**MICROSERVICES**

**MONOLITH ARCHITECTURE**

1.All components are part of single unit .

2.Everything is developed,deployed and scaled as 1 unit.

3.App must be written in 1 tech stack.

4.Teams need be to careful to not affect each other’s work.

5.1 single artifact(WAR file),so you must redeploy the entire application on each update.

**CHALLENGES OF MONOLITH ARCHITECTURE**

1.Application is too large and complex(Large code base).

2.Parts are more tightly coupled each other.

3.You can only scale the entire app,instead of a specific service or feature(higher infrastructure costs).

4.Difficult if services need different dependency versions.

5.Release process takes longer-On every change,entire application needs to be tested and entire application needs to be built and deployed.

6.Bug in one module can potentially bring down the entire application.

**HOW MICROSERVICES SOLVE THE CHALLENGES**

1.Smaller code base for every services.

2.All services are loosely coupled.

3.Can scale a specific service as per requirement.

4.Can add dependency to required services.

5.On any change the specific service can be redeployed.

6.If there is any bug in one service it will not affect others.

**TRANSITION FROM A MONOLITHIC ARCHITECTURE TO MICROSERVICES**

Deciding when to transition from a monolithic architecture to microservices depends on several factors. While microservices offer scalability and flexibility, they also introduce complexity. Here are some key indicators that suggest it's time to make the shift:

Signs You Should Consider Microservices

1. Scalability Issues
   * If your monolithic application struggles to handle increasing load and scaling parts of the system independently becomes difficult.
2. Development Bottlenecks
   * Teams are blocked waiting for deployments or struggling with conflicting dependencies.
   * New features take longer because changes affect multiple parts of the application.
3. Frequent Deployments & Maintenance Challenges
   * If every small change requires a full system deployment, making updates and fixes becomes slow and risky.
4. Need for Technology Diversity
   * If different components require different programming languages or frameworks, a monolithic structure may limit innovation.
5. Complex Codebase
   * If the application is becoming too hard to maintain due to intertwined dependencies, making isolated changes becomes increasingly difficult.
6. Business Agility Requirements
   * If different teams need autonomy to work on independent features without affecting others.

When to Stay Monolithic

* If your application is small and does not require independent scaling.
* If your team lacks experience with microservices and managing distributed systems.
* If operational overhead (like monitoring, logging, and inter-service communication) outweighs the benefits.

How to Approach the Transition

1. Identify Boundaries: Start by defining separate functional areas of your application.
2. Gradually Extract Services: Pick one module and migrate it to a microservice before moving others.
3. Implement API Gateway: Manage communication between services effectively.
4. Monitor & Optimize: Ensure observability to track system health and performance.

**MICROSERVICE ARCHITECTURE BENEFITS AND BEST PRACTICES**

1.The best practice is to break down the application into components or into microservices based on the business functionalities and not technical functionalities.

2.Separation of concerns: 1 Service for 1 specific job (Single Responsibility principle).

3.A very important characteristic of each microservice is that they should be self-contained and independent from each other this means each service must be able to be developed,deployed and scaled separately without any tight dependencies on any other services even though they are part of the same application.

4.Release process don’t take longer time.

5.Can independently scale up highly used services.

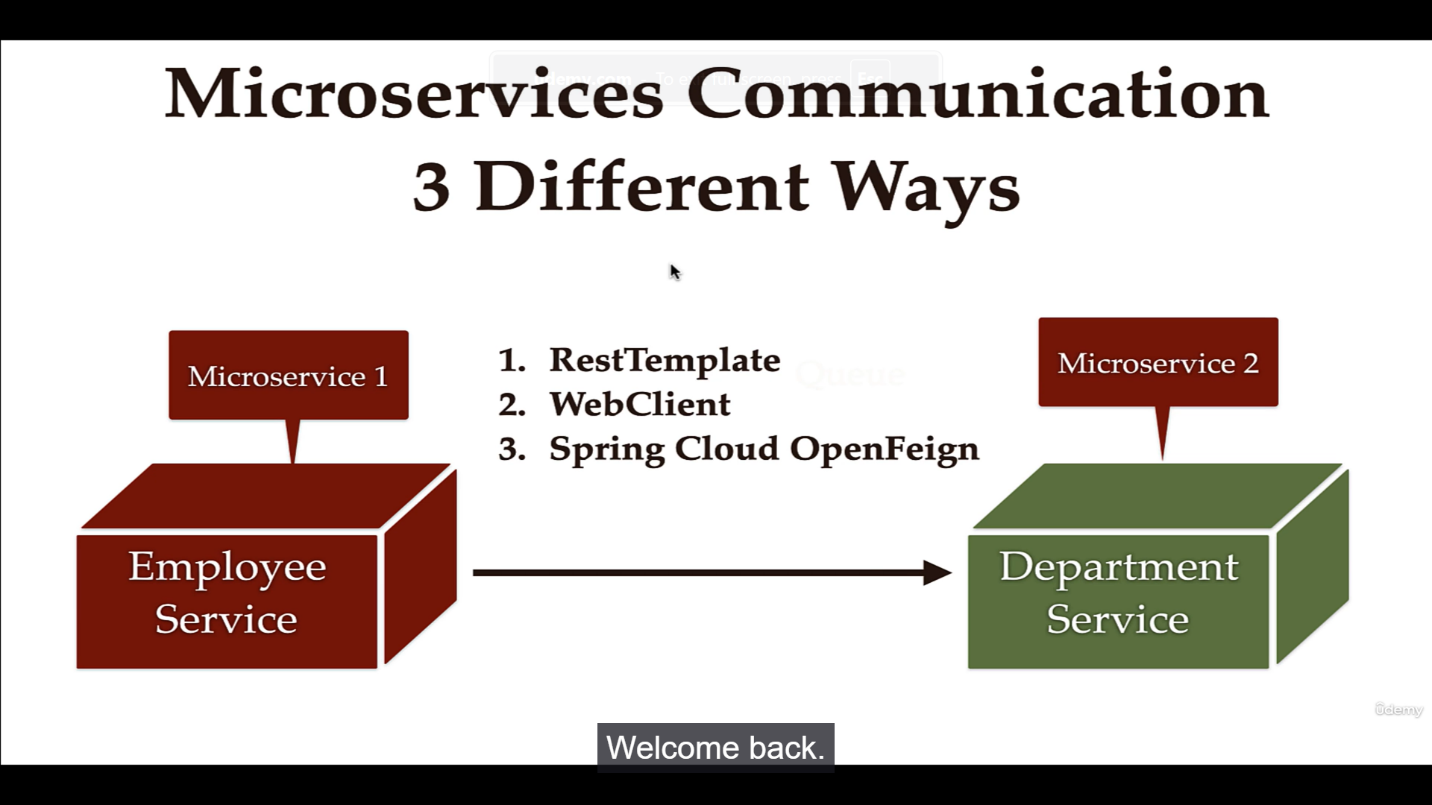
6.Technology updates/rewrites become simpler.

<https://github.com/RameshMF/springboot-microservices/tree/main/v3/springboot-microservices>

[springboot-microservices/v3/springboot-microservices at main · RameshMF/springboot-microservices](https://github.com/RameshMF/springboot-microservices/tree/main/v3/springboot-microservices)

**MICROSERVICES COMMUNICATION**

https://github.com/RameshMF/springboot-microservices/tree/main/v3/springboot-microservices

****

**A diagram of a communication system

AI-generated content may be incorrect.**

**A diagram of a chat

AI-generated content may be incorrect.**

Microservices communication is crucial for building distributed systems where services interact efficiently. In **Spring Boot**, three common ways to facilitate communication between microservices are **RestTemplate**, **WebClient**, and **Spring Cloud OpenFeign**. Here’s how each method works:

Let’s assume we have two microservices:

* **Employee Service** (calls Department Service)
* **Department Service** (provides department data)

**1)MICROSERVICES COMMUNICATION USING RESTTEMPLATE**

**RestTemplate (Synchronous, Traditional Approach)**

RestTemplate is a widely used synchronous HTTP client for calling RESTful APIs. It is simple but requires manual handling of responses.

**Pros:** ✅ Simple and easy to use. ✅ Works well for synchronous communication.

**Cons:** ❌ Blocks the calling thread, leading to potential performance issues in high-traffic environments. ❌ Does not support reactive programming.

There are two microservices employee-service and department-service.

**Development Steps**

1.Add departmentCode field in Employee JPA Entity

2.Create DepartmentDto class

3.Configure RestTemplate as Spring Bean

4.Inject and use RestTemplate to make REST API call in EmployeeServiceImpl class

**Step 1: Define RestTemplate Bean**

java

@Configuration

public class RestTemplateConfig {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

}

**Step 2: Inject & Use RestTemplate**

java

@Service

public class EmployeeService {

private final RestTemplate restTemplate;

@Autowired

public EmployeeService(RestTemplate restTemplate) {

this.restTemplate = restTemplate;

}

public DepartmentDto getDepartment(String departmentCode) {

return restTemplate.getForObject("http://localhost:8081/api/departments/" + departmentCode, DepartmentDto.class);

}

}

✅ **Synchronous request** ✅ **Simple to implement** ❌ **Not ideal for high-load systems**

**RestTemplate Flow (Synchronous)**

* A service calls another microservice via a blocking HTTP request.
* The request waits for the response before proceeding.
* Best suited for traditional synchronous applications.

**2)MICROSERVICES COMMUNICATION USING WEBCLIENT**

WebClient is a modern alternative to RestTemplate, supporting asynchronous and reactive interactions.

**Pros:** ✅ Non-blocking and fully asynchronous. ✅ Ideal for high-performance systems and reactive applications.

**Cons:** ❌ Requires reactive programming knowledge (Mono and Flux). ❌ Might not be the best choice for simple synchronous tasks.

**Development Steps**

1.Add Spring WebFlux Dependency

2.Configure WebClient as Spring bean

3.Inject and use WebClient to call the REST API

4.Test using Postman Client

**Step 1: Define WebClient Bean**

java

@Configuration

public class WebClientConfig {

@Bean

public WebClient.Builder webClientBuilder() {

return WebClient.builder();

}

}

**Step 2: Inject & Use WebClient (Reactive)**

java

@Service

public class EmployeeService {

private final WebClient webClient;

@Autowired

public EmployeeService(WebClient.Builder webClientBuilder) {

this.webClient = webClientBuilder.baseUrl("http://localhost:8081").build();

}

public Mono<DepartmentDto> getDepartment(String departmentCode) {

return webClient.get()

.uri("/api/departments/{code}", departmentCode)

.retrieve()

.bodyToMono(DepartmentDto.class);

}

}

✅ **Asynchronous & Non-blocking** ✅ **Ideal for reactive applications** ❌ **Requires knowledge of reactive programming**

**WebClient Flow (Asynchronous & Reactive)**

* A service **non-blockingly** requests another microservice.
* The response is handled asynchronously (Reactively).
* Best for high-load systems requiring parallel processing.

**3)MICROSERVICES COMMUNICATION USING** **SPRING CLOUD OPENFEIGN**

OpenFeign simplifies microservices communication by allowing service calls with minimal configuration.

Pros: ✅ Simplifies microservice communication with minimal boilerplate. ✅ Supports load balancing when integrated with Spring Cloud Eureka.

Cons: ❌ Requires Spring Cloud dependencies. ❌ Can introduce network latency due to indirect service calls.

**Development Steps**

1.Add Spring cloud open feign Maven dependency to Employee-Service

2.Enable Feign Client using @EnableFeignClients

3.Create Feign API Client

4.Change the getEmployeeById method to use APIClient

5.Test using Postman Client

**Using Spring Cloud OpenFeign**

**Step 1: Enable Feign Clients**

java

@EnableFeignClients

@SpringBootApplication

public class EmployeeServiceApplication {

}

**Step 2: Create Feign Client Interface**

java

@FeignClient(name = "department-service", url = "http://localhost:8081/api/departments")

public interface DepartmentClient {

@GetMapping("/{departmentCode}")

DepartmentDto getDepartment(@PathVariable String departmentCode);

}

**Step 3: Inject & Use Feign Client**

java

@Service

public class EmployeeService {

private final DepartmentClient departmentClient;

@Autowired

public EmployeeService(DepartmentClient departmentClient) {

this.departmentClient = departmentClient;

}

public DepartmentDto getDepartment(String departmentCode) {

return departmentClient.getDepartment(departmentCode);

}

}

✅ **Declarative & Cleaner syntax** ✅ **Built-in support for Spring Cloud** ❌ **Requires Feign dependencies**

**OpenFeign Flow (Declarative HTTP Client)**

* The microservice simply calls another microservice using an **interface**.
* Spring automatically converts the interface method into an HTTP request.
* Best when working in a **Spring Cloud** ecosystem.

**SERVICE REGISTRY AND DISCOVERY USING SPRING CLOUD NETFLIX EUREKA**

Spring Cloud Netflix Eureka is a powerful tool for service registry and discovery in a microservices architecture. It enables services to dynamically register themselves and discover other services without hardcoding their locations.

**How Eureka Works**

1️.Eureka Server → Acts as the service registry where microservices register themselves. 2️.Eureka Client → Each microservice registers itself with the server and queries it to discover other services.

3️.Load Balancing → Eureka integrates with Ribbon or Spring Cloud LoadBalancer to distribute traffic efficiently.

**DEVELOPMENT STEPS**

1.Create Spring boot project as Microservice(service-registry)

2.Add @EnableEurekaServer annotation

3.Disable Eureka Server as Eureka Client(By default,each Eureka Server is also a Eureka client)

4.Launch Eureka Server(Demo)

5.Registering Department-Service Microservice as Eureka Client

6.Run department-service Eureka Client(Demo)

7.Registering Employee-service Microservice as Eureka Client

8.Run employee-service Eureka Client(Demo)

9.Multiple Instances of Department Service

**NOTE:**

**1)**@EnableEurekaServer is a Spring Cloud annotation that enables a Eureka server in a microservices architecture. When you use this annotation in a Spring Boot application, it turns the application into a service registry where other microservices can register themselves and discover one another dynamically.

Eureka, developed by Netflix, is widely used for service discovery in cloud-based architectures. A Eureka server keeps track of service instances and provides clients with updated information about service locations, allowing them to communicate efficiently even when services scale up or down.

**2)** @EnableDiscoveryClient is a Spring Cloud annotation that enables service discovery in a microservices architecture. It allows your application to register itself with a discovery server (such as Eureka, Consul, or Zookeeper), making it easier for other services to locate and communicate with it dynamically.

When you use @EnableDiscoveryClient, your Spring Boot application becomes a client that can both register itself and discover other services in the network. This is particularly useful in cloud environments where service instances may scale up or down and their locations may change.

<https://github.com/RameshMF/springboot-microservices/tree/main/v3/springboot-microservices>

**API GATEWAY USING SPRING CLOUD GATEWAY**

**A black and white text on a black background

AI-generated content may be incorrect.**

**DEVELOPMENT STEPS**

1.Create Spring boot project as Microservice(api-gateway)

2.Register API-Gateway as Eureka Client to Eureka Server(Service Registry)

3.Configuring API Gateway Routes and Test using Postman Client.

4.Using Spring Cloud Gateway to Automatically Create Routes.

<https://github.com/RameshMF/springboot-microservices/tree/main/v3/springboot-microservices>

**CIRCUIT BREAKER USING RESILIENCE4J IMPLEMENTATION**

The Circuit Breaker pattern helps prevent system failures by stopping requests to a failing service, allowing it time to recover. It’s widely used in microservices architectures to handle faults gracefully.

What Problems Does It Solve?

1. Avoid cascading failures: If a service is down, continuing to send requests could overload it further.
2. Improve system resilience: Temporary failures can be handled without crashing the entire application.
3. Enhance user experience: Provides fallback responses instead of long wait times or errors.
4. Optimize resource usage: Prevents unnecessary retries when a service is known to be unavailable.

How It Works:

* Closed State (Normal): Requests go through normally.
* Open State (Failure Detected): Requests are blocked, and an alternative response (fallback) is returned.
* Half-Open State (Recovery): A few requests are tested to see if the service has recovered.

First it will be in the closed state if the failure rate exceed threshold value it will be sent to the open state for 5secs and it will go to the half open and if it is successful it will go to the closed state or else if the failure rate is more than 3 it will go back to open state.

**DEVELOPMENT STEPS**

1.Add dependencies

2.Using @CircuitBreaker annotation to a method(it is calling external service)

3.Fallback method implementation

4.Add Circuit Breaker configuration in application.properties file

5.Restart employee-service and demo

https://github.com/RameshMF/springboot-microservices/tree/main/v3/springboot-microservices