MicroProfile approach to reactive

DRAFT - Last updated: 21 March 2018

This document outlines the approach that MicroProfile will use to adopting the reactive paradigm for programming and architecting microservices.

What is reactive?

Reactive systems are described in The Reactive Manifesto. This calls for a coherent approach for architecting systems that allows them to be responsive, by creating loosely coupled components that are resilient to failure and elastic in the face of changing load. At the heart of the approach to loosely coupling components is the idea of being non blocking, using asynchronous message passing to ensure components are not impacted by each others availability.

Reactive is popularly associated with asynchronous programming, using non blocking IO. While asynchronous programming is certainly an important part of reactive, as it applies to a microservice platform such as MicroProfile, it is also much bigger than just asynchronous programming. Reactive impacts the way we architect our systems, and this most prominently features in the way services communicate with each other - services that communicate asynchronously, and hence are able to run autonomously, are able to scale, fail and evolve independently. Not only does this help a system to be reactive, it allows a microservice based system to realise the full benefits of the microservice architecture, and overcome many of the problems that microservices introduce when compared to monoliths.

Hence there are two primary concerns when applying reactive to MicroProfile. The first is the application of reactive programming to the APIs offered by MicroProfile. The second is in deciding what new features, APIs or projects, if any, are needed to allow developers to create reactive system architectures.

Reactive programming

In order to provide reactive programming APIs, abstractions are required to allow developers to compose multiple asynchronous operations together.

A naïve approach to offering an asynchronous API called `Foo` may be to offer a way to register a callback to receive an asynchronously provided value. For example, that callback interface might look like:

```
interface FooEventListener {
  void onFooEvent(FooEvent fooEvent);
}
```

Just considering the above API, there are a number of problems with this:

- This specific API requires specific code to implement it. If I want to connect this API to the asynchronous API offered by another component, say `Bar`, I need to write an adapter that adapts the API offered by `Bar` to the API offered by `Foo`. And then I need another adapter for every permutation of APIs that I might want to integrate each other. Of course, this isn't so bad given that `FooEventListener` is a SAM and so I can use method references, but...
- The above listener offers no way to handle errors. There are multiple ways that this can be addressed, but none of them are ideal. For example, we might add another method, `onError(Throwable)`. The addition of this method means it's no longer a SAM, and so can no longer exploit the convenience of Java 8 method references, and also means when connecting two APIs, I now need to adapt the error handling semantics of each API, which may be subtly different.
- This API offers no mechanism for exerting backpressure asynchronously. The best I can do is to synchronously block the thread before returning from `onFooEvent`. If in my handling of the event I want to use an asynchronous database API to persist something, there's no way for me to tell the invoker of the listener to wait until I've finished with my database call. If my database operation is slower than the incoming events are arriving, I have two options, buffer the events I receive and risk running out of memory, or drop the events. Even if it did offer a backpressure mechanism, there are many different ways that backpressure can be implemented, and adapting them to each other is non trivial.
- This API offers no mechanism for ensuring thread safety, especially if I am integrating multiple APIs with callbacks registered with all of them. If I'm updating some state using this API, and I also have a `BarEventListener` registered that could execute at the same time to update the state, how do I ensure that these two callbacks execute with exclusion? How do I ensure that the right memory fences are in place before they are executed? The use of the `synchronized` keyword results in blocking, the use of `AtomicReference` and other `java.util.concurrent` constructs is overly burdensome for an application developer who just wants to solve their domain problem.
- When using Java 8 lambdas to implement an API like this, this API is susceptible to callback hell. This is a phenomenon that shows itself in two primary ways, firstly the code becomes unreadable due to sequential operations being spread out through deeply nested lambdas. Secondly, its very easy to drop signals, particularly error signals, by forgetting to handle them correctly. The result is that a process just stops mid way through, leaving the developer with no idea why it stopped, with no way to find out, not even with a thread dump.

- There is no explicitly defined execution context for this API, and no way to customise that context. If thread locals are needed, there's no way to ensure that this API executes its callbacks with those thread locals.
- Any monitoring solutions that want to trace the flow of control through this API need to be explicitly aware of the API and what it does, so they can explicitly instrument it to carry correlation identifiers.

And that's just the problems with one possible trivial API for asynchronous programming. In order to safely provide asynchronous programming features, high level, generally applicable patterns and components are needed to ensure compatibility, correct error propagation, backpressure propagation, thread safety, and clean code when using these features. Unfortunately, there already exist APIs in Java EE (such as JSR 356 WebSockets) that suffer from many of these problems (and I should point out that APIs like JSR 356 were created before standard solutions to these problems existed, so they shouldn't be blamed for this). It's important that going forward, we ensure that these same problems aren't reintroduced.

In general, there are two different types of operations that we want to do when doing asynchronous programming. The first is an operation that produces exactly one value, or fails. Examples of such operations include:

- Producing an asynchronous response for an incoming HTTP request
- Making HTTP requests on remote systems
- Database queries and updates

The second is an operation that produces many, or a stream of values, with failure and completion at the stream level. Examples of such operations include:

- Sending and receiving messages to/from a message broker
- Sending and receiving messages to/from a WebSocket
- Streaming results from a large database result set
- Streaming bytes to/from an HTTP request/response body

There exist two standard APIs in the JDK that provide abstractions for representing these operations, they are `java.util.concurrent.CompletionStage`, and Reactive Streams à la `java.util.concurrent.Flow`.

CompletionStage

`CompletionStage` is a great abstraction for handling a single value. It offers the following features:

- A standard interface that can be shared/passed between many asynchronous APIs
- Values can be transformed using `thenApply`.

- Multiple `CompletionStage`s produced by different asynchronous APIs can be composed using `thenCompose`.
- Error handling is well defined - errors propagate through chains of completion stages, with features for recovering from and handling errors at any point in the chain.
- Thread safety is well defined, with callbacks running between memory fences.
- Backpressure inherent in the redemption of the value.
- Customizable execution contexts, allowed by passing an explicit executor into methods like `thenApplyAsync`, `thenComposeAsync`.

`CompletionStage` should be used for all APIs that asynchronously produce or consume **exactly one** value. In some places the `CompletionStage` may be returned by an API, like so:

```
CompletionStage<Response> response = someApi.makeRequest();
```

In other places, a CompletionStage may be returned by application code, like so:

```
@Path("/")
class MyResource {
  @GET
  CompletionStage<Response> handleRequest() {
    return someOtherApi.doOperation()
      .thenApply(result -> Response.ok());
  }
}
```

The above examples are similar to the way JAX-RS 2.1 handles asynchronous calls. Sometimes, an API might have an existing blocking variant, and the asynchronous API is being added alongside it. To support that, it is recommended that `Async` be added to the asynchronous variant of the method, for example:

```
CompletionStage<Response> response = someApi.makeRequestAsync();
```

This approach is in line with the approach the JDK9 HTTP client has taken. Note that this differs from JAX-RS 2.1's approach of using the name `rx`. Rx is actually a shortening of the name of a specific product called Reactive Extensions, originally created for .NET by Microsoft, and later ported to Java by its creator. The use of this naming is strongly discouraged, as it is a product name, not a standard or a generally applicable name of a concept.

Reactive Streams

Reactive Streams is an asynchronous streaming API, produced by a collaboration of engineers representing Netflix, Red Hat, Pivotal, Oracle, Lightbend and others. It was adopted by the JDK in JDK9.

Reactive Streams provides very well defined semantics for data flow, backpressure, error propagation, completion and cancelling, thread safety, infinite recursion prevention, and other things, allowing two implementations of Reactive Streams to integrate seamlessly with no specific support beyond the Reactive Streams specification in either of them. To get a feel for how well defined the semantics are, read through the spec. It also has a TCK that does a thorough job of ensuring implementations implement the spec correctly and completely.

It should be stressed that Reactive Streams is intended to be used as an integration API, not an application developer API. Libraries are meant to implement Reactive Streams interfaces, not application developers, the most that application developers should do is pass around instances of `Publisher` and `Subscriber`, and perhaps plumb them together via the `subscribe` method. This blog post does a good job of demonstrating why application developers should never implement their own publishers or subscribers, showing how just implementing an incredibly simple publisher is incredibly difficult to get right, not just to implement the requirements of the spec, but to get the thread safety and concurrency concerns correct.

A point of contention among the Reactive Streams community is whether APIs should be Publisher biased, always accepting publishers, never returning subscribers, or whether Subscribers should be a first class citizen in end user APIs. To demonstrate the difference, here's what a publisher biased API might look like, when plumbing the result of an HTTP client request to a servlet response (assuming both support reactive streams):

```
CompletionStage<Response> future = httpClient.makeRequest();
future.thenAccept(response ->
  servletResponse.send(response.getPublisher()));
```

In contrast, this is what an API that uses subscribers might look like:

```
CompletionStage<Void> result = httpClient.makeRequest(response ->
  servletResponse.asSusbscriber());
```

In some ways, the horse has already bolted here, since the JDK9 HTTP client API uses the latter pattern, where you are required to return a Subscriber to it when handling the response. There are advantages and disadvantages to both approaches, the subscriber approach means that the HTTP client can ensure that its response is always consumed, protecting the developer potential resource leaks if they forget to consume it, while the publisher biased approach gives a unified way of always publishing `Publisher`s, and means you're never left in a situation where you need to connect an API that requires you to pass `subscriber` to an API that requires you to pass a publisher.

It's my opinion that it doesn't matter which approaches we support, that a decision should be made on a case by case basis as to what the API should look like based on the concerns of that

particular API, for example if resource leaks is a big problem, but I did want to raise this because there are people in the Reactive Streams community that disagree with this approach.

Byte streams

When offering byte streams, eg, request/response bodies, or database blobs, then `Publisher<ByteBuffer>/Subscriber<ByteBuffer>` should be offered as the API. The byte buffers passed to application developer code should be **unpooled, non reusable, unmodifiable** buffers, and byte buffers received from application developer code should not be mutated by the library.

JDK9 vs org.reactivestreams strategy

There currently exist two Reactive Streams APIs. The first is provided by http://www.reactive-streams.org/, and lives in the `org.reactivestreams` package. The second is provided by JDK9, and lives as inner interfaces of the `java.util.concurrent.Flow` class. Both APIs are identical in everything but namespace. The JDK9 one requires MicroProfile to move to a baseline supported JDK version of JDK9 before it can be adopted.

For APIs that are introduced before that happens, we need a strategy for how to support Reactive Streams using the `org.reactivestreams` version that will be backwards compatible with adding support for the JDK9 version in future, while giving us a path to phase out, rather than breaking, the `org.reactivestream` support.

There are a couple of strategies, multiple will likely apply:

- Some CDI based APIs are not strongly typed, eg a user might implement a method that returns a `Publisher`, and annotates it to indicate that it's a messaging stream. The framework interacts with this method using reflection, and so can transparently add support for JDK9 flows later, with no impact on user code.
- An API that accepts a `Publisher` or `Subscriber` can be overloaded to support the JDK9 types in future.
- An API that accepts a `Publisher` or `Subscriber` as a generic type of another type can't be overloaded, since they will have the same binary signature after erasure. For example, something accepts a `Supplier<Subscriber>`. A possible option here would be to accept purpose built SAMs, this solves the binary problem, however in practice this often doesn't work well with Java type inference with lambdas, it's far too easy for developers to run into edge cases that javac can't resolve.
- An API that returns a `Publisher` or `Subscriber` can't be overloaded, as the Java compiler doesn't allow overloading by return type.
- When an API has to return a `Publisher` or `Subscriber`, or accept a Publisher or Subscriber type parameter, a way to future proof this is to decide on a way to

disambiguate these methods with different names. For example, `getFlowPublisher` might be used for JDK9, while `getPublisher` might be used for `org.reactivestreams`. Alternatively, `getRsPublisher` might be used for `org.reactivestreams`, while `getPublisher` might be used for JDK9.

Reactive Streams manipulation strategy

One of the major shortcomings of Reactive Streams at present is the lack of a standard API for manipulating them. Consider a use case where we want to connect a source, `Publisher<Foo>` to a sink, `Subscriber<Bar>`, and we have a function, `Function<Foo, Bar>` to do the transformation. There doesn't exist any method in Reactive Streams that allows a developer to apply that transformation function to each element. Instead, they would have to write their own Publisher/Subscriber that wrapped the provided publisher/subscriber to do the transformation, which not only is a lot of boilerplate, it's strongly discouraged that users write their own implementations of Publisher and Subscriber. A simple map transformation may be trivial to write, but it gets far more complex with things like filter, where you drop elements and so need to work with demand, and then substreams, asynchronous mappings, etc, get even worse.

Of course, this isn't a problem for most existing users of Reactive Streams, because there exist a number of third party libraries that provide these transformations, such as map/filter/flatMap. These libraries include Akka Streams, RxJava 2 and Reactor. However, for a Java standard like MicroProfile, requiring developers to bring in a third party library to do these elementary operations is not acceptable.

Fortunately there is a solution in the works, a standard library is currently being developed for the JDK, with strong interest from key JCP members for its inclusion in a future JDK version. Of course, the absolute earliest that such an API could be included in the JDK is JDK11, and it's likely to be much later than that, so it's going to be some time before MicroProfile moves to a baseline JDK version that can take advantage of that API.

https://github.com/lightbend/reactive-streams-utils

In the meantime, there are a number of strategies that we could take:

- Always provide variants of APIs that don't require reactive streams manipulation. For example, a messaging API may provide a way of subscribing using a `Subscriber`, as well as a way of subscribing using a callback that returns `CompletionStage<Void>`.
- Have libraries provide their own utilities for the most common use cases. The JDK9 HTTP client does this, it provides built in Subscribers that save request bodies to a file, aggregate them in memory, as well as adapts it to a blocking `OutputStream`.

- Adopt a port of the JDK library Reactive Streams utility library for MicroProfile, which can be offered to developers before the JDK version is available, and then deprecated once MicroProfile moves to a baseline JDK that supports the library. This is probably the best option, however it is contingent on being confident that that library will eventually materialize in the JDK, and that it won't be significantly different to the form that MicroProfile adopts it in.

Reactive system architecture features

Messaging API

In order for services to be autonomous, they need to be able to communicate with each other asynchronously. The term synchronous means "at the same time", and as applied to communication, means that both parties must be involved in the communication at the same time for the communication to be successful, they need to synchronize to communicate. An example of this is HTTP, service A cannot communicate with service B using HTTP if service B has crashed, is overloaded, or if there is a network partition between them. In contrast, asynchronous communication allows two parties to communicate without them both being active participants at the same time. If service A asynchronously sends a message to service B, service B can be crashed, overloaded, or there can be a network partition between them, and this won't impact the success of the communication. Service B can consume that message at a later point in time, when it has recovered from whatever problems it had, no synchronization is needed.

Hence, in order to allow developers to build reactive microservices with MicroProfile, MicroProfile needs an API for asynchronous messaging.