MicroServices:

================ 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 (We can developed backend APIs with multiple technologies)

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

1. Bounded Context (Deciding no. of services to be created)
2. Lot of configurations (Common configurations)
3. No Visibility
4. Pack of cards (crucial microservice is down then remaining microservices are of no use for eg. payment mircoservice)

+++++++++++++++++++++++++
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 contain filter logic to decide request processing (Authentication)
-> API Gateway will contain Routing logic (which request should go to which REST API)
-> API Gateway also will be registered with Service Registry
-> Spring Cloud Gateway we can use as API Gateway

++++++++++++
Admin Server
++++++++++++
→ Admin Server is used to manage all backend APIs actuator endpoints at one place.
→ Our Backend apis will be registered with admin server
→ Admin Server will provide User Interface to monitor backend apis actuator endpoints.

+++++++++++++
Zipkin Server
+++++++++++++
→ Zipkin Server is used for Distributed Tracing.
→ Using this Zipkin, we can monitor which API is taking more time to process our request.

+++
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.properties file

server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
eureka.instance.hostname=localhost

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.

1. 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

1. 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--
  spring:
  application:
  name: 37-SB-MS-Greet-API
  boot:
  admin:
  client:
  url:
  http://localhost:1111/
  instance:
  management-base-url:
  http://localhost:9091/

server:
port: 9091

eureka:
client:
serviceUrl:
defaultZone:
http://localhost:8761/eureka/
instance:
hostname: localhost

management:
endpoints:

web:
exposure:
include: '*'
endpoint:

health:
show-details: always

---

1. Run the application and check in Eureka Dashboard (It should display in eureka dashboard)
2. Check Admin Server Dashboard (It should display) (we can access application details from here)

   Ex: Beans, loggers, heap dump, thred dump, metrics, mappings etc...
3. Send Request to REST API method
4. 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();
```

}

1. 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;
 }
```

}

1. Configure below properties in application.yml file

```
eureka:
client:
serviceUrl:
defaultZone:
http://localhost:8761/eureka/
instance:
hostname: localhost
management:
endpoint:
health:
show-details: always
endpoints:
web:
exposure:
include: '*'
server:
port: 8081

spring:
application:
name: Welcome-API
boot:
admin:
client:
instance:
management-base-url:
http://localhost:8081/
url:
http://localhost:1111/
```

1. Run WELCOME-API project (it should register in Eureka and Admin server)
2. Send Request to welcome-api (it should final response)
3. 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 with 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 our 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 trace-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. Add below dependency in welcome-api and greet-api projects pom.xml

   ```
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-sleuth</artifactId>
   </dependency>

   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-sleuth-zipkin</artifactId>
   </dependency>
   ```

2. Download zipkin-server jar file (https://zipkin.io/pages/quickstart)
3. Run zipkin-server using "java -jar <zipkin-jar-filename"

Note: Zipkin server runs on 9411 port

1. Run spring boot applications and send a request to rest controller method
2. Verify boot application logs display in console (span-id and trace-id will be attached to logs)
3. Go to Zipkin server dashboard and monitor event details

   (URL : http://localhost:9411)

++

1. What is Monolith Architecture ?
2. Pros and Cons of Monolith Architecture
3. Microservies Introduction
4. Pros and Cons of Microservices
5. Microservices Architecture
6. Service Registry (Eureka)

7. Admin Server (Monitor & Manager actuators)
8. Zipkin Server with Sleuth
9. Backend Apis Development
10. Inter-service communication (Feign Client)
11. Load Balancing with Ribbon
12. Api Gateway (Front end gate of all backend apis)
13. Filters & Routings in API Gateway

======================
Cloud Config Server

⇒ As of now we are configuring properties in application.properties or application.yml file

```
             Ex:   DB Props, SMTP props, Kafka Props, Messages etc...
```

⇒ application.properties or application.yml file will be packaged along with our application.

⇒ If we want to make any changes to properties then we have to re-package our application

⇒ To externalize properties from the application we can use Cloud Config Server

===================
Config Server App

1. Create Git Repository and keep ymls files required for projects

```
     Note: We should keep file name as application

     app name : greet  then file name : greet.yml

     app name : welcome then file name : welcome.yml


  ### Git Repo : <https://github.com/kamalesiddhesh/configuration_properties>
```

2. Create Spring Starter application with below dependency

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

3. Write @EnableConfigServer annotation at boot start class

```
@SpringBootApplication
@EnableConfigServer
public class Application {
```

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

```
}
```

1. Configure below properties in application.yml file

```
spring:
 application:
```

    name: 40-SB-MS-Cloud-Config-Server
  cloud:
    config:
      server:
        git:
          uri:
https://github.com/kamalesiddhesh/configuration_properties.git
          default-label: master
          clone-on-start: true

management:
  security:
    enabled: false

1. Run Config Server application

==================================== Config Server Client Development

1. Create Spring Boot application with below dependencies

```
        a) web-starter
        b) config-client
        c) dev-tools
```

<dependency>
<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-config</artifactId>
</dependency>

1. Create Rest Controller with Required methods

@RestController
@RefreshScope
public class WelcomeRestController {

```
@Value("${msg}")
private String msg;

@GetMapping("/")
public String getWelcomeMsg() {
    return msg;
}
```

}

1. Configure ConfigServer url in application.yml file like below

server:
  port: 9090

spring:
  application:
    name: welcome
  config:
    import: optional:configserver:
http://localhost:8080

1. Run the application and test it.

==

==================================== Circuit Breaker

- > Circuit Breaker is a design pattern in Microservices
- > Circuit Breaker is used to implement fault-tolerance systems
- > Fault-tolerance systems are also called as resillence systems
- > Fault-tolerance system means when main logic is failed to execute then we should execute fallback logic to process client request

============ Usecase

⇒ Get data from redis, if redis logic is failing then we should get data from database

Note: If redis logic is failing for 3 requests continuosly then execute db logic for 30 mins. After 30 mins re-try for redis logic execution if it is working then execute redis logic only. If 3 re-try executions failed with redis then execute db logic for next 30 mins.

- > To implement circuit-breaker we should add below dependency in pom.xml file

Add First below dependency ⇒

```
    <dependency>
        <groupId>io.pivotal.spring.cloud</groupId>
        <artifactId>spring-cloud-services-starter-circuit-breaker</artifactId>
    </dependency>
```

- ******************** OR ***************************************

  <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-netflix-hystrix -->
  ```
  <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
     <version>2.2.10.RELEASE</version>
  </dependency>
  ```

  <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-netflix-hystrix-dashboard -->
  ```
  <dependency>
     <groupId>org.springframework.cloud</groupId>
     <artifactId>spring-cloud-netflix-hystrix-dashboard</artifactId>
     <version>2.2.10.RELEASE</version>
  </dependency>
  <dependency>
     <groupId>org.hdrhistogram</groupId>
     <artifactId>HdrHistogram</artifactId>
     <version>2.1.12</version>
  </dependency>
  ```

- *Important Note :

1. [Error] Post-processing of merged bean definition failed (lower spring boot version and also java execution version)

- Cause: Issue caused by incompatibility between spring boot and spring cloud (hystrix) versions
- Solved: Resolved by lowering the spring boot version (another document says 2.3.X is sufficient) (2.5.6 > 2.3.8.RELEASE)
  출처:
  https://co-de.tistory.com/33

How to lower spring boot version and Java Execution Version?
⇒

1. Change Spring boot version in <parent> tag in POM.xml

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.8.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
2. Lower execution environment version in build path by changing to Java SE 1.7(jdk 17)
    [2nd step is optional it should work in first step only]
```

- > Add below annotations at boot start class

  @EnableHystrix
  @EnableCircuitBreaker
  @EnableHystrixDashboard

@RestController
public class DataRestController {

```
@GetMapping("/data")
@HystrixCommand(
        fallbackMethod = "getDataFromDB",
        commandProperties = {
                @HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="3"),
                @HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds", value="1
0000"),
                @HystrixProperty(name="circuitBreaker.enabled", value="true")
        }
)
public String getDataFromRedis() {
    System.out.println("**getDataFromRedis() method called**");
    if (new Random().nextInt(10) <= 10) {
        throw new RuntimeException("Redis Server Is Down");
    }
    // logic to access data from redis
    return "data accessed from redis (main logic) ....";
}

public String getDataFromDB() {
    System.out.println("**getDataFromDB() method called**");
    // logic to access data from db
    return "data accessed from database (fall back logic) ....";
}
```

}

###########################
Load Balancer :
###########################

Note: When we run application on single server then it may not handle huge load

⇒ When many requests are coming then our server might crash and app will go down.

⇒ To avoid these problems, we will use Load Balancer for application execution.

⇒ Load Balancer is used to distribute application load to multiple servers to reduce burden.

Note : Instead of running our application in single server,
we will run our application in multiple servers to reduce burden.

Note : When application is running in multiple servers then we will call them as Instances.

- Command to Send server.port runtime (Need to be add in VM Argument of Run Configuration)

```
                    -Dserver.port=9093
```

Note : Remove hard-coded server.port in application.properties file.

To find which server our app is running ,get that server port number

- → String port = env.getProperty("server.port");

This is CLient-Side Load Balancing - the instances of the microservice are deployed on several servers.