Dataflow mediator for ReactiveStreams reference

FEMA Studios S.r.l.s.

Version 1, revision 1

Table of Contents

Dataflow mediator for ReactiveStreams

This mediator library provides two easy conversions:

- `Publisher` to `Attribute`
- `Field` to `Publisher`

The reasons of this asymmetry is explained in each respective conversion page.

This mediator depends on `reactive-streams`, a library that defines some interfaces implemented by the most common reactive libraries such as ReactiveX and Project Reactor.

The examples in the playground make use of RxJava.

Getting started

Setup

To add this extension all you have to do is add the following dependency:

Gradle

```
implementation 'com.femastudios:dataflow-mediator-reactivestreams:1.0.0'
```

Maven

```
<dependency>
    <groupId>com.femastudios</groupId>
    <artifactId>dataflow-mediator-reactivestreams</artifactId>
    <version>1.0.0</version>
</dependency>
```

Remember also to properly configure your environment as explained as explained here.

Usage

Publisher to Attribute conversion

First of all, we'll explain how to convert a `Publisher` to an `Attribute`. Notice how we are not talking about `Field` here: the reason is that the `Publisher` can enter a state of error, that can only be handled by an `Attribute`.

If you wish, you can convert the returned `Attribute` to a simple `Field` using one of the many functions provided, depending on your needs.

The mediator provides an easy to use extension function to the `Publisher`, called `toAttribute()`:

Kotlin

```
val publisher : Publisher<Int> = Flowable.just(1, 2, 3) //Create a Publisher
    val attribute : Attribute<Int> = publisher.toAttribute() //Converts the Publisher to an
Attribute

    println(attribute.value)
    lifecycle {
        listen(attribute) {
            println(it)
        }
    }
    println(attribute.value)
```

Java

```
Publisher<Integer> publisher = Flowable.just(1, 2, 3); //Create a Publisher
Attribute<Integer> attribute = DataflowReactorMediation.toAttribute(publisher); //Converts
the Publisher to an Attribute

System.out.println(attribute.getValue());
LifecycleOwner.lifecycle(lc -> {
    lc.listen(attribute, value -> {
        System.out.println(value);
    });
});
System.out.println(attribute.getValue());
```

When the `Attribute` becomes active (has at least one listener), it will subscribe to the `Publisher` and update its value with the latest value emitted by the `Publisher`. When the `Attribute` becomes inactive (has not any listener), it will unsubscribe and keep the last seen value.

Until the attribute receives the first value, its status will be *LOADING*.

If the `Publisher` terminates in an error, the attribute state will be set to *ERROR*. More specifically, it will contain an instance of `PublisherError`: this class extends the standard attribute `Error` class and additionally contains the `Throwable` instance that triggered the `Publisher` error. Notice that since the `Publisher` cannot recover from an error state, neither can our attribute.

Field to Publisher conversion

Now we'll explain how to convert a `Field` to a `Publisher`. Why not an `Attribute`? The reason is simple: as already mentioned, once a `Publisher` enters an error state it terminates and cannot recover, oppositely to an `Attribute` where the state can change freely.

For this reason the only possible conversion is from a `Field` to a `Publisher` (that will never terminate with an error).

The function to do this is an extension of `Field` and is called `toPublisher()`:

Kotlin

```
val fld : MutableField<Int> = mutableFieldOf(0) //Create a field
    val publisher : Publisher<Int> = fld.toPublisher() //Converts the Field to a Publisher

    Flowable
        .fromPublisher(publisher) //Example for ReactiveX
        .subscribe {
            println(it)
        }
    fld.increment()
    fld.increment()
    fld.increment()
```

Java

```
MutableField<Integer> attribute = MutableField.of(0); //Create a field
Attribute<Integer> attribute = DataflowReactorMediation.toPublisher(attribute); //Converts
the Field to a Publisher

Flowable
    .fromPublisher(publisher) //Example for ReactiveX
    .subscribe(value -> {
        System.out.println(value);
    });
fld.setValue(1);
fld.setValue(2);
fld.setValue(3);
```

When the value of the `Field` changes and there is backpressure, the new value is emitted.

When there is not backpressure no changes except the last one are emitted. The latest value is emitted the next time a subscriber requests data from the `Publisher`.