**Microservices Interview Questions**

Contents

[1. Explain Microservice Communication - synchronous and asynchronous 1](#_Toc2621774)

# Explain Microservice Communication - synchronous and asynchronous

In a microservices architecture, services often need to talk to each other to complete a business workflow.  
 This communication can be either **synchronous** or **asynchronous**, depending on the use case.

### **1. Synchronous Communication**

Synchronous means that **one service calls another and waits for a response** — just like a function call or an API call.

**Example:**

A OrderService calls PaymentService via a REST API. It waits for confirmation before moving forward.

️ **Common Protocols:**

* HTTP REST (using RestTemplate, WebClient, Feign)
* gRPC
* GraphQL

**Pros:**

* Simple to implement
* Easy to debug
* Immediate feedback

**Cons:**

* Tightly coupled (if one service is down, the other is affected)
* Not resilient to failures
* Causes cascading latency

### **2. Asynchronous Communication**

In asynchronous communication, the calling service **doesn’t wait for a response**.  
 It sends a **message or event** and moves on. Other services consume that message independently.

**Example:**

OrderService sends an event like OrderPlaced to a Kafka topic.  
 InventoryService and NotificationService both listen and react independently.

**Common Tools:**

* Kafka
* RabbitMQ
* AWS SNS/SQS
* ActiveMQ

**Pros:**

* Loose coupling (services don’t depend on each other’s availability)
* Scales better
* Good for event-driven architectures

**Cons:**

* Harder to trace/debug
* More complex error handling
* No immediate response (eventual consistency)

### **Summary:**

So, to summarize:

* **Synchronous** is best when you need an immediate response (e.g., user login).
* **Asynchronous** is better when you're doing background tasks, need loose coupling, or want better fault tolerance — like sending emails, updating logs, or processing payments.

In most real systems, we use a **mix of both**.  
 For example, we may use a **synchronous REST call to verify payment and** then **fire an asynchronous Kafka event** to update inventory or send an email.

# What is an API Gateway?

An **API Gateway** is a **single-entry point** for all client requests going to your microservices.  
 Instead of clients calling each service directly, they talk to the gateway, and it handles routing, security, and traffic management.

### **Analogy:**

Think of an API Gateway like a **reception desk** at a hotel.  
 Guests (clients) don’t go to individual rooms (microservices).  
 They go to the reception (gateway), which directs them to the right place.

## **Responsibilities of an API Gateway**

1. **Request Routing**

It forwards incoming requests to the correct backend service based on URL or method.

1. **Load Balancing**

Distributes traffic across multiple instances of a microservice.

1. **Authentication & Authorization**

Verifies access tokens (JWT, OAuth2) before letting the request through.

1. **Rate Limiting / Throttling**

Prevents abuse by limiting how many requests a client can make per second/minute.

1. **Logging & Monitoring**

Captures metrics, logs, and errors at the gateway level.

1. **Response Aggregation** *(Optional)*

Combines responses from multiple services into a single response for the client.

1. **CORS / HTTPS / Header Manipulation**

Handles client-specific requirements like headers, protocol enforcement, etc.

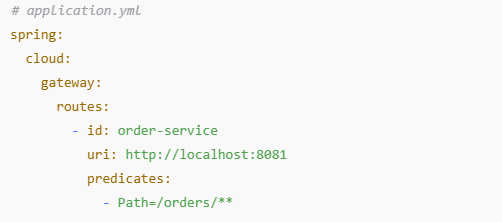
## **Popular API Gateway Tools**

|  |  |
| --- | --- |
| **Tool** | **Description** |
| **Spring Cloud Gateway** | Lightweight, reactive, built into Spring ecosystem |
| **Kong** | Open-source, plugin-based, Lua-powered |
| **NGINX** | High-performance reverse proxy often used as a gateway |
| **AWS API Gateway** | Managed gateway for serverless & microservices |
| **Zuul** (Netflix) | Legacy Spring-based gateway (now replaced by Spring Cloud Gateway) |

## **How It Works**

A client makes a request to /orders/123.  
 The API Gateway receives it, checks auth, rate limits it, and routes it to the **OrderService**.  
 The response flows back the same way — through the gateway to the client.

## **In Spring Boot (Spring Cloud Gateway Example):**



+-------------------+

| Client Apps |

| (Web, Mobile, etc)|

+--------+----------+

|

▼

+-----------------------+

| API Gateway | ◀── Authentication, Routing, etc.

+-----------------------+

| | |

┌────────┘ | └────────┐

▼ ▼ ▼

+---------------+ +---------------+ +----------------+

| Order Service | | Payment Service| | Inventory Svc |

+---------------+ +---------------+ +----------------+

(Internal Microservices - Not exposed directly to clients)

Optional:

+--------------------+

| Kafka / Messaging |

| (Async Communication) |

+--------------------+

### **Key Points:**

* Clients send requests to **one endpoint** — the **API Gateway**.
* The Gateway **routes** requests to the appropriate microservice (e.g., /orders/\*\* → OrderService).
* It may **authenticate**, **log**, **rate-limit**, or **transform** the request before forwarding it.
* Gateway keeps **services hidden** from external access (decoupling and security).
* Supports both **synchronous (HTTP)** and **asynchronous (Kafka, RabbitMQ)** communication.

## **Summary:**

An **API Gateway** is a front-facing service that handles routing, security, and monitoring for your microservices.  
 It simplifies the client interaction, enforces policies centrally, and decouples services from direct public exposure.