Quick Guide to Spring Cloud Circuit Breaker

Last modified: September 15, 2022

Written by: Catalin Burcea

Spring Cloud

1. Overview

In this tutorial, **we'll introduce the Spring Cloud Circuit Breaker project** and learn how we can make use of it.

First, we're going to see what the Spring Cloud Circuit Breaker offers in addition to existing circuit breaker implementations. Next, v

Note that we've got more information about what a circuit breaker is and how they work in Introduction to Hystrix, Spring Cloud Ne

2. Spring Cloud Circuit Breaker

Until recently, Spring Cloud only provided us one way to add circuit breakers in our applications. This was through the use of Netfl

The Spring Cloud Netflix project is really just an annotation-based wrapper library around Hystrix. Therefore, these two libraries ar

The Spring Cloud Circuit Breaker project solves this. **It provides an abstraction layer across different circuit breaker implementa**

For our examples, **we'll focus only on the Resilience4J implementation.** However, these techniques can be used for other plugin

3. Auto Configuration

In order to use a specific circuit breaker implementations in our application, we need to add the appropriate Spring starter. In

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
    <version>1.0.2.RELEASE</version>
</dependency>
```

The auto-configuration mechanism configures the necessary circuit breaker beans if it sees one of the starters in the classpath

If we wanted to disable the Resilience4J auto-configuration, we could set the *spring.cloud.circuitbreaker.resilience4j.enabled* prop

4. A Simple Circuit Breaker Example

Let's create a web application using Spring Boot to allow us to explore how the Spring Cloud Circuit Breaker library works.

We'll build a simple web service returning a list of albums. Let's suppose the raw list is provided by a third-party service. For simpl

```
https://jsonplaceholder.typicode.com/albums
```

4.1. Create a Circuit Breaker

Let's create our first circuit breaker. We'll start by injecting an instance of the *CircuitBreakerFactory* bean:

```
@Service
public class AlbumService {

    @Autowired
    private CircuitBreakerFactory circuitBreakerFactory;

    //...

}
```

Now, we can easily create a circuit breaker using the *CircuitBreakerFactory#create* method. It takes the circuit breaker identifier as

```
CircuitBreaker circuitBreaker = circuitBreakerFactory.create("circuitbreaker");
```

4.2. Wrap a Task in a Circuit Breaker

In order to wrap and run a task protected by the circuit breaker, we need to call the r*un* method which takes a *Supplier* as an argu

```
public String getAlbumList() {
    CircuitBreaker circuitBreaker = circuitBreakerFactory.create("circuitbreaker");
    String url = "https://jsonplaceholder.typicode.com/albums";

    return circuitBreaker.run(() -> restTemplate.getForObject(url, String.class));
}
```

The circuit breaker runs our method for us and provides fault tolerance.

Sometimes, our external service could take too long to respond, throw an unexpected exception or the external service or host dc

```
public String getAlbumList() {
    CircuitBreaker circuitBreaker = circuitBreakerFactory.create("circuitbreaker");
    String url = "http://localhost:1234/not-real";

    return circuitBreaker.run(() -> restTemplate.getForObject(url, String.class),
      throwable -> getDefaultAlbumList());
}
```

The lambda for the fallback receives the *Throwable* as an input, describing the error. This means **we can provide different fallbac**

In this case, we won't take the exception into account. We'll just return a cached list of albums.

If the external call ends with an exception and no fallback is provided, a *NoFallbackAvailableException* is thrown by Spring.

4.3. Build a Controller

Now, let's finish our example and create a simple controller that calls the service methods and presents the results through a brov

```
@RestController
public class Controller {

    @Autowired
```

```
    private Service service;

    @GetMapping("/albums")
    public String albums() {
        return service.getAlbumList();
    }

}
```

Finally, let's call the REST service and see the results:

```
[GET] http://localhost:8080/albums
```

5. Global Custom Configuration

Usually, the default configuration is not enough. For this reason, we need to create circuit breakers with custom configurations bas

In order to override the default configuration, we need to specify our own beans and properties in a *@Configuration* class.

Here, we're going to define a global configuration for all circuit breakers. For this reason, **we need to define a *Customizer<CircuitI***

First, we'll define circuit breaker and time limiter configuration classes as per the Resilience4j tutorial:

```
CircuitBreakerConfig circuitBreakerConfig = CircuitBreakerConfig.custom()
  .failureRateThreshold(50)
  .waitDurationInOpenState(Duration.ofMillis(1000))
  .slidingWindowSize(2)
  .build();
TimeLimiterConfig timeLimiterConfig = TimeLimiterConfig.custom()
  .timeoutDuration(Duration.ofSeconds(4))
  .build();
```

Next, let's embed the configuration in a *Customizer* bean by using the *Resilience4JCircuitBreakerFactory.configureDefault* method

```
@Configuration
public class Resilience4JConfiguration {
    @Bean
    public Customizer<Resilience4JCircuitBreakerFactory> globalCustomConfiguration() {

        // the circuitBreakerConfig and timeLimiterConfig objects

        return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)
          .timeLimiterConfig(timeLimiterConfig)
          .circuitBreakerConfig(circuitBreakerConfig)
          .build());
    }
}
```

6. Specific Custom Configuration

Of course, **we can have multiple circuit breakers in our application.** Therefore, in some cases, we need a specific configuration f

Similarly, we can define one or more *Customizer* beans. Then, we can provide a different configuration for each one by using the *A*

```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> specificCustomConfiguration1() {

    // the circuitBreakerConfig and timeLimiterConfig objects

    return factory -> factory.configure(builder -> builder.circuitBreakerConfig(circuitBreakerConfig)
      .timeLimiterConfig(timeLimiterConfig).build(), "circuitBreaker");
}
```

Here we provide a second parameter, the id of the circuit breaker we're configuring.

We can also set up multiple circuit breakers with the same configuration by providing a list of circuit breaker ids to the same meth

```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> specificCustomConfiguration2() {

    // the circuitBreakerConfig and timeLimiterConfig objects

    return factory -> factory.configure(builder -> builder.circuitBreakerConfig(circuitBreakerConfig)
      .timeLimiterConfig(timeLimiterConfig).build(),
        "circuitBreaker1", "circuitBreaker2", "circuitBreaker3");
}
```

7. Alternative Implementations

We've seen how to use the *Resilience4j* implementation to create one or more circuit breakers with Spring Cloud Circuit Breaker.

However, there are other implementations supported by Spring Cloud Circuit Breaker that we can leverage in our application:

- **Hystrix**
- **Sentinel**
- **Spring Retry**

It's worth mentioning that we can mix and match different circuit breaker implementations in our application. We're not just lim

The above libraries have more capabilities than we've explored here. However, Spring Cloud Circuit Breaker is an abstraction over

8. Conclusion

In this article, we discovered the Spring Cloud Circuit Breaker project.

First, we learned what the Spring Cloud Circuit Breaker is, and how it allows us to add circuit breakers to our application.

Next, we leveraged the Spring Boot auto-configuration mechanism in order to show how to define and integrate circuit breakers. A

Finally, we learned to configure all circuit breakers together, as well as individually.

As always, the source code for this tutorial is available over on GitHub.