www.youtube.com/c/javaexpress
+91 7801007910
javaexpresschannel@gmail.com

Microservices Notes

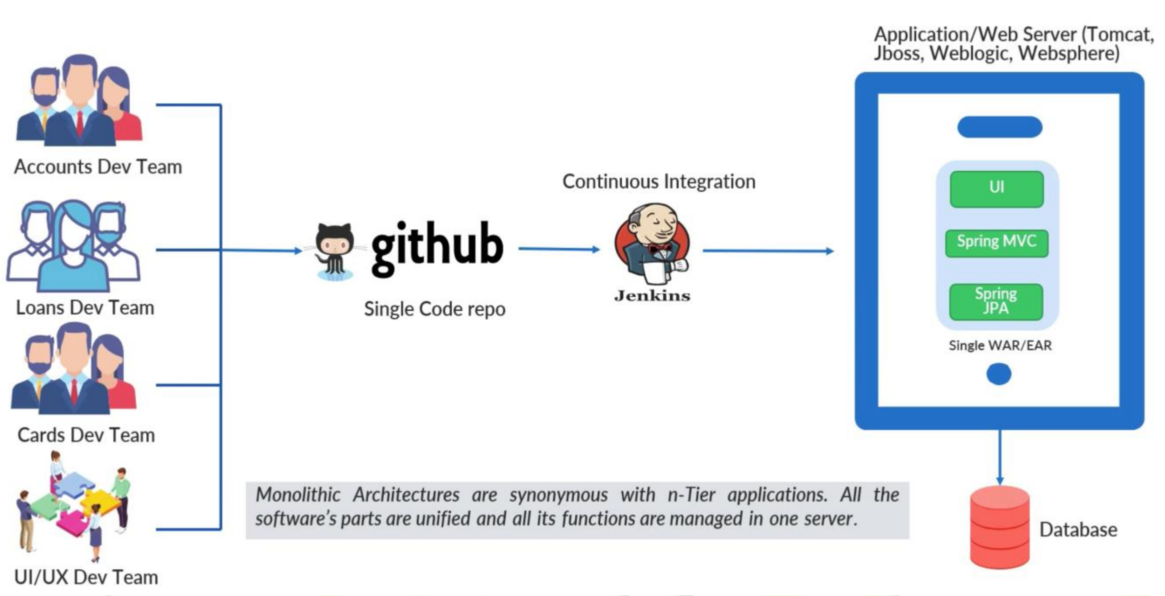
Microservices Course Index:

Monolithic Architecture Background:

Traditionally, the application was divided into modules and these modules were built in a single code base. This kind of architecture which we call “Monolithic Architecture”. Mono means ‘Single’ and lith means ‘a single block of stone or pillar’. In short, the entire application was built in a single code base. Let’s discuss the cons of this kind of architecture.

- **Deployment**: If there is a small change in any part of the code, you will need to deploy the entire application again.

- **Re-usability**: Monolithic applications are not reusable because they use tools out of the box. You will not be able to just extract a feature and integrate it into another project.
- **Testing**: Even a small change causes to test the entire application which can be both time and resource consuming.


Accounts Dev Team
Loans Dev Team
Cards Dev Team
UI/UX Dev Team
github
Single Code repo
Continuous Integration
Jenkins
Application/Web Server (Tomcat, Jboss, Weblogic, Websphere)
UI
Spring MVC
Spring JPA
Single WAR/EAR
Database
Monolithic Architectures are synonymous with n-Tier applications. All the software's parts are unified and all its functions are managed in one server.

What is a Microservice?

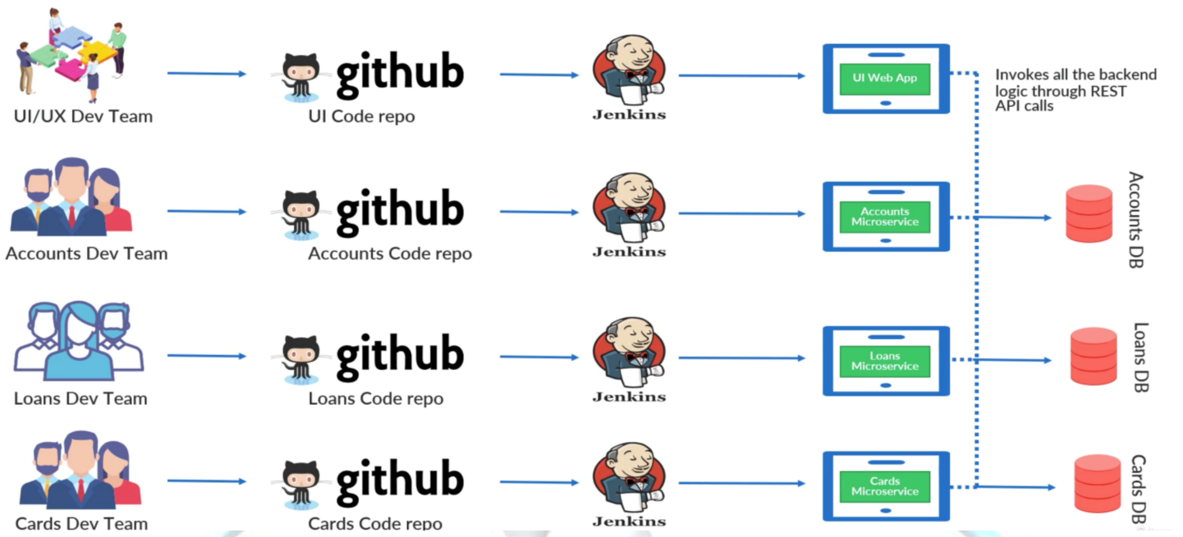
- Microservice is not a technology
- Microservice is not a framework
- Microservice is not an API
- Microservice is an Architectural Design Pattern
- Microservices design pattern came into market to avoid the problems of Monolithic Architecture.
- Microservices design pattern not only related to java. It is a universal design pattern anybody can follow this design pattern.

Microservices Architecture Background:

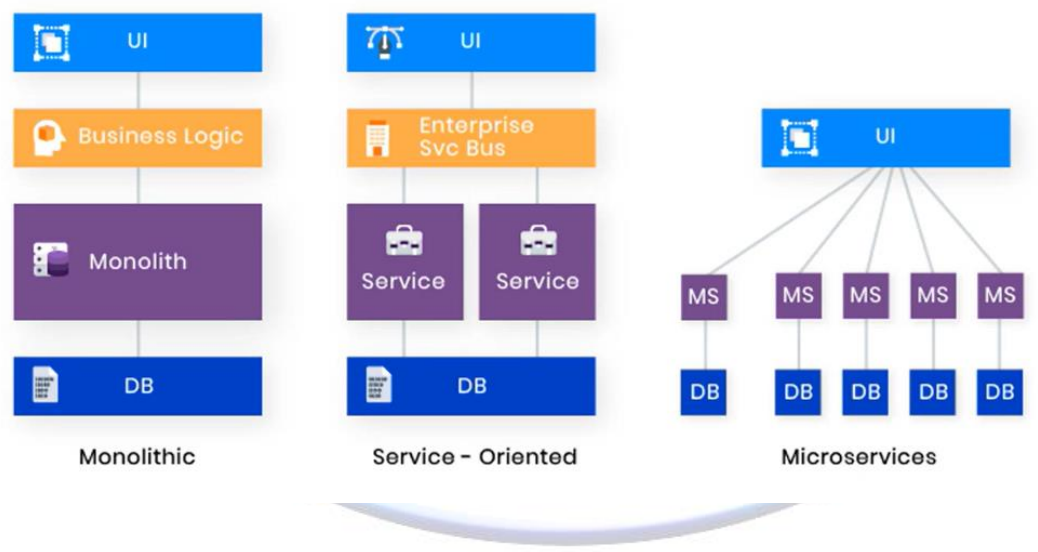
In modern cloud-based architecture, we use microservices to build and deploy applications. Microservices appeared as an alternative to monoliths in order to solve all issues and bottlenecks caused by the limitations of the monolithic architecture. Let's discuss the pros of Microservice architecture.

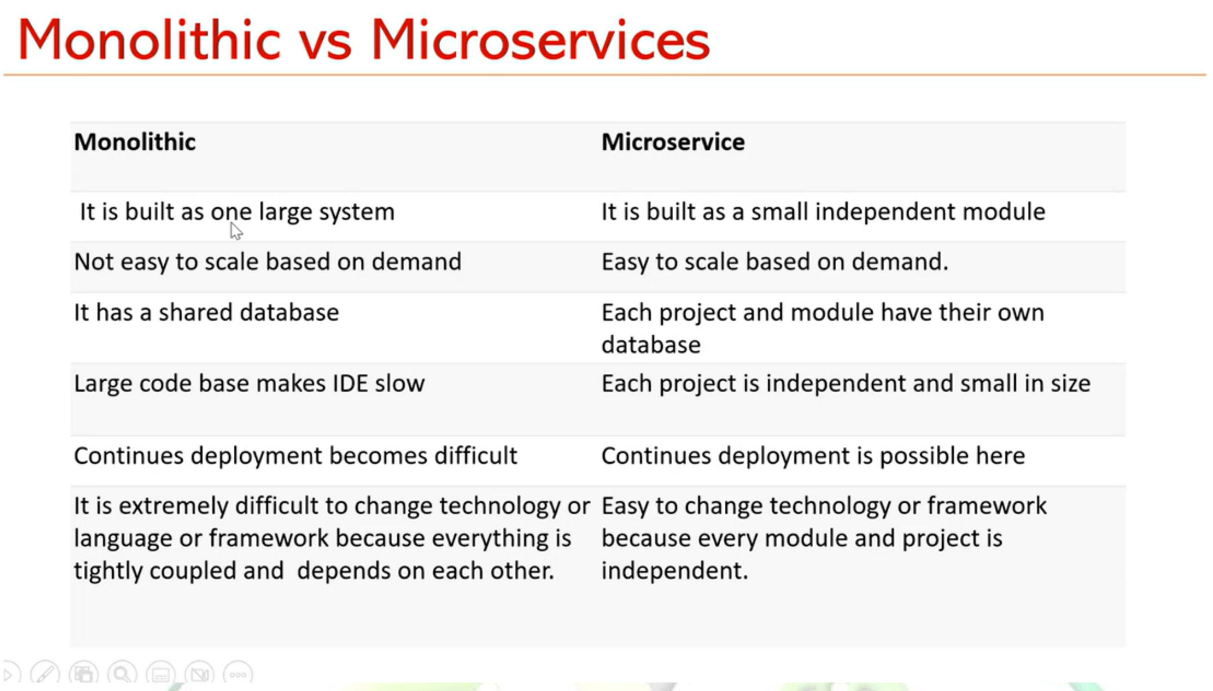
- **Deployment:** As each microservice is an individual unit, it's easy to deploy a unit rather than an entire application.
- **Re-usability:** We can reuse the microservice in any project without any fear due to its zero dependency.

- **Testing:** It's really easy to test the small code base rather than the entire application.



Monolithic vs Microservices

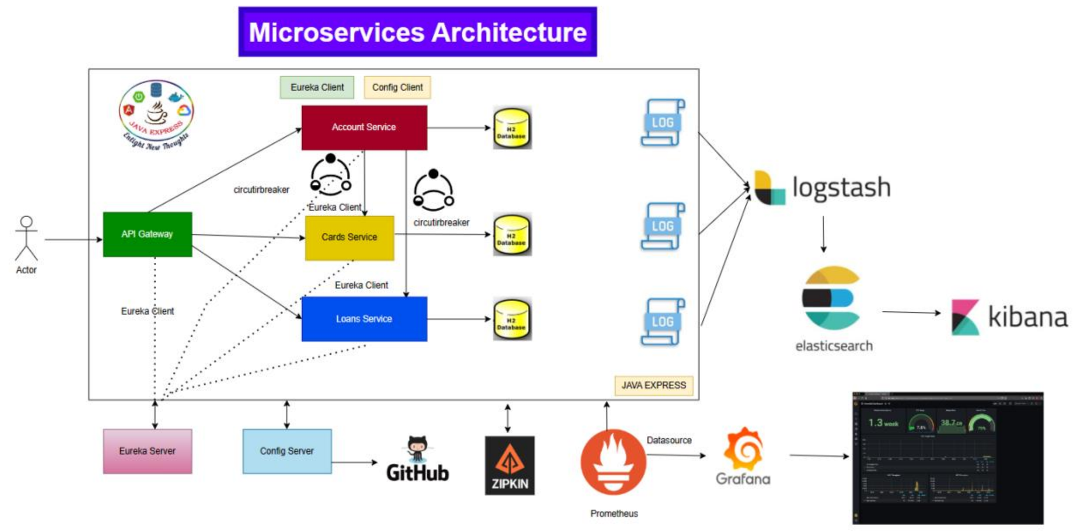




Monolithic vs Microservices

Monolithic	Microservice
It is built as one large system	It is built as a small independent module
Not easy to scale based on demand	Easy to scale based on demand.
It has a shared database	Each project and module have their own database
Large code base makes IDE slow	Each project is independent and small in size
Continues deployment becomes difficult	Continues deployment is possible here
It is extremely difficult to change technology or language or framework because everything is tightly coupled and depends on each other.	Easy to change technology or framework because every module and project is independent.

Microservices Architecture



What is Spring Cloud?

- **Spring Cloud provides frameworks for developers to quickly build some of the common patterns of microservices.**
    - **Spring Cloud Config**
    - **Service Registration & Discovery**
    - **Spring Cloud OpenFeign**
    - **Micrometer Tracing(spring boot 3)**
    - **Spring Cloud Bus**
    - **Routing & Tracing**
    - **Load Balancing**
    - **Spring Cloud Security**
    - **Distributed Tracing & Messaging**

Service Registry & Discovery

- Service discovery and registration are key components of microservice architectures.
- Service discovery & registration deals with the problems about how microservices talk to each other, i.e., perform API Calls.
- Service Registry is used to register microservices available in our project.
- Service Registry will provide a dashboard with services information like Status, Health and URL etc.

Why we need Service Discovery Example:

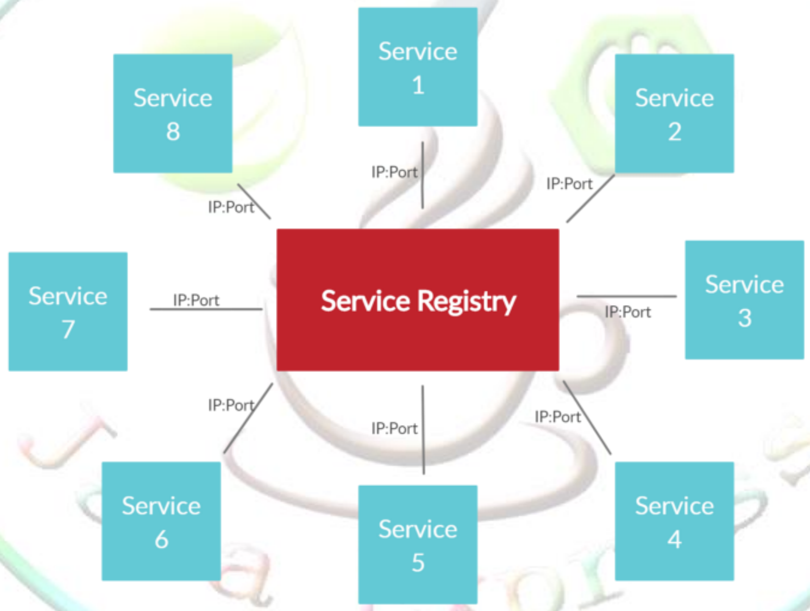
- Let’s imagine you have multiple microservices to collectively form an E-commerce application. So,
    - we have a Product MS, - 8090,8060
    - Order MS, and - 8050
    - Consumer management microservices. - 8030
- Now, how these microservices will talk to each other? So basically, you will have REST API exposed which other services can consume it.
- For example, suppose Order Service needs the product information then the APIs of Product Service will be integrated inside Order microservice.
- So how will you store this API endpoint?
- Will you hard code it in your Order microservice?
- If you hard code the API endpoint, what if the endpoint is changed in the future?
- So here, Service Discovery comes into the picture.

- Instances have dynamically assigned network locations. Consequently, in order for a client to make a request to service, it must use a service-discovery mechanism.

Why Hardcoding URL is wrong?

1. **Changes require code updates** — If one microservice address has changed, then it has to be updated in all the other microservices that are in contact with it.
2. **Dynamic URL in the cloud** — When we deploy on the cloud we get dynamic URLs, and it changes if we stop and start the server again or deploy it elsewhere. We need to accommodate the change; we can't predict the URL changes.
3. **Load balancing** — If we spin up multiple instances, each instance might have different URLs and it is very inefficient to hardcode the URLs of instances as well.
4. **Multiple environments** — The URLs change in each phase of the software engineering process, i.e., the URLs might be different during development, QA testing, and production. We can't change the hardcoded URLs each time.

Centralized Service Registry



Microservices Network:

- Microservices service discovery & registration is a way for applications and Microservices to locate each other on a Network. This includes
    - A Central server that maintains a global view of addresses.
    - Microservices/clients that connect to the central server to register their address when they start & ready.
    - Microservices/clients need to send their heartbeats at regular intervals to central server about the health.

Spring Cloud Support:

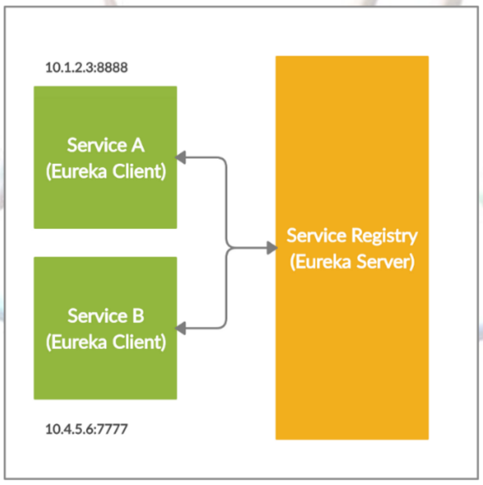
- Spring Cloud Project makes Service discovery & Registration setup to undertake with the help of the below components
    - Spring Cloud Netflix's Eureka service which will act as a service discovery agent
    - Spring Cloud Load Balancer library for client-side load balancing.
    - Netflix Feign client to look up for a service between microservices

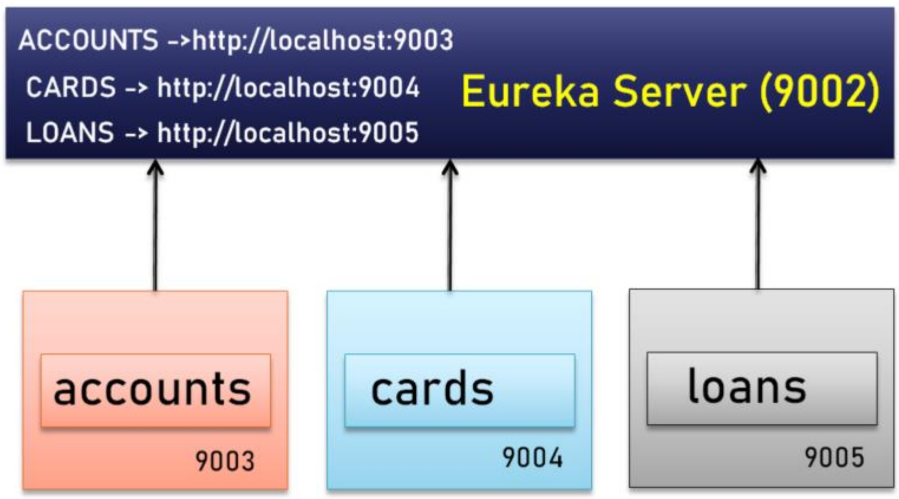
Spring Cloud Support for Client Side Service Discovery

- Eureka mainly consists of main components, let's see what they are:
- **Eureka Server:**
    - We can use Eureka Server as a Service Registry and it is open Source
    - Eureka Server provided by Spring Cloud Netflix Libraries
    - It is an application that contains information about all client service applications. Each microservice is registered with the Eureka server and Eureka knows all the client applications running on each port and IP address.
    - Eureka Server is also known as Discovery Server.
- **Eureka Client:**
    - It's the actual microservice and it registers with the Eureka Server, so if any other microservice wants the Eureka Client's address then they'll contact the Eureka Server.

- In this course we use Eureka since it is mostly used but they are other service registries such as etcd, Consul, and Apache Zookeeper which are also good

Eureka Server & Client


10.1.2.3:8888
Service A (Eureka Client)
Service B (Eureka Client)
10.4.5.6:7777
Service Registry (Eureka Server)


ACCOUNTS ->http://localhost:9003
CARDS -> http://localhost:9004
LOANS -> http://localhost:9005
Eureka Server (9002)
accounts
9003
cards
9004
loans
9005

Steps to build Eureka Server

1. Create Spring Boot Project and add dependency
2. Add the **spring-cloud-starter-netflix-eureka-server** dependency to your pom.xml file.
    I. Web
    II. Actuator
    III. Eureka dependency
3. In your main application class, annotate it with **@EnableEurekaServer**. This will enable the Eureka Server functionality.
4. Configure Below Properties in application
    I. Application Name
    II. Port Number
    III. Enable actuator's
    IV. Eureka configurations
        1. Register with Eureka
        2. Fetch Registry
        3. Default Zone URL
5. Build and run the Eureka Server : Open a browser and navigate to http://localhost:9002. You should see the Eureka server dashboard, which displays information about registered services instances

www.youtube.com/c/javaexpress
+91 7801007910
javaexpresschannel@gmail.com

Dependency

```xml
<properties>
    <java.version>17</java.version>
    <spring-cloud.version>2023.0.1</spring-cloud.version>
</properties>

<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

Eureka server properties

```yaml
eureka:
  client:
    fetch-registry: false
    register-with-eureka: false
    service-url:
      defaultZone: http://localhost:9002/eureka/
  instance:
    prefer-ip-address: true
management:
  endpoints:
    web:
      exposure:
        include: '*'
server:
  port: 9002
spring:
  application:
    name: eurekaserver
logging:
  file:
    name: eurekaserver_logs/eurekaserver.log
```

How to access Eureka Server Dashboard?

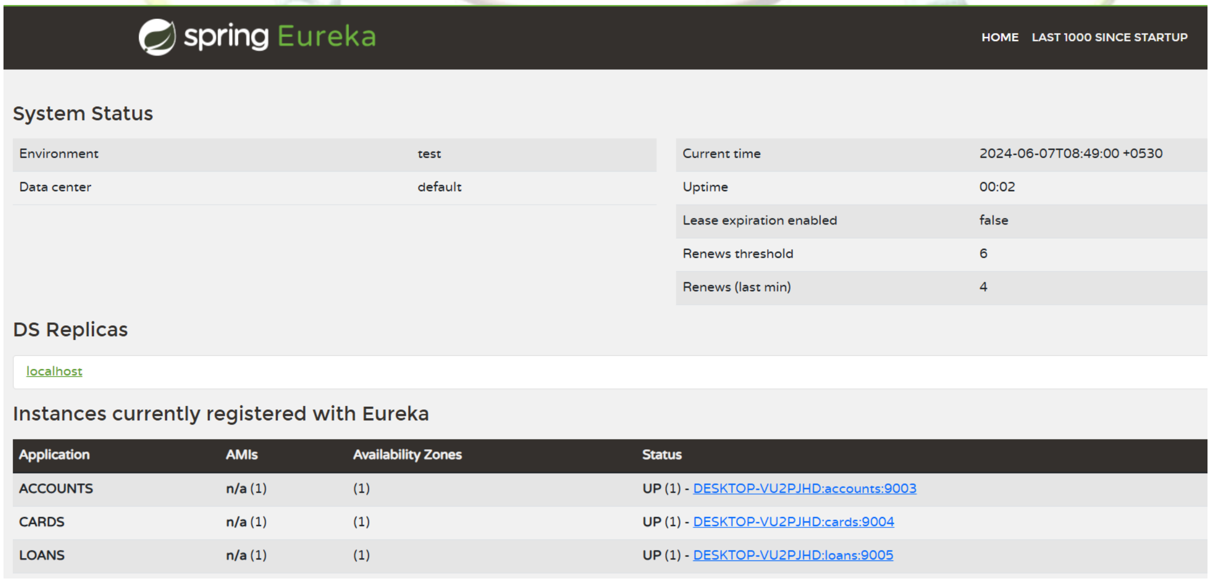
- http://localhost:9002

Steps to register a microservice as a Eureka Client

1. Use existing microservices and add necessary dependency in pom.xml file
2. Add the spring-cloud-starter-netflix-eureka-client dependency to your pom.xml file.
3. Enable the Eureka client in your main application class by adding the **@EnableDiscoveryClient** annotation.
4. Configure Below Properties in application
    1. Application Name
    2. Port Number
    3. Enable actuator's
    4. Eureka configurations
        1. Register with Eureka
        2. Fetch Registry
        3. Default Zone URL
        4. prefer-ip-address (Only for cloud deployment)

Eureka Client Properties

```
eureka:
  client:
    fetch-registry: true
    register-with-eureka: true
    service-url:
      defaultZone: http://localhost:9002/eureka/
  instance:
    prefer-ip-address: true
```



System Status

Environment	test
Data center	default

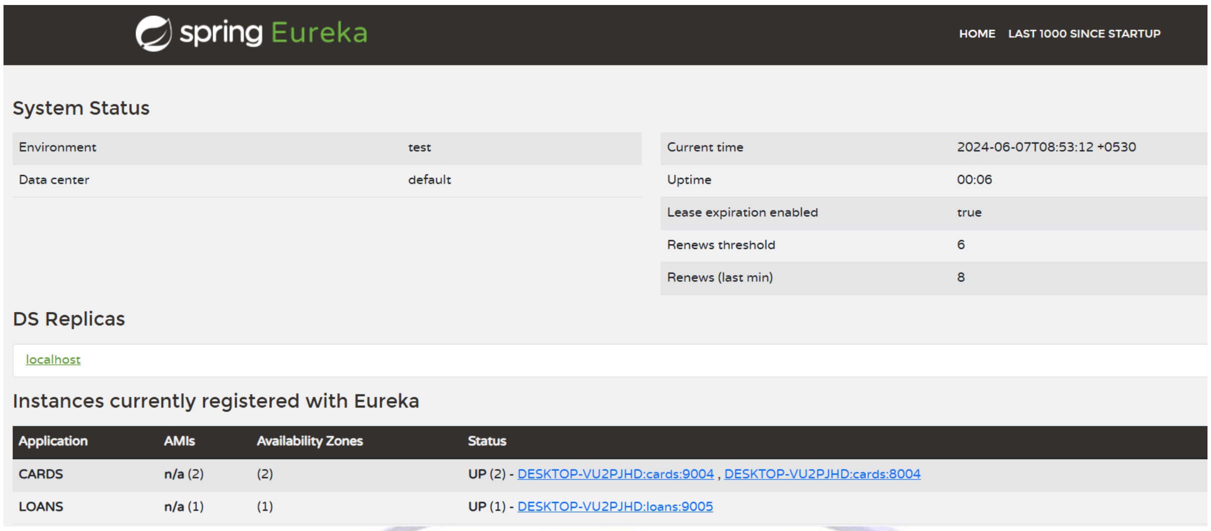
Current time	2024-06-07T08:49:00 +0530
Uptime	00:02
Lease expiration enabled	false
Renews threshold	6
Renews (last min)	4

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNTS	n/a (1)	(1)	UP (1) - DESKTOP-VU2PJHD:accounts:9003
CARDS	n/a (1)	(1)	UP (1) - DESKTOP-VU2PJHD:cards:9004
LOANS	n/a (1)	(1)	UP (1) - DESKTOP-VU2PJHD:loans:9005



System Status

Environment	test
Data center	default

Current time	2024-06-07T08:53:12 +0530
Uptime	00:06
Lease expiration enabled	true
Renews threshold	6
Renews (last min)	8

DS Replicas

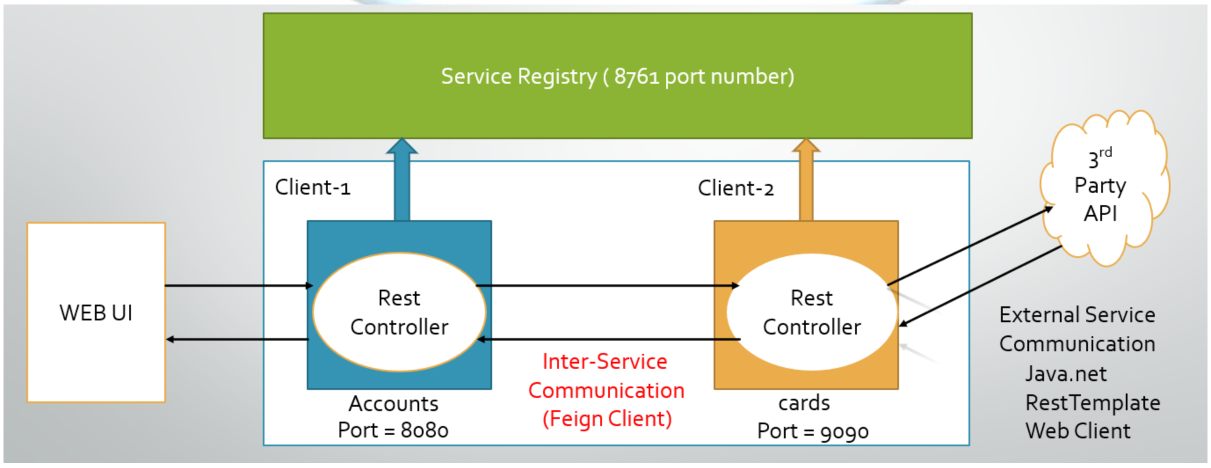
localhost

Instances currently registered with Eureka

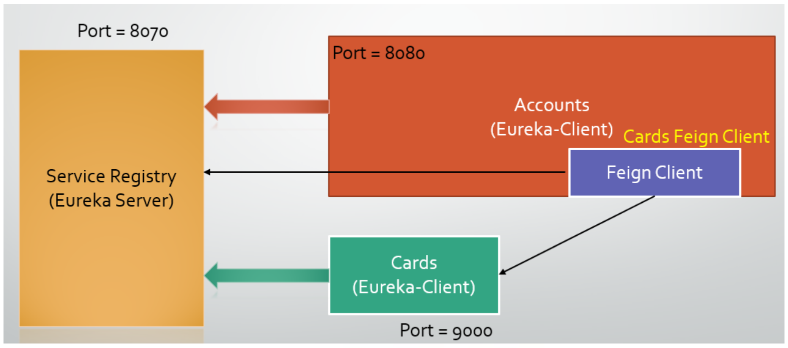
Application	AMIs	Availability Zones	Status
CARDS	n/a (2)	(2)	UP (2) - DESKTOP-VU2PJHD:cards:9004 , DESKTOP-VU2PJHD:cards:8004
LOANS	n/a (1)	(1)	UP (1) - DESKTOP-VU2PJHD:loans:9005

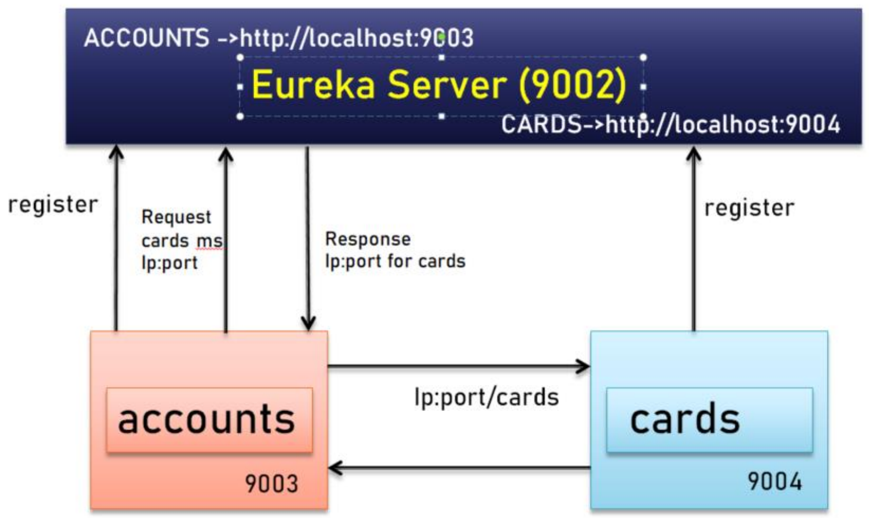
Spring Cloud Open Feign:

- Spring Cloud OpenFeign provides OpenFeign integrations for Spring Boot apps through auto-configuration and binding to the Spring Environment.
- Without Feign, in Spring Boot application, we use Rest Template to call the User service. To use the Feign, we need to add spring-cloud-starter-OpenFeign dependency in the pom.xml file.
- In our project, if one micro-services accessing another microservice then it is called as Inter-service communication.
  1) Rest Client
  2) Web Client
  3) Feign Client
- When we use FeginClient, we no need to configure API URL to access.
- Using api-name we can access api (FeginClient will get API URL from Service Registry)
- Feign Client is part of Spring Cloud Netflix libraries
- OpenFeign is a declarative REST client. It makes writing web service clients easier, just create an interface and add annotations on the interface.



Eureka Server & Multiple Eureka Clients & Feign Client:





Development Process for Feign Client:

1. Select existing project (accounts)
2. Add dependency
    a. spring-cloud-starter-feign
3. Add annotation in entry point for SpringBoot application
    a. @EnableFeignClients("pacakgeName")
4. Create a FeginClient Interface and annotate with @FeignClient and Pass the attributes name and URL.
    a. Create target API Request URLS (using JAX-RS annotations)
    b. Steps:
        i. Create target microservice feign client
        ii. Add which endpoint you want to access
        iii. Create controller to access your feign client through autowiring

Dependency

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

Enable Feign Clients

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class AccountsApplication {

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

Feign Client for Cards MS

```
@FeignClient(name="CARDS")
@LoadBalancerClient("CARDS")
public interface CardsFeignClient {

    @GetMapping("api/fetch")
    public CardsDto fetchCardDetails(@RequestParam String
mobileNumber);
}
```

Feign Client for Loans MS

```
@FeignClient(name="LOANS")
public interface LoansFeignClient {

    @GetMapping("api/fetch")
    public LoansDto fetchLoanDetails(@RequestParam String
mobileNumber);
}
```

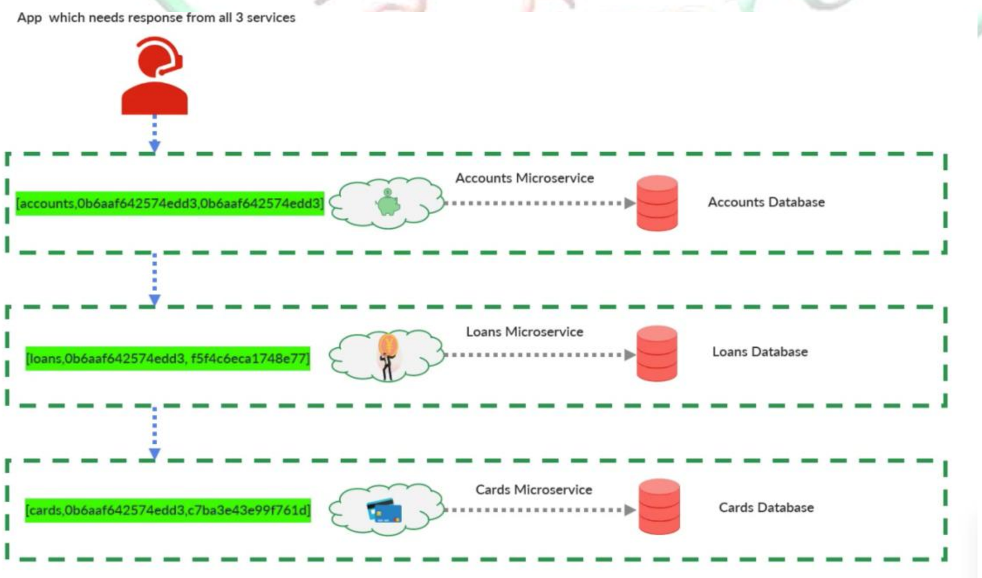
Distributed Tracing

1. **How do we debug where a problem in Microservices?**
    a. How do we trace one or more transactions across multiple services, physical machines and different data stores and try to find where exactly the problem or bug
2. **How do we aggregate all application logs?**
    a. How do we combine all the logs from multiple services into a central location where they can be indexed, searched, filtered and grouped to find bugs that are contributing to a problem?
3. **How do we monitor our chain of service calls?**
    a. How do we understand for a specific chain service call the path it travelled inside our microservices network, time it took at each microservice etc.

Micrometer Tracing

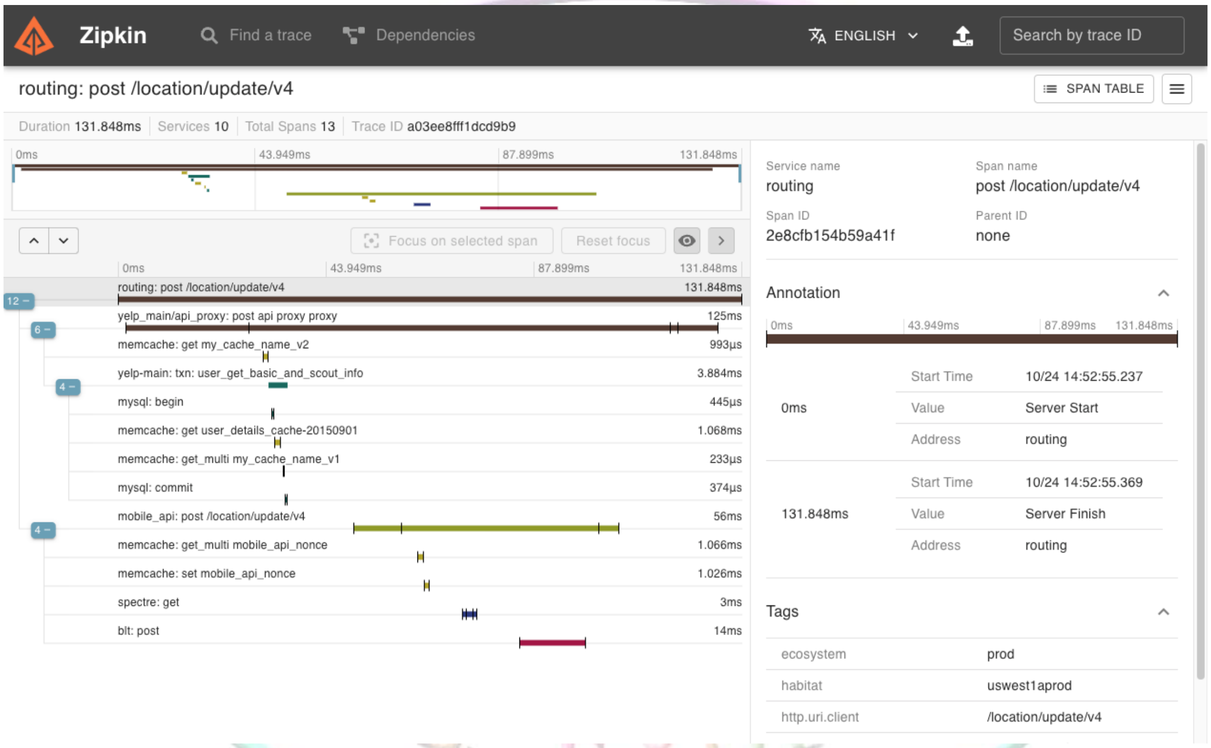
- In a Microservice ecosystem, a workflow/request can talk to multiple services. So tracing helps us to track the workflow in a productive way.
- Micrometer Tracing provides Spring Boot auto-configuration for distributed tracing.
- It adds trace and span ids to all the logs, so you can just extract from a given trace or span in a log aggregator
- Micrometer Tracing will add three pieces of information to all the logs written by microservices: [traceId,spanId, applicationName,]
- **traceId** : It's unique number that represents an entire transaction
- **spanId**:
    - It's a unique Id that represents part of the overall transaction.
    - Each service participating within the transaction will have its own span ID.

Trace Format


App which needs response from all 3 services
[accounts,0b6aaf642574edd3,0b6aaf642574edd3]
Accounts Microservice
Accounts Database
[loans,0b6aaf642574edd3, f5f4c6eca1748e77]
Loans Microservice
Loans Database
[cards,0b6aaf642574edd3,c7ba3e43e99f761d]
Cards Microservice
Cards Database

Zipkin

- Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures.
- If you have a trace ID in a log file, you can jump directly to it. Otherwise, you can query based on attributes such as service, operation name, tags and duration. Some interesting data will be summarized for you, such as the percentage of time spent in a service, and whether or not operations failed.
- The Zipkin UI also presents a Dependency diagram showing how many traced requests went through each application. This can be helpful for identifying aggregate behavior including error paths or calls to deprecated services.



routing: post /location/update/v4

Duration 131.848ms | Services 10 | Total Spans 13 | Trace ID a03ee8fff1dcd9b9

routing: post /location/update/v4	131.848ms
yelp_main/api_proxy: post api proxy proxy	125ms
memcache: get my_cache_name_v2	993µs
yelp-main: txn: user_get_basic_and_scout_info	3.884ms
mysql: begin	445µs
memcache: get user_details_cache-20150901	1.068ms
memcache: get_multi my_cache_name_v1	233µs
mysql: commit	374µs
mobile_api: post /location/update/v4	56ms
memcache: get_multi mobile_api_nonce	1.066ms
memcache: set mobile_api_nonce	1.026ms
spectre: get	3ms
blt: post	14ms

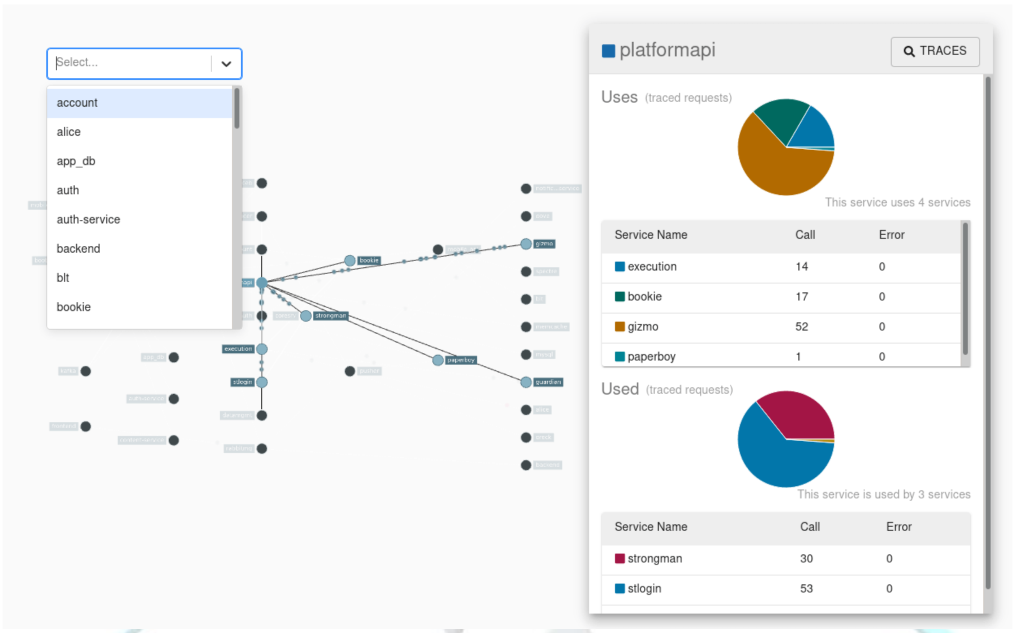
Service name	Span name
routing	post /location/update/v4
Span ID	Parent ID
2e8cfb154b59a41f	none

Annotation

0ms	Start Time	10/24 14:52:55.237
	Value	Server Start
	Address	routing
131.848ms	Start Time	10/24 14:52:55.369
	Value	Server Finish
	Address	routing

Tags

ecosystem	prod
habitat	uswest1aprod
http.uri.client	/location/update/v4



Service Name	Call	Error
🟦 execution	14	0
🟩 bookie	17	0
🟧 gizmo	52	0
🟦 paperboy	1	0

Service Name	Call	Error
🟥 strongman	30	0
🟦 stlogin	53	0

Distributed Tracing Properties

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
  zipkin:
    tracing:
      endpoint: http://localhost:9411/api/v2/spans
  tracing:
    sampling:
      probability: 1.0
logging:
  pattern:
    level: '%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-
}]'
  file:
    name: accounts_logs/accounts.log
```

Zipkin & Micrometer

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.reporter2</groupId>
    <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
```

Feign Micrometer

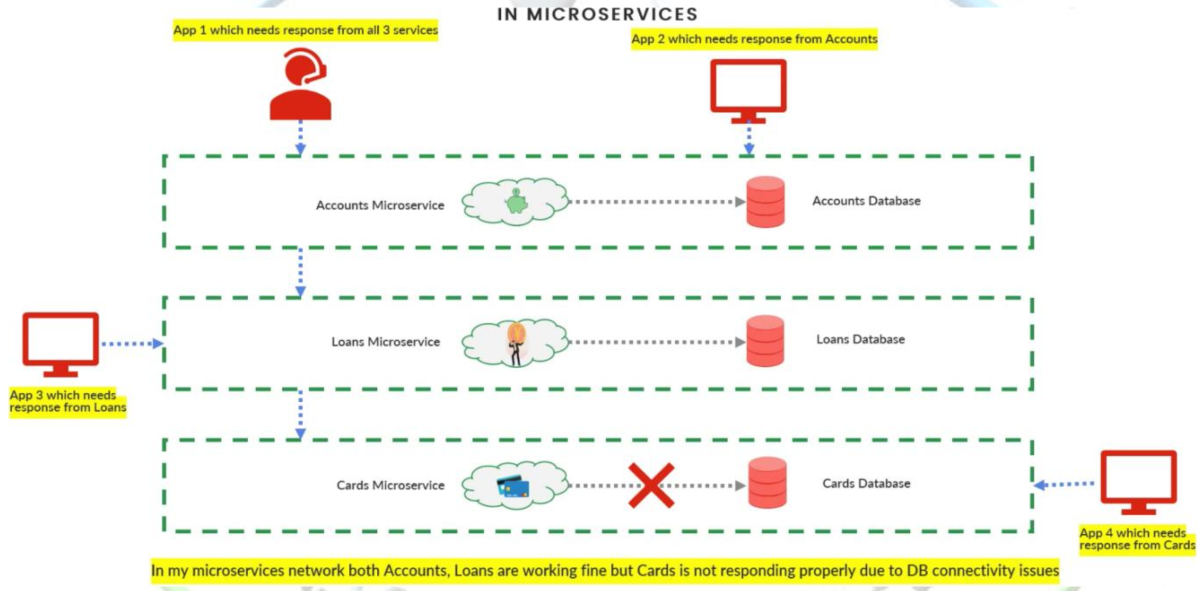
```xml
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-micrometer</artifactId>
</dependency>
```

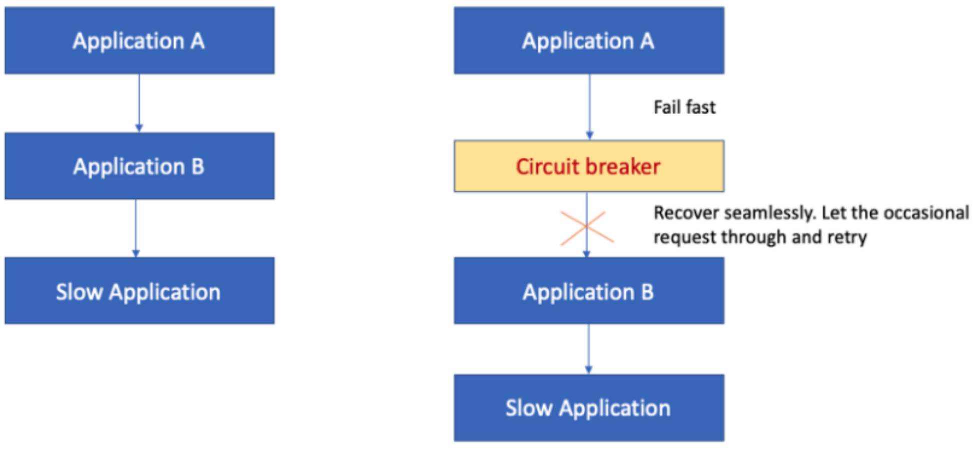
```java
@Bean
MicrometerCapability capability(final MeterRegistry registry) {
    return new MicrometerCapability(registry);
}
```

Spring Cloud Circuit Breaker – Resilience4J

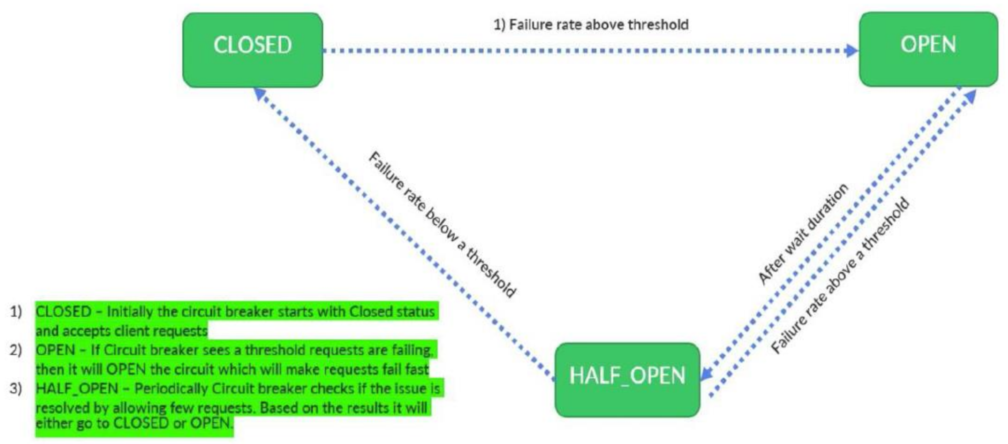
- Whether we are dealing with a Microservice or monolithic architecture, it is pretty common for a software service to call a remote software service
- However, when the remote service is down and thousands of clients are simultaneously trying to use the failing service, eventually all the resources get consumed.
- Circuit breaker pattern helps us in these kinds of scenarios by preventing us from repeatedly trying to call a service or a function that will likely fail and prevent waste of CPU cycles.
- Resilience4j is one of the libraries which implemented the common resilience patterns.
- Resilience4j has become more popular after Netflix announced that Hystrix will no longer be in active development and recommended Resilience4j as a replacement

Scenario:


IN MICROSERVICES
App 1 which needs response from all 3 services
App 2 which needs response from Accounts
Accounts Microservice
Accounts Database
Loans Microservice
Loans Database
App 3 which needs response from Loans
Cards Microservice
Cards Database
App 4 which needs response from Cards
In my microservices network both Accounts, Loans are working fine but Cards is not responding properly due to DB connectivity issues



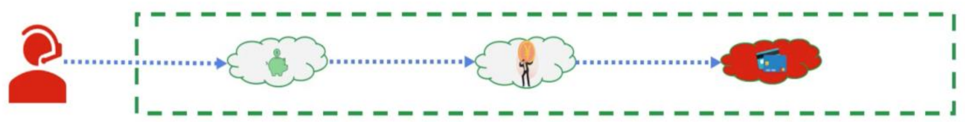
In Resilience4j the circuit breaker is implemented via a finite state machine with the following states


CLOSED
1) Failure rate above threshold
OPEN
Failure rate below a threshold
After wait duration
Failure rate above a threshold
HALF_OPEN

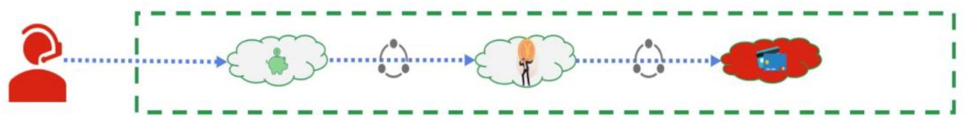
1) CLOSED – Initially the circuit breaker starts with Closed status and accepts client requests
2) OPEN – If Circuit breaker sees a threshold requests are failing, then it will OPEN the circuit which will make requests fail fast
3) HALF_OPEN – Periodically Circuit breaker checks if the issue is resolved by allowing few requests. Based on the results it will either go to CLOSED or OPEN.

Description:

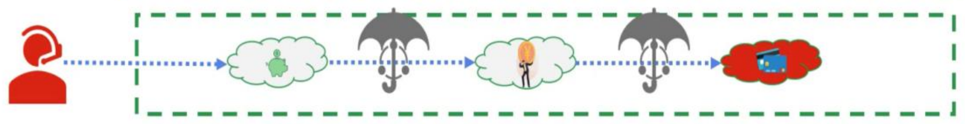
- Circuit breaker works as a state machine which has three states (CLOSED, HALF_OPEN, OPEN) as you see on the above diagram.
- Normally the circuit breaker operates in the CLOSED state.
- When a service tries to communicate with another service and the other system fails to respond, the circuit breaker keeps track of these failures.
- When the number of failures is above a certain threshold it goes into OPEN state. No further requests are sent to the service downstream.
- Meanwhile the circuit breaker may also help the system degrade gracefully by using some fallback mechanism, by returning some cached or default response to the caller.
- After specified wait duration, the circuit breaker goes into HALF_OPEN state and tries to reconnect to the service downstream.
- Again, if the failure rate is above a certain threshold, it goes back into OPEN state, else it goes into CLOSED state and returns to normal mode of operation.



Scenario 1 – If Cards microservice is responding slowly, then with out circuit breaker it will start eating up all the resources threads on the Loans and Accounts microservices which will make them also slow/down eventually



Scenario 2 – If Cards microservice is responding slowly, then with circuit breakers in between it will start acting and failing the services fast with the states OPEN, HALF_OPEN, CLOSED. This way at least Accounts and Loans services will not have any issues for other Apps.



Scenario 3 – If Cards microservice is responding slowly then with circuit breakers and fallback mechanism, we can make sure that at least we are failing gracefully with some default response is being returned and at the same time other microservices will not get impacted

Development Process:

1. Choose existing Project
2. Add below dependencies'
   a. Spring-cloud-starter-circuitbreaker-resilience4j
3. Enable the actuators
4. Configure fallback method for Circuit Breaker
5. Enable Circuit breaker properties in application. properties file

How to test Circuit Breaker in Microservices?

Using Actuators, we can watch events in below URL

- http://localhost:8080/actuators/health
- http://localhost:8080/actuators/circuitbreakers
- http://localhost:9003/actuator/metrics/resilience4j.circuitbreaker.state

Enable Circuit Breaker Events

```
spring:
  cloud:
    openfeign:
      circuitbreaker:
        enabled: true
```

Circuit Breaker Properties

```
resilience4j:
  circuitbreaker:
    configs:
      default:
        sliding-window-size: 5
        permitted-number-of-calls-in-half-open-state: 2
        failure-rate-threshold: 50
        wait-duration-in-open-state: 30s
```

Configure Fallback for Cards MS

```
@FeignClient(name="CARDS",fallback = CardsFallback.class)
@LoadBalancerClient("CARDS")
public interface CardsFeignClient {

    @GetMapping("api/fetch")
    public CardsDto fetchCardDetails(@RequestParam String
mobileNumber);
}
```

Configure Fallback for Loans MS

```
@FeignClient(name="LOANS",fallback = LoansFallBack.class)
public interface LoansFeignClient {

    @GetMapping("api/fetch")
    public LoansDto fetchLoanDetails(@RequestParam String
mobileNumber);
}
```

Cards Fallback Implementation

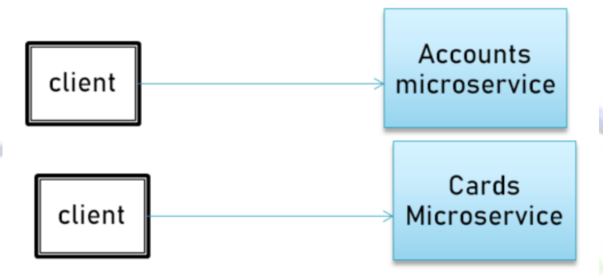
```
@Component
public class CardsFallback implements CardsFeignClient{

    @Override
    public CardsDto fetchCardDetails(String mobileNumber) {
       CardsDto cardsDto = new CardsDto();
       cardsDto.setStatus("Please try again after sometime");
       return cardsDto;
    }

}
```

Spring Cloud API Gateway (Cross Cutting Concerns)

- In a scenario where multiple clients directly connect with various services, several challenges arise.
- For instance, clients must be aware of the URLs of all the services and enforcing common requirements such as security, auditing, logging and routing becomes a repetitive task across all services.
- To address these challenges, it becomes necessary to establish a single gateway as the entry point to the microservices network



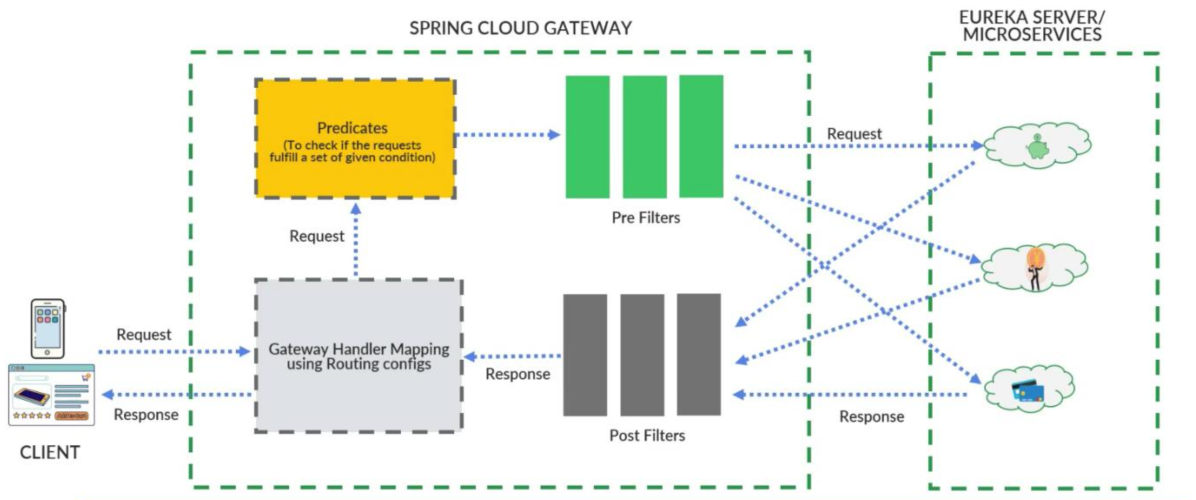
Spring Cloud Gateway

- The service Gateway sits as the gatekeeper for all inbound traffic to microservice calls within our application.
- Our service clients never directly call the URL of an individual service, but instead place all calls to the service gateway
- An API gateway is an API management tool that sits between a client and a collection of backend services.
- Spring Cloud Gateway is a library for building an API gateway, so it looks like any another spring Boot Application.
- API Gateway is used to manage all the apis which are available in our application.
- Spring Cloud gateway is the preferred API Gateway implementation from the Spring Cloud Team. It’s built on Spring 5, Reactor and Spring Webflux.

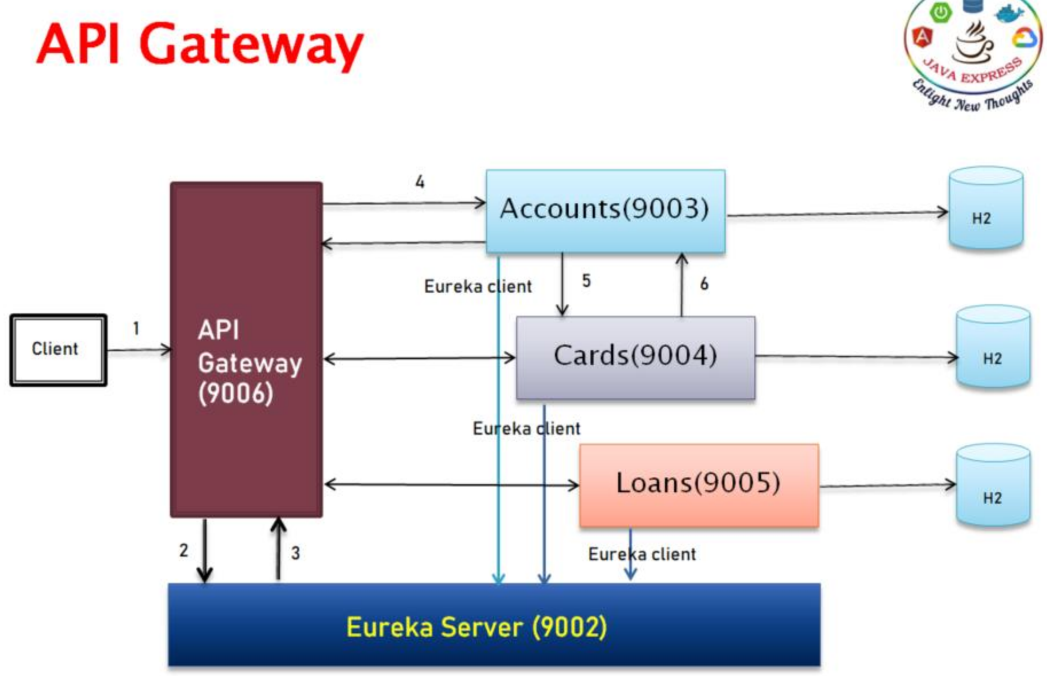
What is Request Proxy?

- Reverse proxy is something that is making requests on behalf of something else. It acts more like simple routing.

Internal Architecture:


SPRING CLOUD GATEWAY
EUREKA SERVER/ MICROSERVICES
Predicates (To check if the requests fulfill a set of given condition)
Pre Filters
Request
Request
Gateway Handler Mapping using Routing configs
Response
Post Filters
Response
Request
Response
CLIENT

When the client makes a request to the Spring Cloud Gateway, the Gateway Handler Mapping first checks if the request matches a route. This matching is done using the predicates. If it matches the predicate then the request is sent to the filters. Post filter it will send to the actual microservices or Eureka Server



Development Process:

1. Create SpringBoot Application
2. Add below dependencies
   a. Actuator
   b. Config client
   c. Spring-cloud-starter-gateway
   d. Eureka-client
3. Enable below Properties in application. properties

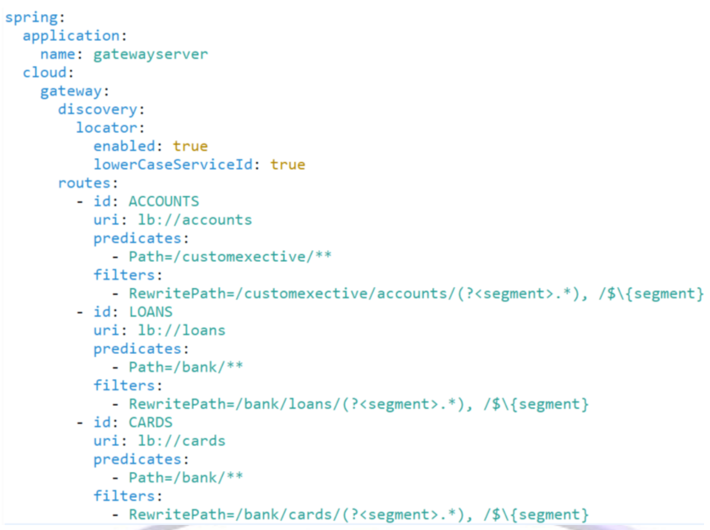
a. Port Number
b. Application Name
c. Enable Discovery Locator
d. Config import
e. Routes

Gateway Properties

```yaml
spring:
  application:
    name: gatewayserver
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lower-case-service-id: true
      httpclient:
        response-timeout: 5s
      routes:
        - id: accounts-service
          uri: lb://accounts
          predicates:
            - Path=/javaexpress/accounts/**
          filters:
            - RewritePath=/javaexpress/accounts/(?<segment>.*), /$\{segment}
            - name: CircuitBreaker
              args:
                name: accountsCircuitBreaker
                fallbackUri: forward:/contactSupport
            - name: Retry
              args:
                retries: 3
                statuses: BAD_GATEWAY
        - id: cards-service
          uri: lb://cards
          predicates:
            - Path=/api/v1/cards/**
          filters:
            - RewritePath=/javaexpress/accounts/(?<segment>.*), /$\{segment}
        - id: loans-service
          uri: lb://loans
          predicates:
            - Path=/api/v1/loans/**
          filters:
            - RewritePath=/javaexpress/accounts/(?<segment>.*), /$\{segment}
```

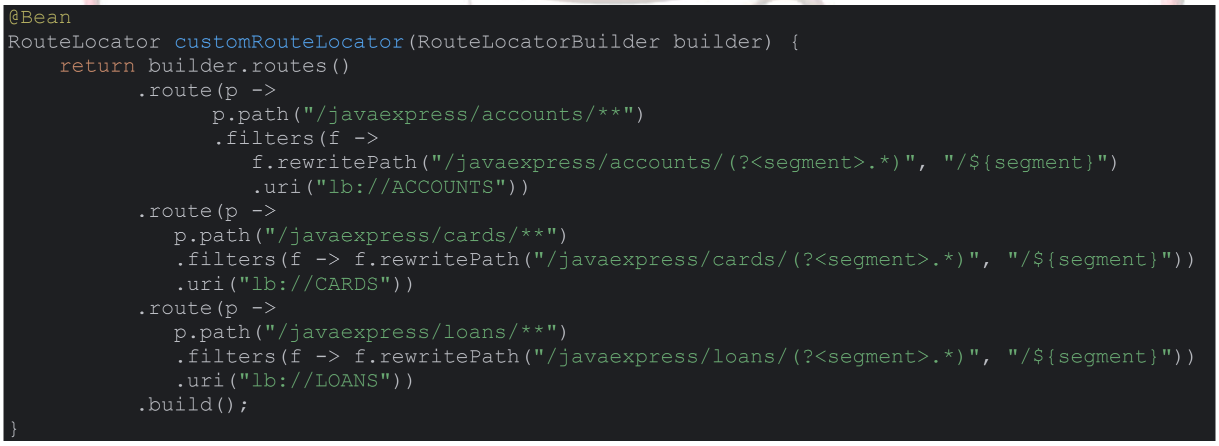
Enable Below Configuration for Gateway Routes

www.youtube.com/c/javaexpress
+91 7801007910
javaexpresschannel@gmail.com



```yaml
spring:
  application:
    name: gatewayserver
  cloud:
    gateway:
      discovery:
        locator:
          enabled: true
          lowerCaseServiceId: true
      routes:
        - id: ACCOUNTS
          uri: lb://accounts
          predicates:
            - Path=/customexective/**
          filters:
            - RewritePath=/customexective/accounts/(?<segment>.*), /$\{segment}
        - id: LOANS
          uri: lb://loans
          predicates:
            - Path=/bank/**
          filters:
            - RewritePath=/bank/loans/(?<segment>.*), /$\{segment}
        - id: CARDS
          uri: lb://cards
          predicates:
            - Path=/bank/**
          filters:
            - RewritePath=/bank/cards/(?<segment>.*), /$\{segment}
```

Configure the Routing Config



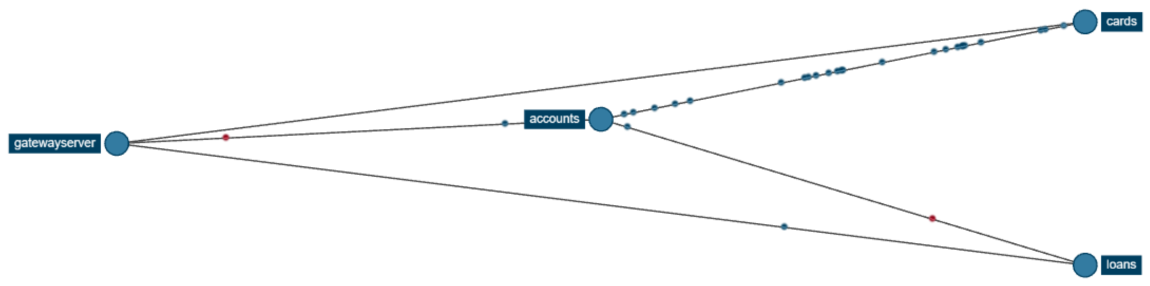
```java
@Bean
RouteLocator customRouteLocator(RouteLocatorBuilder builder) {
    return builder.routes()
          .route(p ->
                p.path("/javaexpress/accounts/**")
                .filters(f ->
                   f.rewritePath("/javaexpress/accounts/(?<segment>.*)", "/${segment}")
                   .uri("lb://ACCOUNTS"))
          .route(p ->
             p.path("/javaexpress/cards/**")
             .filters(f -> f.rewritePath("/javaexpress/cards/(?<segment>.*)", "/${segment}"))
             .uri("lb://CARDS"))
          .route(p ->
             p.path("/javaexpress/loans/**")
             .filters(f -> f.rewritePath("/javaexpress/loans/(?<segment>.*)", "/${segment}"))
             .uri("lb://LOANS"))
          .build();
}
```

Commercial API Gateways available in market

1. APIGEE
2. AWS API Gateway

How to test gateway APIs?

- http://localhost:8072/actuator
- http://localhost:8072/actuator/gateway/routes
- http://localhost:8072/javaexpress/accounts/api/fetchCustomerDetails?mobileNumber=780100791

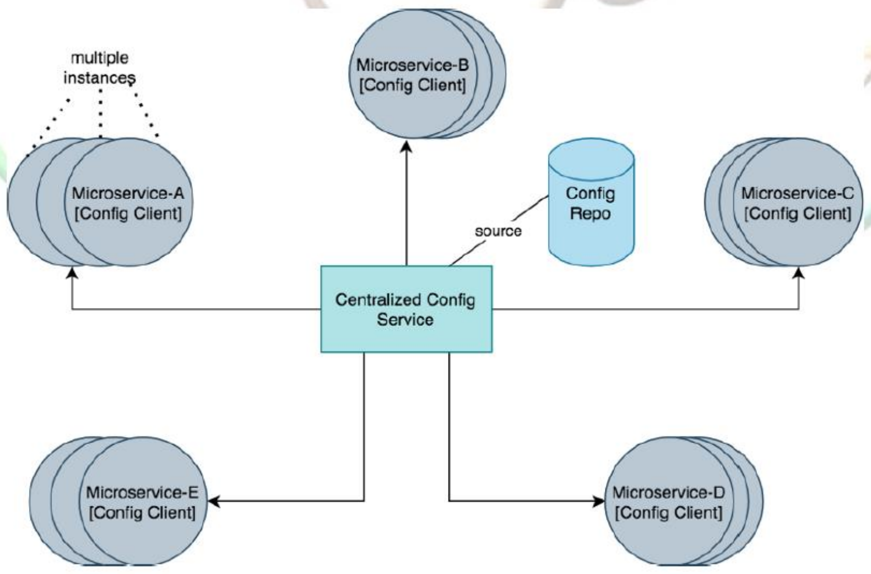

cards
accounts
gatewayserver
loans

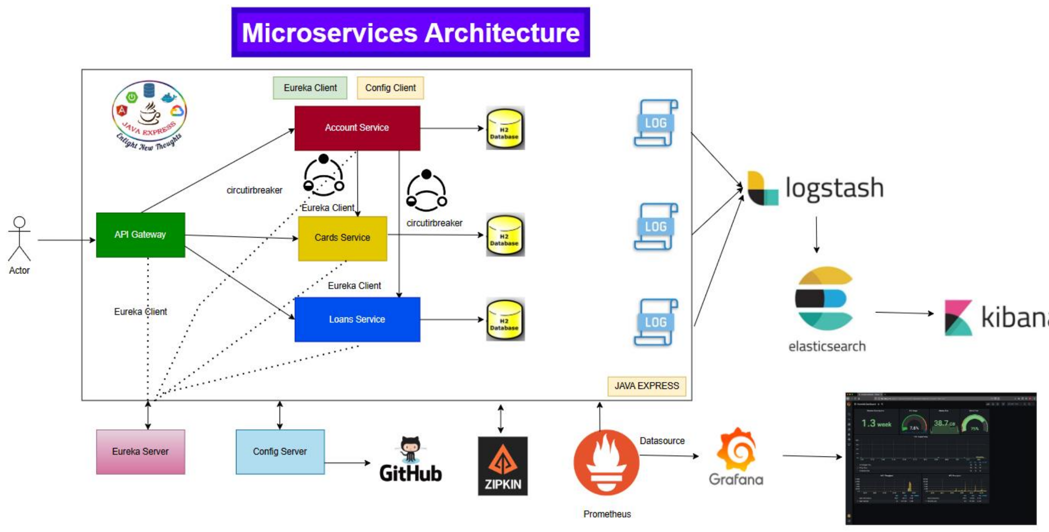
Spring Cloud Centralized Configuration

- Configuration management in microservices is a critical aspect of microservices architecture.
- This configuration information can include database connection strings, URLs for other services, and other environment-specific details.
- In microservices architecture, each service is typically a separate application with its own configuration.
- Managing these configurations can become complex, especially when you have many microservices running in different environments (development, testing, production, etc.).

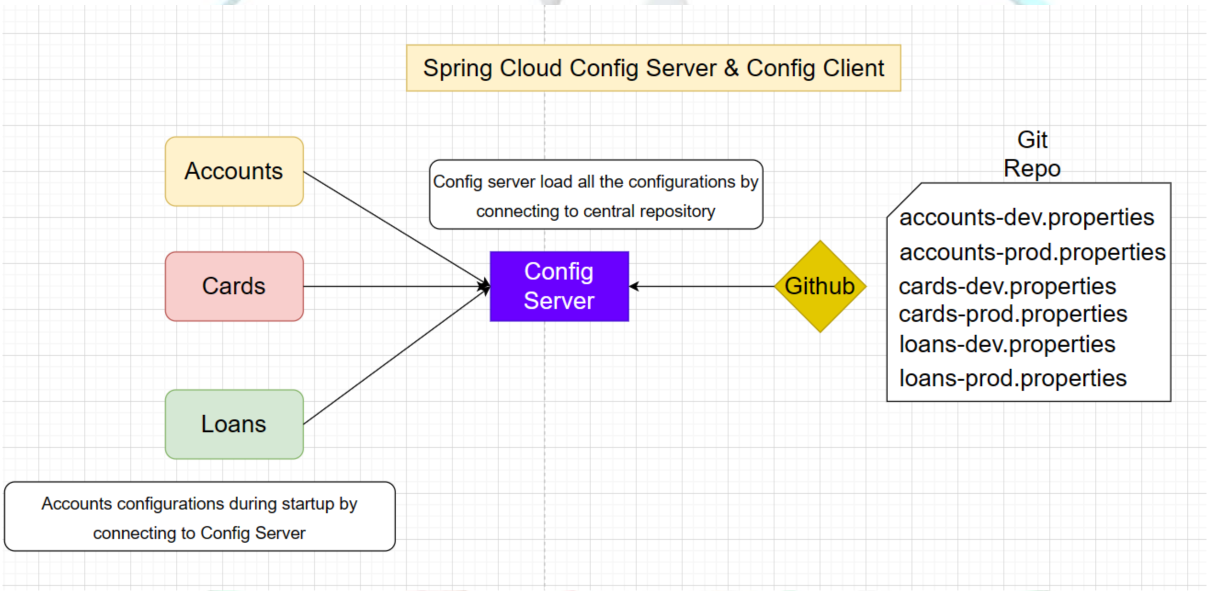
Configuration Management in Microservices

- One common approach to manage configurations in microservices architecture is using a centralized configuration server.
- In Spring Boot, the Config server and Config client work together to provide a centralized configuration management system for microservices.
- The Config server is a Spring Boot application that uses a Git repository to store configuration files for all of the microservices that are registered with it.
- When a service starts up, it queries the configuration server to get its configuration.
- The Config client is a Spring Boot application that retrieves configuration properties from the Config server and uses them in its own configuration.


multiple instances
Microservice-B [Config Client]
Microservice-A [Config Client]
Config Repo
source
Microservice-C [Config Client]
Centralized Config Service
Microservice-E [Config Client]
Microservice-D [Config Client]



Config Server & Config Client


Spring Cloud Config Server & Config Client
Accounts
Cards
Loans
Config server load all the configurations by connecting to central repository
Config Server
Github
Git Repo
accounts-dev.properties
accounts-prod.properties
cards-dev.properties
cards-prod.properties
loans-dev.properties
loans-prod.properties
Accounts configurations during startup by connecting to Config Server

Development steps for Config Server

1. Create a Spring Boot application for the Config Server.
2. Add the spring-cloud-config-server dependency to your pom.xml file.
3. Annotate the main class with @EnableConfigServer.
4. Configure the properties for the Config Server in the application.yml file, including the Git URL.

Dependency

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

Config Server Properties

```
management:
  endpoints:
    web:
      exposure:
        include: '*'
server:
  port: 9001
spring:
  application:
    name: configserver
  cloud:
    config:
      server:
        git:
          default-label: main
          force-pull: true
          timeout: 100
          uri: https://github.com/javaexpresschannel/javaexpress-config-latest
  profiles:
    active: git
build:
  version: 1
```

Development steps for Config Client

1. Use existing microservices and add necessary dependency in pom.xml file
2. Add the spring-cloud-starter-config dependency to your pom.xml file.
3. Configure Configuration Server url in the application.yml file

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

Config Client Properties

```
spring:
  application:
    name: accounts
  config:
    import: "optional:configserver:http://localhost:9001/"
  profiles:
    active: "prod"
```