Reactive Streams + RxJava

Reactive Streams has been a collaborative effort to standardize the protocol for asynchronous streams on the JVM. The RxJava team was part of the effort from the beginning and supports the use of Reactive Streams APIs and eventually the Java 9 Flow APIs which are resulting from the success of the Reactive Stream effort.

How does this relate to RxJava itself?

RxJava 1.x Currently RxJava 1.x does not directly implement the Reactive Streams APIs. This is due to RxJava 1.x already existing and not being able to break public APIs. It does however comply semantically with the non-blocking "reactive pull" approach to backpressure and flow control and thus can use a bridge between types. The RxJavaReactiveStreams module bridges between the RxJava 1.x types and Reactive Streams types for interop between Reactive Streams implementations and passes the Reactive Streams TCK compliance tests.

Its API looks like this:

```
package rx;

import org.reactivestreams.Publisher;

public abstract class RxReactiveStreams {

    public static <T> Publisher<T> toPublisher(Observable<T> observable) { ... }

    public static <T> Observable<T> toObservable(Publisher<T> publisher) { ... }

}
```

RxJava 2.x RxJava 2.x will target Reactive Streams APIs directly for Java 8+. The plan is to also support Java 9 `j.u.c.Flow` types by leveraging new Java multi-versioned jars to support this when using RxJava 2.x in Java 9 while still working on Java 8.

RxJava 2 will truly be "Reactive Extensions" now that there is an interface to extend. RxJava 1 didn't have a base interface or contract to extend so had to define it from scratch. RxJava 2 intends on being a high performing, battle-tested, lightweight (single dependency on Reactive Streams), non-opinionated implementation of Reactive Streams and `j.u.c.Flow` that provides a library of higher-order functions with parameterized concurrency.

Public APIs of Libraries

A strong area of value for Reactive Streams is public APIs exposed in libraries. Following is some guidance and recommendation on how to use both Reactive Streams and RxJava in creating reactive libraries while decoupling the concrete implementations.

Pros of Exposing Reactive Stream APIs instead of RxJava

- Lightweight: Very lightweight dependency on interfaces without any concrete implementations. This keeps dependency graphs and bytesize small.
- Future Proof: Since the Reactive Stream API is so simple, was collaboratively defined and is becoming part of JDK 9 it is a future proof API for exposing async access to data. The `j.u.c.Flow` APIs of JDK 9 match the APIs of Reactive Streams so any types that implement the Reactive Streams `Publisher` will also be able to implement the `Flow.Publisher` type.
- Interop: An API exposed with Reactive Streams types can easily be consumed by any implementation such as RxJava, Akka Streams and Reactor.

Cons of Exposing Reactive Stream APIs instead of RxJava

- A Reactive Stream `Publisher` is not very useful by itself. Without higher-order functions like `flatMap` it is just a better callback. This means that consumption of a `Publisher` will almost always need to be converted or wrapped into a Reactive Stream implementation. This can be verbose and awkward to always be wrapping `Publisher` APIs into a concrete implementation. If the JVM supported extension methods this would be elegant, but since it doesn't it is explicit and verbose.

  Specifically the Reactive Streams and Flow `Publisher` interfaces do not provide any implementations of operators like `flatMap`, `merge`, `filter`, `take`, `zip` and the many others used to compose and transform async streams. A `Publisher` can only be subscribed to. A concrete implementation such as RxJava is needed to provide composition.

- The Reactive Streams specification and binary artifacts do not provide a concrete implementation of `Publisher`. Generally a library will need or want capabilities provides by RxJava, Akka Streams, etc for its internal use or just to produce a valid `Publisher` that supports backpressure semantics (which are non-trivial to implement correctly).

Recommended Approach

Now that Reactive Streams has achieved 1.0 we recommend using it for core APIs that are intended for interop. This will allow embracing asynchronous stream semantics without a hard dependency on any single implementation.

This means a consumer of the API can then choose RxJava 1.x, RxJava 2.x, Akka Streams, Reactor or other stream composition libraries as suits them best. It also provides better future proofing, for example as RxJava moves from 1.x to 2.x.

However, to limit the cons listed above, we also recommend making it easy for consumption without developers needing to explicitly wrap the APIs with their composition library of choice. For this reason we recommend providing wrapper modules for popular Reactive Stream implementations on top of the core API, otherwise your customers will each need to do this themselves.

Note that if Java offered extension methods this approach wouldn't be needed, but until Java offers that (not anytime soon if ever) the following is an approach to achieve the pros and address the cons.

For example, a database driver may have modules such as this:

// core library exposing Reactive Stream Publisher APIs * async-database-driver

// integration jars wrapped with concrete implementations * async-database-driver-rxjava1 * async-database-driver-rxjava2 * async-database-driver-akka-stream

The "core" may expose an API like this:

```
package com.database.driver;

public class Database {
   public org.reactivestreams.Publisher getValue(String key);
}
```

The RxJava 1.x wrapper could then be a separate module that provides RxJava specific APIs like this:

```
package com.database.driver.rxjava1;

public class Database {
    public rx.Observable getValue(String key);
}
```

The core `Publisher` API can be wrapped as simply as this:

```
public rx.Observable getValue(String key) {
   return RxReactiveStreams.toObservable(coreDatabase.getValue(key));
}
```

The RxJava 2.x wrapper would differ like this (once 2.x is available):

```
package com.database.driver.rxjava2;

public class Database {
```

```
    public io.reactivex.Observable getValue(String key);
}
```

The Akka Streams wrapper would in turn look like this:

```
package com.database.driver.akkastream;

public class Database {
    public akka.stream.javadsl.Source getValue(String key);
}
```

A developer could then choose to depend directly on the `async-database-driver` APIs but most will use one of the wrappers that supports the composition library they have chosen.

If something could be clarified further, please help improve this page via discussion at https://github.com/ReactiveX/RxJava/issues/2917