Communication

Microservices Communication

Microservices communicate with each other to perform tasks and exchange data, and choosing the right communication method is crucial to the success of a microservices architecture. There are several common patterns and

techniques for microservices communication, depending on your needs and the interaction between services.

Key Considerations in Microservices Communication

- **Synchronous vs Asynchronous**: Do services need an immediate response (synchronous) or can they work independently and be notified later (asynchronous)?
- **Scalability and Fault Tolerance**: How well does the communication method handle failure or increased load?
- **Coupling**: How tightly should services be connected? The looser the coupling, the more independent the services can be.

Types of Microservices Communication

1. **Synchronous Communication** (Direct Requests/Response)
2. **Asynchronous Communication** (Event-Driven or Message-Based)

---

1. RestTemplate (Traditional Synchronous HTTP Client)

RestTemplate is the older, commonly used synchronous HTTP client for making HTTP requests in Spring applications. It allows you to perform CRUD operations against RESTful APIs in a straightforward manner.

Key Features:

- Synchronous: Blocks the thread until the response is received.
- Simplicity: Ideal for basic use cases (GET, POST, PUT, DELETE).
- Widely used: Part of Spring since early versions.

> Note: Spring recommends using WebClient in newer applications, as RestTemplate is slowly being deprecated. However, it is still widely used.

Example of Using `RestTemplate`:

1. **Add Dependency** (Spring Web Starter):

```
implementation 'org.springframework.boot:spring-boot-starter-web'
```

2. **Configuration**: You need to create a `RestTemplate` bean in your configuration file or service.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.client.RestTemplate;

@Configuration
public class AppConfig {
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

1. **GET Request Example** (Calling another microservice):

```
import org.springframework.web.client.RestTemplate;
import org.springframework.stereotype.Service;

@Service
public class CategoryService {
    private final RestTemplate restTemplate;

    public CategoryService(RestTemplate restTemplate) {
        this.restTemplate = restTemplate;
    }

    public Category getCategoryById(Long id) {
        String url = "http://category-service/api/categories/" + id;
```

```
        return restTemplate.getForObject(url, Category.class);
    }
}
```

- **GET Request**: `restTemplate.getForObject(url, Category.class);`
- **POST Request**: `restTemplate.postForObject(url, newCategory, Category.class);`
- **PUT Request**: `restTemplate.put(url, updatedCategory);`
- **DELETE Request**: `restTemplate.delete(url);`

Pros and Cons of `RestTemplate`:

- **Pros**:
  - Simple to use and well-established.
  - Works out-of-the-box for most use cases.
- **Cons**:
  - Synchronous and blocking (which can lead to scalability issues).
  - Not suitable for reactive or high-performance, non-blocking applications.

2. WebClient (Modern Reactive HTTP Client)

WebClient is a newer, more powerful, and flexible HTTP client introduced in **Spring 5**. It supports both **synchronous** and **asynchronous** operations and is designed for reactive programming using the **Project Reactor** model.

Key Features:

- **Reactive**: Supports non-blocking, asynchronous operations.
- **Functional and Fluent API**: Allows for more control and better performance.
- Supports **reactive streams** and **backpressure** (suitable for high-concurrency applications).

Example of Using `WebClient`:

1. **Add Dependency** (Spring WebFlux for reactive support):

```
implementation 'org.springframework.boot:spring-boot-starter-webflu
x'
```

2. **Configuration**: You can create a `WebClient` bean, or you can create an instance locally within your service.

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class WebClientConfig {
    @Bean
    public WebClient.Builder webClientBuilder() {
        return WebClient.builder();
    }
}
```

1. **GET Request Example** (Calling another microservice asynchronously):

```
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class CourseService {

    private final WebClient.Builder webClientBuilder;

    public CourseService(WebClient.Builder webClientBuilder) {
        this.webClientBuilder = webClientBuilder;
    }
```

```
    public Mono<Course> getCourseById(Long id) {
        return webClientBuilder.build()
            .get()
            .uri("http://course-service/api/courses/{id}", id)
            .retrieve()
            .bodyToMono(Course.class);
    }
}
```

- **GET Request**: `bodyToMono(Course.class)` converts the response to a **Mono** (reactive stream).
- **POST Request**:

```
webClientBuilder.build()
    .post()
    .uri("http://course-service/api/courses")
    .bodyValue(newCourse)
    .retrieve()
    .bodyToMono(Course.class);
```

1. **Blocking Request (if needed)**:

Even though **WebClient** is primarily designed for reactive operations, you can block the request and wait for the result in a synchronous manner:

```
Course course = webClientBuilder.build()
    .get()
    .uri("http://course-service/api/courses/{id}", id)
    .retrieve()
    .bodyToMono(Course.class)
    .block();  // This blocks until the response is received
```

Pros and Cons of `WebClient`:

- **Pros**:
  - Non-blocking and asynchronous by default.
  - More efficient in terms of resource usage (especially in high-concurrency applications).
  - Suitable for reactive programming.
- **Cons**:
  - More complex than `RestTemplate`.
  - May require additional libraries and configuration when not using reactive applications.

WebClient Examples

Here's how you can implement `WebClient` in a Spring Boot microservices project to create `GET`, `POST`, `PUT`, and `DELETE` requests that return either a single object or an array of objects.

1. Create WebClient Bean

First, configure a `WebClient` bean for reuse across the application:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.client.WebClient;

@Configuration
public class WebClientConfig {

    @Bean
    public WebClient webClient() {
        return WebClient.builder().build();
    }
}
```

2. GET Request - Single Object

Example of a `GET` request that retrieves a single object:

```
import org.springframework.stereotype.Service;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.core.publisher.Mono;

@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<MyObject> getObjectById(String id) {
        return webClient.get()
                .uri("http://localhost:8080/api/objects/{id}", id)
                .retrieve()
                .bodyToMono(MyObject.class);
    }
}
```

3. GET Request - Array/List of Objects

To handle a `GET` request returning an array of objects:

```
import reactor.core.publisher.Flux;

public class MyService {

    private final WebClient webClient;

    public MyService(WebClient webClient) {
```

```
        this.webClient = webClient;
    }

    public Flux<MyObject> getAllObjects() {
        return webClient.get()
                .uri("http://localhost:8080/api/objects")
                .retrieve()
                .bodyToFlux(MyObject.class);
    }
}
```

4. POST Request - Creating a New Object

For a `POST` request that sends a new object and retrieves a single object as a response:

```
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;

@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<MyObject> createObject(MyObject newObject) {
        return webClient.post()
                .uri("http://localhost:8080/api/objects")
                .body(Mono.just(newObject), MyObject.class)
                .retrieve()
                .bodyToMono(MyObject.class);
    }
```

```
}
```

5. PUT Request - Updating an Object

For a `PUT` request to update an existing object:

```
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;

@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<MyObject> updateObject(String id, MyObject updatedObje
ct) {
        return webClient.put()
                .uri("http://localhost:8080/api/objects/{id}", id)
                .body(Mono.just(updatedObject), MyObject.class)
                .retrieve()
                .bodyToMono(MyObject.class);
    }
}
```

6. DELETE Request - Deleting an Object

For a `DELETE` request that deletes an object by ID:

```
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;
```

```
@Service
public class MyService {

    private final WebClient webClient;

    public MyService(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<Void> deleteObject(String id) {
        return webClient.delete()
                .uri("http://localhost:8080/api/objects/{id}", id)
                .retrieve()
                .bodyToMono(Void.class);
    }
}
```

Example POJO

```
public class MyObject {
    private String id;
    private String name;

    // Getters and Setters
}
```

Explanation:

1. **WebClient Setup:** We configure a `WebClient` bean using the `WebClientConfig` class.
2. **GET (Single Object):** Fetch a single object by its ID using `.retrieve().bodyToMono()`.
3. **GET (Array/List):** Fetch an array or list of objects using `.retrieve().bodyToFlux()`.

4. **POST:** Create a new object and send it to the API using `.post()` and send the body using `.body()`.
5. **PUT:** Update an existing object by its ID using `.put()` and send the updated body.
6. **DELETE:** Delete an object by its ID with `.delete()` and handle the response using `Void.class`.

Blocking Reqeust

Example: Blocking `GET`, `POST`, `PUT`, and `DELETE` Requests with `WebClient`

1. **Blocking `GET` Request**:

```
java
Copy code
public MyObject getObjectByIdBlocking(String id) {
    return webClient.get()
        .uri("http://localhost:8080/api/objects/{id}", id)
        .retrieve()
        .bodyToMono(MyObject.class)
        .block();  // Blocking call
}
```

2. **Blocking `GET` Request for a List of Objects**:

```
java
Copy code
public List<MyObject> getAllObjectsBlocking() {
    return webClient.get()
        .uri("http://localhost:8080/api/objects")
        .retrieve()
        .bodyToFlux(MyObject.class)
        .collectList()
        .block();  // Blocking call
```

```
}
```

3. **Blocking `POST` Request**:

```
java
Copy code
public MyObject createObjectBlocking(MyObject newObject) {
    return webClient.post()
        .uri("http://localhost:8080/api/objects")
        .bodyValue(newObject)
        .retrieve()
        .bodyToMono(MyObject.class)
        .block();  // Blocking call
}
```

4. **Blocking `PUT` Request**:

```
java
Copy code
public MyObject updateObjectBlocking(String id, MyObject updatedObj
ect) {
    return webClient.put()
        .uri("http://localhost:8080/api/objects/{id}", id)
        .bodyValue(updatedObject)
        .retrieve()
        .bodyToMono(MyObject.class)
        .block();  // Blocking call
}
```

5. **Blocking `DELETE` Request**:

```
java
Copy code
public void deleteObjectBlocking(String id) {
    webClient.delete()
```

```
        .uri("http://localhost:8080/api/objects/{id}", id)
        .retrieve()
        .bodyToMono(Void.class)
        .block();  // Blocking call
}
```

Key Points:

- `.block()`: This method makes the request blocking and returns the result synchronously. For example, in the `getObjectByIdBlocking` method, the thread waits until the response is received, and then it returns the `MyObject`.
- `.collectList().block()`: In the case of a `Flux`, you can use `.collectList()` to convert the `Flux` into a `List`, and then block on that.

When to Use Blocking Requests

- **Synchronous environments**: If you are working in a non-reactive (synchronous) environment, using `.block()` can be necessary to convert reactive `Mono` or `Flux` responses into blocking, synchronous calls.
- **Testing**: You may use blocking calls in unit tests when you don't need the reactive nature of the responses.

Why Avoid Blocking in Reactive Applications?

In a fully reactive environment (like Spring WebFlux), using blocking calls can degrade performance. WebClient is designed to work in non-blocking, reactive pipelines, so excessive blocking can lead to thread exhaustion and reduced scalability.

In such cases, you should aim to use the reactive approach (`Mono`, `Flux`) and avoid blocking unless absolutely necessary.

3. FeignClient (Declarative REST Client)

Feign is a declarative REST client from **Netflix** that allows you to write REST clients using annotations. Feign makes calling REST APIs easier by eliminating boilerplate code for making HTTP requests.

Key Features:

- **Declarative API**: You define interfaces, and Feign automatically generates implementations.
- **Built-in Load Balancing** (with Spring Cloud).
- **Integrated with Eureka/Service Discovery** for easy client-side load balancing and service discovery.

Example of Using FeignClient:

1. **Add Dependencies**:

```
implementation 'org.springframework.cloud:spring-cloud-starter-openfeign'
```

2. **Enable Feign Clients**: Add the `@EnableFeignClients` annotation to your main application class.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.openfeign.EnableFeignClients;

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

1. **Define Feign Client Interface**:

```
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
```

```
@FeignClient(name = "course-service", url = "http://course-service/api")
public interface CourseClient {

    @GetMapping("/courses/{id}")
    Course getCourseById(@PathVariable("id") Long id);
}
```

- **FeignClient Annotation**: `@FeignClient(name = "course-service", url = "http://course-service/api")` indicates that this interface is a Feign client, and Feign will handle the implementation.

1. **Use the Feign Client in a Service**:

```
import org.springframework.stereotype.Service;

@Service
public class VideoService {

    private final CourseClient courseClient;

    public VideoService(CourseClient courseClient) {
        this.courseClient = courseClient;
    }

    public Course getCourseForVideo(Long courseId) {
        return courseClient.getCourseById(courseId);
    }
}
```

Pros and Cons of FeignClient:

- **Pros**:
    - Very simple and declarative syntax.

- Reduces boilerplate code (no need to handle request building or parsing responses).
- Integrates well with Spring Cloud features (like service discovery and load balancing).

- **Cons**:
  - More abstract, meaning less control over the request/response handling.
  - Can be harder to debug than more explicit approaches like `RestTemplate` or `WebClient`.

Example of Feign Client

```
package com.example.videoservice.client;

import com.example.videoservice.model.Course;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@FeignClient(name = "course-service", url = "http://localhost:8080/api/cours
public interface CourseClient {

    @GetMapping
    List<Course> getAllCourses();

    @GetMapping("/{id}")
    Course getCourseById(@PathVariable("id") Long id);

    @PostMapping
    Course createCourse(@RequestBody Course course);

    @PutMapping("/{id}")
    Course updateCourse(@PathVariable("id") Long id, @RequestBody Course
```

```
    @DeleteMapping("/{id}")
    void deleteCourse(@PathVariable("id") Long id);
}
```

Feature	**RestTemplate**	**WebClient**	**FeignClient**
Nature	Synchronous	Synchronous or Asynchronous	Synchronous
Usage	Low-level HTTP client	Reactive and non-blocking	Declarative HTTP client
Complexity	Simple	More complex, but flexible	Very simple and declarative
Reactive Support	No	Yes (designed for reactive)	No
Service Discovery	Requires manual setup	Requires manual setup	Easily integrates with Eureka
Load Balancing	Manual	Manual	Built-in with Spring Cloud
Ideal Use Case	Simple HTTP calls	Reactive, high-concurrency	Simplifying service communication
Blocking by Default	Yes	No	Yes