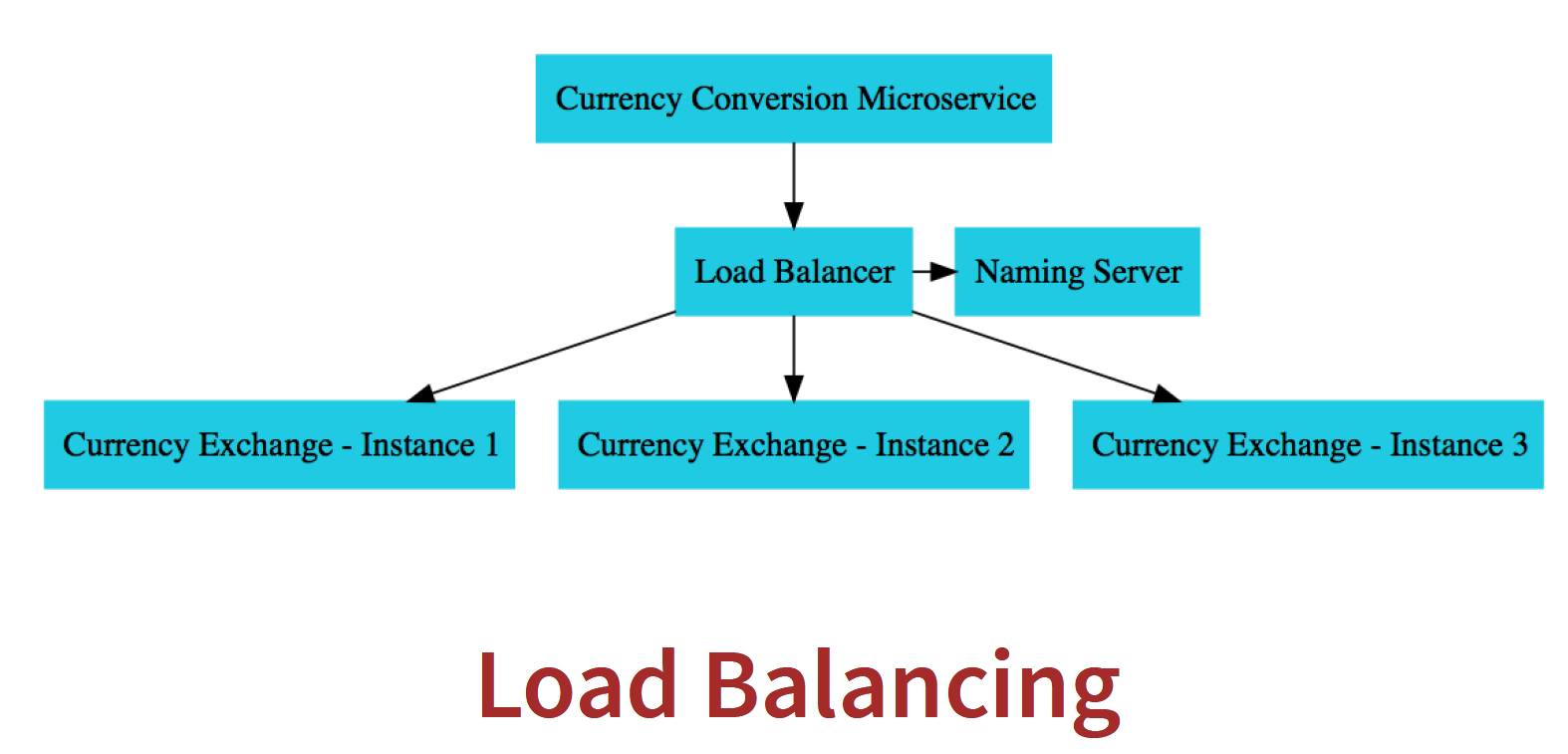
***Step 21 QuickStart by importing Microservices***

***Revision of previous steps***

***Step 22 Load Balancing with Eureka, Feign and Spring Cloud Load Balance***



How load balancing happens between the multiple instances of Currency Exchange from Currency Conversion.

We would want the feign client to talk to Eureka and pick up the instances of currency-exchange and do load balancing between them.

**How many instances of currency-exchange are up and running right now?**

There are two instances of currency-exchange up and running, 8000 and 8001.

And if we go and execute the feign URL again.

Check the port sometimes it will come as 8000, 8001,8000,80001

**What is happening in here is, inside the Currency Conversion Microservice?**

There is a load balancer component which is talking to the naming server, finding the instances, and doing automatic load balancing between them and this is called client-side load balancing, and this is happening through feign.

**How does feign do load balancing?**

We go into our currency-conversion-service pom.xml,

If we look at the dependency hierarchy which is present in here, and just type in **load**.

We would see that there is a load balancer, **spring-cloud-starter-loadbalancer**,

which is brought into the class path by **spring-cloud-starter-netflix-eureka-client**

and this is the load balancer framework that is used by feign. it distribute the load

among the multiple instances which are returned by Eureka.

In the earlier versions of Spring Cloud, the load balancer which was used was **Ribbon**.

In the recent versions, Spring Cloud shifted to using **Spring Cloud Load Balancer** as the load balancer.

The great thing is, if we are using **Eureka and Feign, then load balancing comes for free**.

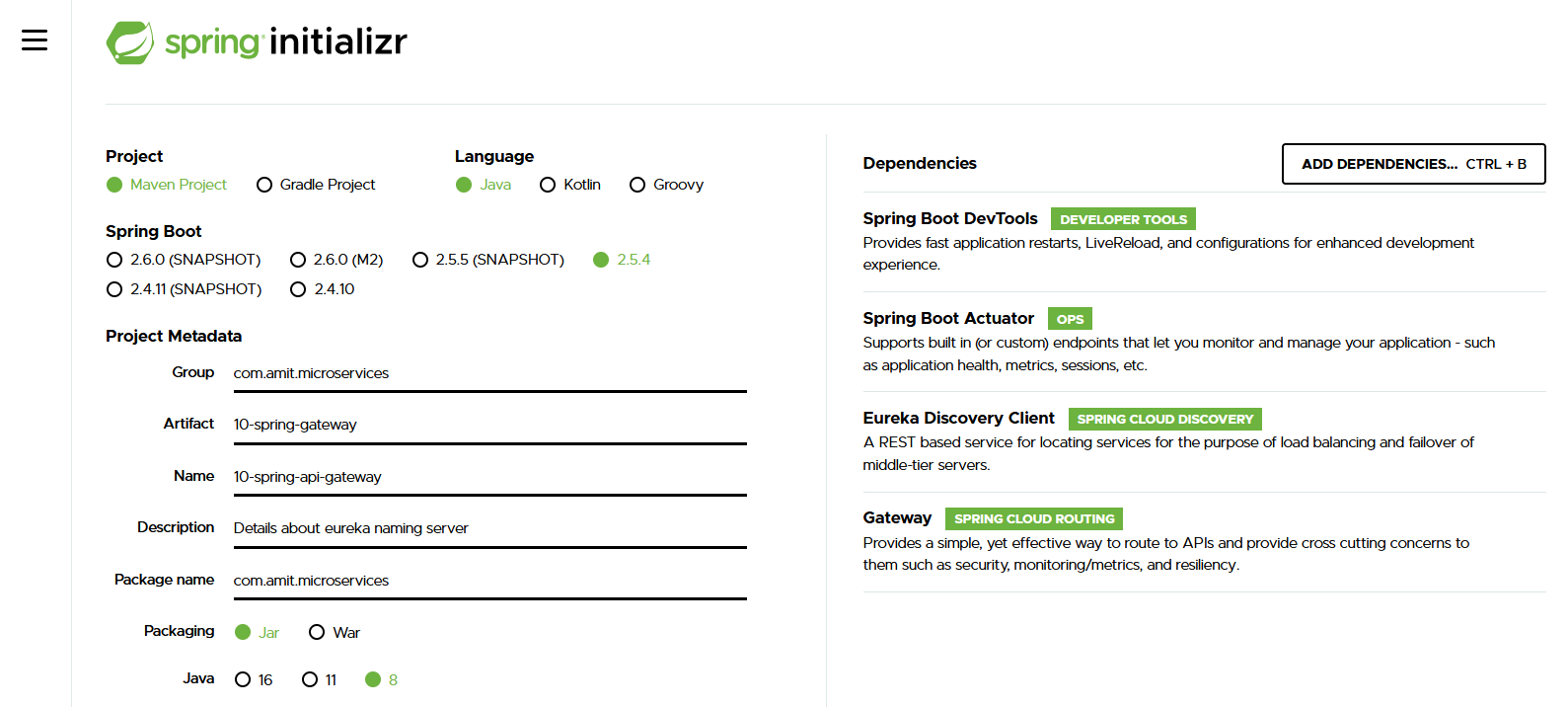
Graphical user interface, text, application

Description automatically generated

Application

Description automatically generated with low confidence

***Setting up Spring cloud API Gateway***



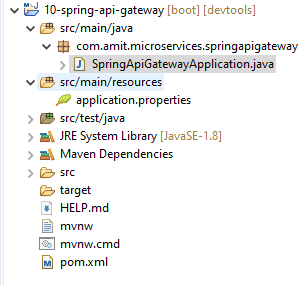
There are a-lots of microservices are there and these micro services have a lot of common features authentication, authorization, logging etc

**Where do you implement all these common features?**

That's one of the typical questions in a micro series architecture.

And the typical solution is to go for an API gateway in the older versions of Spring Cloud, the popular API gateway to use was Zul.

API gateway **spring cloud gateway**.



**application.properties**

**spring.application.name=api-gateway**

**server.port=8765**

**eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka**

<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>

**Because of netflix-eureka-client API gateway get automatically register with the eureka.**

Graphical user interface, text, application

Description automatically generated

**URLs for next Lecture**

Some of these URLs may be complex to write by hand:

Refer to here if you have problems in the next steps.

Initial

- http://localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/USD/to/INR

- http://localhost:8765/CURRENCY-CONVERSION/currency-conversion/from/USD/to/INR/quantity/10

- http://localhost:8765/CURRENCY-CONVERSION/currency-conversion-feign/from/USD/to/INR/quantity/10

Lower Case

- http://localhost:8765/currency-exchange/currency-exchange/from/USD/to/INR

- http://localhost:8765/currency-conversion/currency-conversion/from/USD/to/INR/quantity/10

- http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/INR/quantity/10

Custom Routes

- http://localhost:8765/currency-exchange/from/USD/to/INR

- http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10

- http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10

- http://localhost:8765/currency-conversion-new/from/USD/to/INR/quantity/10

***Step 23 -Enabling Discovery Locator with Eureka for Spring Cloud Gateway***

I want to want to talk to the Currency Exchange Service.

Currency Exchange Service is registered with the name CURRENCY-EXCHANGE in eureka server.

We are picking up the name for Currency Exchange and should be up and running, cross verify it.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

URL: <http://localhost:8000/currency-exchange/from/EUR/to/INR/>

Above URL should be working properly.

Graphical user interface, text, application, email

Description automatically generated

Using API gateway

Syntax

[http://localhost:8000/<Application name in the Eureka server>>/currency-exchange/from/EUR/to/INR/](http://localhost:8000/%3cApplication%20name%20in%20the%20Eureka%20server%3e%3e/currency-exchange/from/EUR/to/INR/)

[http://localhost:8765/**CURRENCY-EXCHANGE**/currency-exchange/from/EUR/to/INR/](http://localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/EUR/to/INR/)

Graphical user interface, text, application

Description automatically generated

Somehow gateway feature is not enabled.

To enable it need to add some of the property in **application.property** file

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

By default, this is disabled.

**What we want to do**?

We would want to enable the service discovery by using discovery client. This is possible

because in our pom.xml, we already have Netflix Eureka client and and the API gateway is already registered with Eureka and it can discover other services also from Eureka.

Graphical user interface, text, application, email

Description automatically generated

**localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/USD/to/INR.**

It's working as is without a problem

We can implement all the common features in our API gateway and the API Gateway would

take care of the common features and then invoke the currency exchange.

Similarly, we can do for **Currency-conversion**

**Similarly, for Currency-conversion**

<http://localhost:8765/CURRENCY-CONVERSION/currency-conversion-feign/from/USD/to/INR/quantity/10>

Graphical user interface, application

Description automatically generated with medium confidence

<http://localhost:8765/CURRENCY-CONVERSION/currency-conversion/from/from/USD/to/INR/quantity/10>

Graphical user interface, text, application, email

Description automatically generated

We can call any microservice which is registered with Eureka through the API Gateway.

If we would want to implement things like authentication, we can implement them on API Gateway and we can only allow those things which are authenticated in our microservices.

So, all the authentication logic can be implemented on the API gateway.

[http://localhost:8765/**CURRENCY-EXCHANGE**/currency-exchange/from/EUR/to/INR/](http://localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/EUR/to/INR/)

[**http://localhost:8765/CURRENCY-CONVERSION/currency-conversion-feign/from/USD/to/INR/quantity/10**](http://localhost:8765/CURRENCY-CONVERSION/currency-conversion-feign/from/USD/to/INR/quantity/10)

[**http://localhost:8765/CURRENCY-CONVERSION/currency-conversion/from/from/USD/to/INR/quantity/10**](http://localhost:8765/CURRENCY-CONVERSION/currency-conversion/from/from/USD/to/INR/quantity/10)

**CURRENCY-EXCHANGE, CURRENCY-CONVERSION** in **caps.** So, the above URL is not looking good.

To write in lower case need to add one more property in the **application.property** file.

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

<http://localhost:8765/currency-exchange/currency-exchange/from/EUR/to/INR/>

[**http://localhost:8765/currency-conversion/currency-conversion/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion/currency-conversion/from/USD/to/INR/quantity/10)

[**http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/INR/quantity/10)

**Still above URL having some repetition.**

**Currency-exchange and currency-conversion used twice. So still URL is not looks good.**

A very quick way to enable Spring Cloud API Gateway to route HTTP requests to a Microservice registered with discovery service is to simply enable the Discovery Locator.

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

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

Debugging Problems with Spring Cloud Gateway - V2

**Debugging** microservices problems can be **difficult** as there are multiple components involved.

**Step by Step instructions** is provided in the **troubleshooting guide** to help you troubleshoot frequently occurring problems.

Using the **Chrome Browser** is recommended.

**COMPLETE DEBUGGING GUIDE**

<https://github.com/in28minutes/spring-microservices-v2/blob/main/03.microservices/01-step-by-step-changes/microservices-v2-1.md#spring-cloud-api-gateway---step-22-to-step-25>

**TOP Recommendation from Debugging Guide:**

(6) Some student reported success when using lower-case-service-id instead of spring.cloud.gateway.discovery.locator.lowerCaseServiceId. See if it helps!

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

***Step 24 -Exploring Routes with Spring Cloud Gateway***

We learned how to build routes through the Eureka Naming Server and redirect all the requests.

Now we can learn **how to build custom routes** through the API Gateway?

We can build it through using the configuration file.

Create a RouteLocator bean by creating a method which would return a gateway router and to be able to create a route locator, we would need a RouteLocatorBuilder.

A picture containing timeline

Description automatically generated

**package** com.springcloud.gateway.routes.configuration;

**import** java.util.function.Function;

**import** org.springframework.cloud.gateway.route.Route;

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

**import** org.springframework.cloud.gateway.route.builder.Buildable;

**import** org.springframework.cloud.gateway.route.builder.PredicateSpec;

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

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

@Configuration

**public** **class** ApiGateWayCOnfiguration {

@Bean

**public** RouteLocator gatewayRouter(RouteLocatorBuilder builder) {

Function<PredicateSpec, Buildable<Route>> routeFunction

= p->p.path("/get")

.filters(f->f.addRequestHeader("MyHeader", "MyURI")

.addRequestParameter("Param", "MyValue"))

.uri("http://httpbin.org:80");

**return** builder.routes()

.route(routeFunction)

.build();

}

}

**We can write inline function as well as below.**

@Bean

**public** RouteLocator gatewayRouter(RouteLocatorBuilder builder) {

**return** builder.routes()

.route(p->p.path("/get")

.filters(f->f.addRequestHeader("MyHeader", "MyURI")

.addRequestParameter("Param", "MyValue"))

.uri("http://httpbin.org:80"))

.build();

}

[**https://www.javainuse.com/spring/cloud-gateway**](https://www.javainuse.com/spring/cloud-gateway)

[**https://cloud.spring.io/spring-cloud-gateway/reference/html/**](https://cloud.spring.io/spring-cloud-gateway/reference/html/)

Spring Cloud Gateway aims to provide a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as: security, monitoring/metrics, and resiliency.

[**http://localhost:8765/currency-exchange/currency-exchange/from/EUR/to/INR/**](http://localhost:8765/currency-exchange/currency-exchange/from/EUR/to/INR/)

[**http://localhost:8765/currency-conversion/currency-conversion/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion/currency-conversion/from/USD/to/INR/quantity/10)

[**http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion/currency-conversion-feign/from/USD/to/INR/quantity/10)

Above URL does not looks good and this URL looks a little complex.

currency-exchange and currency-conversion are repeating twice.

**How can we correct it?**

We can do this by configuring a custom route.

.route(p -> p.path("/currency-exchange/\*\*")

.uri("**lb://currency-exchange**"))

"/currency-exchange/\*\*"

If the request comes to this above URL, anything followed is by putting a regular expression in here.

If a request URL starts with **currency-exchange**, this will redirect to using the naming server, we can also do load balancing.

**How can I do that?**

The way you can do that is by just putting in **lb://** and the name of the registration on the Eureka server.

So, on the Eureka server, this is registered as currency-exchange.

@Bean

**public** RouteLocator gatewayRouter(RouteLocatorBuilder builder) {

**return** builder.routes()

.route(p -> p.path("/get")

.filters(f -> f.addRequestHeader("MyHeader", "MyURI")

.addRequestParameter("Param", "MyValue"))

.uri("http://httpbin.org:80"))

.route(p -> p.path("/currency-exchange/\*\*")

.uri("lb://currency-exchange"))

.route(p -> p.path("/currency-conversion/\*\*")

.uri("lb://currency-conversion"))

.route(p -> p.path("/currency-conversion-feign/\*\*")

.uri("lb://currency-conversion"))

.route(p-> p.path("/currency-conversion-new/\*\*")

.filters(f -> f.rewritePath("/currency-conversion-new/(?<segment>.\*)",

"/currency-conversion-feign/${segment}"))

.uri("lb://currency-conversion"))

.build();

}

After making above changes below ULR started working.

[**http://localhost:8765/currency-exchange/from/EUR/to/INR/**](http://localhost:8765/currency-exchange/from/EUR/to/INR/)

[**http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion/from/USD/to/INR/quantity/10)

[**http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10)

[**http://localhost:8765/currency-conversion-new/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion-new/from/USD/to/INR/quantity/10)

In addition to supporting existing URLs, we can even create new URLs and rewrite them.

.filters(f -> f.rewritePath("/currency-conversion-new/(?<segment>.\*)",

"/currency-conversion-feign/${segment}"))

We can access

[**http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/10)

by using below URL

[**http://localhost:8765/currency-conversion-new/from/USD/to/INR/quantity/10**](http://localhost:8765/currency-conversion-new/from/USD/to/INR/quantity/10)

***Step 25 -Implementing Spring Cloud Gateway Logging Filer***

[**https://www.javainuse.com/spring/cloud-gateway**](https://www.javainuse.com/spring/cloud-gateway)

[**https://cloud.spring.io/spring-cloud-gateway/reference/html/**](https://cloud.spring.io/spring-cloud-gateway/reference/html/)

Graphical user interface, text, application

Description automatically generated

@Component

**public** **class** LoggingFilter **implements** GlobalFilter {

**private** Logger logger = LoggerFactory.*getLogger*(LoggingFilter.**class**);

@Override

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

logger.info("Path of the request received -> {}",

exchange.getRequest().getPath());

**return** chain.filter(exchange);

}

}

Start the application.

For each request above message will gets printed on the console.

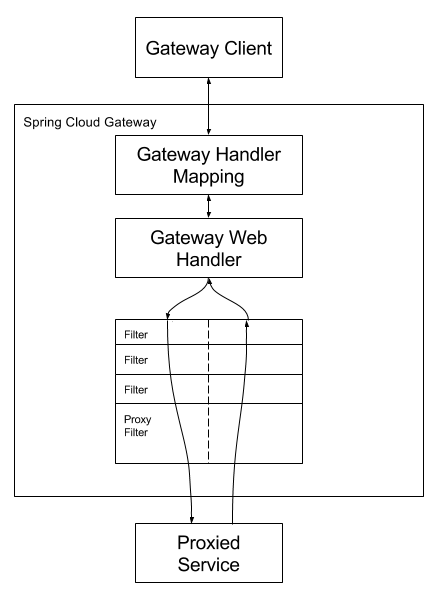


if we want to implement things like authentication for all the requests, then this might be a right place to implement that as well.

<https://www.javainuse.com/spring/cloud-filter>

<https://cloud.spring.io/spring-cloud-gateway/reference/html/>

The following diagram provides a high-level overview of how Spring Cloud Gateway works:



Clients make requests to Spring Cloud Gateway. If the Gateway Handler Mapping determines that a request matches a route, it is sent to the Gateway Web Handler.

This handler runs the request through a filter chain that is specific to the request.

The reason the filters are divided by the dotted line is that filters can run logic both before and after the proxy request is sent.

All “pre” filter logic is executed. Then the proxy request is made. After the proxy request is made, the “post” filter logic is run.

Graphical user interface, application

Description automatically generated

***Step 26 -Getting started with Circuit Breaker – Resilience4j filter***



In a microservices architecture, there is complex call chain. As shown in the above example here, a microservice can call another microservice, that microservice might be dependent on another microservice, and so on and so forth.

And **what would happen if one of these services is down or is very slow?**

Microservice4 is down or it's very, very slow.

**What would happen?**

There would be an impact on the entire chain.

If the Microservice4 is down, then Microservice3 also will be down, Microservice2 also will be down because these are all depending on Microservice4. Even if it's slow, then there is a corresponding impact on the other microservices too.

In these microservices, there will be a build-up of calls. Because this microservice is slow, all these chains also get impacted.

So, the questions are, **can we return a fallback response if a service is down?**

If I see that the Microservice4 is down, in the Microservice3, can I return a fallback response?

**Can I configure a default response?**

For example,

In the case of a credit card transaction or something of that kind, we do not have any fallback responses possible, but in the case of a shopping application, instead of returning a set of products, we might return a default set of products.

that's possible.

Text

Description automatically generated

**Rate limiting**

Rate limiting means it allow only a certain number of calls to a specific microservice in a specific period of time.

<https://reflectoring.io/retry-with-springboot-resilience4j/>

Resilience4j is a lightweight, easy-to-use fault tolerance library inspired by [Netflix Hystrix](https://github.com/Netflix/Hystrix), but designed for Java 8 and functional programming. Lightweight, because the library only uses [Vavr](http://www.vavr.io), which does not have any other external library dependencies.

Text

Description automatically generated with low confidence

**package** com.amit.microservices.currencyexchangeservice.controller;

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

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

@RestController

**public** **class** CircuitBreakerController {

@GetMapping("sample-api")

**public** String sampleApi() {

**return** "Sample API";

}

}

<dependency>

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

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

</dependency>

<dependency>

<groupId>io.github.resilience4j</groupId>

<artifactId>resilience4j-spring-boot2</artifactId>

</dependency>

Graphical user interface, text, application, chat or text message

Description automatically generated

***Step 27 – Playing with Resilience4j – Retry and fallback methods***

**@Retry**

**By default, @Retry, if there is any failure in the execution of this specific method, then it will be retried thrice and if the retry fails all the three times, only then it would return error back.**

<https://reflectoring.io/retry-with-resilience4j/>

@RestController

**public** **class** CircuitBreakerController {

**private** Logger logger = LoggerFactory.*getLogger*(CircuitBreakerController.**class**);

@GetMapping("sample-api")

@Retry(name="default")

**public** String sampleApi() {

logger.info("\*\*\*\*\*\*\*\*\*\* Sample-api call received \*\*\*\*\*\*\*\*\*\*\*\*");

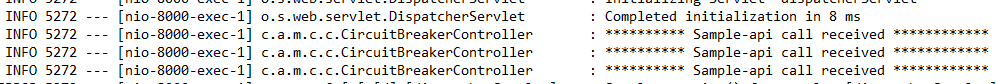
ResponseEntity<String> forEntity =

**new** RestTemplate().getForEntity("http://localhost:8080/some-dummy-rul", String.**class**);

**return** forEntity.getBody();

}

}



**How can we configure a specific number of retry try and in the specified intervals of time?**

resilience4j.retry.instances.sample-api.maxAttempts=5

@GetMapping("sample-api")

//@Retry(name="default")

**@Retry(name="sample-api")**

**public** String sampleApi() {

logger.info("\*\*\*\*\*\*\*\*\*\* Sample-api call received \*\*\*\*\*\*\*\*\*\*\*\*");

ResponseEntity<String> forEntity =

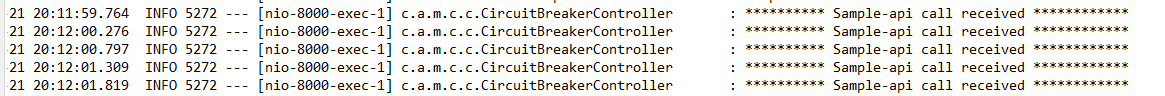
**new** RestTemplate().getForEntity("http://localhost:8080/some-dummy-rul", String.**class**);

**return** forEntity.getBody();

}

**Hit the below URL**

[**http://localhost:8000/sample-api**](http://localhost:8000/sample-api)



**One of the attributes on the annotation is fallback and you can configure what is the fallback method.**

### **Fallback Method**

Sometimes we may want to take a default action when all the retry attempts to the remote operation fail. This could be returning a default value or returning some data from a local cache.

@Retry(name="sample-api", fallbackMethod ="hardcodedResponse")

**public** String hardcodedResponse(Exception exe) {

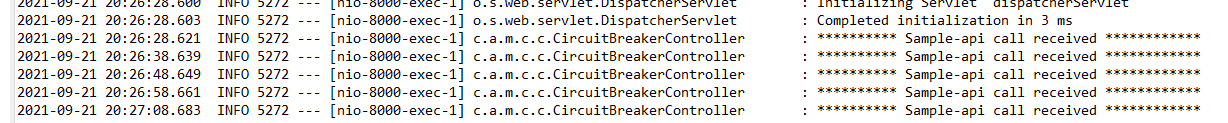
**return** "fallback-response";

}

**We are getting a fallback-response back, we can see that five retries happened and at the end of the five retries, the API call was still not successful and therefore, the fallback-response is returned back.**

**We can also configure time interval between retries.**

resilience4j.retry.instances.sample-api.waitDuration=10s

add above property in the application.property file

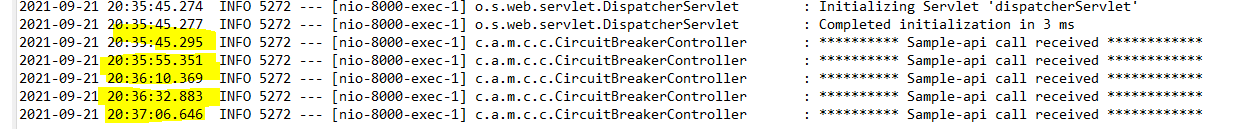
**Each retry happen in between the 10 seconds.**

Graphical user interface, application, Teams

Description automatically generated **What is exponential backoff?**

resilience4j.retry.instances.sample-api.enableExponentialBackoff=true

Because each subsequent request, it would wait for a little longer.



***Step 28 – Playing with Circuite Breakers features of Resilience4j***

**What if the service is down for a long time?**

We must go for a Circuit Breaker pattern.

**CircuitBreaker annotation What is the magic it brings?**

We would need to fire a lot of requests to this specific API.

**How can I fire a lot of requests to this specific API?**

Through the curl command curl http://localhost:8000/sample-api

Or we can manually fire these requests from the web page as well by just refreshing the screen.

The CircuitBreaker is returning the response back without even calling sampleApi method.

We see the logs there is no log being generated in here. That means the method is not being called, directly the fallback method is being called and the response is returned back.



If a microservice4 is down, then other microservice3,2,1 call will also fail.

Then, why do I need to really call microservice4 and add load to it?

**Why can't I return a default response back directly?**

Here CircuitBreaker come into the picture it will break the circuit and it will directly return a response back.

**How do I know if the microservice is back up and I can call it again?**

To understand this, need to understand the working flow of CircuitBreaker.

Diagram

Description automatically generated

A circuit breaker can be in three different states: CLOSED, OPEN and HALF\_OPEN.

**What are the different states define it?**

**Closed** is when we are calling the dependent microservice continuously. So, in a closed state,

I'll always be calling the dependent microservice.

**Open state**, the CircuitBreaker will not call the dependent microservice, it would directly return the fall-back response.

**Half\_open state**, a CircuitBreaker would be sending a percentage of requests to the dependent

microservice and for rest of the requests, it would return the hardcoded response or the fall-back response.

<https://resilience4j.readme.io/docs/circuitbreaker>

## Create and configure a CircuitBreaker – it will gives the list of configuration for the CircuiteBreaker.

resilience4j.circuitbreaker.instances.default.failureRateThreshold=90

**When does a CircuitBreaker switch from one state to another?**

For example,

CircuitBreaker is in the closed state. When we start the application up, the CircuitBreaker

is typically in a closed state.

Let's say, we are calling the dependent microservice 10,000 times and we can see that all of them are failing or 90 percent of them are failing.

**In this kind of scenario, the CircuitBreaker would switch to open state.**

Once it switches to an open state, it waits for a little while, there's a wait duration that we can configure. After that wait duration, the CircuitBreaker would switch to a half\_open state. During the half\_open state, the CircuitBreaker would try and see if the dependent microservice is up. So, it sends a percentage of the request, we can configure how much percentage it would send, let's

say 10% or 20% of the requests to the dependent microservice,

and if it gets proper responses for that, then it would go back to the closed state.

If it does not get proper responses, then it would go back to the open state.

@RestController

**public** **class** CircuitBreakerController {

**private** Logger logger = LoggerFactory.*getLogger*(CircuitBreakerController.**class**);

@GetMapping("sample-api")

//@Retry(name="default")

//@Retry(name="sample-api", fallbackMethod ="hardcodedResponse")

@CircuitBreaker(name="default", fallbackMethod ="hardcodedResponse")

**public** String sampleApi() {

logger.info("\*\*\*\*\*\*\*\*\*\* Sample-api call received \*\*\*\*\*\*\*\*\*\*\*\*");

ResponseEntity<String> forEntity =

**new** RestTemplate().getForEntity("http://localhost:8080/some-dummy-rul", String.**class**);

**return** forEntity.getBody();

}

**public** String hardcodedResponse(Exception exe) {

**return** "fallback-response";

}

}

***Step 29 – Playing with Circuite Breakers features of Resilience4j***

**Change in Configuration**

Use maxAttempts instead of maxRetryAttempts

1. resilience4j.retry.instances.sample-api.maxAttempts=5 #NEW
2. #resilience4j.retry.instances.sample-api.maxRetryAttempts=5 #OLD

**Rate Limiting and the BulkHead features.**

**What is rate limiting?**

Basically, rate limiting is all about saying, in 10 seconds, we want to only allow 10000 calls to the specific API.

We are setting a time period and during that time period, We only want to allow a specific number of.

#for the rate limiter

resilience4j.ratelimiter.instances.default.limitForPeriod=2

resilience4j.ratelimiter.instances.default.limitRefreshPeriod=10s

#We want two request for every 10s seconds.

@GetMapping("sample-ratelimiter")

**@RateLimiter(name="default")**

**public** String rateLimiter() {

logger.info("\*\*\*\*\*\*\*\*\*\* rateLimiter call received \*\*\*\*\*\*\*\*\*\*\*\*");

**return** "\*\*\*\*\*\* Sample Rate limiter \*\*\*\*\*";

}

**What is BulkHead?**

In addition to RateLimiter, we can also configure how many concurrent calls are allowed That's called BulkHead. For each of the APIs inside a microservice, we can configure a bulkhead.

We can allow maximum of 10 concurrent calls by configuring below.

@GetMapping("sample-bulkhead-default")

@Bulkhead(name="**default**")

**public** String bulkHeadDefault() {

logger.info("\*\*\*\*\*\*\*\*\*\* bulkHead call received \*\*\*\*\*\*\*\*\*\*\*\*");

**return** "\*\*\*\*\*\* Bulk Head \*\*\*\*\*";

}

resilience4j.ratelimiter.instances.**default**.maxConcurrentCalls=10

@GetMapping("sample-bulkhead")

@Bulkhead(name="**sample-bulkhead**")

**public** String bulkHead() {

logger.info("\*\*\*\*\*\*\*\*\*\* bulkHead call received \*\*\*\*\*\*\*\*\*\*\*\*");

**return** "\*\*\*\*\*\* Bulk Head \*\*\*\*\*";

}

resilience4j.ratelimiter.instances.default.**sample-bulkhead**.maxConcurrentCalls=2