Spring WebFlux: Error handling in Reactive Streams

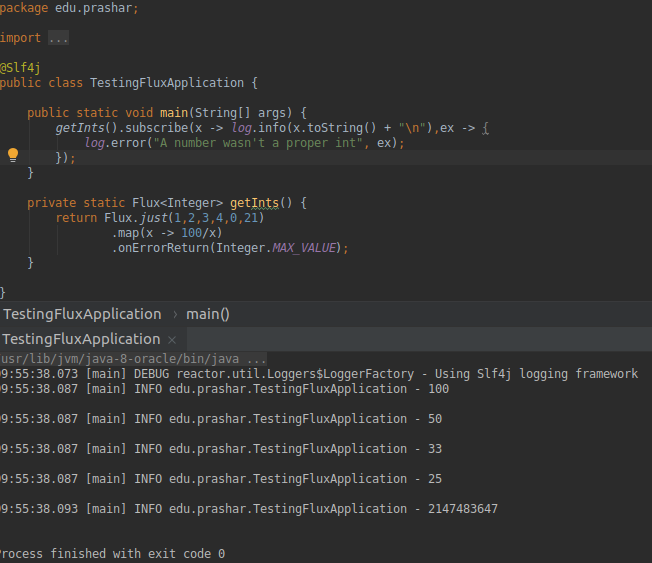
Error handling is one of the ways we ensure we are gracefully handling our failures. While working with streams of data, the Reactive Library [Reactor](https://projectreactor.io/) provides us with multiple ways of dealing with situations where we need to handle our errors. In this blog, I wish to discuss a few of them with you.

In [streams](https://blog.knoldus.com/spring-reactive-programming-in-java/), errors are terminal events. This means that from the point we encounter an error, our stream is not processed by the designated operator. On the contrary, it gets propagated to the subscription, looking for a path defined to follow in case of an error. If not defined, by default we get an UnsupportedOperatorException.

**Error Handling**

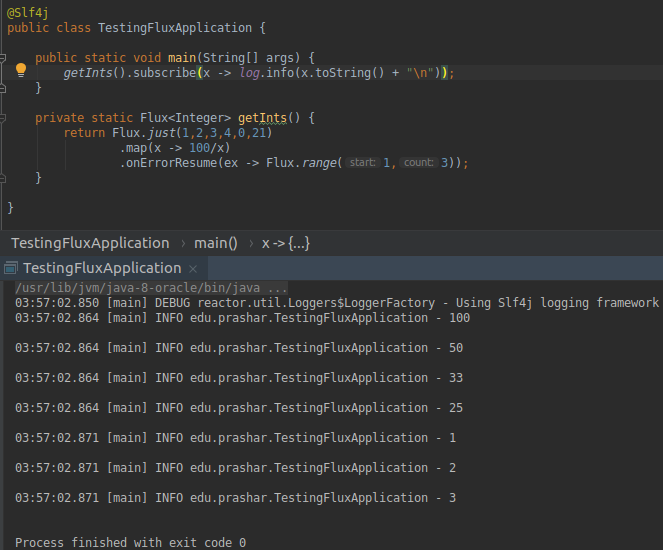
Now we could do multiple things when we encounter an error while working a stream.

1. We may want to substitute a default value in case of an Exception.  
Say we’re processing a stream of integers and we return 100 divided by whatever value we have in the stream. In case we encounter a Zero, we want to return the maximum possible value. In such cases, we have a function like **orErrorReturn( )**.



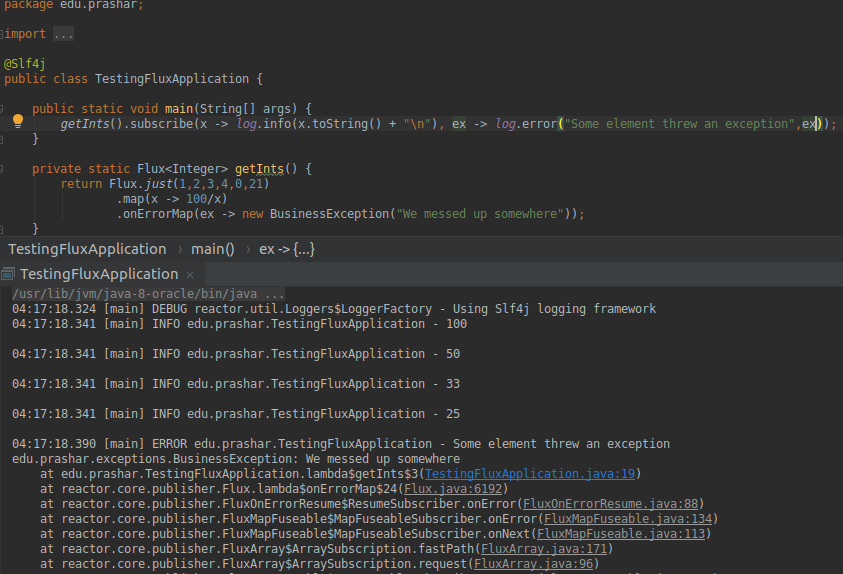
**Note**: See that it didn’t process after getting an Exception. Basically, this means that while processing the stream, whenever you get an error, and you’ve used a onErrorReturn, the processing will stop, control moves to error handling you’ve provided, replaces the erroneous value with the provided value and stops the stream.

2. You may want to provide a fallback publisher in case you encounter an error. Let’s just say you wanted to hit an unreliable service which may produce some error. You may use a **onErrorResume( )**which will return another stream on elements from the point you encountered that error.

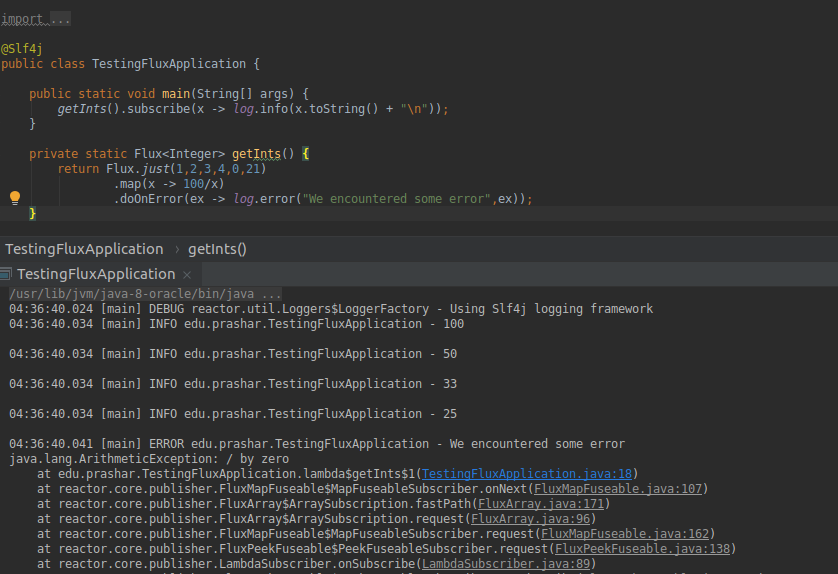


**Note:**Using the instanceOf method, you may be able to verify if its an Exception of some expected type and hence you could handle it accordingly.

3. Sometimes we need to provide custom exceptions in order to be more clear about things. So instead of a DivideByZero we may want DenominatorInvalidException. Such is the use case of **onErrorMap( ).**What you may also notice is we tried to handle the business exception in the subscribe block in the second expression. Subscribe can also expect a throwable parameter and handle stuff. It’s an equivalent of **doOnError( ).**



4. We may want the error to propagate and just want to log stuff up so that we know where it failed. This is where **doOnError( )**can help us. It’ll catch, perform side-effect operation and rethrow the exception.



So this is how we can functionally handle our exceptions in Reactive Streams. Hope you enjoyed this blog.