Microservices Tutorial

Kind of Applications

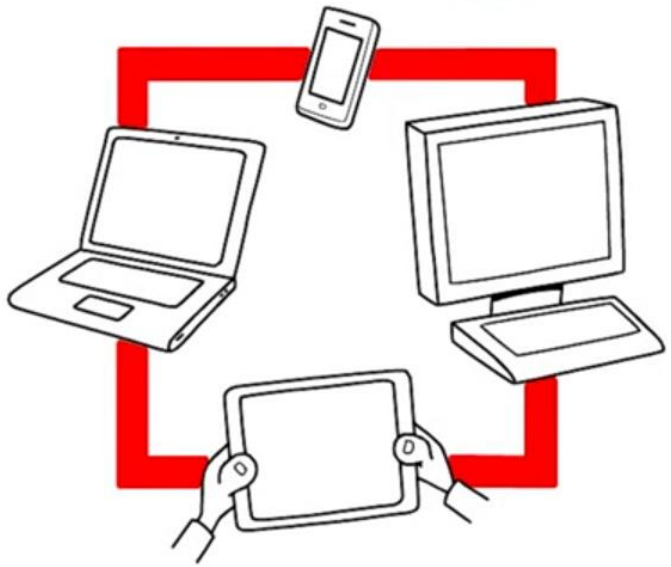
Applications can be classified as:

- **Monolithic Applications**: All features (UI, business logic, data access) are in one single unit.
- **Service-Oriented Architecture (SOA)** Applications: Built using services that communicate over a network.
- **Microservices Applications**: Built as small, independent services that communicate via APIs.

What is SOAP API

Simple Object Access Protocol

used for exchanging structured data between different applications



regardless of the platform or programming language used

SOAP messaging protocol

including text, numbers, dates, and binary data

HTTP, SMTP, and FTP

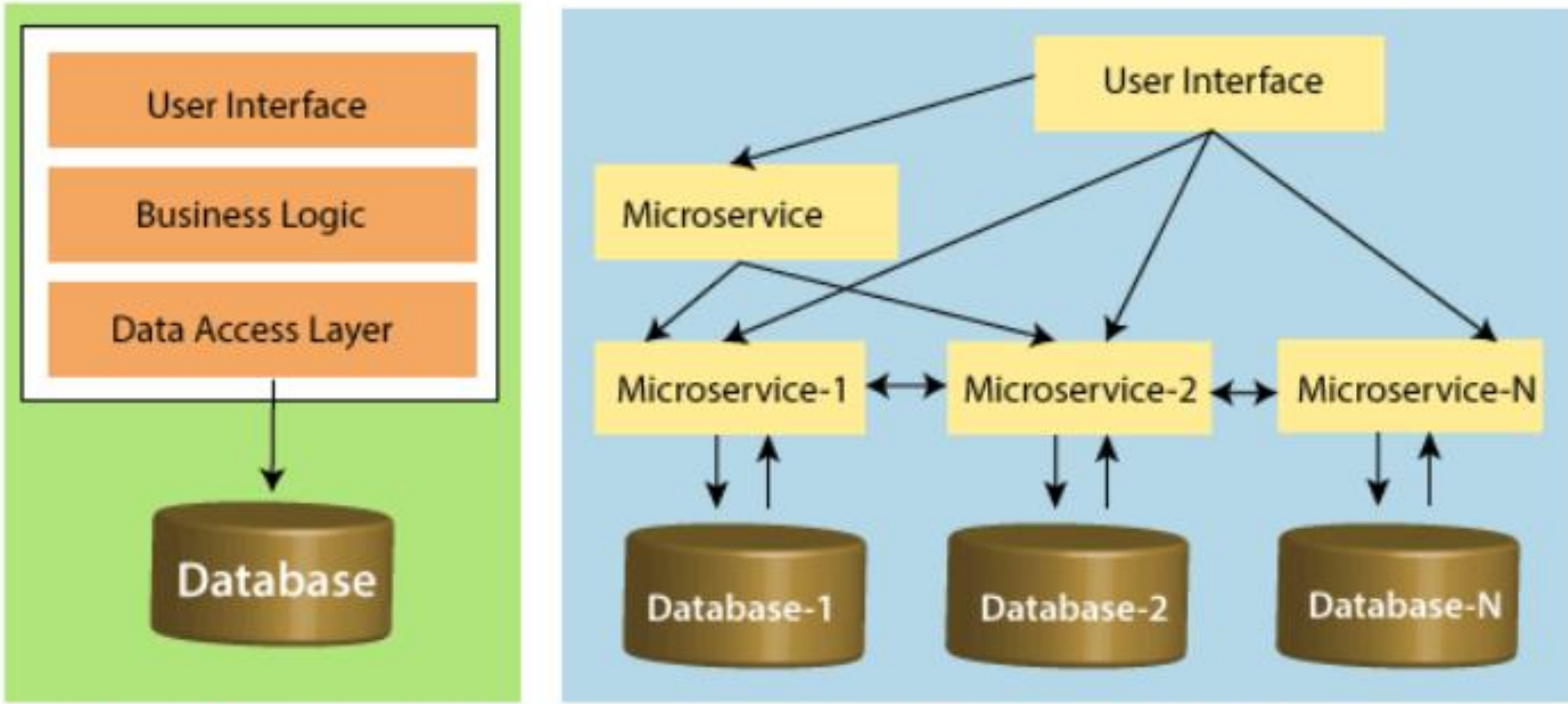
✅ Quiz App Architecture Comparison (Easy Language)

Feature / Layer	Monolithic	SOA (Service-Oriented Architecture)	Microservices
🧱 **Structure**	Everything (Quiz, Questions, Users) is built together as **one single app**.	Quiz, Questions, Users are **separate parts**, but still **somewhat connected** (e.g., using same database).	Quiz, Questions, Users are **fully separate services** that work **independently**.
🧠 **How the App is Split**	Just **one app**: `quiz-app` that does everything.	Multiple services: `quiz-service`, `question-service`, etc. They **talk to each other** using special messages (SOAP).	Multiple services: `quiz-service`, `question-service`, etc. They talk using **simple APIs (REST)**.
🗂️ **Code Organization**	One big project/folder that has everything.	Each feature has its own folder/project.	Each service is like its **own mini app** in a separate folder or Git repo.
🗃️ **Database**	One big database for all features.	Services might share the same database or use different parts of it.	Each service has its **own database** and doesn't touch others.

🔗 **How They Talk**	Functions inside the app. (like calling a method in the same file)	Services talk using **SOAP** messages and **central system (ESB)**.	Services talk using **REST APIs** or messages (e.g., JSON over HTTP).
🛠️ **Technology Used**	Only one tech/language (e.g., Java for all features).	Mostly same tech across services.	Each service can use a different language or framework (e.g., Python for questions, Java for quiz).
📈 **Scaling**	You can only scale the whole app together.	You can scale parts separately but it's hard.	You can easily scale **only the needed service** (e.g., only `question-service`).

Microservice Architecture is a Service Oriented Architecture. In the microservice architecture, there are a large number of **microservices**. By combining all the microservices, it constructs a big service. In the microservice architecture, all the services communicate with each other.

"Microservices are the small services that work together"



Monolithic Architecture

Microservice Architecture

Principles of Microservices

There are the following principles of Microservices:

- Single Responsibility principle
- Modelled around business domain
- Isolate Failure
- Infrastructure automation
- Deploy independently

Single Responsibility Principle

The single responsibility principle states that a class or a module in a program should have only one responsibility. Any microservice cannot serve more than one responsibility, at a time.

Modeled around business domain

Microservice never restrict itself from accepting appropriate technology stack or database. The stack or database is most suitable for solving the business purpose.

Isolated Failure

The large application can remain mostly unaffected by the failure of a single module. It is possible that a service can fail at any time. So, it is important to detect failure quickly, if possible, automatically restore failure.

Infrastructure Automation

The infrastructure automation is the process of scripting environments. With the help of scripting environment, we can apply the same configuration to a single node or thousands of nodes. It is also known as configuration management, scripted infrastructures, and system configuration management.

Deploy independently

Microservices are platform agnostic. It means we can design and deploy them independently without affecting the other services.

Advantages of Microservices

1. **Independent Deployment**
   - Each service can be developed, deployed, and updated independently.
2. **Scalability**
   - Services can be scaled individually based on demand.
3. **Technology Flexibility**
   - Teams can use different programming languages or frameworks for different services.
4. **Faster Development**
   - Smaller teams can work on different services in parallel, speeding up delivery.
5. **Improved Fault Isolation**
   - If one service fails, it doesn't crash the whole system.
6. **Easy Maintenance**
   - Small codebases are easier to understand, test, and maintain.
7. **Continuous Delivery & Deployment**
   - Supports DevOps practices like CI/CD for faster releases.

🔨 8. Microservices Frameworks

Here are popular frameworks used for building microservices:

Language	Frameworks
Java	Spring Boot, Micronaut, Quarkus
Node.js	Express.js, NestJS
Python	FastAPI, Flask
.NET	ASP.NET Core
Go	Go Kit, Gin
Container Orchestration	Docker, Kubernetes

What is Netflix OSS?

Netflix OSS (Open Source Software) is a set of tools created by Netflix to build **reliable, scalable, cloud-native microservices**.

These tools help with:

- Service discovery
- Load balancing
- Circuit breaking
- API gateway
- Monitoring

🧰 Main Netflix OSS Tools:

Tool	Purpose
Eureka	Service Registry (for Service Discovery)
Ribbon	Client-side Load Balancer
Hystrix	Circuit Breaker (fallback when a service fails)
Zuul	API Gateway

🌩️ 2. What is Spring Cloud?

✅ In Simple Words:

Spring Cloud is a collection of tools that make it **easy to build microservices** using the Spring framework.

It provides:

- Service discovery
- Load balancing
- API Gateway
- Config Server
- Circuit Breaker
- Declarative REST clients

> 💬 Think of Spring Cloud as a **toolbox to connect, manage, and deploy microservices** efficiently.

🔍 3. Service Discovery with Spring Cloud Netflix Eureka

✅ What is Service Discovery?

In microservices, services must **find and talk to each other**. You don't hard-code URLs. Instead, use a **Service Registry**.

Eureka helps services:

- **Register themselves**
- **Discover other services**

🧠 Example (Quiz App):

- `quiz-service` registers as `QUIZ-SERVICE`
- `question-service` registers as `QUESTION-SERVICE`
- Both register with **Eureka Server**

⚖️ 4. Client-Side Load Balancing with Spring Cloud Load Balancer

✅ What is Client-Side Load Balancing?

When multiple instances of a service are running (e.g., 3 `QUESTION-SERVICE`), the client **decides which one to call**.

Spring Cloud LoadBalancer (replaces Netflix Ribbon) picks one instance using **round-robin** or other strategies.

> 💬 Load balancing is done by the **caller**, not by an external proxy.

🧠 Example:

`quiz-service` needs questions → it asks Eureka for `QUESTION-SERVICE` → picks one instance using LoadBalancer.

🗂️ Eureka Components:

1. **Eureka Server** – Central registry
2. **Eureka Clients** – Microservices that register themselves

🛠️ Simple Code Snippet:

java

```java
@EnableEurekaServer  // In Eureka Server main class
@SpringBootApplication
public class EurekaServerApplication { }

@EnableEurekaClient  // In each microservice
@SpringBootApplication
public class QuizServiceApplication { }
```

⚙️ 5. Implementing Microservices with Eureka and Load Balancer

🔧 Microservices You Create:

- `eureka-server` (service registry)
- `quiz-service` (Eureka client)
- `question-service` (Eureka client)

💬 How It Works:

1. All services **register with Eureka**
2. `quiz-service` uses **Spring Cloud LoadBalancer** to call `question-service`
3. LoadBalancer picks the best instance

🔁 Dependencies:

xml

```xml
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

📦 6. OpenFeign – Declarative RestClient

✅ What is OpenFeign?

OpenFeign is a way to call other services using **just an interface** — no need to write boilerplate code like `RestTemplate`.

🧠 Example:

You want `quiz-service` to call `question-service`.

Instead of writing this:

```java
RestTemplate restTemplate = new RestTemplate();
Question q = restTemplate.getForObject("http://QUESTION-SERVICE/api/questions", Question.class);
```

```java
@FeignClient(name = "QUESTION-SERVICE")
public interface QuestionClient {
    @GetMapping("/api/questions")
    List<Question> getAllQuestions();
}
```

```java
@Autowired
private QuestionClient questionClient;

List<Question> questions = questionClient.getAllQuestions();
```

✅ What is Eureka Server?

Eureka Server is a **Service Registry** from Netflix, used in **Spring Cloud Netflix**.

In a microservices environment:

- Services need to discover and communicate with each other.
- Eureka Server helps them do that dynamically.

It acts as a **phonebook** where services **register themselves** and **lookup** others by name.

🔍 Why Do We Need Eureka?

In traditional apps:

- Services had **fixed IPs/ports** or were manually configured.
- This doesn't scale in cloud environments where services may **scale up/down**, move, or crash and restart dynamically.

In microservices:

- **Hardcoding URLs/IPs** is **fragile**.
- We need **dynamic service discovery**.

➡️ **Eureka solves that.**

How Eureka Works

1. Eureka Server:

- A central registry (like DNS for microservices).
- Runs on a separate port (e.g., `http://localhost:8761`).

2. Eureka Clients (your services):

- Register themselves (`spring.application.name=question-service`)
- Periodically send heartbeats to the server.
- Can also **discover other services** via Eureka.

✅ Advantages of Using Eureka

Benefit	Description
✅ **Dynamic Discovery**	Services can find each other without hardcoded URLs.
✅ **Self-healing**	If a service goes down, Eureka stops routing to it.
✅ **Built-in Load Balancing**	Using `lb://service-name`, requests are auto-balanced.
✅ **Easier Scaling**	Spin up multiple instances, all auto-registered.
✅ **Failover Support**	You can have multiple Eureka servers (clusters).

🔁 Alternatives to Eureka

1. Consul (by HashiCorp)

- Service registry + key/value config store + health checks
- Supports DNS-based service discovery
- Widely used in production (cloud-native)

2. Zookeeper

- Strong consistency
- Often used with Apache Kafka, not preferred for microservices discovery due to complexity

3. Nacos (by Alibaba)

- Supports configuration, naming, and service discovery
- More popular in Chinese tech ecosystems

4. Kubernetes Service Discovery

- If you're using Kubernetes, you often don't need Eureka
- Kubernetes handles service discovery natively (via DNS)

```java
@EnableEurekaServer
@SpringBootApplication
public class DiscoveryServicesApplication {

    public static void main(String[] args) {
        SpringApplication.run(DiscoveryServicesApplication.class, args);
    }

}
```

DiscoveryServicesApplication.java | application.properties

```
spring.application.name=DiscoveryServices

server.port=8761
eureka.instance.hostname=localhost
eureka.client.fetch-registry=false
eureka.client.register-with-eureka=false
```

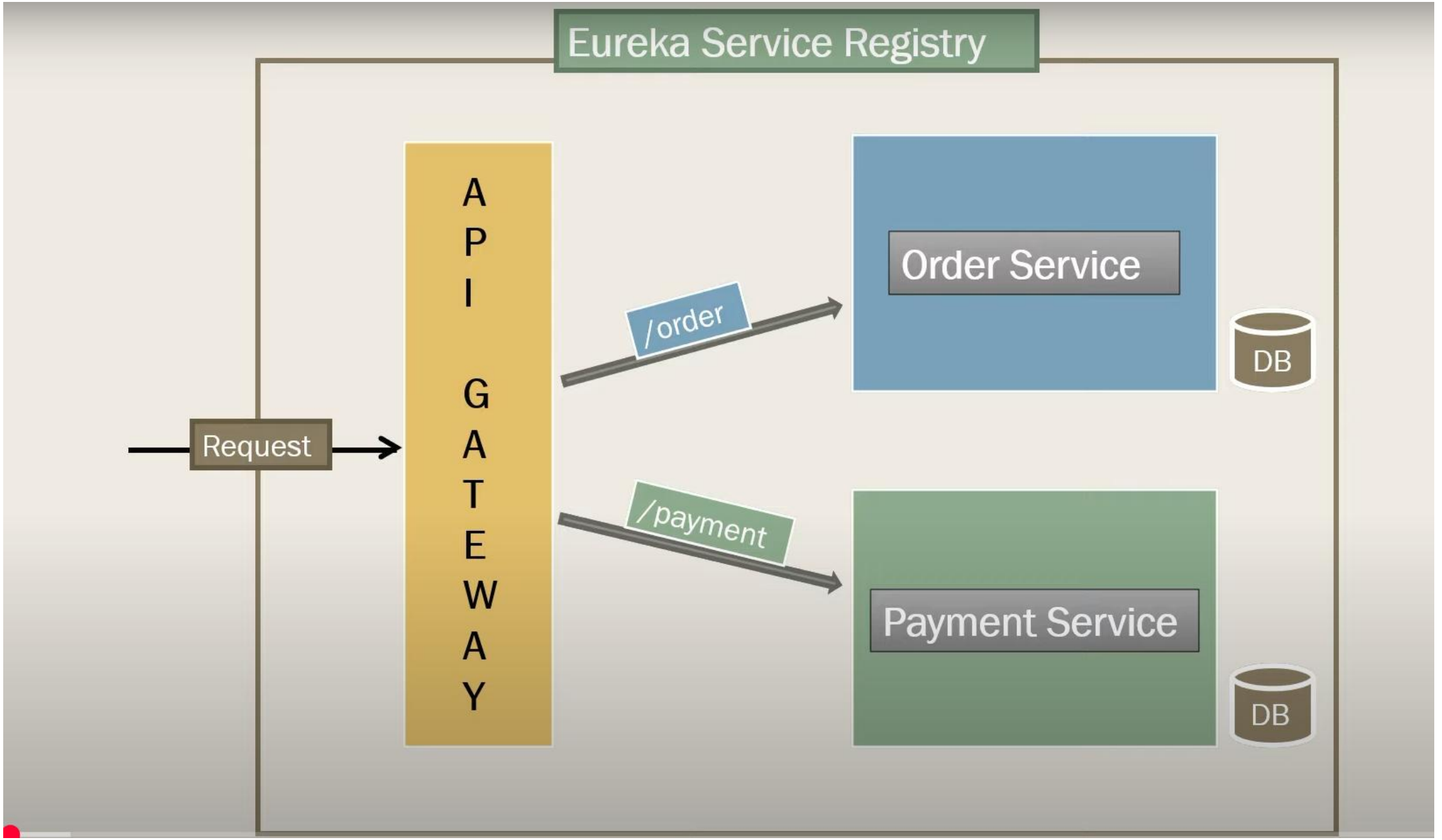
✅ Why Use API Gateway? (Advantages)

✅ Advantage	📄 Description
🎯 Centralized Routing	Route incoming requests to different microservices based on path/URL.
🔒 Security	Apply **auth**, **rate limiting**, **IP filtering**, **CORS** in one place.
🧠 Load Balancing	Works with Eureka + LoadBalancer to auto-balance traffic.
🎛️ Request Filtering	Add/modify headers, logging, validation, etc.
📉 Reduce Client Complexity	Clients don't need to know service URLs — just one URL.
📈 Scalability & Flexibility	Easily scale microservices, and manage versions (v1, v2, etc.).

🧠 What is an API Gateway?

An **API Gateway** is a single **entry point** for all your microservices.

Instead of clients calling each service directly, they go through the **gateway**, which forwards the request to the right service.


Eureka Service Registry
Request
API GATEWAY
/order
/payment
Order Service
Payment Service
DB
DB

✅ Why Use API Gateway? (Advantages)

✅ Advantage	📄 Description
🎯 Centralized Routing	Route incoming requests to different microservices based on path/URL.
🔒 Security	Apply **auth**, **rate limiting**, **IP filtering**, **CORS** in one place.
🧠 Load Balancing	Works with Eureka + LoadBalancer to auto-balance traffic.
🎛️ Request Filtering	Add/modify headers, logging, validation, etc.
📉 Reduce Client Complexity	Clients don't need to know service URLs — just one URL.
📈 Scalability & Flexibility	Easily scale microservices, and manage versions (v1, v2, etc.).

⚙️ What Dependencies Do You Need?

```xml
<!-- Spring Cloud Gateway -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>

<!-- Eureka Discovery Client -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>

<!-- Resilience4j for Circuit Breaker and Rate Limiting -->
<dependency>
    <groupId>io.github.resilience4j</groupId>
    <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

`application.properties`

```
# Application Info
spring.application.name=api-gateway
server.port=8080

# Eureka Client Config
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true
```

```
# Enable Gateway Routes
spring.cloud.gateway.discovery.locator.enabled=true
spring.cloud.gateway.discovery.locator.lower-case-service-id=true

# Optional: Circuit Breaker Default Config
resilience4j.circuitbreaker.instances.order-service.register-health-indicator=true
resilience4j.circuitbreaker.instances.order-service.sliding-window-size=5
resilience4j.circuitbreaker.instances.order-service.failure-rate-threshold=50
```

✅ `spring.cloud.gateway.discovery.locator.enabled=true`

What it does (in simple terms):

- It **automatically enables routing to services registered in Eureka**.
- You don't need to manually define routes for each service in your gateway.

Let's say your Eureka registry has this service:

```yaml
spring.application.name=order-service
```

✅ With `discovery.locator.enabled=true`, you can now access it like:

```bash
http://localhost:8080/order-service/some-api
```

And Spring Cloud Gateway will:

- Lookup `order-service` from Eureka.
- Forward the request to a healthy instance.
- Do this **without you manually defining routes**.

✅ Alternatives to Spring Cloud Gateway

Alternative	Description
Zuul 1 (Netflix)	Legacy gateway, replaced by Spring Cloud Gateway.
Kong	Open-source API Gateway written in Lua; supports plugins.
NGINX	Lightweight, fast gateway/reverse proxy.
Envoy	Proxy used in service mesh (Istio).
API Gateway (AWS/GCP)	Managed service for APIs in cloud.

🧠 What is Resilience4j?

Resilience4j is a tool to **protect your app** when something goes wrong — like when another service is **slow** or **crashes**.

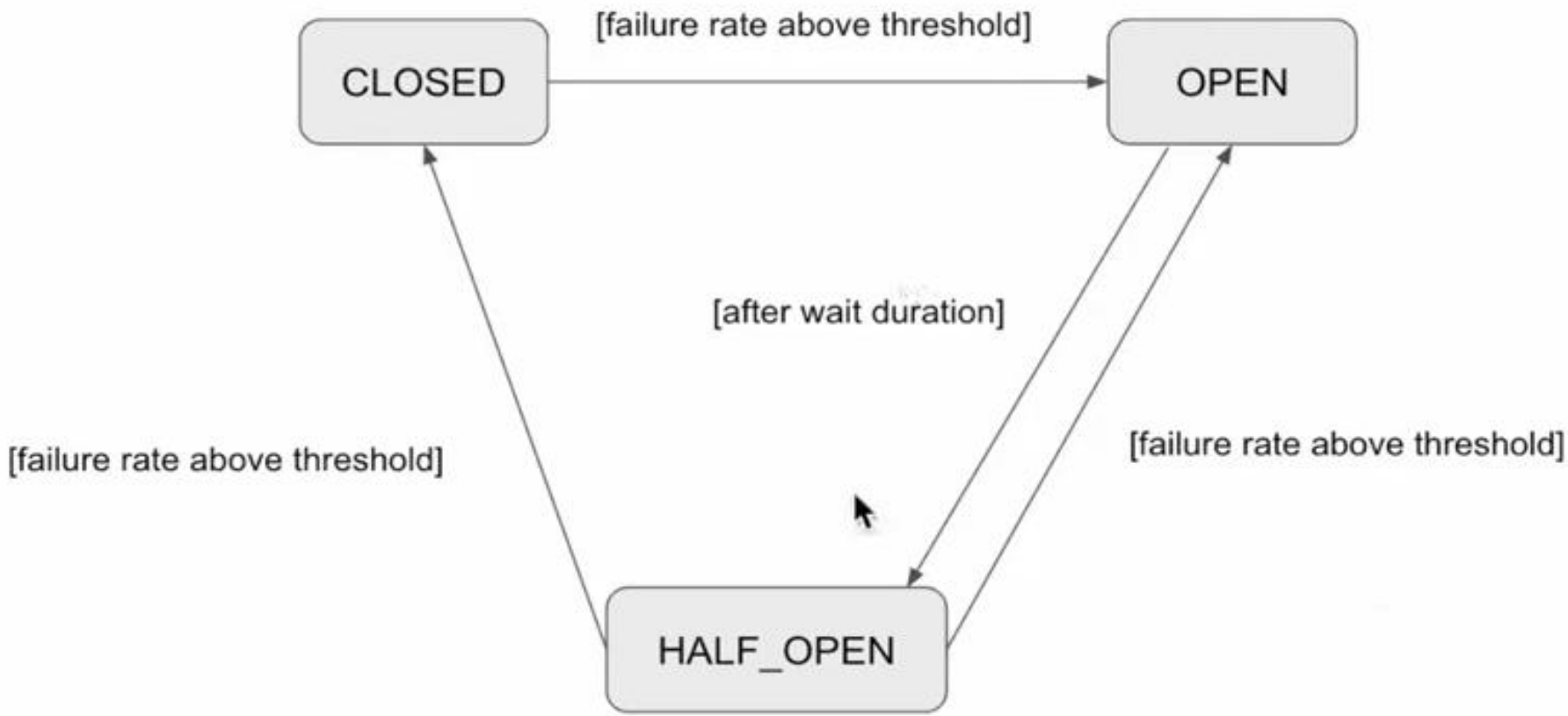
Think of it as a **shock absorber** or **safety net** for your microservices.

😟 The Problem (Without Resilience4j)

Imagine this:

- Your service calls another service (like `order-service`)
- That service is **down** or **too slow**
- Your app **waits too long**, users get **errors**
- If you keep calling it again and again, it can **crash your app too**

Circuit Breaker



How your circuit breaker works in 5 easy steps:

1. **It watches the last 10 calls** your app makes to the service to check if things are okay.
   (sliding-window-size=10)
2. **It waits until at least 5 calls happen** before deciding if the service is healthy or not.
   (minimum-number-of-calls=5)
3. **If 50% or more of those calls fail**, it opens the circuit breaker to stop calling the service.
   (failure-rate-threshold=50)
4. **When open, it waits 5 seconds** before trying again to see if the service got better.
   (wait-duration-in-open-state=5s)
5. **Then it lets 3 test calls through** to check if the service is healthy. If those succeed, it closes the breaker and resumes normal calls.
   (permitted-number-of-calls-in-half-open-state=3)

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

```java
@CircuitBreaker(name = "createQuiz", fallbackMethod = "fallBackMethodForCreateQuiz")
@PostMapping("/create")
public Quiz createQuiz(
        @RequestParam String category,
        @RequestParam String level,
        @RequestParam String title) {
    return quizService.createQuiz(category, level, title);
}

public Quiz fallBackMethodForCreateQuiz(String category, String level, String title, Throwable throwable) {
    log.warn("Fallback triggered for createQuiz: {}", throwable.getMessage());
    Quiz fallbackQuiz = new Quiz();
    fallbackQuiz.setTitle("Question server is down please try later - Service Unavailable");
    return fallbackQuiz;
}
```

```
# ================================
# Actuator & Circuit Breaker Health
# ================================
management.health.circuitbreakers.enabled=true
management.endpoints.web.exposure.include=health
management.endpoint.health.show-details=always

# ================================
# Resilience4j Circuit Breaker Defaults
# ================================
resilience4j.circuitbreaker.configs.default.register-health-indicator=true
resilience4j.circuitbreaker.configs.default.sliding-window-size=10
resilience4j.circuitbreaker.configs.default.minimum-number-of-calls=5
resilience4j.circuitbreaker.configs.default.permitted-number-of-calls-in-half-open-state=3
resilience4j.circuitbreaker.configs.default.automatic-transition-from-open-to-half-open-enabled=true
resilience4j.circuitbreaker.configs.default.wait-duration-in-open-state=5s
resilience4j.circuitbreaker.configs.default.failure-rate-threshold=50
resilience4j.circuitbreaker.configs.default.event-consumer-buffer-size=10
```

```
# ================================
# Resilience4j Circuit Breaker Defaults
# ================================

#Registers the circuit breaker with Spring Boot’s health system so Actuator can monitor it.
resilience4j.circuitbreaker.configs.default.register-health-indicator=true
#The breaker looks at the last 10 calls to decide whether to open or stay closed.
resilience4j.circuitbreaker.configs.default.sliding-window-size=10
#Don’t judge the service until at least 5 calls have been made.
resilience4j.circuitbreaker.configs.default.minimum-number-of-calls=5
#When the breaker is HALF_OPEN, only let 3 test calls go through before deciding if the service is healthy again.
resilience4j.circuitbreaker.configs.default.permitted-number-of-calls-in-half-open-state=3
#The breaker will automatically try again after a wait time – you don’t have to manually reset it.
resilience4j.circuitbreaker.configs.default.automatic-transition-from-open-to-half-open-enabled=true
#If the breaker is OPEN, wait 5 seconds before testing again.
resilience4j.circuitbreaker.configs.default.wait-duration-in-open-state=5s
# If 50% or more of the calls fail in the window, the breaker will OPEN.
resilience4j.circuitbreaker.configs.default.failure-rate-threshold=50
#Keep the last 10 events (success, failure, state changes) in memory for logging and monitoring.
resilience4j.circuitbreaker.configs.default.event-consumer-buffer-size=10
```

```java
@CircuitBreaker(name = "createQuiz", fallbackMethod = "fallBackMethodForCreateQuiz")
@Retry(name = "createQuiz")                    // Retry configured for the same instance "createQuiz"
@RateLimiter(name = "createQuiz")              // Rate limiter configured for "createQuiz"
@PostMapping("/create")
public Quiz createQuiz(
        @RequestParam String category,
        @RequestParam String level,
        @RequestParam String title) {
    return quizService.createQuiz(category, level, title);
}

public Quiz fallBackMethodForCreateQuiz(String category, String level, String title, Throwable throwable) {
    log.warn("Fallback triggered for createQuiz: {}", throwable.getMessage());
    Quiz fallbackQuiz = new Quiz();
    fallbackQuiz.setTitle("Question server is down please try later - Service Unavailable");
    return fallbackQuiz;
}
```

```
# ================================
# Resilience4j Retry Defaults
# ================================
resilience4j.retry.configs.default.max-attempts=3
  # Retry max 3 times before failing
resilience4j.retry.configs.default.wait-duration=2s
      # Wait 2 seconds between retries
resilience4j.retry.configs.default.retry-exceptions=java.io.IOException,java.util.concurrent.TimeoutException
 # Retry only on these exceptions




# ================================
# Resilience4j Rate Limiter Defaults
# ================================
resilience4j.ratelimiter.configs.default.limit-for-period=5
   # Max 5 calls allowed per refresh period
resilience4j.ratelimiter.configs.default.limit-refresh-period=10s
  # Refresh period for rate limiter is 10 seconds
resilience4j.ratelimiter.configs.default.timeout-duration=1s
   # Wait max 1 second to acquire permission before failing
```

1. @CircuitBreaker

- **Stops calling the method if too many errors happen.**
- Think of it like a switch: if the service is failing too much, it "opens" the switch and stops trying for a while.
- After some time, it tests if the service is okay again.

---

2. @Retry

- **If the method fails, try it again a few times.**
- Like saying, “Oops, try one more time!”
- Good for temporary problems, like a short network glitch.

---

3. @RateLimiter

- **Limits how often you can call the method.**
- Like a gatekeeper allowing only a certain number of calls in a time frame.
- Protects your service from being overloaded.

What is Spring Cloud Config Server?

- It's a centralized **configuration server**.
- Stores all config properties for multiple microservices in one place (e.g., Git repo or local files).
- Microservices fetch config from Config Server dynamically.
- This helps you **manage configuration easily and consistently** across many services.
