The **TimeLimiter** in Resilience4j is used to **set a maximum execution duration** for a method (usually asynchronous).  
If the method doesn't complete within that time, it will **timeout and trigger a fallback**.

👉 Think of it as:

⏱ “If the service doesn’t respond in 2 seconds, stop waiting and return a fallback response.”

**🧰 1️⃣ Add Dependency**

If not already added:

<dependency>

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

<artifactId>resilience4j-spring-boot3</artifactId>

</dependency>

**⚙️ 2️⃣ Configure in application.yml**

Here’s a sample config for **currencyConversionService**:

resilience4j:

timelimiter:

instances:

currencyConversionService:

timeoutDuration: 2s # ⏱ Max time to wait for the call

cancelRunningFuture: true # Cancels the running task if it times out

**📝 Description**

| **Property** | **Description** |
| --- | --- |
| timeoutDuration | Max time allowed for the method to complete. If exceeded → fallback triggers |
| cancelRunningFuture | If true, cancels the running task after timeout. |

**🧠 3️⃣ Annotate Your Method**

**TimeLimiter works with asynchronous methods** → typically with CompletableFuture or Mono/Flux in reactive apps.

Here’s an example using CompletableFuture:

import io.github.resilience4j.timelimiter.annotation.TimeLimiter;

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

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

import java.util.concurrent.CompletableFuture;

@RestController

public class CurrencyConversionController {

@GetMapping("/convert")

@TimeLimiter(name = "currencyConversionService", fallbackMethod = "timeLimiterFallback")

public CompletableFuture<String> convertCurrency() {

return CompletableFuture.supplyAsync(() -> {

try {

// Simulate a slow service

Thread.sleep(5000);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

return "Conversion completed successfully!";

});

}

public CompletableFuture<String> timeLimiterFallback(Throwable t) {

return CompletableFuture.completedFuture("⚠ Request timed out. Please try again later.");

}

}

📝 In this example:

* The method sleeps for 5 seconds.
* The TimeLimiter is set to 2 seconds.
* After 2 seconds → timeout → fallback is executed.

**🧪 4️⃣ Test It**

Hit the endpoint:

GET http://localhost:8100/convert

👉 You should get the **fallback** response after 2 seconds.

**🧠 5️⃣ Combine with CircuitBreaker**

Often, TimeLimiter is used **with CircuitBreaker**, especially for Feign or RestTemplate calls:

@CircuitBreaker(name = "currencyConversionService", fallbackMethod = "fallback")

@TimeLimiter(name = "currencyConversionService", fallbackMethod = "fallback")

public CompletableFuture<String> getExchangeRate() {

return CompletableFuture.supplyAsync(() -> {

// Slow or unreliable call

return restTemplate.getForObject("http://currency-exchange-service/exchange", String.class);

});

}

👉 Here:

* TimeLimiter ensures the call doesn’t hang too long.
* CircuitBreaker opens if timeouts/failures happen repeatedly.

**📊 6️⃣ Monitor via Actuator**

If Spring Boot Actuator is enabled, check:

GET http://localhost:8100/actuator/timelimiters

GET http://localhost:8100/actuator/timelimiter-events

You’ll see events like timeouts, successes, and fallbacks.

**✅ Summary Table**

| **Feature** | **TimeLimiter ⏱** |
| --- | --- |
| **Main Goal** | Limit max execution time for a call |
| **When to Use** | When remote services can **hang / delay** |
| **Works Best With** | CircuitBreaker + asynchronous methods |
| **Fallback** | Triggered if timeout occurs |

**🚀 Real-world Usage Example**

In a **currency-conversion** microservice calling a **currency-exchange** service, you can use:

* **CircuitBreaker** → avoid cascading failures
* **TimeLimiter** → avoid waiting too long for a slow downstream
* **Bulkhead** → limit concurrent requests
* **RateLimiter** → prevent abuse

👉 This combo gives you **full fault tolerance** 🧠⚡