Spring Boot Microservices Tutorial - Part 6

In Part 6 of this Spring Boot Microservices Tutorial, we will learn how to implement Resiliency in our project by implementing the Circuit Breaker pattern. We will use the library **Resilience4J** together with **Spring Cloud Circuit Breaker Resilience4J** to implement the circuit breaker pattern in our project

What is Circuit Breaker Pattern?

Circuit Breaker is one of the widely used best practice in the real world distributed systems

Consider a scenario where your application **A** makes synchronous calls to a remote service **R**. If service **R** becomes unavailable or responds very slowly due to performance issues, this situation will negatively impact application **A** as well.

If the application **A** receives a large number of requests, then there will be lot of threads in the waiting state, waiting for the response from R, leading to ultimately crashing the application **A**. To avoid this issue, we can make use of the Circuit Breaker Pattern, which works very similar to the Circuit Breaker used in our homes to protect the electrical devices from the power spikes. If there is a power spike, then the Circuit Breaker is tripped and will stop the flow of electricity. Similarly, when the remote service **R** in our case, if it's unavailable or responding very slowly, we can introduce a Circuit Breaker that will stop the calls to the service, for a certain amount of time. After this timeout, the Circuit Breaker will again start allowing calls to the service **R** gradually.

In our Microservices Project, we can introduce this Circuit Breaker mechanism in the API Gateway and the Order Service.

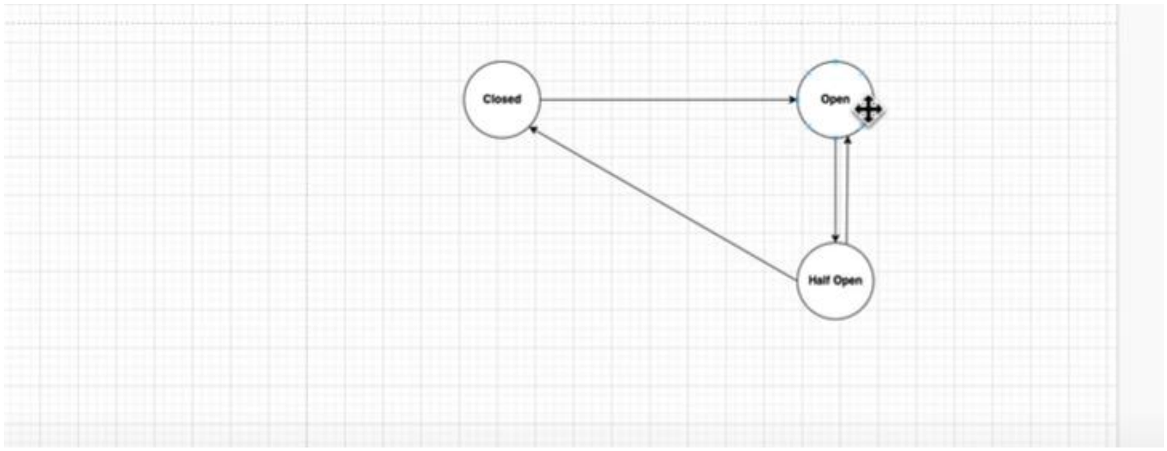
API Gateway is the main service that is calling 3 other services, so this will be the best place to use Circuit Breaker, similarly we can also implement this feature in the Order Service as the service is calling Inventory Service to fetch the inventory information.

Different States in the Circuit Breaker Pattern

At any given point of time, a circuit breaker will be in different states like:

- **Open**: This states indicates that the Circuit Breaker is open, and all the traffic going through the Circuit Breaker will be blocked.

- **Half-Open**: In this state, the Circuit Breaker will start allowing gradually the traffic to the remote service **R**
- **Closed**: In this state, the Circuit Breaker will allow all the requests to the service, which means that the service **R** is working well without any problems.



By default circuit breaker will be in closed state. If requested service is not reachable it will wait for 3sec as we mentioned in properties file.

```
resilience4j.timelimiter.configs.default.timeout-duration=3s
```

if still not reached it will go to the open state, means all the requests are blocked here. and after that it will go into half open state, here few requests will be allowed to check weather external service is working now or not. if still external service not reachable it will again go to open state. if its reachable it will go to closed state and all the requests are not blocked.

```
resilience4j.circuitbreaker.configs.default.slidingWindowType=COUNT_BASED
resilience4j.circuitbreaker.configs.default.slidingWindowSize=10
resilience4j.circuitbreaker.configs.default.failureRateThreshold=50
resilience4j.circuitbreaker.configs.default.waitDurationInOpenState=5s
resilience4j.circuitbreaker.configs.default.permittedNumberOfCallsInHalfOpenS
tate=3
resilience4j.circuitbreaker.configs.default.automaticTransitionFromOpenToHalf
OpenEnabled=true resilience4j.circuitbreaker.configs.default.minimum-number-
of-calls=5
```

Implement Circuit Breaker in the API Gateway

Now let's implement this pattern in our API Gateway project, for that I am going to add the following dependencies to the **pom.xml** of the API Gateway project

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

The first dependency adds the **Resilience4J** library in our project and the second dependency adds the Spring Boot Actuator that provides us with useful endpoints to get useful information about our application like **Metrics**, we can make use of these endpoints to check the state of the Resilience4J Circuit Breaker.

After adding the above dependency, we need to add the **circuitBreaker()** method to our Route Configuration for all the routes.

```java
@Bean
public RouterFunction<ServerResponse> productServiceRoute() {
   return GatewayRouterFunctions.route("product_service")
            .route(RequestPredicates.path("/api/product"),
HandlerFunctions.http("http://localhost:8080"))
            .filter(circuitBreaker("productServiceCircuitBreaker",
URI.create("forward:/fallbackRoute")))
            .build();
}

@Bean
public RouterFunction<ServerResponse> productServiceSwaggerRoute() {
    return GatewayRouterFunctions.route("product_service_swagger")
            .route(RequestPredicates.path("/aggregate/product-service/v3/api-docs"),
HandlerFunctions.http("http://localhost:8080"))
            .filter(circuitBreaker("productServiceSwaggerCircuitBreaker",
URI.create("forward:/fallbackRoute")))
            .filter(setPath("/api-docs"))
            .build();
}

@Bean
public RouterFunction<ServerResponse> orderServiceRoute() {
    return GatewayRouterFunctions.route("order_service")
            .route(RequestPredicates.path("/api/order"),
HandlerFunctions.http("http://localhost:8081"))
            .filter(circuitBreaker("orderServiceCircuitBreaker",
URI.create("forward:/fallbackRoute")))
            .build();
}

@Bean
public RouterFunction<ServerResponse> orderServiceSwaggerRoute() {
    return GatewayRouterFunctions.route("order_service_swagger")
            .route(RequestPredicates.path("/aggregate/order-service/v3/api-docs"),
HandlerFunctions.http("http://localhost:8081"))
            .filter(circuitBreaker("orderServiceSwaggerCircuitBreaker",
URI.create("forward:/fallbackRoute")))
            .filter(setPath("/api-docs"))
            .build();
}

@Bean
public RouterFunction<ServerResponse> inventoryServiceRoute() {
    return GatewayRouterFunctions.route("inventory_service")
            .route(RequestPredicates.path("/api/inventory"),
HandlerFunctions.http("http://localhost:8082"))
            .filter(circuitBreaker("inventoryServiceCircuitBreaker",
URI.create("forward:/fallbackRoute")))
            .build();
}

@Bean
public RouterFunction<ServerResponse> inventoryServiceSwaggerRoute() {
    return GatewayRouterFunctions.route("inventory_service_swagger")
            .route(RequestPredicates.path("/aggregate/inventory-service/v3/api-docs"),
HandlerFunctions.http("http://localhost:8082"))
            .filter(circuitBreaker("inventoryServiceSwaggerCircuitBreaker",
URI.create("forward:/fallbackRoute")))
            .filter(setPath("/api-docs"))
            .build();
}

@Bean
public RouterFunction<ServerResponse> fallbackRoute() {
    return route("fallbackRoute")
            .GET("/fallbackRoute", request ->
ServerResponse.status(HttpStatus.SERVICE_UNAVAILABLE).body("Service Unavailable, please
try again later"))
            .build();
}
```

You can see that the **circuitBreaker()** method is taking an ID which is a string and then a URL parameter which points to a fallback endpoint that will be displayed when the requests are blocked when the CircuitBreaker is **OPEN**

We have the **fallbackRoute()** bean that is defined as a fallback route at the path - **/fallbackRoute** that sends a HTTP 503 Service Unavailable response back to the client.

After adding this configuration for our routes, we have to now configure Resilience4J in our project, for that open application.properties file:

```
#Actuator Endpoints

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


#resilience4j properties

resilience4j.circuitbreaker.configs.default.registerHealthIndicator=true
resilience4j.circuitbreaker.configs.default.slidingWindowType=COUNT_BASED
resilience4j.circuitbreaker.configs.default.slidingWindowSize=10
resilience4j.circuitbreaker.configs.default.failureRateThreshold=50
resilience4j.circuitbreaker.configs.default.waitDurationInOpenState=5s
resilience4j.circuitbreaker.configs.default.permittedNumberOfCallsInHalfOpenState=3
resilience4j.circuitbreaker.configs.default.automaticTransitionFromOpenToHalfOpenEnabled=
true
resilience4j.circuitbreaker.configs.default.minimum-number-of-calls=5
```

```
resilience4j.circuitbreaker.configs.default.minimum-number-of-calls=5
```

if we are not specifying above property then it will wait till all 10 requests fails that we defined in above property(`slidingWindowSize=10`). it wont consider `failureRateThreshold=50` also.

The above properties make sure that Resilience4J is configured in our project.

Enable Circuit Breaker for Timeouts

We can enable Circuit Breaker to implement a timeout, when the remote service is taking a very long time to respond, for that all we have to do is add the following property:

```
resilience4j.timelimiter.configs.default.timeout-duration=3s
```

With this configuration, the circuit breaker will be OPEN, when the remote service is taking more than 3 seconds to send back the response.

Implement Retries

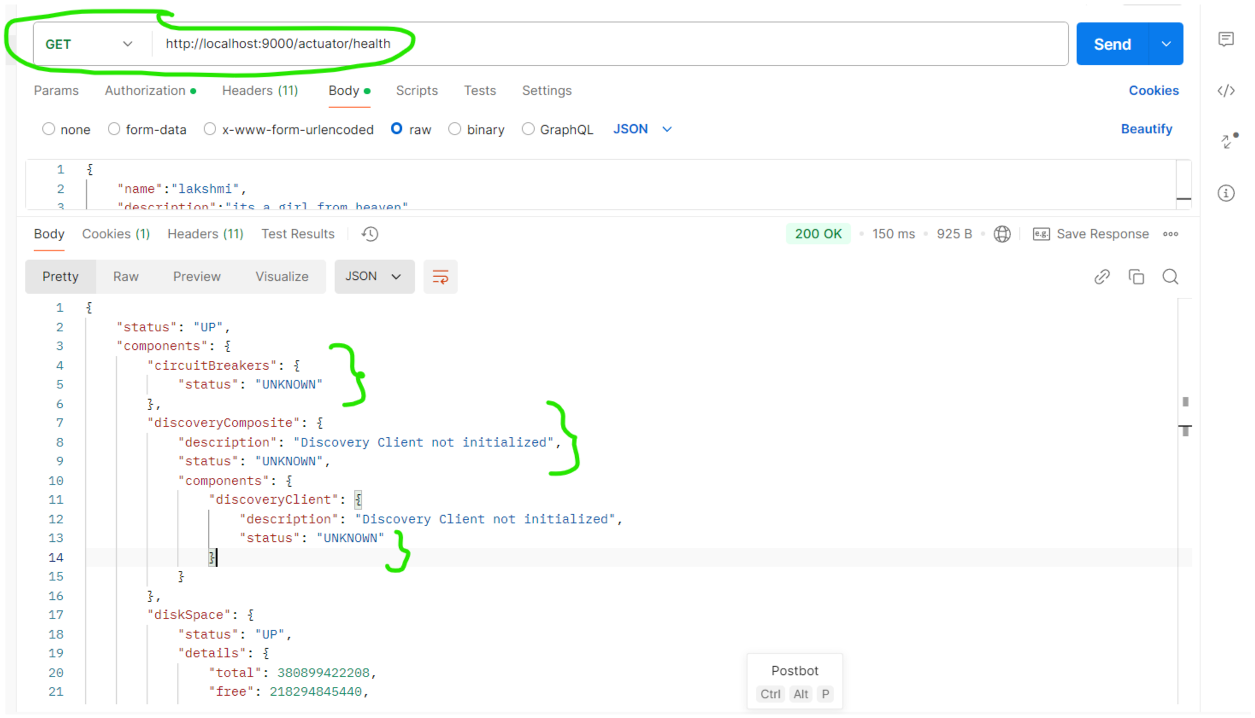
Sometimes, the service can be unavailable due to a small network issue (or) any other minor issue, in those cases, it's better to retry the call instead of directly activating the Circuit Breaker. For this reason, the Resilience4J library allows us to implement retries by adding the following configuration:

```
#Resilience4J Retry Properties
resilience4j.retry.configs.default.max-attempts=3
resilience4j.retry.configs.default.wait-duration=2s
```

The above configuration will retry for a maximum of 3 times, with a wait of 5 seconds in between the retries.

Actuator:

Just we need to add actuator dependency that we mentioned in abve pom.xml file. and after adding all properties in application.properties file, just run application and hit below end point.

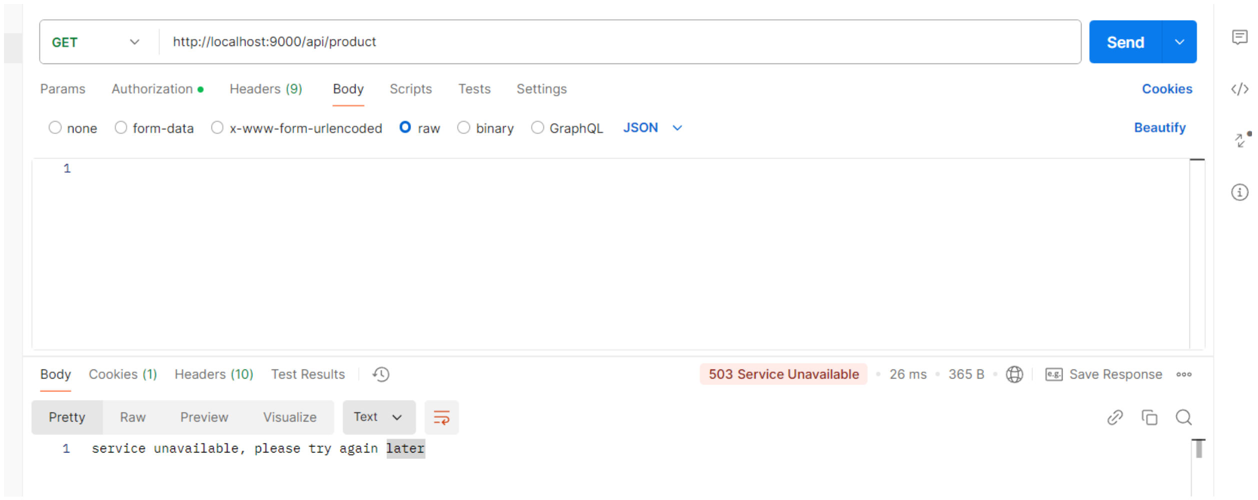


```
http://localhost:9000/actuator/health
```

```
{
    "name":"lakshmi",
    "description":"its a girl from heaven"
```

```
{
    "status": "UP",
    "components": {
        "circuitBreakers": {
            "status": "UNKNOWN"
        },
        "discoveryComposite": {
            "description": "Discovery Client not initialized",
            "status": "UNKNOWN",
            "components": {
                "discoveryClient": {
                    "description": "Discovery Client not initialized",
                    "status": "UNKNOWN"
                }
            }
        },
        "diskSpace": {
            "status": "UP",
            "details": {
                "total": 380899422208,
                "free": 218294845440,
```

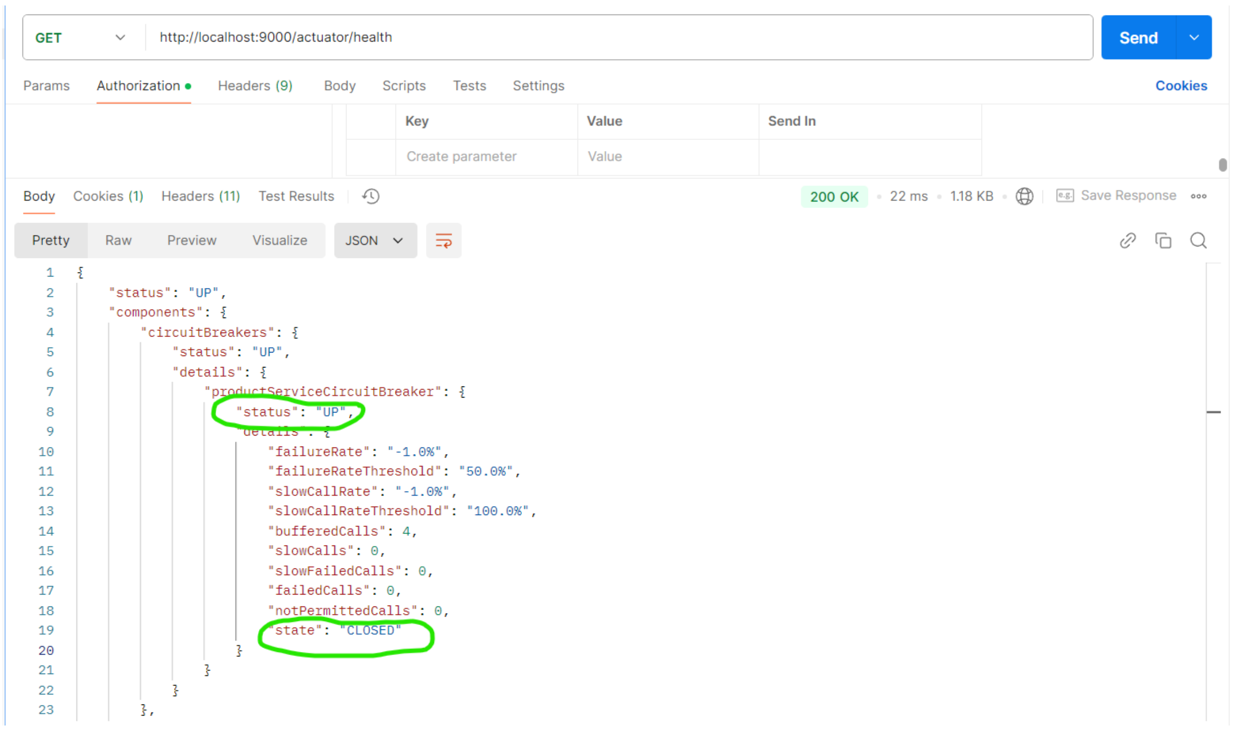
After that we stopped product service and called `/api/product` endpoint. we can see `service unavailable, please try again later` message in console.



```
http://localhost:9000/api/product
```

```
service unavailable, please try again later
```

After 3 to 5 failures, health will come into open state. as we mentioned in properties file 5s it will be in open state again it will go into Half-open state. in this state we will restart our product service and send request again. after 4 to 5 successfull response it will come into closed state.



http://localhost:9000/actuator/health

```
{
    "status": "UP",
    "components": {
        "circuitBreakers": {
            "status": "UP",
            "details": {
                "productServiceCircuitBreaker": {
                    "status": "UP",
                    "details": {
                        "failureRate": "-1.0%",
                        "failureRateThreshold": "50.0%",
                        "slowCallRate": "-1.0%",
                        "slowCallRateThreshold": "100.0%",
                        "bufferedCalls": 4,
                        "slowCalls": 0,
                        "slowFailedCalls": 0,
                        "failedCalls": 0,
                        "notPermittedCalls": 0,
                        "state": "CLOSED"
                    }
                }
            }
        },
```

we can implement time out also by adding thread.sleep method in product controller get all products method.

Implement Circuit Breaker in the Order Service

Now let's implement the Circuit Breaker also in the Order Service, as we are making a synchronous call to the inventory service in this service.

For that, I am going to add the below dependencies, in the **pom.xml** of the project:

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

After that, let's add the configuration for Resilience4J in the application.properties file, like below:

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

#Resilinece4j Properties
resilience4j.circuitbreaker.instances.inventory.registerHealthIndicator=true
resilience4j.circuitbreaker.instances.inventory.event-consumer-buffer-size=10
resilience4j.circuitbreaker.instances.inventory.slidingWindowType=COUNT_BASED
resilience4j.circuitbreaker.instances.inventory.slidingWindowSize=5
resilience4j.circuitbreaker.instances.inventory.failureRateThreshold=50
resilience4j.circuitbreaker.instances.inventory.waitDurationInOpenState=5s
resilience4j.circuitbreaker.instances.inventory.permittedNumberOfCallsInHalfOpenState=3
resilience4j.circuitbreaker.instances.inventory.automaticTransitionFromOpenToHalfOpenEnab
led=true
resilience4j.circuitbreaker.instances.inventory.minimum-number-of-calls=5

#Resilience4J Timeout Properties
resilience4j.timelimiter.instances.inventory.timeout-duration=3s

#Resilience4J Retry Properties
resilience4j.retry.instances.inventory.max-attempts=3
resilience4j.retry.instances.inventory.wait-duration=5s
```

After that to enable Circuit Breaker on the specific endpoints we can add the @CircuitBreaker annotation, similarly to enable retries, we can add the @Retry annotation respectively.

We can add these above annotations in the Inventory Client class, this is how the class looks like after adding the necessary annotations:

```
package com.techie.microservices.order.client;

import groovy.util.logging.Slf4j;
import io.github.resilience4j.circuitbreaker.annotation.CircuitBreaker;
import io.github.resilience4j.retry.annotation.Retry;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.service.annotation.GetExchange;

@Slf4j
public interface InventoryClient {
    Logger log = LoggerFactory.getLogger(InventoryClient.class);

    @GetExchange("/api/inventory")
    @CircuitBreaker(name = "inventory", fallbackMethod = "fallbackMethod")
    @Retry(name = "inventory")
    boolean isInStock(@RequestParam String skuCode, @RequestParam Integer quantity);

    default boolean fallbackMethod(String code, Integer quantity, Throwable throwable) {
        log.info("Cannot get inventory for skucode {}, failure reason: {}", code,
throwable.getMessage());
        return false;
    }
}
```

In the above class, you can notice that we defined a method called **fallbackMethod** that will be executed whenever the Circuit Breaker is **OPEN**.

To implement Timeout, we can configure the RestClient to have a connection and read time out through the requestFactory() method. This is how the RestClientConfig.java class looks like:

```
package com.techie.microservices.order.config;

import com.techie.microservices.order.client.InventoryClient;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.web.client.ClientHttpRequestFactories;
import org.springframework.boot.web.client.ClientHttpRequestFactorySettings;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.client.ClientHttpRequestFactory;
import org.springframework.http.client.HttpComponentsClientHttpRequestFactory;
import org.springframework.web.client.RestClient;
import org.springframework.web.client.support.RestClientAdapter;
import org.springframework.web.service.invoker.HttpServiceProxyFactory;

import java.time.Duration;

@Configuration
public class RestClientConfig {

    @Value("${inventory.url}")
    private String inventoryServiceUrl;
```

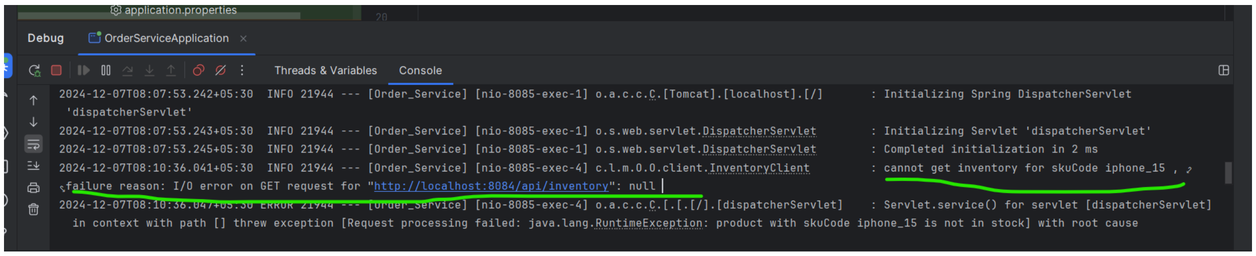
```java
    @Bean
    public InventoryClient inventoryClient() {
        RestClient restClient = RestClient.builder()
                .baseUrl(inventoryServiceUrl)
                .requestFactory(getClientRequestFactory())
                .build();
        var restClientAdapter = RestClientAdapter.create(restClient);
        var httpServiceProxyFactory =
HttpServiceProxyFactory.builderFor(restClientAdapter).build();
        return httpServiceProxyFactory.createClient(InventoryClient.class);
    }

    private ClientHttpRequestFactory getClientRequestFactory() {
        ClientHttpRequestFactorySettings clientHttpRequestFactorySettings =
ClientHttpRequestFactorySettings.DEFAULTS
                .withConnectTimeout(Duration.ofSeconds(3))
                .withReadTimeout(Duration.ofSeconds(3));
        return ClientHttpRequestFactories.get(clientHttpRequestFactorySettings);
    }
}
```

Testing the Circuit Breaker Pattern

To test the Circuit Breaker in the API Gateway, make sure that one of the services like Product, Order or Inventory Service is unavailable, and then call the corresponding service.

You should see an error - **Service Unavailable, please try again later** with the status **HTTP_503**

You can try the same thing also for the Order Service project, by stopping the Inventory Service. After running order services we can see below.



```
2024-12-07T08:07:53.242+05:30  INFO 21944 --- [Order_Service] [nio-8085-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring DispatcherServlet
 'dispatcherServlet'
2024-12-07T08:07:53.243+05:30  INFO 21944 --- [Order_Service] [nio-8085-exec-1] o.s.web.servlet.DispatcherServlet        : Initializing Servlet 'dispatcherServlet'
2024-12-07T08:07:53.245+05:30  INFO 21944 --- [Order_Service] [nio-8085-exec-1] o.s.web.servlet.DispatcherServlet        : Completed initialization in 2 ms
2024-12-07T08:10:36.041+05:30  INFO 21944 --- [Order_Service] [nio-8085-exec-4] c.l.m.0.0.client.InventoryClient         : cannot get inventory for skuCode iphone_15 ,
failure reason: I/O error on GET request for "http://localhost:8084/api/inventory": null
2024-12-07T08:10:36.047+05:30 ERROR 21944 --- [Order_Service] [nio-8085-exec-4] o.a.c.c.C.[.[.[/].[dispatcherServlet]    : Servlet.service() for servlet [dispatcherServlet]
 in context with path [] threw exception [Request processing failed: java.lang.RuntimeException: product with skuCode iphone_15 is not in stock] with root cause
```

To test the Timeout and Retry, we can introduce a slight delay by adding something like Thread.sleep() to simulate latency for our requests and you can observe that Circuit Breaker will be activated also in these cases.

Conclusion

In this blogpost, you learned about Circuit Breaker pattern, why and when to use it. We also learned how to enable the pattern using libraries like Resilience4J and Spring Cloud Circuit Breaker.

In the next part of the Spring Boot Microservice Tutorial series, we will learn how to implement asynchronous communication using Kafka.