Available online www.jsaer.com

***Journal of Scientific and Engineering Research*, 2021, 8(2):250-253**

Research Article

ISSN: 2394-2630
CODEN(USA): JSERBR

Building Reactive Applications with Kotlin Flows

Nilesh Jagnik

Mountain View, USA
nileshjagnik@gmail.com

Abstract: Reactive programming can be used to make programs faster and more responsive. The support for reactive programming is available in many languages with ReactiveX libraries. However, the learning curve and code complexity remain the main drawbacks of these libraries. Kotlin introduces a simple, out of the box language feature for reactive programming called Flows. In this paper we present the benefits of using reactive programming in general and discuss the use of Kotlin Flows to apply the reactive paradigm to software pipelines.

Keywords: reactive programming, responsive applications, asynchronous programming, observer pattern

1. Introduction

Building scalable and performant systems requires handling data and events generated at a very high rate and quantity. Software systems need to be fast and responsive to support real world applications. One way to ensure good performance of systems is to adopt the asynchronous programming paradigm. In the async (or asynchronous) programming paradigm, execution of code is suspended while waiting on inputs from external events. This explicit suspension allows CPU resources to be spent only on executing code that is actually ready to execute and does not need to wait on external events. In this way, async programming is event driven. Meaning that one event allows another to occur.

However, even in the async programming the data/events originating from external sources are treated as all or none. Async frameworks only allow asynchronous gathering of data followed by asynchronous processing. But what if the data being gathered is large? Instead of waiting for the entire data set to be gathered, can we start partial processing of some parts of data as it becomes available?

This is where reactive streams come in. In this paradigm, instead of processing a collection of data, we process an asynchronously generated stream of data. Using this paradigm, we can develop code that asynchronously processes a single unit of data at a time. This allows asynchronously processing of data as it is generated.

Reactive streams are quite popular in web applications. However, any software system dealing with a high volume of data can benefit from it. Kotlin provides support for reactive streams with in-built language features, making it an excellent choice for such applications.

Due to the fact that Kotlin and Java are interoperable, we can also make use of Kotlin Flows in Java based systems. In this paper we start with a background on reactive streams, and why they are favorable. We then discuss Kotlin Flows which are motivated by reactive streams but also simplify them.

Reactive streams were initially released as a part of the Reactive Extensions (also known as ReactiveX or Rx) libraries, only available in .NET in 2010, but were ported to Java (called RxJava) in 2013. Kotlin Flows are heavily inspired by reactive streams and maintain most of the advantages while removing the main disadvantages.

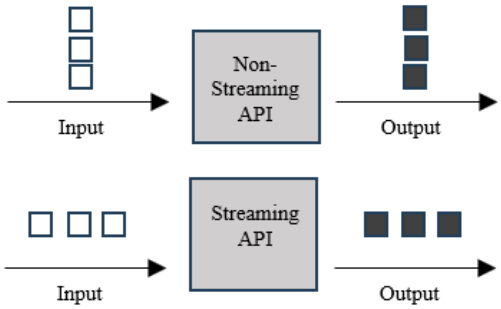

Fig. 1. *Non-Streaming vs Streaming APIs*

2. Reactive Streams

A. Reactive Programming

Reactive streams come from the broader concept of reactive programming in computer science. In the reactive programming paradigm, relationships between static collections and dynamic data streams are defined in a declarative manner. Any changes in one such entity should automatically reflect into other related entities according to defined relationships. This update should happen without the program needing to explicitly update these entities.

B. Observer Pattern

Reactive streams use the observer pattern as a way to achieve reactive programming. In the observer pattern, an object (known as subject) maintains a list of dependents (called observers) and notifies them when there are any changes to it. For reactive streams, the consumers of data streams are observers which are notified to take actions when new data is added to the data stream.

C. Backpressure

Imagine a case where the input stream is producing data at a very high rate and the consumer of the stream isn't capable of processing at the same rate. This could cause buffering of data while waiting to be processed. The buffers could eventually grow to such a limit that some data may not fit in the consumer's memory leading to data being dropped. To solve this problem, reactive streams have the concept of backpressure. With backpressure the consumer of data can signal to the producer to slow down production. This feature allows pipelines to run smoothly by allowing the bottleneck components to dictate the rate of data processing.

D. Cold vs Hot Data Sources

Hot data sources are ones that are generate data even if no listeners/consumers of this data exist. Cold data sources are those can generate data only when it can be utilized by a consumer. In general, cold data sources are easier to deal with so they should be used wherever possible. Reactive streams have support for both hot and cold data sources.

3. Benefits of Reactive Streams

A. Responsiveness

Reactive systems are highly responsive because they process data quickly and effectively. Errors in processing are also propagated quickly.

B. Resilience

Reactive systems are divided into several components due to being comprised of producer and consumer chains. Failures in one component are handled within itself without causing any other components to fail.

C. Elasticity

Utilizing the backpressure feature, reactive systems stay stable under varying load patterns. This leads to reactive systems being more stable.

D. Data Driven

Reactive systems rely on asynchronous processing of data streams. Data streams are used for transfers between component boundaries. This ensures loose coupling of components. Asynchronous processing allows components to efficiently utilize system resources.

4. Caveats of Reactive Streams

A. Code Complexity

The main problem with reactive streams is that it introduces complex programming structures. It is difficult to understand for a beginner and therefore, is prone to programming errors.

B. Debuggability

The asynchronous and loosely coupled nature of components in a reactive system makes it difficult to debug errors. The call stack showed by the stacktrace is less useful. Code can also become hard to dig into due to abstractions.

C. Error Handling

Error handling in reactive streams is different than normal programming. Normally, programmers are forced to catch checked exceptions in code by the compiler. In async and by extension reactive programming, error handling needs careful consideration. The listeners of observables need to handle exceptions.

5. Kotlin Flows

Kotlin Flows are based on the concept of reactive streams. However, they are simpler to understand and use due to a few factors.

A. Structured Concurrency

Kotlin has native support for asynchronous programming through coroutines. Coroutines provide structured concurrency due to their ability to be suspended. Suspension also provides a way for the language to support flow control (backpressure).

B. Cold Data Streams

Kotlin Flows are a way to do reactive programming in Kotlin, but they only support cold data streams. Hot data sources require resource management and stream closing which isn't required for cold sources. Due to this, Kotlin Flows are very simple to understand, read and work with. This eliminates one of the main drawbacks of reactive streams.

```
fun coldSource(): Flow<Int> = flow {
    for (i in 1..3) {
        delay(100)
        emit(i*10)
    }
}

fun main() = runBlocking<Unit> {
    // Collect the flow
    coldSource().collect { value -> println(value) }
}
```

Fig. 2. *Kotlin Flow example*

C. Flow Control / Backpressure

Unlike reactive streams in other languages, code doesn't need to explicitly handle flow control. Kotlin Flows automatically suspend a producer if a consumer is unable to process more data. To optimize flow control in a pipeline, the buffer() operator can be used on a Flow to specify the size of a buffer.

D. Flow Builders

Fig. 2 shows the use of the flow {} builder which is the most basic way to build flows. The flowOf() operator can take a variable list of arguments. Various collections can be converted into flows using the .asFlow() extension.

E. Intermediate Operators

For transformations on flow values, many stream operators like map() and filter() are supported. The transform() operator is the most generic way to perform flow transformations.

F. Terminal Operators

As shown in Fig. 2, at the end of flow execution, the collect() operator can be used to collect emitted values. There are also other collectors like reduce() and fold(). Other options for collection include toList() and toSet().

G. Compositions

The zip() and combine() operators can be used to merge multiple flows into one.

H. Cancellations

Flows can be cancelled similar to any other coroutines in Kotlin. The cancel() operator can be used from within a Flow to cancel its operation. The CancellationException is thrown when this happens.

I. Error Handling

The terminal operators of a flow can be surrounded by try/catch blocks to handle errors. This way, any exceptions raised during the flow executions can be caught and handled. The catch() operator can also be used on the emitter of a flow to handle errors raised by the emitter.

J. Flow Completion

In addition to using the try/catch blocks to surround terminal flow operators, a finally block can be used to do any post processing. Kotlin also offers an onCompletion() intermediate operator to perform any actions once a flow is complete.

6. Conclusion

Reactive programming can have many benefits when using it for building highly responsive systems. These can drastically improve the performance of a data processing pipeline when used correctly. Reactive programming patterns are a bit different as compared to traditional programming. For this reason, it is a good idea to familiarize oneself with the it's subtleties before using it. The use of Kotlin Flows offers a very simple and easy to understand way for using reactive programming. There are also several safeguards in place to prevent programmers from making errors in code. This along with the fact that Kotlin is interoperable with Java makes it easier to include reactive programming in Java codebases.

References

[1]. André Staltz, "The introduction to Reactive Programming you've been missing (Nov 2018)," https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

[2]. Roman Elizarov, "Reactive Streams and Kotlin Flows (Jun 2019)," https://elizarov.medium.com/reactive-streams-and-kotlin-flows-bfd12772cda4

[3]. Daniel Caldas, "Reactive Programming: The Good and the Bad (Dec 2020)," https://goodguydaniel.com/blog/reactive-rxjs-pros-cons

[4]. Tompee Balauag, "RxJava Ninja: Hot and Cold Observables (Aug 2018)," https://medium.com/tompee/rxjava-ninja-hot-and-cold-observables-19b30d6cc2fa

[5]. Roman Elizarov, "Cold flows, hot channels (Apr 2019)," https://elizarov.medium.com/cold-flows-hot-channels-d74769805f9

[6]. "Asynchronous Flow (Feb 2021)," https://kotlinlang.org/docs/flow.html