Challenges while dealing external communication in microservices

Now inside this section, let us try to focus on the challenges that we may face while we are trying

to accept the external traffic into our microservices.

Like I said before, we should always have a single-entry point into our microservice network.

We should not allow different, different external clients talking with our microservices directly.

Instead, we should have a single-entry point into our microservices by using the products like API

Gateway.

So, let us try to understand what are the challenges that we may face While we are trying to accept the external communication into our microservice.

We are going to focus more on how we can perform routing, cross-cutting concerns inside our microservice whenever we are receiving the external traffic into our microservice network.

How we are going to maintain a single-entry point into your microservice network?

Here you may have a question like what is the problem if I do not maintain a single-entry point?

What if I allow all my external clients to communicate directly with my microservice?

Primary reason why we should maintain a single-entry point is all your external clients, they

can communicate with a single component inside your microservices network.

Otherwise, your external clients, they need to keep a track of all the different services that you have

inside your microservice network.

But the question is how we are going to build that and how we are going to maintain it.

The next question that I have here is, how we are going to handle the cross-cutting concerns like logging, auditing, tracing, and security whenever the external traffic is coming into your microservice network, like in any other web application inside microservices also, we need to make sure the incoming external request is properly authenticated and authorized, and we need to make sure we are doing enough logging, auditing and tracing, which will help us to identify the defects inside our code as part of debugging.

So, the question is how we are going to handle all these cross-cutting concerns.

Obviously, it will not be a good decision to implement all these logic inside the each of the microservices that we are going to build.

If you start building all these cross-cutting concerns inside all your hundreds of microservice,

you will end up with a lot of duplicate code inconsistency behaviour.

So that is why it is always recommended to implement all these cross-cutting concerns in a single place.

So, the question here is, how we are going to implement that.

And the next question that I want to ask you here is, how we are going to do the routing based upon your custom requirements.

Sometimes we may want to do some dynamic routing inside our microservice network, like if some external client is trying to invoke a particular path, I want to redirect that request to the microservice.

And very similarly, if someone is having some Http header inside their request based upon the header value, like inside the header, we can have some information like version 1 or version 2, which

indicates that my external client, they want to invoke a microservice with a particular version.

So how we are going to achieve all these dynamic routing capabilities inside your microservice network.

These challenges in microservice can be solved using a edge server.

We can call it as Edge server or API Gateway.

So, whatever the name it is, since this server is going to sit on the edge of your microservice network

and monitor all the incoming and outgoing requests.

That is why we call this as Edge server.

Some people, they will also call it as API Gateway, or Gateway.

Why we need an Edge Server or API Gateway inside microservices

Inside this lecture, let me try to convince you why we need a separate edge server inside our microservice network.

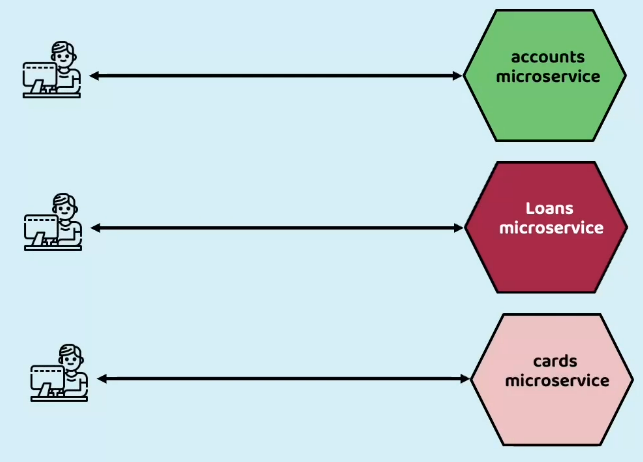
For few seconds let us assume you are not agreeing with me and you decided to not go with a separate edge server for all your external communication.

With that assumption, let us try to understand what are the challenges or what are the disadvantages that you may face inside your microservices development.

Like you can see first we will be having our individual microservices like accounts, loans, and cards

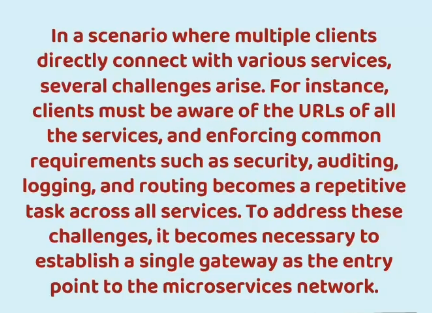
deployed in the respective servers and containers.

Now all the external clients who want to communicate with our microservices, they are going to send the request directly to the respective microservice.



Here we have three different client applications.

They are trying to communicate with the three different microservices.



In real projects you may have more than 100 microservices as well, and all those individual microservices, if they start receiving the external communication directly from the clients, we will be facing a lot of challenges and disadvantages.

For example, if you want to implement some non-functional requirements or cross-cutting concerns like security, auditing, logging, and routing, you need to repeat the business logic related to these and cross-cutting concerns inside all of your microservices.

And in the same process, different developers may get assigned to build these cross-cutting concerns

in different microservices.

Here, there is no guarantee that all the developers, they will follow same standards while implementing the cross-cutting concerns.

With that, there will not be any consistency inside your microservices in terms of enforcing security,

auditing, logging, and routing etc.

To overcome this challenge of inconsistency from various developers, you may try to propose a solution saying that, why can't we build all this logic inside a common library and try to add this common library as a dependency for all the microservices?

That is a great advice, but let me try to explain what are the disadvantages that we may have here.

Whenever we try to build these non-functional requirements and cross-cutting concerns into a common shared library, you are bringing a tight coupling between your common library and your microservices.

In future, whenever you are trying to change some business logic on the security implementation or auditing or logging, you need to make sure that your common library is working with all the microservices without affecting them negatively.

Which means you need to do regression testing and enough impact analysis.

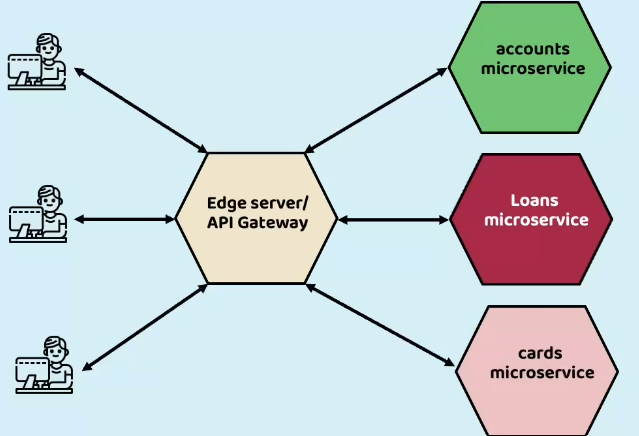
If you have a greater number of microservices inside your organization, doing the impact analysis and doing the regression may not be a feasible option.

That is why even having all these cross-cutting concerns inside a common library is also a not recommended approach.

To address all these challenges, it will be a wise decision to have a single gateway which will act as an entry point into the microservice network.

So, let us try to understand how the Edge server or a Gateway server is going to help us to overcome these challenges.

First, we will be having similar accounts, loans, and cards microservices inside our microservice network.



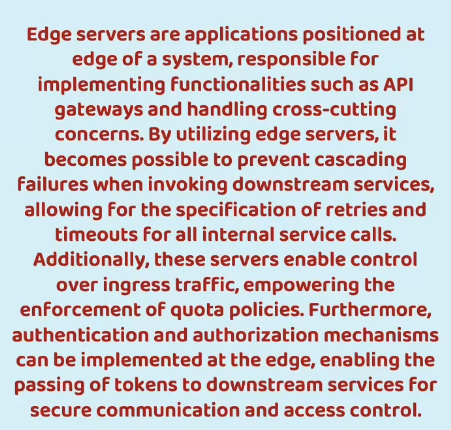
Now, in between the client applications and these individual microservices, there will be an Edge

server or the API Gateway server sitting and it is going to accept all the requests from the external

clients and execute any business logic related to cross-cutting concerns.

Once everything about the request is validated, then only it is going to forward the request to the

actual microservice.



So, the advantage with this approach is, your edge server can implement a lot of functionalities, including cross-cutting concerns like security, logging, auditing.

Apart from these cross-cutting concerns, with the help of these edge server, you can also prevent

cascading failures and make your downstream services fault tolerant and resilient to the exceptions or errors.

So, I am going to explain how to make our microservices fault tolerant with the help of Edge Server in

the coming sections.

But for now, please note that using Gateway or Edge server, we can also make our downstream services fault tolerant and resilient in nature.

Apart from these functionalities, we can also implement retries timeouts for all the internal service

calls with the help of this edge server.

And if needed, we can also enforce some quota related policies on the incoming traffic from a particular client.

For example, think like you have various plans inside your organization and based upon a plan that

is subscribed by a client, you want to enforce some quota limitations.

Maybe you have plans like standard, premium and advance.

So based upon these plans, you want to enforce different, different quota limitations on a respective

client.

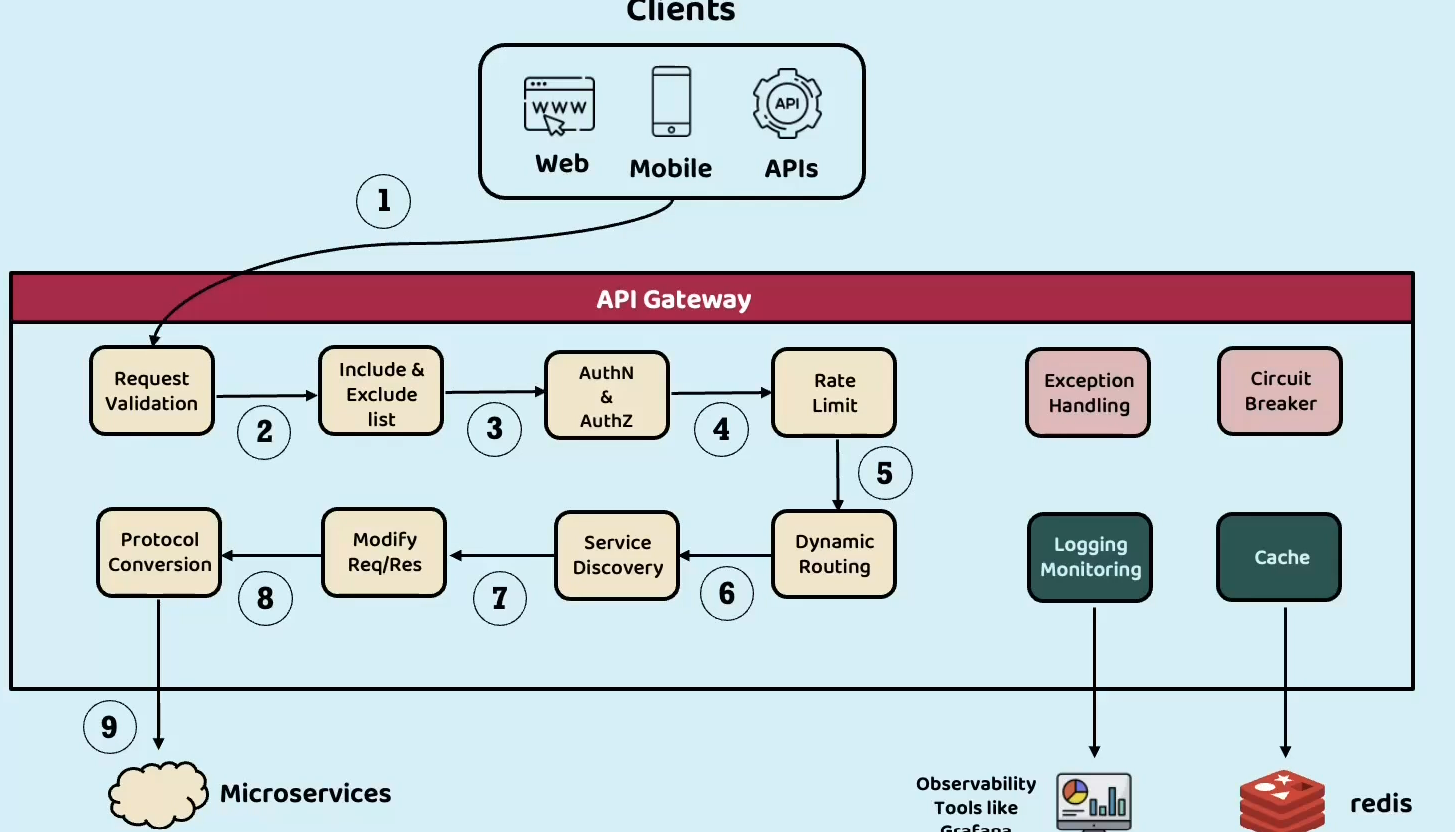
All such quota limitations you can also achieve with the help of this edge server or API gateway.

Since we have all these advantages, it will be a wise decision to have a separate edge server for all

your microservices.

Still, if you are not convinced with me on why we should have a separate edge server, let me try to

show you a different perspective on what API Gateway or Edge server can handle inside your microservices.



Like you can see here, we have clients’ applications and microservices.

In between the microservices and client applications, our API Gateway or Edge Server is going to act as a mediator.

Inside this API gateway I can implement many functionalities in the very first step, we have the external clients sending the request to the API gateway and my API gateway can perform validations on the request.

It can do some include and exclude list where it can perform some checks related to blacklisting or

whitelisting of the IP address.

And very similarly, it can also perform some authentication and authorization.

And after the authentication and authorization, we can also enforce some rate limit, which means we

can limit the quantity of requests or the number of requests coming from a particular client.

And very similarly, API Gateway can perform dynamic routing, service discovery, modify request and response, and if needed, it can also perform protocol conversion.

Maybe all the external traffic you are trying to accept with the Https protocol, but you want to forward the request using Http protocol to all your microservices.

So, such kind of protocol conversion you can perform with the help of API gateway.

So, you can activate or deactivate all these functionalities of API gateway based upon your business requirements.

There is no mandatory that you need to implement all these components inside your API gateway.

And at the same time, this is not the full exhaustive list on what API gateway is capable of.

API gateway can do more, but I am trying to highlight here most used functionalities of

API gateway.

So, after performing all these validations and executing all the business logic related to these components, API gateway is going to forward the request to the actual microservices.

Apart from these regular positive scenarios, we can also handle negative scenarios like implementing

the exception handling, implementing circuit breaker to make our microservices fault tolerant and resilient in nature.

We are going to discuss about Circuit Breaker in the coming sections, but for now, please note that

using this circuit breaker implementation, we can make our microservices resilient in nature.

And apart from these positive and negative scenarios, API gateway is also capable of sending all

the logging and monitoring related information to tools like Grafana, where using these tools we

can look all the logs or errors or monitoring related information using beautiful dashboards.

And if needed, we can also integrate our API gateway with an Redis cache, which means you can write some business logic by leveraging the cache that you can store inside this Redis component.

So, these are all the capabilities of API Gateway.

**And here you may have a question like why can't our Eureka Server do all these tasks?**

It is a very valid question, but the purpose of Eureka Server is strictly to implement a separate pattern, which is service discovery and service registry.

Apart from service discovery and service registration, your Eureka Server is not capable of performing all these non-functional related requirements or implementing cross-cutting concerns.

That is why we need to go with the API gateway and this will also give flexibility to the organizations.

If they do not want API gateway related functionality, they can simply ignore it and go with the Eureka server.

If someone need both, they can use both these components having different, different servers or components handling different, different problems inside microservices will also give options to the organizations which one they want to pick and which one they want to ignore.

Now, I am assuming you are convinced that we need to implement this gateway inside our microservices.

The very immediate question that you may have here is, this looks very complex to me how we are going to build such an edge server inside our microservices.

But do not worry my friend, we have our friend like Spring Boot and Spring Cloud.

Using these frameworks, we can easily build the gateway or edge server inside microservices.

In the next lecture, let us try to understand more details on how Spring Cloud is going to help us in

implementing this gateway inside our microservices.

Introduction to Spring Cloud Gateway

Using this Spring Cloud Gateway, we can create the edge service very easily, which is of production

standard and production ready.

Moreover, the Spring Cloud Gateway is built based upon the spring reactive framework.

Due to that, it is going to seamlessly process any amount of workload that you can receive inside your microservices network.

Since this gateway is going to be the front face to receive all external communication, we can

expect a lot of traffic to be handled by this gateway.

That's why to make this work efficiently with very less memory and with very less amount of threads,

the spring team, they build this project based upon the spring reactive framework.

Whenever we built an application with the help of Spring Cloud Gateway, it is going to sit as an gatekeeper for all the inbound traffic that your microservices can expect.

Since this is going to sit in between your client applications and the actual microservices, your client

applications, they can never directly interact with your actual individual microservices because they

never know what is the actual physical address location of an particular microservice instance.

On top of that, you might have enforced some security.

With all these restrictions.

It will be impossible for my external applications to directly invoke my microservices and with that

there is no option for them except sending the traffic to the gatekeeper, which is Spring Cloud Gateway.

Here you may think like building this gatekeeper using Spring Cloud Gateway is super, super complex process, but believe me, this is going to be building like any other spring boot application.

Apart from just sitting in between the clients and the microservices, whatever application you are

going to build with the help of Spring Cloud Gateway, it is capable of executing a lot of non-functional

requirements and cross-cutting concerns.

For example, you can dynamically route the incoming request based upon the context.

Maybe inside an request header you have an API version value.

So based upon this version value, your gateway server is capable of routing that request to the appropriate versioned backend microservice.

It can also handle complex requirements like sticky sessions.

If you want to make sure that a particular user request or client request should always go to the same instance where the initial request is processed.

So, this kind of sticky sessions or user sessions we can also achieve with the help of Gateway Server.

But Spring Cloud Gateway is the most preferred API gateway compared to Jewel, because Spring Cloud Gateway built on the spring reactor, which and it also has integration with Circuit Breaker, Service discovery and it is non-blocking in nature.

Spring Cloud Gateway, it is going to sit between your client applications and the individual microservices and act as a central policy enforcement point.

So, at this centralized location, we can enforce any kind of policy requirements like routing both static

and dynamic security, logging, auditing, metric collection, whatnot.

<https://spring.io/projects/spring-cloud-gateway>

Here there is a summary about Spring Cloud Gateway, what it is capable of and how it is going to help you.

So, you can see here Spring Cloud Gateway aims to provide simple and effective way to route to APIs and providing cross-cutting concerns such as security monitoring and resiliency.

Apart from this spring Cloud gateway features also you can see here it is built upon the Spring project reactor and it is capable to match the roots on any request attribute.

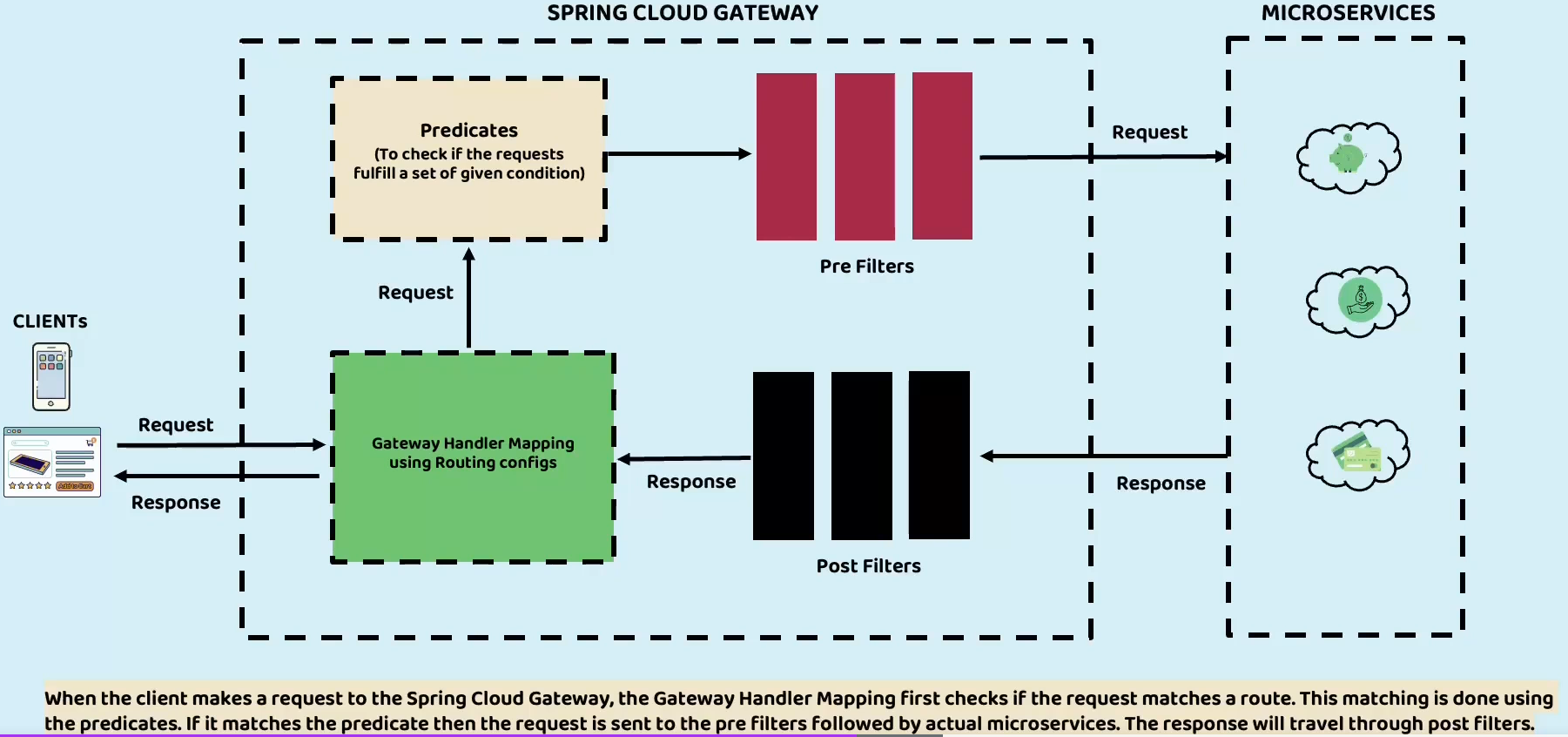
It can use predicates and filters that are specific to roots.

It has a good integration with the circuit breaker and spring cloud discovery client like Eureka Server.

We can easily implement predicates and filters.

We can request rate limiting path rewriting.

Deep Dive on Sprong Cloud Gateway Internal architecture



This is the internal architecture of Spring Cloud Gateway.

Let us try to understand more details.

First, we always receive the request from our external clients, and these client applications can be in the form of mobile applications or website or it can be other REST based API as well.

So, when my client applications, when they send the request to my gateway server, which is acting as

an edge server inside our microservices, it is going to take help from the **gateway handler mapping**.

So, this is an important component which is going to help my gateway server to identify what is the path that has been invoked by my client application and to which microservice it has to redirect the request.

And there is no artificial intelligence here.

My gateway handler mapping it cannot take a decision to which microservice it has to forward the request.

To help my gateway handler Mapping my developer must do some routing related configurations, saying that if someone send a request to this particular path, please redirect that request to a particular microservice.

I am going to show you in the coming lectures how to perform those routing configurations.

For now, please note that this gateway handler mapping is an important and very first component inside the Spring Cloud Gateway, which is going to help to identify the path and routing the request to the corresponding microservice.

After my spring Cloud Gateway identifies to which microservice the request must be forwarded.

First it will check if there are any **predicates** or **pre-filters** configured by the developer.

So, these two components will be executed before forwarding the request to the actual microservice.

So, let us try to understand what are these components?

A predicate is a logic which will return a boolean value.

You can define some conditions to your spring cloud gateway, saying that if a particular condition

is met, then only forward the request to the microservices.

Otherwise simply reject it with whatever reason.

So, such kind of conditions we can define with the help of predicates.

*If you are familiar with functional interface and lambda expressions which are introduced in Java 8,*

*we have a functional interface with the name predicate.*

*So, what that functional interface is going to do is, it is going to execute some business logic and will*

*return a boolean value true or false. Based upon the boolean value, the next logic can be executed. Very similarly here also, these predicates will help my Spring cloud gateway to decide whether the conditions defined by my developer is met or not.*

*If they are not met, then it is not going to forward the request and simply deny the request with a*

*proper error to the client application.*

Let us assume all the predicates defined by the developer are passed and now Spring Cloud Gateway will look for are there any pre-filters configured by the developer.

Inside these pre-filters we can execute any business logic like you can do some request validations, you can do some auditing, logging, you can modify the request, you can perform some security checks.

So, your requirement can be anything and you may have any number of requirements.

All such non-functional requirements are cross-cutting requirements.

You can implement using any number of pre-filters.

Once all these pre filters are executed.

Finally, your request will be forwarded to the actual microservice like loans, cards, or accounts

microservice.

Once my microservice processes the request, it is going to send the response and the response will

be intercepted by the post filters.

The pre-filters will act on the request.

Very similarly, the post-filters, they will act on the response.

So, before you try to send the response to the actual client application, you may want to modify some response or you may want to perform some validations.

All such validations and business logic you can define inside the post-filters and after executing all

the post filters configured by the developer, the same response will be sent back to the gateway handler mapping and from gateway handler mapping it will send to the actual client application.

So, this is the internal architecture.

Building Edge Server using Spring Cloud Gateway

Create new spring starter project for API gateway  
D:\Experiments\Microservices\sb-bank-application\gatewayserver\pom.xml  
  
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent>

        <groupId>org.springframework.boot</groupId>

        <artifactId>spring-boot-starter-parent</artifactId>

        <version>3.3.0</version>

        <relativePath/> <!-- lookup parent from repository -->

    </parent>

    <groupId>com.eazybytes</groupId>

    <artifactId>gatewayserver</artifactId>

    <version>0.0.1-SNAPSHOT</version>

    <name>gatewayserver</name>

    <description>Demo project for Spring Boot</description>

    <properties>

        <java.version>17</java.version>

        <spring-cloud.version>2023.0.2</spring-cloud.version>

    </properties>

    <dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-actuator</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-config</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-gateway</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.cloud</groupId>

            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-devtools</artifactId>

            <scope>runtime</scope>

            <optional>true</optional>

        </dependency>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-test</artifactId>

            <scope>test</scope>

        </dependency>

        <dependency>

            <groupId>io.projectreactor</groupId>

            <artifactId>reactor-test</artifactId>

            <scope>test</scope>

        </dependency>

    </dependencies>

    <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>

    <build>

        <plugins>

            <plugin>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-maven-plugin</artifactId>

            </plugin>

        </plugins>

    </build>

</project>

package com.eazybytes.gatewayserver;

@SpringBootApplication

public class GatewayserverApplication {

    public static void main(String[] args) {

        SpringApplication.run(GatewayserverApplication.class, args);

    }

}

Note: Update all project docker image tag to s9

Add the jib plugin for image creation

<build>

        <plugins>

….

            <plugin>

                <groupId>com.google.cloud.tools</groupId>

                <artifactId>jib-maven-plugin</artifactId>

                <version>3.4.1</version>

                <configuration>

                  <to>

                    <image>nileshzarkar/${project.artifactId}:s9</image>

                  </to>

                </configuration>

              </plugin>

        </plugins>

    </build>

As a next step, we need to define certain configurations inside the **application.properties** file of gateway server.

D:\Experiments\Microservices\sb-bank-application\gatewayserver\src\main\resources\application.properties

spring.application.name=gatewayserver

spring.config.import=optional:configserver:http://localhost:8071/

management.endpoints.web.exposure.include=\*

management.endpoint.gateway.enabled=true

## Configuring info endpoint

info.app.name=Gateway Server Microservice

info.app.description=Eazy Bank Gateway Server Application

info.app.version=1.0.0

management.info.env.enabled = true

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

# spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true

logging.level.com.eaztbytes.gatewayserver: DEBUG

D:\Experiments\Microservices\sb-bank-application-config\gatewayserver.properties

<https://github.com/nileshzarkarudemy/sb-bank-application-config/blob/main/gatewayserver.properties>

server.port=8072

eureka.instance.preferIpAddress=true

eureka.client.registerWithEureka=true

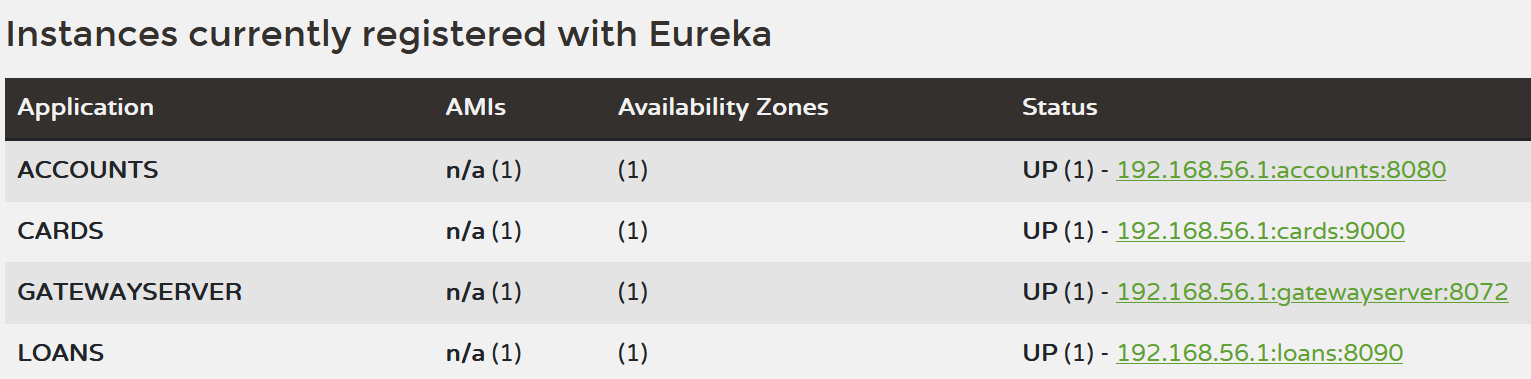
eureka.client.fetchRegistry=true

eureka.client.serviceUrl.defaultZone=http://localhost:8070/eureka/

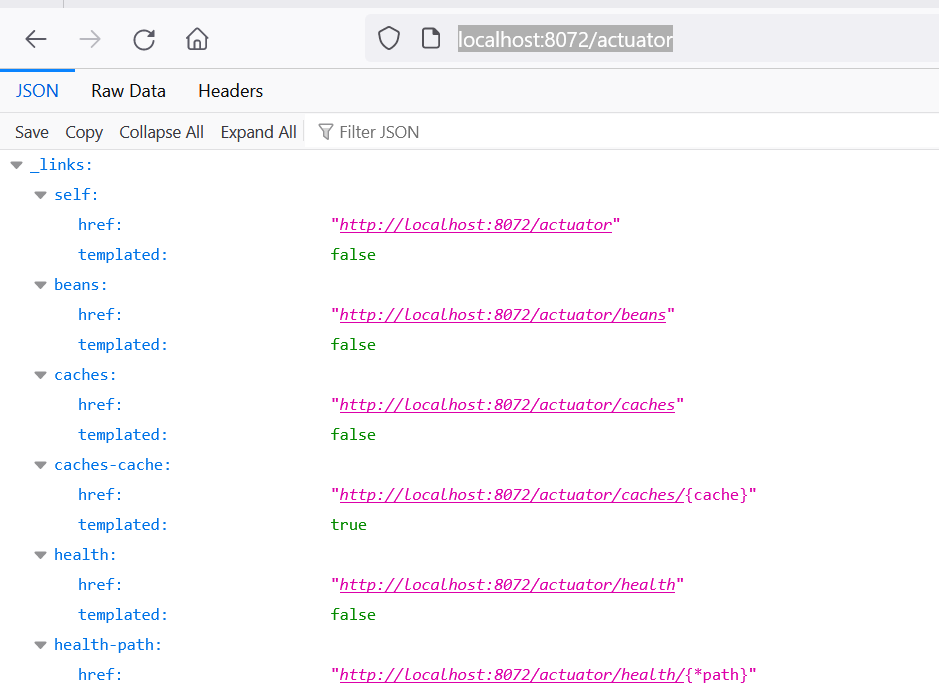
Demo of Edge Server with default routing configs

Start the configserver, eurekaserver, accounts, loans cards microservices.  
  
Now start the gateway server.

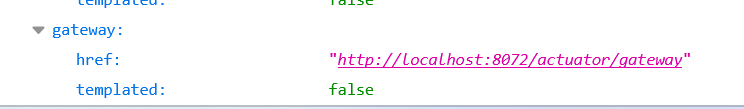
<http://localhost:8070/>



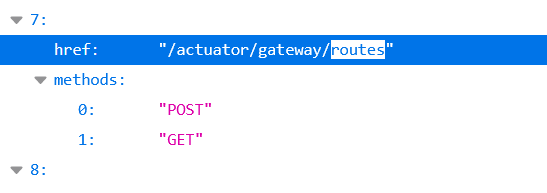
<http://localhost:8072/actuator>



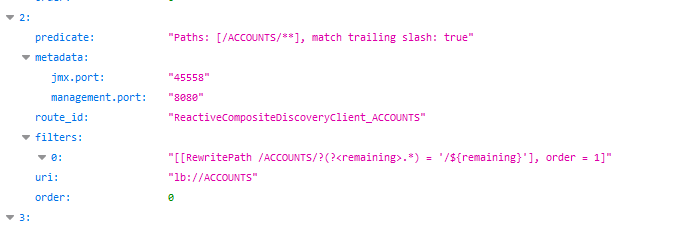
Look for below endpoint

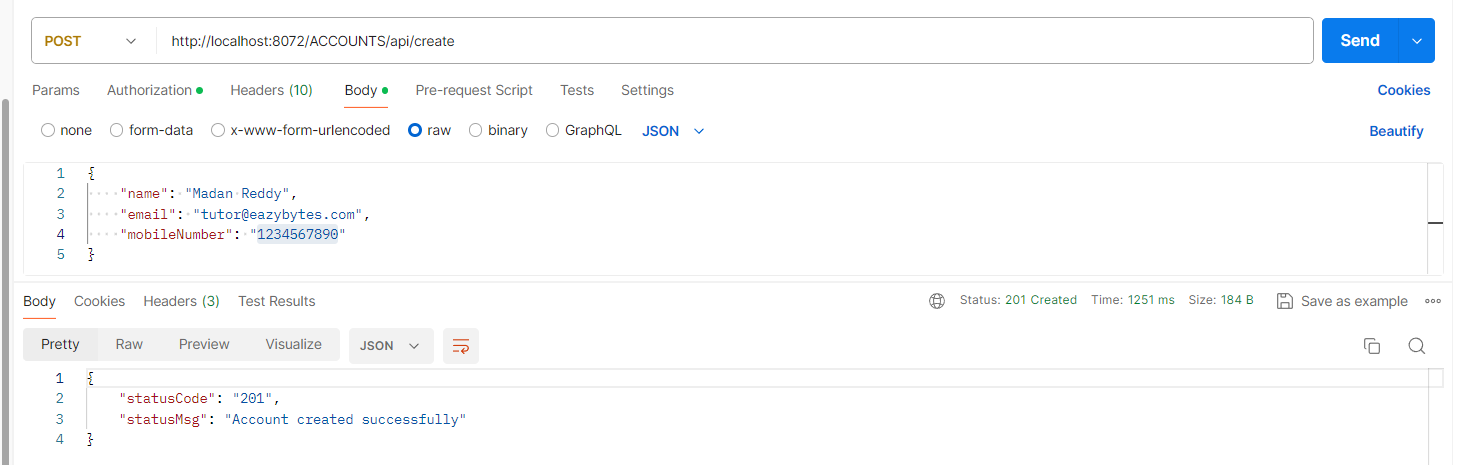


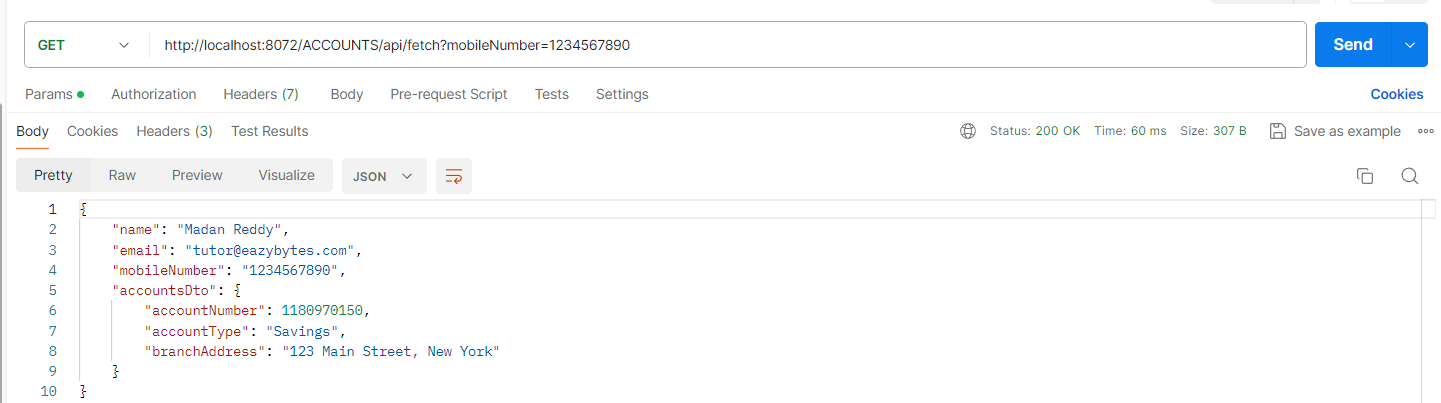
<http://localhost:8072/actuator/gateway>



<http://localhost:8072/actuator/gateway/routes>







So, this way now all my external traffic, I can request my client applications to send to my edge server

and my edge server is going to forward the request to the actual individual microservices.

So, we are not exposing our individual microservices to the external client applications.

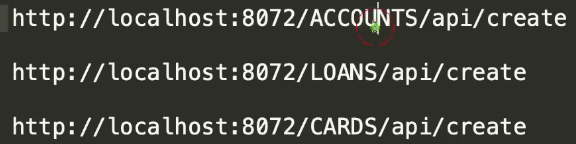
Here you may have a question like who is stopping our clients to directly invoking our loans microservice and accounts microservice.

Off course, with this code setup my clients can invoke the direct URLs also, but in the coming sections

when we try to enhance these applications, such that my clients cannot invoke the microservices directly because I am going to enforce some security, so that they must come only through the gateway server.

But with this simple setup of Gateway server, we can set up the edge server inside our microservices network.

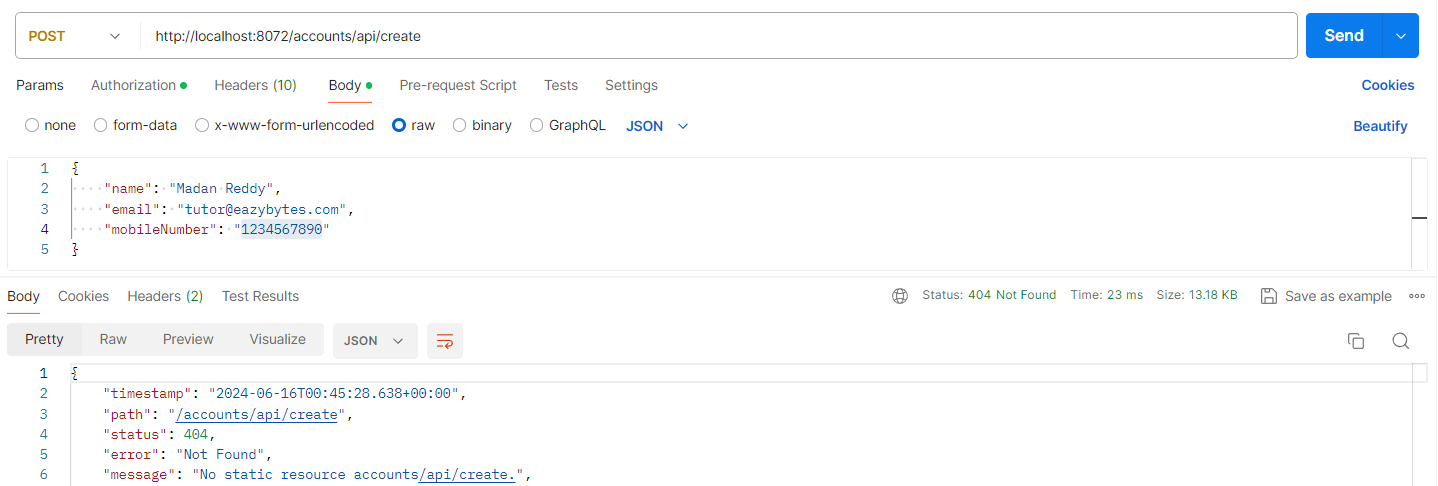
Make changes inside Gateway server to accept service names with lower case

  
As of now, we are trying to send the request to the Gateway server by using the application name in

complete capital letters like you can see accounts, loans, and cards.

If I try to send in a small case letter, it is not going to work.

You can see I'm getting an error, which is 404 not found.



If you are fine using completely capital letters, you should be good with the default behaviour of the

gateway server.

But mentioning a value inside the path in complete capital letters is very odd.

Even we cannot request our client applications to send only the application name or the microservice

name in capital letters.

That will be a very weird ask or weird behaviour inside our application.

So, to overcome this, we can introduce a new property inside the application.properties of Gateway server.

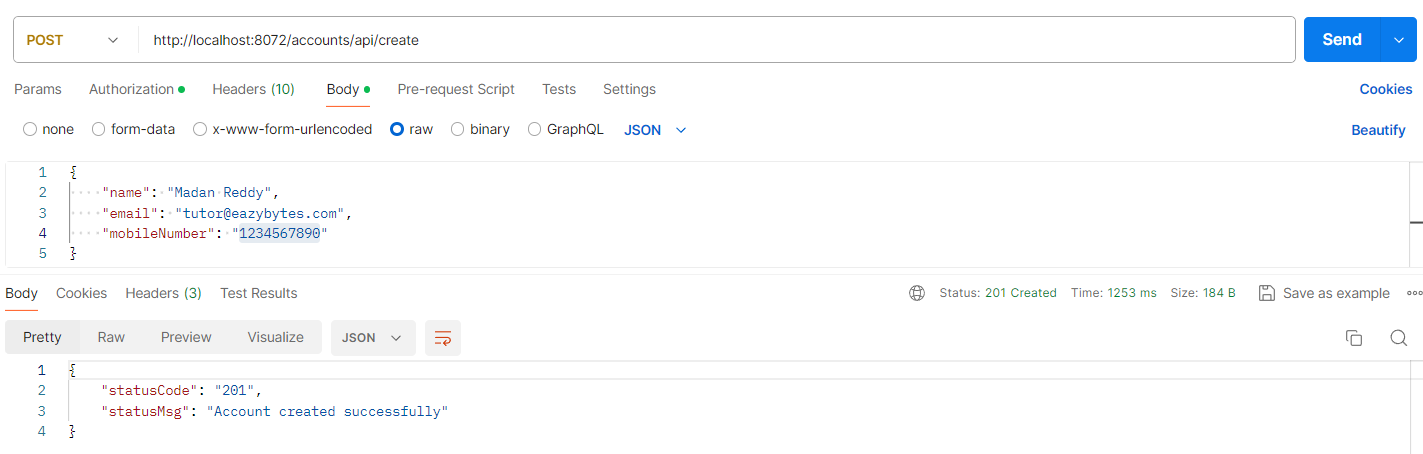
I'm going to mention a new property which is lowercase service ID.

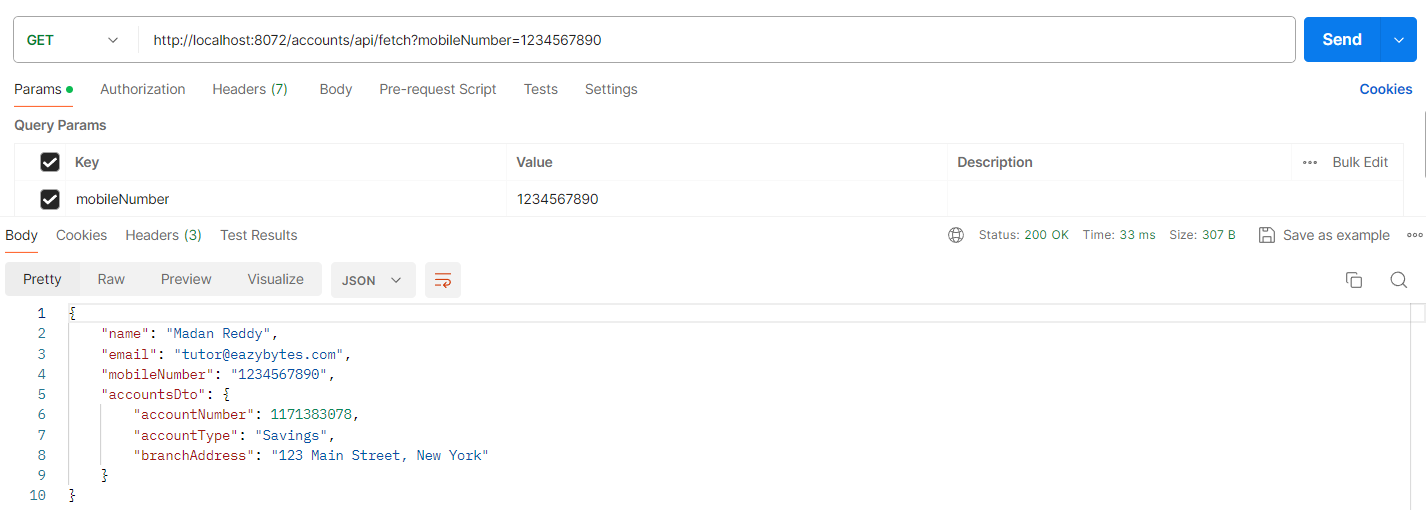
D:\Experiments\Microservices\sb-bank-application\gatewayserver\src\main\resources\application.properties



With this we are telling to our gateway server.

Please accept the service name with the completely lowercase.





Implementing Custom Routing using Spring Cloud Gateway

As of now, our gateway server can accept the external traffic and it is able to forward the

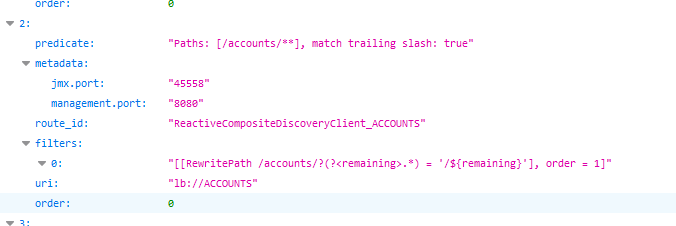
same request to the actual microservices.

And how it can do is, with the help of these routing details that it fetched from the Eureka server.

So, you can see here this is the default behaviour by default, if someone is going to send the loans

inside the request to path with completely small case letters, then what our filter and the configurations are present here will be triggered.

<http://localhost:8072/actuator/gateway/routes>



So, this is the default behaviour.

And with this default behaviour, this is how our clients are external applications

they must send the request to the gateway server.

So, this is fine Perfectly.

Sometimes this default behaviour may not be suitable for the projects.

For example, here I have some custom requirement, which is, I want my client applications to send the request to a path which is easybank/accounts.

So, this will look more professional in front of my client applications that since they are going to

invoke the easybank related microservices, this is going to be make sense.

Like inside easybank organization, we have a microservice with the name accounts and inside this accounts microservice we have this path.

So, the same way I want to follow for other microservice as well, like for all my microservices, I

want this prefix apart from this application name our microservice name.

So, whenever you have these kind of custom or dynamic routing related requirements, then in such scenarios, Spring Cloud Gateway also gives flexibility to developers to define their own routing configurations.

package com.eazybytes.gatewayserver;

import org.springframework.cloud.gateway.route.RouteLocator;

import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;

@SpringBootApplication

public class GatewayserverApplication {

    public static void main(String[] args) {

        SpringApplication.run(GatewayserverApplication.class, args);

    }

    @Bean

    public RouteLocator eazyBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder){

        return routeLocatorBuilder.routes()

                .route(p -> p.path("/eazybank/accounts/\*\*")

                .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.\*)", "/${segment}"))

                .uri("lb://ACCOUNTS"))

                .route(p -> p.path("/eazybank/loans/\*\*")

                .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.\*)", "/${segment}"))

                .uri("lb://LOANS"))

                .route(p -> p.path("/eazybank/cards/\*\*")

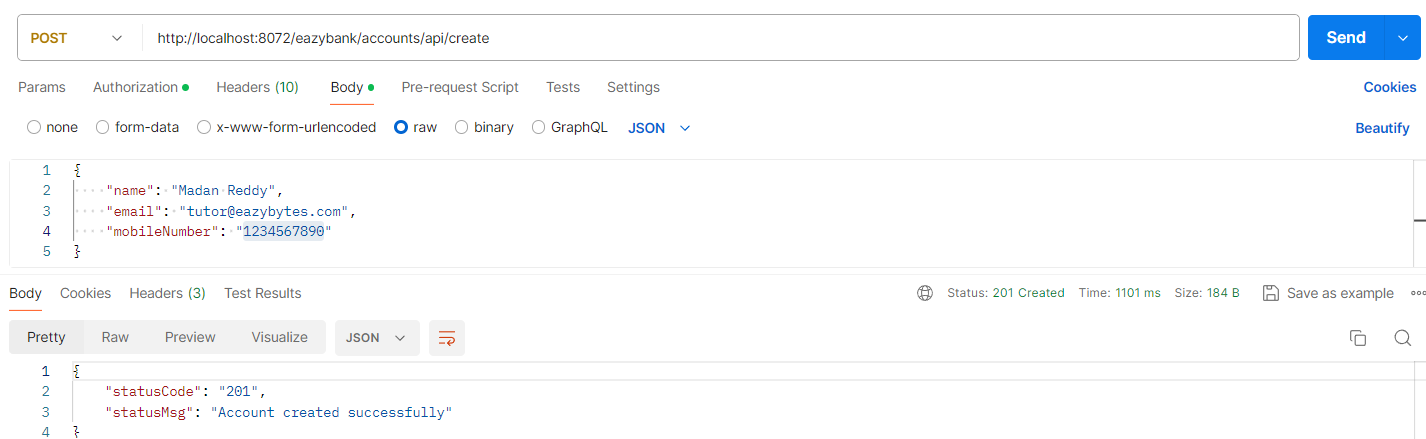
                .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.\*)", "/${segment}"))

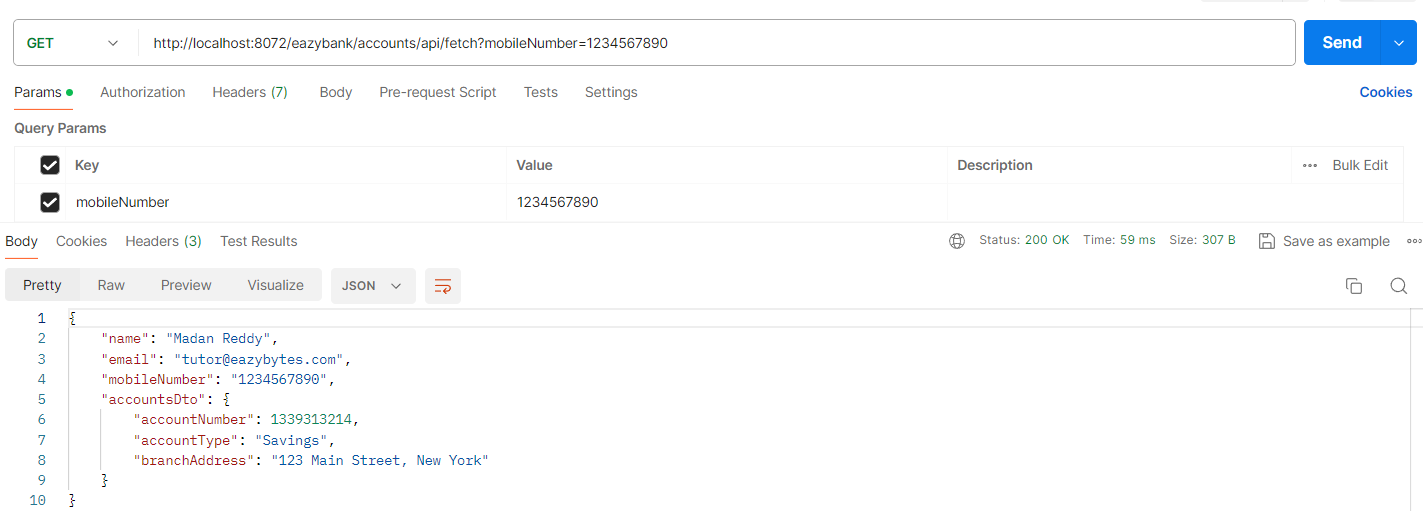
                .uri("lb://CARDS"))

                .build();

    }

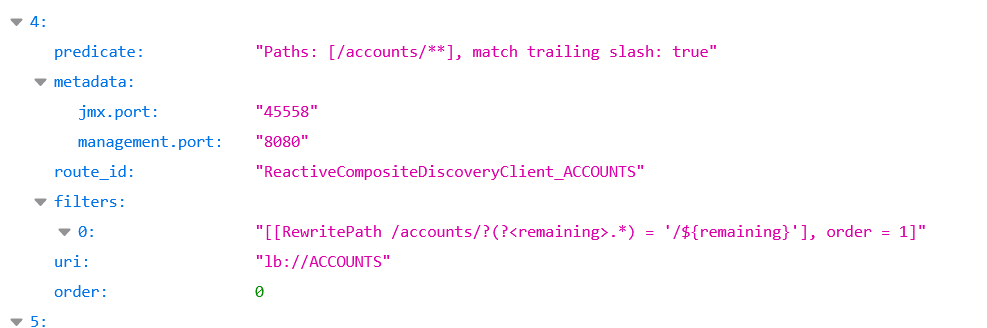
}

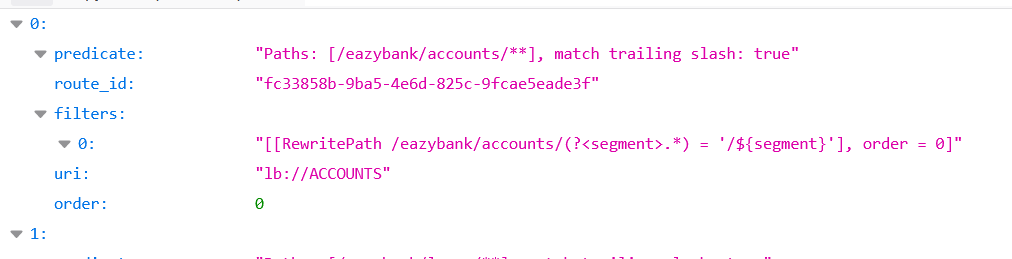




<http://localhost:8072/actuator/gateway/routes>

We have default and our custom routing configurations, Let us disable the default routing configuration’s.  
**DEFAULT**



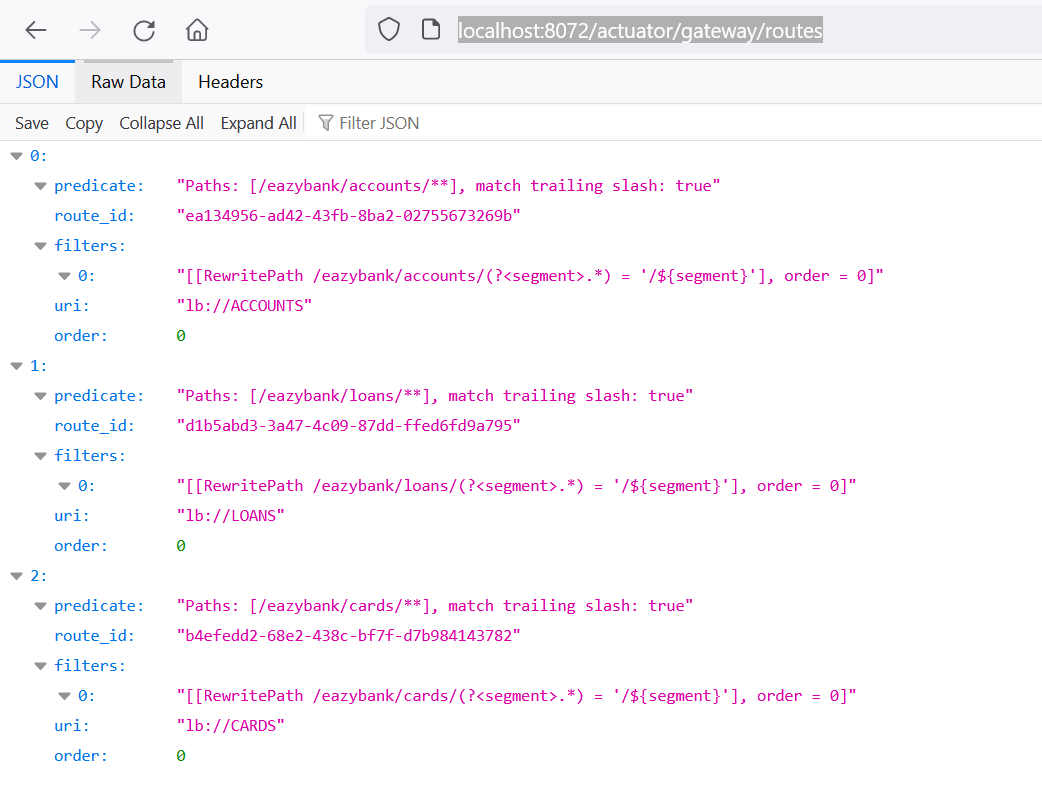
**CUSTOM **

To disable the default routing configuration disable below property.

D:\Experiments\Microservices\sb-bank-application\gatewayserver\src\main\resources\application.properties

spring.cloud.gateway.discovery.locator.enabled=false

<http://localhost:8072/actuator/gateway/routes>



Demo of addResponseHeader filter

package com.eazybytes.gatewayserver;

@SpringBootApplication

public class GatewayserverApplication {

    public static void main(String[] args) {

        SpringApplication.run(GatewayserverApplication.class, args);

    }

    @Bean

    public RouteLocator eazyBankRouteConfig(RouteLocatorBuilder routeLocatorBuilder){

        return routeLocatorBuilder.routes()

                .route(p -> p.path("/eazybank/accounts/\*\*")

                .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString()))

                .uri("lb://ACCOUNTS"))

                .route(p -> p.path("/eazybank/loans/\*\*")

                .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString()))

                .uri("lb://LOANS"))

                .route(p -> p.path("/eazybank/cards/\*\*")

                .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.\*)", "/${segment}")

                               .addResponseHeader("X-Response-Time", LocalDateTime.now().toString()))

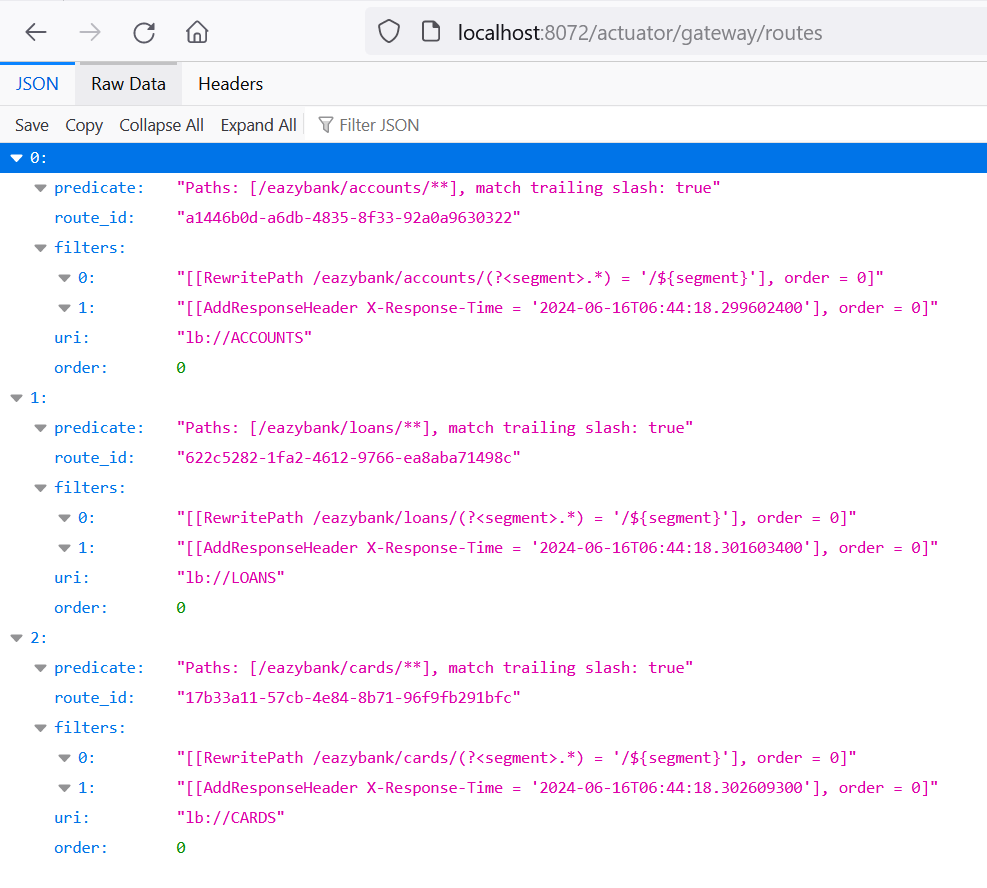
                .uri("lb://CARDS"))

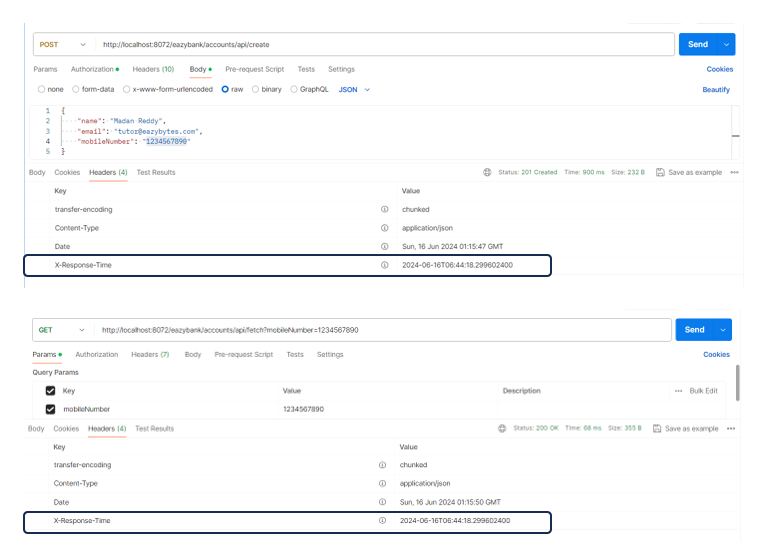
                .build();

    }

}

<http://localhost:8072/actuator/gateway/routes>





Implementing Cross Cutting concern Tracing and Logging using Gateway – Part 1

Now, inside this lecture, let us try to create our own custom filters inside the Gateway server.

Before we try to create these custom filters, let us try to decide on what business logic we want to

implement with the help of these custom filters.

I want to take a business scenario where as soon as my gateway server receives an external traffic or

external request, it must generate a correlation ID.

So here ID is a randomly generated value and the same correlation ID I want to send for all the further microservices that my request is going to travel.

Maybe if I am trying to send a request from Gateway server to accounts, from accounts to loans and cards, then I want to make sure the same trace ID are the correlation ID to be sent to all the respective microservices.

And using the same correlation ID and the trace ID, I am going to add some logger statements inside

my microservices.

While I am sending the response back to the client, I want to add the same correlation ID inside the

response header so that in future, whenever my client has some issues with a particular request saying that the data is not correct or there is an exception happened.

So, using that correlation ID my developer, he can look at the logs and identify up to which microservice the request has travelled and if there is an exception under which microservice the exception received.

So, this is what we are trying to achieve with the help of these custom filters.

This RequestTraceFilter is responsible to generate a trace ID or correlation ID whenever a new request came to my gateway server from the external client applications.

package com.eazybytes.gatewayserver.filters;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.cloud.gateway.filter.GatewayFilterChain;

import org.springframework.cloud.gateway.filter.GlobalFilter;

import org.springframework.core.annotation.Order;

import org.springframework.web.server.ServerWebExchange;

import reactor.core.publisher.Mono;

@Order(1)

@Component

public class RequestTraceFilter implements GlobalFilter {

    private static final Logger logger = LoggerFactory.getLogger(RequestTraceFilter.class);

    @Autowired

    FilterUtility filterUtility;

    @Override

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

        HttpHeaders requestHeaders = exchange.getRequest().getHeaders();

        if (isCorrelationIdPresent(requestHeaders)) {

            logger.debug("eazyBank-correlation-id found in RequestTraceFilter : {}",

                    filterUtility.getCorrelationId(requestHeaders));

        } else {

            String correlationID = generateCorrelationId();

            exchange = filterUtility.setCorrelationId(exchange, correlationID);

            logger.debug("eazyBank-correlation-id generated in RequestTraceFilter : {}", correlationID);

        }

        return chain.filter(exchange);

    }

    private boolean isCorrelationIdPresent(HttpHeaders requestHeaders) {

        if (filterUtility.getCorrelationId(requestHeaders) != null) {

            return true;

        } else {

            return false;

        }

    }

    private String generateCorrelationId() {

        return java.util.UUID.randomUUID().toString();

    }

}

The ResponseTraceFilter is it is going to add the trace ID or the correlation ID into the response

so that my client also is aware about that trace ID that is associated with a particular request.

package com.eazybytes.gatewayserver.filters;

@Configuration

public class ResponseTraceFilter {

    private static final Logger logger = LoggerFactory.getLogger(ResponseTraceFilter.class);

    @Autowired

    FilterUtility filterUtility;

    @Bean

    public GlobalFilter postGlobalFilter() {

        return (exchange, chain) -> {

            return chain.filter(exchange).then(Mono.fromRunnable(() -> {

                HttpHeaders requestHeaders = exchange.getRequest().getHeaders();

                String correlationId = filterUtility.getCorrelationId(requestHeaders);

                logger.debug("Updated the correlation id to the outbound headers: {}", correlationId);

                exchange.getResponse().getHeaders().add(filterUtility.CORRELATION\_ID, correlationId);

            }));

        };

    }

}

package com.eazybytes.gatewayserver.filters;

import org.springframework.web.server.ServerWebExchange;

@Component

public class FilterUtility {

    public static final String CORRELATION\_ID = "eazybank-correlation-id";

    public String getCorrelationId(HttpHeaders requestHeaders) {

        if (requestHeaders.get(CORRELATION\_ID) != null) {

            List<String> requestHeaderList = requestHeaders.get(CORRELATION\_ID);

            return requestHeaderList.stream().findFirst().get();

        } else {

            return null;

        }

    }

    public ServerWebExchange setRequestHeader(ServerWebExchange exchange, String name, String value) {

        return exchange.mutate().request(exchange.getRequest().mutate().header(name, value).build()).build();

    }

    public ServerWebExchange setCorrelationId(ServerWebExchange exchange, String correlationId) {

        return this.setRequestHeader(exchange, CORRELATION\_ID, correlationId);

    }

}

D:\Experiments\Microservices\sb-bank-application\gatewayserver\src\main\resources\application.properties

logging.level.com.eaztbytes.gatewayserver: DEBUG

Implementing Cross Cutting concern Tracing and Logging using Gateway – Part 2

Now inside this lecture, let us try to make changes inside the individual microservices to accept the

request header that my gateway is going to forward along with the request.

We do not have to make the changes in all the APIs available inside the microservices.

Instead, we can take a single API that is going to travel all the microservices.

We have such REST API inside the CustomerController.

***Modifications inside the accounts microservice***

package com.eazybytes.accounts.controller;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.validation.annotation.Validated;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.bind.annotation.RestController;

import com.eazybytes.accounts.dto.CustomerDetailsDto;

import com.eazybytes.accounts.service.ICustomerService;

import jakarta.validation.constraints.Pattern;

@RestController

@RequestMapping(path = "/api", produces = "application/json")

@Validated

public class CustomerController {

    private static final Logger logger = LoggerFactory.getLogger(CustomerController.class);

    private ICustomerService iCustomerService;

    public CustomerController(ICustomerService iCustomerService){

        this.iCustomerService = iCustomerService;

    }

    @GetMapping("/fetchCustomerDetails")

    public ResponseEntity<CustomerDetailsDto> fetchCustomerDetails(@RequestHeader("eazybank-correlation-id")

                                                                   String correlationId,

                                                                   @RequestParam

                                                                   @Pattern(regexp = "($|[0-9]{10})", message = "Mobile number should be 10 digits value")

                                                                   String mobileNumber) {

        logger.debug("eazyBank-correlation-id found: {} ", correlationId);

        CustomerDetailsDto customerDetailsDto = iCustomerService.fetchCustomerDetails(mobileNumber, correlationId);

        return ResponseEntity.status(HttpStatus.OK).body(customerDetailsDto);

    }

}

package com.eazybytes.accounts.service;

import com.eazybytes.accounts.dto.CustomerDetailsDto;

public interface ICustomerService {

    //CustomerDetailsDto fetchCustomerDetails(String mobileNumber);

    CustomerDetailsDto fetchCustomerDetails(String mobileNumber, String correlationId);

}

package com.eazybytes.accounts.service.impl;

import org.springframework.http.ResponseEntity;

import org.springframework.stereotype.Service;

import com.eazybytes.accounts.dto.AccountsDto;

import com.eazybytes.accounts.dto.CardsDto;

import com.eazybytes.accounts.dto.CustomerDetailsDto;

import com.eazybytes.accounts.dto.LoansDto;

import com.eazybytes.accounts.entity.Accounts;

import com.eazybytes.accounts.entity.Customer;

import com.eazybytes.accounts.exception.ResourceNotFoundException;

import com.eazybytes.accounts.mapper.AccountsMapper;

import com.eazybytes.accounts.mapper.CustomerMapper;

import com.eazybytes.accounts.repository.AccountRepository;

import com.eazybytes.accounts.repository.CustomerRepository;

import com.eazybytes.accounts.service.ICustomerService;

import com.eazybytes.accounts.service.client.CardsFeignClient;

import com.eazybytes.accounts.service.client.LoansFeignClient;

import lombok.AllArgsConstructor;

@Service

@AllArgsConstructor

public class CustomerServiceImpl implements ICustomerService {

    private AccountRepository accountRepository;

    private CustomerRepository customerRepository;

    private LoansFeignClient loansFeignClient;

    private CardsFeignClient cardsFeignClient;

    @Override

    //public CustomerDetailsDto fetchCustomerDetails(String mobileNumber) {

    public CustomerDetailsDto fetchCustomerDetails(String mobileNumber, String correlationId) {

         Customer customer = customerRepository.findByMobileNumber(mobileNumber).orElseThrow(

            () -> new ResourceNotFoundException("Customer", "mobileNumber", mobileNumber)

        );

        Accounts accounts = accountRepository.findByCustomerId(customer.getCustomerId()).orElseThrow(

            () -> new ResourceNotFoundException("Account", "customerId", customer.getCustomerId().toString())

        );

        CustomerDetailsDto customerDetailsDto = CustomerMapper.mapToCustomerDetailsDto(customer, new CustomerDetailsDto());

        customerDetailsDto.setAccountsDto(AccountsMapper.mapToAccountsDto(accounts, new AccountsDto()));

        // ResponseEntity<LoansDto> loansDtoResponseEntity = loansFeignClient.fetchLoansDetails(mobileNumber);

        // customerDetailsDto.setLoansDto(loansDtoResponseEntity.getBody());

        // ResponseEntity<CardsDto> cardsDtoResponseEntity = cardsFeignClient.fetchCardDetails(mobileNumber);

        // customerDetailsDto.setCardsDto(cardsDtoResponseEntity.getBody());

        ResponseEntity<LoansDto> loansDtoResponseEntity = loansFeignClient.fetchLoanDetails(correlationId, mobileNumber);

        customerDetailsDto.setLoansDto(loansDtoResponseEntity.getBody());

        ResponseEntity<CardsDto> cardsDtoResponseEntity = cardsFeignClient.fetchCardDetails(correlationId, mobileNumber);

        customerDetailsDto.setCardsDto(cardsDtoResponseEntity.getBody());

        return customerDetailsDto;

    }

}

package com.eazybytes.accounts.service.client;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.http.MediaType;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestParam;

import com.eazybytes.accounts.dto.CardsDto;

@FeignClient("cards")

public interface CardsFeignClient {

    @GetMapping(value = "/api/fetch", consumes = MediaType.APPLICATION\_JSON\_VALUE)

    // public ResponseEntity<CardsDto> fetchCardDetails(@RequestParam String mobileNumber);

    public ResponseEntity<CardsDto> fetchCardDetails(@RequestHeader("eazybank-correlation-id") String correlationId, @RequestParam String mobileNumber);

}

package com.eazybytes.accounts.service.client;

import org.springframework.cloud.openfeign.FeignClient;

import org.springframework.http.MediaType;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.GetMapping;

import org.springframework.web.bind.annotation.RequestHeader;

import org.springframework.web.bind.annotation.RequestParam;

import com.eazybytes.accounts.dto.LoansDto;

@FeignClient("loans")

public interface LoansFeignClient {

    @GetMapping(value = "/api/fetch", consumes = MediaType.APPLICATION\_JSON\_VALUE)

    // public ResponseEntity<LoansDto> fetchLoansDetails(@RequestParam String mobileNumber);

     public ResponseEntity<LoansDto> fetchLoanDetails(@RequestHeader("eazybank-correlation-id") String correlationId, @RequestParam String mobileNumber);

}

D:\Experiments\Microservices\sb-bank-application\accounts\src\main\resources\application.properties

logging.level.com.eaztbytes.accounts=DEBUG

***Modifications inside the loans microservice***

package com.eazybytes.loans.controller;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})

@Validated

public class LoansController {

    private static final Logger logger = LoggerFactory.getLogger(LoansController.class);

    ….

    @GetMapping("/fetch")

    public ResponseEntity<LoansDto> fetchLoanDetails(@RequestHeader("eazybank-correlation-id") String correlationId,

                                                    @RequestParam

                                                    @Pattern(regexp="(^$|[0-9]{10})",message = "Mobile number must be 10 digits")

                                                    String mobileNumber) {

        logger.debug("eazyBank-correlation-id found: {} ", correlationId);

        LoansDto loansDto = iLoansService.fetchLoan(mobileNumber);

        return ResponseEntity.status(HttpStatus.OK).body(loansDto);

    }

    …

}

D:\Experiments\Microservices\sb-bank-application\loans\src\main\resources\application.properties  
  
logging.level.com.eaztbytes.loans=DEBUG

***Modifications inside the cards microservice***

package com.eazybytes.cards.controller;

@RestController

@RequestMapping(path = "/api", produces = {MediaType.APPLICATION\_JSON\_VALUE})

@Validated

public class CardsController {

    private static final Logger logger = LoggerFactory.getLogger(CardsController.class);

    …

    @GetMapping("/fetch")

    public ResponseEntity<CardsDto> fetchCardDetails(@RequestHeader("eazybank-correlation-id") String correlationId,

                                                    @RequestParam

                                                    @Pattern(regexp="(^$|[0-9]{10})",message = "Mobile number must be 10 digits")

                                                    String mobileNumber) {

        logger.debug("eazyBank-correlation-id found: {} ", correlationId);

        CardsDto cardsDto = iCardsService.fetchCard(mobileNumber);

        return ResponseEntity.status(HttpStatus.OK).body(cardsDto);

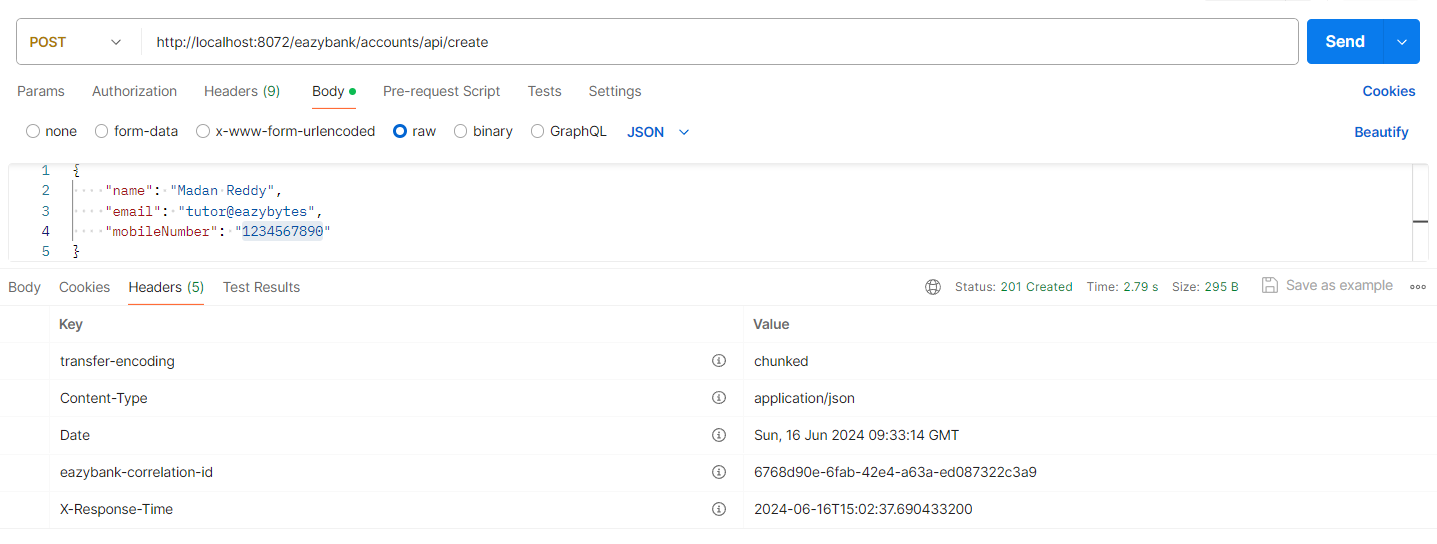
    }

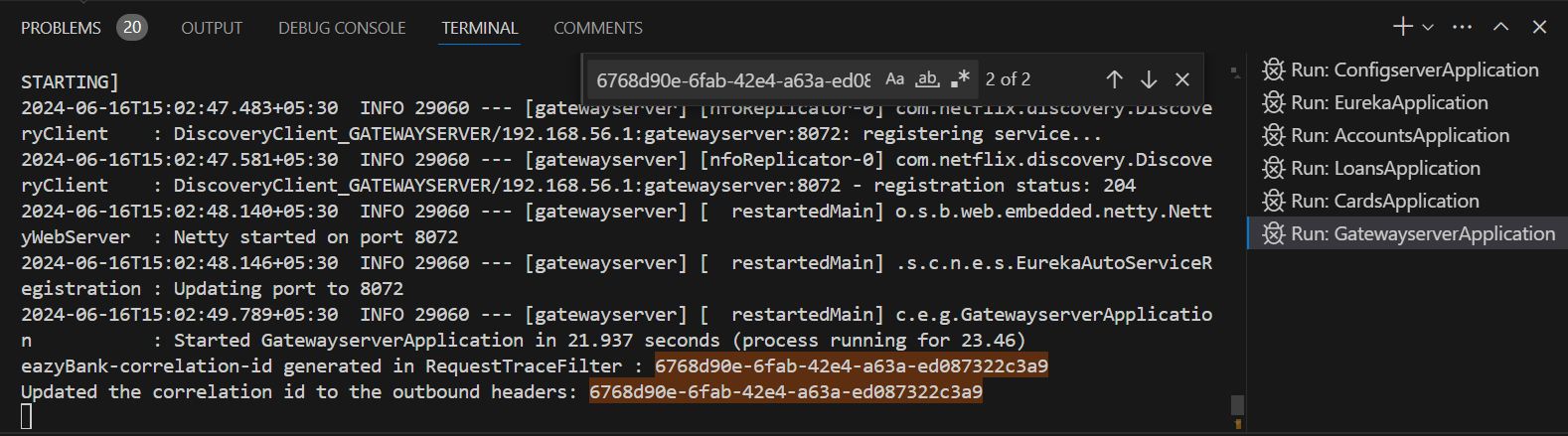
   …

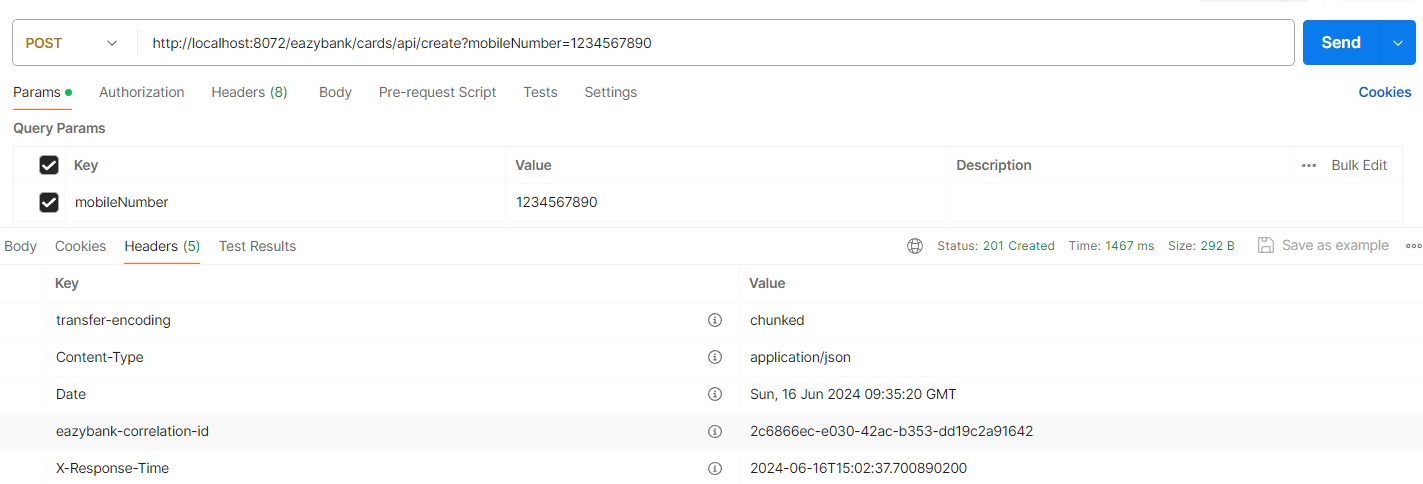
}

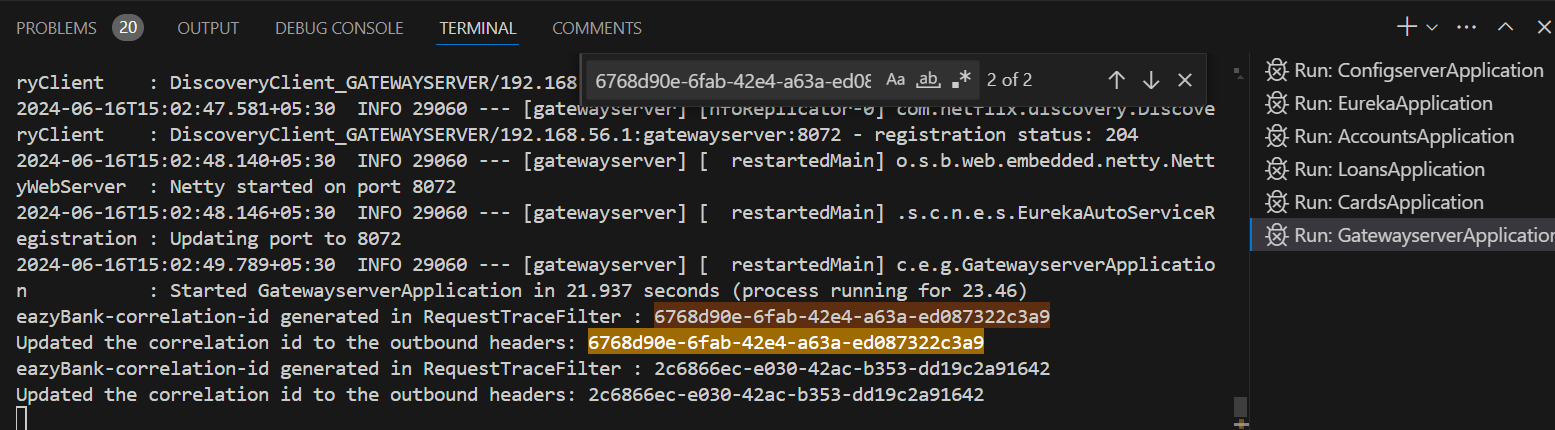
D:\Experiments\Microservices\sb-bank-application\cards\src\main\resources\application.properties

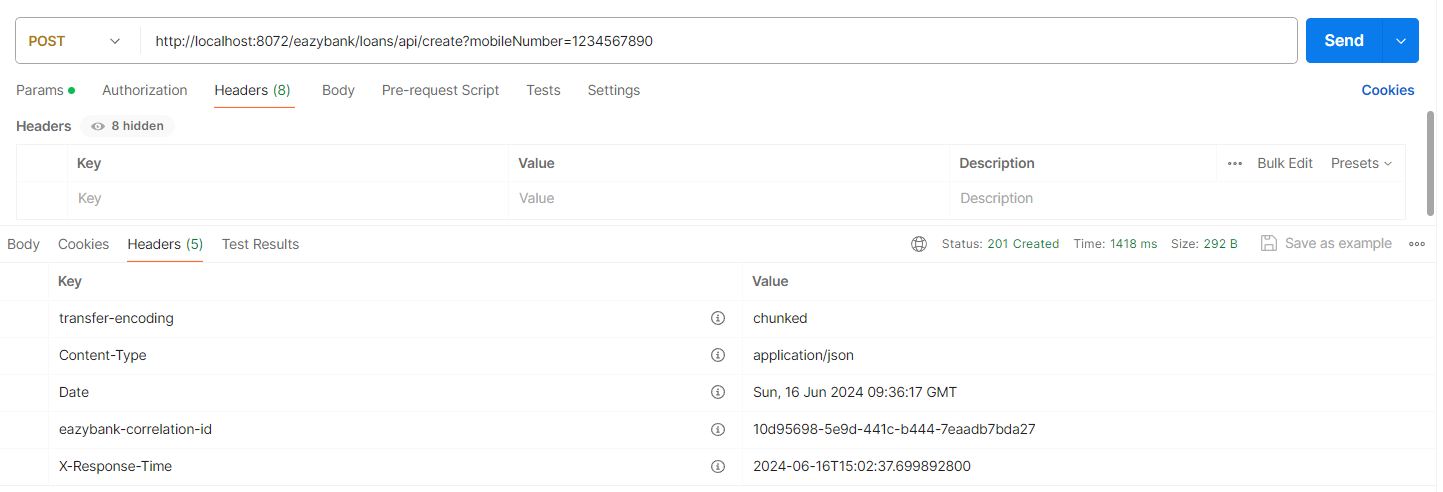
logging.level.com.eaztbytes.cards=DEBUG

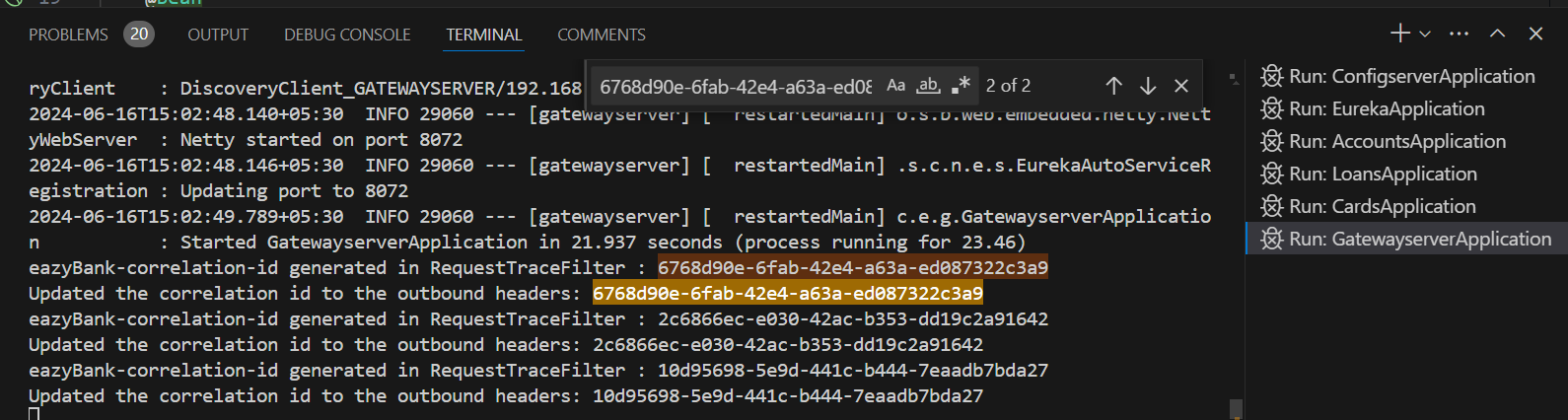


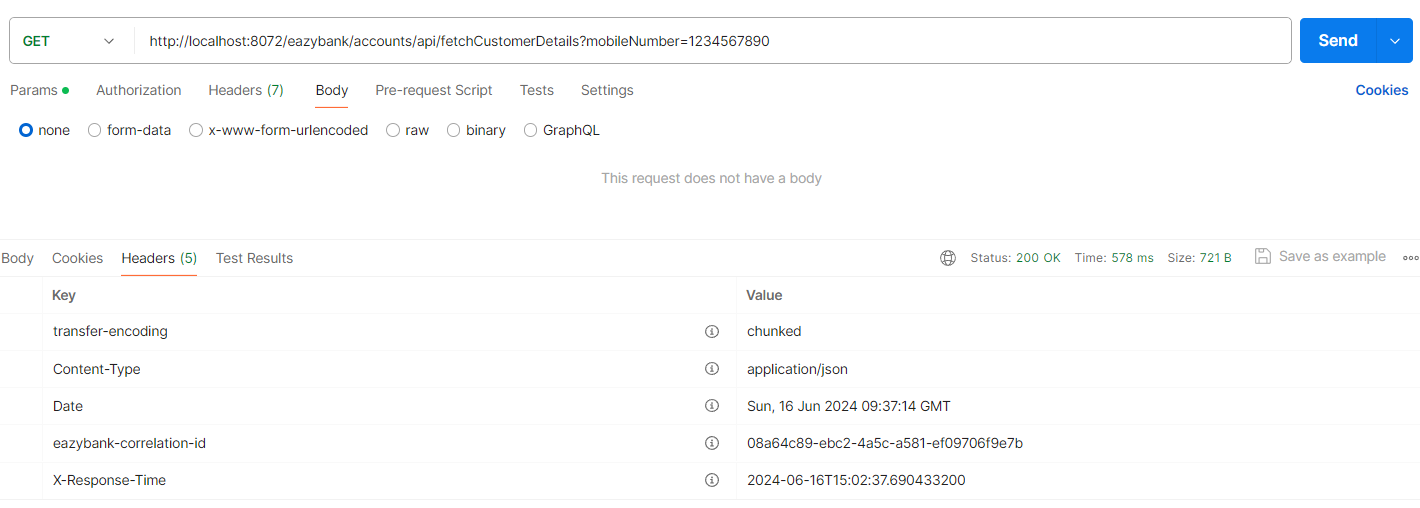


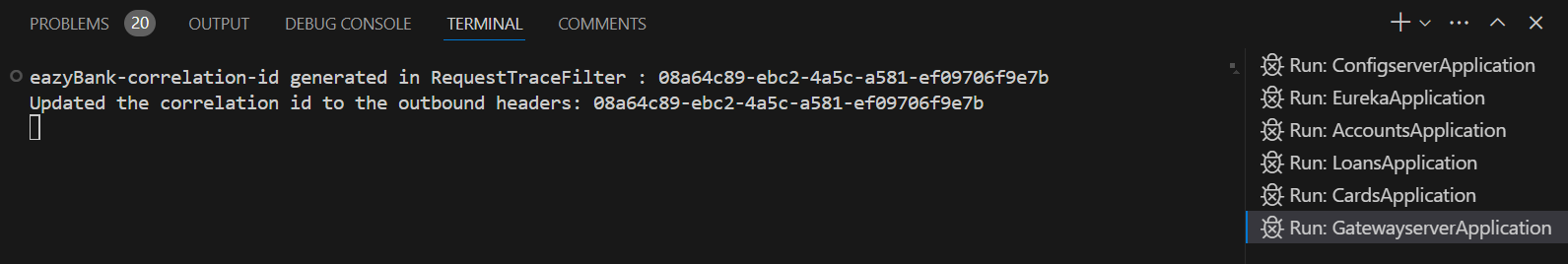


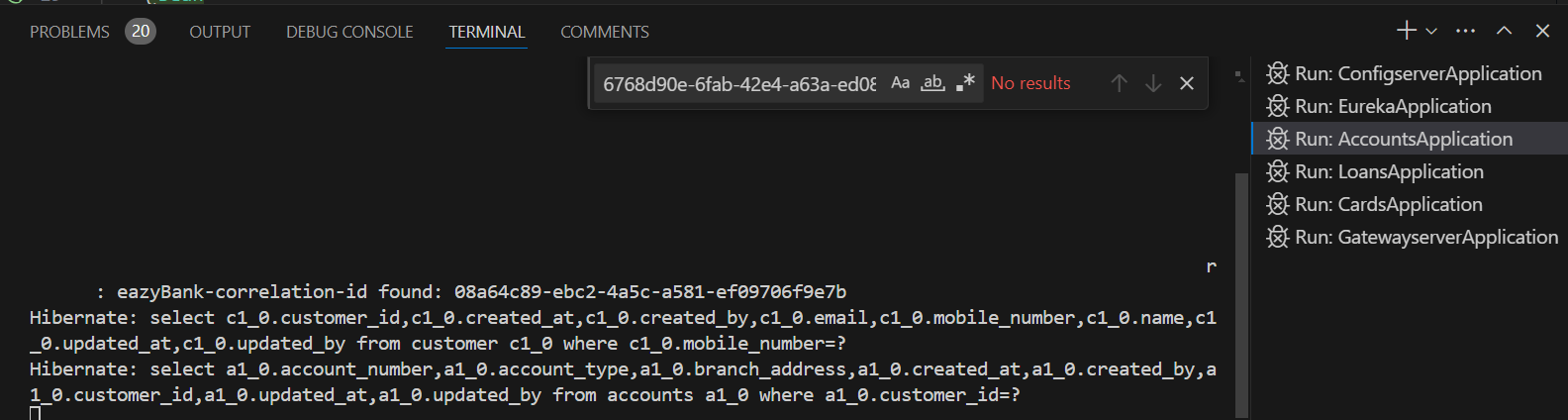


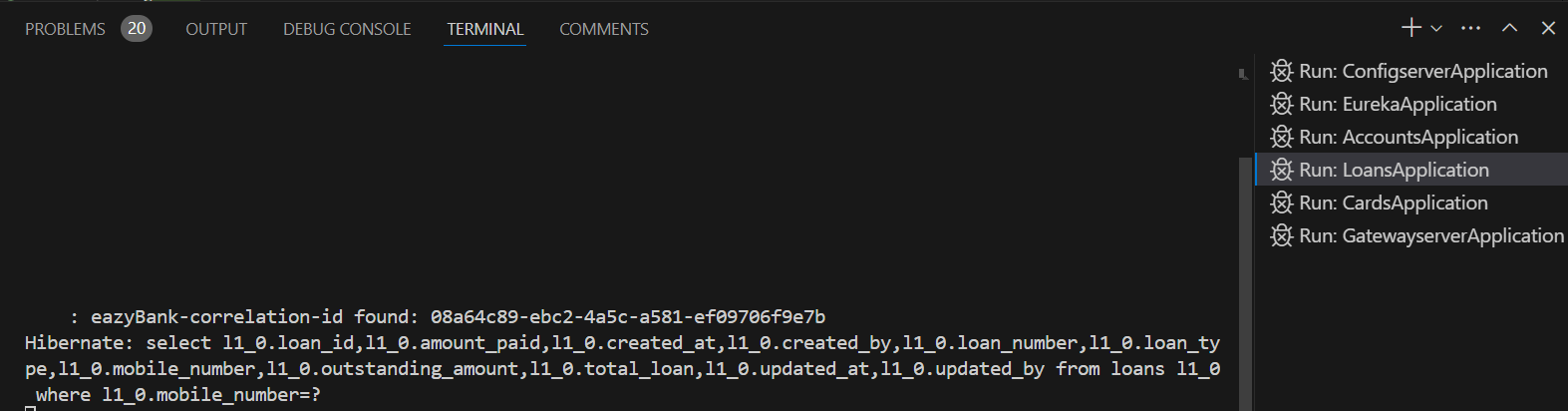


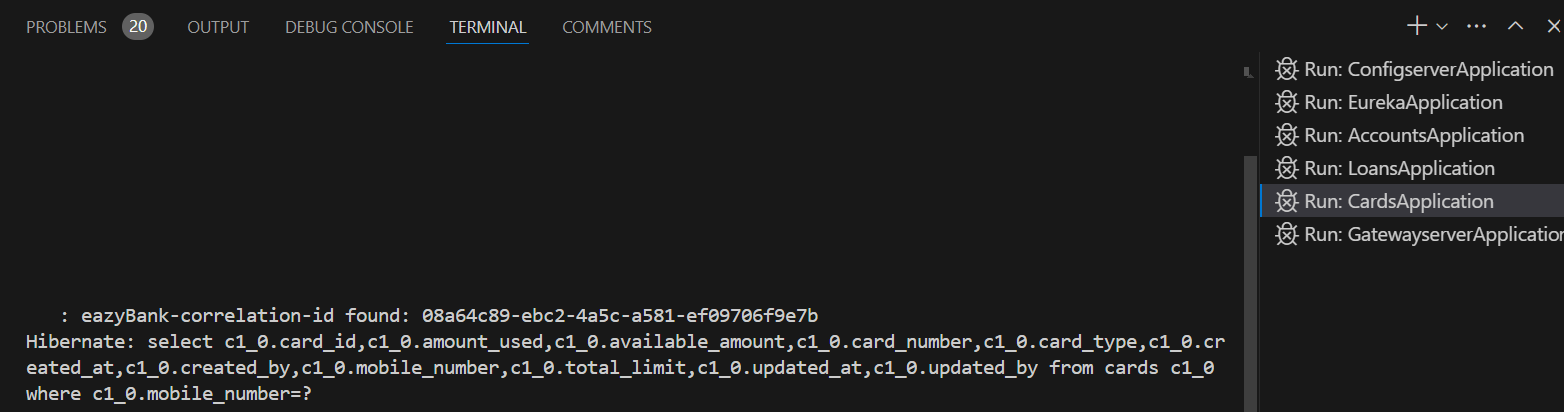












Generating and Pushing Docker images with Spring Cloud Gateway changes

Updating Docker Compose file to adapt Spring Cloud Gateway changes