Retry functionality in a reactive programming context

Reactive programming can be used to solve a lot of different use cases. For instance, reactive programming can be really helpful for cases where timing is an issue. An example of such a case is retry logic with delay or backoff functionality.

Lets have a look at some different implementations using Project Reactor (https://projectreactor.io) for Java. The sourcecode of the examples can be found at this repository.

Using Reactor to retry with exponentional backoff

In the next examples we will be calling the callAPI function as shown below. This function simulates throwing an exception while calling an API.

```
private Mono<String> callAPI() {
        return Mono.just("").flatMap(v -> {
            System.out.println("API call ");
            return api.monoWithException();
        });
    }
```

In our first attempt we use the reactor operator retryWhen. This operator can be used to resubscribe to the publisher in case of an exception. As a parameter it takes a function that allows us to conditionaly retry. The following marble diagram (from project reactor's javadoc) describes this behavior:

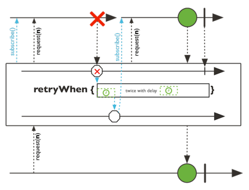

Figure 1: retryWhen

```
callAPI()
        .retryWhen(errors -> errors
                .zipWith(Flux.range(1, 4), (n, i) -> i)
                .flatMap(error -> Mono.delay(ofMillis(10))));
```

We combine each error with a entry from the range, using zipWith and than delay the emit using a flatMap:

```
zipWith(error1,1).flatMap(delay)     // resubscribes after 10ms
zipWith(error2,2).flatMap(delay)     // resubscribes after 10ms
zipWith(error3,3).flatMap(delay)    // resubscribes after 10ms
zipWith(error4,4).flatMap(delay)    // completes
```

After the first try and 4 retries, the producer completes when the api call keeps producing errors. In this example the delay time is constant, with a little change we can make it exponential to implement the exponential backoff pattern.

```
callAPI()
        .retryWhen(errors -> errors
                .zipWith(Flux.range(1, 4), (n, i) -> i)
                .flatMap(error -> Mono.delay(ofMillis((int) Math.pow(10, error)))));
```

The downside of this approach is that after the retries are done and the api call keeps producing errors, the operation terminates without an error. Ideally, you want the error to be propagated to be able to handle it.

Existing retry logic in reactor and reactor-extras

The reactor core library has a retryBackoff method, which takes as arguments the maximum number of retries, and initial backoff delay and a maximum backoff delay. This retry uses a randomized exponentional backoff strategy. If the callAPI function still gives an error after the specified number of retries, the retryBackoff returns an IllegalStateException. This allows us to log the exception, call exception logic and/or return a default value.

```
callAPI().retryBackoff(5, ofMillis(10), ofMillis(1000))
```

For our final example we make use of the reactor-extra library. This library consists additional operators. It also allows to create a Retry object which can be passed into the retryWhen() function. We use Retry.any() to retry all exception types. There are also functions to conditionally select when to retry (anyOf, allBut, onlyIf). The exponentialBackoff operator can be used to construct an exponential backoff strategy (to emulate a random backoff like the previouse example use exponentialBackoffWithJitter):

```
callAPI().retryWhen(any().exponentialBackoff(ofMillis(10), ofMillis(1000)).retryMax(5))
```

In case there are still errors after we are done retrying, a RetryExhaustedException is thrown. As an alternative to the exponentionalBackoff, reactor-extra also allows us to create a scheduled retry with withBackoffScheduler, a fixedBackoff or a randomBackoff.

Conclusion

Project Reactor has several expressive solutions to construct retry logic. This allows us to have great control over how and when we retry certain pieces of logic without having to write much code. When you don't want to use an extra library, the core reactor library can be used to create a simpel retry statement. Additionaly the reactor-extra library gives us several different retry and backoff strategies.