**Resilience in Microservices: A Scenario**

In a microservices architecture, consider a scenario where the **Accounts microservice** relies on both the **Loans** and **Cards microservices**. If the **Cards microservice** experiences issues such as network latency, high traffic, or downtime, it may respond slowly or become unreachable. Since the **Accounts microservice** depends on responses from both services, it waits for the **Cards microservice**, causing delays in responding to the client.

This situation highlights the importance of **resilience in microservices**—the system's ability to handle disruptions, manage delays gracefully, and minimize impact on the user experience.

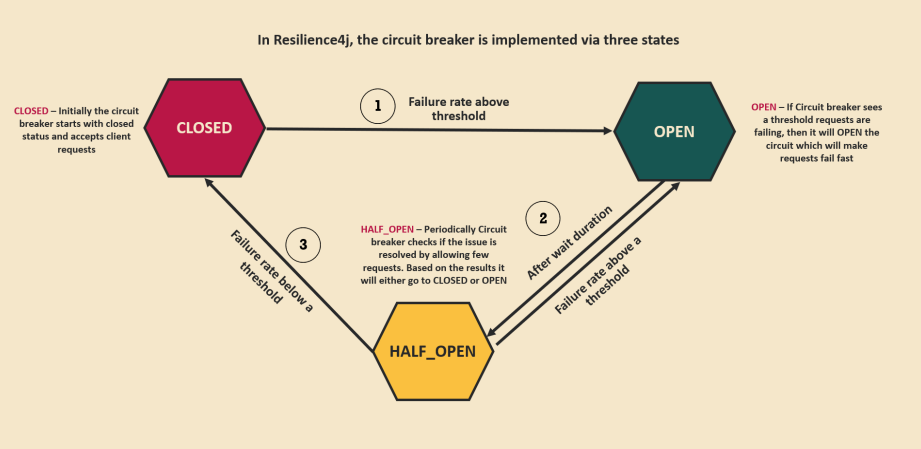
**What Are Resilient Microservices?**

Resilient microservices are designed to remain functional during network failures, service downtimes, or high loads, ensuring service availability and performance. A key pattern that enhances resilience is the Circuit Breaker Pattern.

**The Circuit Breaker Pattern**

Inspired by electrical circuit breakers that prevent damage by stopping excessive current flow, the **Circuit Breaker Pattern** in microservices acts as a safeguard. It stops repeated failed calls to a service by "tripping" after a defined threshold of failures. This prevents overloading the failing service and allows the system to handle issues more gracefully.

In microservices, the **Resilience4j library** is commonly used to implement circuit breakers, operating in three states: **CLOSED**, **OPEN**, and **HALF\_OPEN**.



**Circuit Breaker States and Transitions**

1. **CLOSED State**
   * All requests are allowed to pass through to the dependent service.
   * If the failure rate (e.g., timeouts or errors) exceeds the configured threshold, the circuit breaker transitions to the **OPEN** state.
2. **OPEN State**
   * Requests to the failing service are blocked.
   * This state is triggered when the failure rate crosses the threshold.
   * While in the **OPEN** state, requests are immediately rejected to prevent overloading the service.
   * After a predefined wait duration, the circuit breaker transitions to the **HALF\_OPEN** state.
3. **HALF\_OPEN State**
   * A limited number of requests are allowed to test if the service has recovered.
   * If these requests succeed and the failure rate drops below the threshold, the circuit breaker moves back to the **CLOSED** state.
   * If failures persist, it transitions back to the **OPEN** state.

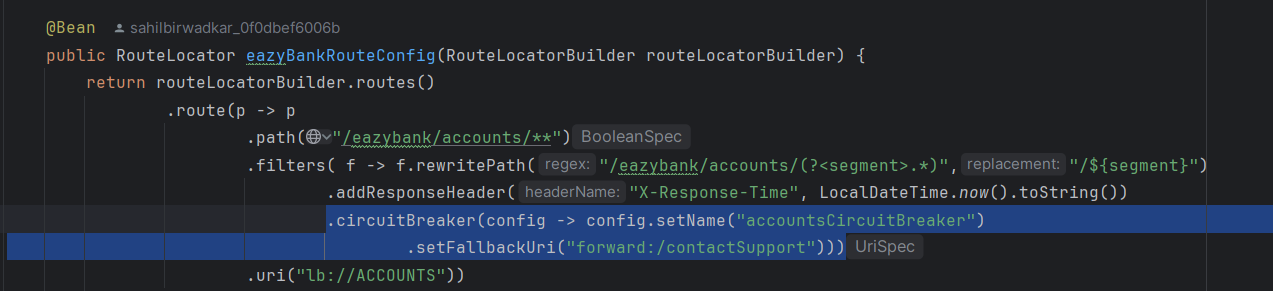
The **Circuit Breaker Pattern** enhances system stability and performance by controlling the flow of requests to problematic services.

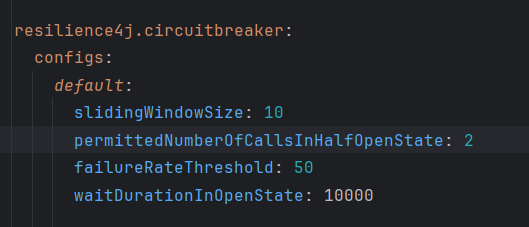
**Implementing a Circuit Breaker**

There are two common ways to implement circuit breakers in a microservices setup:

1. **Using an Edge Server**

If you’re using a gateway or edge server, circuit breakers can be configured centrally for all incoming requests.

* + Add the required dependency in the pom.xml of the **gateway server**:
  + Use the RouteLocator class to define routes, set circuit breaker names, and specify fallback URIs to handle failures gracefully.
  + Define parameters in the application.yml file:
* **Sliding Window Size**: Specifies the size of the window used to monitor metrics (e.g., failure rate).
* **Permitted Calls in Half-Open State**: Defines the number of requests allowed during the **HALF\_OPEN** state to test recovery.
* **Failure Threshold**: The percentage of failed requests that triggers the **OPEN** state.
* **Wait Duration in OPEN State**: The duration the circuit breaker remains **OPEN** before moving to the **HALF\_OPEN** state.

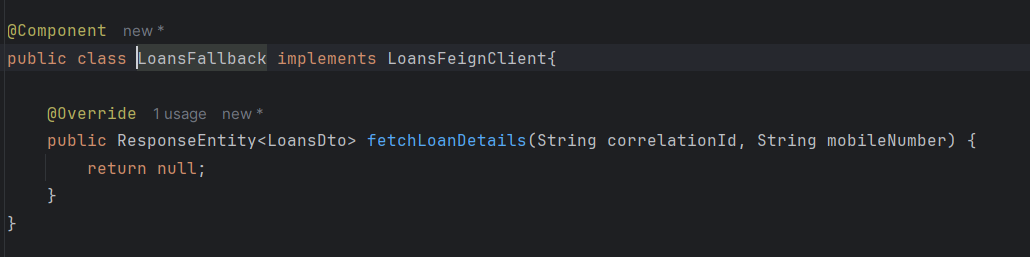


Sometimes, you need to handle failures directly within a service. Let’s take the **Accounts** microservice as an example.

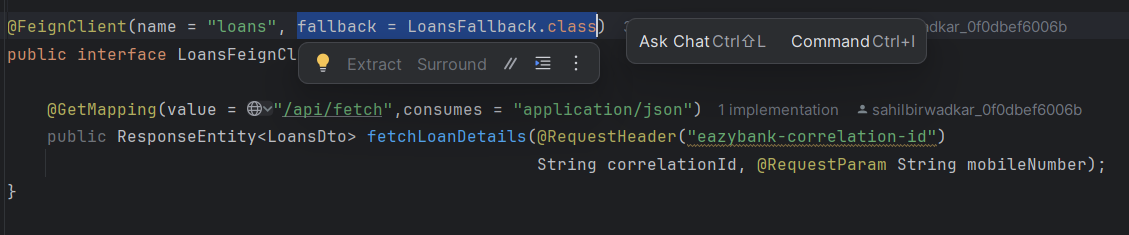
* **The Situation:**  
  The Accounts microservice needs to fetch details about loans and cards for a customer. To do this, it communicates with other microservices using **FeignClient**.
* **The Problem:**  
  If one of these services (e.g., Loans microservice) is slow or unavailable, it could block the Accounts service, causing delays or timeouts for the entire request
* **The Solution:**

Add a circuit breaker to the Loans service call:

* **Fallback Method:** Create a method that returns a default response (like null) when the Loans service fails.



* **Configure FeignClient:** In the FeignClient for Loans, link this fallback method so it automatically handles failures.



**Retry Pattern**

In microservices architecture, intermittent issues like network disruptions can cause temporary failures. These failures are often not permanent, and retrying the operation can help recover. This is where the **Retry Pattern** becomes crucial—it attempts the operation again to check if the service is back online.

Here are some key components and consideration if implementing the retry pattern in microservices:

1) Retry logic Decide how and when to retry failed operations. For instance, define the maximum number of attempts or specific scenarios where retries should occur.

2) Backoff strategy: Introduce a delay between retries to avoid overwhelming the system. A common approach is **exponential backoff**, where the delay increases progressively (e.g., 100ms → 200ms → 400ms).

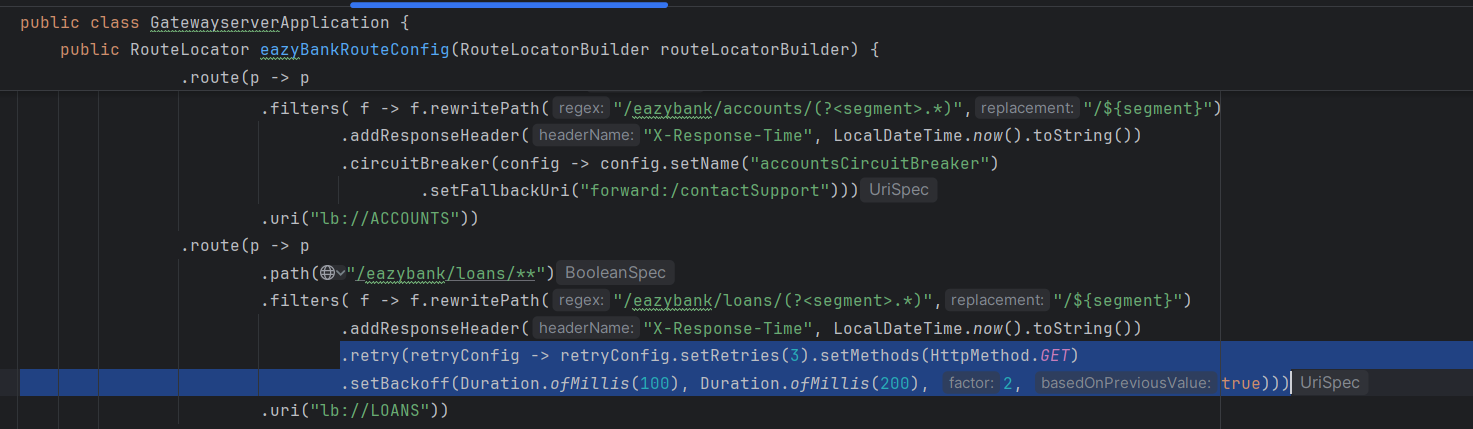
* Add randomness (jitter) to delay intervals to prevent synchronized retries from multiple clients, reducing the risk of server overload.

3) Circuit Breaker Integration: Combine the retry pattern with the circuit breaker pattern. If repeated retries fail, the circuit breaker opens, halting further attempts to protect system resources.

**Ways to Implement the Retry Pattern**

**1) At the Edge Server**

Centralize retry logic in the gateway or edge server:



* **Configure Routes with Retries:**

Use the RouteLocator class to define API routes and specify the number of retries for each route, as well as the HTTP methods for which retries are allowed.

* **Add Backoff plan**

Define the retry delays in the following format:

* **InitialInterval:** The initial delay before the first retry.
* **MaxInterval:** The maximum delay allowed between retries.
* **Multiplier:** The factor by which the delay increases after each attempt.
* **Jitter:** Randomizes retry intervals to reduce contention from synchronized requests.  
  Example: Instead of fixed intervals (e.g., 100ms, 200ms), the intervals could vary slightly, like 90ms or 210ms.

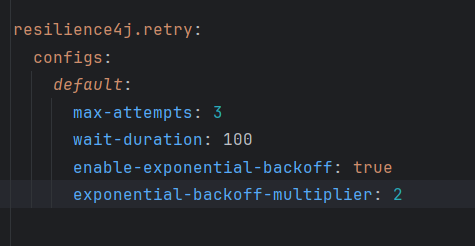
The backoff strategy is essential to prevent excessive and immediate retry attempts, which could further strain the server or network. It implements an **exponential backoff** approach:

**2)** **Within Individual Microservices (Implement retries directly in the microservice)**

**Step 1: Configure in application.yml:**

Define properties such as:

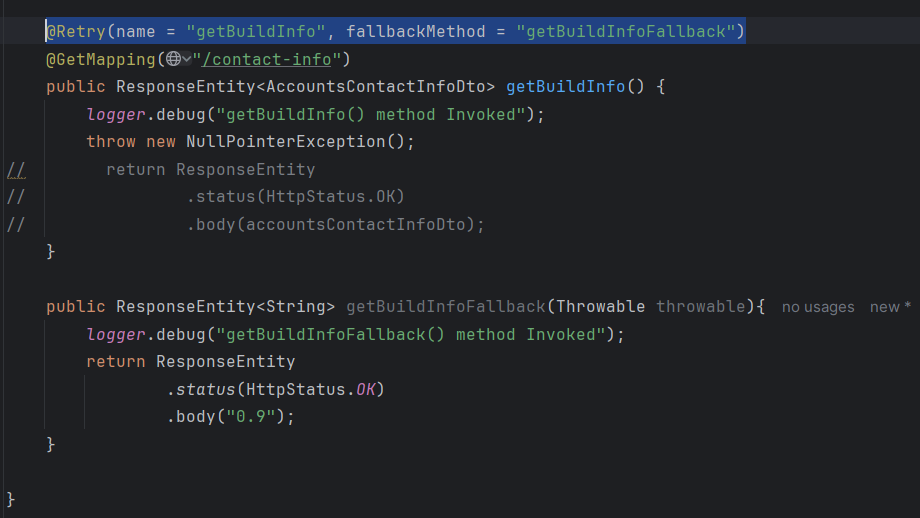
* Maximum retry attempts.
* Delay between retries (e.g., 500ms).
* Whether to use backoff strategy.
* Multiplier for exponential backoff.



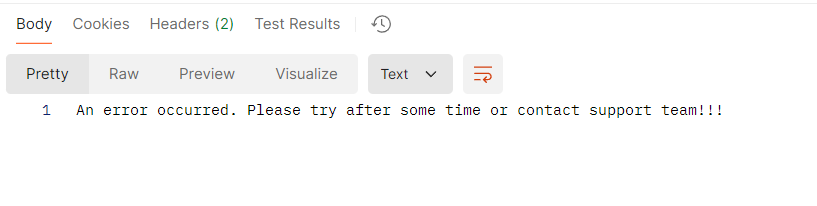
**Step 2: Annotate the Operation with @Retry**

Use the @Retry annotation to specify the retry logic:

* Set the name of the retry policy.
* Define a fallback method to handle cases where retries fail.

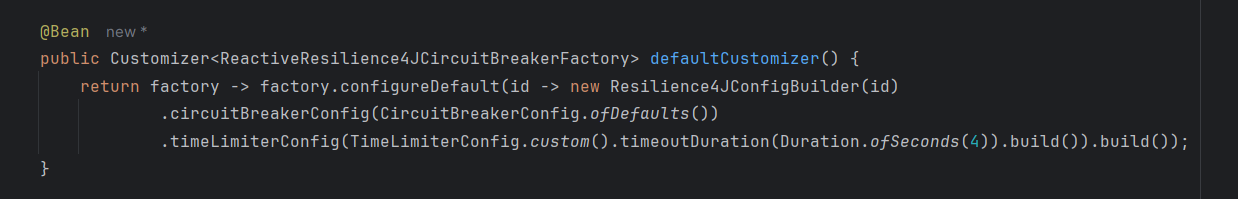


**Handling Circuit Breaker Integration**

**Scenario:** If you’ve implemented both retry and circuit breaker patterns in a service, the retry delay might conflict with the circuit breaker timeout. For example, if the retry interval (500ms) exceeds the circuit breaker timeout (100ms), the circuit breaker’s fallback method will be invoked before retries can complete.

**Solution:** Customize the circuit breaker configuration to account for retry behavior:

* **Use a Customizer in the Gateway Server:**  
  Create a @Bean that customizes the configuration of circuit breakers, aligning the timeouts with your retry logic.



**@Bean annotation:**

* Marks the method as a bean producer, meaning Spring will invoke this method and register its return value as a bean in the application context.

**Customize Interface**

* The method returns a Customizer for ReactiveResilience4JCircuitBreakerFactory.
* A Customizer is used to modify or enhance the default configuration of a component. In this case, it customizes the factory used to create **Reactive Circuit Breakers**.

**ReactiveResilience4JCircuitBreakerFactory**

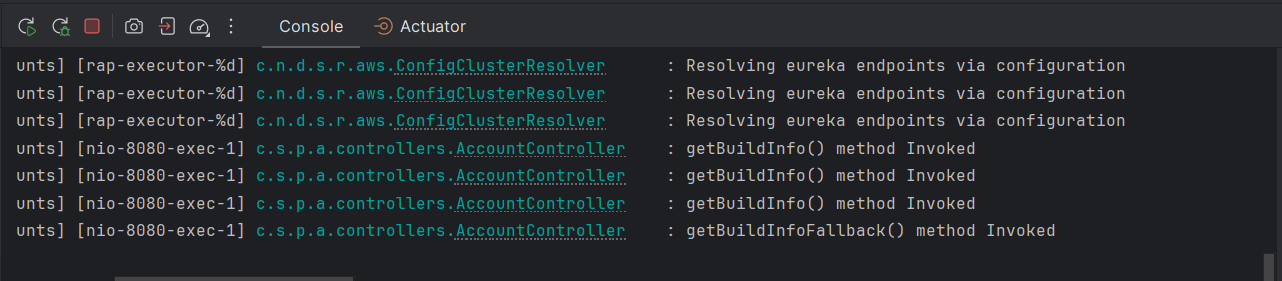
* A factory for creating reactive circuit breakers specifically for applications using **Reactive Streams** (e.g., WebFlux).
* Circuit breakers prevent systems from performing operations that are likely to fail, thereby avoiding unnecessary resource consumption.

**Factory Configuration**

* The lambda function factory -> factory.configureDefault(...) sets a **default configuration** for all circuit breakers created by the factory.

**Inside the Lambda**

* **new Resilience4JConfigBuilder(id):** Initializes a configuration builder for a specific circuit breaker identified by id.
* **circuitBreakerConfig(CircuitBreakerConfig.ofDefaults()):**
* Applies a default circuit breaker configuration.
* This configuration includes sensible defaults for properties like failure rate threshold, slow call rate threshold, etc.
* **timeLimiterConfig(...):** Sets up a **time limiter** configuration for the circuit breaker.
* **TimeLimiterConfig.custom():** Creates a custom configuration for the time limiter.
* **timeoutDuration(Duration.ofSeconds(4))**
* Sets the timeout for operations wrapped by the circuit breaker to **4 seconds**.
* If an operation takes longer than this, it will time out.



Now, suppose we want our microservice to avoid retrying an operation when a **NullPointerException** occurs. We can specify this behavior directly in the application.yml file of the respective microservice.

Similarly, we can explicitly configure the microservice to retry operations only when specific exceptions occur, tailoring the retry logic to handle particular failure scenarios.



Also, as our microservice is doing retry, its creating multiple correlation ID at client side. To avoid that we have to add if condition in ResponseTraceFilter of gateway server.

