Joe | 28 Feb 2021

Filosofía

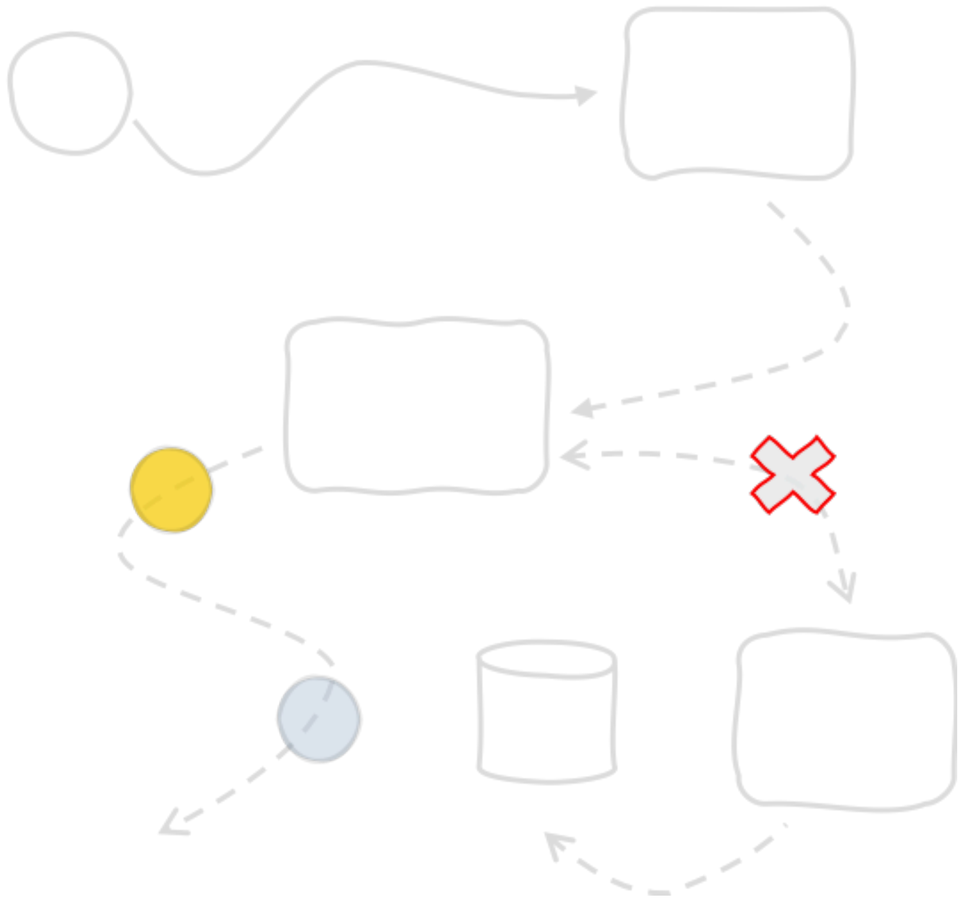


FIGURE 1. DISTRIBUTED SYSTEMS ARE ASYNCHRONOUS

La programación reactiva combina programación funcional, el patrón observador, y el patrón iterable

Programación Reactiva es, acerca de,
programación con streams de datos

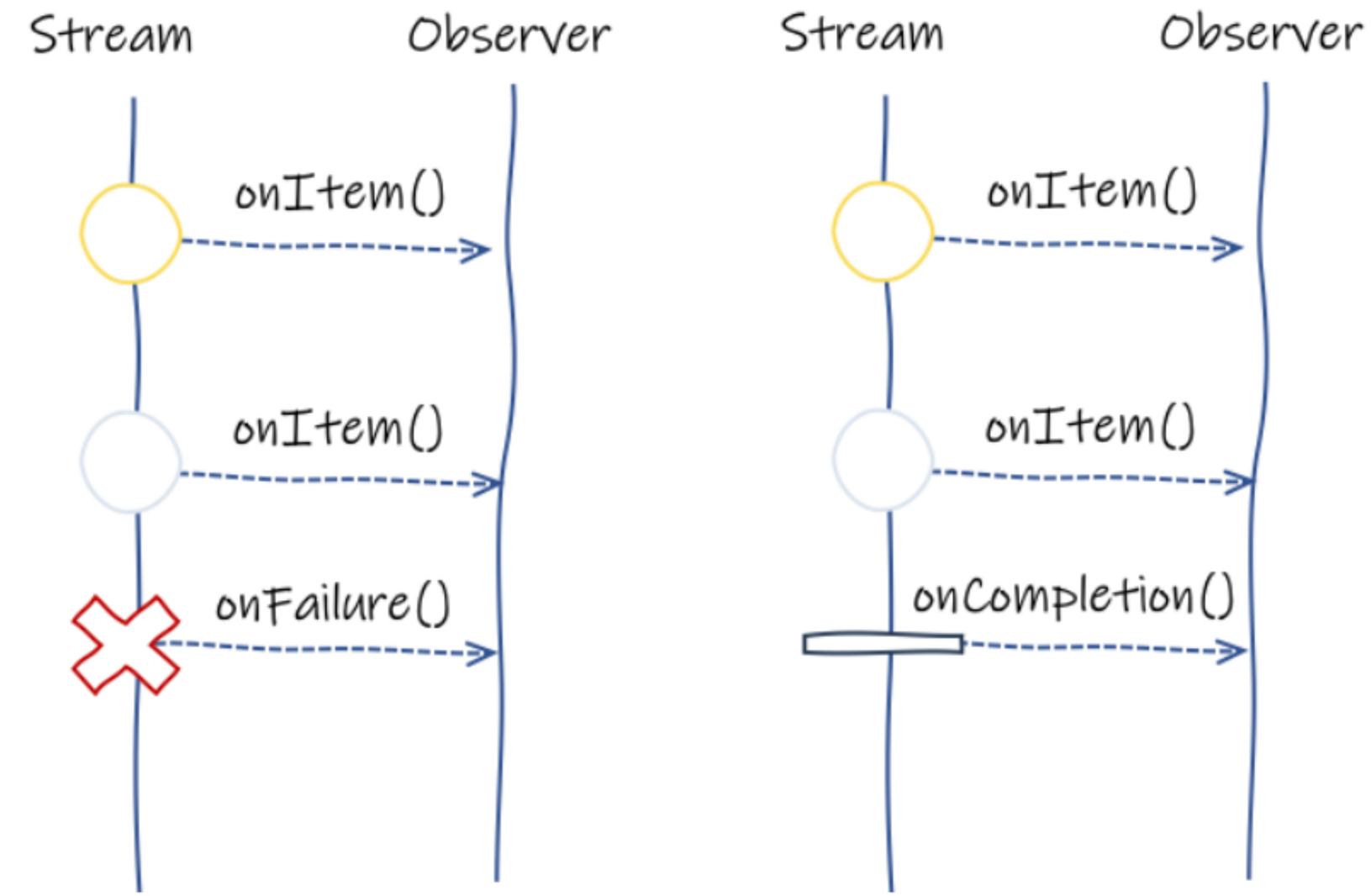
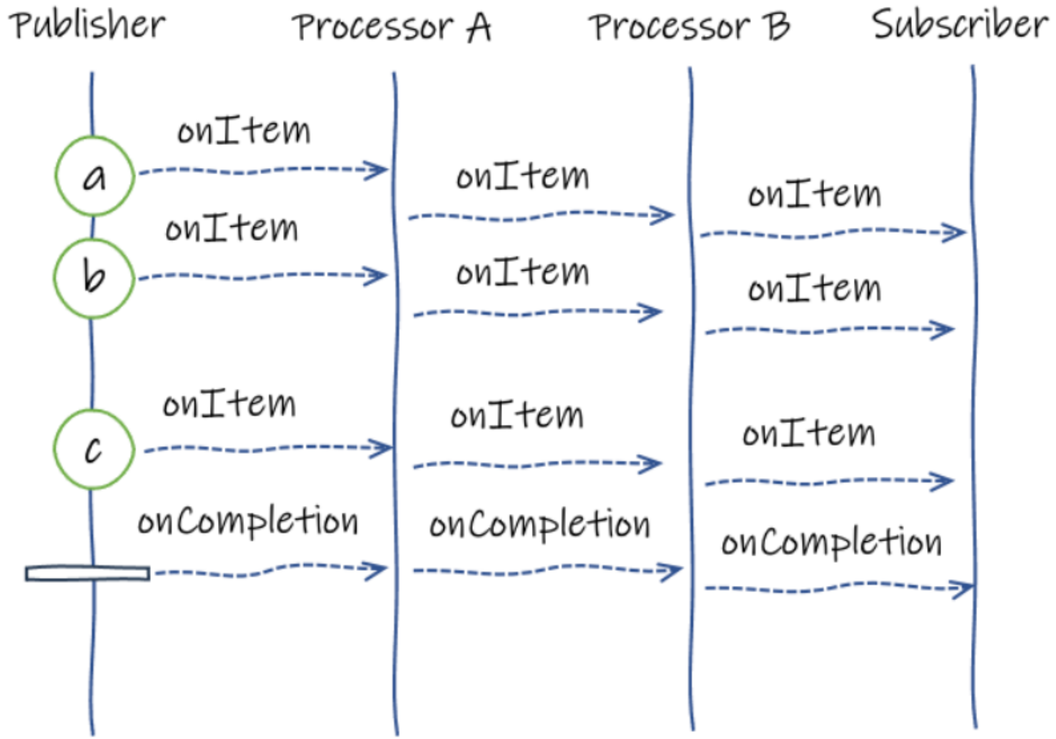


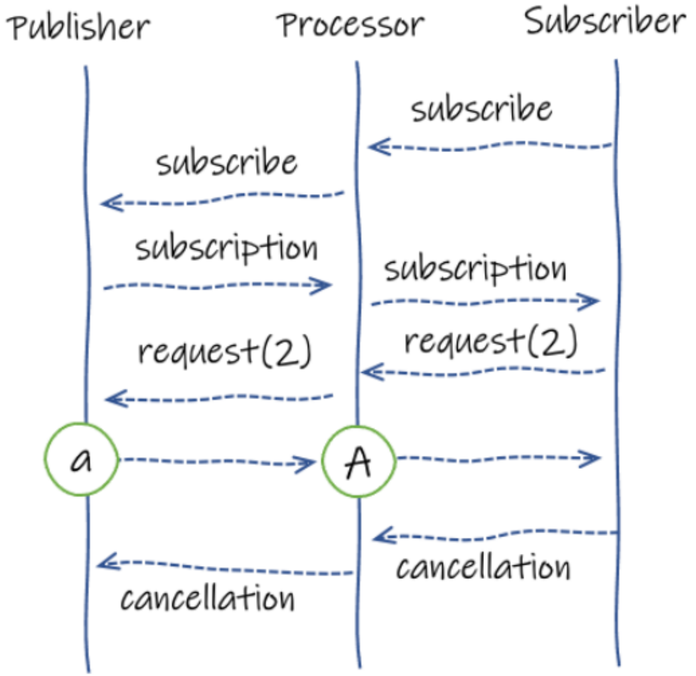
FIGURE 2. REACTIVE PROGRAMMING IS ABOUT OBSERVING STREAMS

¿Qué hace a Mutiny diferente?

- Competencia: Reactor y RxJava
- La diferencia: **El API**
- **Pilares:**
  - **Orientado a Eventos**
  - **Navegabilidad del API**
  - **Simplicidad**

Eventos


Publisher
Processor A
Processor B
Subscriber
onItem
a
onItem
onItem
onItem
onItem
b
onItem
onItem
onItem
onItem
c
onItem
onCompletion
onCompletion
onCompletion


Publisher
Processor
Subscriber
subscribe
subscribe
subscription
subscription
request(2)
request(2)
a
A
cancellation
cancellation

REACTIVE STREAMS

Mutiny's back-pressure is based on Reactive Streams.

DON'T FORGET TO SUBSCRIBE

If no subscriber *subscribes*, no items would be emitted. More importantly, nothing will ever happen. If your program does not do anything, check that it subscribes, it's a very common error.

API orientado a eventos

THE VARIOUS TYPES OF EVENTS

```
Multi<String> source = Multi.createFrom().items("a", "b", "c");
source
  .onItem() // Called for every item
    .invoke(item → log("Received item " + item))
  .onFailure() // Called on failure
    .invoke(failure → log("Failed with " + failure))
  .onCompletion() // Called when the stream completes
    .invoke(() → log("Completed"))
  .onSubscribe() // Called when the upstream is ready
    .invoke(subscription → log("We are subscribed!"))
  .onCancellation() // Called when the downstream cancels
    .invoke(() → log("Cancelled :-("))
  .onRequest() // Called on downstream requests
    .invoke(n → log("Downstream requested " + n + " items"))
  .subscribe()
    .with(item → log("Subscriber received " + item));
```

Uni y Multi

- Multi representa streams de 0..* items
- Uni representa streams que recibe un ítem o una falla.

USAGE OF UNI AND MULTI

```
Multi.createFrom().items("a", "b", "c")
  .onItem().transform(String::toUpperCase)
  .subscribe().with(
    item -> System.out.println("Received: " + item),
    failure -> System.out.println("Failed with " + failure)
);

Uni.createFrom().item("a")
  .onItem().transform(String::toUpperCase)
  .subscribe().with(
    item -> System.out.println("Received: " + item),
    failure -> System.out.println("Failed with " + failure)
);
```

Usando Mutiny en una aplicación Java

MAVEN

```
<dependency>
  <groupId>io.smallrye.reactive</groupId>
  <artifactId>mutiny</artifactId>
  <version>0.14.0</version>
</dependency>
```

GRADLE WITH GROOVY

```
implementation 'io.smallrye.reactive:mutiny:0.14.0'
```

GRADLE WITH KOTLIN

```
implementation("io.smallrye.reactive:mutiny:0.14.0")
```

JBANG

```
//DEPS io.smallrye.reactive:mutiny:0.14.0
```

Usando Mutiny con Quarkus

```
mvn quarkus:add-extension -Dextensions=mutiny
```

Or directly add the dependency to your `pom.xml`

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-mutiny</artifactId>
</dependency>
```

Usando Mutiny BOM

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.smallrye.reactive</groupId>
            <artifactId>mutiny-bom</artifactId>
            <version>0.14.0</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>
    <dependency>
        <groupId>io.smallrye.reactive</groupId>
        <artifactId>mutiny</artifactId>
    </dependency>
</dependencies>
```

In Gradle, add:

```
dependencies {
  implementation platform("io.smallrye.reactive:mutiny-bom:0.14.0")
  implementation("io.smallrye.reactive:mutiny")
}
```

Usando Mutiny con Eclipse Vert.x

```
<dependency>
  <groupId>io.smallrye.reactive</groupId>
  <artifactId>smallrye-mutiny-vertx-core</artifactId>
  <version>1.5.0</version>
</dependency>
```

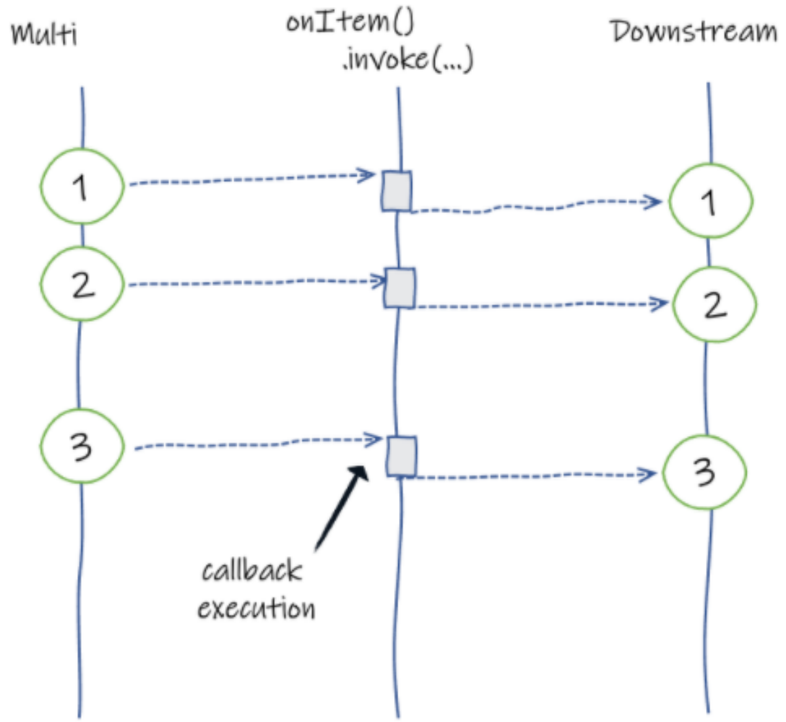
Observando eventos

Uni y Multi emiten eventos.

EVENT	UNI / MULTI	DIRECTION	NOTE
item	Uni and Multi	upstream → downstream	The upstream sent an item.
failure	Uni and Multi	upstream → downstream	The upstream failed.
completion	Multi only	upstream → downstream	The upstream completed.
subscribe	Uni and Multi	downstream → upstream	A downstream subscriber is interested in the data.
subscription	Uni and Multi	upstream → downstream	Even happening after a `subscribe` event to indicate that the upstream acknowledged the subscription.
cancellation	Uni and Multi	downstream → upstream	A downstream subscriber does not want any more event
overflow	Multi only	upstream → downstream	The upstream has emitted more than the downstream can handle
request	Multi only	downstream → upstream	The downstream indicates its capacity to handle *n* items

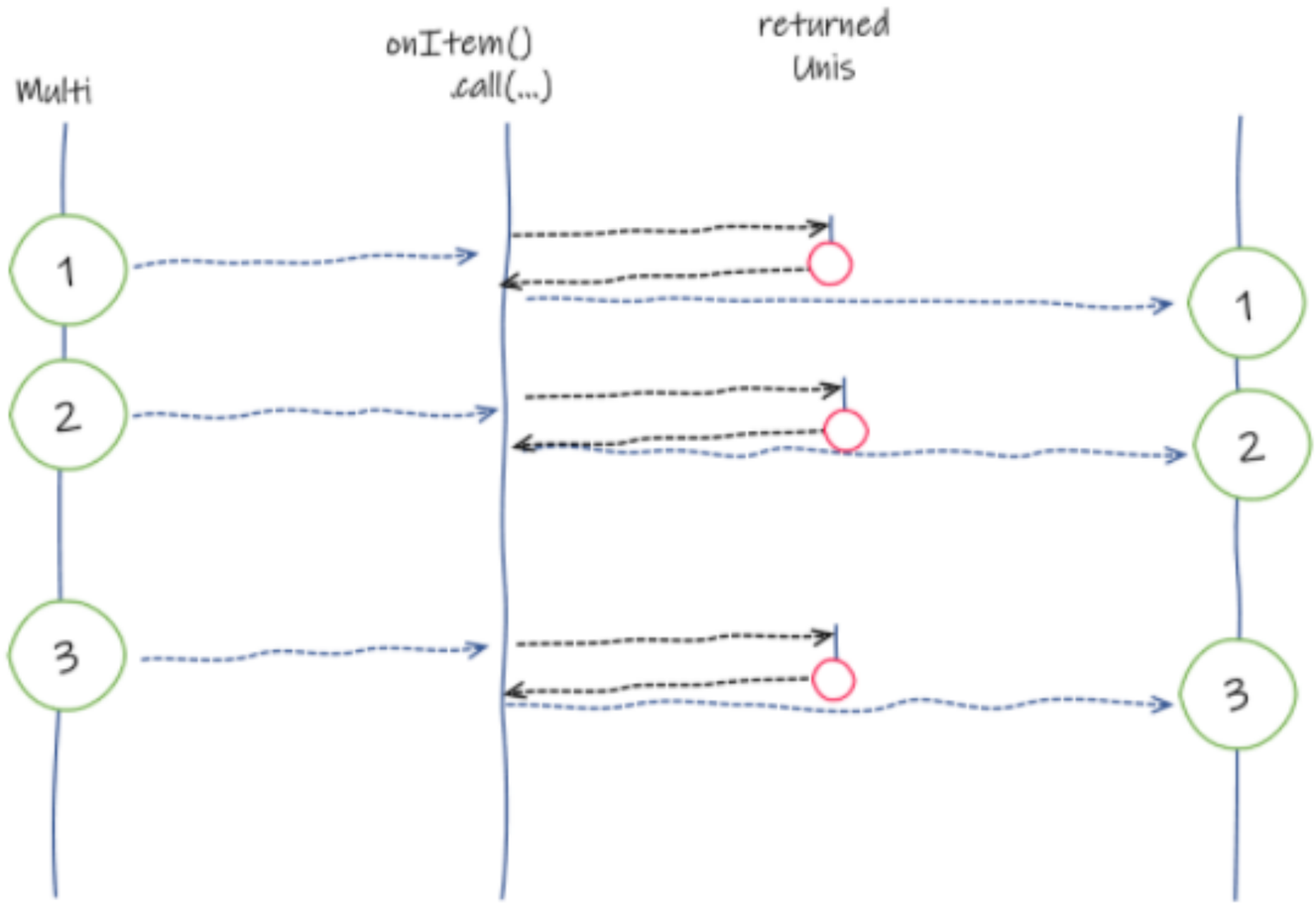
El método invoke

```
Uni<String> u = uni.onItem()
    .invoke(i → System.out.println("Received item: " + i));
Multi<String> m = multi.onItem()
    .invoke(i → System.out.println("Received item: " + i));
```



```
multi
    .onSubscribe().invoke(() → System.out.println("⬇️ Subscribed"))
    .onItem().invoke(i → System.out.println("⬇️ Received item: " + i))
    .onFailure().invoke(f → System.out.println("⬇️ Failed with " + f))
    .onCompletion().invoke(() → System.out.println("⬇️ Completed"))
    .onCancellation().invoke(() → System.out.println("⬆️ Cancelled"))
    .onRequest().invoke(l → System.out.println("⬆️ Requested: " + l));
```

El método call



```
multi
    .onItem().call(i →
        Uni.createFrom().voidItem()
            .onItem().delayIt().by(Duration.ofSeconds(1)
    )
);
```

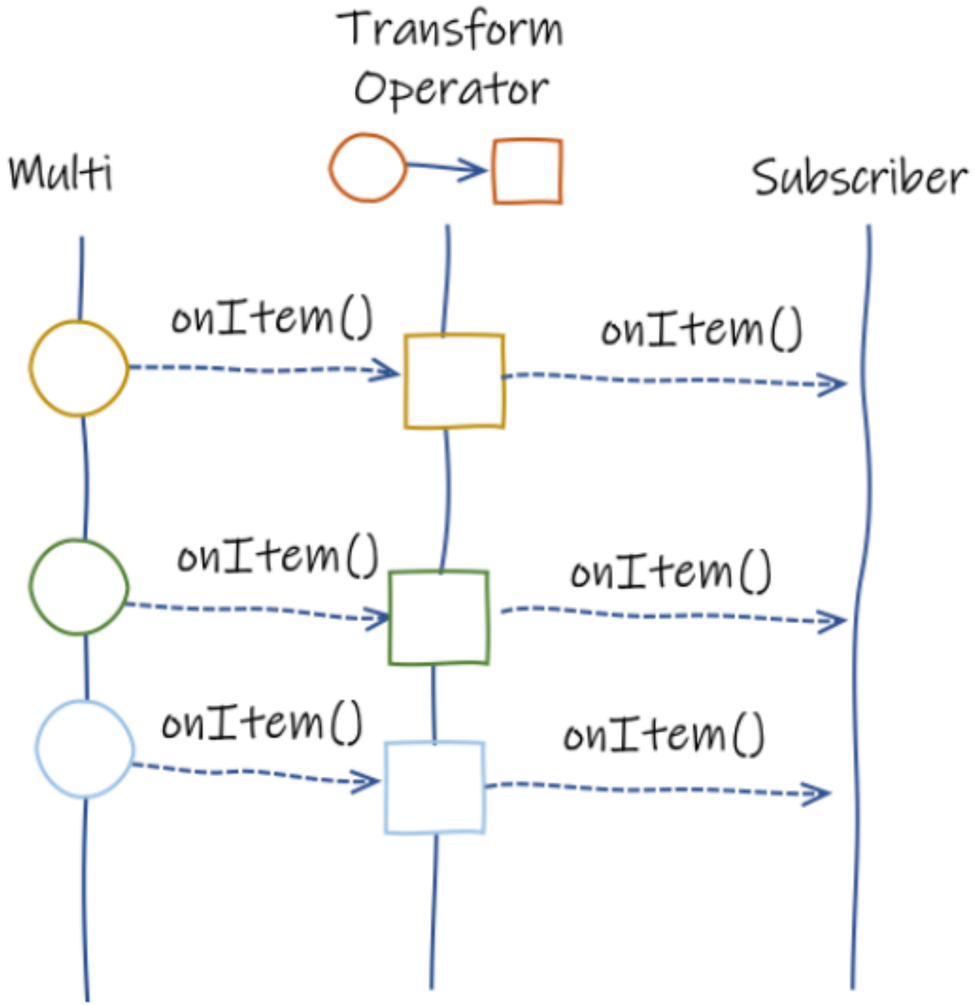
```
multi
    .onCompletion().call(() → resource.close());
```

Resumen

	`INVOKE`	`CALL`
Nature	synchronous	asynchronous
Return type	`void`	`Uni<?>`
Main Use cases	logging, synchronous side-effect	closing resources, flushing data

Transformando Items

Ambos Unis y Multis emiten items.




```
onItem().transform(Function<T, U>)
```

Transformando items producidos por un Uni

```
Uni<String> uni = Uni.createFrom().item("hello");
uni
        .onItem().transform(i → i.toUpperCase())
        .subscribe().with(
        item → System.out.println(item)); // Print HELLO
```

Transformando items producidos por un Multi

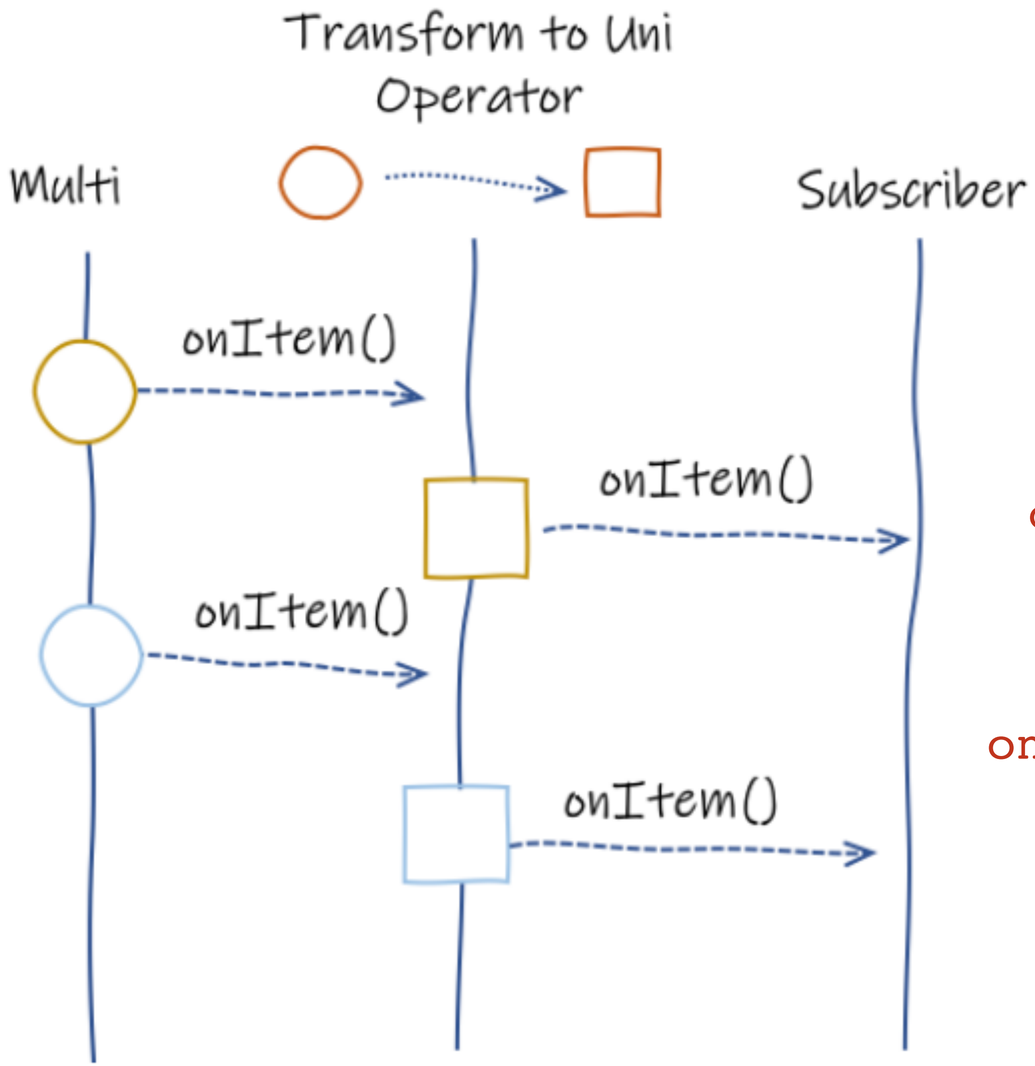
```
Multi<String> m = multi
        .onItem().transform(i → i.toUpperCase());
```

```
Multi<String> multi = Multi.createFrom().items("a", "b", "c");
multi
        .onItem().transform(i → i.toUpperCase())
        .subscribe().with(
            item → System.out.println(item)); // Print A B C
```

Encadenando múltiples transformaciones

```
Uni<String> u = uni
        .onItem().transform(i → i.toUpperCase())
        .onItem().transform(i → i + "!");
```

Transformando ítems a Uni/Multi




```
onItem().transformToUni(Function<T, Uni<O>>
```

```
onItem().transformToMulti(Function<T, Multi<O>>)
```

Transformando un item a Uni

Llamar a un servicio remoto es una acción asíncrona representada por un Uni, como se muestra a continuación:

```
Uni<String> invokeRemoteGreetingService(String name);
```

Para llamar a este servicio, se necesita transformar el ítem recibido desde el primer Uni a un Uni retornado por el servicio:

```
Uni<String> result = uni
    .onItem().transformToUni(name → invokeRemoteGreetingService(name));
```

Este fragmento encadena el primer Uni con otro. El Uni devuelto (resultado) emite el resultado del servicio remoto o una falla si sucede algo malo:

```
Uni<String> uni = Uni.createFrom().item("Cameron");
uni
    .onItem().transformToUni(name → invokeRemoteGreetingService(name))
    .subscribe().with(
            item → System.out.println(item), // Print "Hello Cameron",
            fail → fail.printStackTrace()); // Print the failure stack trace
```

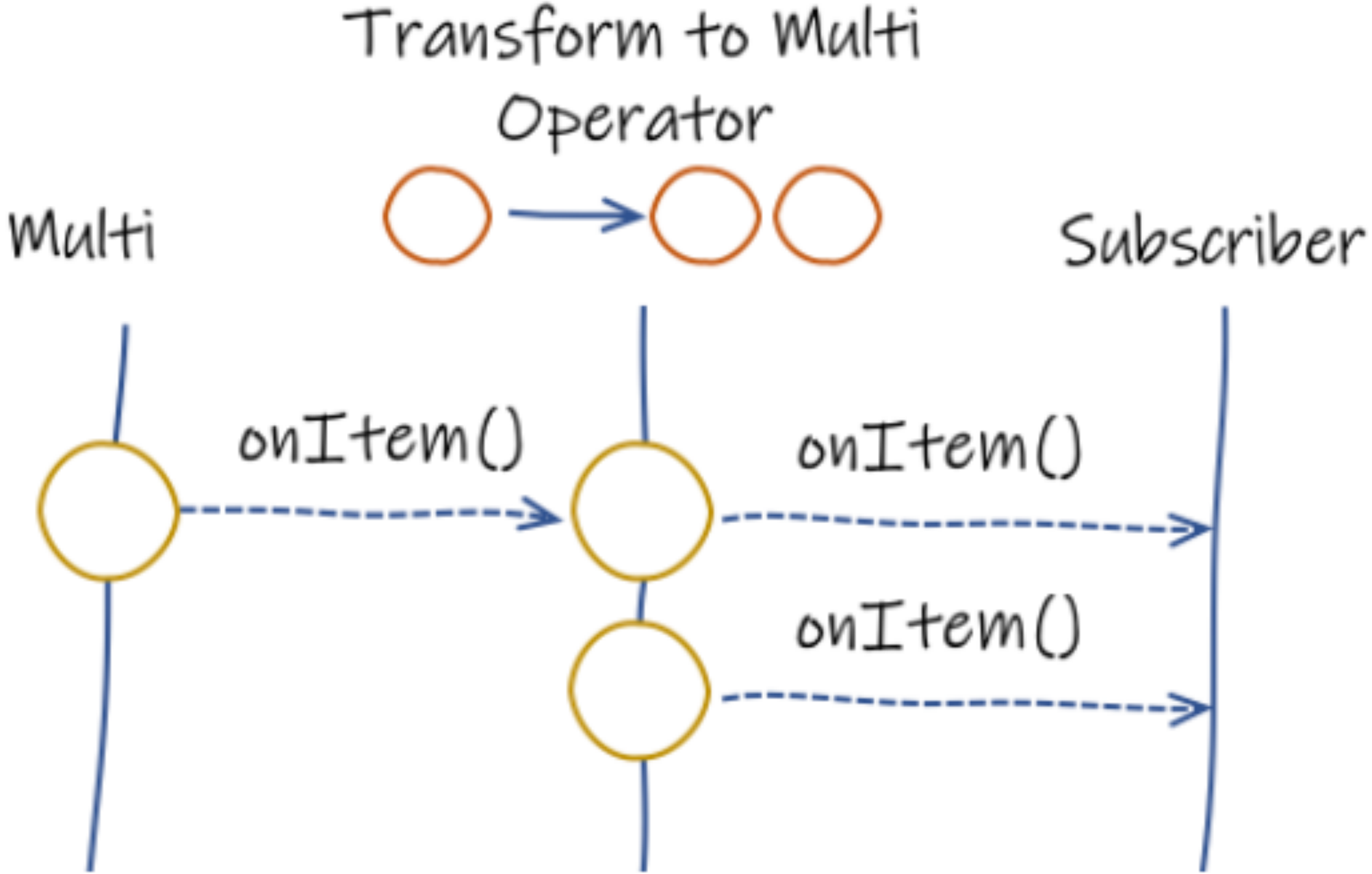
Transformando un item a Multi

El ejemplo previo producía un simple ítem. Tú puedes desear transformar el ítem recibido a un stream el cual es un Multi.

```
Multi<String> result = uni
    .onItem().transformToMulti(item → Multi.createFrom().items(item, item));
```

Este código crea un stream de dos elementos, duplicando el ítem recibido.

```
uni
    .onItem().transformToMulti(item → Multi.createFrom().items(item, item))
    .subscribe().with(
            item → System.out.println(item)); // Called twice
```


Transform to Multi
Operator
Multi
Subscriber
onItem()
onItem()
onItem()

Cuando transformas ítems de Multi a streams, tu necesitas decidir en cual orden serán emitidos: merge vs concatenate.

merge - no preserva el orden y emitirá los ítems desde el stream productor como vengan.

concatenate - Este mantiene y concatena los streams producidos por cada ítem.

`onItem().transformToUniAndMerge` or `onItem().transformToUniAndConcatenate()`

```
Multi<String> merged = multi
    .onItem().transformToUniAndMerge(name → invokeRemoteGreetingService(name));

Multi<String> concat = multi
    .onItem().transformToUniAndConcatenate(name →
invokeRemoteGreetingService(name));
```

`onItem().transformToMultiAndMerge` and `onItem().transformToMultiAndConcatenate`

```
Multi<String> merged = multi
    .onItem().transformToMultiAndMerge(item → someMulti(item));

Multi<String> concat = multi
    .onItem().transformToMultiAndConcatenate(item → someMulti(item));
```

Administrando fallas

Administrando Fallas

- Mutiny nos da diferentes operadores para manejar las fallas.
- Las fallas son eventos terminales enviados por el stream observado, que indica que algo malo sucedió. Después de una falla, no mas ítems son recibidos.
- Cuando un evento es recibido, nosotros podemos:
  - Propagar la falla (por defecto), o
  - Transformar la falla a otra falla, o
  - Recuperarnos de esto y saltar a otro stream, pasando un fallback ítem, o completando, o
  - Reintentar
- Si no manejas el evento de falla, este se propaga hasta que alguien maneje la falla o alcance al subscriptor final.

Observando fallas

```
Uni<String> u = uni
        .onFailure().invoke(failure → log(failure));

Multi<String> m = multi
        .onFailure().invoke(failure → log(failure));
```

También se puede realizar una acción asíncrona usando `onFailure().call(Function<Throwable, Uni<?>)`

Transformando fallas

```
Uni<String> u = uni
        .onFailure().transform(failure →
                new ServiceUnavailableException(failure));
```

Recuperando usando fallback item(s)

```
Uni<String> u1 = uni
        .onFailure().recoverWithItem("hello");

Uni<String> u2 = uni
        .onFailure().recoverWithItem(f → getFallback(f));
```

Completar en caso de falla

```
Multi<String> m = multi
        .onFailure().recoverWithCompletion();
```

Saltando a otro stream

```
Uni<String> u = uni
        .onFailure().recoverWithUni(f → getFallbackUni(f));

Multi<String> m = multi
        .onFailure().recoverWithMulti(f → getFallbackMulti(f));
```

Reintentos en Fallas

Reintentando múltiples veces

```
Uni<String> u = uni
        .onFailure().retry().atMost(3);
Multi<String> m = multi
        .onFailure().retry().atMost(3);
```

`onFailure().retry()`

`.onFailure().retry().indefinitely()`

Introduciendo delays

```
Uni<String> u = uni
        .onFailure().retry()
        .withBackOff(Duration.ofMillis(100), Duration.ofSeconds(1))
        .atMost(3);
```

Decidir reintentar

```
Uni<String> u = uni
        .onFailure().retry()
        .until(f → shouldWeRetry(f));
```