01_Why, What and How?

Microservices

Why, What and How?

Why to use Microservices?

Microservices solve some of the key problems faced in Monolithic Application.

- **Huge JAR** files.
- **Small change** in code leads to **Heavy Deployment.**
- It forces us to **scale the whole Application**, even if only one API is facing heavy traffic.
- One API Failure means complete **Application outage**.
- Tightly coupled technologies. -> 1 language requirement limits our options
- **Large teams**, less communication, more Conflict.
- Slow and **risky deployment**.

What is Microservices?

Microservice is an Architecture

- Small and **Independent Services**.
- Built around business capabilities.
- **Each Service** owns it's **own data**.
- Communicate via API's or Events.
- Deployed and Scaled Independently.
- Resilient to failures.
- Not Just multiple SpringBoot Applications.

How to Implement Microservices?

There are several stages for reaching complete MicroServices Application.

- **Stage-1: Monolithic Application**
  - Creating a single Huge Application which contains only one Database and only one Programming Language.
- **Stage-2: Splitting Code-Base**
  - Splitting the Huge Application into Several Microservices, but points to same Database.
- **Stage-3: Splitting Data-Base**
  - Splitting the Single Database into several specified Databases.
- **Stage-4: Centralized Configuration**

    - Suppose There are common properties for all the Microservices instead of repeating the code.
    - We store it in a GitHub / Bitbucket / GitLab and use **Spring Cloud Config Server** to access.
    - Here **Spring Cloud Config Server** will be our new Application.
- **Stage-5: Client - Side Load Balancing**
    - Suppose there is a Huge traffic on a API then we create multiple instances of that API(ex: Mail Server-1 and Mail Server-2.
    - So we need to send the request to all the instances equally we Use Spring Cloud LoadBalancer which by default uses Round-Robin technique to distribute the requests.
- **Stage-6: Service Discovery**
    - Since we have many Applications in a single Application it is difficult to manage all the Port numbers.
    - So, we Use **Eureka Server.**
    - All the Applications are clients of Eureka Server, when we up the application it registers itself In Eureka Server.
    - Eureka Server is our new Application.
- **Stage-7: Fault Tolerance**
    - Suppose one Application is Down and the Other Application calls it then there will be an Exception, and also we need to clear it when the Application is Ready, To handle this Exception Gracefully
    - We use **Resilience4j.**
- **Stage-8: API Gateway**
    - API Gateway is a centralized entry point that routes client requests to appropriate microservices while handling cross-cutting concerns like security and logging.
    - Clients talk to one gateway, and the gateway talks to multiple microservices.
- **Stage-9: Declarative Client**
    - Declarative Client is a way to call another microservice by **just defining an interface**, without writing the actual HTTP call code.
    - You declare what service you want to call, and the framework handles the communication.
- **Stage-10: Distributed Tracing**
    - Distributed Tracing is a technique used to track a single request as it travels across multiple microservices.
    - It shows the **full journey of a request across all services**.
- **Stage-11: Centralized Configuration**
- **Stage-12: Service Discovery**
- **Stage-13: Distributed Tracing**
- **Stage-14: Event Driven Communication, Asynchronous, Kafka.**

- **Kafka** is a distributed platform used to send, store, and process data streams **(events/messages)** in real time between systems or services.