# Circuit Breaker

Spring Cloud Circuit breaker provides an abstraction across different circuit breaker implementations

# Introduction

- Spring Cloud Circuit Breaker provides a consistent API to use in your applications allowing you the developer to choose the circuit breaker implementation that best fits your needs for your app
- Netfix Hystrix
- Resilience4J
- Sentinel
- Spring Retry

# How to implement?

- CircuitBreakerFactory API is used to implement circuit breaker.
- By including a Spring Cloud Circuit Breaker starter on your classpath a bean implementing this API will automatically be created for you.
- **CircuitBreakerFactory.create** API will create an instance of a class called CircuitBreaker. The run method takes a Supplier and a Function.
- The Supplier is the code that you are going to wrap in a circuit breaker. The Function is the fallback that will be executed if the circuit breaker is tripped.
- The function will be passed the Throwable that caused the fallback to be triggered. You can optionally exclude the fallback if you do not want to provide one

# Example

```
@Service
public static class DemoControllerService {
    private RestTemplate rest;
    private CircuitBreakerFactory cbFactory;
    public DemoControllerService(RestTemplate rest, CircuitBreakerFactory cbFactory)
    {
        this.rest = rest;
        this.cbFactory = cbFactory;
    }
    public String slow() {
         return cbFactory.create("slow").run(() -> rest.getForObject("/slow", String.class),
throwable -> "fallback");
     }
}
```

# Circuit Breakers In Reactive Code

```
@Service
public static class DemoControllerService {
    private ReactiveCircuitBreakerFactory cbFactory;
    private WebClient webClient;
    public DemoControllerService(WebClient webClient, ReactiveCircuitBreakerFactory
cbFactory) {
        this.webClient = webClient;
        this.cbFactory = cbFactory;
    }
    public Mono<String> slow() {
        return webClient.get().uri("/slow").retrieve().bodyToMono(String.class).transform(
it -> cbFactory.create("slow").run(it, throwable -> return Mono.just("fallback")));
     }
}
```

# Spring Retry

- Spring Retry provides an ability to automatically re-invoke a failed operation. This is helpful where the errors may be transient (like a momentary network glitch

```
@Configuration
@EnableRetry
public class AppConfig { ... }
```

# Spring Retry

```
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
    <version>1.2.5.RELEASE</version>
</dependency>
```

We also need to add Spring AOP into our project:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
```

# @Retryable Without Recovery

- **@Retryable** annotation to add retry functionality to methods:

```
@Service
public interface MyService {
    @Retryable(value = RuntimeException.class)
    void retryService(String sql);
}
```

- The retry is attempted when a RuntimeException is thrown. Per @Retryable‘s default behavior, the retry may happen up to three times, with a delay of one second between retries.

# @Retryable and @Recover

```
@Service
public interface MyService {
    @Retryable(value = SQLException.class)
    void retryServiceWithRecovery(String sql) throws SQLException;
    @Recover
    void recover(SQLException e, String sql);
}
```

- The retry is attempted when an SQLException is thrown. The @Recover annotation defines a separate recovery method when a @Retryable method fails with a specified exception

# Customizing @Retryable's Behavior

To customize a retry's behavior, we can use the parameters maxAttempts and backoff:

```
@Service
public interface MyService {
    @Retryable( value = SQLException.class,
      maxAttempts = 2, backoff = @Backoff(delay = 100))
    void retryServiceWithCustomization(String sql) throws SQLException;
}
```

There will be up to two attempts and a delay of 100 milliseconds.

# Using Spring Properties

We can also use properties in the @Retryable annotation. To externalize the values of delay and maxAttempts into a properties file. First, let's define the properties in a file called **retryConfig.properties**:

```
retry.maxAttempts=2
retry.maxDelay=100
```

We then instruct our **@Configuration** class to load this file:

```
@PropertySource("classpath:retryConfig.properties")
public class AppConfig { ... }
```

# Using Spring Properties

we can inject the values of retry.maxAttempts and retry.maxDelay in our @Retryable definition:

```
@Service
public interface MyService {
  @Retryable( value = SQLException.class,
                    maxAttemptsExpression = "${retry.maxAttempts}",
                    backoff = @Backoff(delayExpression = "${retry.maxDelay}"))
  void retryServiceWithExternalizedConfiguration(String sql) throws SQLException;
}
```

Please note that we are now using **maxAttemptsExpression** and **delayExpression** instead of **maxAttempts** and delay.

# RetryTemplate

```
public interface RetryOperations {
    <T> T execute(RetryCallback<T> retryCallback) throws Exception;
    ...
}
```

The **RetryCallback**, which is a parameter of the **execute()**, is an interface that allows insertion of business logic that needs to be retried upon failure:

```
public interface RetryCallback<T> {
    T doWithRetry(RetryContext context) throws Throwable;
}
```

# RetryTemplate Configuration

```
@Configuration
public class AppConfig {
    //...
    @Bean
    public RetryTemplate retryTemplate() {
        RetryTemplate retryTemplate = new RetryTemplate();
        FixedBackOffPolicy fixedBackOffPolicy = new FixedBackOffPolicy();
        fixedBackOffPolicy.setBackOffPeriod(2000l);
        retryTemplate.setBackOffPolicy(fixedBackOffPolicy);
        SimpleRetryPolicy retryPolicy = new SimpleRetryPolicy();
        retryPolicy.setMaxAttempts(2);
        retryTemplate.setRetryPolicy(retryPolicy);
        return retryTemplate;
    }
}
```

# RetryTemplate Configuration

- The RetryPolicy determines when an operation should be retried.
- A SimpleRetryPolicy is used to retry a fixed number of times. On the other hand, the BackOffPolicy is used to control backoff between retry attempts.
- Finally, a FixedBackOffPolicy pauses for a fixed period of time before continuing.

# Using the RetryTemplate

To run code with retry handling, we can call the retryTemplate.execute() method:

```
retryTemplate.execute(new RetryCallback<Void, RuntimeException>() {
    @Override
    public Void doWithRetry(RetryContext arg0) {
        myService.templateRetryService();
        ...
    }
});
```

Instead of an anonymous class, we can use a lambda expression:

```
retryTemplate.execute(arg0 -> {
    myService.templateRetryService();
    return null;
});
```

# Adding Callbacks

```
public class DefaultListenerSupport extends RetryListenerSupport {
    @Override
    public <T, E extends Throwable> void close(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {
        logger.info("onClose);
        ...
        super.close(context, callback, throwable);
    }
    @Override
    public <T, E extends Throwable> void onError(RetryContext context, RetryCallback<T, E> callback, Throwable throwable) {
        logger.info("onError");
        ...
        super.onError(context, callback, throwable);
    }
    @Override
    public <T, E extends Throwable> boolean open(RetryContext context, RetryCallback<T, E> callback) {
        logger.info("onOpen);
        ...
        return super.open(context, callback);
    }
}
```

# Registering the Listener

register our listener (DefaultListenerSupport) to our RetryTemplate bean:

```
@Configuration
public class AppConfig {
    ...
    @Bean
    public RetryTemplate retryTemplate() {
        RetryTemplate retryTemplate = new RetryTemplate();
        ...
        retryTemplate.registerListener(new DefaultListenerSupport());
        return retryTemplate;
    }
}
```

# Testing the Results

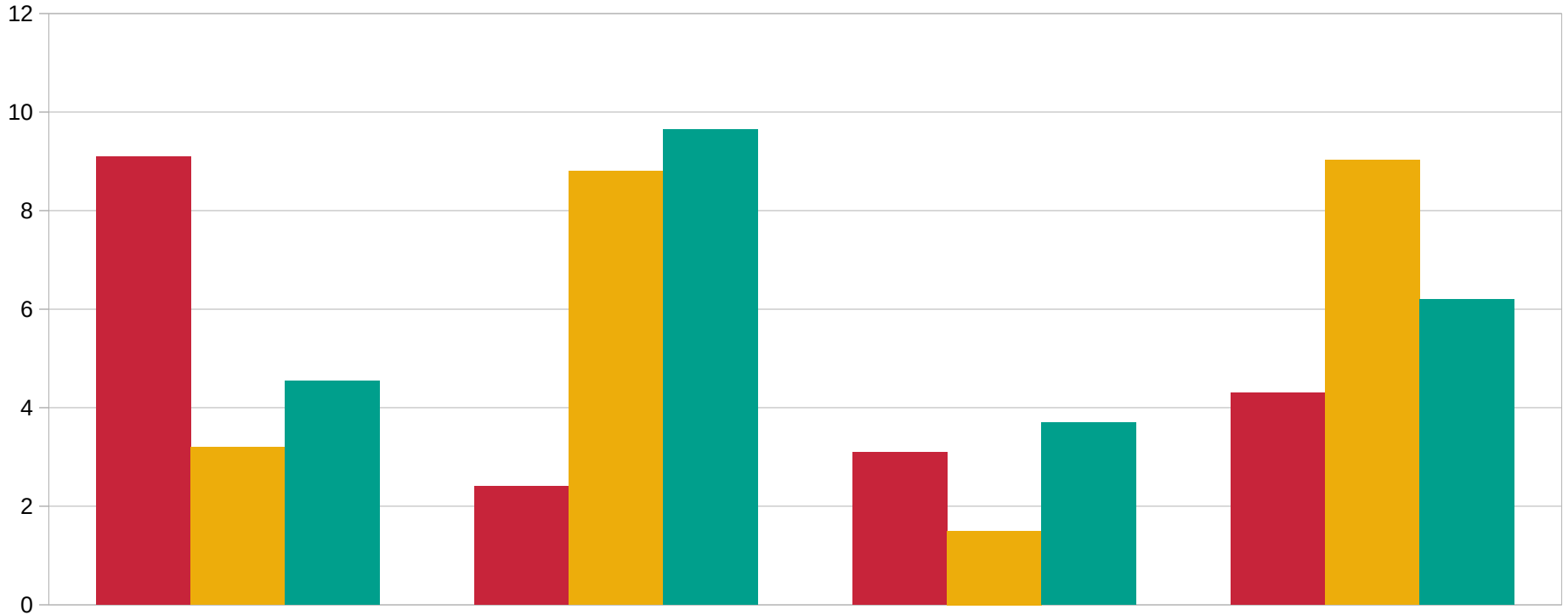
```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration( classes = AppConfig.class, loader = AnnotationConfigContextLoader.class)
public class SpringRetryIntegrationTest {

    @Autowired
    private MyService myService;

    @Autowired
    private RetryTemplate retryTemplate;

    @Test(expected = RuntimeException.class)
    public void givenTemplateRetryService_whenCallWithException_thenRetry() {
        retryTemplate.execute(arg0 -> {
            myService.templateRetryService();
            return null;
        });
    }
}
```

| | | | | |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |


12
10
8
6
4
2
0