO> blogs = blogFeignClient.getAllBlogs();
        return ResponseEntity.ok(blogs);
    }
}
```

API Gateway: An API Gateway is a central entry point for managing and routing requests in a microservices architecture. It acts as a reverse proxy, handling authentication, load balancing, logging, and security before forwarding requests to backend services. It's also one separate service in microservices.

Key Features:

- Single Entry Point : Clients interact with one endpoint instead of multiple services.
- Request Routing : Directs requests to the appropriate microservice.
- Security & Authentication : Handles authentication (JWT, OAuth, API Keys).
- Load Balancing : Distributes requests efficiently across service instances.
- Rate Limiting & Monitoring : Prevents API abuse and logs requests.

Even after registering services with the Service Registry, we still call each service using its own port. However, by adding an API Gateway, we can access any service through a single API Gateway endpoint instead of using individual service ports.

```
<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

This configuration defines a route in Spring Cloud Gateway for the USER-SERVICE. It forwards all requests matching the /user/** path to the corresponding microservice using Eureka service discovery. The lb://USER-SERVICE URI enables load balancing, ensuring requests are distributed across available instances. This setup centralizes API management, improving scalability, security, and maintainability. For other services also the same..

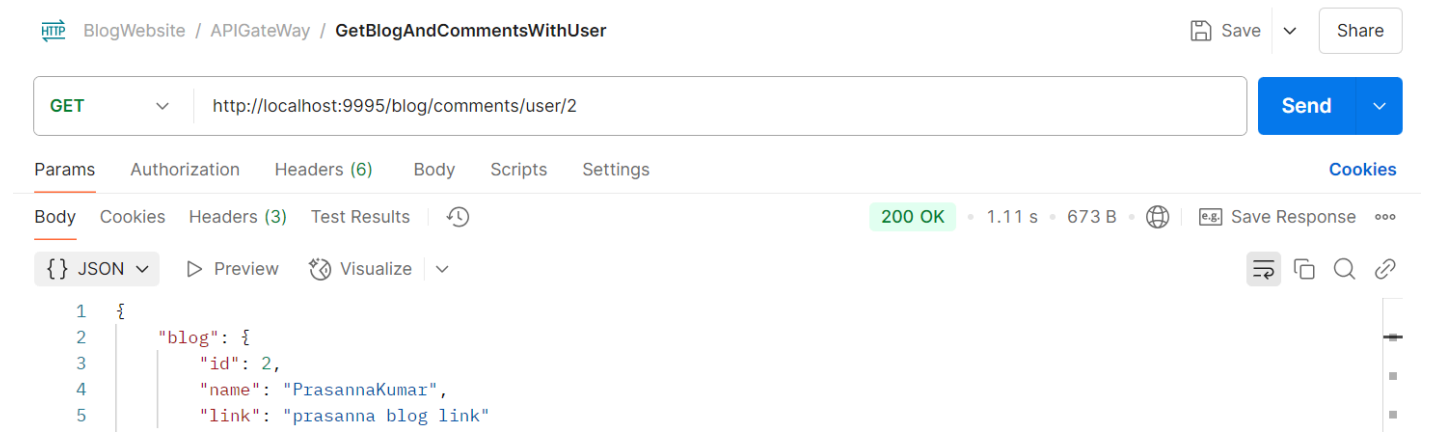
```
spring.application.name=API-GATEWAY
server.port=9995
# Eureka Client Configuration
eureka.instance.prefer-ip-address=true
eureka.client.service-url.defaultZone=http://localhost:9994/eureka/
# Defining Routes for Services
spring.cloud.gateway.routes[0].id=USER-SERVICE
spring.cloud.gateway.routes[0].uri=lb://USER-SERVICE
spring.cloud.gateway.routes[0].predicates[0]=Path=/user/**

spring.cloud.gateway.routes[1].id=BLOG-SERVICE
spring.cloud.gateway.routes[1].uri=lb://BLOG-SERVICE
spring.cloud.gateway.routes[1].predicates[0]=Path=/blog/**

spring.cloud.gateway.routes[2].id=COMMENT-SERVICE
spring.cloud.gateway.routes[2].uri=lb://COMMENT-SERVICE
spring.cloud.gateway.routes[2].predicates[0]=Path=/comment/**
```

```
@SpringBootApplication
@EnableDiscoveryClient
public class ApiGatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(ApiGatewayApplication.class, args);
    }
}
```

Here, we are calling the Blog Service, which runs on port 9992, through the API Gateway, which runs on port 9995



The screenshot shows a REST client interface with the following details:

- URL: `http://localhost:9995/blog/comments/user/2`
- Method: `GET`
- Status: `200 OK`
- Response Body (JSON):

```
{
  "blog": {
    "id": 2,
    "name": "PrasannaKumar",
    "link": "prasanna blog link"
  }
}
```

Configuration Server: A Configuration Server in microservices is a centralized system for managing configuration settings across multiple services. Instead of storing configurations inside each microservice, they are externalized and fetched dynamically from a central repository (e.g., Git, database, or filesystem).

Key Features:

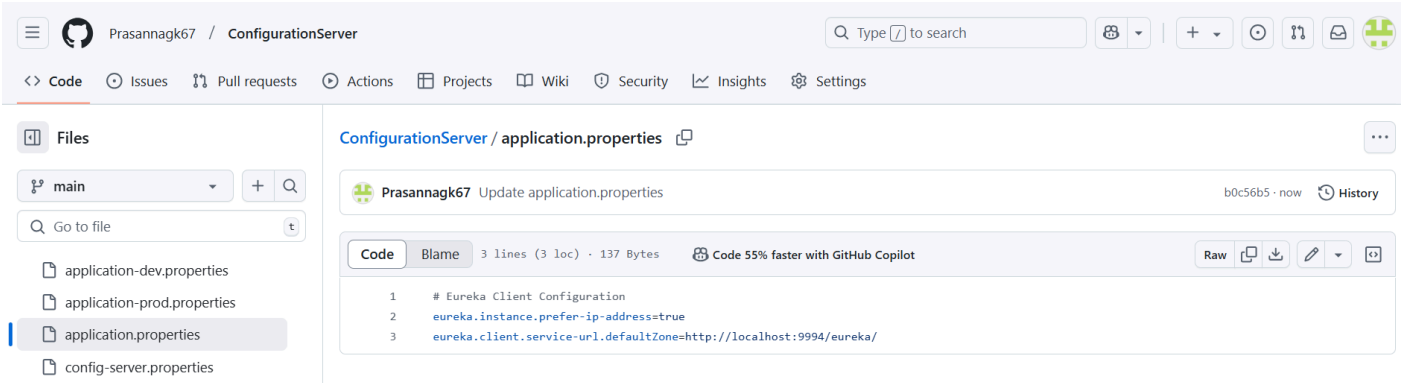
- Centralized Configuration Management : Stores all service configurations in one place.
- Dynamic Configuration Updates : Changes can be applied without redeploying services.
- Environment-Specific Configurations : Supports different settings for dev, test, and production.
- Security & Encryption : Manages sensitive properties securely (e.g., database passwords).

spring-cloud-config-server dependency enables a microservice to act as a centralized configuration server, managing external configurations for multiple services

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

The spring-cloud-starter-config dependency is used in client services to fetch configurations dynamically from the Config Server.	<pre><dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-starter-config</artifactId> </dependency></pre>
---	---

<pre>spring.application.name=CONFIG-SERVER server.port=9996 spring.cloud.config.server.git.uri=https://github.com/Prasannagk67/ConfigurationServer spring.cloud.config.server.git.clone-on-start=true</pre>	
<pre>@SpringBootApplication @EnableConfigServer public class ConfigServerApplication { public static void main(String[] args) { SpringApplication.run(ConfigServerApplication.class, args); } }</pre>	

	<pre>spring.application.name=BLOG-SERVICE server.port=9992 # Import Configuration from Config Server spring.config.import=configserver: spring.cloud.config.uri=http://localhost:9996 spring.profiles.active=application</pre>
--	--

Circuit Breaker: A Circuit Breaker helps prevent system failures in microservices by stopping requests to a service if it becomes too slow or unresponsive. It keeps checking the service and allows requests again once it recovers.

How Does It Work ?

- Closed State → Requests are sent normally.
- Open State → After failures exceed a threshold, the circuit opens, blocking requests.
- Half-Open State → After a cooldown period, limited requests are tested. If successful, it returns to Closed; otherwise, it stays Open.

resilience4j-spring-boot3 : Adds Resilience4j support for fault tolerance mechanisms like Circuit Breaker, Rate Limiter, and Retry in Spring Boot 3.	<pre><dependency> <groupId>io.github.resilience4j</groupId> <artifactId>resilience4j-spring-boot3</artifactId> <version>2.2.0</version> </dependency></pre>
spring-boot-starter-aop : Provides Aspect-Oriented Programming (AOP) support for handling cross-cutting concerns like logging and security.	<pre><dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-aop</artifactId> </dependency></pre>

spring-boot-starter-actuator : Enables monitoring and management of Spring Boot applications with built-in endpoints.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

```
@GetMapping("/comments/{blogId}")
@CircuitBreaker(name = COMMENT_SERVICE, fallbackMethod = "fallbackForComments")
public ResponseEntity<BlogAndComment> getCommentsByBlogId(@PathVariable int blogId) {
    Optional<Blog> blog = blogServiceImpl.getBlogById(blogId);
    if (blog.isEmpty()) {
        return ResponseEntity.notFound().build();
    }
    String commentServiceUrl = "http://COMMENT-SERVICE/comment/blogs/"+blogId;
    List<CommentDTO> comments = restTemplate.getForObject(commentServiceUrl, List.class);
    BlogAndComment blogAndComment = new BlogAndComment(blog.get(), comments);
    return ResponseEntity.ok(blogAndComment);
}

public ResponseEntity<BlogAndComment> fallbackForComments(int blogId, Throwable ex) {
    Optional<Blog> blog = blogServiceImpl.getBlogById(blogId);
    if (blog.isEmpty()) {
        return ResponseEntity.notFound().build();
    }
    List<CommentDTO> emptyComments = Collections.emptyList();
    BlogAndComment blogAndComment = new BlogAndComment(blog.get(), emptyComments);
    return ResponseEntity.ok(blogAndComment);
}
```

Retry Circuit Breaker: A Retry Circuit Breaker is a combination of Retry and Circuit Breaker patterns in microservices. Retry Automatically retries a failed request a few times before giving up. Circuit Breaker Prevents further requests to a failing service after repeated failures, allowing recovery time.

```
@GetMapping("/blogs")
@CircuitBreaker(name = BLOG_SERVICE, fallbackMethod = "fallbackForGetAllBlogs")
@Retry(name = BLOG_SERVICE, fallbackMethod = "fallbackForGetAllBlogs")
public ResponseEntity<Object> getAllBlogs() {
    String blogServiceUrl = "http://BLOG-SERVICE/blog";
    Object blogs = restTemplate.getForObject(blogServiceUrl, Object.class);
    return ResponseEntity.ok(blogs);
}

public ResponseEntity<Object> fallbackForGetAllBlogs(Throwable ex) {
    return ResponseEntity.ok("Blog Service is currently unavailable. Please try again later.");
}
```

Rate Limiter - JMeter: A **Rate Limiter** controls the number of requests a service can handle within a specific time to prevent overloading. **JMeter** is a performance testing tool used to simulate high traffic and test how a Rate Limiter behaves under load.

How Does It Work?

- Rate Limiter restricts requests (e.g., 10 requests per second).
- JMeter generates multiple requests to check if the limit is enforced.
- If requests exceed the limit, the service returns HTTP 429 (Too Many Requests).

```
@GetMapping("/blog/comments/{blogId}")
@RateLimiter(name = BLOG_RATE_LIMITER, fallbackMethod = "fallbackForRateLimiter")
public ResponseEntity<Object> getBlogAndComments(@PathVariable int blogId) {
    String blogAndComment = "http://BLOG-SERVICE/blog/comments/"+blogId;
    Object blogAndComments = restTemplate.getForObject(blogAndComment, Object.class);
    return ResponseEntity.ok(blogAndComments);
}

public ResponseEntity<Object> fallbackForRateLimiter(int blogId, Throwable ex) {
    return ResponseEntity.status(429).body("Too many requests! Please try again later.");
}
```