Microservices

+++++++++++++

What is Monolith Architecture

+++++++++++++++++++++++++

-> If we develop all the functionalities in single project then it is called as Monolith architecture based application

-> We will package our application as a jar/war to deploy into server

-> As monolith application contains all functionalities, it will become fat jar/war

Advantages

+++++++++++++

1) Simple to develop
2) Everything is available at once place
3) Configuration required only once

Dis-Advantages

+++++++++++++

1) Difficult to maintain
2) Dependencies among the functionalites
3) Single Point Of Failure
4) Entire Project Deployment

*** To overcome the problems of Monolith, Microservices architecture came into market****

-> Microservices is not a programming language

-> Microservices is not a framework

-> Microservices is not an Specification API

-> Microservices is an architectural design pattern

-> Microservices suggesting to develop application functionalities with loosely coupling

-> In Microservices architecture we don't develop all the functionalities in single project. We will divide project functionalities into several REST APIs

***Note: One REST API is called as one Microservice .

-> Microservices architecture based project means collection of REST APIs.
-> Microservices is not related to only java. Any programming language specific project can use Microservices Architecture.

Advantages
+++++++++++++

1) Loosely Coupling
2) Easy To maintain
3) Faster Development
4) Quick Deployment
5) Faster Releases
6) Less Downtime
7) Technology Independence

Dis-Advantages
+++++++++++++

1) Bounded Context
2) Lot of configurations
3) Visibility
4) Pack of cards

++++++++++++++++++++++++++

Microservices Architecture

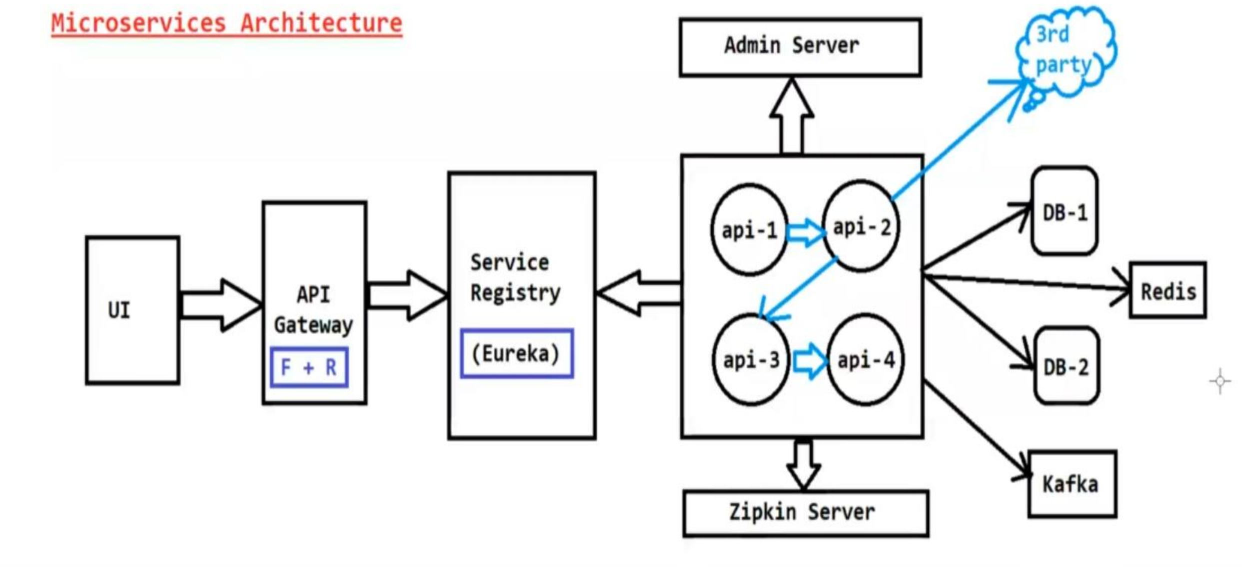
++++++++++++++++++++++++++

-> We don't have any fixed architecture for Microservices

-> People are customizing microservices architecture according to their requirement

-> Most of the projects will use below components in Microservices Architecture

1) Service Registry (Eureka Server)
2) Services (REST APIs)
3) Interservice Communication (FeginClient)
4) API Gateway (Zuul Proxy)
5) Admin Server
6) Sleuth & Zipkin Server



Service Registry

++++++++++++++++

-> Service Registry acts as DB of services available in the project

-> It provides the details of all the services which are registered with Service Registry

-> We can identify how many services available in the project

-> We can identify how many instances available for each service

-> We can use "Eureka Server" as service registry

-> Eureka Server provided by "Spring Cloud Netflix" library

Services

+++++++++

-> Services means REST APIs / Microservices
-> Services contains backend business logic
-> In the project, some services will interact with DB
-> In the project, some services will interact with third party REST API (external communication)
-> In the project, some services will interact with another services with in the project (inter-service communication)
-> For inter-service communication we will use feign-client
-> To distribute the load, we can run one service with Multiple Instances (Load Balancing)

Note: We will register every service with Service Registry

API Gateway

+++++++++++

-> API Gateway is used to manage our backend apis of the project
-> API Gateway acts as mediator between end users and backend apis
-> API Gateway can filter logic to decide request processing
-> API Gateway will contain Routing logic (which request should go to which REST API)
-> API Gateway also will be registered with Service Registry

+++

Mini Project Implementation using Microservices Architecture

+++

1) Service Registry (Eureka Server)
2) Spring Boot Admin Server (To monitor & manage boot applications)

3) Zipkin Server (Distributed Log Tracing)
(https://zipkin.io/pages/quickstart.html)

Steps to develop Service Registry Application (Eureka Server)

+++

1) Create Service Registry application with below dependency

    a) EurekaServer (spring-cloud-starter-netflix-eureka-server)
    b) web-starter
    c) devtools

2) Configure @EnableEurekaServer annotation in boot start class

3) Configure below properties in application.yml file

```
server:
  port: 8761

eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
```

Note: If Service-Registry project port is 8761 then clients can discover service-registry and will register automatically with service-registry.

If service-registry project running on any other port number then we have To register clients with service-registry manually.

4) Once application started we can access Eureka Dashboard using below URL

URL : http://localhost:8761/

Steps to develop Spring Boot Admin Server Project

+++

1) Create Boot application with below dependencies

    a) web-starter
    b) devtools
    c) admin-server (codecentric)

2) Configure @EnableAdminServer annotation at boot start class

3) Configure the port number and run the application (port : 1111)

4) After application started, access Admin Server UI using app-url

URL : http://localhost:1111/

Steps to work with Zipkin Server

++++++++++++++++++++++++++++++++

1) Download Zipkin server jar from website

URL : https://zipkin.io/pages/quickstart.html

2) Run the zipkin server jar from command prompot

Cmd : java -jar <jar-file-name>

Note: Zipkin server will run on 9411 port number

3) Access Zipkin server dashboard in browser

URL : http://localhost:9411/

Steps to develop GREET-API

++++++++++++++++++++++++++

1) Create Spring Boot application with below dependencies

- eureka-discovery-client
- starter-web
- devtools
- actuator
- sleuth
- zipkin
- admin-client

2) Configure @EnableDiscoveryClient annotation at start class

3) Create RestController with required method

4) Configure below properties in application.yml file

```
----------------------------------application.yml----------------------------------------
server:
  port: 9090
spring:
  application:
    name: GREET-API
  boot:
    admin:
      client:
        url: http://localhost:1111/
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
management:
  endpoints:
    web:
      exposure:
        include: '*'

-------------------------------------------------------------------------------------------
```

5) Run the application and check in Eureka Dashboard (It should display in eureka dashboard)

6) Check Admin Server Dashboard (It should display) (we can access application details from here)

Ex: Beans, loggers, heap dump, thred dump, metrics, mappings etc...

}

7) Send Request to REST API method

8) Check Zipkin Server UI and click on Run Query button
(it will display trace-id with details)

Steps To Develop WELCOME-API

++++++++++++++++++++++++++++++++

1) Create Spring Boot application with below dependencies

- web-starter
- devtools
- eureka-discovery-client
- fegin-client
- admin-client
- zipkin-client
- sleuth
- actuator

2) Configure @EnableDiscoveryClient & @EnableFeignClients annotations at boot start class

3) Create FeignClient to access GREET-API

```
@FeignClient(name = "GREET-API")
public interface GreetApiClient {

	@GetMapping("/greet")
	public String invokeGreetApi();

}
```

4) Create RestController with required method

Note: In Rest Controller we should have logic to access another REST API (GREET-API)

-> For Interservice Communication we will use FeignClient

-> Using FeginClient we can make rest call to another service using name of the service (no need of url)

-> FeginClient will get service URL from service-registry based on service-name

```
@RestController
public class WelcomeRestController {

private Logger logger = LoggerFactory.getLogger(WelcomeRestController.class);

        @Autowired
        private GreetApiClient greetClient;

        @GetMapping("/welcome")
        public String welcomeMsg() {

                logger.info("welcomeMsg() execution - start");
                String welcomeMsg = "Welcome to Ashok IT..!!";
                String greetMsg = greetClient.invokeGreetApi();
                logger.info("welcomeMsg() execution - end ");
                return greetMsg + ", " + welcomeMsg;
        }
}
```

5) Configure below properties in application.yml file

```
server:
  port: 9091
spring:
  application:
    name: WELCOME-API
  boot:
    admin:
      client:
        url: http://localhost:1111/
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka
management:
  endpoints:
    web:
      exposure:
        include: '*'
```

6) Run WELCOME-API project (it should register in Eureka and Admin server)
7) Send Request to welcome-api (it should final response)
8) Verify Zipkin Server Dashboard for log tracing

+++

-> We are running Service Registry project with Eureka Server on 8761 port number

-> Eureka Discovery Client applications are auto-registering with Eureka Server when port is 8761

-> If we change Eureka Server port number then we have to register Eureka Client application with Eureka Server using below property in application.yml file

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:9090/eureka
```

Note: We should configure this property in eureka client application yml file

++

GREET API URL : DESKTOP-BDG00U7:GREET-API:9090/

WELCOME API URL : DESKTOP-BDG00U7:WELCOME-API:9091/

+++++++++++++

API Gateway

++++++++++++++

-> API Gateway will act as mediator between client requests & backend apis
-> API Gateway will provide single entrypoint to access our backend apis
-> In Api Gateway we will write mainley below 2 types of logics

1) Filters
2) Routing

-> Filters are used to execute some logic before request processing and after request processing
-> Routing is used to tell which request should go to which REST API
-> In Spring Cloud, we have 2 options to create API Gateway

1) Zuul Proxy (old approach)
2) Spring Cloud Gateway (latest approach)

Note: Zuul Proxy is not supported by latest versions of spring boot

+++++++++++++++++++++++++++++++++++++

Working with Spring Cloud API Gateway

++++++++++++++++++++++++++++++++++++

1) Create Spring boot application with below dependencies

-> web-stater
-> eureka-client
-> cloud-gateway
-> devtools

2) Configure @EnableDiscoveryClient annotation at boot start class
3) Configure API Gateway Routings in application.yml file like below

```
--------------------------------------application.yml file--------------------------------
spring:
  cloud:
    gateway:
      discovery.locator:
        enabled: true
        lowerCaseServiceId: true
      routes:
      - id: welcome-api
        uri: lb://WELCOME-API
        predicates:
        - Path=/welcome
      - id: greet-api
        uri: lb://GREET-API
        predicates:
        - Path=/greet
  application:
    name: CLOUD-API-GATEWAY
```

```
server:
  port: 3333
```

In API gateway we will have 3 types of logics

1) Route
2) Predicate
3) Filters

-> Routing is used to defined which request should be processed by which REST API in backend. Routes will be configured using Predicate

-> Predicate : This is a Java 8 Function Predicate. The input type is a Spring Framework ServerWebExchange. This lets you match on anything from the HTTP request, such as headers or parameters.

-> Filters are used to manipulate incoming request and outgoing response of our application

Note: Using Filters we can implement security also for our application.

```
@Component
public class MyPreFilter implements GlobalFilter {

        private Logger logger = LoggerFactory.getLogger(MyPreFilter.class);

        @Override
        public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

                logger.info("MyPreFilter :: filter () method executed...");
```

```
            // Accessing HTTP Request information
            ServerHttpRequest request = exchange.getRequest();

            HttpHeaders headers = request.getHeaders();
            Set<String> keySet = headers.keySet();

            keySet.forEach(key -> {
                List<String> values = headers.get(key);
                System.out.println(key +" :: "+values);
            });

            return chain.filter(exchange);
        }
}
```

++

-> We can validate client given token in the request using Filter for security purpose

-> We can write request and response tracking logic in Filter

-> Filters are used to manipulate request & response of our application

-> Any cross-cutting logics like security, logging, moniroing can be implemented using Filters

++++++++++++++

Sleuth & Zipkin

++++++++++++++

-> Microservices application means several REST APIs will be available

-> As part of application execution one Rest API can communicate another REST API

-> When we send request from UI, it will process by Multiple REST APIs with Interservice communication

** How we can understand which rest api is taking more time to process request ? ***

-> If we add Sleuth dependency in REST API then it will add span-id and trace-id for log messages
-> For every request once span-id will be generated by Sleuth
-> If one request is processing multiple REST API then Sleuth will use same span-id for REST APIs to generate log message.
-> Trace-id is specific to one REST API
-> By using span-id and trace-id we can understand which REST api has taken more time process request
-> To monitor span-id and trace-id details we will use ZipKin server
-> Zipkin server is providing user interface (UI) to monitor all the details

Note: The REST APIs which are having sleuth dependency should register with Zipkin server
Note: By using Sleuth and Zipkin we achieve Distributed Log Tracing

Steps to work with Sleuth and Zipkin
+++++++++++++++++++++++++++++++

1) create spring-boot application with below dependencies

a) web-starter
b) sleuth
c) zipkin
d) devtools

2) Create a REST Controller with required methods
3) Download zipkin-server jar file (https://zipkin.io/pages/quickstart)
4) Run zipkin-server using "java -jar <zipkin-jar-filename"

Note: Zipkin server runs on 9411 port

5) Run spring boot application and send a request to rest controller method

6) Verify boot application logs display in console (span-id and trace-id will be attached to logs)

7) Go to Zipkin server dashboard and monitor event details

(URL : http://localhost:9411)

++