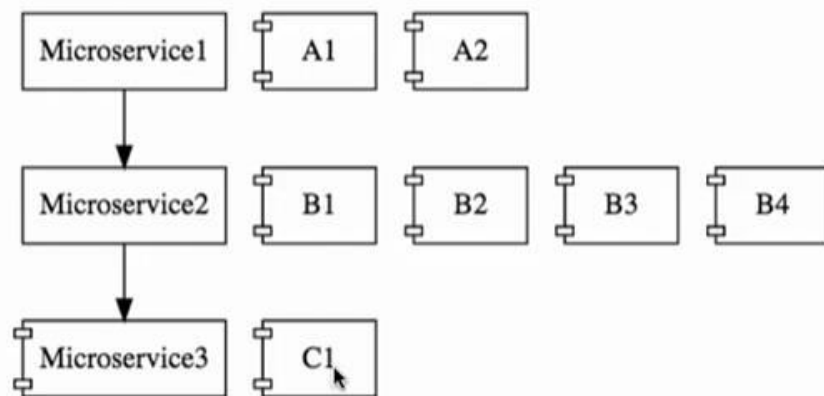


Udemy Spring Microservices

Microservices

- Small autonomous services that work together.
- Microservice Architectural style is an approach to develop a single application as a suite of small services, each running in its own process & communicating with lightweight mechanism like HTTP resource API.
- These services are built around business capabilities & independently deployable by fully automated deployment machinery.
- There is a bare minimum set of centralized management of these services, which may be written in different programming languages & use different data storage technologies.
- Microservices are
 1. RESTful web services
 2. & Small Well Chosen Deployable Units
 3. & **Cloud Enabled** (cloud enabled means if there is more load on Microservice3, then we should easily bring up another instance of Microservice3 like C2, C3 or bring down instance of Microservice2.... Without involving lot of configurations.



• Challenges with building microservices

1. **Bounded context:** how do we identify the boundary for each of the microservices i.e., how do we decide what a microservice should do & what a microservice should not. Deciding the boundaries of microservices is an evolutionary process.

Note: This is especially more difficult for new application because probably we don't really have the business knowledge to be able to establish the right boundaries b/w the microservices.

2. **Configuration Management:** Let's say there are 10 microservices with 5 environments & let's say 50 instances, then we're talking about basically tons of configuration & that's a lot of work for the operation team to maintain.
3. **Dynamic Scale up & Scale down:** We need the ability to dynamically bring in new instances & also to distribute the load among the new instances.
4. **Visibility & Monitoring:** Let's say the functionality is now distributed among 10 microservices & there's a bug, how do we identify where the bug is? We need to have a centralized log from where we can find out what happened for a specific request & which microservice caused the problem. Also, we need the microservice monitoring to know if they are down or identify servers where there is not enough disc space.

5. Pack of Cards or Fault Tolerance: If Microservice architecture is not well designed then it's like Pack of Cards i.e., if we have one microservice calling another, another calling another & so on then there would be certain microservices which would be the fundamental for the whole architecture & if that microservice goes down then whole design will fail. That's why it's very important to have fault tolerance in our microservices.

- To the typical problems which are present for distributed systems in the cloud, Spring Cloud provides a range of solutions.

Spring Cloud

- Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g., Configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state)

- Features of Spring Cloud

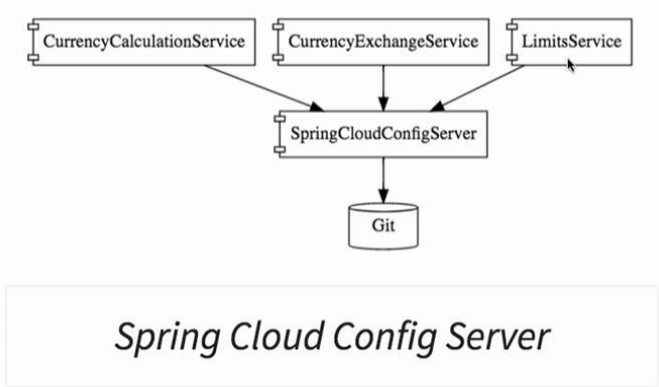
Distributed / versioned configuration	Service Registration & discovery
Routing	Service – to – Service calls
Load balancing	Circuit breakers
Distributed messaging	Short lived microservices (tasks)
Consumer – driven & producer – driven contract testing	

- How Spring Cloud resolves the microservices limitations

1. Configuration Management:

- Spring Cloud Config Server provides an approach where we can store all the configuration for all the different environments of all the microservices in a Git repository and Spring Cloud Config server can be used to expose that configuration to all the microservices.

- This helps us to keep the configuration in one place and that makes it very easy to maintain the configuration for all microservices.



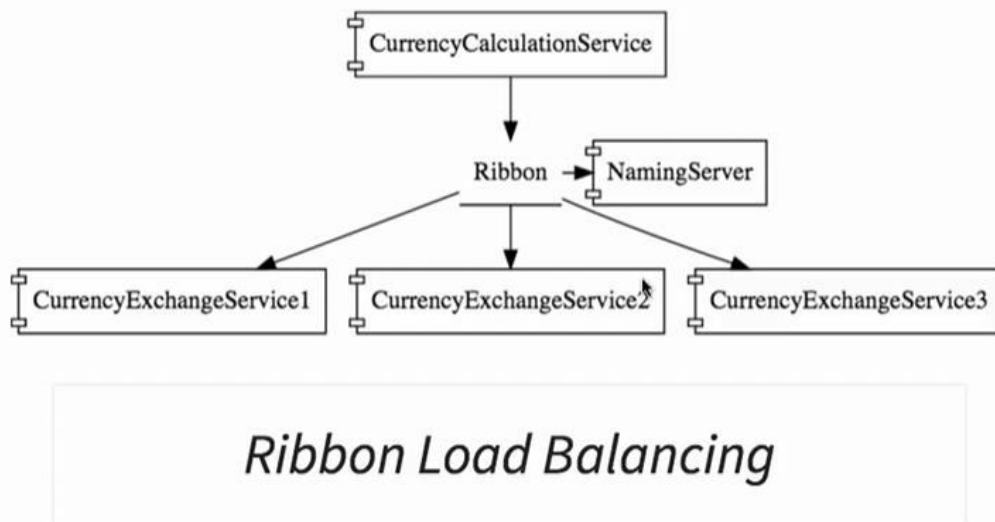
2. Dynamic Scale up & Scale down:

- To bring new microservice instances dynamically & distribute load among the instances, all of the microservice instances would need to register with the naming server.
- The naming server (Eureka) has 2 important features
 - a) Service Registration
 - b) Service Discovery.

DYNAMIC SCALE UP AND DOWN

- Naming Server (Eureka)
- Ribbon (Client Side Load Balancing)
- Feign (Easier REST Clients)

- In the below diagram, we can see there are 3 instances of *CurrencyExchangeService* microservice, Now the *CurrencyCalculationService* microservice will ask the **Naming Server** to give it the current instances of *CurrencyExchangeService*. The **Naming Server** will provide the URLs of current *CurrencyExchangeService* instances to the *CurrencyCalculationService*.
- This helps to establish the dynamic relationship b/w the **CurrencyCalculationService** & the instances of the **CurrencyExchangeService**.



- We will use **Ribbon for client-side load balancing** i.e., *CurrencyCalculationService* will host Ribbon & it would make sure that the load is evenly distributed among the existing instances that it gets from the Naming Server.
- We'll also use Feign in the *CurrencyCalculationService* as a mechanism to write simple RESTful clients.

3. Visibility & Monitoring

- The solution for visibility & monitoring are **Zipkin Distributed Tracing**.
- We would use Spring Cloud Sleuth to assign an ID to a request across multiple components & we would use Zipkin Distributed Tracing to trace a request across multiple components.
- One of the important things about microservices is these microservices have a lot of common features like Logging, Security, Analytics & things like that & we don't want to implement all these common features in each microservices.
- API Gateways (**Netflix Zuul API Gateway**) provide great Solutions to this kind of challenges.

4. Fault Tolerance

- We can implement fault tolerance using **Hystrix**.
- If a service is down, *Hystrix* helps us to configure a default response.

- Spring Cloud is not really one project as such instead there are a wide variety of projects under the umbrella of Spring Cloud.

No.	Project	Description
1.	Spring Cloud Config	Centralized external configuration management backed by git repository. The Configuration resources map directly to Spring Environment but could be used by non-Spring application if needed.
2.	Spring Cloud Gateway	Spring Cloud Gateway is an intelligent & programmable router based on Spring Framework & Spring Boot.
3.	Spring Cloud Netflix	Integration with Eureka Service Discovery from Netflix OSS.
4.	Spring Cloud Consul	Service discovery & configuration management with Hashicorp Consul.
5.	Spring Cloud Data Flow	A cloud-native orchestration service for composable microservice apps

		on modern runtimes. Easy-to-use DSL, drag-and-drop GUI, and REST-APIs together simplifies the overall orchestration of microservice based data pipelines.
6.	Spring Cloud Function	Spring Cloud Function promotes the implementation of business logic via functions. It supports a uniform programming model across serverless providers, as well as the ability to run standalone (locally or in a PaaS).
7.	Spring Cloud Stream	A lightweight event-driven microservices framework to quickly build applications that can connect to external systems. Simple declarative model to send and receive messages using Apache Kafka or RabbitMQ between Spring Boot apps.
Spring Cloud Stream Application, Spring Cloud Task, Spring Cloud Task App Starters, Spring Cloud Zookeeper, Spring Cloud Contract, Spring Cloud OpenFeign, Spring Cloud Bus, Spring Cloud Open Service Broker		

- Advantages of Microservice Architecture

- New technology & Process adaption:** It enables us to adapt to new technology & process very easily i.e., each of the microservices can be built in different technologies.
- Dynamic Scaling:** Depending on the amount of traffic / user / load, if our microservices are cloud enabled, then we can scale up or down microservices dynamically.
- Faster Release cycle:** Since we're developing smaller components, it's much easier to release microservices compared to monolith applications i.e., we can bring new features faster to the market.

- Since we will be using lots of different projects, so its better to standardize the ports on which we would run different projects.

Ports

Application	Port
Limits Service	8080, 8081, ...
Spring Cloud Config Server	8888
Currency Exchange Service	8000, 8001, 8002, ..
Currency Conversion Service	8100, 8101, 8102, ...
Netflix Eureka Naming Server	8761
Netflix Zuul API Gateway Server	8765
Zipkin Distributed Tracing Server	9411

URLs

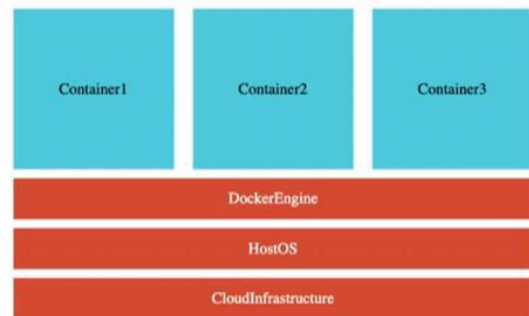
Application	URL
Limits Service	http://localhost:8080/limits POST -> http://localhost:8080/application/refresh
Spring Cloud Config Server	http://localhost:8888/limits-service/default http://localhost:8888/limits-service/dev
Currency Converter Service - Direct Call	http://localhost:8100/currency-converter/from/USD/to/INR/quantity/10
Currency Converter Service - Feign	http://localhost:8100/currency-converter-feign/from/EUR/to/INR/quantity/10000
Currency Exchange Service	http://localhost:8000/currency-exchange/from/EUR/to/INR http://localhost:8001/currency-exchange/from/USD/to/INR
Eureka	http://localhost:8761/
Zuul - Currency Exchange & Exchange Services	http://localhost:8765/currency-exchange-service/currency-exchange/from/EUR/to/INR http://localhost:8765/currency-conversion-service/currency-converter-feign/from/USD/to/INR/quantity/10
Zipkin	http://localhost:9411/zipkin/
Spring Cloud Bus Refresh	http://localhost:8080/bus/refresh

- There has been a lot of evolution in microservices in the last few years. For Spring Boot V2.3 or lower, we were using old technologies.
- From New Spring Boot versions, we will be using Spring Cloud, Docker, Kubernetes.

Microservices - V2

In
M

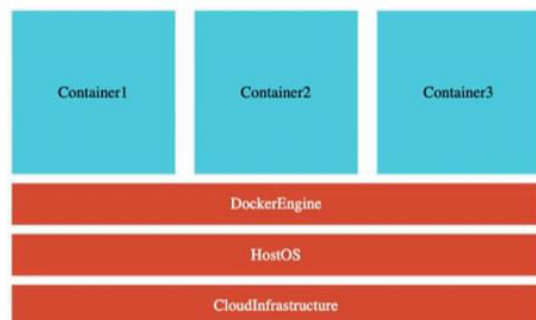
- **V2 - Latest Releases of**
 - Spring Boot
 - Spring Cloud
 - Docker and
 - Kubernetes
 - **Skip to Next Section :)**
- **V1 - Old Versions**
 - Spring Boot v2.3 and LOWER



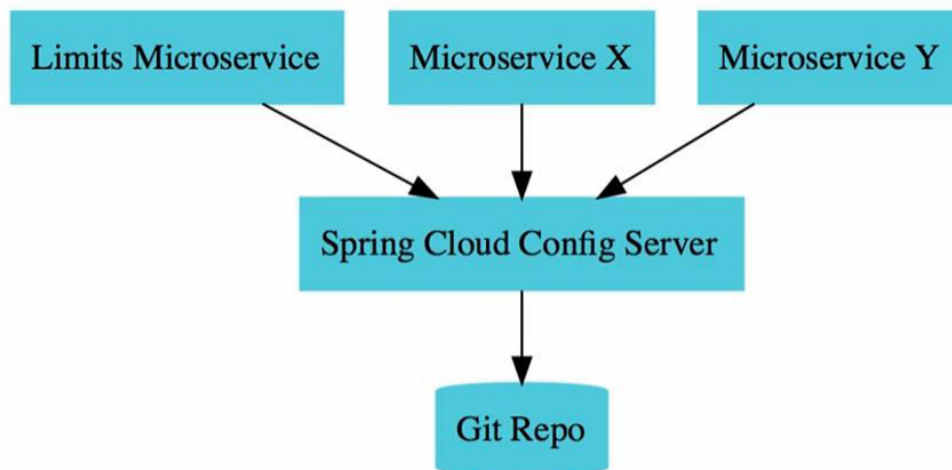
Microservices - V2 - What's New

In
Mi

- Microservices Evolve Quickly
- **Important Updates:**
 - Latest Versions of Spring Boot & Spring Cloud
 - Spring Cloud LoadBalancer instead of Ribbon
 - Spring Cloud Gateway instead of Zuul
 - Resilience4j instead of Hystrix
 - **Docker:** Containerize Microservices
 - Run microservices using Docker and Docker Compose
 - **Kubernetes:** Orchestrate all your Microservices with Kubernetes



Spring Microservice Example



Config Server

1. Create a microservice named **limit-service** (will act as **spring-cloud-config-client** & will use spring-cloud-config-server to get configurations). Here limit-service name act as the default microservice id or spring cloud config name for config-server. If we want to give customized config name, we can add this property to application.properties "**spring.cloud.config.name=**"

URL: <http://localhost:8080/limits>

```
LimitsController.java ×
9  @RestController
10 public class LimitsController {
11     2 usages
12     @Autowired
13     private Configuration config;
14
15     no usages
16     @GetMapping("/limits")
17     public Limits retrieveLimits() {
18         return new Limits(config.getMinimum(), config.getMaximum());
19     }
20 }

Configuration.java ×
7  @Data
8  @Component
9  @ConfigurationProperties("limits-service")
10 public class Configuration {
11     private int minimum;
12     private int maximum;
13 }

application.properties ×
1  spring.application.name=limits-service
2  spring.config.import=optional:configserver:http://localhost:8888
3  limits-service.minimum=3
4  limits-service.maximum=997
5  # limits-service values in application.properties
6  # has less priority than values in spring-cloud-config-server
```

2. Create a microservice named **spring-cloud-config-server** (will act as **config-server** & communicate with git repo to fetch configurations).

URL: <http://localhost:8888/limits-service/default> (For default profile)

```

SpringCloudConfigServerApplication.java
6
7 @EnableConfigServer
8 @SpringBootApplication
9 public class SpringCloudConfigServerApplication {
10
11     public static void main(String[] args) {
12         SpringApplication.run(SpringCloudConfigServerApplication.class, args);
13     }
14 }
15 }

application.properties
1 spring.application.name=spring-cloud-config-server
2 server.port=8888
3 spring.cloud.config.server.git.uri=file:///D:/IntelliJWS/MicroServicesCourse/git-localconfig-repo
4 # file:///C:/Users/home/Desktop/yourproject/git-repo

```

Note: We can have configuration based on different profiles or environment like dev, qa etc. We can create separate config files for each environment & add these 2 properties to limit-service microservice to specify the active profile.

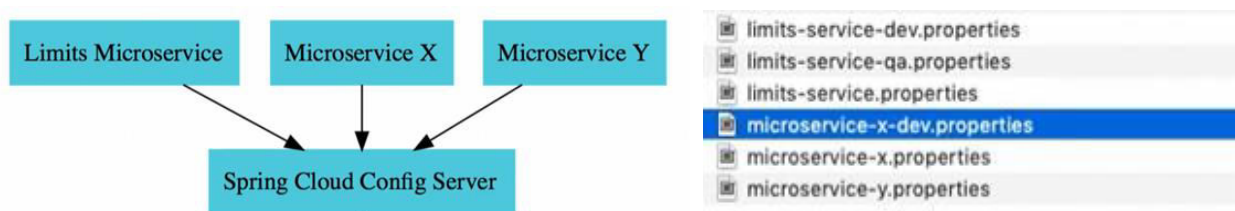
```

MicroServicesCourse
├── git-localconfig-repo
│   ├── limits-service.properties
│   ├── limits-service-dev.properties
│   └── limits-service-qa.properties
├── limit-service
└── spring-cloud-config-server

# Profile can be [dev, qa]
spring.profiles.active=dev
spring.cloud.config.profile=dev

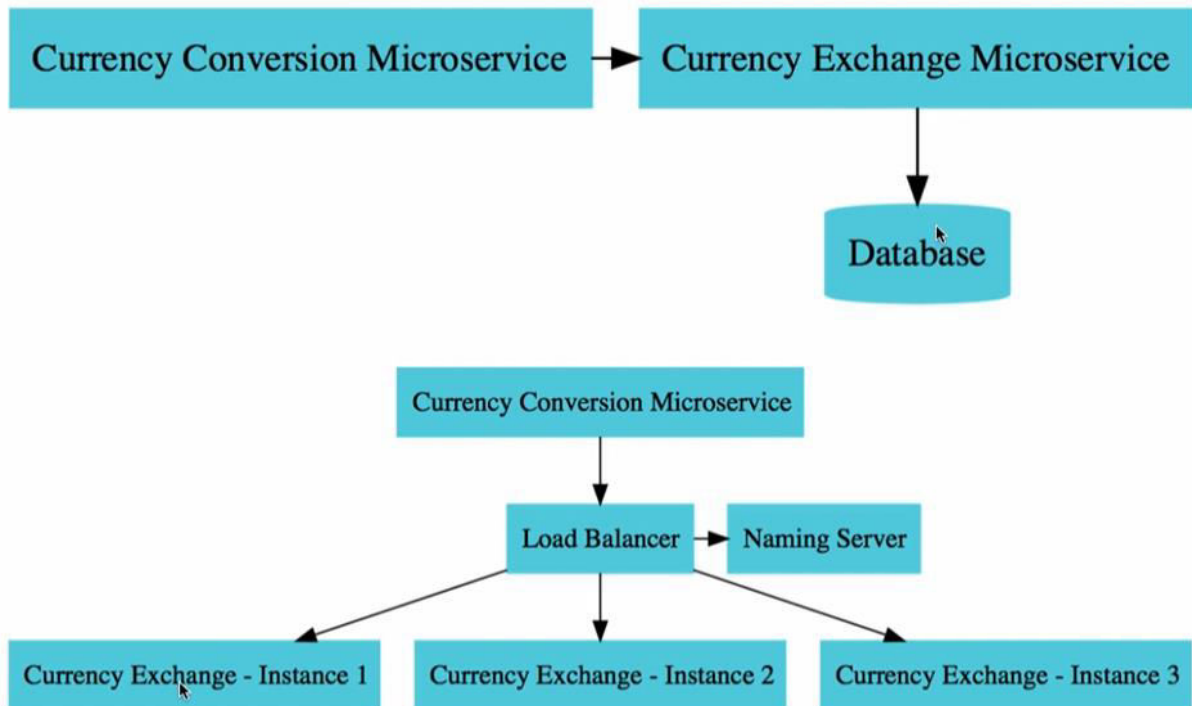
```

- In case we're having multiple instances of limit-service like microservice-x, microservice-y then we can create separate configuration file in git-localconfig-repo.
- This way we can control all configuration related to all microservices for multiple environments in a single location. So, whenever there would be change in an application, then we can make a change in GitHub repository & commit it.



Ports Standardization

Application	Port
Limits Microservice	8080, 8081, ...
Spring Cloud Config Server	8888
Currency Exchange Microservice	8000, 8001, 8002, ..
Currency Conversion Microservice	8100, 8101, 8102, ...
Netflix Eureka Naming Server	8761
API Gateway	8765
Zipkin Distributed Tracing Server	9411



Load Balancing


Naming Server or Service Registration (Eureka)

- In order to dynamically launch currency-exchange ms instances & distribute load among them, we will use Eureka as Naming server & we register all the microservices to eureka.

Eureka URL: <http://localhost:8761/>

- This way as instances come up or go down, naming-server will be able to automatically discover them & load balance among them.
- Suppose currency-conversion ms wants to talk to currency-exchange ms, it would ask the service registry/naming server for the address of the currency-exchange ms & then currency-conversion ms can send request out to the currency-exchange ms.
- Dependency to add to pom.xml to create naming-server as Eureka: **spring-cloud-starter-netflix-eureka-server**

```
> naming-server > src > main > resources > application.properties
application.properties x
1  spring.application.name=naming-server
2  server.port=8761
3
4  eureka.client.register-with-eureka=false
5  eureka.client.fetch-registry=false
```



HOME LAST 1000 SINCE STARTUP

System Status

Environment	test	Current time	2024-02-17T16:04:09 +0530
Data center	default	Uptime	00:00
		Lease expiration enabled	false
		Renews threshold	6
		Renews (last min)	0

DS Replicas

[localhost](#)

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-CONVERSION	n/a (1)	(1)	UP (1) - LAPTOP-J489HHHD:currency-conversion:8100
CURRENCY-EXCHANGE	n/a (2)	(2)	UP (2) - LAPTOP-J489HHHD:currency-exchange:8000 , LAPTOP-J489HHHD:currency-exchange:8001

Currency Exchange Microservice

- It's all about what is the exchange rate of one current in another. It would return back the conversion multiple saying 1 USD = 65 INR.

Currency Exchange Microservice

What is the exchange rate of one currency in another?

```
http://localhost:8000/currency-exchange/from/USD/to/INR

{
  "id": 10001,
  "from": "USD",
  "to": "INR",
  "conversionMultiple": 65.00,
  "environment": "8000 instance-id"
}
```

- We will connect this microservice to DB using spring-data-jpa to fetch the conversionMultiple from DB.
- Later on, we will add Eureka as naming server & **register this microservice to the naming-server**. For this we need to add this property: **"eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka"**
- Pom.xml used dependency: spring-cloud-starter-netflix-eureka-client to specify the microservice to register with naming server.
- We can create multiple instances of this microservice by creating different intelliJ / eclipse run configuration for each port 8000, 8001, 8002 (add VM property as -Dserver.port=8001)

```

CurrencyExchangeController.java
13 @RestController
14 public class CurrencyExchangeController {
15     1 usage
16     @Autowired
17     private CurrencyExchangeRepository currencyExchangeRepository;
18     1 usage
19     @Autowired
20     private Environment environment;
21     no usages
22     @GetMapping("/currency-exchange/from/{from}/to/{to}")
23     public CurrencyExchange retrieveExchangeValue(@PathVariable String from,
24                                                  @PathVariable String to) {
25         CurrencyExchange currencyExchange = currencyExchangeRepository.findByFromAndTo(from, to);
26         if(currencyExchange == null)
27             throw new RuntimeException("Unable to find data for " + from + " to " + to);
28         String port = environment.getProperty("local.server.port");
29         currencyExchange.setEnvironment(port);
30         return currencyExchange;
31     }
32 }

application.properties
1 spring.application.name=currency-exchange
2 server.port=8000
3
4 spring.config.import=optional:configserver:http://localhost:8888
5
6 spring.jpa.show-sql=true
7 spring.datasource.url=jdbc:h2:mem:testdb
8 spring.h2.console.enabled=true
9 spring.jpa.defer-datasource-initialization=true
10
11 eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

```

- URLs: (For 2 instances of currency-exchange-microservice at port 8000 & 8001, Records are fetched from DB)
<http://localhost:8000/currency-exchange/from/USD/to/INR>
or <http://localhost:8001/currency-exchange/from/USD/to/INR>

<http://localhost:8000/currency-exchange/from/EUR/to/INR>
or <http://localhost:8001/currency-exchange/from/EUR/to/INR>

<http://localhost:8000/currency-exchange/from/AUD/to/INR>
or <http://localhost:8001/currency-exchange/from/AUD/to/INR>

```

{
  "id": 10001,
  "from": "USD",
  "to": "INR",
  "conversionMultiple": 65,
  "environment": "8000"
}

```

```

{
  "id": 10002,
  "from": "EUR",
  "to": "INR",
  "conversionMultiple": 75,
  "environment": "8000"
}

```

```

{
  "id": 10003,
  "from": "AUD",
  "to": "INR",
  "conversionMultiple": 25,
  "environment": "8000"
}

```

- We can see these 2 instances of currency-exchange-microservice in the naming-server (Eureka server)
Eureka server URL: <http://localhost:8761/>

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
CURRENCY-EXCHANGE	n/a (2)	(2)	UP (2) - LAPTOP-J489HHHD:currency-exchange:8000 , LAPTOP-J489HHHD:currency-exchange:8001

Currency Conversion Microservice

- It is responsible for converting USD into INR by using conversionMultiple from currency-exchange & multiplying it with the quantity to get totalCalculatedAmount.
- Similar to currency-exchange ms, we can register this ms to eureka naming server.

Currency Conversion Microservice

Convert 10 USD into INR

```
http://localhost:8100/currency-conversion/from/USD/to/INR/quantity/10

{
  "id": 10001,
  "from": "USD",
  "to": "INR",
  "conversionMultiple": 65.00,
  "quantity": 10,
  "totalCalculatedAmount": 650.00,
  "environment": "8000 instance-id"
}
```

- In order to communicate with currency-exchange-microservice, we need to perform a REST call to currency-exchange-microservice.
- Point to think while performing REST call to currency-exchange-microservice
 1. Since there can be multiple instances of currency-exchange-microservice running at different ports. If we hardcode one port to call then we're losing load balancing feature.
 2. We have to write a lot of tedious code around RestTemplate to get the currency-conversion ms to talk with currency-exchange ms. Imagine what will happen if we have 100s of microservices calling each other then we need to repeat this code everywhere.
- **Solution to above problems:** Spring Cloud provides a framework called **FEIGN** or Open FEIGN to handle call these service invocations or calls using a proxy class & to use this, we need to add specific dependency to pom.xml. i.e., **spring-cloud-starter-openfeign** (it has inbuilt spring-cloud-starter-loadbalancer)
- Using Open FEIGN, we will call currency-exchange ms from currency-conversion ms using currency-exchange-proxy.

```
1 package com.srvcode.microservices.controller;
2
3 import ...
4
5
6
7
8 // @FeignClient(name = "currency-exchange", url = "localhost:8000")
9 @FeignClient(name = "currency-exchange")
10 public interface CurrencyExchangeProxy {
11     1 usage
12     @GetMapping("/currency-exchange/from/{from}/to/{to}")
13     public CurrencyConversion retrieveExchangeValue(@PathVariable String from, @PathVariable String to);
14 }
```

```

CurrencyConversionServiceApplication.java x
7  @SpringBootApplication
8  @EnableFeignClients
9  public class CurrencyConversionServiceApplication {
10
11  public static void main(String[] args) {
12      SpringApplication.run(CurrencyConversionServiceApplication.class, args);
13  }
14
15  }

CurrencyConversionController.java x
13
14  no usages
15  @RestController
16  public class CurrencyConversionController {
17      1 usage
18      @Autowired
19      private CurrencyExchangeProxy proxy;
20      no usages
21      @GetMapping("/currency-conversion-feign/from/{from}/to/{to}/quantity/{quantity}")
22      public CurrencyConversion calculateCurrencyConversionFeign(
23          @PathVariable String from, @PathVariable String to, @PathVariable BigDecimal quantity) {
24
25          CurrencyConversion currencyConversion = proxy.retrieveExchangeValue(from, to);
26
27          return new CurrencyConversion(currencyConversion.getId(), from, to, quantity,
28              currencyConversion.getConversionMultiple(),
29              quantity.multiply(currencyConversion.getConversionMultiple()),
30              environment: currencyConversion.getEnvironment() + " feign");
31  }
32  }

application.properties x
1  spring.application.name=currency-conversion
2  server.port=8100
3  spring.config.import=optional:configserver:http://localhost:8888
4
5  eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka

```

- URLs: (For currency-conversion-microservice running at port 8100)

<http://localhost:8100/currency-conversion-feign/from/USD/to/INR/quantity/100>

<http://localhost:8100/currency-conversion-feign/from/EUR/to/INR/quantity/100>

<http://localhost:8100/currency-conversion-feign/from/AUD/to/INR/quantity/100>

```

{
  "id": 10001,
  "from": "USD",
  "to": "INR",
  "quantity": 100,
  "conversionMultiple": 65,
  "totalCalculatedAmount": 6500,
  "environment": "8000 feign"
}

{
  "id": 10002,
  "from": "EUR",
  "to": "INR",
  "quantity": 100,
  "conversionMultiple": 75,
  "totalCalculatedAmount": 7500,
  "environment": "8001 feign"
}

{
  "id": 10003,
  "from": "AUD",
  "to": "INR",
  "quantity": 100,
  "conversionMultiple": 25,
  "totalCalculatedAmount": 2500,
  "environment": "8001 feign"
}

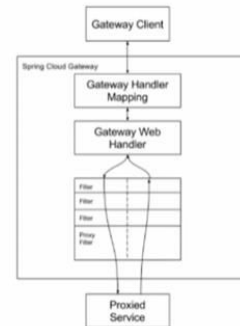
```

API Gateway

- In a typical microservices architecture, there would be hundreds of microservices & these microservices have a lot of common features like Authentication, Authorization, logging, rate limiting etc.

Spring Cloud Gateway

- Simple, yet effective way to route to APIs
- Provide cross cutting concerns:
 - Security
 - Monitoring/metrics
- Built on top of Spring WebFlux (Reactive Approach)
- Features:
 - Match routes on any request attribute
 - Define Predicates and Filters
 - Integrates with Spring Cloud Discovery Client (Load Balancing)
 - Path Rewriting



From <https://docs.spring.io>

- We use API Gateway to provide these common features to all microservices.
- We can register the API gateway with the Eureka naming server.
- Popular API Gateway: Zool (no longer supported), Spring Cloud Gateway (Recommended)
- Dependencies to pom.xml

Eureka Discovery Client SPRING CLOUD DISCOVERY
A REST based service for locating services for the purpose of load balancing and failover of middle-tier servers.

Gateway SPRING CLOUD ROUTING
Provides a simple, yet effective way to route to APIs and provide cross cutting concerns to them such as security, monitoring/metrics, and resiliency.

```
<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>
```

Enabling Discovery locator with Eureka for Spring Cloud Gateway

- API Gateway URLs (Here we're passing the microservice registered name with Eureka to API Gateway URLs.

<http://localhost:8765/>

Instances currently registered with Eureka

Application	AMIs	Availability Zones
API-GATEWAY	n/a (1)	(1)
CURRENCY-CONVERSION	n/a (1)	(1)
CURRENCY-EXCHANGE	n/a (2)	(2)

<http://localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/USD/to/INR>
<http://localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/EUR/to/INR>
<http://localhost:8765/CURRENCY-EXCHANGE/currency-exchange/from/AUD/to/INR>

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

<http://localhost:8765/CURRENCY-CONVERSION/currency-conversion-feign/from/USD/to/INR/quantity/100>
<http://localhost:8765/CURRENCY-CONVERSION/currency-conversion-feign/from/EUR/to/INR/quantity/100>
<http://localhost:8765/CURRENCY-CONVERSION/currency-conversion-feign/from/AUD/to/INR/quantity/100>

- We can call any microservices like currency-exchange or currency-conversion ms which is registered with Eureka through the API Gateway (assuming API Gateway is already registered with Eureka) by passing the microservice registered name with Eureka to API Gateway URLs.
- To enable this feature we need to add this property: "**spring.cloud.gateway.discovery.locator.enabled=true**"
- This way we can implement all the common features in API gateway & the gateway would take care of the common features & then invoke required microservice.
- To enable **lower case microservice id**, we can add this property:
spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true
- This is the way we can proxy through the API Gateway to other microservices registered to Eureka using gateway discovery locator feature.

Custom Routes

- We can remove registered service id with eureka with some custom route in API gateway.
- We can build custom routes by creating a configuration file & **also disable locator** in application.properties file.

```
ApiGatewayConfiguration.java
8  @Configuration
9  public class ApiGatewayConfiguration {
10     no usages
11     @Bean
12     @ public RouteLocator gatewayRouter(RouteLocatorBuilder builder) {
13         return builder.routes()
14             .route(p-> p.path( ...patterns: "/get") BooleanSpec
15                 .filters(f -> f
16                     .addRequestHeader( headerName: "MyHeader", headerValue: "MyURI")
17                     .addRequestParameter( param: "Param", value: "MyValue")) UriSpec
18                 .uri("http://httpbin.org:80"))
19             .route(p -> p.path( ...patterns: "/currency-exchange/**")
20                 .uri(("lb://currency-exchange")))
21             .route(p -> p.path( ...patterns: "/currency-conversion/**")
22                 .uri(("lb://currency-conversion")))
23             .route(p -> p.path( ...patterns: "/currency-conversion-feign/**")
24                 .uri(("lb://currency-conversion")))
25             .route(p -> p.path( ...patterns: "/currency-conversion-new/**") BooleanSpec
26                 .filters(f -> f.rewritePath(
27                     regex: "/currency-conversion-new/(?<segment>.*)",
28                     replacement: "/currency-conversion-feign/${segment}")) UriSpec
29                 .uri("lb://currency-conversion"))
30             .build();
31     }
32 }
```

```
application.properties
1  spring.application.name=api-gateway
2  server.port=8765
3  eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka
4  spring.cloud.gateway.discovery.locator.enabled=true
5  #spring.cloud.gateway.discovery.locator.lowerCaseServiceId=true
```

- Here lb://url mean load balancing & url is the registered name on the eureka server.

- API Gateway Custom URLs:

<http://localhost:8765/get>

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

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

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

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

<http://localhost:8765/currency-conversion/from/EUR/to/INR/quantity/10>

<http://localhost:8765/currency-conversion/from/AUD/to/INR/quantity/10>

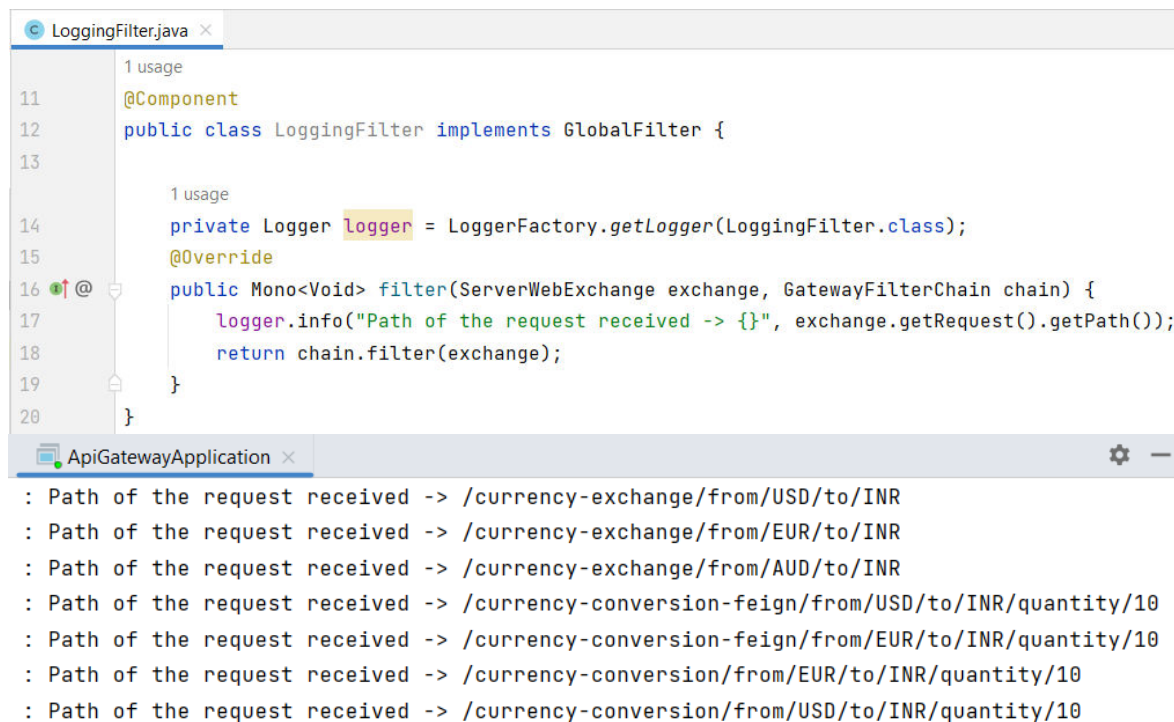
<http://localhost:8765/currency-conversion-feign/from/USD/to/INR/quantity/100>

<http://localhost:8765/currency-conversion-feign/from/EUR/to/INR/quantity/100>

<http://localhost:8765/currency-conversion-feign/from/AUD/to/INR/quantity/100>

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

- We can also add global filters in API Gateway Configuration. Let's say we would want to log every request that goes through the API Gateway.



```
LoggingFilter.java x
1 usage
11 @Component
12 public class LoggingFilter implements GlobalFilter {
13
14     1 usage
15     private Logger logger = LoggerFactory.getLogger(LoggingFilter.class);
16     @Override
17     public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
18         logger.info("Path of the request received -> {}", exchange.getRequest().getPath());
19         return chain.filter(exchange);
20     }
}

ApiGatewayApplication x
: Path of the request received -> /currency-exchange/from/USD/to/INR
: Path of the request received -> /currency-exchange/from/EUR/to/INR
: Path of the request received -> /currency-exchange/from/AUD/to/INR
: Path of the request received -> /currency-conversion-feign/from/USD/to/INR/quantity/10
: Path of the request received -> /currency-conversion-feign/from/EUR/to/INR/quantity/10
: Path of the request received -> /currency-conversion/from/EUR/to/INR/quantity/10
: Path of the request received -> /currency-conversion/from/USD/to/INR/quantity/10
```


Circuit Breaker (<https://resilience4j.readme.io/docs/getting-started>)

Circuit Breaker



- What if one of the services is down or is slow?
 - Impacts entire chain!
 - Questions:
 - Can we return a fallback response if a service is down?
 - Can we implement a Circuit Breaker pattern to reduce load?
 - Can we retry requests in case of temporary failures?
 - Can we implement rate limiting?
 - Solution: Circuit Breaker Framework - Resilience4j
- In a microservice architecture, there can be complex call chain where microservices can be dependent on other microservices & what if one or more microservices go down or become slow, then it will impact other microservices ultimately to the architecture.
 - Resilience4j, Netflix Hystrix (old) are some popular circuit breaker frameworks in spring cloud. With the evolution of Java 8 & functional programming, Resilience4j has become the recommended framework.
 - Resilience4j dependency:

```
<dependency>
  <groupId>io.github.resilience4j</groupId>
  <artifactId>resilience4j-spring-boot2</artifactId>
</dependency>
```

- **Resilience4j** is a lightweight fault tolerance library designed for functional programming.
- Resilience4j provides higher-order functions (decorators) to enhance any functional interface, lambda expression or method reference with a Circuit breaker, Rate limiter, Retry or Bulkhead.
- We can stack more than one decorator on any functional interface, lambda expression or method reference.
- The advantage is that we have the choice to select the decorators we need or nothing else.
- **Resilience4j 2** requires Java 17

Circuit Breaker Framework Feature – Retry

- If we add `@Retry` to an API, then that api will be retried the specific no. of times to get Successful response. It won't be retried if the API successfully executes first time.
- We can also add **fallbackMethod** property to `@Retry` in order to execute a method in case of unsuccessful API response.

```
CircuitBreakerController.java
12 @RestController
13 public class CircuitBreakerController {
14     1 usage
15     private Logger logger = LoggerFactory.getLogger(CircuitBreakerController.class);
16     no usages
17     @GetMapping("/sample-api")
18     @Retry(name = "sample-api", fallbackMethod = "hardcodedResponse")
19     public String sampleApi() {
20         logger.info("Sample API call received!");
21         ResponseEntity<String> forEntity = new RestTemplate()
22             .getForEntity("http://localhost:8080/some-dummy-url", String.class);
23         return forEntity.getBody();
24     }
25
26     no usages
27     public String hardcodedResponse(Exception ex) { return "fallback-response"; }
28 }
```

```
application.properties
12 resilience4j.retry.instances.sample-api.maxAttempts=5
13 resilience4j.retry.instances.sample-api.waitDuration=2s
14 resilience4j.retry.instances.sample-api.enableExponentialBackoff=true
15
```

```
2024-02-18T12:48:12.388+05:30 c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T12:48:14.403+05:30 c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T12:48:17.426+05:30 c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T12:48:21.940+05:30 c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T12:48:28.697+05:30 c.s.m.CircuitBreakerController : Sample API call received!
```

<http://localhost:8000/sample-api> : fallback-response

- When we're using **enableExponentialBackoff** then each subsequent request would wait for a little longer than specified wait Duration. In above example we can see 1st request took 2 sec then 3 sec then 4 sec then 7 sec & so on. For e.g., Most of AWS APIs use exponential back off.
- These are useful when a service is momentary down, we just give the service a little bit of time & then call it again. However if the service is really down for a long time, then we should go for Circuit Breaker Pattern.

Circuit Breaker Pattern (<https://resilience4j.readme.io/docs/circuitbreaker>)

- We can implement it using @CircuitBreaker annotation with name & fallbackMethod.

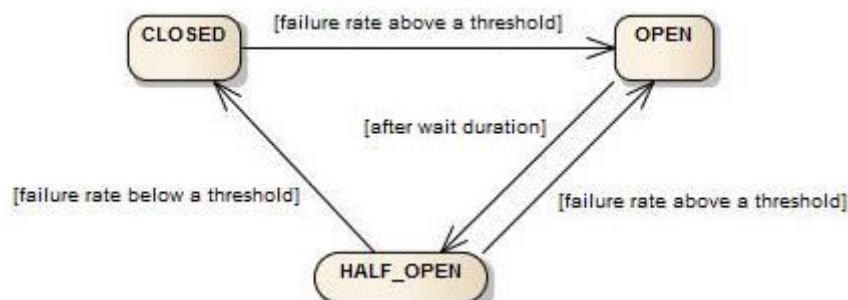
<http://localhost:8000/sample-api>

```
@RestController
public class CircuitBreakerController {
    1 usage
    private Logger logger = LoggerFactory.getLogger(CircuitBreakerController.class);
    no usages
    @GetMapping("/sample-api")
    @CircuitBreaker(name = "default", fallbackMethod = "hardcodedResponse")
    public String sampleApi() {
        logger.info("Sample API call received!");
        ResponseEntity<String> forEntity = new RestTemplate().getForEntity( url: "http://localhost:8080/some-dummy-url",
            String.class);
        return forEntity.getBody();
    }

    no usages
    public String hardcodedResponse(Exception ex) { return "fallback-response"; }
}
```

```
2024-02-18T16:02:50.501+05:30 [nio-8000-exec-5] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:50.661+05:30 [nio-8000-exec-4] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:50.807+05:30 [nio-8000-exec-9] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:50.964+05:30 [nio-8000-exec-8] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:51.107+05:30 [nio-8000-exec-2] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:51.289+05:30 [nio-8000-exec-5] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:51.442+05:30 [nio-8000-exec-4] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:51.608+05:30 [nio-8000-exec-9] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:02:51.755+05:30 [nio-8000-exec-8] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:51.837+05:30 [io-8000-exec-10] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:51.980+05:30 [nio-8000-exec-1] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:52.125+05:30 [nio-8000-exec-5] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:52.282+05:30 [nio-8000-exec-6] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:52.441+05:30 [nio-8000-exec-7] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:52.581+05:30 [io-8000-exec-10] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:52.723+05:30 [nio-8000-exec-1] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:52.874+05:30 [nio-8000-exec-5] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:53.022+05:30 [nio-8000-exec-6] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:03:53.166+05:30 [nio-8000-exec-7] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:53.190+05:30 [nio-8000-exec-7] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:53.340+05:30 [io-8000-exec-10] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:53.500+05:30 [nio-8000-exec-1] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:53.648+05:30 [nio-8000-exec-5] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:53.790+05:30 [nio-8000-exec-6] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:53.940+05:30 [nio-8000-exec-7] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:54.075+05:30 [io-8000-exec-10] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:54.213+05:30 [nio-8000-exec-1] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:54.371+05:30 [nio-8000-exec-5] c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:04:54.524+05:30 [nio-8000-exec-6] c.s.m.CircuitBreakerController : Sample API call received!
```

- The circuit breaker starts with Close state in order to allow API calls to execute but after a certain no. of failed API calls, the Circuit changes its status from Close to Open state & it waits for some time... then again the Circuit changes its status from Open to Half_Open in order to allow a certain percentage of API calls to pass.
- If API calls still fails then again it will wait for sometime before again sending request.



- We can refer these configs to customize circuit breaker behavior: <https://resilience4j.readme.io/docs/getting-started-3>

Circuit Breaker Framework – RateLimiter

- Rate Limiting means allowing only a specific number of API calls during a specified time period.
- For e.g., We are only allowing 10,000 API calls in 10 seconds
- We can implement this feature by using @RateLimiter annotation & providing required configuration to application.properties. Refer this for config: <https://resilience4j.readme.io/docs/getting-started-3>

<http://localhost:8000/sample-api>

```
resilience4j.ratelimiter.instances.default.limitForPeriod=2
resilience4j.ratelimiter.instances.default.limitRefreshPeriod=10s
```

```
@GetMapping("/sample-api")
@RateLimiter(name = "default")
public String sampleApi() {
    logger.info("Sample API call received!");
    return "sample-api";
}
```

```
2024-02-18T16:43:20.031 INFO c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:43:21.118 INFO c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:43:27.031 ERROR o.a.c.c.C.[.[]].[dispatcherServlet] :
Servlet.service() for servlet [dispatcherServlet] in context with path []
threw exception [Request processing failed: io.github.resilience4j.ratelimiter.RequestNotPermitted:
RateLimiter 'default' does not permit further calls] with root cause
io.github.resilience4j.ratelimiter.RequestNotPermitted: RateLimiter 'default' does not permit further calls
2024-02-18T16:43:29.247 INFO c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:43:29.247 INFO c.s.m.CircuitBreakerController : Sample API call received!
2024-02-18T16:43:34.623 ERROR o.a.c.c.C.[.[]].[dispatcherServlet] :
Servlet.service() for servlet [dispatcherServlet] in context with path []
threw exception [Request processing failed: io.github.resilience4j.ratelimiter.RequestNotPermitted:
RateLimiter 'default' does not permit further calls] with root cause
io.github.resilience4j.ratelimiter.RequestNotPermitted: RateLimiter 'default' does not permit further calls
```

- We can see in above example in the span of 10 sec, only 2 API calls are allowed & others are failing.

Circuit Breaker Framework – BulkHead

- For each of the APIs inside a microservice we can configure a bulkhead i.e., how many concurrent calls are allowed.

```
resilience4j.bulkhead.instances.default.maxConcurrentCalls=10
```

```
@GetMapping("/sample-api")
@Bulkhead(name = "default")
public String sampleApi() {
    logger.info("Sample API call received!");
    return "sample-api";
}
```