<https://www.baeldung.com/java-8-new-features>

<https://leetcode.com/studyplan/top-interview-150/>

<https://github.com/donnemartin/system-design-primer>

Microservices:

# What is Mono Flux?

“In reactive programming, Mono represents zero or one result, like fetching a single user, while Flux represents zero or many results, like fetching a list of users asynchronously.”

**Uses of Mono and Flux**

1. **Asynchronous & non-blocking**
   * They allow your microservices to handle many requests without waiting for each one to complete.
2. **Reactive Programming**
   * Helps build **responsive, resilient, and scalable applications**.
3. **Streaming Data**
   * Flux can handle **multiple events or records** as they arrive, useful for real-time feeds.
   * Mono is great for **single responses**, like fetching a user or creating a record.
4. **Error Handling & Composition**
   * You can easily handle errors, retries, and combine multiple async calls in a clean way.

# New microservices → WebClient

* Use WebClient when building **modern apps** that need **high performance** and **reactive/non-blocking calls**.

**Service-to-service calls → Feign Client**

* Use Feign when one microservice needs to call **another microservice**, because it’s **easy to use**: just define an interface and Spring handles the REST calls.

**Legacy apps → RestTemplate**

* Use RestTemplate in **older applications** that already use blocking calls. It works but is **deprecated for new apps**.

# Hystrix

* A **circuit breaker library** by Netflix.
* Handles service failures, timeouts, and fallbacks.
* Used to **stop cascading failures** in microservices.
* **Now in maintenance mode**, not actively developed.

# Resilience4j

* Modern **fault tolerance library** for Java.
* Supports **circuit breaker, retry, rate limiter, bulkhead, and time limiter**.
* Lightweight, functional, and works well with **Spring Boot 2+**.
* Recommended for **new microservices projects**.

**NOTE:** “Hystrix is the older Netflix circuit breaker library, while Resilience4j is a modern, lightweight alternative that provides circuit breaker, retry, and other fault-tolerance features for Spring Boot microservices.”

# How to Run Two Microservices at a Time – Key Points

**How to run two microservices at a time – Key Points:**

* **Use different ports** for each service in application.properties or application.yml.
* server.port=8081 # Service 1
* server.port=8082 # Service 2
* **Start each microservice independently** (e.g., via IDE, mvn spring-boot:run, or Docker).
* **Service Discovery (optional)**: Register services with **Eureka/Consul** for inter-service communication.
* **API Gateway (optional)**: Route requests to the correct service from a single entry point.
* **Check logs and endpoints** to ensure both services are running and accessible.

**@EnableEurekaServer:** Enables a Spring Boot application to act as a Eureka service registry.

**@EnableEurekaClient:** Registers the service with Eureka for discovery.

**@LoadBalanced:** Makes Rest Template or Web Client use client-side load balancing.

**@FeignClient (name = "service-name"):** Enables Feign to call other services using Eureka + load balancing.

**@EnableFeignClients:** Enables scanning for interfaces annotated with @FeignClient.

# How services communicate using Feign?

**What is Feign (Simple Words)**

* Feign is a **tool that helps one service call another service**.
* Instead of writing full HTTP code, you **just create an interface**.
* Feign **automatically sends the request** for you.

**✅ Real-Life Example (Easy to Imagine)**

You have 2 services:

* 👤 **User Service** — gives user details.
* 🛒 **Order Service** — needs user info to show order with user name.

**Without Feign:**

You would write this manually:

RestTemplate restTemplate = new RestTemplate();

User user = restTemplate.getForObject("http://USER-SERVICE/users/1", User.class);

**With Feign: You just write:**

@FeignClient(name = "user-service")

public interface UserClient {

@GetMapping("/users/{id}")

User getUser(@PathVariable("id") Long id);

**Now in your service, you just do:**

User user = userClient.getUser(1L);

And Feign will **automatically call the other service**, fetch the user, and give you the result

**✅ Step-by-Step Summary**

1. **You create a Java interface** using @FeignClient.
2. **You write a method** that matches the API of the other service.
3. **Feign sends the HTTP request** in the background.
4. **You just call a method like normal Java code.**
5. **Feign gives you the result** without extra work.

**NOTE:** Feign = Call other services **just by writing an interface** — no HTTP code needed.

# Synchronous Communication – Interview Examples

* **“Synchronous communication means one service waits for the other to respond.**

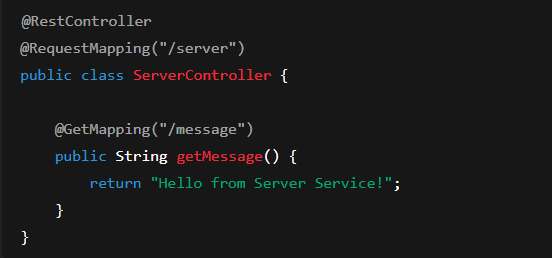
**1. User Login (Auth Service)**

* **Client:** Sends login credentials.
* **Server:** Validates credentials and returns token.
* **Why synchronous?** You can’t proceed without the response.

**2. Booking System (Flights, Hotels)**

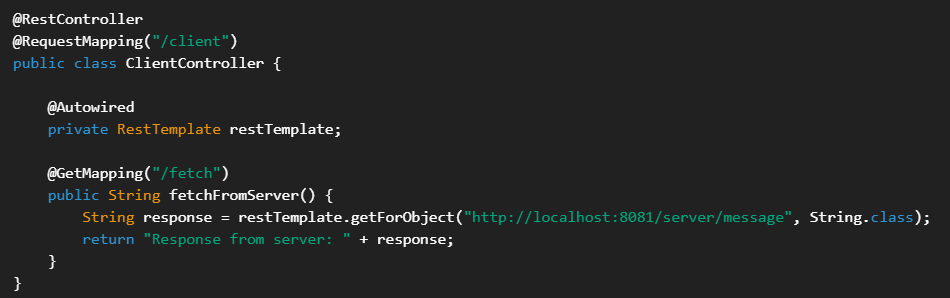
* **Booking Service → Payment Service → Confirmation**
* **Why synchronous?** Each step depends on the successful completion of the previous one.
* **Implementation of Synchronous**

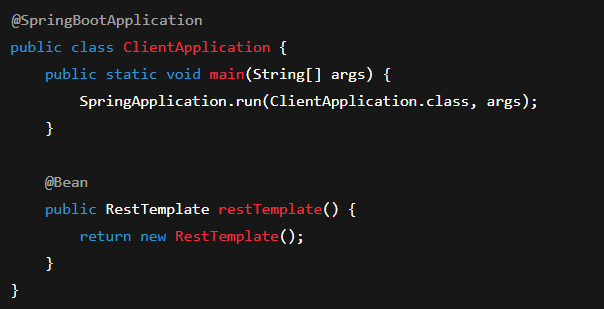
1. **Create Two Spring Boot Projects and run with Two different ports.**
2. **Configure application.properties : server.port=8081**
3. **Create ServerController class and annotated with @RestController@RequestMapping("/server")**



* **Configure application.properties : server.port=8082**

1. Create Client Controller class



Client Main Class  


Run Postman: run get mapping for fetching the data: **http://localhost:8080/client/fetch**

OUTPUT: Response from server: **Hello from Server Service!**

# Asynchronous Communication

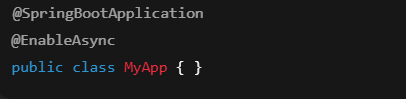
* **Services to send messages and continue processing without waiting.**
* If you sending an email or message but the receiver can respond anytime. So, you no need to wait; send and move on.

**✅ How to Implement Asynchronous in Your Application**

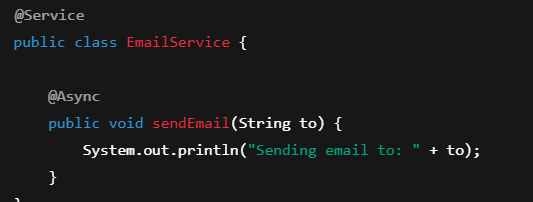
**🔧 Step-by-step (Single-Line Format for Interview):**

* **Enable async support** with @EnableAsync in your main class.
* **Use @Async** on a method you want to run asynchronously.
* **Call the method** — it will execute in a separate thread.

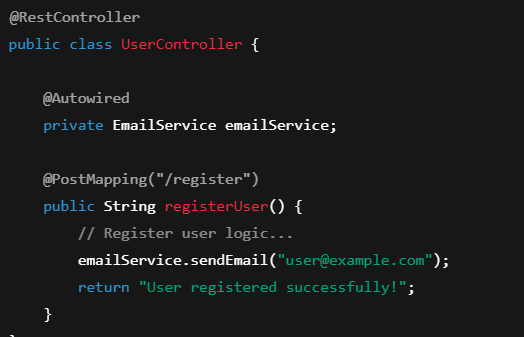
1. **Enable async in your main app:**



1. **Create a service with an async method:**



1. **Call the async method:**



**NOTE: “**I use @Async to make methods run in the background, like sending emails or logging. It improves performance by not blocking the main flow.”

# How to monitor our microservices using hystrix dash board

we need to add two dependencies in your Application

1. Hystrix dash board

2. Spring Actuator.

The purpose of Spring boot actuator is used to expose the endpoints.

like info, health, hystrix stream, beans and many other end points

Monitor our microservices can be done in two ways

* 1st way is creating separate project a dash board project. and add the two dependencies from here. you can monitor every other microservice.
* 2nd way is you can add the two dependencies one existing project and monitor the health and performance of that particular micro service.

Add property in your App. properties file.

* Management:endpoints:web:exposure:include: "\*" //get all the endpoint details.
* Annotated with @EnableHystrix, @EnableHystrixDashboard in your main class.
* Right click on project --> select spring --> select edit startrs : give required dependencies.

Ex:

* There are two services service A and service B where service A is trying to connecting service B, service B is available, Service A is return the result properly.
* If service B is not available. if we added to hystrix to service A, all the time will be getting the result of fallback method.
* If no of requests coming from service A to service B is more at one particular of time but
* then circuit breaks open and it a take a period of time to come back to normal scenario. the circuit will be closed and again try to call you will be getting the fallback method.
* If you want to monitor multiple microservices, in that case instead of hystrix you need to go for Turbine stream as of now we need to downgrade spring boot version 2.2.2 and cloud version SR1.

# Why Monitoring is Needed

* To **check health** and **performance** of microservices.
* To track **failures** and **timeouts** in API calls.
* To understand **response times**, **circuit breaker states**, and **traffic load**.
* To get **real-time insights** into microservice behaviours.

**Conclusion**

* If you have **one microservice** → Use Hystrix Dashboard inside it.
* If you have **multiple microservices** → Use a **separate monitoring dashboard** with **Turbine Stream**.
* Turbine Stream is **scalable, real-time, and event-driven**, making it the **preferred approach** for modern architectures.

# Fault tolerance

I am having three services A,B and C

A connecting to B, B connecting to C in this case if perticular Micro services C is

Down it will throw an Exception to B,B should not casecade the failure passed to the A,

insted you should have mechanism to handle the Exception and send the fallback message to A

This is how fault tolerence is taken care incase of microservice.

Note: instead of throwing exception, we can send the fallback method, with some dummy result.

# 🔍 Service Discovery Pattern in Microservices

* In a microservices architecture, different services often need to find and communicate with each other.
* The Service Discovery Pattern helps services automatically locate each other without hardcoding network addresses like IPs or URLs.
* Instead of manually updating service locations, a Service Registry keeps track of where each service is running.
* When a service needs to call another, it queries the registry to find the correct address.

# Externalizing Microservices Configuration in Spring Boot

## Step 1: Set Up a Centralized Config Server

- Add Dependency → Include 'spring-cloud-config-server' in pom.xml.  
- Enable Config Server → Use @EnableConfigServer in the main class.  
- Configure Port & Git Repo → In application.properties or application.yml:  
 server.port=8888  
 spring.cloud.config.server.git.uri=https://github.com/company/config-repo  
- The Config Server fetches configs from the Git repo and serves them to microservices.

## Step 2: Create a Central Git Repository

- Create a config-repo to store all microservice config files.  
- Each microservice has its own config file:  
 vehicle-service.yml  
 payment-service.yml  
 user-service.yml  
- The Config Server reads these files and supplies them at runtime.

## Step 3: Connect Microservices to Config Server

- Add 'spring-cloud-starter-config' dependency to each microservice.  
- Create bootstrap.yml in each service:  
 spring:  
 application:  
 name: vehicle-service  
 config:  
 import: "optional:configserver:http://localhost:8888"  
- Important → spring.application.name must match the config file name in Git.

## Step 4: Use Profiles for Environment-Specific Configs

- In Git repo, create files like:  
 vehicle-service-dev.yml  
 vehicle-service-qa.yml  
 vehicle-service-prod.yml  
- Activate a profile at runtime:  
 java -jar vehicle-service.jar --spring.profiles.active=prod  
- Config Server provides the correct environment-specific configs.

## Step 5: Enable Dynamic Config Refresh

- Add Spring Boot Actuator dependency.  
- Use @RefreshScope on beans to auto-reload properties.  
- Refresh configs without restarting:  
 POST http://localhost:8081/actuator/refresh

* **Create a Config Server** – Add spring-cloud-config-server, enable with @EnableConfigServer, set port and Git repo for configs.
* **Central Git Repository** – Store all microservice configuration files (YAML/properties) in one Git repo.
* **Connect Microservices** – Add spring-cloud-starter-config, configure bootstrap.yml to fetch configs from the server.
* **Use Spring Profiles** – Maintain environment-specific files (dev, QA, prod) and activate the required profile at runtime.
* **Refresh Configs Dynamically** – Use Spring Boot Actuator and @RefreshScope to update configurations without restarting services.

# Difference B/W Rest Template and FeignClient

**Using Rest Template**

* You need to **manually build URLs** for each request.
* You must **handle different HTTP methods** (GET, POST, PUT, DELETE, etc.) manually.
* You need to **configure headers** for every request.
* **Serialization and deserialization** often need manual handling.
* Leads to **a lot of boilerplate code**.

**Using FeignClient**

* **No manual URL construction** – just define the endpoint in an interface.
* **No repetitive request setup** – Feign handles headers, methods, and data automatically.
* **Automatic serialization and deserialization**.
* Makes **inter-service communication simpler, cleaner, and more declarative**.

# How many ways to handle exceptions in microservices

|  |  |  |
| --- | --- | --- |
| **Approach** | **Scope** | **Best For** |
| **@ControllerAdvice + @ExceptionHandler** | Single Service | Centralized exception handling |
| **ResponseStatusException** | Single Service | Quick mapping of exceptions |
| **Custom Exceptions** | Single Service | Better readability |
| **ProblemDetail (Spring Boot 3+)** | Single Service | Standardized error responses |
| **Feign ErrorDecoder** | Inter-Service | Handling API errors |
| **Circuit Breaker / Fallback** | Inter-Service | Fault tolerance |
| **API Gateway Error Handling** | Multi-Service | Centralized error management |
| **Sleuth + Zipkin + ELK** | Multi-Service | Exception tracking & debugging |

# How do you handle service failures in production?

**Situation:**

* User is trying to pay for an order.
* The **Payment Service** is down or very slow.

**How to Handle It (Step by Step)**

1. **Stop it from breaking everything**
   * Don’t let the whole website crash.
   * Use a **circuit breaker**: if the payment service fails, temporarily stop calling it.
2. **Keep the system usable**
   * Let the user **save items in the cart** or **continue browsing**.
   * Show a message like: *“Payment service is temporarily unavailable. Please try again later.”*
   * Optionally, you can **retry automatically** a couple of times.
3. **Fix the problem quickly**
   * Automatically **restart the Payment Service** if it crashed.
   * **Send alerts** to the team: “Payment service is down!”
   * Check **logs and metrics** to find the cause (server overload, network issue, etc.)

# How do you handle service-to-service security in microservices?

* service-to-service communication using authentication and authorization. Typically, services send JWT tokens or use mutual TLS to prove their identity.
* I also use HTTPS to encrypt data, and optionally a service mesh like Istio for automatic security between services.

**API Gateway / Service Mesh (Optional but Recommended)**

* **API Gateway:** Acts as a secure entry point for all services.
* **Service Mesh (Istio, Linkerd):** Handles **authentication, authorization, and encryption automatically** between services.

# How do you manage API versioning across microservices?

I manage API versioning using URI, header, or query parameter versions. For example, /api/v1/orders is the old version, and /api/v2/orders is the new version. I keep backward compatibility, deprecate old versions gracefully, and document all versions to avoid breaking clients.

# How do you handle circular dependencies between microservices?

* Happens when **Service A depends on B** and **B depends on A**.
* Causes deadlocks, deployment issues, and tight coupling.

**How to Handle It**

1. **Refactor Services**
   * Move shared logic to a **new service**.
   * Example: A and B need user data → create **User Service**.
2. **Use Asynchronous Communication**
   * Replace direct calls with **events**.
   * Example: A publishes OrderCreated, B processes it later without calling A back.

# When would you use REST API vs gRPC for inter-service communication?

"I use REST APIs when services need to be public-facing or flexible, like serving web/mobile clients. I use gRPC for internal, high-performance, or streaming communication between microservices because it’s faster and strongly typed."

# A service is responding slowly under load. How do you diagnose and optimize it?

"If a service is slow under load, I first check logs, metrics, and profiling to identify the bottleneck. I look at database queries, downstream services, and network issues. Then I optimize by caching frequently used data, scaling the service horizontally, moving heavy tasks to asynchronous processing, and using circuit breakers to prevent blocking calls.

# How do you implement transaction rollback if one microservice in a workflow fails?

Here’s a **simple and clear explanation** of how to handle **transaction rollback in microservices**:

**Problem**

* In microservices, a business workflow often spans multiple services.
* Traditional ACID transactions don’t work across services.
* Example: Placing an order involves **Order Service**, **Payment Service**, and **Inventory Service**. If Payment fails, the order and inventory updates should be rolled back.

**Solution: Use SAGA Pattern**

**SAGA** manages distributed transactions with **local transactions + compensating actions**.

**1. Choreography (Event-based)**

* Each service performs its local transaction and publishes an event.
* If a service fails, other services execute **compensating actions** to undo their work.

**Example:**

1. Order Service creates order → publishes OrderCreated.
2. Payment Service processes payment → publishes PaymentProcessed.
3. Inventory Service updates stock → fails.
4. Payment Service performs **refund**, Order Service cancels order.

**2. Orchestration (Coordinator-based)**

* A central orchestrator **tells services what to do** and monitors success/failure.
* If one step fails, the orchestrator triggers compensating actions.

**Example:**

1. Orchestrator: Create order → Success
2. Orchestrator: Process payment → Success
3. Orchestrator: Update inventory → Fails
4. Orchestrator: Trigger refund → Cancel order

**Key Points**

* **No distributed ACID transactions**; use local transactions + compensating actions.
* Ensure **idempotency** for compensating actions to avoid double rollback.
* Works well with **event-driven or orchestrator-based workflows**.

# If one microservice is failing and affecting others. How to investigate?

**1. Investigate**

* Check **logs** for errors.
* Monitor **CPU, memory, and response times**.
* Check **dependencies** (databases, other services).
* Use **distributed tracing** to find where requests fail.
* Review **recent deployments** for potential causes.

**2. Fix**

* **Restart** the service if crashed.
* Use **circuit breakers/fallbacks** to isolate it.
* **Scale** if overloaded.
* Apply **bug fixes or config updates**.
* Do a **post-mortem** to prevent future issues.

# How would you migrate a monolith to microservices gradually?

To migrate a monolith gradually, I first analyze and identify modules to split. I extract small, independent modules into microservices, use an API gateway or the strangler pattern to route traffic gradually, maintain data consistency with SAGA or database per service, and monitor everything to ensure stability."

# How do you handle version conflicts when multiple teams update different services?

**Problem:**

* Multiple teams work independently on different services.
* Version conflicts occur when **changes in one service break others** or **API changes are incompatible**.

**How to Handle It**

1. **Semantic Versioning** – MAJOR for breaking, MINOR for features, PATCH for fixes.
2. **API Contracts** – Define clear contracts (Swagger/OpenAPI) and use contract testing.
3. **Backward Compatibility** – Make non-breaking changes and deprecate old APIs gradually.
4. **CI/CD & Version Registry** – Independently build/deploy services and track versions.
5. **Team Communication** – Share docs, release notes, and coordinate major changes.