MicroServices With SpringBoot

by

Dilip Singh

dilipsingh1306@gmail.com

dilipsingh1306

Microservices, also known as the microservice architecture, is an architectural style used in software development to design and build applications as a collection of small, independent, and loosely coupled services. Each service represents a specific business capability and can be developed, deployed, and maintained independently of other services.

In a traditional monolithic application, all functionality is bundled together into a single large codebase, making it difficult to scale, maintain, and update. Microservices, on the other hand, promote a more modular approach, breaking down the application into smaller, manageable components.

Key characteristics of microservices include:

Decentralization: Each microservice operates independently and has its own database (if required) and codebase.

Loose coupling: Microservices communicate with each other through well-defined interfaces, usually using lightweight protocols like HTTP/REST or messaging systems.

Independently deployable: Since each microservice is self-contained, updates or bug fixes can be made to a single service without affecting the others, reducing the risk of unintended side effects.

Scalability: Specific services can be scaled independently based on demand, optimizing resource utilization.

Resilience: If one microservice fails, the entire application doesn't necessarily collapse. The other services can continue to function as long as they don't depend on the failed service.

Technology diversity: Microservices allow the use of different programming languages, frameworks, and data storage systems for each service, as long as they can communicate effectively.

Focused on business capabilities: Each microservice is designed to address a specific business capability or function, making it easier to understand and maintain.

Though microservices offer several advantages, they also introduce complexities like distributed system management, inter-service communication, and the need for robust monitoring. Implementing microservices requires careful planning, and it is essential to choose the right architecture based on the specific needs and goals of the application or project. When done right, microservices can lead to a more agile, scalable, and maintainable software development process.

Some examples of applications that use microservices architectures include:

Amazon: Amazon uses microservices to build its e-commerce platform. Each microservice is responsible for a specific function, such as product search, checkout, or order fulfilment.

Netflix: Netflix uses microservices to stream movies and TV shows. Each microservice is responsible for a specific content type, such as movies, TV shows, or documentaries.

Uber: Uber uses microservices to power its ride-hailing platform. Each microservice is responsible for a specific aspect of the ride-hailing process, such as finding drivers, matching riders with drivers, and tracking rides.

If you are considering using a microservices architecture for your next application, there are a few things you should keep in mind:

- Microservices can be complex to design and implement. It is important to have a clear understanding of your application's requirements before you start designing your microservices architecture.
- Microservices require a good DevOps culture. You need to have a way to automate the deployment, scaling, and monitoring of your microservices.
- Microservices can be more expensive to maintain than monolithic architectures. You need to have a way to track and manage the dependencies between your microservices.

Overall, microservices architectures can be a good choice for applications that need to be scalable, resilient, and evolvable. However, they can be complex to design and implement, so you need to carefully consider your application's requirements before you decide to use a microservices architecture.

Difference Between Monolithic and Microservices:

Monolithic and microservices are two different architectural styles used in software development. They have distinct characteristics that impact how applications are designed, built, deployed, and maintained. Here are the key differences between monolithic and microservices:

Architecture:

Monolithic: In a monolithic architecture, the entire application is built as a single, cohesive unit. All the components, functionalities, and services are tightly integrated and deployed together.

Microservices: In a microservices architecture, the application is divided into smaller, independent services that communicate with each other through APIs. Each service is developed and deployed independently, and they can be written in different programming languages.

Scalability:

Monolithic: Scaling a monolithic application often involves replicating the entire application, including all its components, even if only a specific part of the application requires more resources.

Microservices: Microservices offer finer-grained scalability. You can scale individual services independently based on their specific resource needs, allowing for more efficient resource utilization.

Deployment and Release Cycle:

Monolithic: Since the entire application is deployed as one unit, making changes to specific parts requires deploying the whole application. This can result in longer release cycles and a higher risk of introducing bugs.

Microservices: Microservices enable continuous deployment and faster release cycles. Changes to a specific service can be made and deployed independently, without affecting the rest of the application. This makes it easier to release updates and bug fixes more frequently.

Development Team Organization:

Monolithic: In a monolithic architecture, all developers typically work on the same codebase. As the application grows, it may become more challenging for developers to work simultaneously without causing conflicts.

Microservices: Microservices promote a decentralized development approach. Different development teams can work on separate services, which fosters autonomy and faster development cycles.

Technology Diversity:

Monolithic: Monolithic applications usually stick to a single technology stack since they are all built as a single unit.

Microservices: In microservices, different services can use different technology stacks that are best suited for their specific tasks. This allows for greater flexibility and choice in technology.

Fault Isolation and Resilience:

Monolithic: A failure in one part of the monolithic application can bring down the entire system since all components are tightly integrated.

Microservices: Microservices are designed for fault isolation. If a single service fails, it does not necessarily affect the other services, increasing the overall system's resilience.

Complexity and Maintenance:

Monolithic: As a monolithic application grows, it can become more complex and challenging to maintain, especially when many developers are working on it simultaneously.

Microservices: While each microservice is simpler in structure, managing and coordinating multiple services can introduce its own complexities.

Both monolithic and microservices architectures have their advantages and challenges, and the choice between them depends on various factors, including the specific requirements of the application, team structure, scalability needs, and development philosophy.

Microservices with Spring Boot:

Microservices with Spring Boot is a popular combination for building scalable and flexible applications based on the microservices architectural style. Spring Boot is a powerful framework from the Spring ecosystem that simplifies the development of Java-based applications, while microservices is an architectural approach that breaks down applications into smaller, loosely-coupled services that can be developed, deployed, and maintained independently.

Here are some key aspects of building microservices with Spring Boot:

Service Creation: Spring Boot provides a convenient way to create microservices using various starters and auto-configuration. You can quickly set up a new microservice project with just a few lines of code.

Independence: Each microservice developed using Spring Boot is an independent unit of functionality, allowing developers to focus on specific business capabilities without being affected by the implementation details of other services.

Communication: Microservices need to communicate with each other to fulfill complex business processes. Spring Boot offers various ways to implement communication between services, such as RESTful APIs, messaging systems (e.g., RabbitMQ or Apache Kafka), and gRPC.

Spring Cloud: Spring Cloud is an extension of the Spring ecosystem that provides additional tools and libraries to simplify the development of microservices. It offers features like service discovery (Eureka), load balancing (Ribbon), centralized configuration management (Spring Cloud Config), circuit breakers (Hystrix), and more.
Containerization: Microservices are often deployed using containerization technologies like Docker and container orchestration platforms like Kubernetes. Spring Boot applications are well-suited for containerization due to their lightweight nature and easy deployment.

Scalability: Spring Boot microservices can be scaled individually, allowing you to allocate resources based on the specific needs of each service. This fine-grained scalability is one of the advantages of the microservices architecture.

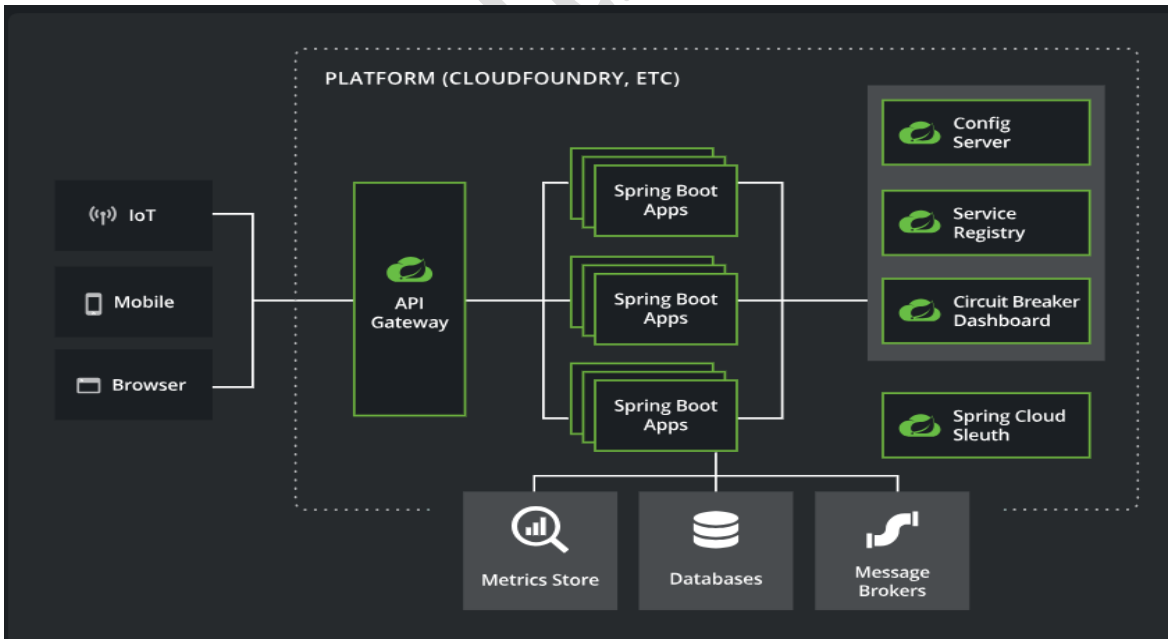
Monitoring and Logging: Monitoring and logging are crucial for maintaining the health of microservices. Spring Boot provides integration with popular logging frameworks like Logback and supports metrics and monitoring tools like Spring Actuator and Micrometer.

Testing: Spring Boot offers a testing framework that makes it easier to write unit tests and integration tests for microservices. You can test each service independently, ensuring that it functions correctly in isolation and when interacting with other services.

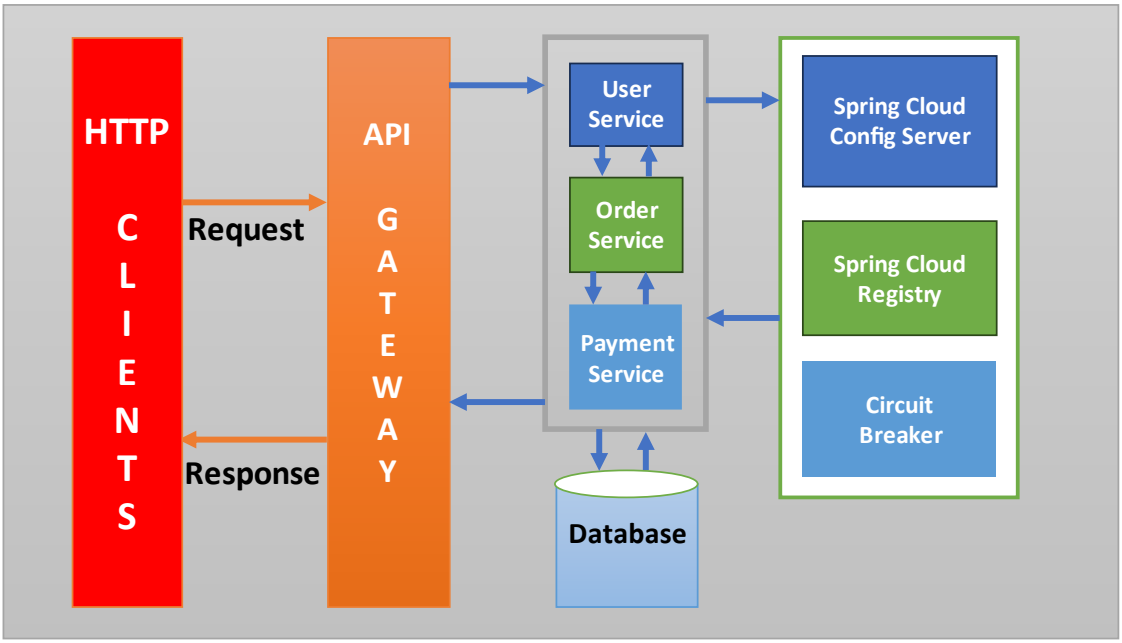
Remember that while Spring Boot simplifies many aspects of microservices development, building a successful microservices architecture still requires careful design and consideration of factors such as service boundaries, communication patterns, data consistency, and deployment strategies. Spring Boot's many purpose-built features make it easy to build and run your microservices in production at scale. And don't forget, no microservice architecture is complete without Spring Cloud API.

SpringBoot and Microservices Architecture:

The distributed nature of microservices brings challenges. Spring helps you mitigate these. With several ready-to-run cloud patterns, Spring Cloud can help with service discovery, load-balancing, circuit-breaking, distributed tracing, and monitoring. It can even act as an API gateway.



Here is what a typical microservice architecture. For example, consider this microservice architecture for a simple shopping cart application. It has different services like **User service**, **Order service**, and **Payment service**, and these are the independent and loosely coupled services in the microservices projects.



API Gateway: Spring Cloud Gateway is a part of the Spring Cloud ecosystem and is an API gateway built on top of Spring Framework and Spring Boot. An API gateway is a server that acts as an intermediary between clients (such as web or mobile applications) and the microservices that provide various functionalities. It is a crucial component in a microservices architecture.

Spring Cloud Config Server: Spring Cloud Config Server is another component of the Spring Cloud ecosystem. It provides a centralized configuration management solution for microservices-based applications. In a microservices architecture, you typically have multiple instances of services running on different servers, and managing their configurations can become challenging.

Circuit Breaker: A Circuit Breaker is a software design pattern used in distributed systems to handle failures and prevent cascading failures across interconnected micro services. It is an important component in building resilient and fault-tolerant microservices architectures.

Implementation of Micro Services:

1. Primarily we will Create 3 Spring Boot applications as per our architecture. After Creation of Micro Services, we will integrate those with other components as defined in Architecture.

user :

Port : 8001

Context Path: /user

order :

Port: 8002

Context Path: /order

payment:

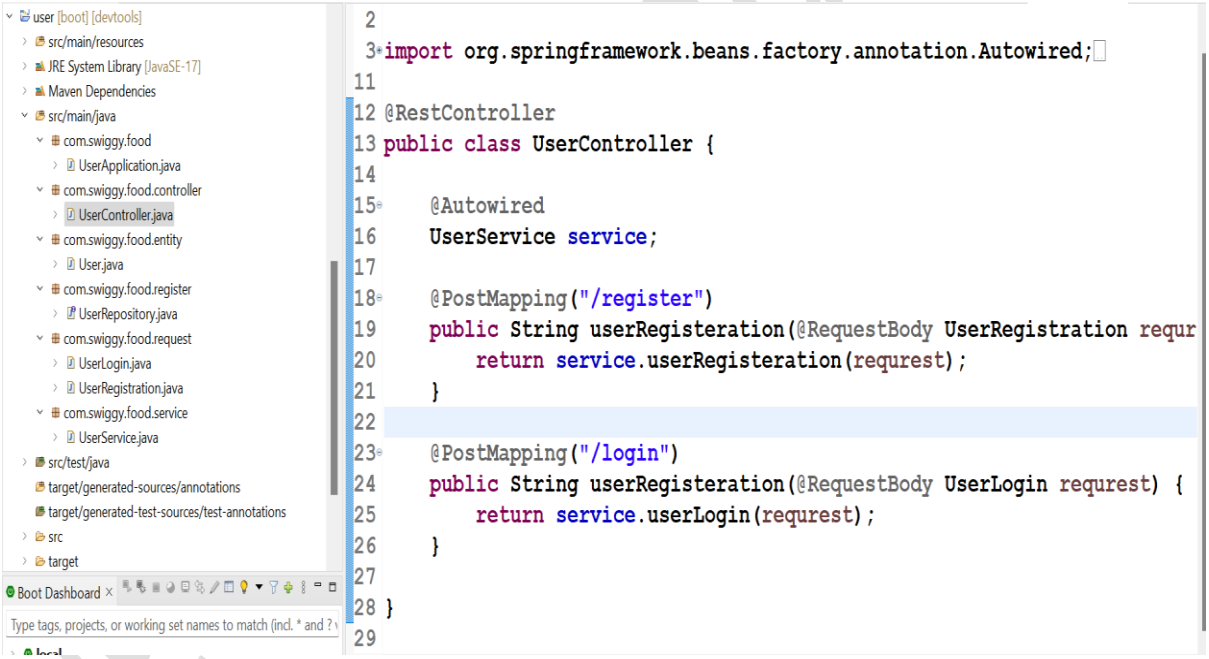
Port: 8003

Context Path: /payment

User Micro Service

Please implement a Spring Boot Web Application for User Functionalities.

Server Port : 8001
Context Path: /user
Application Name: user-service



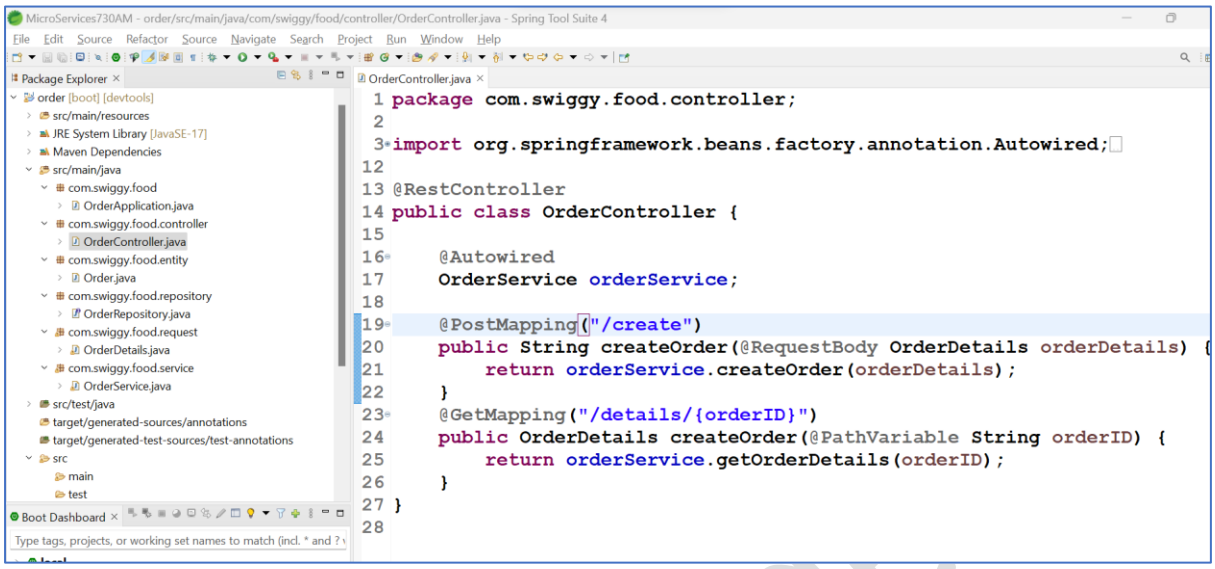
Similarly Implement two other Spring Boot Web Applications.

Order Micro Service :

order :

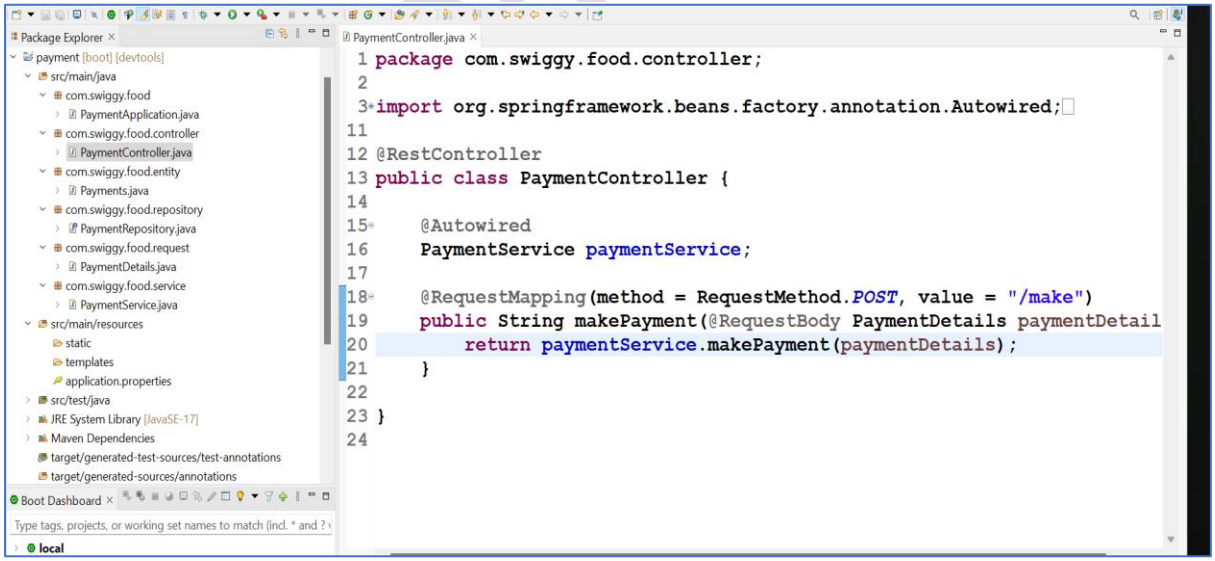
Server Port: 8002

Context Path: /order
Application Name: order-service



Payment Micro Service :

payment:
Server Port : 8003
Context Path : /payment
Application Name: payment-service



Now We are ready with 3 Micro Services, Let's integrate with Micro Services Architecture. As part of MicroServices Architecture, primarily we should implement API Gateway application.

API Gateway:

Spring Cloud Gateway is a library for building API gateways on top of Spring and Java. It provides a flexible way of routing requests based on a number of criteria, as well as focuses on cross-cutting concerns such as security, resiliency, and monitoring.

Here are some of the key features of Spring Cloud Gateway:

Routing: Spring Cloud Gateway can route requests to different microservices based on a variety of criteria, such as the request path, the HTTP method, or the request headers.

Filtering: Spring Cloud Gateway can filter requests before they are routed to the microservices. This can be used to add security, logging, or other functionality.

Resiliency: Spring Cloud Gateway can provide resilience to your microservices by using circuit breakers and other mechanisms. This can help to prevent your microservices from becoming unavailable if they are overloaded or experiencing errors.

Monitoring: Spring Cloud Gateway can be monitored using the Spring Boot Actuator. This allows you to track the performance of your gateway and the microservices that it routes to.

Here are some of the benefits of using Spring Cloud Gateway:

Ease of use: Spring Cloud Gateway is easy to use, even for developers who are not familiar with API gateways.

Flexibility: Spring Cloud Gateway is very flexible and can be used to route requests in a variety of ways.

Performance: Spring Cloud Gateway is performant and can handle a high volume of requests.

Reliability: Spring Cloud Gateway is reliable and can help to prevent your microservices from becoming unavailable.

Here are some of the drawbacks of using Spring Cloud Gateway:

Complexity: Spring Cloud Gateway can be complex to configure, especially for large and complex applications.

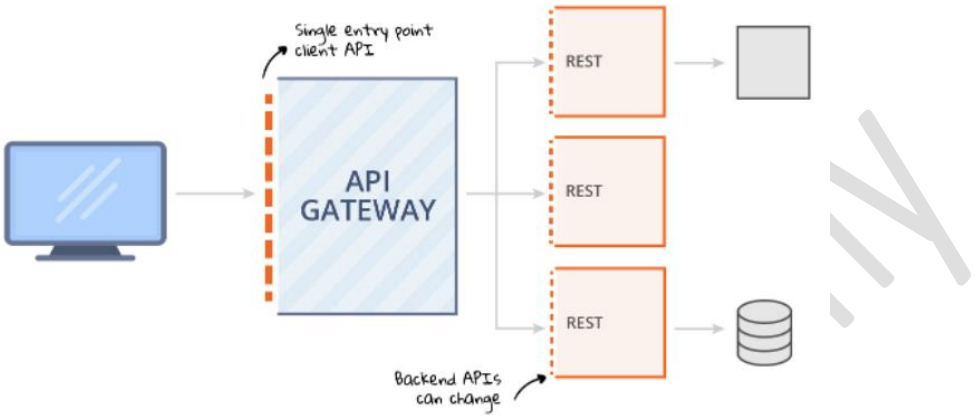
Performance: Spring Cloud Gateway can have a negative impact on the performance of your microservices, especially if you are using a lot of filters.

Security: Spring Cloud Gateway does not provide any security out of the box. You will need to implement your own security mechanisms.

Overall, Spring Cloud Gateway is a powerful and flexible API gateway that can be used to build scalable and reliable microservices applications. However, it is important to be aware

of the potential drawbacks of using Spring Cloud Gateway before you decide to use it in your application.

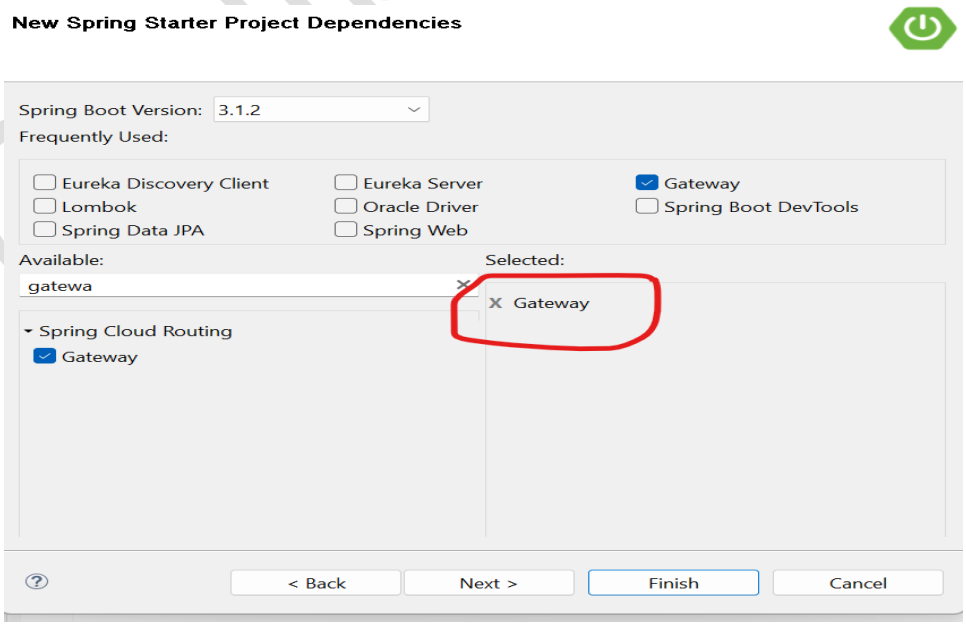
An API Gateway acts as a single entry point for a collection of microservices. Any external client cannot access the microservices directly but can access them only through the application gateway.



We are going to configure our three Miro Services with Gateway application i.e. All clients Request and response of our three MicroServices will be accessed via Gateway application instead of providing direct access. With help of API gateway application, we are bringing all our micro services under one port number instead of individual ports.

How to Create Spring Cloud API Gateway:

While creating Spring Boot Gateway Application, From Sprint Boot Starters add **Gateway starter** and finish**.**



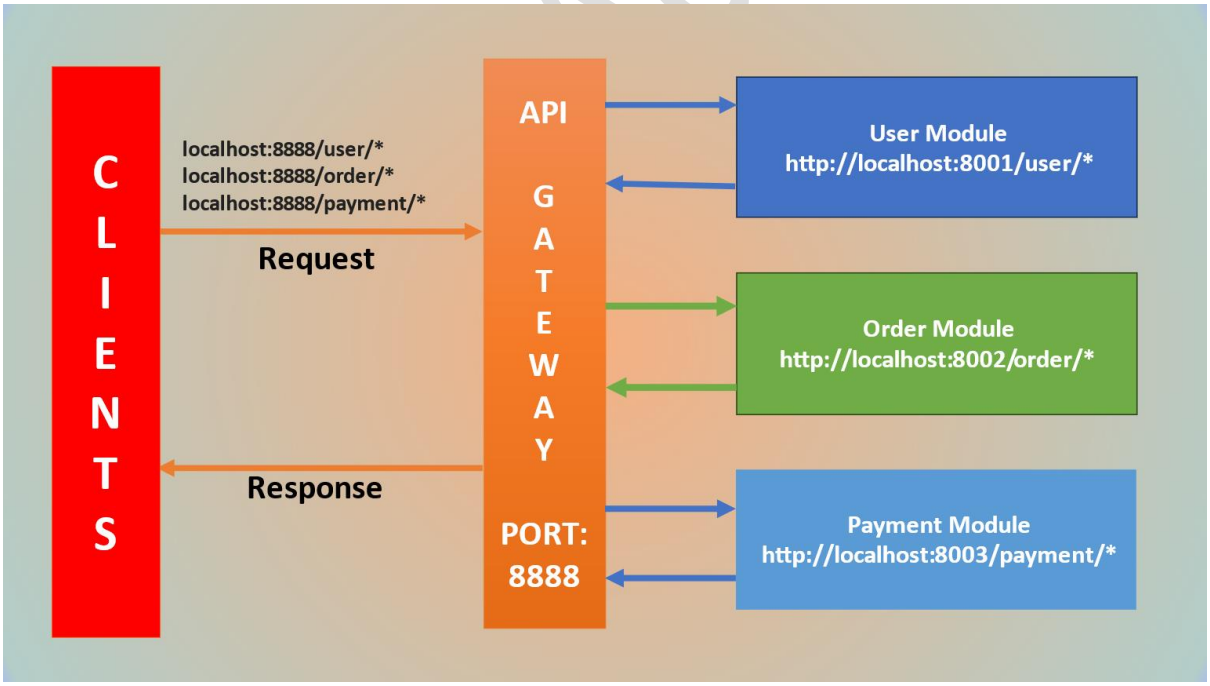
Now our Spring Boot API gateway application is Ready, So Let' configure our micro services with Gateway.

Micro Services individual Port Numbers and Context paths what we created in previous steps.

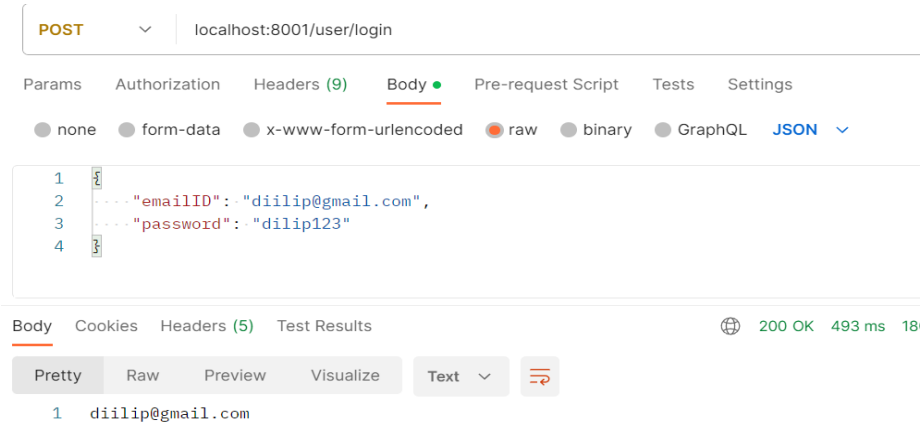
users-mgmt	**:**
	Port : 8001
	Context Path: /user
order-mgmt	**:**
	Port: 8002
	Context Path: /order
payment-mgmt	**:**
	Port: 8003
	Context Path: /payment

How to Configure Micro Services with Gateway application?

Following image explains how External HTTP clients will access our Micro Services endpoints via Gateway application. So here we should configure our micro services information inside Gateway application. This Process will be called as **Routing Configuration.**



Generally, If we want to access REST services User Micro Service, then we will access with User Micro Service Port number **8001** in URL as followed.



If we follow similar approach, then for every individual micro Service access we should use individual port numbers, it is a tedious process to HTTP clients to manage multiple port numbers while they are integrating or consuming our all MicroServices REST Services. In this scenario, we should make sure only port number being used across multiple micro services access from our application.

So Now we will configure our micro services with Gateway application, in a way as our Micro Services are accessible by only Gateway port number instead of their individual port numbers.

What is Routing in Gateway?

In Spring Boot API Gateway, routing is the process of determining which microservice to send a request to. The gateway uses a routing table to store information about the microservices it is connected to and the paths between them. When a request arrives at the gateway, the gateway looks up the destination path in its routing table and determines the microservice that the request should be forwarded to.

Routing is an important part of Spring Boot API Gateway. It allows the gateway to route requests to the correct microservices, which can improve the performance, reliability, and security of network communication.

There are two main ways to configure routing in Spring Boot API Gateway:

1. Declaratively:

This is done by configuring the routing table in the **application.properties** file. The routing table can be configured using a number of different predicates and filters.
Here is an example of how to configure routing declaratively in Spring Boot API Gateway:

```
spring.cloud.gateway.routes[0].id=user-route
spring.cloud.gateway.routes[0].uri=http://localhost:8080/users
spring.cloud.gateway.routes[0].predicates=Path=/api/users/**
```

These properties creates a route with the ID "**user-route**" that routes requests of the **/api/users/**** path to the microservice running at **http://localhost:8080/users**

2. Programmatically:

This is done by creating a **RouteLocator** bean and registering it with the Spring Boot application. The **RouteLocator** bean is responsible for creating and managing the routing table. Here is an example of how to configure routing programmatically in Spring Boot API Gateway:

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
      return builder.routes()
     .route("user-route", r -> r.path("/api/users/**")
     .uri("http://localhost:8080/users"))
     .build();
}
```

This code creates a route with the ID "**user-route**" that routes requests of the **/api/users/**** path to the microservice running at **http://localhost:8080/users**

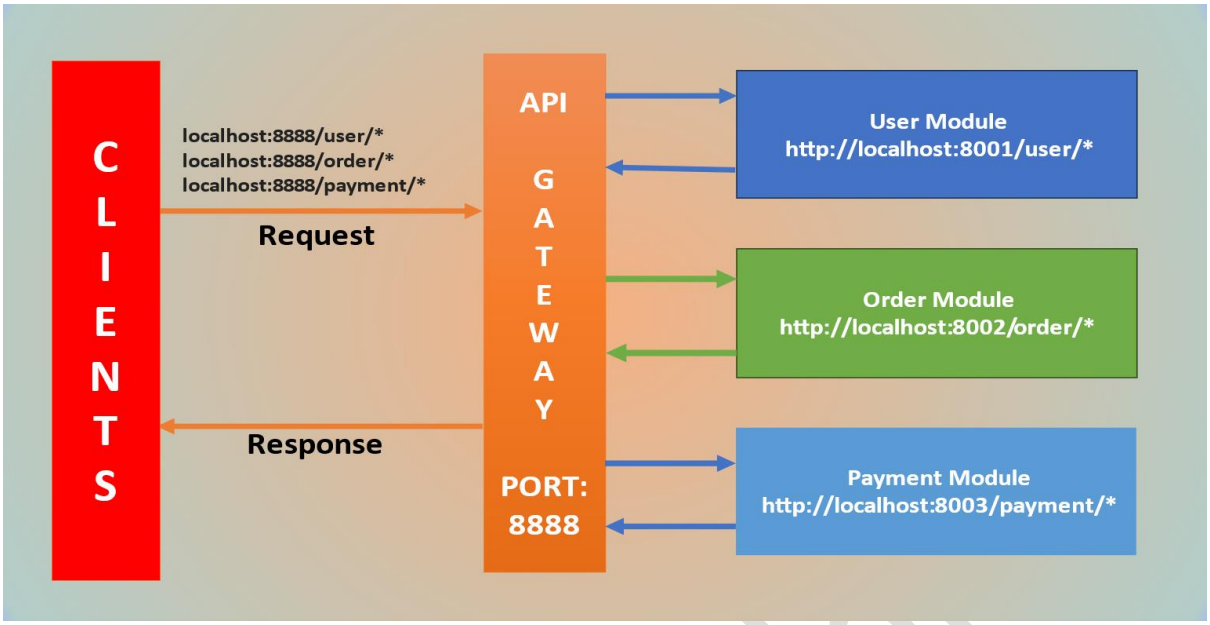
Now Let's configure our three micro services with Gateway Declaratively i.e. in Properties file Level.

MicroServices with Gateway Configuration:

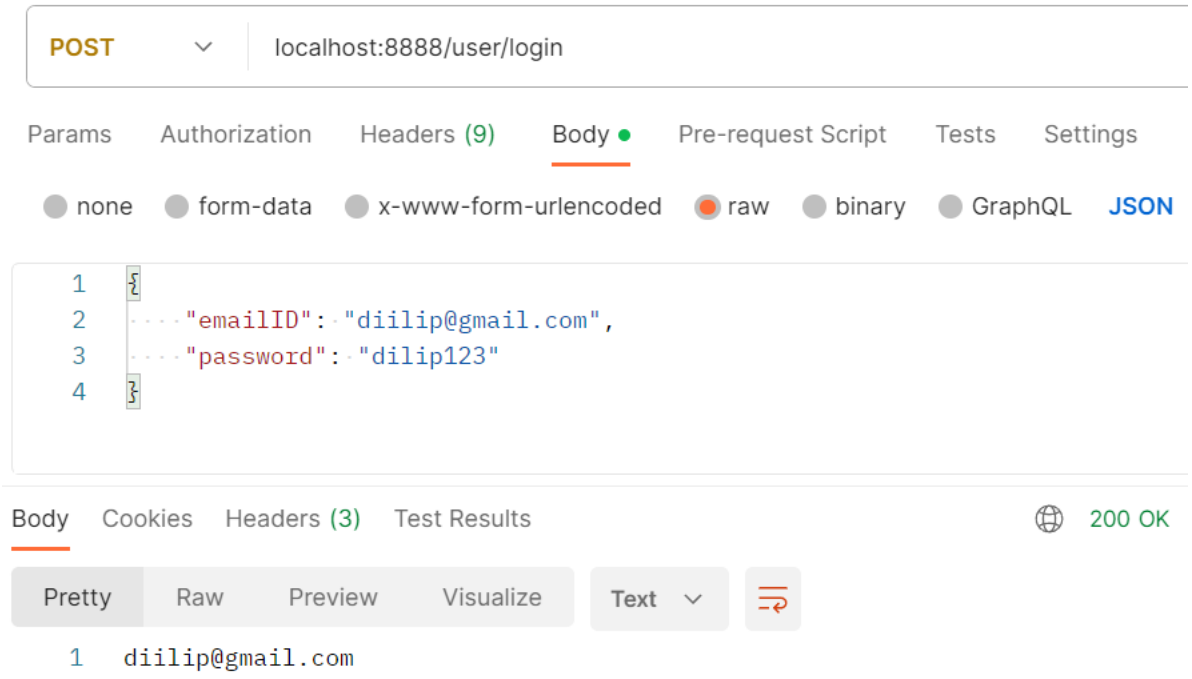
```
#Gateway Application Details
server.port=8888
spring.application.name= swiggy-gateway

#user api mapping
spring.cloud.gateway.routes[0].id=user
spring.cloud.gateway.routes[0].uri=http://localhost:8001
spring.cloud.gateway.routes[0].predicates[0]=Path=/user/**

#order api mapping
spring.cloud.gateway.routes[1].id=order
spring.cloud.gateway.routes[1].uri=http://localhost:8002
spring.cloud.gateway.routes[1].predicates[0]=Path=/order/**

#payment api mapping
spring.cloud.gateway.routes[2].id=payment
spring.cloud.gateway.routes[2].uri=http://localhost:8003
spring.cloud.gateway.routes[2].predicates[0]=Path=/payment/**
```

With above Configuration, we can access our MicroServices with below URL patterns i.e. via Gateway Application.

localhost:8888/user/**
localhost:8888/order/**
localhost:8888/product/**



Testing: Testing User Login Endpoint with gateway Port.



Now Our Gateway Application Is Ready.

Service Registry and Discovery:

Service registry and discovery are two important concepts in microservices architecture. Service registry is a centralized repository that stores information about all the microservices in a system. This information includes the microservice's name, address, and port number. Service discovery is the process of finding the location of a microservice based on its name.

Spring Boot provides support for service registry and discovery using the Netflix Eureka project. Eureka is a service registry that provides a REST API for registering and discovering microservices.

When a Spring Boot application registers with Eureka, it provides the following information:

- The application's name
- The application's address
- The application's port number
- The application's health status

Other Spring Boot applications can discover the location of a microservice by querying Eureka. The Eureka REST API provides a number of different endpoints for querying the service registry.

Service registry and discovery are essential for microservices architecture. They allow microservices to find each other and communicate with each other. This makes it possible to build large, complex applications that are composed of many small, independent microservices.

Here are some of the benefits of using service registry and discovery in Spring Boot microservices:

Scalability: Service registry and discovery make it easy to scale microservices. When you need to add more instances of a microservice, you simply register the new instances with Eureka. Eureka will then distribute requests to the new instances.

Fault tolerance: Service registry and discovery make it easy to handle failures in microservices. If a microservice fails, Eureka will remove it from the service registry. This will prevent other microservices from trying to communicate with the failed microservice.

Load balancing: Service registry and discovery can be used to load balance requests across multiple instances of a microservice. This can improve the performance of microservices by distributing requests evenly across the available instances.

Service Registry:

A Service Registry is a centralized directory where microservices can register themselves and provide metadata about their location and capabilities. In other words, it acts as a database of all available services in the system. Each microservice registers its network location (IP address and port) and any other relevant information (e.g., service name, version, health status) with the registry.

In Spring Boot, **Netflix Eureka** is a popular service registry implementation. Eureka provides a server component to run the registry and a client library to enable microservices to register themselves and discover other services.

Service Discovery:

Service Discovery is the process by which a microservice finds the network location (IP address and port) of other services it wants to communicate with. Instead of hardcoding the locations of other services, microservices can dynamically discover them through the service registry.

In Spring Boot, the Eureka client library is used to implement service discovery. When a microservice starts up, it registers itself with the Eureka server. Likewise, when a microservice needs to communicate with another service, it queries the Eureka server to get the location information of the required service.

Step-by-step explanation of how Service Registry and Discovery work in Spring Boot:

Service Registration:

1. Each microservice (e.g., Service A, Service B) running in the system registers itself with the Service Registry (Eureka Server).
2. The registration typically occurs when a microservice starts up. It sends a registration request to the Eureka server, providing its metadata (service name, IP address, port, health status, etc.).
3. The Eureka server maintains a registry of all the registered services and their metadata.

Service Discovery:

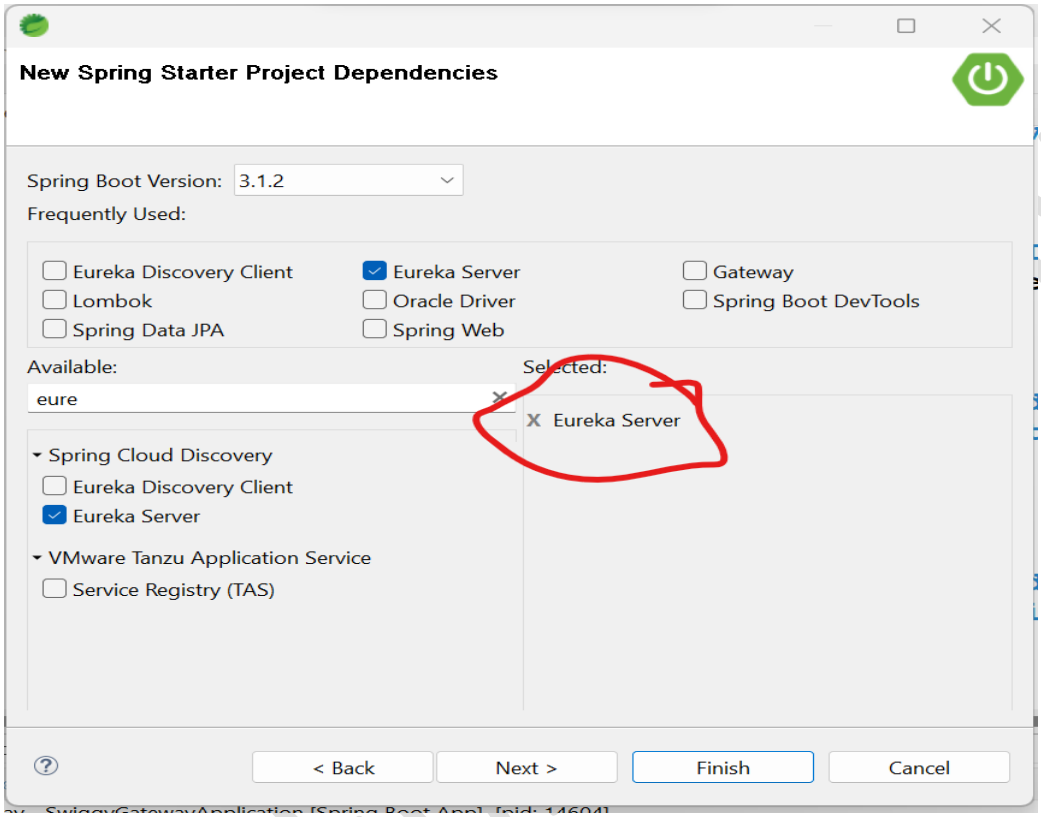
- When a microservice (e.g., Service A) needs to communicate with another service (e.g., Service B), it uses the Eureka client library to perform service discovery.
- The Eureka client in Service A queries the Eureka server to get the network location (IP address and port) of Service B.
- The Eureka server responds with the location information for Service B.
- Service A can then use this location information to communicate with Service B over the network.

Spring Boot provides excellent integration with Eureka through the **spring-cloud-starter-netflix-eureka-client** dependency for client-side service discovery and the **spring-cloud-starter-netflix-eureka-server** dependency for setting up the Eureka server.

It's worth noting that while Eureka is one of the popular choices for Service Registry and Discovery in Spring Boot, there are other alternatives like Consul, ZooKeeper etc..

Create Our Own Eureka Server for Services Registry:

Create SpringBoot Application With Eureka Server Dependency as followed.



- **After Application Created, Please add an annotation with @SpringBootApplication in main Spring Boot Application class, we need to add @EnableEurekaServer annotation.**
- **The @EnableEurekaServer annotation is used to make your Spring Boot application acts as a Eureka Server.**

```
package com.swiggy.eureka;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class SwiggyEurekaServerApplication {

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

```
}
```

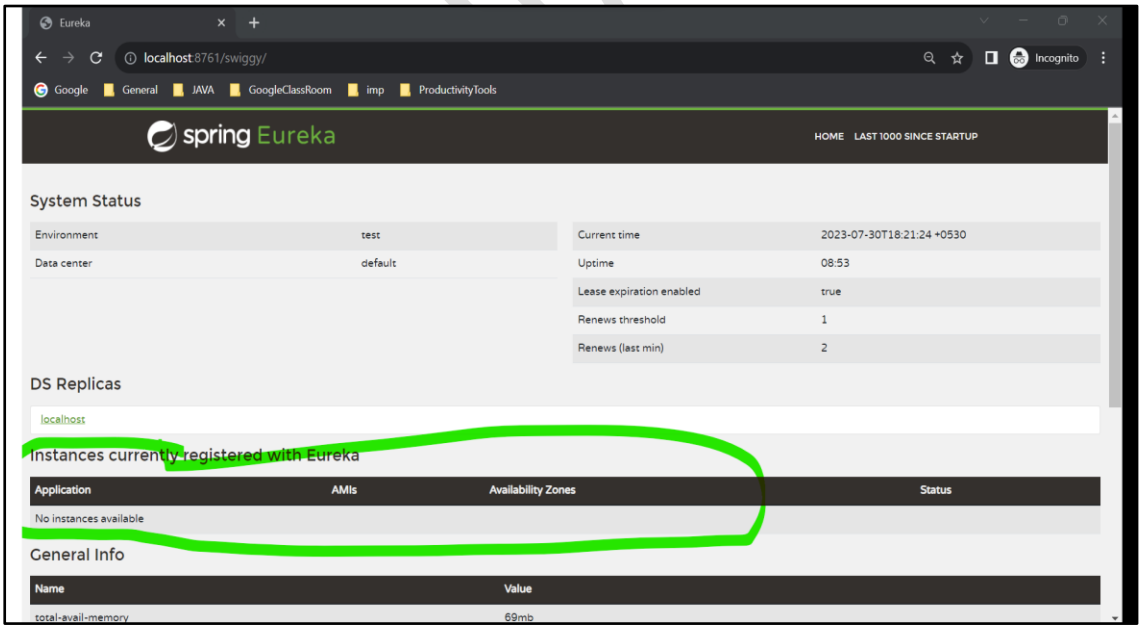
- By default, the Eureka Server registers itself into the discovery. You should add the below given configuration into your application.properties file or application.yml file.

```
server.port=8761
server.servlet.context-path=/swiggy

#stopping self registration as a client again
eureka.client.register-with-eureka=false
#Fetching other Micro Services registration
eureka.client.fetch-registry=false
```

Now you can start Eureka Server Application. It will be started on port 8761 always but we can provide any context path.

Once Spring Cloud Eureka Server is started , Please access below URL for Discovered Services of MicroServices.

Eureka Server URL : **http://localhost:8761/swiggy/**



In Highlighted section we are seeing as **No Instances available.**

Now, It's time to make our Micro Services as Discovery Clients to register with Eureka Server.

In our MicroServices Project, we should make 3 Micro Services as Eureka Clients.

- **User**
- **Order**
- **Payment**

Step 1 : We should add a dependency in our existing Micro Services Applications or add while creating application.

```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
	<version>2022.0.3</version>
</dependency>
```

Step 2 : After adding it We should add an annotation in every MicroService Main Spring Boot Application class level a followed.

```
package com.swiggy.food;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
public class UserApplication {

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

Step 3: Add Eureka Server Details in Client application Properties file to register with it.

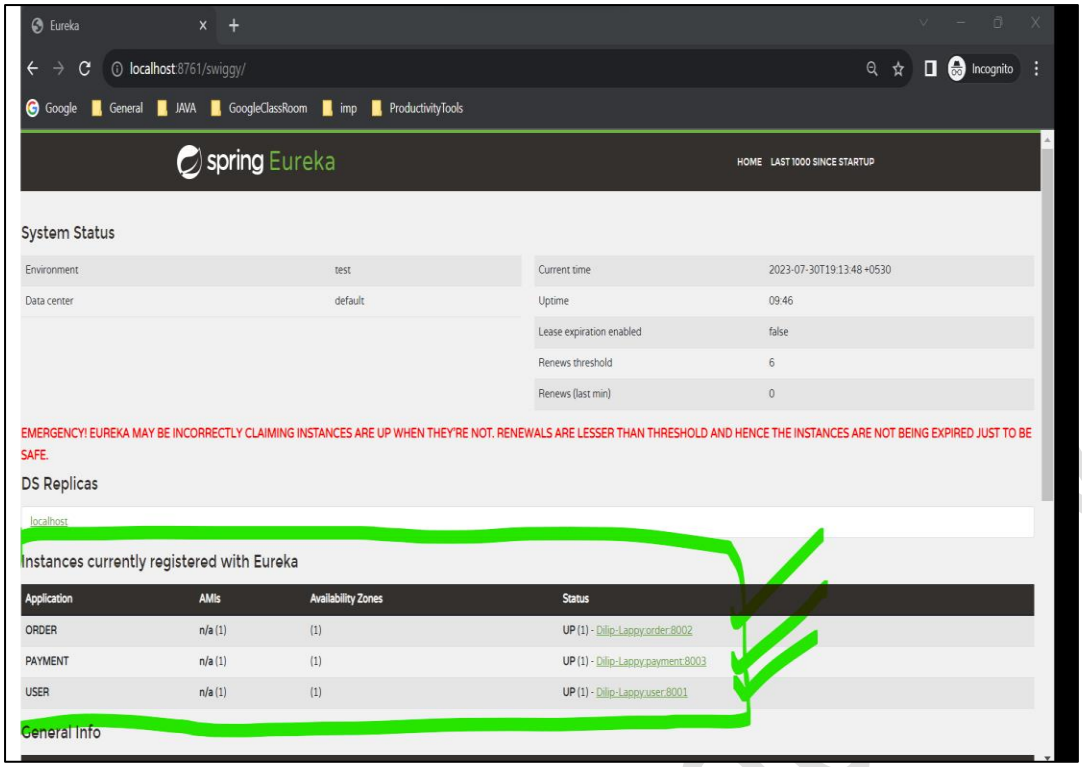
```
#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka
```

NOTE: These are basic steps for any Eureka client application. So please repeat same Steps for all Micro Services User, Order and Payment.

NOTE: Please start Eureka Server Application first and then Clients Applications.

- **Start Eureka Server Application**
- **Now Start User, Order and Payment One by One.**

Now Access Eureka Server URL again : http://localhost:8761/swiggy/



Now In Eureka Server our 3 MicroService Applications are Registered and same Details We can find above.

API Gateway Integration with Eureka Server:

With integration of Gateway application and Eureka Server, we can achieve auto routing to our Micro Service Instances. With the Discovery Locator enabled in API Gateway, you do not have to manually configure the routes of individual micro services unless absolutely needed. The way API Gateway knows which Eureka Service to route the incoming request.

There are several advantages of connecting Eureka server with gateway in microservices.

Centralized service discovery: Eureka server provides a centralized service discovery registry that allows the gateway to discover the microservices that are available. This makes it easy for the gateway to route requests to the appropriate microservices.

Load balancing: Eureka server can also be used to load balance requests across multiple instances of a microservice. This can help to improve the performance of the microservices architecture.

Fault tolerance: If a microservice instance fails, Eureka server will remove the instance from the registry. The gateway will then be able to route requests to the remaining instances of the microservice. This helps to ensure that the microservices architecture remains available even if some of the microservice instances fail.

Monitoring: Eureka server can also be used to monitor the health of the microservices. This information can be used to identify microservices that are not performing well or that are experiencing errors.

Overall, connecting Eureka server with gateway in microservices can provide a number of advantages, including centralized service discovery, load balancing, fault tolerance, and monitoring.

Adding Discovery Client Functionalities to Gateway Application:

As usual we will follow same Eureka Client Process as followed.

Step 1 : We should add a dependency in our existing Micro Services Applications or add while creating application.

```
<dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2022.0.3</version>
</dependency>
```

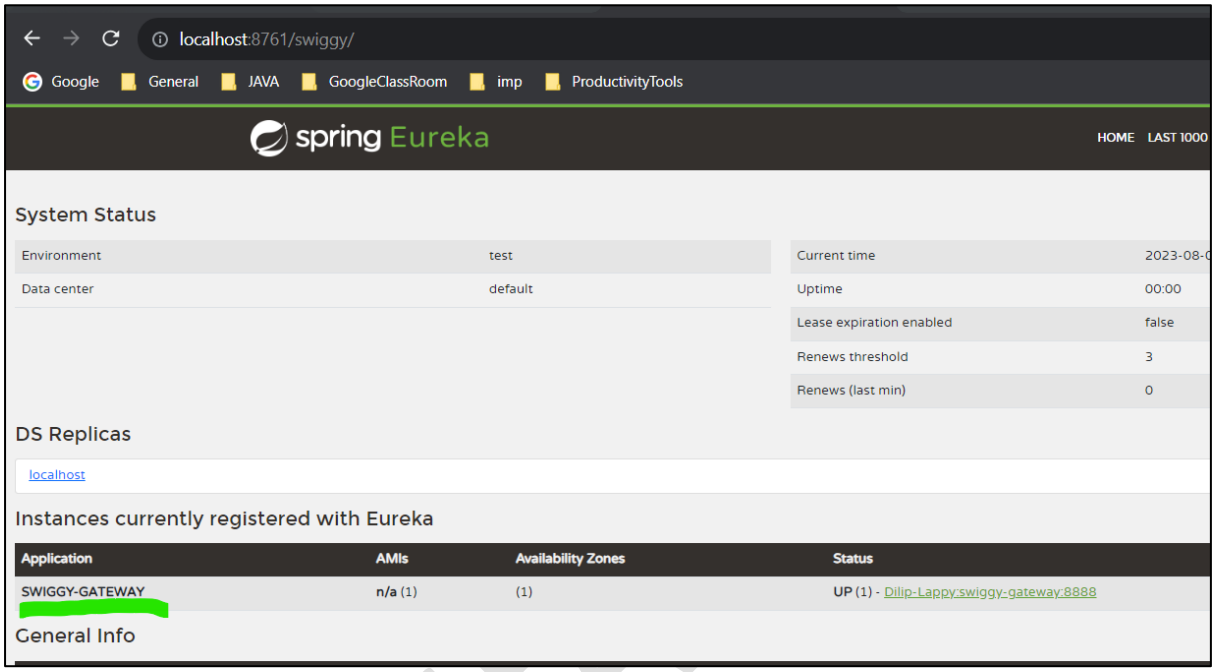
Step 2 : After adding it We should add an annotation in every Gateway Main Spring Boot Application class level a followed.

```
package com.swiggy.gateway;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@EnableDiscoveryClient
@SpringBootApplication
public class SwiggyGatewayApplication {

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

Step 3: Add Eureka Server Details in application.properties file to register with it.

```
#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka
```

With This changes in Gateway, It's ready to register and discoverable. Please Start Eureka Server and after Gateway application. We can find Gateway application details in Eureka server.



Now We can trigger Micro Services with Gateway and Eureka Server. To access we should follow below URL format.

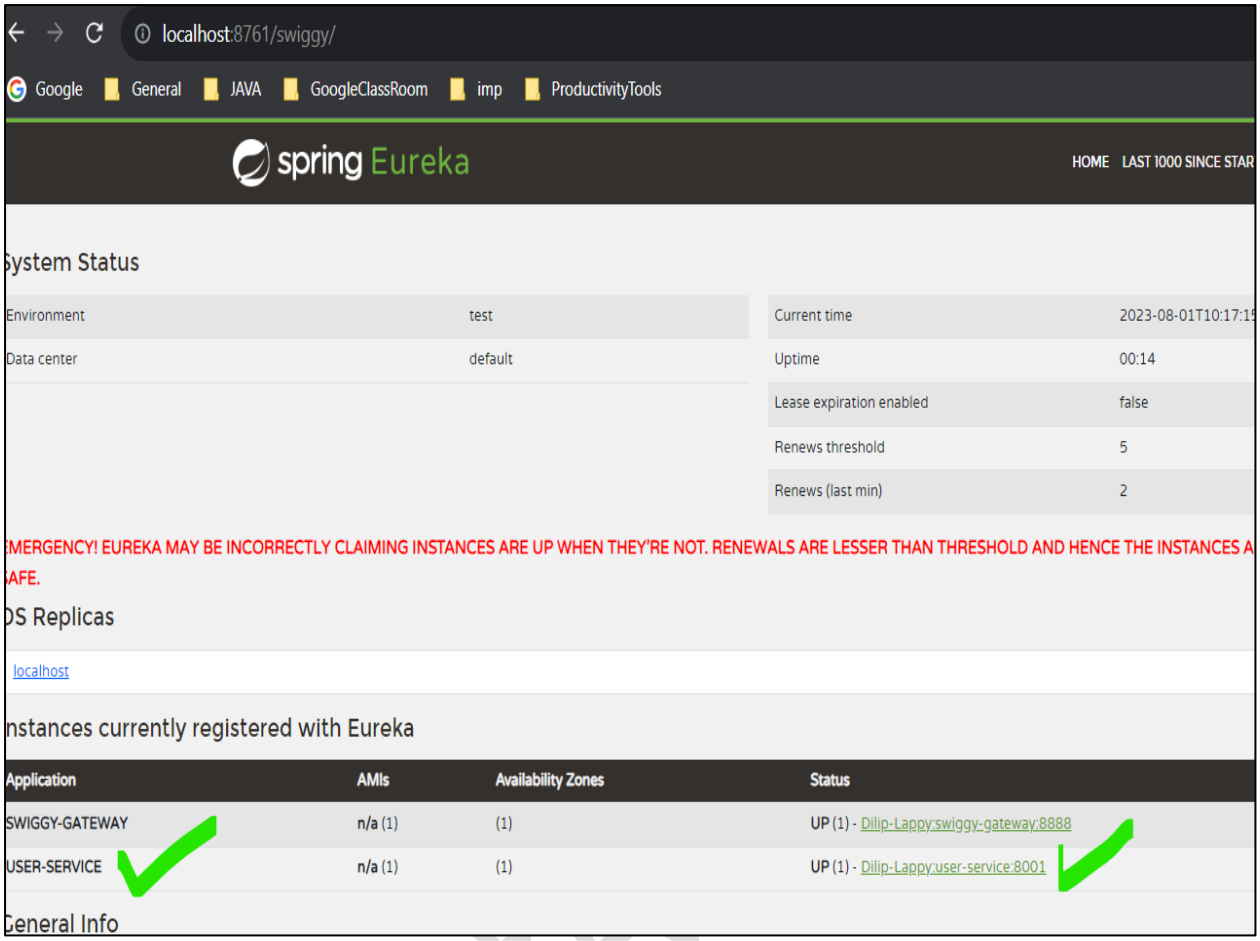
http://{ApiGatewayHost}:{port}/{EurekaServiceId}/{ActualEndpoint}

What is Eureka Service Id: The Micro Service Application Name what we given individually In every **application.properties** file. With those names every Eureka client application will be registered with Eureka Server.

For Example, For User MicroService provided name as

spring.application.name=user-service

After starting User Service, open Eureka server home page and we can see details as followed.



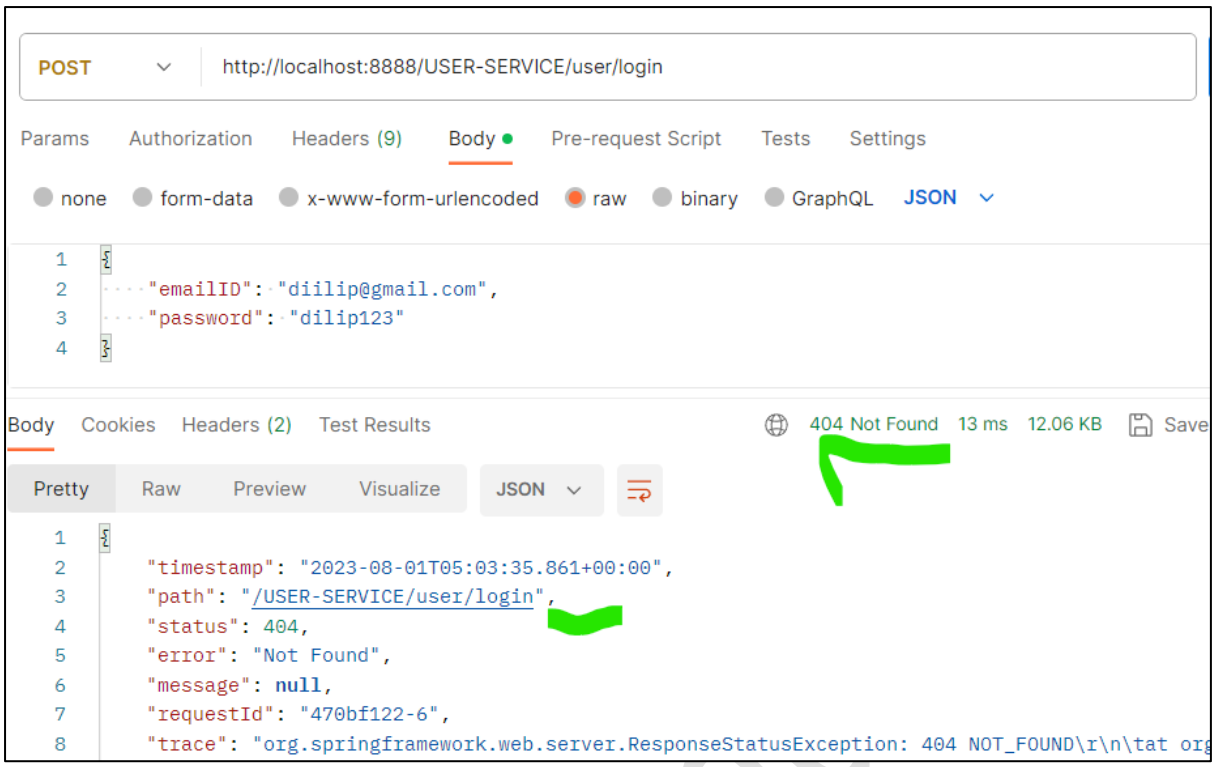
NOTE: Eureka sever will maintain Service ID'S in UPPERCAE always by default as shown in above image.

Similarly every Micro Service, will be provided with a name and if it registered with Eureka server, we will consider it as Eureka Service Id.

Access User Service With Gateway and Eureka Service ID:

Gateway : localhost:8888
Service Id : USER-SERVICE
Actual URI: /user/login

URL: **http://localhost:8888/USER-SERVICE/user/login**



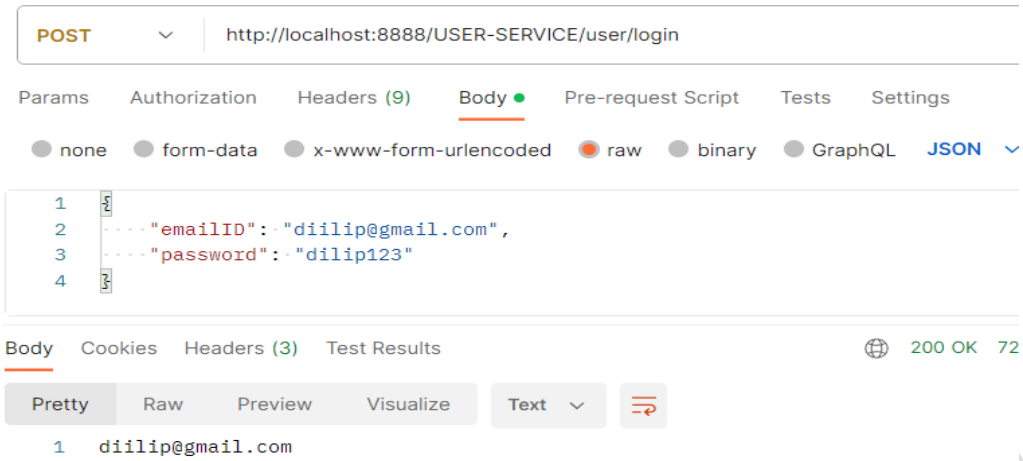
We got as **Not Found**. That means, by default Eureka server will not allow locating Micro Service details to route request internally.

To enable this functionality of locating Micro Services Information from Eureka Server we should add a property in **Gateway** application. With the Discovery Locator enabled in API Gateway, you do not have to manually configure the routes unless absolutely needed.

Add Below Property in Gateway Application Properties file.

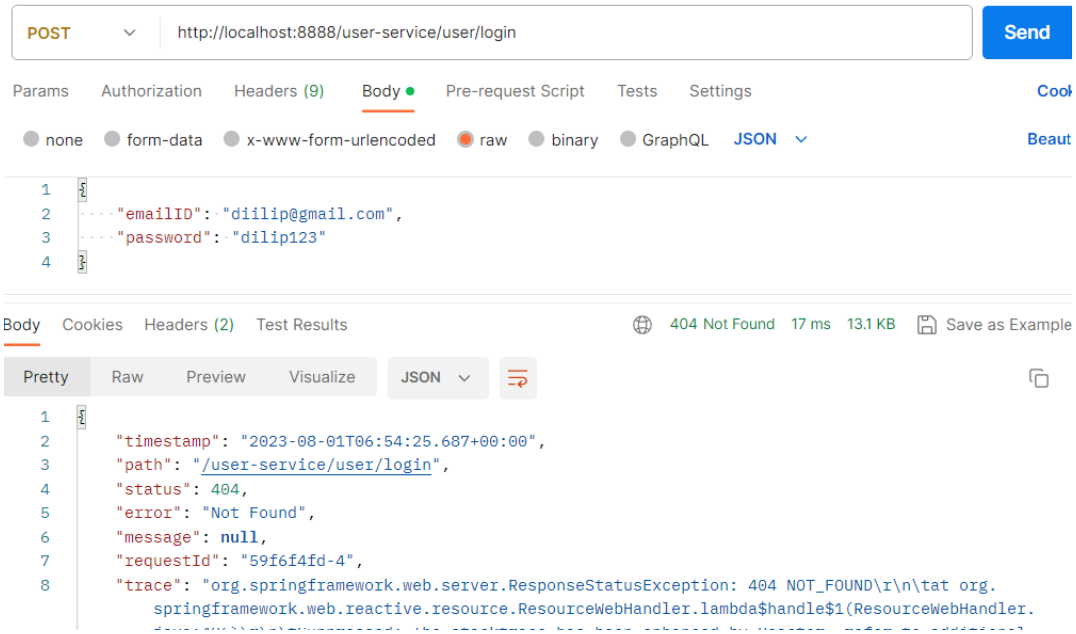
```
spring.cloud.gateway.discovery.locator.enabled=true
```

Restart Gateway Application and try to access same URL again.



Now Eureka Sever allowing Gateway Application for locating micro service details based on Eureka Server Registered **Service ID** provided in URL. Now with that information, Gateway routes current request to User MicroService instances.

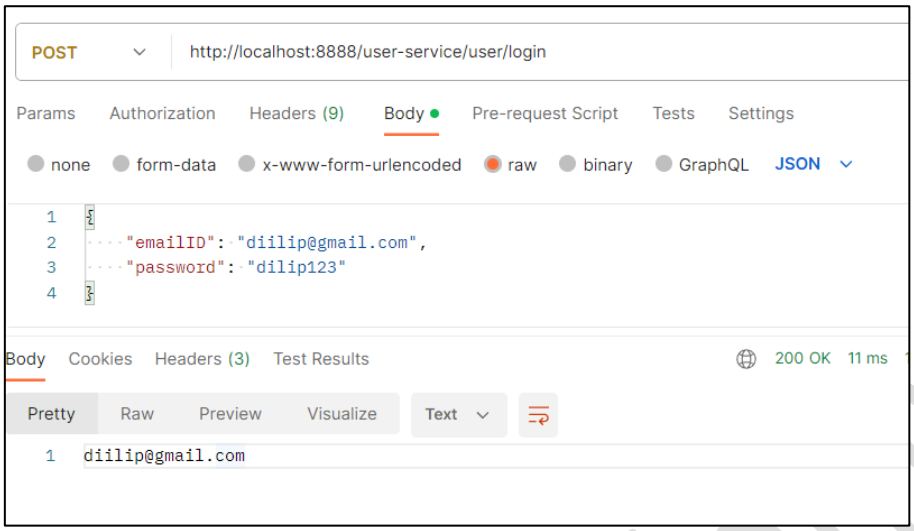
If we observe URL , Service ID given in Upper case. If we try to give lower case what will happen?



That means Eureka Server allowing only with Upper Case Service Id of A Micro Service always from Gateway Application by default. If we want to send service-id in Lower Case then we should add a property in Gateway allocation properties file.

spring.cloud.gateway.discovery.locator.lower-case-service-id=true

Now Access URL with Lower Case Service ID.



After lower case Property enabled, we can access only with Lower case Service Id but Not With Upper Case ID.

Gateway Integration with Eureka Server Advantages:

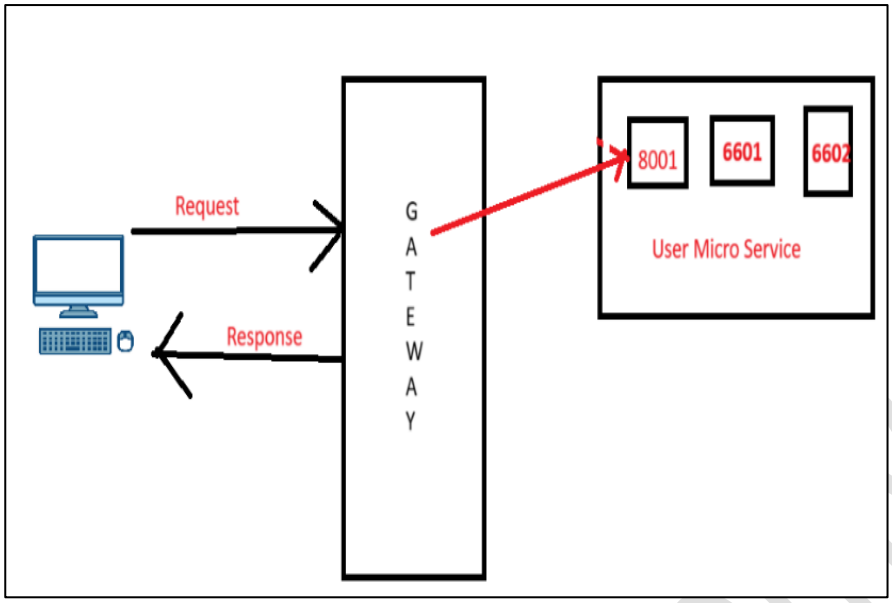
Case 1: Gateway Without Eureka Server:

Use Case : Assume User MicroService running on multiple port numbers like 8001 , 6601, 6602 etc..

Before Eureka Server implementation, we provided routing configuration in Gateway application for User MicroService to make sure Requests are always entered via Gateway to User Application.

```
spring.cloud.gateway.routes[0].id=user
spring.cloud.gateway.routes[0].uri=http://localhost:8001
spring.cloud.gateway.routes[0].predicates[0]=Path= /user/**
```

From above configuration, Request will be transferred to an instance running on port 8001 always by Gateway. **Then what about other instances running on 6601, 6602 etc. Ports. Those are not used anytime.**



To utilize all instances of a Micro Service application, then you have to do multiple routing configurations in Gateway for induvial port numbers. But this not looking good because in real time we may increase or decrease number of instances of Micro service application i.e. auto scaling.

Case 2: Gateway With Eureka Server Integration :

Use Case: Assume User MicroService running on multiple port numbers like 8001 , 6601, 6602 etc..

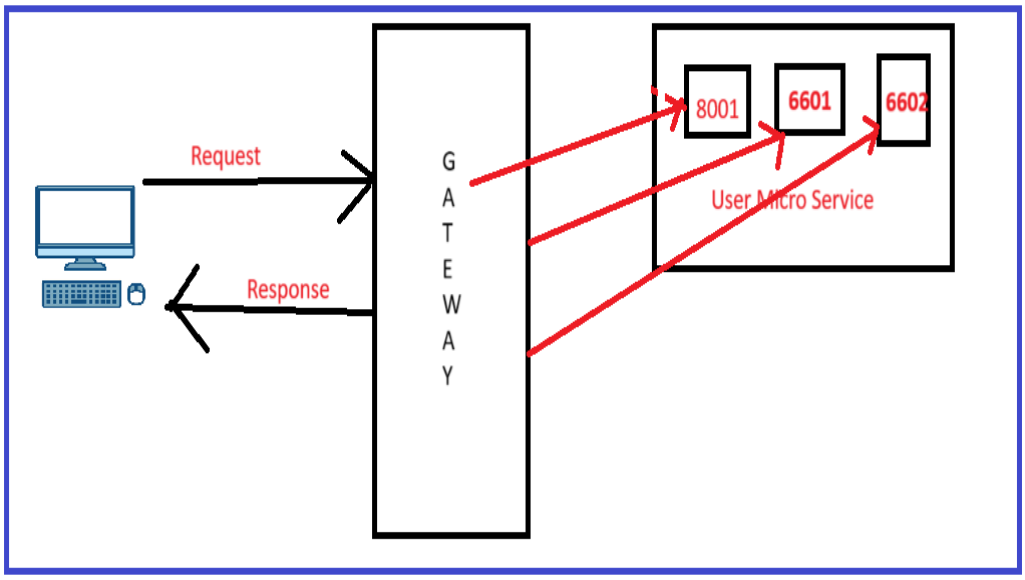
Now whenever we sent a request to User Micro Service application via Gateway with Eureka Service Id as followed.

URL : http://localhost:8888/USER-SERVICE/user/login

Now Gateway application will locate all details of User MicroService application from Eureka Server like what are current running instances and port numbers. These Details are registered with Eureka Server while every Micro service instance we started.

In this Case, Gateway gets data of User MicroService application current running instances and port numbers like **8001, 6601, 6002** etc.. Now Gateway dynamically/automatically routes the Requests to less load instances internally. This auto routing will be taken care by load Balancer by default in Gateway.

Conclusion: Same functionality will be applicable for all Micro Services application in a project. Over all achievement now here is no manual configuration of routing in Gateway i.e. we can remove routing logic from Gateway applications for all Micro Services.



MicroServices Communication:

Feign Clients and **RestTemplate** both are Java libraries commonly used for making HTTP requests in microservices applications. They provide a way for your application to communicate with external services, APIs, or other microservices over the network. However, they have different approaches and features, which I'll explain below:

RestTemplate: RestTemplate is a synchronous HTTP client that is part of the Spring Framework. It provides a straightforward way to make HTTP requests to external services or APIs. You can use RestTemplate to send HTTP GET, POST, PUT, DELETE, and other types of requests.

Key features of RestTemplate:

Synchronous: RestTemplate operates in a synchronous manner, meaning that when you make a request, your application waits for the response before continuing execution.

Blocking: Since RestTemplate is synchronous, making multiple requests concurrently can lead to blocking behaviour and potentially reduced performance.

Flexibility: RestTemplate offers a variety of methods to customize request headers, parameters, and handling of responses.

Widely Used: RestTemplate has been a popular choice for making HTTP requests in Spring-based applications.

Feign Client: Feign is a declarative web service client also **developed by Spring Cloud**. It simplifies the process of making HTTP requests by allowing you to define interfaces with annotated methods that describe the API endpoints you want to call. Feign dynamically

generates the implementation for these interfaces, abstracting away the actual HTTP request details.

Key features of Feign Client:

Declarative: With Feign, you define an interface with annotations, and Feign generates the implementation automatically. This promotes a more concise and clean approach to API consumption.

Integration with Service Discovery: Feign can integrate seamlessly with service discovery mechanisms, such as Netflix Eureka, making it easy to call other microservices without hardcoding URLs.

Load Balancing: Feign can work in conjunction with load balancers, distributing requests across multiple instances of a service.

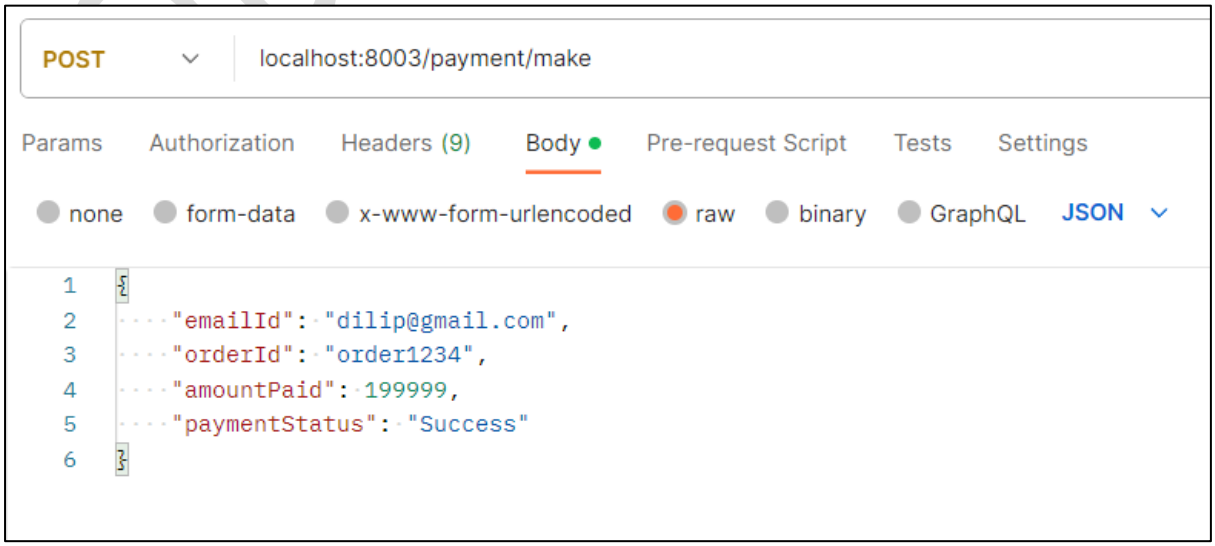
Integration with Spring Cloud: Feign is often used in Spring Cloud-based applications as part of the broader microservices ecosystem.

In summary, both Feign Client and RestTemplate are viable options for making HTTP requests in a microservices architecture. The choice between them depends on your project's specific requirements, development style, and whether you prefer a more declarative approach (Feign) or a more traditional, programmatic approach (RestTemplate). Additionally, consider factors like integration with service discovery, load balancing, and asynchronous support when making your decision.

Earlier As Part of Spring Boot Training, We Used RestTemplate for consuming micro services. Now We will continue with **Feign Clients**.

Requirement : Consume Payment MicroService REST Services From User Services.

REST API call of Payment Service:



Consumer : User Service
Producer: Payment Services

So, Now we should make changes in User Micro Service.

How To Enable Feign Clients:

Add Dependencies: In **pom.xml** file, add necessary dependency for Feign Client

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

Enable Feign Client: In your main application class, add the **@EnableFeignClients** annotation to enable Feign Client functionality in Micro Service.

```
package com.swiggy.food;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
import org.springframework.cloud.openfeign.EnableFeignClients;

@EnableDiscoveryClient
@SpringBootApplication
@EnableFeignClients
public class UserApplication {
        public static void main(String[] args) {
                SpringApplication.run(UserApplication.class, args);
        }
}
```

Define Feign Client Interface: Create an interface that defines the API endpoints you want to call. Annotate the interface with **@FeignClient** and provide the **name of the service** you're communicating with. Also, define the methods corresponding to the endpoints you want to interact with. Similar or Looks Like to Controller Layer Endpoint Methods.

As per Payment Service, we should create Request Body class and we should define same in Feign client level.

```
package com.swiggy.food.feign.clients;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RequestMethod;
import com.swiggy.food.request.PaymentDetails;

//Service ID/Name  Producer

@FeignClient("payment-service")
public interface PaymentMicroServiceFeignCleint {

	@RequestMapping(method = RequestMethod.POST, value = "/payment/make")
	public String makePayment(@RequestBody PaymentDetails paymentDetails);

}
```

From the above Feign Client Configuration of REST service of Payment, we have Request Body class. So Create a POJO of request to pass the values with properties given in Postman.

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class PaymentDetails {
	private String emailId;
	private String orderId;
	private double amountPaid;
	private String paymentStatus;
}
```

With this we are done with Feign Client Setup for Payment Service Endpoint. If we want to trigger another endpoint of Payment Micro Service then we will add another abstract method in Feign Client Interface.

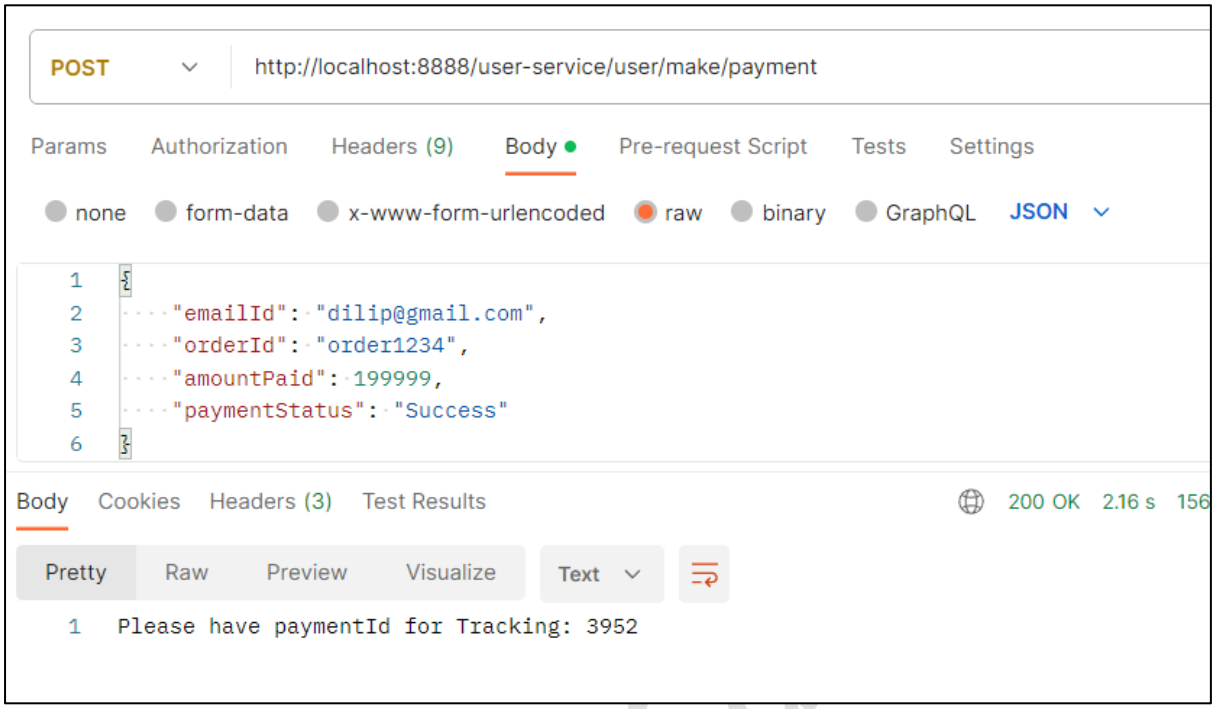
Now Test It From User Service i.e. trigger Payment Endpoint of Feign Client.

I am creating an endpoint in User Controller with Request Body and same Request data I will forward to Request Body of Payment endpoint.

4. Autowire Feign Client in Controller
5. Call Feign Client Method with Request Data

```
@Autowired
PaymentMicroServiceFeignCleint paymentClinet;

@PostMapping("/make/payment")
public String paymentStatus(@RequestBody PaymentDetails details) {
	return paymentClinet.makePayment(details);
}
```

Now Trigger Request to User Controller.



We got Response From Payment Endpoint internally and same forwarded as response to Client.

Internally Feign Client Implementation will trigger Payment Service Endpoint. Feign Client Logic internally Connected to Eureka Server with Service name what we configured with Feign Client. Eureka server will provide all available instances of Payment Mirco Services like hostname and port details. Finally Feign client contains all details of Payment Service including URI of endpoint. Internally Feign client triggers request with Full URL of endpoint.

Spring Cloud Config Server & Config Clients:

Spring Cloud **Config Server** and **Config Clients** are two components of Spring Cloud Config, which is a Spring Boot starter that provides server-side and client-side support for externalized configuration in a distributed system.

Spring Cloud Config Server and Config Clients are a powerful way to manage configuration properties in a distributed system. They make it easy to store configuration properties in a central location, and they allow you to consume configuration properties from your applications without having to store them in individual application configuration files.

The Config Server is a centralized repository for configuration properties that can be accessed by applications throughout the system. This makes it easy to manage configuration properties for all of your applications in one place, and it also makes it easy to change configuration properties without having to redeploy your applications.

The Config Client is a Spring Boot starter that allows applications to consume configuration properties from the Config Server. This means that your applications can get their configuration properties from a central location, rather than having to store them in individual application configuration files.

There are two ways to configure the Spring Cloud Config Server:

- **Git:** The Config Server can be configured to store configuration properties in a Git repository. This is the most common way to configure the Config Server, as it allows you to version your configuration properties and easily roll back to a previous version if necessary.
- **JDBC:** The Config Server can also be configured to store configuration properties in a JDBC database. This is less common than the Git configuration, but it can be a good option if you already have a JDBC database that you want to use for configuration.

Once the Config Server is configured, you can start consuming configuration properties from it in your MicroService applications. To do this, you need to add the Spring Cloud Config Client starter to your client application's pom.xml or build.gradle file. You also need to specify the URL of the Config Server in your client application's configuration.

Once you have configured the Config Client, your application will be able to consume configuration properties from the Config Server. The configuration properties will be loaded into the Spring Environment, and you can access them using the @Value annotation as well.

Here are some of the benefits of using Spring Cloud Config Server and Config Clients:

- **Centralized configuration:** All of your configuration properties can be stored in a central location, making it easy to manage them.
- **Version control:** Configuration properties can be versioned in the same way as your application code, making it easy to roll back to a previous version if necessary.
- **Dynamic configuration:** Configuration properties can be changed at runtime, without having to redeploy your applications.
- **Support for multiple environments:** The same configuration properties can be used for different environments, such as development, staging, and production.
- **Easy to use:** Spring Cloud Config Server and Config Clients are easy to configure and use.

If you are looking for a way to manage configuration properties in a distributed system, then Spring Cloud Config Server and Config Clients are a good option. They are powerful, flexible, and easy to use.

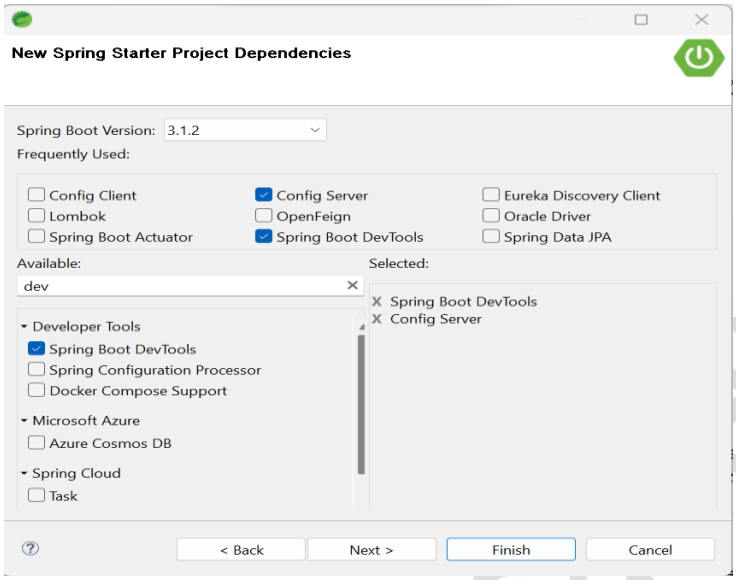
To set up Spring Cloud Config, you typically follow these steps:

- Create a Spring Boot application for the Config Server.
- Configure the Config Server to connect to a version-controlled repository (e.g., Git) where your configuration files are stored.

- Create Spring Boot applications for your microservices that will act as Config Clients.
- Configure the Config Clients to fetch their configuration from the Config Server.

Config Server Setup: With GitHub:

Step 1: Create Config Server with Below Dependency



Step 2: Add An annotation called as **@EnableConfgiServer** on Spring Boot Application class level.

```
package com.swiggy.config;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer
@SpringBootApplication
public class ConfigServerApplication {

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

Step 3: Configure GitHub Details inside Config Server Properties file. By Default GitHub integration supported by Config Server.

Now We are chosen All configuration Properties should be available in Github Repository.

So create A Github Repository. Created Public Repository of below.

https://github.com/dilipsingh1306/swiggy-config-server-data.git

Now Configure the GitHub Repository Details in Config Server Properties File.

- **spring.cloud.config.server.git.uri :** URL of The GitHub Repository.
- **spring.cloud.config.server.git.skip-ssl-validation :** Skipping SSL Certificate Validation. The configuration server's validation of the Git server's SSL certificate can be disabled by setting property to **true** (default is false).
- **spring.cloud.config.server.git.default-label :** The default label i.e. Branch Name to be used with the remote repository.

application.properties

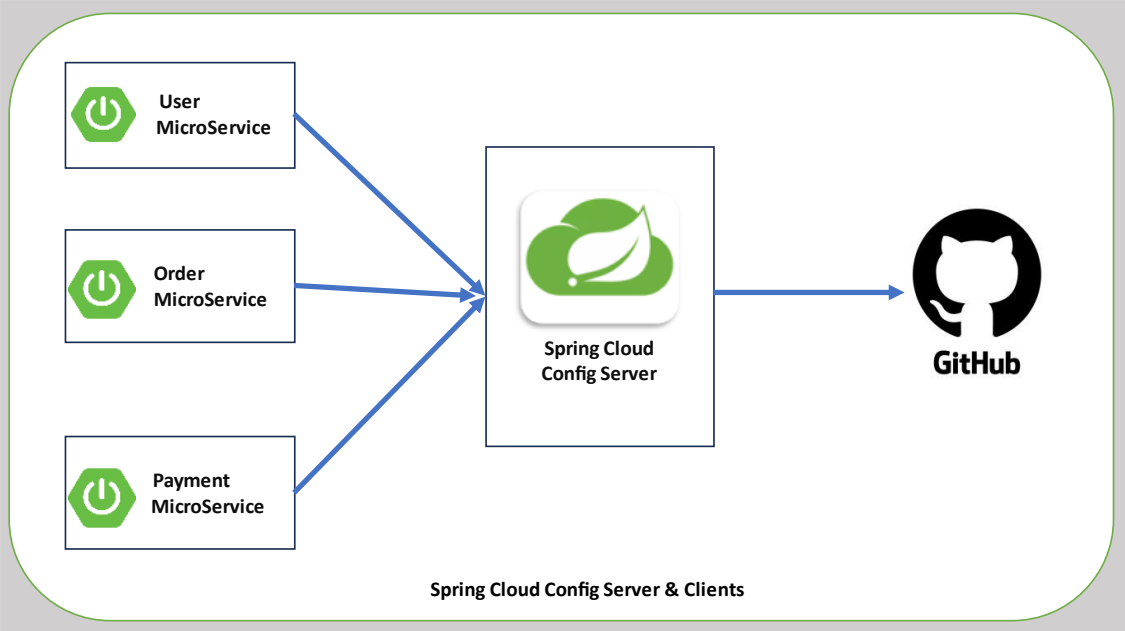
```
server.port=9988
server.servlet.context-path=/config
spring.application.name=config-server

#github properties
spring.cloud.config.server.git.uri=https://github.com/dilipsingh1306/swiggy-config-server-data.git
spring.cloud.config.server.git.skip-ssl-validation=true
spring.cloud.config.server.git.default-label=master
```

Now We can Start Our Config Server. With this Configuration server Setup is completed with Backend as GitHub Repository.

Let's Work on How to Maintain Configuration of MicroServices Properties.

In our Case we have total three MicroServices **user, order and payment**. So we have to move Configuration from MicroService level to GitHub repositories and access via Config Server.

We should make **user, order** and **payment** as Config Clients as a first step.



Enabling Spring Cloud Config Clients to Our Micro Services:

Repeat Steps For All Three Micro Services.

Step 1: Add Dependency of Spring Cloud Config Client Starter to pom.xml file.

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

Step 2: Add Config Server URL to MicroService application.properties file.

```
spring.config.import=optional:configserver:http://localhost:9988/config
```

That's all Our Three Micro Services are Ready to Fetch Configuration from Config Server.

Configuring MicroServices Properties inside Config Server:

Means we should Create Configuration Properties inside GitHub Repository. Those configuration properties pulled by Config Server internally and distributed to Respective MicroServices.

In general, We should maintain MicroServices Configuration as follows.

- Common Properties of All or Multiple MicroServices.
- MicroService Individual/Specific Properties.

Common Properties Configuration:

As per Config Server, Common Properties Should be maintained inside **application.properties** file and that should be created inside GitHub Repository.

First Create **application.properties** file inside GitHub. This properties File is common for all Micro Services Configuration i.e. We have to maintain properties only common across multiple MicroServices.

Current Configuration of Our Micro Services:

User MicroService

```
#Project Info
server.port = 8001
server.servlet.context-path=/user
spring.application.name=user-service

#Database
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka

spring.config.import=optional:configserver:http://localhost:9988/config
```

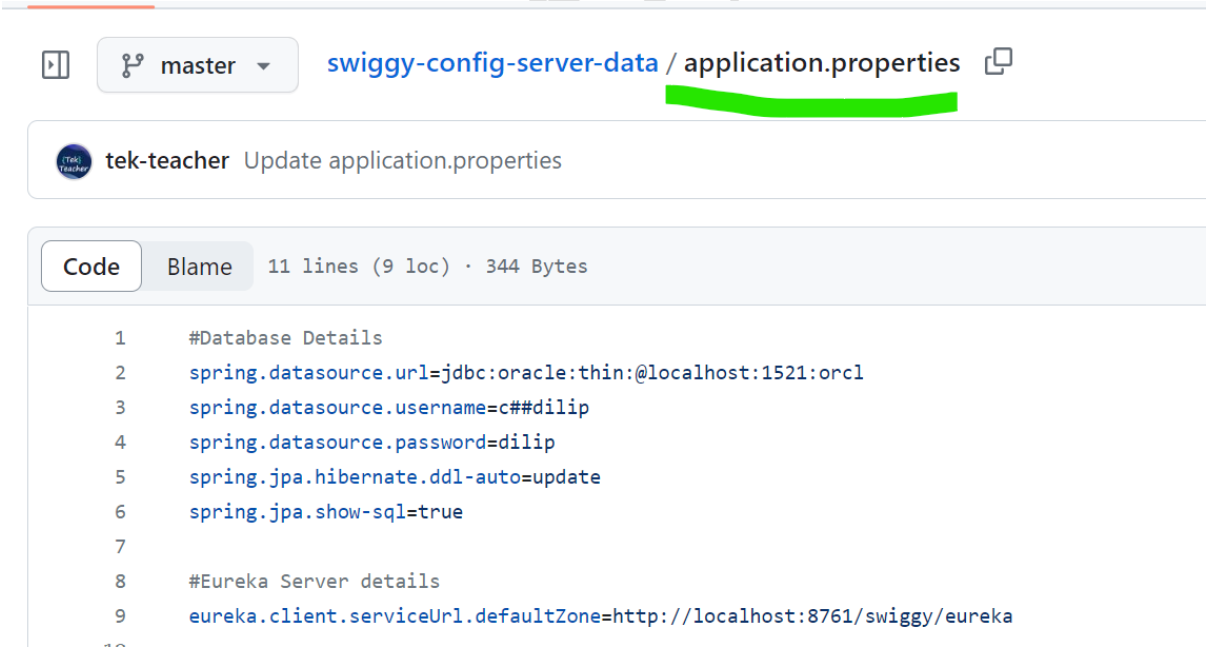
Order MicroService:

```
#Project Info
server.port = 8002
server.servlet.context-path=/order
spring.application.name=order-service

#Database
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka

spring.config.import=optional:configserver:http://localhost:9988/config
```

Payment MicroService:

```
#Project Info
server.port = 8003
server.servlet.context-path=/payment
spring.application.name=payment-service

#Database
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl
spring.datasource.username=c##dilip
spring.datasource.password=dilip
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

#Eureka Server details
eureka.client.serviceUrl.defaultZone=http://localhost:8761/swiggy/eureka

spring.config.import=optional:configserver:http://localhost:9988/config
```

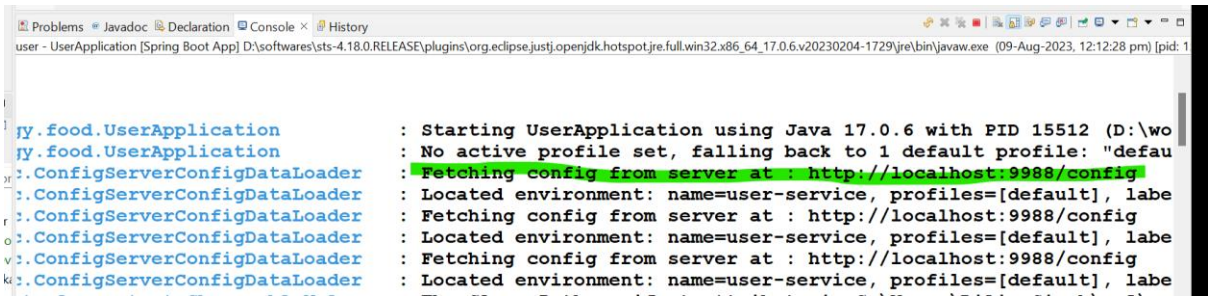
From Above Three MicroServices Properties, If we observe **Database and Eureka Server Details are common across apart from Config Server Property.** These Type of data should be maintained/moved inside **application.properties** in GitHub Repository which is integrated with Cloud Config Sever.



Delete Same Properties from Individual Micro services and try to start up Micro Services.

Note: Please Start Config Server First and then MicroServices i.e. Config Clients.

While Starting Config Clients i.e. MicroServices in Console Logs printed as Configuration Fetching From Config Server and Details.



All 3 Micro services are Up and Running, that means Database and Eureka Server Details Are fetched from Config Server.

So in future If we have any common configuration properties and data either predefined or user-defined we have to Define inside **application.properties** so that all MicroServices will fetch same.

MicroServices Individual/Specific Properties Configuration:

For example, Every MicroService will have it's own context path and Port number. In such case, we should not maintain those properties inside **application.properties** because same context path and port will be assigned to all micro services while starting and fails.

For these kind of Scenarios, like property is same but value is individual to each MicroService then Config Server provided an option like creating individual properties files with application/service names.

Now we have to create 3 properties files with Micro Service name(i.e. value of **spring.application.name**) what we provided inside the properties file.

Current Configuration of Our Micro Services:

User MicroService

```
#Project Info
server.port = 8001
server.servlet.context-path=/user
spring.application.name=user-service

spring.config.import=optional:configserver:http://localhost:9988/config
```

Order MicroService:

```
#Project Info
server.port = 8002
server.servlet.context-path=/order
spring.application.name=order-service
```

```
spring.config.import=optional:configserver:http://localhost:9988/config
```

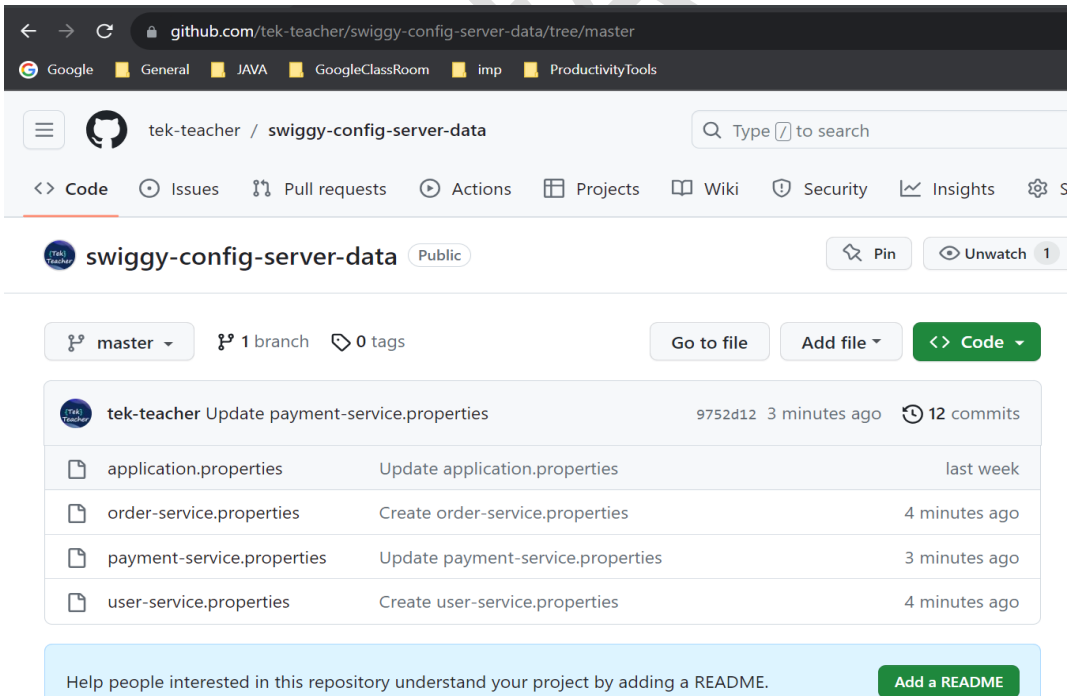
Payment MicroService:

```
#Project Info
server.port = 8003
server.servlet.context-path=/payment
spring.application.name=payment-service

spring.config.import=optional:configserver:http://localhost:9988/config
```

From Above Three MicroServices Properties, If we observe **Port and Context Details are different.** These Type of data should be maintained/moved inside **<service-name>.properties** in GitHub Repository which is integrated with Cloud Config Sever.

So create 3 individual Micro Services properties files in side GitHub Repository. After that move properties of context path and port to GitHub properties files level.

- **user-service.properties**
- **order-service.properties**
- **payment-service.properties**



swiggy-config-server-data / user-service.properties

tek-teacher Create user-service.properties

Code Blame 2 lines (2 loc) · 53 Bytes

```
1   server.port = 8001
2   server.servlet.context-path=/user
```

swiggy-config-server-data / order-service.properties

tek-teacher Create order-service.properties

Code Blame 2 lines (2 loc) · 54 Bytes

```
1   server.port = 8002
2   server.servlet.context-path=/order
```

swiggy-config-server-data / payment-service.properties

tek-teacher Update payment-service.properties

Code Blame 2 lines (2 loc) · 56 Bytes

```
1   server.port = 8003
2   server.servlet.context-path=/payment
```

Current Configuration of Our Micro Services in Local:

User MicroService

```
spring.application.name=user-service
```

```
spring.config.import=optional:configserver:http://localhost:9988/config
```

Order MicroService:

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

spring.config.import=optional:configserver:http://localhost:9988/config
```

Payment MicroService:

```
spring.application.name=payment-service

spring.config.import=optional:configserver:http://localhost:9988/config
```

Now Start Your Services in An Order.

- Config Server
- Eureka Server
- Gateway
- User
- Order
- Payment

Now we can access All Services REST Services with Gateway.

Can we define our own User Defined Properties in side Config Server i.e. GitHub Repository?

Yes, We can Define user defined properties inside **application.properties** or individual MicroService Properties File level. i.e. When we need a properties across multiple services level, then we have to define in common **application.properties** file level. If Properties are Specific to MicroServices, then define in individual Micro Service Properties file Level.

Profile Based Properties Files in Config Server:

application.properties	->	All Micro Services + All profiles
application-dev.properties	->	All Micro Services + Only DEV profile
application-sit.properties	->	All Micro Services + Only SIT profile
user.properties	->	User MicroService + All profiles
user-dev.properties	->	User MicroService + Only DEV profile
user-sit.properties	->	User MicroService + Only SIT profile
order.properties	->	Order MicroService + All profiles
order-dev.properties	->	Order MicroService + Only DEV profile
order-sit.properties	->	Order MicroService + Only SIT profile

payment.properties -> Payment MicroService + All profiles
payment-dev.properties -> Payment MicroService + Only DEV profile
payment-sit.properties -> Payment MicroService + Only SIT profile

If a Property available in common, individual, profile based properties file of Micro Services, then which Property will be loaded dynamically?

1st Priority : Property Loaded from Micro Service Profile Based
2nd Priority : Property Loaded from Micro Service Properties file
3rd Priority : Property Loaded from Common Properties file

Circuit Breaker in MicroServices:

In a microservice architecture, it's common for a service to call another service. And there is always the possibility that the other service being called is unavailable or unable to respond. So, what can we do when this happens?

A circuit breaker is a pattern used to protect a system from cascading failures. It works by monitoring the number of failures for a particular operation and, if the failure rate exceeds a threshold, it stops making calls to that operation. This prevents the failure from spreading to other parts of the system. In Spring microservices, the circuit breaker pattern can be implemented using the Spring Cloud Circuit Breaker project. This project provides an abstraction layer across different circuit breaker implementations, so you can choose the one that best suits your needs.

This pattern comes into the picture while **communicating between services**. Let's take a simple scenario. Let's say we have two services: Service A and B. Service A is calling Service B(API call) to get some information needed. When Service A is calling to Service B, if Service B is down due to some infrastructure outage, what will happen? **Service A is not getting a result and it will be hang by throwing an exception**. Then another request comes and it also faces the same situation. Like this request threads will be blocked/hanged until Service B is coming up! As a result, the network resources will be exhausted with low performance and bad user experience. **Cascading failures** also can happen due to this.

In such scenarios, we can use this **Circuit Breaker pattern to solve the problem**. It is giving us a way to handle the situation without bothering the end user or application resources.

How the pattern works?

Basically, it will behave same as an electrical circuit breaker. **When the application gets remote service call failures more than a given threshold, circuit breaker trips for a particular time period. After this timeout expires, the circuit breaker allows a limited number of requests to go through it. If those requests are getting succeeded, then circuit breaker will**

be closed and normal operations are resumed. Otherwise, it they are failing, timeout period starts again and do the rest as previous.

Let's figure out this using the upcoming example scenario that I'm going to explain.

Life Cycle of Pattern States:

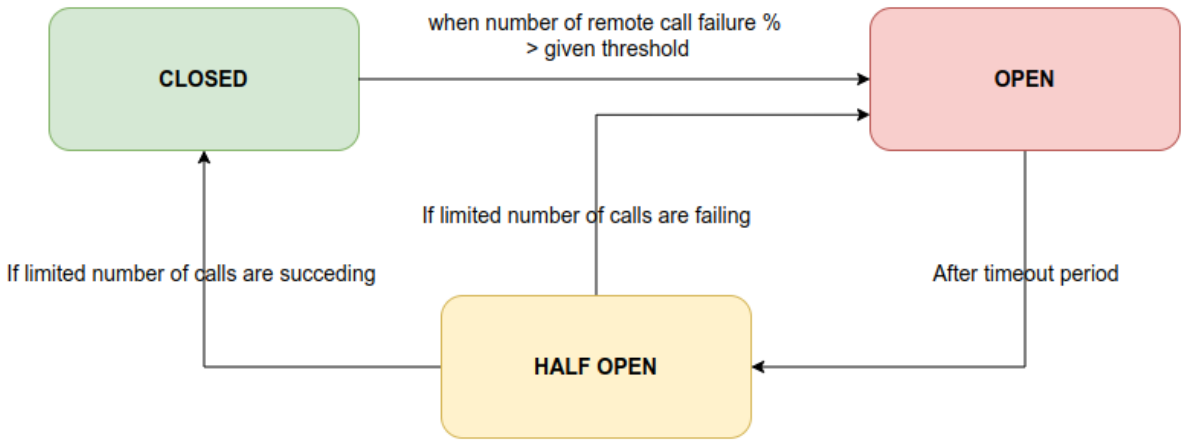
There are 3 main states discussed in Circuit Breaker pattern. They are:

1. CLOSED
2. OPEN
3. HALF OPEN

CLOSED: When both services which are interacting are up and running, circuit breaker is CLOSED. Circuit breaker is counting the number of remote API calls continuously.

OPEN: As soon as the percentage of failing remote API calls is exceeding the given threshold, circuit breaker changes its state to OPEN state. Calling micro service will fail immediately, and an exception will be returned. That means, the flow is interrupted.

HALF OPEN: After staying at OPEN state for a given timeout period, breaker automatically turns its state into HALF OPEN state. In this state, only a LIMITED number of remote API calls are allowed to pass through. If the failing calls count is greater than this limited number, breaker turns again into OPEN state. Otherwise it is CLOSED.



To demonstrate the pattern practically, I will use Spring Boot framework to create the micro services. Resilience4j library is used to implement the circuit breaker.

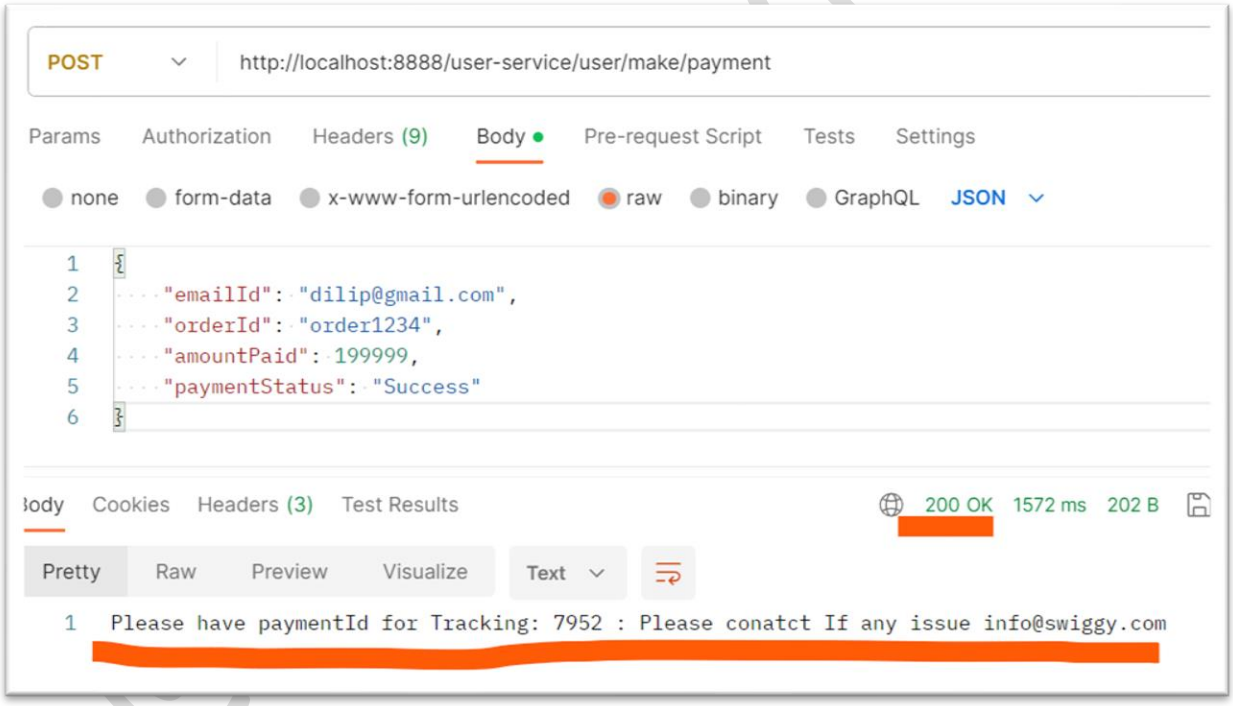
What is Resilience4j?

Resilience4j is a lightweight, easy-to-use fault tolerance library inspired by Netflix Hystrix. It provides various features. If you are looking for a lightweight and easy-to-use fault tolerance library for your Java application, Resilience4j is a good choice.

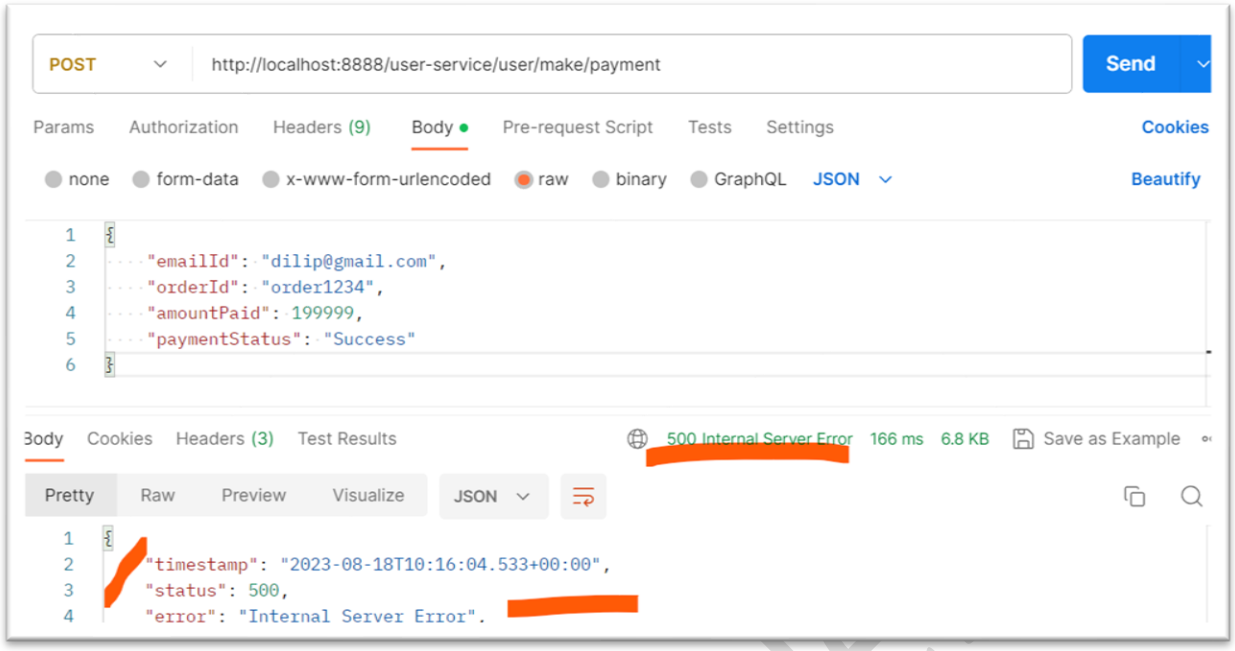
Here are some of the use cases of Resilience4j:

- To protect your application from cascading failures.
- To improve the availability of your application.
- To reduce the impact of failures on your application.
- To improve the performance of your application.
- To make your application more resilient to unexpected events.

- Now In Our existing Micro Services, User Service calling Payment Service API's. When Payment API Services are Good and working as expected, we get below Response.



- In case, Payment Service is Down then we will get an Exception as shown below.



In these cases, we should integrate or enable Circuit Breaker, to handle fault tolerance. This Breaker should be enabled in Consumer Service Side.

Steps To Enable Circuit Breaker:

- **Add Below Starter Dependencies in Side User Service to enable Resilience4J Circuit Breaker.**

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

Here, aop functionalities used by circuitbreaker-resilience4j internally so added. Similarly, actuator should be required to enable endpoint of circuit breaker to monitor circuit breaker state and statistics.

- **Add Circuit Breaker Configuration in Properties of User Micro Service.**

```
management.health.circuitbreakers.enabled=true
management.endpoints.web.exposure.include=health
management.endpoint.health.show-details=always

#Resilience Circuit Breaker
resilience4j.circuitbreaker.instances.user-service.registerHealthIndicator=true
resilience4j.circuitbreaker.instances.user-service.eventConsumerBufferSize=10
resilience4j.circuitbreaker.instances.user-service.failureRateThreshold=50
resilience4j.circuitbreaker.instances.user-service.minimumNumberOfCalls=5
resilience4j.circuitbreaker.instances.user-
service.automaticTransitionFromOpenToHalfOpenEnabled=true
resilience4j.circuitbreaker.instances.user-service.waitDurationInOpenState=6s
resilience4j.circuitbreaker.instances.user-
service.permittedNumberOfCallsInHalfOpenState=3
resilience4j.circuitbreaker.instances.user-service.slidingWindowSize=10
resilience4j.circuitbreaker.instances.user-service.slidingWindowType=COUNT_BASED
```

- **Now Create Circuit Breaker and fall back method, where actually we are integrated Payment API Service from User Service.**

What is fallback Method?

The **fallbackMethod** attribute in the **@CircuitBreaker** annotation specifies a method that will be called if the circuit breaker is open. This method is called the fallback method.

The fallback method can be used to provide a graceful degradation of service when the circuit breaker is open. For example, the fallback method could return a default value, or it could retry the call to the protected method a few times before giving up.

The fallback method must have the same method signature as the protected method, with the addition of one extra parameter: the exception that caused the circuit breaker to open.

The fallback method is a powerful tool that can be used to protect your application from cascading failures. By providing a fallback method, you can ensure that your application will continue to function even if the protected method fails.

Here are some of the things to keep in mind when using the fallback method:

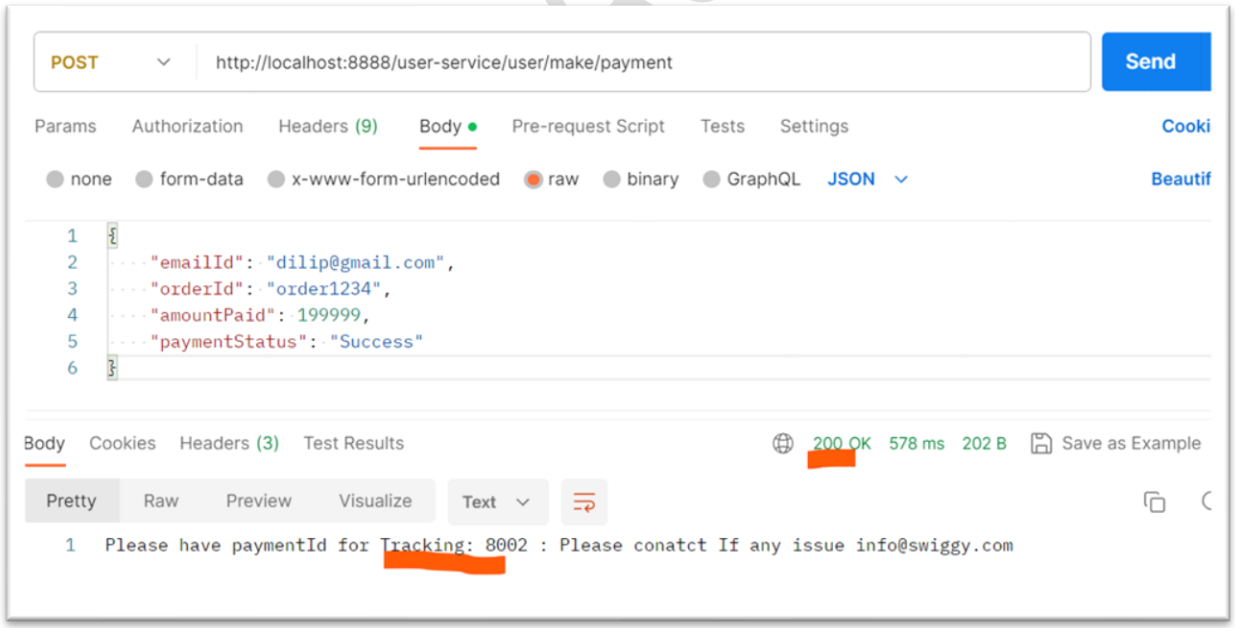
- The fallback method should be lightweight and fast. It should not do anything that could cause the circuit breaker to open again.
- The fallback method should be used to provide a graceful degradation of service. It should not be used to provide the same level of service as the protected method.

UserController.java

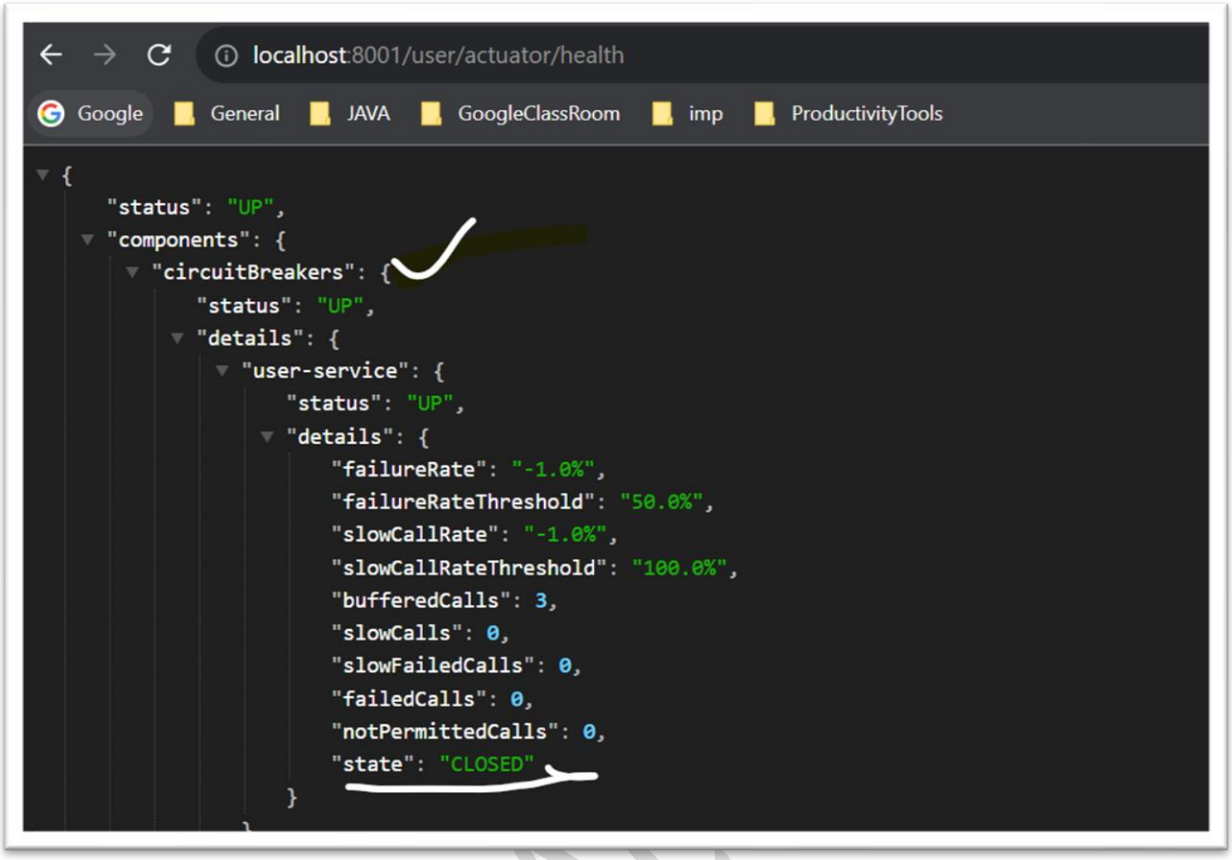
```
@PostMapping("/make/payment")
@CircuitBreaker( name="user-service" , fallbackMethod = "paymentStatus")
public String paymentStatus(@RequestBody PaymentDetails details) {
    return paymentClinet.makePayment(details) + " : Please contact If any issue " + emailId;
}

//fallbackMethod
public String paymentStatus(Throwable  ex) {
                return "Backend Baking Systems are Not functioning. Please Contact Admin";
}
```

Now Test Above endpoint and check circuit breaker status.

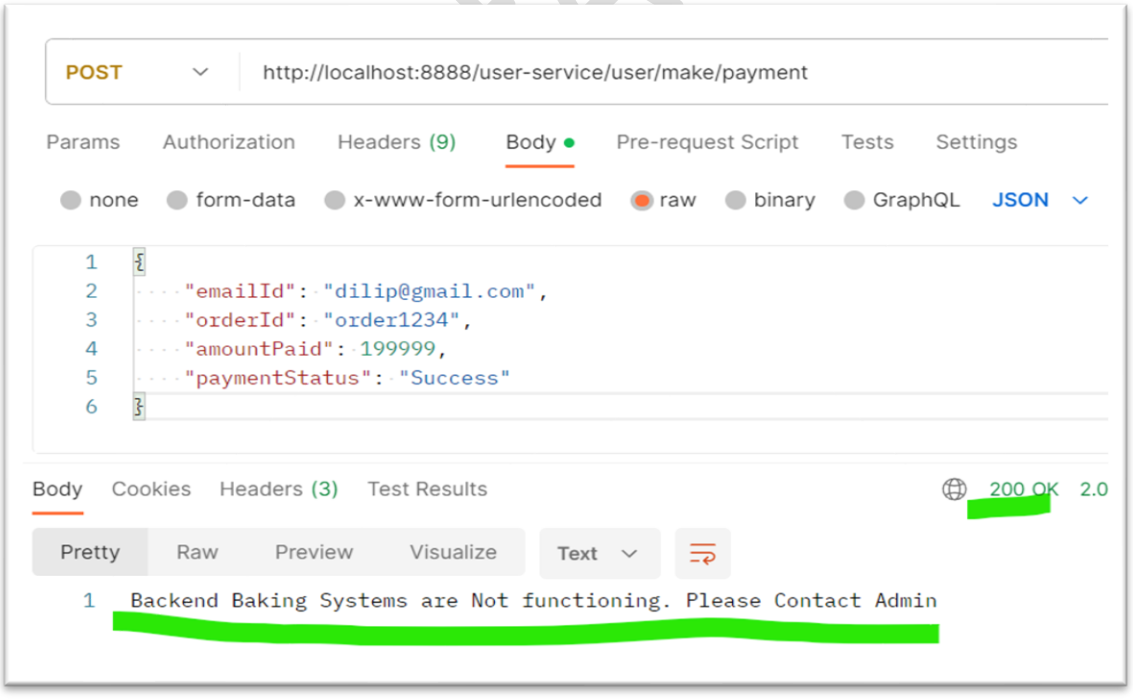
When Payment Service is UP: Getting Actual response from Payment Service



Now Access actuator health endpoint, for Watching Status of Circuit Breaker. Showing as Closed.

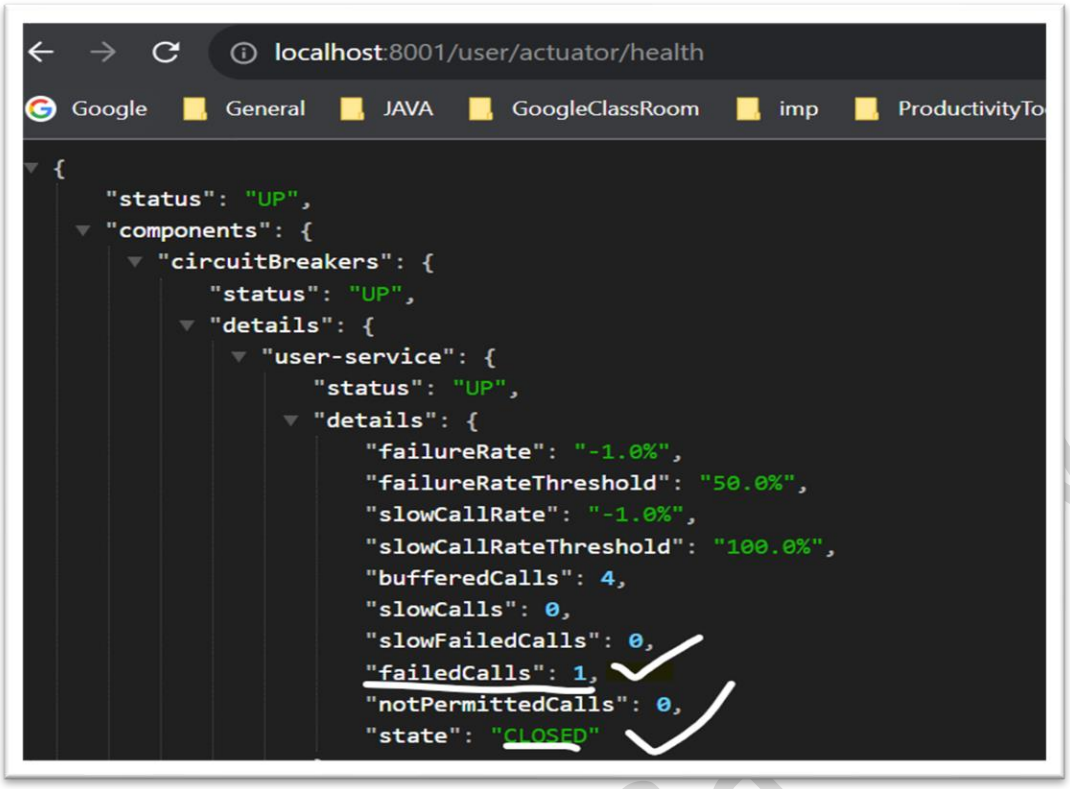


- **When Payment Service is Down: Bring Down Payment Service.**

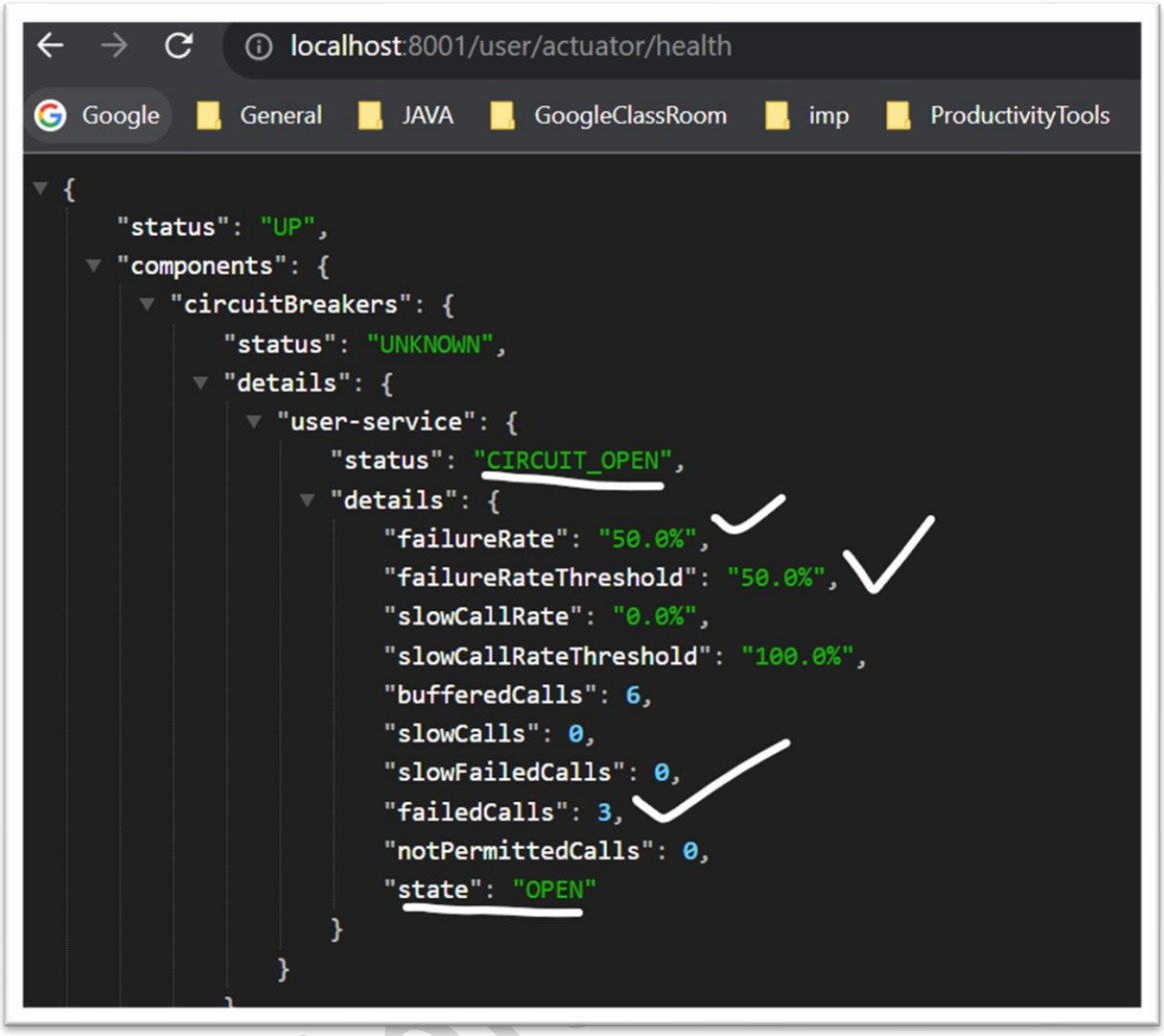


In Above, even though Payment Service is Down, instead of Internal Server Error, we are getting response of fallback method with status 200 Ok.

Now Check Circuit Breaker Status from actuator health endpoint.



After 50% of threshold calls, Circuit opened as per our configuration in properties files.



Until Actual Payment Service is Up and Running and getting original Response Data, Circuit breaker will be opened. Once Actual service is Good, then Circuit breaker will be closed.

This is how we are going to implement Circuit Breaker to achieve fault tolerance where we have API service calls are integrated in any Micro Service.

Note: If we want to move Circuit Breaker Configuration from local to Config Server i.e. GitHub, we can do that.

All latest Configuration and Micro Services are located in GitHub.

https://github.com/dilipsingh1306/SpringBoot-MicroServices

https://github.com/dilipsingh1306/swiggy-config-server-data