Reactor 3 Reference Guide

Stephane Maldini, Simon Baslé

Version 3.2.23.BUILD-SNAPSHOT, 3.1.4.RELEASE

Table of Contents

Chapter 1. About the Documentation

This section provides a brief overview of Reactor reference documentation. You do not need to read this guide in a linear fashion. Each piece stands on its own, though they often refer to other pieces.

1.1. Latest Version & Copyright Notice

The Reactor reference guide is available as HTML documents. The latest copy is available at https://projectreactor.io/docs/core/release/reference/docs/index.html

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1.2. Contributing to the Documentation

The reference guide is written in Asciidoc format and sources can be found at https://github.com/reactor/reactor-core/tree/master/docs/asciidoc.

If you have an improvement, we will be happy to get a pull request from you!

We recommend that you check out a local copy of the repository, so that you can generate the documentation using the `asciidoctor` gradle task and check the rendering. Some of the sections rely on included files, so GitHub rendering is not always complete.

1.3. Getting Help

There are several ways to reach out for help with Reactor.

- Get in touch with the community on Gitter.
- Ask a question on stackoverflow.com at `project-reactor`.
- Report bugs in Github issues. The following repositories are closely monitored: reactor-core (which covers the essential features) and reactor-addons (which covers reactor-test and adapters issues).

> All of Reactor is open source, including this documentation. If you find problems with the docs or if you just want to improve them, please get involved.

1.4. Where to Go from Here

- Head to Getting Started if you feel like jumping straight into the code.
- If you are new to *Reactive Programming*, though, you should probably start with the Introduction to Reactive Programming.
- If you are familiar with Reactor concepts and are just looking for the right tool for the job but cannot think of a relevant operator, try the Which operator do I need? Appendix.

- In order to dig deeper into the core features of Reactor, head to Reactor Core Features, to learn:
  - More about Reactor's reactive types in the "`Flux`, an Asynchronous Sequence of 0-N Items" and "`Mono`, an Asynchronous 0-1 Result" sections.
  - How to switch threading contexts using a Scheduler.
  - How to handle errors in the Handling Errors section.
- Unit testing? Yes it is possible with the `reactor-test` project! See Testing.
- Programmatically creating a sequence offers a more advanced way of creating of reactive sources.
- Other advanced topics are covered in Advanced Features and Concepts.

Chapter 2. Getting Started

This section contains information that should help you get going with Reactor. It includes the following information:

2.1. Introducing Reactor

Reactor is a fully non-blocking reactive programming foundation for the JVM, with efficient demand management (in the form of managing "backpressure"). It integrates directly with the Java 8 functional APIs, notably `CompletableFuture`, `Stream`, and `Duration`. It offers composable asynchronous sequence APIs `Flux` (for [N] elements) and `Mono` (for [0|1] elements), extensively implementing the [Reactive Streams](https://www.reactive-streams.org/) specification.

Reactor also supports non-blocking inter-process communication with the `reactor-netty` project. Suited for Microservices Architecture, Reactor Netty offers backpressure-ready network engines for HTTP (including Websockets), TCP, and UDP. Reactive Encoding and Decoding are fully supported.

2.2. Prerequisites

Reactor Core runs on `Java 8` and above.

It has a transitive dependency on `org.reactivestreams:reactive-streams:1.0.2`.

Android support:

- Reactor 3 does not officially support or target Android (consider using RxJava 2 if such support is a strong requirement).
- However, it should work fine with Android SDK 26 (Android O) and above.
- We are open to evaluating changes that benefit Android support in a best-effort fashion. However, we cannot make guarantees. Each decision must be made on a case-by-case basis.

2.3. Understanding the BOM

Reactor 3 uses a BOM model since `reactor-core 3.0.4`, with the `Aluminium` release train. This curated list groups artifacts that are meant to work well together, providing the relevant versions despite potentially divergent versioning schemes in these artifacts.

The BOM (Bill of Materials) is itself versioned, using a release train scheme with a codename followed by a qualifier. Here are a few examples:

> Aluminium-RELEASE
> Californium-BUILD-SNAPSHOT
> Aluminium-SR1
> Bismuth-RELEASE
> Californium-SR32

The codenames represent what would traditionally be the MAJOR.MINOR number. They (mostly) come from the Periodic Table of Elements, in increasing alphabetical order.

The qualifiers are (in chronological order):

- `BUILD-SNAPSHOT`
- `M1`..`N`: Milestones or developer previews
- `RELEASE`: The first GA (General Availability) release in a codename series
- `SR1`..`N`: The subsequent GA releases in a codename series (equivalent to PATCH number, SR stands for "Service Release").

2.4. Getting Reactor

As mentioned earlier, the easiest way to use Reactor in your core is to use the BOM and add the relevant dependencies to your project. Note that, when adding such a dependency, you must omit the version so that the version gets picked up from the BOM.

However, if you want to force the use of a specific artifact's version, you can specify it when adding your dependency, as you usually would. You can also forgo the BOM entirely and specify dependencies by their artifact versions.

2.4.1. Maven Installation

The BOM concept is natively supported by Maven. First, you need to import the BOM by adding the following snippet to your `pom.xml`. If the top section (`dependencyManagement`) already exists in your pom, add only the contents.

```
<dependencyManagement> ①
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>Bismuth-RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

① Notice the `dependencyManagement` tag. This is in addition to the regular `dependencies` section.

Next, add your dependencies to the relevant reactor projects, as usual, except without a `<version>`, as shown here:

```
<dependencies>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-core</artifactId> ①
        ②
    </dependency>
    <dependency>
        <groupId>io.projectreactor</groupId>
        <artifactId>reactor-test</artifactId> ③
        <scope>test</scope>
    </dependency>
</dependencies>
```

① Dependency on the core library

② No version tag here

③ `reactor-test` provides facilities to unit test reactive streams

2.4.2. Gradle installation

Gradle has no core support for Maven BOMs, but you can use Spring's gradle-dependency-management plugin.

First, apply the plugin from the Gradle Plugin Portal:

```
plugins {
    id "io.spring.dependency-management" version "1.0.6.RELEASE" ①
}
```

① as of this writing, 1.0.6.RELEASE is the latest version of the plugin. Check for updates.

Then use it to import the BOM:

```
dependencyManagement {
     imports {
          mavenBom "io.projectreactor:reactor-bom:Bismuth-RELEASE"
     }
}
```

Finally add a dependency to your project, without a version number:

```
dependencies {
     compile 'io.projectreactor:reactor-core' ①
}
```

① There is no third `:` separated section for the version. It is taken from the BOM.

2.4.3. Milestones and Snapshots

Milestones and developer previews are distributed through the Spring Milestones repository rather than Maven Central. To add it to your build configuration file, use the following snippet:

Milestones in Maven

```
<repositories>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones Repository</name>
        <url>https://repo.spring.io/milestone</url>
    </repository>
</repositories>
```

For Gradle, use the following snippet:

Milestones in Gradle

```
repositories {
  maven { url 'https://repo.spring.io/milestone' }
  mavenCentral()
}
```

Similarly, snapshots are also available in a separate dedicated repository:

BUILD-SNAPSHOTs in Maven

```
<repositories>
    <repository>
        <id>spring-snapshots</id>
        <name>Spring Snapshot Repository</name>
        <url>https://repo.spring.io/snapshot</url>
    </repository>
</repositories>
```

BUILD-SNAPSHOTs in Gradle

```
repositories {
  maven { url 'https://repo.spring.io/snapshot' }
  mavenCentral()
}
```

Chapter 3. Introduction to Reactive Programming

Reactor is an implementation of the Reactive Programming paradigm, which can be summed up as:

> Reactive programming is an asynchronous programming paradigm concerned with data streams and the propagation of change. This means that it becomes possible to express static (e.g. arrays) or dynamic (e.g. event emitters) data streams with ease via the employed programming language(s).
>
> — https://en.wikipedia.org/wiki/Reactive_programming

As a first step in the direction of reactive programming, Microsoft created the Reactive Extensions (Rx) library in the .NET ecosystem. Then RxJava implemented reactive programming on the JVM. As time went on, a standardization for Java emerged through the **Reactive Streams** effort, a specification that defines a set of interfaces and interaction rules for reactive libraries on the JVM. Its interfaces have been integrated into Java 9 under the parent `Flow` class.

The reactive programming paradigm is often presented in object-oriented languages as an extension of the Observer design pattern. One can also compare the main reactive streams pattern with the familiar Iterator design pattern, as there is a duality to the `Iterable`-`Iterator` pair in all of these libraries. One major difference is that, while an Iterator is **pull**-based, reactive streams are **push**-based.

Using an iterator is an imperative programming pattern, even though the method of accessing values is solely the responsibility of the `Iterable`. Indeed, it is up to the developer to choose when to access the `next()` item in the sequence. In reactive streams, the equivalent of the above pair is `Publisher-Subscriber`. But it is the `Publisher` that notifies the Subscriber of newly available values *as they come*, and this push aspect is the key to being reactive. Also, operations applied to pushed values are expressed declaratively rather than imperatively: the programmer expresses the logic of the computation rather than describing its exact control flow.

In addition to pushing values, the error handling and completion aspects are also covered in a well defined manner. A `Publisher` can push new values to its `Subscriber` (by calling `onNext`) but can also signal an error (by calling `onError`) or completion (by calling `onComplete`). Both errors and completion terminate the sequence. This can be summed up as:

```
onNext x 0..N [onError | onComplete]
```

This approach is very flexible. The pattern supports use cases where there is no value, one value, or n values (including an infinite sequence of values, such as the continuing ticks of a clock).

But let's first consider, why would we need such an asynchronous reactive library in the first place?

3.1. Blocking Can Be Wasteful

Modern applications can reach huge numbers of concurrent users, and, even though the capabilities of modern hardware have continued to improve, performance of modern software is still a key concern.

There are broadly two ways one can improve a program's performance:

1. **parallelize**: use more threads and more hardware resources.
2. **seek more efficiency** in how current resources are used.

Usually, Java developers write programs using blocking code. This practice is fine until there is a performance bottleneck, at which point the time comes to introduce additional threads, running similar blocking code. But this scaling in resource utilization can quickly introduce contention and concurrency problems.

Worse still, blocking wastes resources. If you look closely, as soon as a program involves some latency (notably I/O, such as a database request or a network call), resources are wasted because a thread (or many threads) now sits idle, waiting for data.

So the parallelization approach is not a silver bullet. It is necessary in order to access the full power of the hardware, but it is also complex to reason about and susceptible to resource wasting.

3.2. Asynchronicity to the Rescue?

The second approach (mentioned earlier), seeking more efficiency, can be a solution to the resource wasting problem. By writing *asynchronous*, *non-blocking* code, you let the execution switch to another active task **using the same underlying resources** and later come back to the current process when the asynchronous processing has finished.

But how can you produce asynchronous code on the JVM? Java offers two models of asynchronous programming:

- **Callbacks**: Asynchronous methods do not have a return value but take an extra `callback` parameter (a lambda or anonymous class) that gets called when the result is available. A well known example is Swing's `EventListener` hierarchy.
- **Futures**: Asynchronous methods return a `Future<T>` **immediately**. The asynchronous process computes a `T` value, but the `Future` object wraps access to it. The value is not immediately available, and the object can be polled until the value is available. For instance, `ExecutorService` running `Callable<T>` tasks use `Future` objects.

Are these techniques good enough? Not for every use case, and both approaches have limitations.

Callbacks are hard to compose together, quickly leading to code that is difficult to read and maintain (known as "Callback Hell").

Consider an example: showing the top five favorites from a user on the UI or suggestions if she doesn't have a favorite. This goes through three services (one gives favorite IDs, the second fetches favorite details, and the third offers suggestions with details):

Example of Callback Hell

```
userService.getFavorites(userId, new Callback<List<String>>() { ①
  public void onSuccess(List<String> list) { ②
    if (list.isEmpty()) { ③
      suggestionService.getSuggestions(new Callback<List<Favorite>>() {
        public void onSuccess(List<Favorite> list) { ④
          UiUtils.submitOnUiThread(() -> { ⑤
            list.stream()
                .limit(5)
                .forEach(uiList::show); ⑥
            });
        }

        public void onError(Throwable error) { ⑦
          UiUtils.errorPopup(error);
        }
      });
    } else {
      list.stream() ⑧
          .limit(5)
          .forEach(favId -> favoriteService.getDetails(favId, ⑨
            new Callback<Favorite>() {
              public void onSuccess(Favorite details) {
                UiUtils.submitOnUiThread(() -> uiList.show(details));
              }

              public void onError(Throwable error) {
                UiUtils.errorPopup(error);
              }
            }
          ));
    }
  }

  public void onError(Throwable error) {
    UiUtils.errorPopup(error);
  }
});
```

① We have callback-based services: a `Callback` interface with a method invoked when the async process was successful and one invoked in case of an error.

② The first service invokes its callback with the list of favorite IDs.

③ If the list is empty, we must go to the `suggestionService`.

④ The `suggestionService` gives a `List<Favorite>` to a second callback.

⑤ Since we deal with a UI, we need to ensure our consuming code will run in the UI thread.

⑥ We use Java 8 `Stream` to limit the number of suggestions processed to five, and we show them in a graphical list in the UI.

⑦ At each level, we deal with errors the same way: show them in a popup.

⑧ Back to the favorite ID level. If the service returned a full list, then we need to go to the `favoriteService` to get detailed `Favorite` objects. Since we want only five, we first stream the list of IDs to limit it to five.

⑨ Once again, a callback. This time we get a fully-fledged `Favorite` object that we push to the UI inside the UI thread.

That is a lot of code, and it is a bit hard to follow and has repetitive parts. Consider its equivalent in Reactor:

Example of Reactor code equivalent to callback code

```
userService.getFavorites(userId) ①
           .flatMap(favoriteService::getDetails) ②
           .switchIfEmpty(suggestionService.getSuggestions()) ③
           .take(5) ④
           .publishOn(UiUtils.uiThreadScheduler()) ⑤
           .subscribe(uiList::show, UiUtils::errorPopup); ⑥
```

① We start with a flow of favorite IDs.

② We *asynchronously transform* these into detailed `Favorite` objects (`flatMap`). We now have a flow of `Favorite`.

③ In case the flow of `Favorite` is empty, we switch to a fallback through the `suggestionService`.

④ We are only interested in, at most, five elements from the resulting flow.

⑤ At the end, we want to process each piece of data in the UI thread.

⑥ We trigger the flow by describing what to do with the final form of the data (show it in a UI list) and what to do in case of an error (show a popup).

What if you want to ensure the favorite IDs are retrieved in less than 800ms or, if it takes longer, get them from a cache? In the callback-based code, that is a complicated task. In Reactor it becomes as easy as adding a `timeout` operator in the chain:

Example of Reactor code with timeout and fallback

```
userService.getFavorites(userId)
           .timeout(Duration.ofMillis(800)) ①
           .onErrorResume(cacheService.cachedFavoritesFor(userId)) ②
           .flatMap(favoriteService::getDetails) ③
           .switchIfEmpty(suggestionService.getSuggestions())
           .take(5)
           .publishOn(UiUtils.uiThreadScheduler())
           .subscribe(uiList::show, UiUtils::errorPopup);
```

① If the part above emits nothing for more than 800ms, propagate an error.

② In case of an error, fall back to the `cacheService`.

③ The rest of the chain is similar to the previous example.

Futures are a bit better than callbacks, but they still do not do well at composition, despite the improvements brought in Java 8 by `CompletableFuture`. Orchestrating multiple futures together is doable but not easy. Also, `Future` has other problems: It is easy to end up with another blocking situation with `Future` objects by calling the `get()` method, they do not support lazy computation and they lack support for multiple values and advanced error handling.

Consider another example: We get a list of IDs from which we want to fetch a name and a statistic and combine these pair-wise, all of it asynchronously.

Example of `CompletableFuture` combination

```
CompletableFuture<List<String>> ids = ifhIds(); ①

CompletableFuture<List<String>> result = ids.thenComposeAsync(l -> { ②
    Stream<CompletableFuture<String>> zip =
            l.stream().map(i -> { ③
                CompletableFuture<String> nameTask = ifhName(i); ④
                CompletableFuture<Integer> statTask = ifhStat(i); ⑤

                return nameTask.thenCombineAsync(statTask, (name, stat) -> "Name " +
name + " has stats " + stat); ⑥
            });
    List<CompletableFuture<String>> combinationList =
zip.collect(Collectors.toList()); ⑦
    CompletableFuture<String>[] combinationArray = combinationList.toArray(new
CompletableFuture[combinationList.size()]);

    CompletableFuture<Void> allDone = CompletableFuture.allOf(combinationArray); ⑧
    return allDone.thenApply(v -> combinationList.stream()
            .map(CompletableFuture::join) ⑨
            .collect(Collectors.toList()));
});

List<String> results = result.join(); ⑩
assertThat(results).contains(
        "Name NameJoe has stats 103",
        "Name NameBart has stats 104",
        "Name NameHenry has stats 105",
        "Name NameNicole has stats 106",
        "Name NameABSLAJNFOAJNFOANFANSF has stats 121");
```

① We start off with a future that gives us a list of `id` values to process.

② We want to start some deeper asynchronous processing once we get the list.

③ For each element in the list:

④ Asynchronously get the associated name.

⑤ Asynchronously get the associated task.

⑥ Combine both results.

⑦ We now have a list of futures that represent all the combination tasks. In order to execute these tasks, we need to convert the list to an array.

⑧ Pass the array to `CompletableFuture.allOf`, which outputs a `Future` that completes when all tasks have completed.

⑨ The tricky bit is that `allOf` returns `CompletableFuture<Void>`, so we reiterate over the list of futures, collecting their results via `join()` (which here doesn't block since `allOf` ensures the futures are all done).

⑩ Once the whole asynchronous pipeline has been triggered, we wait for it to be processed and return the list of results that we can assert.

Since Reactor has more combination operators out of the box, this process can be simplified:

Example of Reactor code equivalent to future code

```
Flux<String> ids = ifhrIds(); ①

Flux<String> combinations =
        ids.flatMap(id -> { ②
            Mono<String> nameTask = ifhrName(id); ③
            Mono<Integer> statTask = ifhrStat(id); ④

            return nameTask.zipWith(statTask, ⑤
                    (name, stat) -> "Name " + name + " has stats " + stat);
        });

Mono<List<String>> result = combinations.collectList(); ⑥

List<String> results = result.block(); ⑦
assertThat(results).containsExactly( ⑧
        "Name NameJoe has stats 103",
        "Name NameBart has stats 104",
        "Name NameHenry has stats 105",
        "Name NameNicole has stats 106",
        "Name NameABSLAJNFOAJNFOANFANSF has stats 121"
);
```

① This time, we start from an asynchronously provided sequence of `ids` (a `Flux<String>`).

② For each element in the sequence, we asynchronously process it (inside the function that is the body `flatMap` call) twice.

③ Get the associated name.

④ Get the associated statistic.

⑤ Asynchronously combine the 2 values.

⑥ Aggregate the values into a `List` as they become available.

⑦ In production, we would continue working with the `Flux` asynchronously by further combining it or subscribing to it. Most probably, we would return the `result` `Mono`. Since we are in a test, we block, waiting for the processing to finish instead, and then directly return the aggregated list of values.

⑧ Assert the result.

These perils of Callback and Future are similar and are what reactive programming addresses with the `Publisher-Subscriber` pair.

3.3. From Imperative to Reactive Programming

Reactive libraries such as Reactor aim to address these drawbacks of "classic" asynchronous approaches on the JVM while also focusing on a few additional aspects:

- **Composability** and **readability**
- Data as a **flow** manipulated with a rich vocabulary of **operators**
- Nothing happens until you **subscribe**
- **Backpressure** or *the ability for the consumer to signal the producer that the rate of emission is too high*
- **High level** but **high value** abstraction that is *concurrency-agnostic*

3.3.1. Composability and Readability

By composability, we mean the ability to orchestrate multiple asynchronous tasks, using results from previous tasks to feed input to subsequent ones or executing several tasks in a fork-join style, as well as reusing asynchronous tasks as discrete components in a higher-level system.

The ability to orchestrate tasks is tightly coupled to the readability and maintainability of code. As the layers of asynchronous processes increase in both number and complexity, being able to compose and read code becomes increasingly difficult. As we saw, the callback model is simple, but one of its main drawbacks is that, for complex processes, you need to have a callback executed from a callback, itself nested inside another callback, and so on. That mess is known as **Callback Hell**. As you can guess (or know from experience), such code is pretty hard to go back to and reason about.

Reactor offers rich composition options, wherein code mirrors the organization of the abstract process, and everything is generally kept at the same level (nesting is minimized).

3.3.2. The Assembly Line Analogy

You can think of data processed by a reactive application as moving through an assembly line. Reactor is both the conveyor belt and the workstations. The raw material pours from a source (the original `Publisher`) and ends up as a finished product ready to be pushed to the consumer (or `Subscriber`).

The raw material can go through various transformations and other intermediary steps or be part of a larger assembly line that aggregates intermediate pieces together. If there is a glitch or clogging at one point (perhaps boxing the products takes a disproportionately long time), the afflicted workstation can signal upstream to limit the flow of raw material.

3.3.3. Operators

In Reactor, operators are the workstations in our assembly analogy. Each operator adds behavior to a `Publisher` and wraps the previous step's `Publisher` into a new instance. The whole chain is thus linked, such that data originates from the first `Publisher` and moves down the chain, transformed by each link. Eventually, a `Subscriber` finishes the process. Remember that nothing happens until a `Subscriber` subscribes to a `Publisher`, as we see shortly.

> Understanding that operators create new instances can help you avoid a common mistake that would lead you to believe that an operator you used in your chain is not being applied. See this item in the FAQ.

While the Reactive Streams specification does not specify operators at all, one of the best added values of reactive libraries such as Reactor is the rich vocabulary of operators that they provide. These cover a lot of ground, from simple transformation and filtering to complex orchestration and error handling.

3.3.4. Nothing Happens Until You `subscribe()`

In Reactor, when you write a `Publisher` chain, data does not start pumping into it by default. Instead, you create an abstract description of your asynchronous process (which can help with reusability and composition).

By the act of **subscribing**, you tie the `Publisher` to a `Subscriber`, which triggers the flow of data in the whole chain. This is achieved internally by a single `request` signal from the `Subscriber` that is propagated upstream, all the way back to the source `Publisher`.

3.3.5. Backpressure

Propagating signals upstream is also used to implement **backpressure**, which we described in the assembly line analogy as a feedback signal sent up the line when a workstation processes more slowly than an upstream workstation.

The real mechanism defined by the Reactive Streams specification is pretty close to the analogy: a subscriber can work in *unbounded* mode and let the source push all the data at its fastest achievable rate or it can use the `request` mechanism to signal the source that it is ready to process at most `n` elements.

Intermediate operators can also change the request in-transit. Imagine a `buffer` operator that groups elements in batches of 10. If the subscriber requests 1 buffer, it is acceptable for the source to produce 10 elements. Some operators also implement **prefetching** strategies, which avoids `request(1)` round-trips and is beneficial if producing the elements before they are requested is not too costly.

This transforms the push model into a **push-pull hybrid** where the downstream can pull n elements from upstream if they are readily available. But if the elements are not ready, they get pushed by the upstream whenever they are produced.

3.3.6. Hot vs Cold

In the Rx family of reactive libraries, one can distinguish two broad categories of reactive sequences: **hot** and **cold**. This distinction mainly has to do with how the reactive stream reacts to subscribers:

- A **Cold** sequence starts anew for each `Subscriber`, including at the source of data. For example if the source wraps an HTTP call, a new HTTP request is made for each subscription.
- A **Hot** sequence does not start from scratch for each `Subscriber`. Rather, late subscribers receive signals emitted *after* they subscribed. Note, however, that some hot reactive streams can cache or replay the history of emissions totally or partially. From a general perspective, a hot sequence can even emit when no subscriber is listening (an exception to the "nothing happens before you subscribe" rule).

For more information on hot vs cold in the context of Reactor, see this reactor-specific section.

Chapter 4. Reactor Core Features

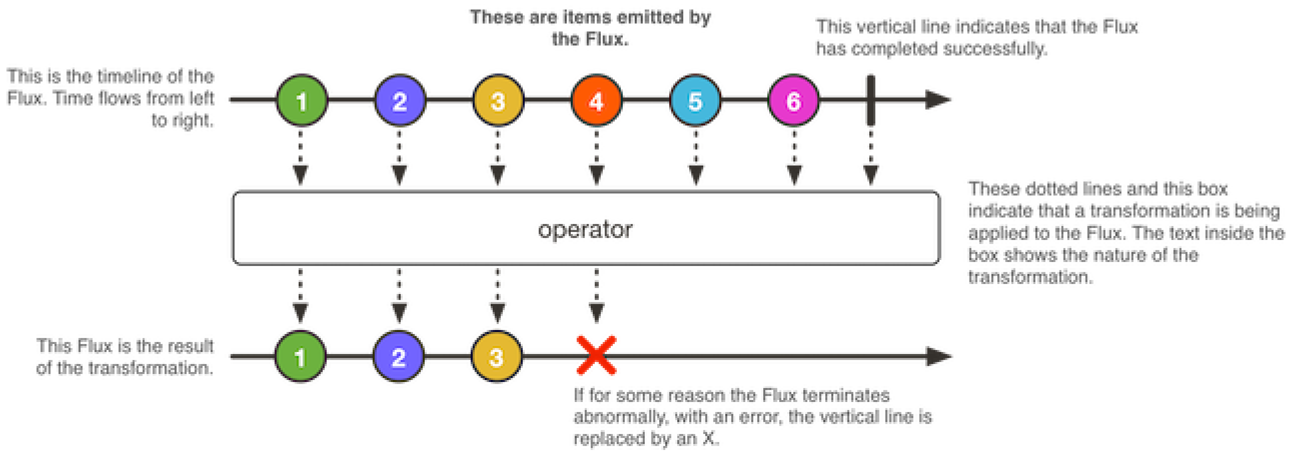
The Reactor project main artifact is `reactor-core`, a reactive library that focuses on the Reactive Streams specification and targets Java 8.

Reactor introduces composable reactive types that implement `Publisher` but also provide a rich vocabulary of operators: `Flux` and `Mono`. A `Flux` object represents a reactive sequence of 0..N items, while a `Mono` object represents a single-value-or-empty (0..1) result.

This distinction carries a bit of semantic information into the type, indicating the rough cardinality of the asynchronous processing. For instance, an HTTP request produces only one response, so there is not much sense in doing a `count` operation. Expressing the result of such an HTTP call as a `Mono<HttpResponse>` thus makes more sense than expressing it as a `Flux<HttpResponse>`, as it offers only operators that are relevant to a context of zero items or one item.

Operators that change the maximum cardinality of the processing also switch to the relevant type. For instance, the `count` operator exists in `Flux`, but it returns a `Mono<Long>`.

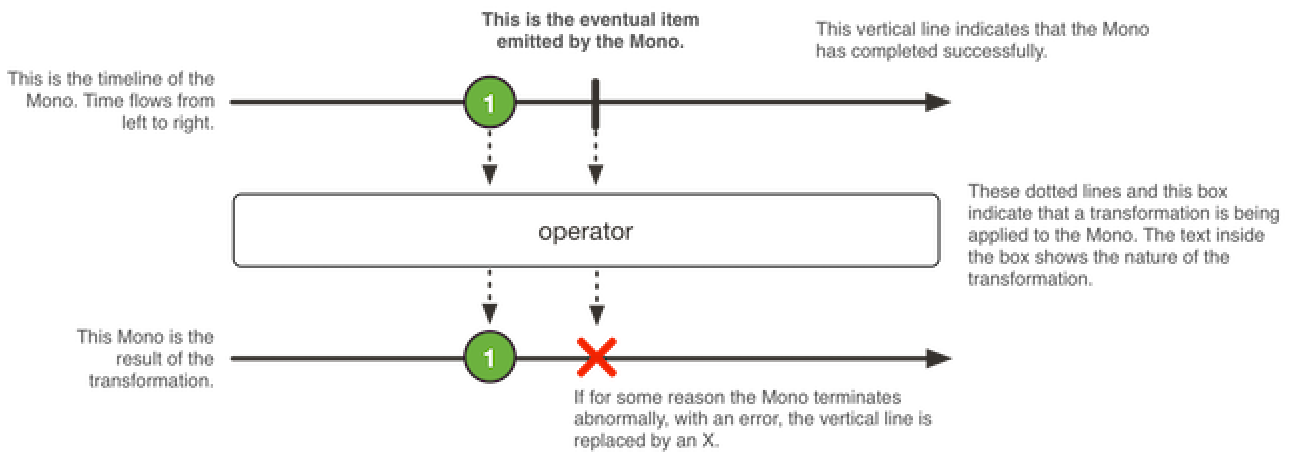
4.1. `Flux`, an Asynchronous Sequence of 0-N Items


These are items emitted by the Flux.
This vertical line indicates that the Flux has completed successfully.
This is the timeline of the Flux. Time flows from left to right.
1
2
3
4
5
6
operator
These dotted lines and this box indicate that a transformation is being applied to the Flux. The text inside the box shows the nature of the transformation.
This Flux is the result of the transformation.
1
2
3
If for some reason the Flux terminates abnormally, with an error, the vertical line is replaced by an X.

A `Flux<T>` is a standard `Publisher<T>` representing an asynchronous sequence of 0 to N emitted items, optionally terminated by either a completion signal or an error. As in the Reactive Streams spec, these 3 types of signal translate to calls to a downstream Subscriber’s `onNext`, `onComplete` or `onError` methods.

With this large scope of possible signals, `Flux` is the general-purpose reactive type. Note that all events, even terminating ones, are optional: no `onNext` event but an `onComplete` event represents an *empty* finite sequence, but remove the `onComplete` and you have an *infinite* empty sequence (not particularly useful, except for tests around cancellation). Similarly, infinite sequences are not necessarily empty. For example, `Flux.interval(Duration)` produces a `Flux<Long>` that is infinite and emits regular ticks from a clock.

4.2. `Mono`, an Asynchronous 0-1 Result


This is the eventual item emitted by the Mono.
This vertical line indicates that the Mono has completed successfully.
This is the timeline of the Mono. Time flows from left to right.
1
operator
These dotted lines and this box indicate that a transformation is being applied to the Mono. The text inside the box shows the nature of the transformation.
This Mono is the result of the transformation.
1
If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X.

A `Mono<T>` is a specialized `Publisher<T>` that emits at most one item and then optionally terminates with an `onComplete` signal or an `onError` signal.

It offers only a subset of the operators that are available for a `Flux`, and some operators (notably those that combine the `Mono` with another `Publisher`) switch to a `Flux`.
For example, `Mono#concatWith(Publisher)` returns a `Flux` while `Mono#then(Mono)` returns another `Mono`.

Note that a `Mono` can be used to represent no-value asynchronous processes that only have the concept of completion (similar to a `Runnable`). To create one, use an empty `Mono<Void>`.

4.3. Simple Ways to Create a Flux or Mono and Subscribe to It

The easiest way to get started with `Flux` and `Mono` is to use one of the numerous factory methods found in their respective classes.

For instance, to create a sequence of `String`, you can either enumerate them or put them in a collection and create the Flux from it, as follows:

```
Flux<String> seq1 = Flux.just("foo", "bar", "foobar");

List<String> iterable = Arrays.asList("foo", "bar", "foobar");
Flux<String> seq2 = Flux.fromIterable(iterable);
```

Other examples of factory methods include the following:

```
Mono<String> noData = Mono.empty(); ①

Mono<String> data = Mono.just("foo");

Flux<Integer> numbersFromFiveToSeven = Flux.range(5, 3); ②
```

① Notice the factory method honors the generic type even though it has no value.

② The first parameter is the start of the range, while the second parameter is the number of items to produce.

When it comes to subscribing, `Flux` and `Mono` make use of Java 8 lambdas. You have a wide choice of `.subscribe()` variants that take lambdas for different combinations of callbacks, as shown in the following method signatures:

Lambda-based subscribe variants for `Flux`

```
subscribe(); ①

subscribe(Consumer<? super T> consumer); ②

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer); ③

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer); ④

subscribe(Consumer<? super T> consumer,
          Consumer<? super Throwable> errorConsumer,
          Runnable completeConsumer,
          Consumer<? super Subscription> subscriptionConsumer); ⑤
```

① Subscribe and trigger the sequence.

② Do something with each produced value.

③ Deal with values but also react to an error.

④ Deal with values and errors but also execute some code when the sequence successfully completes.

⑤ Deal with values and errors and successful completion but also do something with the `Subscription` produced by this `subscribe` call.

These variants return a reference to the subscription that you can use to cancel the subscription when no more data is needed. Upon cancellation, the source should stop producing values and clean up any resources it created. This cancel and clean-up behavior is represented in Reactor by the general-purpose `Disposable` interface.

4.3.1. `subscribe` Method Examples

This section contains minimal examples of each of the five signatures for the `subscribe` method. The following code shows an example of the basic method with no arguments:

```
Flux<Integer> ints = Flux.range(1, 3); ①
ints.subscribe(); ②
```

① Set up a `Flux` that produces three values when a subscriber attaches.

② Subscribe in the simplest way.

The preceding code produces no visible output, but it does work. The `Flux` produces three values. If we provide a lambda, we can make the values visible. The next example for the `subscribe` method shows one way to make the values appear:

```
Flux<Integer> ints = Flux.range(1, 3); ①
ints.subscribe(i -> System.out.println(i)); ②
```

① Set up a `Flux` that produces three values when a subscriber attaches.

② Subscribe with a subscriber that will print the values.

The preceding code produces the following output:

```
1
2
3
```

To demonstrate the next signature, we intentionally introduce an error, as shown in the following example:

```
Flux<Integer> ints = Flux.range(1, 4) ①
      .map(i -> { ②
        if (i <= 3) return i; ③
        throw new RuntimeException("Got to 4"); ④
      });
ints.subscribe(i -> System.out.println(i), ⑤
      error -> System.err.println("Error: " + error));
```

① Set up a Flux that produces four values when a subscriber attaches.

② We need a map so that we can handle some values differently.

③ For most values, return the value.

④ For one value, force an error.

⑤ Subscribe with a subscriber that includes an error handler.

We now have two lambda expressions: one for the content we expect and one for errors. The preceding code produces the following output:

```
1
2
3
Error: java.lang.RuntimeException: Got to 4
```

The next signature of the `subscribe` method includes both an error handler and a handler for completion events, as shown in the following example:

```
Flux<Integer> ints = Flux.range(1, 4); ①
ints.subscribe(i -> System.out.println(i),
    error -> System.err.println("Error " + error),
    () -> System.out.println("Done")); ②
```

① Set up a Flux that produces four values when a Subscriber attaches.

② Subscribe with a Subscriber that includes a handler for completion events.

Error signals and completion signals are both terminal events and are exclusive of one another (you never get both). To make the completion consumer work, we must take care not to trigger an error.

The completion callback has no input, as represented by an empty pair of parentheses: it matches the `run` method in the `Runnable` interface. The preceding code produces the following output:

```
1
2
3
4
Done
```

The last signature of the `subscribe` method includes a `Consumer<Subscription>`. That variant requires you to do something with the `Subscription` (perform a `request(long)` on it, or `cancel()` it), otherwise the `Flux` will just hang:

```
Flux<Integer> ints = Flux.range(1, 4);
ints.subscribe(i -> System.out.println(i),
    error -> System.err.println("Error " + error),
    () -> System.out.println("Done"),
    sub -> sub.request(10)); ①
```

① When we subscribe we receive a `Subscription`. Signal that we want up to `10` elements from the source (which will actually emit 4 elements and complete).

4.3.2. Cancelling a `subscribe()` with its `Disposable`

All these lambda-based variants of `subscribe()` have a `Disposable` return type. In this case, the `Disposable` interface represents the fact that the subscription can be *cancelled*, by calling its `dispose()` method.

For a `Flux` or `Mono`, cancellation is a signal that the source should stop producing elements. However, it is NOT guaranteed to be immediate: some sources might produce elements so fast that they could complete even before receiving the cancel instruction.

Some utilities around `Disposable` are available in the `Disposables` class. Among these, `Disposables.swap()` creates a `Disposable` wrapper that allows you to atomically cancel and replace a concrete `Disposable`. This can be useful, for instance, in a UI scenario where you want to cancel a request and replace it with a new one whenever the user clicks on a button. Disposing the wrapper itself closes it, disposing the current concrete value and all future attempted replacements.

Another interesting utility is `Disposables.composite(…)`. This composite allows to collect several `Disposable`, for instance multiple in-flight request associated with a service call, and dispose all of them at once later on. Once the composite's `dispose()` method has been called, any attempt at adding another `Disposable` immediately disposes it.

4.3.3. Alternative to lambdas: `BaseSubscriber`

There is an additional `subscribe` method that is more generic and takes a full-blown `Subscriber` rather than composing one out of lambdas. In order to help you writing such a `Subscriber`, we provide an extendable class called `BaseSubscriber`.

Let's implement one of these, we'll call it a `SampleSubscriber`. The following example shows how it would be attached to a `Flux`:

```
SampleSubscriber<Integer> ss = new SampleSubscriber<Integer>();
Flux<Integer> ints = Flux.range(1, 4);
ints.subscribe(i -> System.out.println(i),
    error -> System.err.println("Error " + error),
    () -> {System.out.println("Done");},
    s -> s.request(10));
ints.subscribe(ss);
```

Now let's have a look at what `SampleSubscriber` could look like, as a minimalistic implementation of a `BaseSubscriber`:

```
package io.projectreactor.samples;

import org.reactivestreams.Subscription;

import reactor.core.publisher.BaseSubscriber;

public class SampleSubscriber<T> extends BaseSubscriber<T> {

    public void hookOnSubscribe(Subscription subscription) {
        System.out.println("Subscribed");
        request(1);
    }

    public void hookOnNext(T value) {
        System.out.println(value);
        request(1);
    }
}
```

The SampleSubscriber class extends `BaseSubscriber`, which is the recommended abstract class for user-defined `Subscribers` in Reactor. The class offers hooks that can be overridden to tune the subscriber's behavior. By default, it will trigger an unbounded request and behave exactly like `subscribe()`. However, extending `BaseSubscriber` is much more useful when you want a custom request amount.

For custom request amount, the bare minimum is to implement `hookOnSubscribe(Subscription subscription)` and `hookOnNext(T value)` like we did. In our case, the `hookOnSubscribe` method prints a statement to standard out and makes the first request. Then the `hookOnNext` method prints a statement and performs additional requests, one request at a time.

The `SampleSubscriber` class produces the following output:

```
Subscribed
1
2
3
4
```

`BaseSubscriber` also offers a `requestUnbounded()` method to switch to unbounded mode (equivalent to `request(Long.MAX_VALUE)`), as well as a `cancel()` method.

It has additional hooks: `hookOnComplete`, `hookOnError`, `hookOnCancel`, and `hookFinally` (which is always called when the sequence terminates, with the type of termination passed in as a `SignalType` parameter)

You almost certainly want to implement the `hookOnError`, `hookOnCancel`, and `hookOnComplete` methods. You may also want to implement the `hookFinally` method. `SampleSubscribe` is the absolute minimum implementation of a `Subscriber` *that performs bounded requests.*

4.3.4. On Backpressure, and ways to reshape requests

When implementing backpressure in Reactor, the way consumer pressure is propagated back to the source is by sending a `request` to the upstream operator. The sum of current requests is sometimes referenced to as the current "demand", or "pending request". Demand is capped at `Long.MAX_VALUE`, representing an unbounded request ("produce as fast as you can", basically disabling backpressure).

The first request comes from the final subscriber, at subscription time, yet the most direct ways of subscribing all immediately trigger an unbounded request of `Long.MAX_VALUE`:

- `subscribe()` and most of its lambda-based variants (to the exception of the one that has a Consumer<Subscription>)
- `block()`, `blockFirst()` and `blockLast()`
- iterating over a `toIterable()`/`toStream()`

The simplest way of customizing the original request is to `subscribe` with a `BaseSubscriber` with the `hookOnSubscribe` method overridden:

```
Flux.range(1, 10)
    .doOnRequest(r -> System.out.println("request of " + r))
    .subscribe(new BaseSubscriber<Integer>() {

      @Override
      public void hookOnSubscribe(Subscription subscription) {
        request(1);
      }

      @Override
      public void hookOnNext(Integer integer) {
        System.out.println("Cancelling after having received " + integer);
        cancel();
      }
    });
```

This snippets prints out:

```
request of 1
Cancelling after having received 1
```

⚠ When manipulating a request, you must be careful to produce enough demand for the sequence to advance or your Flux will get "stuck". That is why `BaseSubscriber` defaults to an unbounded request in `hookOnSubscribe`. When overriding this hook, you should usually call `request` at least once.

Operators changing the demand from downstream

One thing to keep in mind is that demand expressed at the subscribe level **can** be reshaped by each operator in the chain upstream. A textbook case is the `buffer(N)` operator: if it receives a `request(2)`, it is interpreted as a demand for **two full buffers**. As a consequence, since buffers need `N` elements to be considered full, the `buffer` operator reshapes the request to `2 x N`.

You might also have noticed that some operators have variants that take an `int` input parameter called `prefetch`. This is another category of operators that modify the downstream request. These are usually operators that deal with inner sequences, deriving a `Publisher` from each incoming element (like `flatMap`).

Prefetch is a way to tune the initial request made on these inner sequences. If unspecified, most of these operators start with a demand of `32`.

These operators usually also implement a **replenishing optimization**: once the operator has seen 75% of the prefetch request fulfilled, it re-requests 75% from upstream. This is a heuristic optimization made so that these operators proactively anticipate the upcoming requests.

Finally, a couple of operators are made to directly let you tune the request: `limitRate` and `limitRequest`.

`limitRate(N)` splits the downstream requests so that they are propagated upstream in smaller batches. For instance, a request of `100` made to `limitRate(10)` would result in *at most* `10` requests of `10` being propagated to the upstream. Note that in this form `limitRate` actually implements the replenishing optimization discussed above.

The operator has a variant that also lets you tune the replenishing amount, referred to as the `lowTide` in the variant: `limitRate(highTide, lowTide)`. Choosing a `lowTide` of `0` will result in **strict** batches of `highTide` requests, instead of batches further reworked by the replenishing strategy.

`limitRequest(N)` on the other hand **caps** the downstream request to a maximum total demand. It adds up requests up to `N`. If a single `request` doesn't make the total demand overflow over `N`, that particular request is wholly propagated upstream. After that amount has been emitted by the source, `limitRequest` will consider the sequence complete and send an `onComplete` signal downstream and cancel the source.

4.4. Programmatically creating a sequence

In this section, we introduce the creation of a `Flux` or a `Mono` by programmatically defining its associated events (`onNext`, `onError`, and `onComplete`). All these methods share the fact that they expose an API to trigger the events that we call a **sink**. There are actually a few sink variants, which we'll get to shortly.

4.4.1. Synchronous `generate`

The simplest form of programmatic creation of a `Flux` is through the `generate` method, which takes a generator function.

This is for **synchronous** and **one-by-one** emissions, meaning that the sink is a `SynchronousSink` and that its `next()` method can only be called at most once per callback invocation. You can then additionally call `error(Throwable)` or `complete()`, but this is optional.

The most useful variant is probably the one that also lets you keep a state that you can refer to in your sink usage to decide what to emit next. The generator function then becomes a `BiFunction<S, SynchronousSink<T>, S>`, with `<S>` the type of the state object. You have to provide a `Supplier<S>` for the initial state, and your generator function now returns a new state on each round.

For instance, you could use an `int` as the state:

Example of state-based `generate`

```
Flux<String> flux = Flux.generate(
    () -> 0, ①
    (state, sink) -> {
      sink.next("3 x " + state + " = " + 3*state); ②
      if (state == 10) sink.complete(); ③
      return state + 1; ④
    });
```

① We supply the initial state value of 0.

② We use the state to choose what to emit (a row in the multiplication table of 3).

③ We also use it to choose when to stop.

④ We return a new state that we use in the next invocation (unless the sequence terminated in this one).

The code above generates the table of 3, as the following sequence:

```
3 x 0 = 0
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30
```

You can also use a mutable `<S>`. The example above could for instance be rewritten using a single `AtomicLong` as the state, mutating it on each round:

Mutable state variant

```
Flux<String> flux = Flux.generate(
    AtomicLong::new, ①
    (state, sink) -> {
      long i = state.getAndIncrement(); ②
      sink.next("3 x " + i + " = " + 3*i);
      if (i == 10) sink.complete();
      return state; ③
    });
```

① This time, we generate a mutable object as the state.

② We mutate the state here.

③ We return the **same** instance as the new state.

If your state object needs to clean up some resources, use the `generate(Supplier<S>, BiFunction, Consumer<S>)` variant to clean up the last state instance.

Here is an example of using the generate method that includes a `Consumer`:

```
Flux<String> flux = Flux.generate(
    AtomicLong::new,
      (state, sink) -> { ①
      long i = state.getAndIncrement(); ②
      sink.next("3 x " + i + " = " + 3*i);
      if (i == 10) sink.complete();
      return state; ③
    }, (state) -> System.out.println("state: " + state)); ④
}
```

① Again, we generate a mutable object as the state.

② We mutate the state here.

③ We return the **same** instance as the new state.

④ We see the last state value (11) as the output of this `Consumer` lambda.

In the case of the state containing a database connection or other resource that needs to be handled at the end of the process, the `Consumer` lambda could close the connection or otherwise handle any tasks that should be done at the end of the process.

4.4.2. Asynchronous & multi-threaded: `create`

`create` is a more advanced form of programmatic creation of a `Flux` which is suitable for multiple emissions per round, even from multiple threads.

It exposes a `FluxSink`, with its `next`, `error`, and `complete` methods. Contrary to `generate`, it doesn't have a state-based variant. On the other hand, it can trigger multi-threaded events in the callback.

`create` can be very useful to bridge an existing API with the reactive world - such as an asynchronous API based on listeners.

`create` **doesn't parallelize your code nor does it make it asynchronous**, even though it *can* be used with asynchronous APIs. If you block within the `create` lambda, you expose yourself to deadlocks and similar side effects. Even with the use of `subscribeOn`, there's the caveat that a long-blocking `create` lambda (such as an infinite loop calling `sink.next(t)`) can lock the pipeline: the requests would never be performed due to the loop starving the same thread they are supposed to run from. Use the `subscribeOn(Scheduler, false)` variant: `requestOnSeparateThread = false` will use the `Scheduler` thread for the `create` and still let data flow by performing `request` in the original thread.

Imagine that you use a listener-based API. It processes data by chunks and has two events: (1) a chunk of data is ready and (2) the processing is complete (terminal event), as represented in the `MyEventListener` interface:

```
interface MyEventListener<T> {
    void onDataChunk(List<T> chunk);
    void processComplete();
}
```

You can use `create` to bridge this into a `Flux<T>`:

```
Flux<String> bridge = Flux.create(sink -> {
    myEventProcessor.register( ④
      new MyEventListener<String>() { ①

        public void onDataChunk(List<String> chunk) {
          for(String s : chunk) {
            sink.next(s); ②
          }
        }

        public void processComplete() {
            sink.complete(); ③
        }
    });
});
```

① Bridge to the `MyEventListener` API

② Each element in a chunk becomes an element in the `Flux`.

③ The `processComplete` event is translated to `onComplete`.

④ All of this is done asynchronously whenever the `myEventProcessor` executes.

Additionally, since `create` can bridge asynchronous APIs and manages backpressure, you can refine how to behave backpressure-wise, by indicating an `OverflowStrategy`:

- `IGNORE` to Completely ignore downstream backpressure requests. This may yield `IllegalStateException` when queues get full downstream.
- `ERROR` to signal an `IllegalStateException` when the downstream can't keep up.
- `DROP` to drop the incoming signal if the downstream is not ready to receive it.
- `LATEST` to let downstream only get the latest signals from upstream.
- `BUFFER` (the default) to buffer all signals if the downstream can't keep up. (this does unbounded buffering and may lead to `OutOfMemoryError`).

`Mono` also has a `create` generator. The `MonoSink` of Mono's create doesn't allow several emissions. It will drop all signals after the first one.

4.4.3. Asynchronous but single-threaded: `push`

`push` is a middle ground between `generate` and `create` which is suitable for processing events from a single producer. It is similar to `create` in the sense that it can also be asynchronous and can manage backpressure using any of the overflow strategies supported by `create`. However, **only one producing thread** may invoke `next`, `complete` or `error` at a time.

```
Flux<String> bridge = Flux.push(sink -> {
    myEventProcessor.register(
      new SingleThreadEventListener<String>() { ①

        public void onDataChunk(List<String> chunk) {
          for(String s : chunk) {
            sink.next(s); ②
          }
        }

        public void processComplete() {
            sink.complete(); ③
        }

        public void processError(Throwable e) {
            sink.error(e); ④
        }
    });
});
```

① Bridge to the `SingleThreadEventListener` API.

② Events are pushed to the sink using `next` from a single listener thread.

③ `complete` event generated from the same listener thread.

④ `error` event also generated from the same listener thread.

An hybrid push/pull model

Most Reactor operators, like `create`, follow an hybrid **push/pull** model. What we mean by that is that despite most of the processing being asynchronous (suggesting a *push* approach), there is a small *pull* component to it: the request.

The consumer *pulls* data from the source in the sense that it won't emit anything until first requested. The source *pushes* data to the consumer whenever it becomes available, but within the bounds of its requested amount.

Note that `push()` and `create()` both allow to set up an `onRequest` consumer in order to manage the request amount and to ensure that data is pushed through the sink only when there is pending request.

```
Flux<String> bridge = Flux.create(sink -> {
    myMessageProcessor.register(
      new MyMessageListener<String>() {

        public void onMessage(List<String> messages) {
          for(String s : messages) {
            sink.next(s); ③
          }
        }
    });
    sink.onRequest(n -> {
        List<String> messages = myMessageProcessor.getHistory(n); ①
        for(String s : messages) {
           sink.next(s); ②
        }
    });
});
```

① Poll for messages when requests are made.

② If messages are available immediately, push them to the sink.

③ The remaining messages that arrive asynchronously later are also delivered.

Cleaning up after `push()` or `create()`

Two callbacks, `onDispose` and `onCancel`, perform any cleanup on cancellation or termination. `onDispose` can be used to perform cleanup when the `Flux` completes, errors out, or is cancelled. `onCancel` can be used to perform any action specific to cancellation prior to cleanup with `onDispose`.

```
Flux<String> bridge = Flux.create(sink -> {
    sink.onRequest(n -> channel.poll(n))
        .onCancel(() -> channel.cancel()) ①
        .onDispose(() -> channel.close())  ②
    });
```

① `onCancel` is invoked first, for cancel signal only.

② `onDispose` is invoked for complete, error, or cancel signals.

4.4.4. Handle

The `handle` method is a bit different: it is an instance method, meaning that it is chained on an existing source (as are the common operators). It is present in both `Mono` and `Flux`.

It is close to `generate`, in the sense that it uses a `SynchronousSink` and only allows one-by-one emissions. However, `handle` can be used to generate an arbitrary value out of each source element, possibly skipping some elements. In this way, it can serve as a combination of `map` and `filter`. The signature of handle is as follows:

```
Flux<R> handle(BiConsumer<T, SynchronousSink<R>>);
```

Let's consider an example. The reactive streams specification disallows `null` values in a sequence. What if you want to perform a `map` but you want to use a preexisting method as the map function, and that method sometimes returns null?

For instance, the following method can be applied safely to a source of integers:

```
public String alphabet(int letterNumber) {
    if (letterNumber < 1 || letterNumber > 26) {
        return null;
    }
    int letterIndexAscii = 'A' + letterNumber - 1;
    return "" + (char) letterIndexAscii;
}
```

We can then use `handle` to remove any nulls:

Using `handle` for a "map and eliminate nulls" scenario

```
Flux<String> alphabet = Flux.just(-1, 30, 13, 9, 20)
    .handle((i, sink) -> {
        String letter = alphabet(i); ①
        if (letter != null) ②
            sink.next(letter); ③
    });

alphabet.subscribe(System.out::println);
```

① Map to letters.

② If the "map function" returns null....

③ Filter it out by not calling `sink.next`.

Which will print out:

```
M
I
T
```

4.5. Threading and Schedulers

Reactor, like RxJava, can be considered **concurrency agnostic**. That is, it does not enforce a concurrency model. Rather it leaves you, the developer, in command. However, that does not prevent the library from helping you with concurrency.

Obtaining a `Flux` or a `Mono` doesn't necessarily mean it will run in a dedicated `Thread`. Instead, most operators continue working in the Thread on which the previous operator executed. Unless specified, the topmost operator (the source) itself runs on the Thread in which the subscribe() call was made.

```
public static void main(String[] args) {
  final Mono<String> mono = Mono.just("hello "); ①

  new Thread(() -> mono
      .map(msg -> msg + "thread ")
      .subscribe(v -> ②
          System.out.println(v + Thread.currentThread().getName()) ③
      )
  ).join();

}
```

① The Mono<String> is assembled in thread main…

② …but it is subscribed to in thread Thread-0.

③ As a consequence, both the map and onNext callback actually run in Thread-0

The code above produces the following output:

```
hello thread Thread-0
```

In Reactor, the execution model and where the execution happens is determined by the Scheduler that is used. A Scheduler has scheduling responsibilities similar to an ExecutorService, but having a dedicated abstraction allows to do more, notably acting as a clock and enabling a wider range of implementations (virtual time for tests, trampolining or immediate scheduling, etc…).

The Schedulers class has static methods that give access to the following execution contexts:

- No execution context (Schedulers.immediate()): at processing time, the submitted Runnable will be directly executed, effectively running them on the current Thread (can be seen as a "null object" or no-op Scheduler).
- A single, reusable thread (Schedulers.single()). Note that this method reuses the same thread for all callers, until the Scheduler is disposed. If you want a per-call dedicated thread, use Schedulers.newSingle() for each call.
- An elastic thread pool (Schedulers.elastic()). It creates new worker pools as needed, and reuse idle ones. Worker pools that stay idle for too long (default is 60s) are disposed. This is a good choice for I/O blocking work for instance. Schedulers.elastic() is a handy way to give a blocking process its own thread, so that it does not tie up other resources. See How Do I Wrap a Synchronous, Blocking Call?.
- a fixed pool of workers that is tuned for parallel work (Schedulers.parallel()). It creates as many workers as you have CPU cores.

Additionally, you can create a Scheduler out of any pre-existing ExecutorService by using

`Schedulers.fromExecutorService(ExecutorService)`. (You can also create one from an `Executor`, although doing so is discouraged.)

You can also create new instances of the various scheduler types by using the `newXXX` methods. For example, `Schedulers.newElastic(yourScheduleName)` creates a new elastic scheduler named `yourScheduleName`.

> ⚠ While `elastic` is made to help with legacy blocking code if it cannot be avoided, `single` and `parallel` are not. As a consequence, the use of Reactor blocking APIs (`block()`, `blockFirst()`, `blockLast()`, as well as iterating over `toIterable()` or `toStream()`) inside the default single and parallel Schedulers will result in an `IllegalStateException` being thrown.
>
> Custom `Schedulers` can also be marked as "non blocking only" by creating instances of `Thread` that implement the `NonBlocking` marker interface.

Some operators use a specific Scheduler from `Schedulers` by default (and usually give you the option of providing a different one). For instance, calling the factory method `Flux.interval(Duration.ofMillis(300))` produces a `Flux<Long>` that ticks every 300ms. By default, this is enabled by `Schedulers.parallel()`. The following line changes the Scheduler to a new instance similar to `Schedulers.single()`:

```
Flux.interval(Duration.ofMillis(300), Schedulers.newSingle("test"))
```

Reactor offers two means of switching the execution context (or `Scheduler`) in a reactive chain: `publishOn` and `subscribeOn`. Both take a `Scheduler` and let you switch the execution context to that scheduler. But the placement of `publishOn` in the chain matters, while the placement of `subscribeOn` does not. To understand that difference, you first have to remember that nothing happens until you subscribe().

In Reactor, when you chain operators, you can wrap as many `Flux` and `Mono` implementations inside one another as you need. Once you subscribe, a chain of `Subscriber` objects is created, backward (up the chain) to the first publisher. This is effectively hidden from you. All you can see is the outer layer of `Flux` (or `Mono`) and `Subscription`, but these intermediate operator-specific subscribers are where the real work happens.

With that knowledge, we can have a closer look at the `publishOn` and `subscribeOn` operators:

4.5.1. `publishOn`

`publishOn` applies in the same way as any other operator, in the middle of the subscriber chain. It takes signals from upstream and replays them downstream while executing the callback on a worker from the associated `Scheduler`. Consequently, it **affects where the subsequent operators will execute** (until another `publishOn` is chained in):

- changes the execution context to one `Thread` picked by the `Scheduler`
- as per the specification, `onNext` happen in sequence, so this uses up a single thread

- unless they work on a specific `Scheduler`, operators after `publishOn` continue execution on that same thread

```
Scheduler s = Schedulers.newParallel("parallel-scheduler", 4); ①

final Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i)  ②
    .publishOn(s)  ③
    .map(i -> "value " + i);  ④

new Thread(() -> flux.subscribe(System.out::println));  ⑤
```

① Creates a new `Scheduler` backed by 4 `Thread`

② The first `map` runs on the anonymous thread in <5>

③ The `publishOn` switches the whole sequence on a `Thread` picked from <1>

④ The second `map` runs on said `Thread` from <1>

⑤ This anonymous `Thread` is the one where the *subscription* happens. The print happens on the latest execution context which is the one from `publishOn`

4.5.2. `subscribeOn`

`subscribeOn` applies to the subscription process, when that backward chain is constructed. As a consequence, no matter where you place the `subscribeOn` in the chain, **it always affects the context of the source emission**. However, this does not affect the behavior of subsequent calls to `publishOn`. They still switch the execution context for the part of the chain after them.

- changes the `Thread` from which the **whole chain** of operators above subscribes
- picks one thread from the `Scheduler`

Only the earliest `subscribeOn` call in the chain is actually taken into account.

```
Scheduler s = Schedulers.newParallel("parallel-scheduler", 4); ①

final Flux<String> flux = Flux
    .range(1, 2)
    .map(i -> 10 + i)  ②
    .subscribeOn(s)  ③
    .map(i -> "value " + i);  ④

new Thread(() -> flux.subscribe(System.out::println));  ⑤
```

① Creates a new `Scheduler` backed by 4 `Thread`

② The first `map` runs on one of these 4 threads…

③ …because `subscribeOn` switches the whole sequence right from subscription time (<5>)

④ The second `map` also runs on same thread

⑤ This anonymous `Thread` is the one where the *subscription* initially happens, but `subscribeOn` immediately shifts it to one of the 4 scheduler threads…

4.6. Handling Errors

For a quick look at the available operators for error handling, see the relevant operator decision tree.

In Reactive Streams, errors are terminal events. As soon as an error occurs, it stops the sequence and gets propagated down the chain of operators to the last step, the `Subscriber` you defined and its `onError` method.

Such errors should still be dealt with at the application level. For instance, you might display an error notification in a UI or send a meaningful error payload in a REST endpoint. For this reason, the subscriber's `onError` method should always be defined.

If not defined, `onError` throws an `UnsupportedOperationException`. You can further detect and triage it with the `Exceptions.isErrorCallbackNotImplemented` method.

Reactor also offers alternative means of dealing with errors in the middle of the chain, as error-handling operators. Here is an example:

```
Flux.just(1, 2, 0)
    .map(i -> "100 / " + i + " = " + (100 / i)) //this triggers an error with 0
    .onErrorReturn("Divided by zero :("); // error handling example
```

Before you learn about error-handling operators, you must keep in mind that **any error in a reactive sequence is a terminal event**. Even if an error-handling operator is used, it does not allow the **original** sequence to continue. Rather, it converts the `onError` signal into the start of a **new** sequence (the fallback one). In other words, it replaces the terminated sequence *upstream* of it.

Now we can consider each means of error handling one-by-one. When relevant, we make a parallel with imperative programming's `try` patterns.

4.6.1. Error Handling Operators

You may be familiar with several ways of dealing with exceptions in a try-catch block. Most notably, these include the following:

1. Catch and return a static default value.
2. Catch and execute an alternative path with a fallback method.
3. Catch and dynamically compute a fallback value.
4. Catch, wrap to a `BusinessException`, and re-throw.

5. Catch, log an error-specific message, and re-throw.
6. Use the `finally` block to clean up resources or a Java 7 "try-with-resource" construct.

All of these have equivalents in Reactor, in the form of error-handling operators. Before looking into these operators, let's first establish a parallel between a reactive chain and a try-catch block.

When subscribing, the `onError` callback at the end of the chain is akin to a `catch` block. There, execution skips to the catch in case an `Exception` is thrown:

```
Flux<String> s = Flux.range(1, 10)
    .map(v -> doSomethingDangerous(v)) ①
    .map(v -> doSecondTransform(v)); ②
s.subscribe(value -> System.out.println("RECEIVED " + value), ③
            error -> System.err.println("CAUGHT " + error) ④
);
```

① A transformation is performed that can throw an exception.

② If everything went well, a second transformation is performed.

③ Each successfully transformed value is printed out.

④ In case of an error, the sequence terminates and an error message is displayed.

This is conceptually similar to the following try/catch block:

```
try {
    for (int i = 1; i < 11; i++) {
        String v1 = doSomethingDangerous(i); ①
        String v2 = doSecondTransform(v1); ②
        System.out.println("RECEIVED " + v2);
    }
} catch (Throwable t) {
    System.err.println("CAUGHT " + t); ③
}
```

① If an exception is thrown here...

② ...the rest of the loop is skipped...

③ ...the execution goes straight to here.

Now that we have established a parallel, we'll look at the different error handling cases and their equivalent operators.

Static Fallback Value

The equivalent of "***Catch and return a static default value***" is `onErrorReturn`:

```
try {
  return doSomethingDangerous(10);
}
catch (Throwable error) {
  return "RECOVERED";
}
```

becomes:

```
Flux.just(10)
    .map(this::doSomethingDangerous)
    .onErrorReturn("RECOVERED");
```

You also have the option of applying a `Predicate` on the exception to decided whether or not to recover:

```
Flux.just(10)
    .map(this::doSomethingDangerous)
    .onErrorReturn(e -> e.getMessage().equals("boom10"), "recovered10"); ①
```

① only recover if the message of the exception is `"boom10"`

Fallback Method

If you want more than a single default value and you have an alternative safer way of processing your data, you can use `onErrorResume`. This would be the equivalent of "***Catch and execute an alternative path with a fallback method***".

For example, if your nominal process is fetching data from an external and unreliable service, but you also keep a local cache of the same data that *can* be a bit more out of date but is more reliable, you could do the following:

```
String v1;
try {
  v1 = callExternalService("key1");
}
catch (Throwable error) {
  v1 = getFromCache("key1");
}

String v2;
try {
  v2 = callExternalService("key2");
}
catch (Throwable error) {
  v2 = getFromCache("key2");
}
```

becomes:

```
Flux.just("key1", "key2")
    .flatMap(k -> callExternalService(k) ①
        .onErrorResume(e -> getFromCache(k)) ②
    );
```

① For each key, we asynchronously call the external service.

② If the external service call fails, we fallback to the cache for that key. Note that we always apply the same fallback, whatever the source error, `e`, is.

Like `onErrorReturn`, `onErrorResume` has variants that let you filter which exceptions to fallback on, based either on the exception's class or a `Predicate`. The fact that it takes a `Function` also allows you to choose a different fallback sequence to switch to, depending on the error encountered:

```
Flux.just("timeout1", "unknown", "key2")
    .flatMap(k -> callExternalService(k)
        .onErrorResume(error -> { ①
            if (error instanceof TimeoutException) ②
                return getFromCache(k);
            else if (error instanceof UnknownKeyException)  ③
                return registerNewEntry(k, "DEFAULT");
            else
                return Flux.error(error); ④
        })
    );
```

① The function allows dynamically choosing how to continue.

② If the source times out, hit the local cache.

③ If the source says the key is unknown, create a new entry.

④ In all other cases, "re-throw".

Dynamic Fallback Value

Even if you do not have an alternative safer way of processing your data, you might want to compute a fallback value out of the exception you received. This would be the equivalent of "***Catch and dynamically compute a fallback value***".

For instance, if your return type `MyWrapper` has a variant dedicated to holding an exception (think `Future.complete(T success)` vs `Future.completeExceptionally(Throwable error)`), you could instantiate the error-holding variant and pass the exception.

An imperative example would look like the following:

```
try {
  Value v = erroringMethod();
  return MyWrapper.fromValue(v);
}
catch (Throwable error) {
  return MyWrapper.fromError(error);
}
```

This can be done reactively in the same way as the fallback method solution, using `onErrorResume`, with a tiny bit of boilerplate:

```
erroringFlux.onErrorResume(error -> Mono.just( ①
        MyWrapper.fromError(error) ②
));
```

① Since you expect a `MyWrapper` representation of the error, you'll need to get a `Mono<MyWrapper>` for `onErrorResume`. We use `Mono.just()` for that.

② We need to compute the value out of the exception. Here we achieved That by wrapping the exception using a relevant `MyWrapper` factory method.

Catch and Rethrow

"***Catch, wrap to a*** **`BusinessException`*****, and re-throw***" in the imperative world looks like the following:

```
try {
  return callExternalService(k);
}
catch (Throwable error) {
  throw new BusinessException("oops, SLA exceeded", error);
}
```

In the "fallback method" example, the last line inside the `flatMap` gives us a hint at achieving the same reactively:

```
Flux.just("timeout1")
    .flatMap(k -> callExternalService(k))
    .onErrorResume(original -> Flux.error(
            new BusinessException("oops, SLA exceeded", original))
    );
```

However, there is a more straightforward way of achieving the same with `onErrorMap`:

```
Flux.just("timeout1")
    .flatMap(k -> callExternalService(k))
    .onErrorMap(original -> new BusinessException("oops, SLA exceeded", original));
```

Log or React on the Side

For cases where you want the error to continue propagating but you still want to react to it without modifying the sequence (logging it, for instance) there is the `doOnError` operator. This is the equivalent of "***Catch, log an error-specific message, and re-throw***" pattern, shown below:

```
try {
  return callExternalService(k);
}
catch (RuntimeException error) {
  //make a record of the error
  log("uh oh, falling back, service failed for key " + k);
  throw error;
}
```

The `doOnError` operator, as well as all operators prefixed with `doOn` , are sometimes referred to as a "side-effect". They let you peek inside the sequence's events without modifying them.

Like the imperative example above, the following example still propagates the error yet ensures that we at least log that the external service had a failure. We can also imagine we have statistic counters to increment as a second error side-effect.

```
LongAdder failureStat = new LongAdder();
Flux<String> flux =
Flux.just("unknown")
    .flatMap(k -> callExternalService(k) ①
        .doOnError(e -> {
            failureStat.increment();
            log("uh oh, falling back, service failed for key " + k); ②
        })
        ③
    );
```

① The external service call that can fail…

② …is decorated with a logging and stats side-effect…

③ …after which point it will still terminate with an error, unless we use an error-recovery operator here

Using Resources and the Finally Block

The last parallel to draw with imperative programming is the cleaning up that can be done either via a "***Use of the `finally` block to clean up resources***" or a "***Java 7 try-with-resource construct***", both shown below:

Imperative use of finally

```
Stats stats = new Stats();
stats.startTimer();
try {
  doSomethingDangerous();
}
finally {
  stats.stopTimerAndRecordTiming();
}
```

Imperative use of try-with-resource

```
try (SomeAutoCloseable disposableInstance = new SomeAutoCloseable()) {
  return disposableInstance.toString();
}
```

Both have their Reactor equivalents, `doFinally` and `using`.

`doFinally` is about side-effects that you want to be executed whenever the sequence terminates (with `onComplete` or `onError`) or is cancelled. It gives you a hint as to what kind of termination triggered the side-effect:

Reactive finally: `doFinally()`

```
Stats stats = new Stats();
LongAdder statsCancel = new LongAdder();

Flux<String> flux =
Flux.just("foo", "bar")
    .doOnSubscribe(s -> stats.startTimer())
    .doFinally(type -> { ①
        stats.stopTimerAndRecordTiming();②
        if (type == SignalType.CANCEL) ③
          statsCancel.increment();
    })
    .take(1); ④
```

① `doFinally` consumes a `SignalType` for the type of termination.

② Similarly to `finally` blocks, we always record the timing.

③ Here we also increment statistics in case of cancellation only.

④ `take(1)` will cancel after 1 item is emitted.

On the other hand, `using` handles the case where a `Flux` is derived from a resource, and that resource must be acted upon whenever processing is done. Let’s replace the `AutoCloseable` interface of "try-with-resource" with a `Disposable`:

The Disposable resource

```
AtomicBoolean isDisposed = new AtomicBoolean();
Disposable disposableInstance = new Disposable() {
    @Override
    public void dispose() {
        isDisposed.set(true); ④
    }

    @Override
    public String toString() {
        return "DISPOSABLE";
    }
};
```

Now we can do the reactive equivalent of "try-with-resource" on it, which looks like the following:

Reactive try-with-resource: `using()`

```
Flux<String> flux =
Flux.using(
        () -> disposableInstance, ①
        disposable -> Flux.just(disposable.toString()), ②
        Disposable::dispose ③
);
```

① The first lambda generates the resource. Here we return our mock `Disposable`.

② The second lambda processes the resource, returning a `Flux<T>`.

③ The third lambda is called when the `Flux` from 2) terminates or is cancelled, to clean up resources.

④ After subscription and execution of the sequence, the `isDisposed` atomic boolean would become `true`.

Demonstrating the Terminal Aspect of `onError`

In order to demonstrate that all these operators cause the upstream original sequence to terminate when the error happens, we can use a more visual example with a `Flux.interval`. The interval operator ticks every x units of time with an increasing `Long` value:

```
Flux<String> flux =
Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .onErrorReturn("Uh oh");

flux.subscribe(System.out::println);
Thread.sleep(2100); ①
```

① Note that `interval` executes on a **timer** `Scheduler` by default. Assuming we want to run that example in a main class, we add a `sleep` call here so that the application does not exit immediately without any value being produced.

This prints out one line every 250ms, as follows:

```
tick 0
tick 1
tick 2
Uh oh
```

Even with one extra second of runtime, no more tick comes in from the `interval`. The sequence was indeed terminated by the error.

Retrying

There is another operator of interest with regards to error handling, and you might be tempted to use it in the case above. `retry`, as its name indicates, lets you retry an error-producing sequence.

The thing to keep in mind is that it works by **re-subscribing** to the upstream `Flux`. This is really a different sequence, and the original one is still terminated. To verify that, we can re-use the previous example and append a `retry(1)` to retry once instead of using `onErrorReturn`:

```
Flux.interval(Duration.ofMillis(250))
    .map(input -> {
        if (input < 3) return "tick " + input;
        throw new RuntimeException("boom");
    })
    .retry(1)
    .elapsed() ①
    .subscribe(System.out::println, System.err::println); ②

Thread.sleep(2100); ③
```

① `elapsed` associates each value with the duration since previous value was emitted.

② We also want to see when there is an `onError`.

③ Ensure we have enough time for our 4x2 ticks.

This produces the following output:

```
259,tick 0
249,tick 1
251,tick 2
506,tick 0 ①
248,tick 1
253,tick 2
java.lang.RuntimeException: boom
```

① A new `interval` started, from tick 0. The additional 250ms duration is coming from the 4th tick, the one that causes the exception and subsequent retry.

As you can see above, `retry(1)` merely re-subscribed to the original `interval` once, restarting the tick from 0. The second time around, since the exception still occurs, it gives up and propagates the error downstream.

There is a more advanced version of `retry` (called `retryWhen`) that uses a "companion" `Flux` to tell whether or not a particular failure should retry. This companion `Flux` is created by the operator but decorated by the user, in order to customize the retry condition.

The companion `Flux` is a `Flux<Throwable>` that gets passed to a `Function`, the sole parameter of `retryWhen`. As the user, you define that function and make it return a new `Publisher<?>`. Retry cycles will go like this:

1. Each time an error happens (potential for a retry), the error is emitted into the companion `Flux`, which has been decorated by your function. Having a `Flux` here gives a bird eye's view of all the attempts so far.
2. If the companion `Flux` emits a value, a retry happens.
3. If the companion `Flux` completes, the error is swallowed, the retry cycle stops and the resulting sequence **completes** too.
4. If the companion `Flux` produces an error `e`, the retry cycle stops and the resulting sequence **errors** with `e`.

The distinction between the previous two cases is important. Simply completing the companion would effectively swallow an error. Consider the following way of emulating `retry(3)` using `retryWhen`:

```
Flux<String> flux = Flux
    .<String>error(new IllegalArgumentException()) ①
    .doOnError(System.out::println) ②
    .retryWhen(companion -> companion.take(3)); ③
```

① This continuously produces errors, calling for retry attempts.

② `doOnError` **before** the retry will let us log and see all failures.

③ Here, we consider the first 3 errors as retry-able (`take(3)`) and then give up.

In effect, this results in an **empty** `Flux`, but it completes **successfully**. Since `retry(3)` on the same `Flux` would have terminated with the latest error, this `retryWhen` example is not exactly the same as a `retry(3)`.

Getting to the same behavior involves a few additional tricks:

```
Flux<String> flux =
Flux.<String>error(new IllegalArgumentException())
    .retryWhen(companion -> companion
    .zipWith(Flux.range(1, 4), ①
          (error, index) -> { ②
            if (index < 4) return index; ③
            else throw Exceptions.propagate(error); ④
          })
    );
```

① Trick one: use `zip` and a `range` of "number of acceptable retries + 1".

② The `zip` function lets you count the retries while keeping track of the original error.

③ To allow for three retries, indexes before 4 return a value to emit.

④ In order to terminate the sequence in error, we throw the original exception after these three retries.

Similar code can be used to implement an *exponential backoff and retry* pattern, as shown in the FAQ.

4.6.2. Handling Exceptions in Operators or Functions

In general, all operators can themselves contain code that potentially trigger an exception or calls to a user-defined callback that can similarly fail, so they all contain some form of error handling.

As a rule of thumb, an **Unchecked Exception** is always propagated through `onError`. For instance, throwing a `RuntimeException` inside a `map` function translates to an `onError` event, as shown in the following code:

```
Flux.just("foo")
    .map(s -> { throw new IllegalArgumentException(s); })
    .subscribe(v -> System.out.println("GOT VALUE"),
               e -> System.out.println("ERROR: " + e));
```

The preceding code prints out:

```
ERROR: java.lang.IllegalArgumentException: foo
```

The `Exception` can be tuned before it is passed to `onError`, through the use of a hook.

Reactor, however, defines a set of exceptions (such as `OutOfMemoryError`) that are always deemed **fatal**. See the `Exceptions.throwIfFatal` method. These errors mean that Reactor cannot keep operating and are thrown rather than propagated.

Internally, there are also cases where an unchecked exception still cannot be propagated (most notably during the subscribe and request phases), due to concurrency races that could lead to double `onError` or `onComplete` conditions. When these races happen, the error that cannot be propagated is "dropped". These cases can still be managed to some extent via customizable hooks, see Dropping Hooks.

You may ask: "What about **Checked Exceptions**?"

If, for example, you need to call some method that declares it `throws` exceptions, you still have to deal with those exceptions in a `try-catch` block. You have several options, though:

1. Catch the exception and recover from it. The sequence continues normally.
2. Catch the exception and wrap it into an *unchecked* exception, then throw it (interrupting the sequence). The `Exceptions` utility class can help you with that (we get to that next).
3. If you are expected to return a `Flux` (for example, you are in a `flatMap`), wrap the exception in an error-producing `Flux`: `return Flux.error(checkedException)`. (The sequence also terminates.)

Reactor has an `Exceptions` utility class that you can use to ensure that exceptions are wrapped only if they are checked exceptions:

- Use the `Exceptions.propagate` method to wrap exceptions, if necessary. It also calls `throwIfFatal` first and does not wrap `RuntimeException`.
- Use the `Exceptions.unwrap` method to get the original unwrapped exception (going back to the root cause of a hierarchy of reactor-specific exceptions).

Consider an example of a `map` that uses a conversion method that can throw an `IOException`:

```
public String convert(int i) throws IOException {
    if (i > 3) {
        throw new IOException("boom " + i);
    }
    return "OK " + i;
}
```

Now imagine that you want to use that method in a `map`. You must now explicitly catch the exception, and your map function cannot re-throw it. So you can propagate it to the map's `onError` method as a `RuntimeException`:

```
Flux<String> converted = Flux
    .range(1, 10)
    .map(i -> {
        try { return convert(i); }
        catch (IOException e) { throw Exceptions.propagate(e); }
    });
```

Later on, when subscribing to the above `Flux` and reacting to errors (such as in the UI), you could revert back to the original exception in case you want to do something special for IOExceptions, as shown in the following example:

```
converted.subscribe(
    v -> System.out.println("RECEIVED: " + v),
    e -> {
        if (Exceptions.unwrap(e) instanceof IOException) {
            System.out.println("Something bad happened with I/O");
        } else {
            System.out.println("Something bad happened");
        }
    }
);
```

4.7. Processors

Processors are a special kind of `Publisher` that are also a `Subscriber`. That means that you can `subscribe` to a `Processor` (generally, they implement `Flux`), but you can also call methods to manually inject data into the sequence or terminate it.

There are several kinds of Processors, each with a few particular semantics, but before you start looking into these, you need to ask yourself the following question:

4.7.1. Do I Need a Processor?

Most of the time, you should try to avoid using a `Processor`. They are harder to use correctly and prone to some corner cases.

If you think a `Processor` could be a good match for your use case, ask yourself if you have tried these two alternatives:

1. Could an operator or combination of operators fit the bill? (See Which operator do I need?)
2. Could a "generator" operator work instead? (Generally, these operators are made to bridge APIs that are not reactive, providing a "sink" that is similar in concept to a `Processor` in the sense that it lets you manually populate the sequence with data or terminate it).

If, after exploring the above alternatives, you still think you need a `Processor`, read the Overview of Available Processors section below to learn about the different implementations.

4.7.2. Safely Produce from Multiple Threads by Using the `Sink` Facade

Rather than directly using Reactor `Processors`, it is a good practice to obtain a `Sink` for the `Processor` by calling `sink()` **once**.

`FluxProcessor` sinks safely gate multi-threaded producers and can be used by applications that generate data from multiple threads concurrently. For example, a thread-safe serialized sink can be created for `UnicastProcessor`:

```
UnicastProcessor<Integer> processor = UnicastProcessor.create();
FluxSink<Integer> sink = processor.sink(overflowStrategy);
```

Multiple producer threads may concurrently generate data on the following serialized sink:

```
sink.next(n);
```

> ⚠ Despite the `FluxSink` being adapted for multi-threaded **manual** feeding of the `Processor`, it is not possible to mix the subscriber approach with the sink approach: **you have to *either* subscribe your `FluxProcessor` to a source `Publisher` *or* feed it manually though its `FluxSink`**.

Overflow from `next` behaves in two possible ways, depending on the `Processor` and its configuration:

- An unbounded processor handles the overflow itself by dropping or buffering.
- A bounded processor blocks or "spins" on the `IGNORE` strategy or applies the `overflowStrategy` behavior specified for the `sink`.

4.7.3. Overview of Available Processors

Reactor Core comes with several flavors of `Processor`. Not all processors have the same semantics but are roughly split into three categories. The following list briefly describes the three kinds of processors:

- **direct** (`DirectProcessor` and `UnicastProcessor`): These processors can only push data through direct user action (calling their `Sink's methods directly).
- **synchronous** (`EmitterProcessor` and `ReplayProcessor`): These processors can either push data through user interaction or by subscribing to an upstream `Publisher` and synchronously draining it.
- **asynchronous** (`WorkQueueProcessor` and `TopicProcessor`): These processors can either push data obtained from multiple upstream `Publishers` or through user interaction. They are more robust and are backed by a `RingBuffer` data structure in order to deal with their multiple upstreams.

The asynchronous processors are the most complex to instantiate, with a lot of different options. Consequently, they expose a `Builder` interface. The simpler processors have static factory methods instead.

Direct Processor

A **Direct** `Processor` is a processor that can dispatch signals to zero to many `Subscribers`. It is the simplest one to instantiate, with a single `DirectProcessor#create()` static factory method. On the other hand, **it has the limitation of not handling backpressure**. As a consequence, a `DirectProcessor` signals an `IllegalStateException` to its subscribers if you push N elements through it but at least one of its subscribers has requested less than N.

Once the `Processor` has terminated (usually through its sink's `error(Throwable)` or `complete()` methods being called), it lets more subscribers subscribe but replays the termination signal to them immediately.

Unicast Processor

A **Unicast** `Processor` can deal with backpressure using an internal buffer. The trade-off is that it can have **at most one** `Subscriber`.

A `UnicastProcessor` has a few more options, reflected by a few `create` static factory methods. For instance, by default it is *unbounded*: if you push any amount of data through it while its `Subscriber` has not yet requested data, it will buffer all of the data.

This can be changed by providing a custom `Queue` implementation for the internal buffering in the `create` factory method. If that queue is bounded, the processor could reject the push of a value when the buffer is full and not enough requests from downstream have been received.

In that *bounded* case, the processor can also be built with a callback that is invoked on each rejected element, allowing for cleanup of these rejected elements.

Emitter Processor

An **Emitter** `Processor` is capable of emitting to several subscribers, while honoring backpressure for each of its subscribers. It can also subscribe to a `Publisher` and relay its signals synchronously.

Initially, when it has no subscriber, it can still accept a few data pushes up to a configurable `bufferSize`. After that point, if no `Subscriber` has come in and consumed the data, calls to `onNext` block until the processor is drained (which can only happen concurrently by then).

Thus, the first `Subscriber` to subscribe receives up to `bufferSize` elements upon subscribing. However, after that, the processor stops replaying signals to additional subscribers. These subsequent subscribers instead only receive the signals pushed through the processor **after** they have subscribed. The internal buffer is still used for backpressure purposes.

By default, if all of its subscribers are cancelled (which basically means they have all un-subscribed), it will clear its internal buffer and stop accepting new subscribers. This can be tuned by the `autoCancel` parameter in the `create` static factory methods.

Replay Processor

A **Replay** `Processor` caches elements that are either pushed directly through its `sink()` or elements from an upstream `Publisher` and replays them to late subscribers.

It can be created in multiple configurations:

- Caching a single element (`cacheLast`).
- Caching a limited history (`create(int)`), unbounded history (`create()`).
- Caching time-based replay window (`createTimeout(Duration)`).
- Caching combination of history size and time window (`createSizeOrTimeout(int, Duration)`).

Topic Processor

A **Topic** `Processor` is an asynchronous processor capable of relaying elements from multiple upstream `Publishers` when created in the `shared` configuration (see the `share(boolean)` option of the `builder()`).

Note that the share option is mandatory if you intend to concurrently call `TopicProcessor's `onNext`, `onComplete`, or `onError` methods directly or from a concurrent upstream Publisher.

Otherwise, such concurrent calls are illegal, as the processor is then fully compliant with the Reactive Streams specification.

A `TopicProcessor` is capable of fanning out to multiple `Subscribers`. It does so by associating a `Thread` to each `Subscriber`, which will run until an `onError` or `onComplete` signal is pushed through the processor or until the associated `Subscriber` is cancelled. The maximum number of downstream subscribers is driven by the `executor` builder option. Provide a bounded `ExecutorService` to limit it to a specific number.

The processor is backed by a `RingBuffer` data structure that stores pushed signals. Each `Subscriber` thread keeps track of its associated demand and the correct indexes in the `RingBuffer`.

This processor also has an `autoCancel` builder option: If set to `true` (the default), it results in the source `Publisher`(s) being cancelled when all subscribers are cancelled.

WorkQueue Processor

A **WorkQueue** `Processor` is also an asynchronous processor capable of relaying elements from multiple upstream `Publishers` when created in the `shared` configuration (it shares most of its builder options with `TopicProcessor`).

It relaxes its compliance with the Reactive Streams specification, but it acquires the benefit of requiring fewer resources than the `TopicProcessor`. It is still based on a `RingBuffer` but avoids the overhead of creating one consumer `Thread` per `Subscriber`. As a result, it scales better than the `TopicProcessor`.

The trade-off is that its distribution pattern is a little bit different: Requests from each subscriber all add up together, and the processor relays signals to only one `Subscriber` at a time, in a kind of round-robin distribution rather than fan-out pattern.

ℹ A fair round-robin distribution is not guaranteed.

The `WorkQueueProcessor` mostly has the same builder options as the `TopicProcessor`, such as `autoCancel`, `share`, and `waitStrategy`. The maximum number of downstream subscribers is also driven by a configurable `ExecutorService` with the `executor` option.

> ⚠ You should take care not to subscribe too many `Subscribers` to a `WorkQueueProcessor`, as doing so **could lock the processor**. If you need to limit the number of possible subscribers, prefer doing so by using a `ThreadPoolExecutor` or a `ForkJoinPool`. The processor can detect their capacity and throw an exception if you subscribe one too many times.

Chapter 5. Kotlin support

5.1. Introduction

Kotlin is a statically-typed language targeting the JVM (and other platforms) which allows writing concise and elegant code while providing a very good interoperability with existing libraries written in Java.

Reactor 3.1 introduces first-class support for Kotlin which is described in this section.

5.2. Requirements

Reactor supports Kotlin 1.1+ and requires `kotlin-stdlib` (or one of its `kotlin-stdlib-jre7` / `kotlin-stdlib-jre8` variants).

5.3. Extensions

Thanks to its great Java interoperability and to Kotlin extensions, Reactor Kotlin APIs leverage regular Java APIs and are additionally enhanced by a few Kotlin specific APIs available out of the box within Reactor artifacts.

Keep in mind that Kotlin extensions need to be imported to be used. This means for example that the `Throwable.toFlux` Kotlin extension will only be available if `import reactor.core.publisher.toFlux` is imported. That said, similar to static imports, an IDE should automatically suggest the import in most cases.

For example, Kotlin reified type parameters provide a workaround for JVM generics type erasure, and Reactor provides some extensions to take advantage of this feature.

You can see bellow a quick comparison of Reactor with Java versus Reactor with Kotlin + extensions.

Java	**Kotlin with extensions**
`Mono.just("foo")`	`"foo".toMono()`
`Flux.fromIterable(list)`	`list.toFlux()`
`Mono.error(new RuntimeException())`	`RuntimeException().toMono()`
`Flux.error(new RuntimeException())`	`RuntimeException().toFlux()`
`flux.ofType(Foo.class)`	`flux.ofType<Foo>()` or `flux.ofType(Foo::class)`
`StepVerifier.create(flux).verifyComplete()`	`flux.test().verifyComplete()`

Reactor KDoc API lists and documents all the Kotlin extensions available.

5.4. Null-safety

One of Kotlin's key features is null-safety - which cleanly deals with `null` values at compile time rather than bumping into the famous `NullPointerException` at runtime. This makes applications safer through nullability declarations and expressing "value or no value" semantics without paying the cost of wrappers like `Optional`. (Kotlin allows using functional constructs with nullable values; check out this comprehensive guide to Kotlin null-safety.)

Although Java does not allow one to express null-safety in its type-system, Reactor now provides null-safety of the whole Reactor API via tooling-friendly annotations declared in the `reactor.util.annotation` package. By default, types from Java APIs used in Kotlin are recognized as platform types for which null-checks are relaxed. Kotlin support for JSR 305 annotations + Reactor nullability annotations provide null-safety for the whole Reactor API to Kotlin developers, with the advantage of dealing with `null` related issues at compile time.

The JSR 305 checks can be configured by adding the `-Xjsr305` compiler flag with the following options: `-Xjsr305={strict|warn|ignore}`.

For kotlin versions 1.1.50+, the default behavior is the same to `-Xjsr305=warn`. The `strict` value is required to have Reactor API full null-safety taken in account but should be considered experimental since Reactor API nullability declaration could evolve even between minor releases and more checks may be added in the future).

Generic type arguments, varargs and array elements nullability are not supported yet, but should be in an upcoming release, see this dicussion for up-to-date information.

Chapter 6. Testing

Whether you have written a simple chain of Reactor operators or your own operator, automated testing is always a good idea.

Reactor comes with a few elements dedicated to testing, gathered into their own artifact: `reactor-test`. You can find that project on Github, inside of the *reactor-core* repository.

To use it in your tests, add it as a test dependency:

reactor-test in Maven, in `<dependencies>`

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
    ①
</dependency>
```

① If you use the BOM, you do not need to specify a `<version>`.

reactor-test in Gradle, amend the `dependencies` block

```
dependencies {
   testCompile 'io.projectreactor:reactor-test'
}
```

The three main uses of `reactor-test` are:

- Testing that a sequence follows a given scenario, step-by-step, with `StepVerifier`.
- Producing data in order to test the behavior of operators (including you own operators) downstream with `TestPublisher`.
- In sequences that can go through several alternative `Publisher` (eg. a chain that uses `switchIfEmpty`, probing such a `Publisher` to ensure it was used (as in, subscribed to).

6.1. Testing a Scenario with `StepVerifier`

The most common case for testing a Reactor sequence is to have a `Flux` or `Mono` defined in your code (for example, it might be returned by a method) and wanting to test how it behaves when subscribed to.

This situation translates well to defining a "test scenario", where you define your expectations in terms of events, step-by-step: what is the next expected event? Do you expect the `Flux` to emit a particular value? Or maybe to do nothing for the next 300ms? All of that can be expressed through the `StepVerifier` API.

For instance, you could have the following utility method in your codebase that decorates a `Flux`:

```
public <T> Flux<T> appendBoomError(Flux<T> source) {
  return source.concatWith(Mono.error(new IllegalArgumentException("boom")));
}
```

In order to test it, you want to verify the following scenario:

> I expect this `Flux` to first emit `foo`, then emit `bar`, and then **produce an error** with the message, `boom`. Subscribe and **verify** these expectations.

In the `StepVerifier` API, this translates to the following test:

```
@Test
public void testAppendBoomError() {
  Flux<String> source = Flux.just("foo", "bar"); ①

  StepVerifier.create( ②
    appendBoomError(source)) ③
    .expectNext("foo") ④
    .expectNext("bar")
    .expectErrorMessage("boom") ⑤
    .verify(); ⑥
}
```

① Since our method needs a source `Flux`, define a simple one for testing purposes.

② Create a `StepVerifier` builder that wraps and verifies a `Flux`.

③ Pass the `Flux` to be tested (the result of calling our utility method).

④ The first signal we expect to happen upon subscription is an `onNext`, with a value of `foo`.

⑤ The last signal we expect to happen is a termination of the sequence with an `onError`. The exception should have `boom` as a message.

⑥ It is important to trigger the test by calling `verify()`.

The API is a builder. You start by creating a `StepVerifier` and passing the sequence to be tested. This offers a choice of methods that allow you to:

- Express *expectations* about the next signals to occur. If any other signal is received (or the content of the signal does not match the expectation), the whole test fails with a meaningful `AssertionError`. For example, you might use `expectNext(T…)` and `expectNextCount(long)`.
- *Consume* the next signal. This is used when you want to skip part of the sequence or when you want to apply a custom `assertion` on the content of the signal (for example, to check that there is an `onNext` event and assert that the emitted item is a list of size 5). For example, you might use `consumeNextWith(Consumer<T>)`.
- Take *miscellaneous actions* such as pausing or running arbitrary code. For example, if you want to manipulate a test-specific state or context. To that effect, you might use `thenAwait(Duration)` and `then(Runnable)`.

For terminal events, the corresponding expectation methods (`expectComplete()` and `expectError()` and all their variants) switch to an API where you cannot express expectations anymore. In that last step, all you can do is perform some additional configuration on the `StepVerifier` and then **trigger the verification**, often with `verify()` or one of its variants.

What happens at this point is that the StepVerifier subscribes to the tested `Flux` or `Mono` and plays the sequence, comparing each new signal with the next step in the scenario. As long as these match, the test is considered a success. As soon as there is a discrepancy, an `AssertionError` is thrown.

> Remember the `verify()` step, which triggers the verification. In order to help, the API includes a few shortcut methods that combine the terminal expectations with a call to `verify()`: `verifyComplete()`, `verifyError()`, `verifyErrorMessage(String)`, and others.

Note that, if one of the lambda-based expectations throws an `AssertionError`, it is reported as is, failing the test. This is useful for custom assertions.

> By default, the `verify()` method and derived shortcut methods (`verifyThenAssertThat`, `verifyComplete()`, etc.) has no timeout. It can block indefinitely. You can use `StepVerifier.setDefaultTimeout(Duration)` to globally set a timeout for these methods, or specify one on a per-call basis with `verify(Duration)`.

6.1.1. Better identifying test failures

`StepVerifier` provides two options to better identify exactly which expectation step caused a test to fail:

- `as(String)`: this method can be used **after** most `expect*` methods to give a description to the preceding expectation. If the expectation fails, its error message will contain the description. Terminal expectations and `verify` cannot be described that way.
- `StepVerifierOptions.create().scenarioName(String)`: Using `StepVerifierOptions` to create you `StepVerifier`, you can use the `scenarioName` method to give the whole scenario a name, which will also be used in assertion error messages.

Note that in both cases, the use of the description/name in messages is only guaranteed for `StepVerifier` methods that produce their own `AssertionError` (eg. throwing an exception manually or through an assertion library in `assertNext` won't add the description/name to said error's message).

6.2. Manipulating Time

`StepVerifier` can be used with time-based operators to avoid long run times for corresponding tests. This is done through the `StepVerifier.withVirtualTime` builder.

It looks like the following example:

```
StepVerifier.withVirtualTime(() -> Mono.delay(Duration.ofDays(1)))
//... continue expectations here
```

This **virtual time** feature plugs in a custom `Scheduler` in Reactor’s `Schedulers` factory. Since these timed operators usually use the default `Schedulers.parallel()` scheduler, replacing it with a `VirtualTimeScheduler` does the trick. However, an important prerequisite is that the operator be instantiated *after* the virtual time scheduler has been activated.

In order to increase the chances this happens correctly, the `StepVerifier` does not take a simple `Flux` as input. `withVirtualTime` takes a `Supplier`, which guides you into lazily creating the instance of the tested flux *after* having done the scheduler set up.

Take extra care to ensure the `Supplier<Publisher<T>>` can be used in a lazy fashion. Otherwise, virtual time is not guaranteed. Especially avoid instantiating the `Flux` earlier in the test code and having the `Supplier` return that variable. Instead, always instantiate the `Flux` inside the lambda.

There are two expectation methods that deal with time, and they are both valid with or without virtual time:

- `thenAwait(Duration)` pauses the evaluation of steps (allowing a few signals to occur or delays to run out).
- `expectNoEvent(Duration)` also lets the sequence play out for a given duration but fails the test if **any** signal occurs during that time.

Both methods pause the thread for the given duration in classic mode and advance the virtual clock instead in virtual mode.

`expectNoEvent` also considers the `subscription` as an event. If you use it as a first step, it usually fails because the subscription signal is detected. Use `expectSubscription().expectNoEvent(duration)` instead.

In order to quickly evaluate the behavior of our `Mono.delay` above, we can finish writing our code like this:

```
StepVerifier.withVirtualTime(() -> Mono.delay(Duration.ofDays(1)))
    .expectSubscription() ①
    .expectNoEvent(Duration.ofDays(1)) ②
    .expectNext(0L) ③
    .verifyComplete(); ④
```

① See the tip above.

② Expect nothing to happen during a full day.

③ Then expect a delay that emits `0`.

④ Then expect completion (and trigger the verification).

We could have used `thenAwait(Duration.ofDays(1))` above, but `expectNoEvent` has the benefit of guaranteeing that nothing happened earlier than it should have.

Note that `verify()` returns a `Duration` value. This is the **real-time** duration of the entire test.

> ⚠ Virtual time is not a silver bullet. Keep in mind that *all* `Schedulers` are replaced with the same `VirtualTimeScheduler`. In some cases, you can lock the verification process because the virtual clock has not moved forward before an expectation is expressed, resulting in the expectation waiting on data that can only be produced by advancing time. In most cases, you need to advance the virtual clock for sequences to emit. Virtual time also gets very limited with infinite sequences, which might hog the thread on which both the sequence and its verification run.

6.3. Performing Post-execution Assertions with `StepVerifier`

After having described the final expectation of your scenario, you can switch to a complementary assertion API instead of triggering `verify()`. To do so, use `verifyThenAssertThat()` instead.

`verifyThenAssertThat()` returns a `StepVerifier.Assertions` object, which you can use to assert a few elements of state once the whole scenario has played out successfully (because it **also calls `verify()`**). Typical (albeit advanced) usage is to capture elements that have been dropped by some operator and assert them (see the section on Hooks).

6.4. Testing the `Context`

For more information about the `Context`, see Adding a Context to a Reactive Sequence.

`StepVerifier` comes with a couple of expectations around the propagation of a `Context`:

- `expectAccessibleContext`: returns a `ContextExpectations` object that you can use to set up expectations on the propagated `Context`. Be sure to call `then()` to return to the set up of sequence expectations.
- `expectNoAccessibleContext`: set up an expectation that NO `Context` can be propagated up the chain of operators under test. This most likely occurs when the `Publisher` under test is not a Reactor one, or doesn't have any operator that can propagate the `Context` (e.g. just a *generator* source).

Additionally, one can associate a test-specific initial `Context` to a `StepVerifier` by using `StepVerifierOptions` to create the verifier.

These features are demonstrated in the following snippet:

```
StepVerifier.create(Mono.just(1).map(i -> i + 10),
                StepVerifierOptions.create().withInitialContext(Context.of("foo",
"bar"))) ①
                    .expectAccessibleContext() ②
                    .contains("foo", "bar") ③
                    .then() ④
                    .expectNext(11)
                    .verifyComplete(); ⑤
```

① Create the `StepVerifier` using `StepVerifierOptions` and pass in an initial `Context`

② Start setting up expectations about `Context` propagation. This alone ensures that a `Context` **was** propagated.

③ An example of a `Context`-specific expectation: it must contain value "bar" for key "foo".

④ We `then()` switch back to setting up normal expectations on the data.

⑤ Let's not forget to `verify()` the whole set of expectations.

6.5. Manually Emitting with `TestPublisher`

For more advanced test cases, it might be useful to have complete mastery over the source of data, in order to trigger finely chosen signals that closely match the particular situation you want to test.

Another situation is when you have implemented your own operator and you want to verify how it behaves with regards to the Reactive Streams specification, especially if its source is not well behaved.

For both cases, `reactor-test` offers the `TestPublisher` class. This is a `Publisher<T>` that lets you programmatically trigger various signals:

- `next(T)` and `next(T, T…)` triggers 1-n `onNext` signals.
- `emit(T…)` does the same and does `complete()`.
- `complete()` terminates with an `onComplete` signal.
- `error(Throwable)` terminates with an `onError` signal.

A well behaved `TestPublisher` can be obtained through the `create` factory method. Also, a misbehaving `TestPublisher` can be created using the `createNonCompliant` factory method. The latter takes a value or multiple values from the `TestPublisher.Violation` enum. The values define which parts of the specification the publisher can overlook. These enum values include:

- `REQUEST_OVERFLOW`: Allows `next` calls to be made despite an insufficient request, without triggering an `IllegalStateException`.
- `ALLOW_NULL`: Allows `next` calls to be made with a `null` value without triggering a `NullPointerException`.
- `CLEANUP_ON_TERMINATE`: Allows termination signals to be sent several times in a row. This includes `complete()`, `error()` and `emit()`.

- `DEFER_CANCELLATION`: Allow the `TestPublisher` to ignore cancellation signals and continue emitting signals as if the cancellation lost race against said signals.

Finally, the `TestPublisher` keeps track of internal state after subscription, which can be asserted through its various `assert*` methods.

It can be used as a `Flux` or `Mono` by using the conversion methods `flux()` and `mono()`.

6.6. Checking the Execution Path with `PublisherProbe`

When building complex chains of operators, you could come across cases where there are several possible execution paths, materialized by distinct sub-sequences.

Most of the time, these sub-sequences produce a specific-enough `onNext` signal that you can assert it was executed by looking at the end result.

For instance, consider the following method, which builds a chain of operators from a source and uses a `switchIfEmpty` to fallback to a particular alternative if the source is empty:

```
public Flux<String> processOrFallback(Mono<String> source, Publisher<String> fallback)
{
    return source
            .flatMapMany(phrase -> Flux.fromArray(phrase.split("\\s+")))
            .switchIfEmpty(fallback);
}
```

It is easy enough to test which logical branch of the switchIfEmpty was used, as follows:

```
@Test
public void testSplitPathIsUsed() {
    StepVerifier.create(processOrFallback(Mono.just("just a  phrase with    tabs!"),
            Mono.just("EMPTY_PHRASE")))
                .expectNext("just", "a", "phrase", "with", "tabs!")
                .verifyComplete();
}

@Test
public void testEmptyPathIsUsed() {
    StepVerifier.create(processOrFallback(Mono.empty(), Mono.just("EMPTY_PHRASE")))
                .expectNext("EMPTY_PHRASE")
                .verifyComplete();
}
```

But think about an example where the method produces a `Mono<Void>` instead. It waits for the source to complete, performs an additional task, and completes. If the source is empty, a fallback Runnable-like task must be performed instead, as follows:

```
private Mono<String> executeCommand(String command) {
    return Mono.just(command + " DONE");
}

public Mono<Void> processOrFallback(Mono<String> commandSource, Mono<Void>
doWhenEmpty) {
    return commandSource
            .flatMap(command -> executeCommand(command).then()) ①
            .switchIfEmpty(doWhenEmpty); ②
}
```

① The `then()` forgets about the command result. It cares only that it was completed.

② How to distinguish between two cases that both are empty sequences?

In order to verify that your processOrFallback indeed goes through the `doWhenEmpty` path, you need to write a bit of boilerplate. Namely you need a `Mono<Void>` that:

- Captures the fact that it has been subscribed to
- Lets you assert that fact **after** the whole processing has terminated.

Before version 3.1, you would need to manually maintain one `AtomicBoolean` per state you wanted to assert and attach a corresponding `doOn*` callback to the publisher you wanted to evaluate. This could be a lot of boilerplate when having to apply this pattern regularly. Fortunately, since 3.1.0 there's an alternative with `PublisherProbe`, as follows:

```
@Test
public void testCommandEmptyPathIsUsed() {
    PublisherProbe<Void> probe = PublisherProbe.empty(); ①

    StepVerifier.create(processOrFallback(Mono.empty(), probe.mono())) ②
                .verifyComplete();

    probe.assertWasSubscribed(); ③
    probe.assertWasRequested(); ④
    probe.assertWasNotCancelled(); ⑤
}
```

① Create a probe that translates to an empty sequence.

② Use the probe in place of `Mono<Void>` by calling `probe.mono()`.

③ After completion of the sequence, the probe lets you assert that it was used. You can check that is was subscribed to…

④ …as well as actually requested for data…

⑤ …and whether or not it was cancelled.

You can also use the probe in place of a `Flux<T>` by calling `.flux()` instead of `.mono()`. For cases where you need to probe an execution path but also need the probe to emit data, you can wrap any

`Publisher<T>` using `PublisherProbe.of(Publisher)`.

Chapter 7. Debugging Reactor

Switching from an imperative and synchronous programming paradigm to a reactive and asynchronous one can sometimes be daunting. One of the steepest steps in the learning curve is how to analyze and debug when something goes wrong.

In the imperative world, debugging is usually pretty straightforward: just read the stacktrace and you see where the problem originated and more: Was it entirely a failure of your code? Did the failure occur in some library code? If so, what part of your code called the library, potentially passing in improper parameters that ultimately caused the failure?

7.1. The Typical Reactor Stack Trace

When you shift to asynchronous code, things can get much more complicated.

Consider the following stack trace:

A typically Reactor stack trace

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
    at reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:129)
    at
reactor.core.publisher.FluxFlatMap$FlatMapMain.tryEmitScalar(FluxFlatMap.java:445)
    at reactor.core.publisher.FluxFlatMap$FlatMapMain.onNext(FluxFlatMap.java:379)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onNext(FluxMapFuseable.ja
va:121)
    at reactor.core.publisher.FluxRange$RangeSubscription.slowPath(FluxRange.java:154)
    at reactor.core.publisher.FluxRange$RangeSubscription.request(FluxRange.java:109)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.request(FluxMapFuseable.j
ava:162)
    at
reactor.core.publisher.FluxFlatMap$FlatMapMain.onSubscribe(FluxFlatMap.java:332)
    at
reactor.core.publisher.FluxMapFuseable$MapFuseableSubscriber.onSubscribe(FluxMapFuseab
le.java:90)
    at reactor.core.publisher.FluxRange.subscribe(FluxRange.java:68)
    at reactor.core.publisher.FluxMapFuseable.subscribe(FluxMapFuseable.java:63)
    at reactor.core.publisher.FluxFlatMap.subscribe(FluxFlatMap.java:97)
    at reactor.core.publisher.MonoSingle.subscribe(MonoSingle.java:58)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3096)
    at reactor.core.publisher.Mono.subscribeWith(Mono.java:3204)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3090)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3057)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3029)
    at reactor.guide.GuideTests.debuggingCommonStacktrace(GuideTests.java:995)
```

There is a lot going on there. We get an `IndexOutOfBoundsException`, which tells us that a "**source**

emitted *more than one item*".

We can probably quickly come to assume that this source is a Flux/Mono, as confirmed by the line below that mentions `MonoSingle`. So it appears to be some sort of complaint from a `single` operator.

Referring to the Javadoc for `Mono#single` operator, we see that `single` has a contract: The source must emit exactly one element. It appears we had a source that emitted more than one and thus violated that contract.

Can we dig deeper and identify that source? The following rows are not very helpful. They take us on a travel inside the internals of what seems to be a reactive chain, through multiple calls to `subscribe` and `request`.

By skimming over these rows, we can at least start to form a picture of the kind of chain that went wrong: It seems to involve a `MonoSingle`, a `FluxFlatMap`, and a `FluxRange` (each gets several rows in the trace, but overall these three classes are involved). So a `range().flatMap().single()` chain maybe?

But what if we use that pattern a lot in our application? This still does not tell us much, and simply searching for `single` isn't going to find the problem. Then the last line refers to some of our code. Finally, we are getting close.

Hold on, though. When we go to the source file, all we see is that a pre-existing `Flux` is subscribed to, as follows:

```
toDebug.subscribe(System.out::println, Throwable::printStackTrace);
```

All of this happened at subscription time, but the `Flux` itself was not *declared* there. Worse, when we go to where the variable is declared, we see:

```
public Mono<String> toDebug; //please overlook the public class attribute
```

The variable is not *instantiated* where it is declared. We must assume a worst-case scenario where we find out that there could be a few different code paths that set it in the application. We remain unsure of which one caused the issue.

This is kind of the Reactor equivalent of a runtime error, as opposed to a compilation error.

What we want to find out more easily is where the operator was added into the chain - that is, where the `Flux` was declared. We usually refer to that as the **assembly** of the `Flux`.

7.2. Activating Debug Mode - aka tracebacks

Even though the stacktrace was still able to convey some information for someone with a bit of experience, we can see that it is not ideal by itself in more advanced cases.

Fortunately, Reactor comes with a debugging-oriented capability of **assembly-time instrumentation**.

This is done by customizing the `Hooks.onOperator` hook **at application start** (or at least before the incriminated `Flux` or `Mono` can be instantiated), as follows:

```
Hooks.onOperatorDebug();
```

This starts instrumenting the calls to the `Flux` (and `Mono`) operator methods (where they are assembled into the chain) by wrapping the construction of the operator and capturing a stacktrace there. Since this is done when the operator chain is declared, the hook should be activated **before** that, so the safest way is to activate it right at the start of your application.

Later on, if an exception occurs, the failing operator is able to refer to that capture and append it to the stack trace. We call this captured assembly information a **traceback**.

In the next section, we see how the stack trace differs and how to interpret that new information.

7.3. Reading a Stack Trace in Debug Mode

When we reuse our initial example but activate the `operatorStacktrace` debug feature, the stack trace is as follows:

```
java.lang.IndexOutOfBoundsException: Source emitted more than one item
    at reactor.core.publisher.MonoSingle$SingleSubscriber.onNext(MonoSingle.java:129)
    at
reactor.core.publisher.FluxOnAssembly$OnAssemblySubscriber.onNext(FluxOnAssembly.java:
375) ①
...
②
...
    at reactor.core.publisher.Mono.subscribeWith(Mono.java:3204)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3090)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3057)
    at reactor.core.publisher.Mono.subscribe(Mono.java:3029)
    at reactor.guide.GuideTests.debuggingActivated(GuideTests.java:1000)
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException: ③
Assembly trace from producer [reactor.core.publisher.MonoSingle] : ④
    reactor.core.publisher.Flux.single(Flux.java:6676)
    reactor.guide.GuideTests.scatterAndGather(GuideTests.java:949)
    reactor.guide.GuideTests.populateDebug(GuideTests.java:962)
    org.junit.rules.TestWatcher$1.evaluate(TestWatcher.java:55)
    org.junit.rules.RunRules.evaluate(RunRules.java:20)
Error has been observed by the following operator(s): ⑤
    |_  Flux.single    reactor.guide.GuideTests.scatterAndGather(GuideTests.java:949)
⑥
```

① This is new: We see the wrapper operator that captures the stack.

② Apart from that, the first section of the stack trace is still the same for the most part, showing a bit of the operator's internals (so we removed a bit of the snippet here).

③ This is where the traceback starts to appear.

④ First, we get some details on where the operator was assembled.

⑤ We also get a traceback of the error as it propagated through the operator chain, from first to last (error site to subscribe site).

⑥ Each operator that saw the error is mentioned along with the user class and line where it was used.

As you can see, the captured stack trace is appended to the original error as a suppressed `OnAssemblyException`. There are two parts to it, but the first section is the most interesting. It shows the path of construction for the operator that triggered the exception. Here it shows that the `single` that caused our issue was created in the `scatterAndGather` method, itself called from a `populateDebug` method that got executed through JUnit.

Now that we are armed with enough information to find the culprit, we can have a meaningful look at that `scatterAndGather` method:

```
private Mono<String> scatterAndGather(Flux<String> urls) {
    return urls.flatMap(url -> doRequest(url))
           .single(); ①
}
```

① Sure enough, here is our `single`.

Now we can see what the root cause of the error was a `flatMap` that performs several HTTP calls to a few URLs is chained with `single`, which is too restrictive. After a short `git blame` and a quick discussion with the author of that line, we find out he meant to use the less restrictive `take(1)` instead.

We have solved our problem.

> Error has been observed by the following operator(s):

That second part of the debug stack trace was not necessarily interesting in this particular example, because the error was actually happening in the last operator in the chain (the one closest to `subscribe`). Considering another example might make it more clear:

```
FakeRepository.findAllUserByName(Flux.just("pedro", "simon", "stephane"))
              .transform(FakeUtils1.applyFilters)
              .transform(FakeUtils2.enrichUser)
              .blockLast();
```

Now imagine that, inside `findAllUserByName`, there is a `map` that fails. Here we would see the following final traceback:

```
Error has been observed by the following operator(s):
    |_  Flux.map
reactor.guide.FakeRepository.findAllUserByName(FakeRepository.java:27)
    |_  Flux.map
reactor.guide.FakeRepository.findAllUserByName(FakeRepository.java:28)
    |_  Flux.filter    reactor.guide.FakeUtils1.lambda$static$1(FakeUtils1.java:29)
    |_  Flux.transform
reactor.guide.GuideDebuggingExtraTests.debuggingActivatedWithDeepTraceback(GuideDebugg
ingExtraTests.java:40)
    |_  Flux.elapsed    reactor.guide.FakeUtils2.lambda$static$0(FakeUtils2.java:30)
    |_  Flux.transform
reactor.guide.GuideDebuggingExtraTests.debuggingActivatedWithDeepTraceback(GuideDebugg
ingExtraTests.java:41)
```

This corresponds to the section of the chain of operators that gets notified of the error:

1. The exception originates in the first `map`.
2. It is seen by a second `map` (both in fact correspond to the `findAllUserByName` method).
3. It is then seen by a `filter` and a `transform`, which indicate that part of the chain is constructed via a reusable transformation function (here, the `applyFilters` utility method).
4. Finally, it is seen by an `elapsed` and a `transform`. Once again, `elapsed` is applied by the transformation function of that second transform.

As tracebacks are appended to original errors as suppressed exceptions, this can somewhat interfere with another type of exception that uses this mechanism: composite exceptions. Such exceptions can be created directly via `Exceptions.multiple(Throwable…)`, or by some operators that might join multiple erroring sources (like `Flux#flatMapDelayError`). They can be unwrapped into a `List` via `Exceptions.unwrapMultiple(Throwable)`, in which case the traceback would be considered a component of the composite and be part of the returned `List`. If that is somehow not desirable, tracebacks can be identified thanks to `Exceptions.isTraceback(Throwable)` check, and excluded from such an unwrap by using `Exceptions.unwrapMultipleExcludingTracebacks(Throwable)` instead.

We deal with a form of instrumentation here, and creating a stack trace is costly. That is why this debugging feature should only be activated in a controlled manner, as a last resort.

7.3.1. The `checkpoint()` Alternative

The debug mode is global and affects every single operator assembled into a `Flux` or `Mono` inside the application. This has the benefit of allowing **after-the-fact debugging**: whatever the error, we will obtain additional info to debug it.

As we saw earlier, this global knowledge comes at the cost of an impact on performance (due to the number of populated stack traces). That cost can be reduced if we have an idea of likely problematic operators. However, we usually do not know which operators are likely to be problematic unless we observed an error in the wild, saw we were missing assembly information, and then modified the code to activate assembly tracking, hoping to observe the same error again.

In that scenario, we have to switch into debugging mode and make preparations in order to better observe a second occurrence of the error, this time capturing all the additional information.

If you can identify reactive chains that you assemble in your application for which serviceability is critical, **a mix of both techniques can be achieved with the `checkpoint()` operator.**

You can chain this operator into a method chain. The `checkpoint` operator works like the hook version, but only for its link of that particular chain.

There is also a `checkpoint(String)` variant that lets you add a unique String identifier to the assembly traceback. This way, the stack trace is omitted and you rely on the description to identify the assembly site. A `checkpoint(String)` imposes less processing cost than a regular `checkpoint`.

`checkpoint(String)` includes "light" in its output (which can be handy when searching), as shown in the following example:

```
...
    Suppressed: reactor.core.publisher.FluxOnAssembly$OnAssemblyException:
Assembly site of producer [reactor.core.publisher.ParallelSource] is identified by
light checkpoint [light checkpoint identifier].
```

Last but not least, if you want to add a more generic description to the checkpoint but still rely on the stack trace mechanism to identify the assembly site, you can force that behavior by using the `checkpoint("description", true)` version. We are now back to the initial message for the traceback, augmented with a `description`, as shown in the following example:

```
Assembly trace from producer [reactor.core.publisher.ParallelSource], described as
[descriptionCorrelation1234] : ①
    reactor.core.publisher.ParallelFlux.checkpoint(ParallelFlux.java:215)

reactor.core.publisher.FluxOnAssemblyTest.parallelFluxCheckpointDescriptionAndForceSta
ck(FluxOnAssemblyTest.java:225)
Error has been observed by the following operator(s):
    |_  ParallelFlux.checkpoint
reactor.core.publisher.FluxOnAssemblyTest.parallelFluxCheckpointDescriptionAndForceSta
ck(FluxOnAssemblyTest.java:225)
```

① `descriptionCorrelation1234` is the description provided in the `checkpoint`.

The description could be a static identifier or user-readable description or a wider **correlation ID** (for instance, coming from a header in the case of an HTTP request).

When both global debugging and local `checkpoint()` are enabled, checkpointed snapshot stacks are appended as suppressed error output after the observing operator graph and following the same declarative order.

==

7.4. Logging a sequence

In addition to stack trace debugging and analysis, another powerful tool to have in your toolkit is the ability to trace and log events in an asynchronous sequence.

The `log()` operator can do just that. Chained inside a sequence, it will peek at every event of the `Flux` or `Mono` upstream of it (including `onNext`, `onError`, and `onComplete` and *subscriptions*, *cancellations*, and *requests*).

Side note on logging implementation

The `log` operator uses the `Loggers` utility class, which picks up common logging frameworks like Log4J and Logback through **`SLF4J`** and defaults to logging to the console in case SLF4J is unavailable.

The Console fallback uses `System.err` for the `WARN` and `ERROR` log levels and `System.out` for everything else.

If you prefer a JDK `java.util.logging` fallback, as in 3.0.x, you can get it by setting the `reactor.logging.fallback` System property to `JDK`.

In all cases, when logging in production **you should take care to configure the underlying logging framework to use its most asynchronous and non-blocking approach**. For instance, an `AsyncAppender` in logback or `AsyncLogger` in Log4j 2.

For instance, suppose we have *logback* activated and configured and a chain like `range(1,10).take(3)`. By placing a `log()` just before the *take*, we can get some insight into how it works and what kind of events it propagates upstream to the *range*, as shown in the following example:

```
Flux<Integer> flux = Flux.range(1, 10)
                         .log()
                         .take(3);
flux.subscribe();
```

This prints out (through the logger's console appender):

```
10:45:20.200 [main] INFO  reactor.Flux.Range.1 - | onSubscribe([Synchronous Fuseable]
FluxRange.RangeSubscription) ①
10:45:20.205 [main] INFO  reactor.Flux.Range.1 - | request(unbounded) ②
10:45:20.205 [main] INFO  reactor.Flux.Range.1 - | onNext(1) ③
10:45:20.205 [main] INFO  reactor.Flux.Range.1 - | onNext(2)
10:45:20.205 [main] INFO  reactor.Flux.Range.1 - | onNext(3)
10:45:20.205 [main] INFO  reactor.Flux.Range.1 - | cancel() ④
```

Here, in addition to the logger’s own formatter (time, thread, level, message), the `log()` operator outputs a few things in its own format:

① `reactor.Flux.Range.1` is an automatic *category* for the log, in case you use the operator several times in a chain. It allows you to distinguish which operator’s events are logged (in this case, the `range`). The identifier can be overwritten with your own custom category by using the `log(String)` method signature. After a few separating characters, the actual event gets printed. Here we get an `onSubscribe` call, an `request` call, three `onNext` calls, and a `cancel` call. For the first line, `onSubscribe`, we get the implementation of the `Subscriber`, which usually corresponds to the operator-specific implementation. Between square brackets, we get additional information, including whether the operator can be automatically optimized via synchronous or asynchronous fusion.

② On the second line, we can see that an unbounded request was propagated up from downstream.

③ Then the range sends three values in a row.

④ On the last line, we see `cancel()`.

The last line, **(4)**, is the most interesting. We can see the `take` in action there. It operates by cutting the sequence short after it has seen enough elements emitted. In short, `take()` causes the source to `cancel()` once it has emitted the user-requested amount.

Chapter 8. Advanced Features and Concepts

This chapter covers advanced features and concepts of Reactor, including the following:

- Mutualizing Operator Usage
- Hot vs Cold
- Broadcasting to Multiple Subscribers with `ConnectableFlux`
- Three Sorts of Batching
- Parallelizing Work with `ParallelFlux`
- Replacing Default `Schedulers`
- Using Global Hooks
- Adding a Context to a Reactive Sequence
- Null-safety
- Dealing with Objects that need cleanup

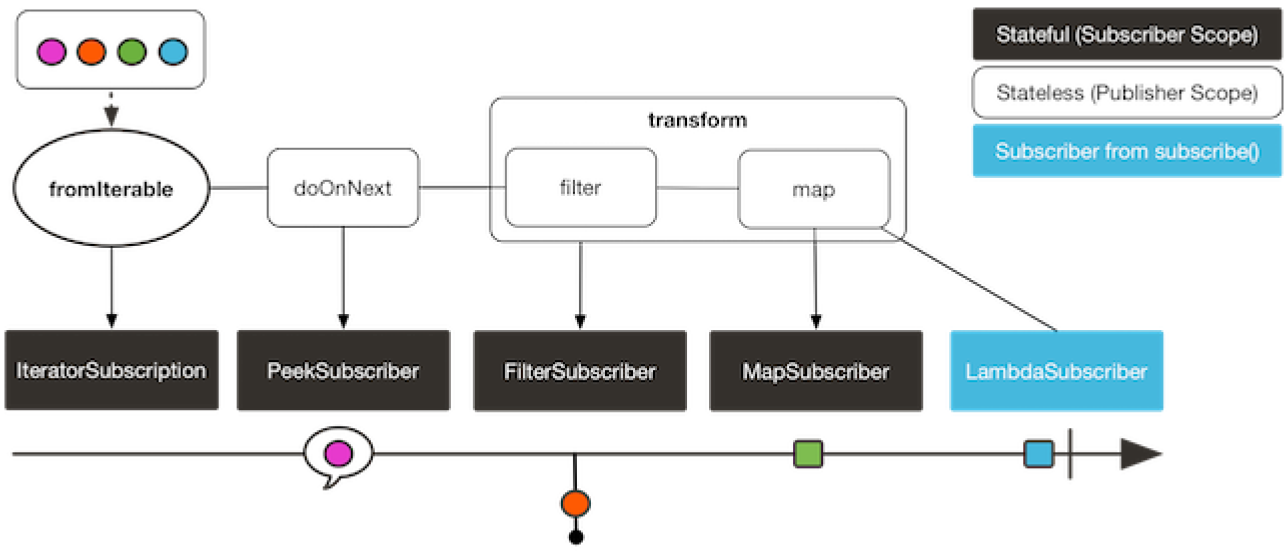
8.1. Mutualizing Operator Usage

From a clean-code perspective, code reuse is generally a good thing. Reactor offers a few patterns that can help you reuse and mutualize code, notably for operators or combination of operators that you might want to apply regularly in your codebase. If you think of a chain of operators as a recipe, you can create a cookbook of operator recipes.

8.1.1. Using the `transform` Operator

The `transform` operator lets you encapsulate a piece of an operator chain into a function. That function is applied to an original operator chain at assembly time to augment it with the encapsulated operators. Doing so applies the same operations to all the subscribers of a sequence and is basically equivalent to chaining the operators directly. The following code shows an example:

```
Function<Flux<String>, Flux<String>> filterAndMap =
f -> f.filter(color -> !color.equals("orange"))
      .map(String::toUpperCase);

Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))
    .doOnNext(System.out::println)
    .transform(filterAndMap)
    .subscribe(d -> System.out.println("Subscriber to Transformed MapAndFilter: "+d));
```



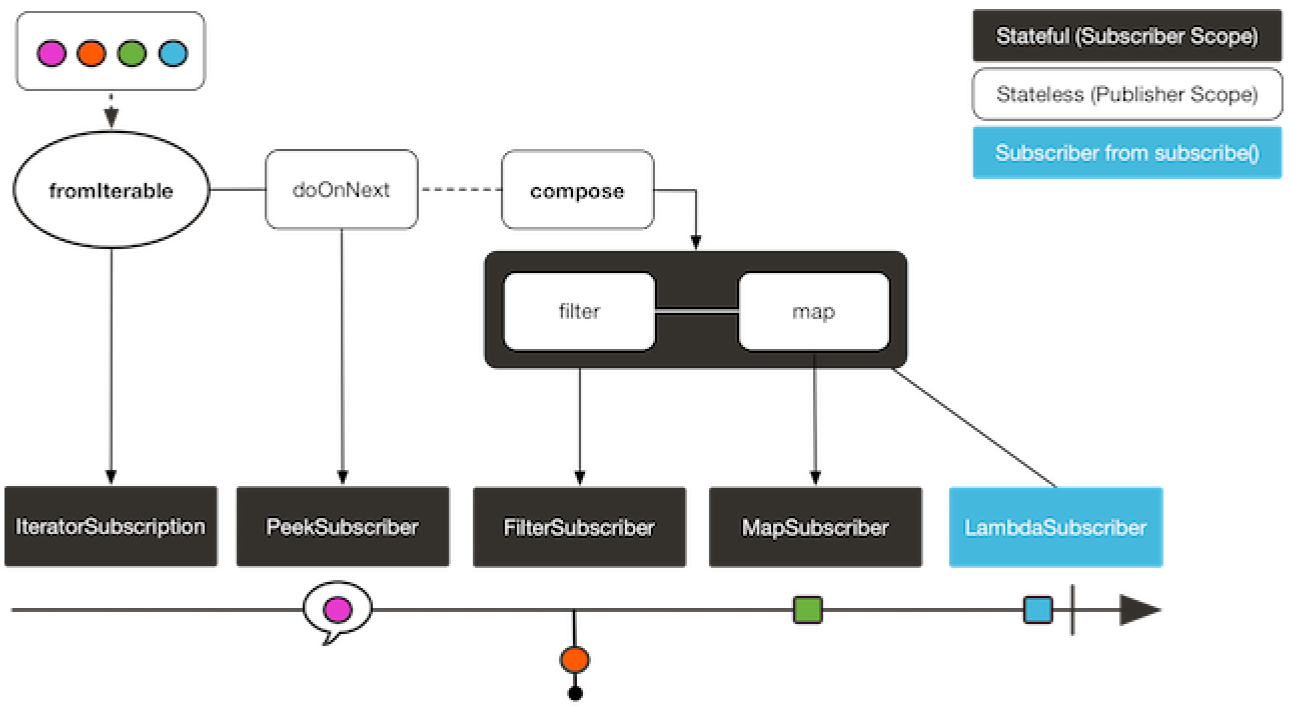
The preceding example produces the following output:

```
blue
Subscriber to Transformed MapAndFilter: BLUE
green
Subscriber to Transformed MapAndFilter: GREEN
orange
purple
Subscriber to Transformed MapAndFilter: PURPLE
```

8.1.2. Using the `compose` Operator

The `compose` operator is similar to `transform` and also lets you encapsulate operators in a function. The major difference is that this function is applied to the original sequence **on a per-subscriber basis**. It means that the function can actually produce a different operator chain for each subscription (by maintaining some state). The following code shows an example:

```
AtomicInteger ai = new AtomicInteger();
Function<Flux<String>, Flux<String>> filterAndMap = f -> {
    if (ai.incrementAndGet() == 1) {
return f.filter(color -> !color.equals("orange"))
        .map(String::toUpperCase);
    }
    return f.filter(color -> !color.equals("purple"))
            .map(String::toUpperCase);
};

Flux<String> composedFlux =
Flux.fromIterable(Arrays.asList("blue", "green", "orange", "purple"))
    .doOnNext(System.out::println)
    .compose(filterAndMap);

composedFlux.subscribe(d -> System.out.println("Subscriber 1 to Composed MapAndFilter
:"+d));
composedFlux.subscribe(d -> System.out.println("Subscriber 2 to Composed MapAndFilter:
"+d));
```



The preceding example produces the following output:

```
blue
Subscriber 1 to Composed MapAndFilter :BLUE
green
Subscriber 1 to Composed MapAndFilter :GREEN
orange
purple
Subscriber 1 to Composed MapAndFilter :PURPLE
blue
Subscriber 2 to Composed MapAndFilter: BLUE
green
Subscriber 2 to Composed MapAndFilter: GREEN
orange
Subscriber 2 to Composed MapAndFilter: ORANGE
purple
```

8.2. Hot vs Cold

So far, we have considered that all `Flux` (and `Mono`) are the same: They all represent an asynchronous sequence of data, and nothing happens before you subscribe.

Really, though, there are two broad families of publishers: **hot** and **cold**.

The description above applies to the **cold** family of publishers. They generate data anew for each subscription. If no subscription is created, then data never gets generated.

Think of an HTTP request: Each new subscriber will trigger an HTTP call, but no call is made if no one is interested in the result.

Hot publishers, on the other hand, do not depend on any number of subscribers. They might start publishing data right away and would continue doing so whenever a new `Subscriber` comes in (in which case said subscriber would only see new elements emitted *after* it subscribed). For hot publishers, *something* does indeed happen before you subscribe.

One example of the few hot operators in Reactor is `just`: It directly captures the value at assembly time and replays it to anybody subscribing to it later. To re-use the HTTP call analogy, if the captured data is the result of an HTTP call, then only one network call is made, when instantiating *just*.

To transform `just` into a *cold* publisher, you can use `defer`. It defers the HTTP request in our example to subscription time (and would result in a separate network call for each new subscription).

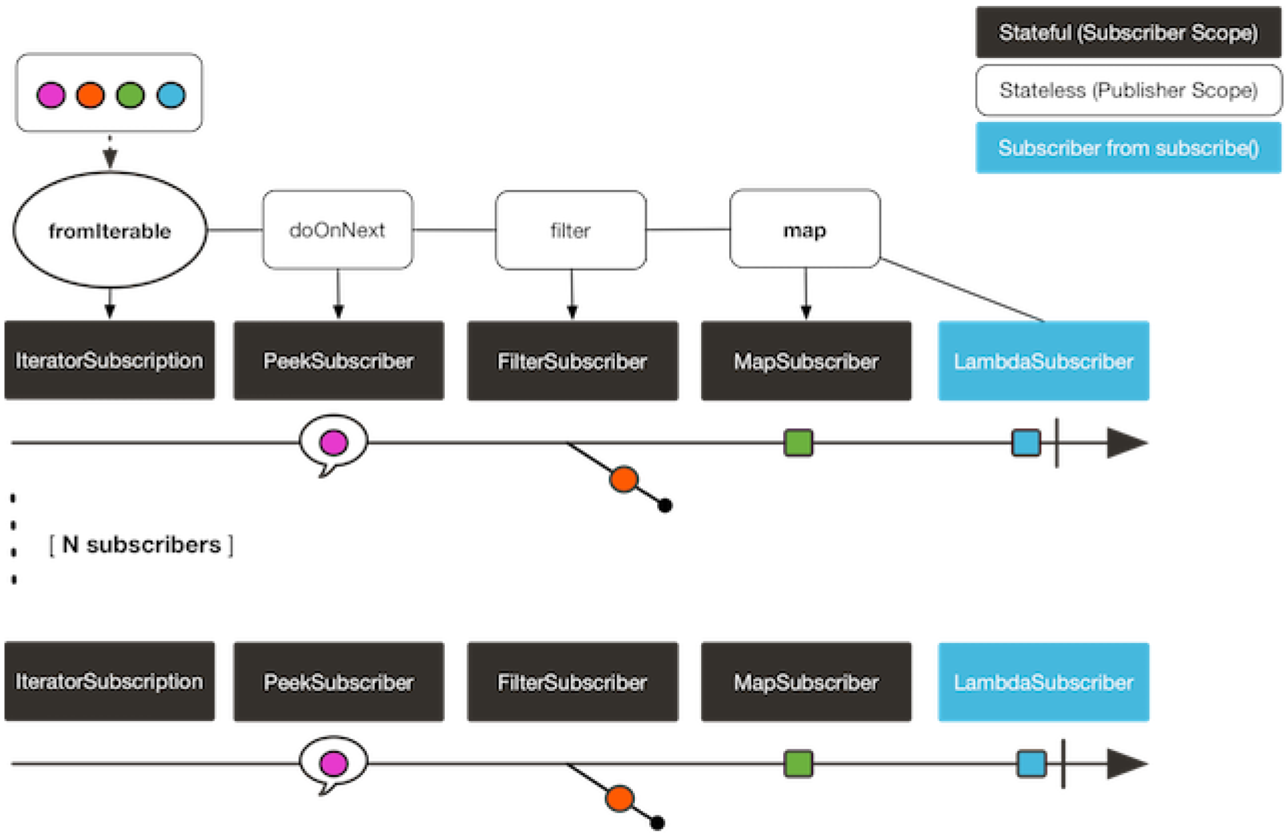
Most other *hot* publishers in Reactor extend `Processor`.

Consider two other examples. The following code shows the first example:

```
Flux<String> source = Flux.fromIterable(Arrays.asList("blue", "green", "orange",
"purple"))
                          .map(String::toUpperCase);

source.subscribe(d -> System.out.println("Subscriber 1: "+d));
source.subscribe(d -> System.out.println("Subscriber 2: "+d));
```

This first example produces the following output:

```
Subscriber 1: BLUE
Subscriber 1: GREEN
Subscriber 1: ORANGE
Subscriber 1: PURPLE
Subscriber 2: BLUE
Subscriber 2: GREEN
Subscriber 2: ORANGE
Subscriber 2: PURPLE
```



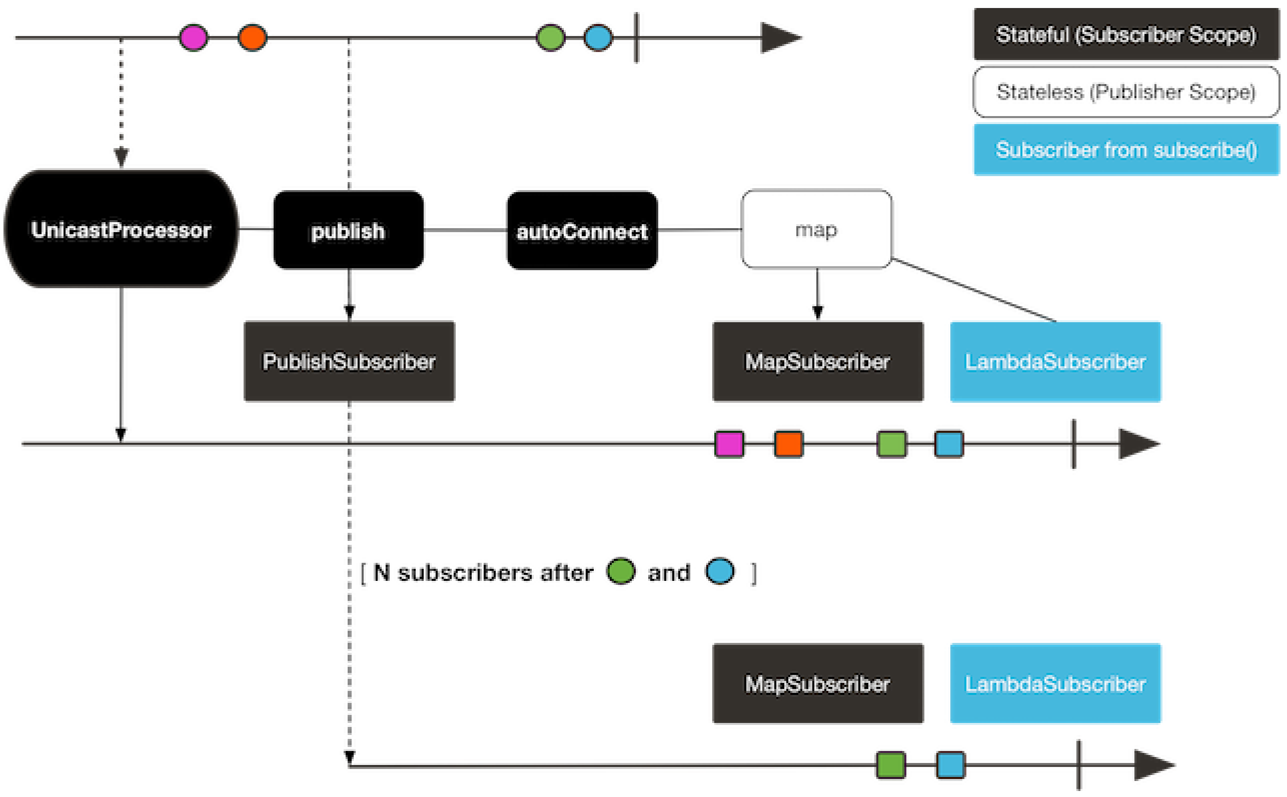
Both subscribers catch all four colors, because each subscriber causes the process defined by the operators on the `Flux` to run.

Compare the first example to the second example, shown in the following code:

```
DirectProcessor<String> hotSource = DirectProcessor.create();

Flux<String> hotFlux = hotSource.map(String::toUpperCase);


hotFlux.subscribe(d -> System.out.println("Subscriber 1 to Hot Source: "+d));

hotSource.onNext("blue");
hotSource.onNext("green");

hotFlux.subscribe(d -> System.out.println("Subscriber 2 to Hot Source: "+d));

hotSource.onNext("orange");
hotSource.onNext("purple");
hotSource.onComplete();
```

The second example produces the following output:

```
Subscriber 1 to Hot Source: BLUE
Subscriber 1 to Hot Source: GREEN
Subscriber 1 to Hot Source: ORANGE
Subscriber 2 to Hot Source: ORANGE
Subscriber 1 to Hot Source: PURPLE
Subscriber 2 to Hot Source: PURPLE
```


Stateful (Subscriber Scope)
Stateless (Publisher Scope)
Subscriber from subscribe()
UnicastProcessor
publish
autoConnect
map
PublishSubscriber
MapSubscriber
LambdaSubscriber
[N subscribers after ● and ●]
MapSubscriber
LambdaSubscriber

Subscriber 1 catches all four colors. Subscriber 2, having been created after the first two colors were produced, catches only the last two colors. This difference accounts for the doubling of "ORANGE" and "PURPLE" in the output. The process described by the operators on this Flux runs regardless of when subscriptions have been attached.

8.3. Broadcasting to Multiple Subscribers with `ConnectableFlux`

Sometimes, you want to not only defer some processing to the subscription time of one subscriber, but you might actually want for several of them to *rendezvous* and **then** trigger the subscription and data generation.

This is what `ConnectableFlux` is made for. Two main patterns are covered in the `Flux` API that return a `ConnectableFlux`: `publish` and `replay`.

- `publish` dynamically tries to respect the demand from its various subscribers, in terms of backpressure, by forwarding these requests to the source. Most notably, if any subscriber has a pending demand of `0`, publish **pauses** its requesting to the source.
- `replay` buffers data seen through the first subscription, up to configurable limits (in time and buffer size). It replays the data to subsequent subscribers.

A `ConnectableFlux` offers additional methods to manage subscriptions downstream versus subscriptions to the original source. These additional methods include the following:

- `connect()` can be called manually once you reach enough subscriptions to the flux. That triggers the subscription to the upstream source.
- `autoConnect(n)` can do the same job automatically once `n` subscriptions have been made.
- `refCount(n)` not only automatically tracks incoming subscriptions but also detects when these subscriptions are cancelled. If not enough subscribers are tracked, the source is "disconnected", causing a new subscription to the source later if additional subscribers appear.
- `refCount(int, Duration)` adds a "grace period": Once the number of tracked subscribers becomes too low, it waits for the `Duration` before disconnecting the source, potentially allowing for enough new subscribers to come in and cross the connection threshold again.

Consider the following example:

```
Flux<Integer> source = Flux.range(1, 3)
                           .doOnSubscribe(s -> System.out.println("subscribed to
source"));

ConnectableFlux<Integer> co = source.publish();

co.subscribe(System.out::println, e -> {}, () -> {});
co.subscribe(System.out::println, e -> {}, () -> {});

System.out.println("done subscribing");
Thread.sleep(500);
System.out.println("will now connect");

co.connect();
```

The preceding code produces the following output:

```
done subscribing
will now connect
subscribed to source
1
1
2
2
3
3
```

With `autoConnect`:

```
Flux<Integer> source = Flux.range(1, 3)
                           .doOnSubscribe(s -> System.out.println("subscribed to
source"));

Flux<Integer> autoCo = source.publish().autoConnect(2);

autoCo.subscribe(System.out::println, e -> {}, () -> {});
System.out.println("subscribed first");
Thread.sleep(500);
System.out.println("subscribing second");
autoCo.subscribe(System.out::println, e -> {}, () -> {});
```

The preceding code produces the following output:

```
subscribed first
subscribing second
subscribed to source
1
1
2
2
3
3
```

8.4. Three Sorts of Batching

When you have lots of elements and you want to separate them into batches, you have three broad solutions in Reactor: grouping, windowing, and buffering. These three are conceptually close, because they redistribute a `Flux<T>` into an aggregate. Grouping and windowing create a `Flux<Flux<T>>`, while buffering aggregates into a `Collection<T>`.

8.4.1. Grouping with `Flux<GroupedFlux<T>>`

Grouping is the act of splitting the source `Flux<T>` into multiple batches by a **key**.

The associated operator is `groupBy`.

Each group is represented as a `GroupedFlux<T>`, which lets you retrieve the key via its `key()` method.

There is no necessary continuity in the content of the groups. Once a source element produces a new key, the group for this key is opened and elements that match the key end up in the group (several groups could be open at the same time).

This means that groups:

1. Are always disjoint (a source element belongs to 1 and only 1 group).
2. Can contain elements from different places in the original sequence.
3. Are never empty.

```
StepVerifier.create(
    Flux.just(1, 3, 5, 2, 4, 6, 11, 12, 13)
        .groupBy(i -> i % 2 == 0 ? "even" : "odd")
        .concatMap(g -> g.defaultIfEmpty(-1) //if empty groups, show them
                .map(String::valueOf) //map to string
                .startWith(g.key())) //start with the group's key
    )
    .expectNext("odd", "1", "3", "5", "11", "13")
    .expectNext("even", "2", "4", "6", "12")
    .verifyComplete();
```

Grouping is best suited for when you have a medium to low number of groups. The groups must also imperatively be consumed (such as by a `flatMap`) so that `groupBy` continues fetching data from upstream and feeding more groups. Sometimes, these two constraints multiply and lead to hangs, such as when you have a high cardinality and the concurrency of the `flatMap` consuming the groups is too low.

8.4.2. Windowing with `Flux<Flux<T>>`

Windowing is the act of splitting the source `Flux<T>` into *windows*, by criteria of size, time, boundary-defining predicates, or boundary-defining `Publisher`.

The associated operators are `window`, `windowTimeout`, `windowUntil`, `windowWhile`, and `windowWhen`.

Contrary to `groupBy`, which randomly overlaps according to incoming keys, most of the time windows are opened sequentially.

Some variants **can** still overlap, though. For instance in `window(int maxSize, int skip)` the `maxSize` parameter is the number of elements after which a window closes, and the `skip` parameter is the number of elements in the source after which a new window is opened. So if `maxSize > skip`, a new window opens before the previous one closes and the two windows overlap.

The following example shows overlapping windows:

```
StepVerifier.create(
    Flux.range(1, 10)
        .window(5, 3) //overlapping windows
        .concatMap(g -> g.defaultIfEmpty(-1)) //show empty windows as -1
    )
        .expectNext(1, 2, 3, 4, 5)
        .expectNext(4, 5, 6, 7, 8)
        .expectNext(7, 8, 9, 10)
        .expectNext(10)
        .verifyComplete();
```

With the reverse configuration (`maxSize` < `skip`), some elements from the source are dropped and are not part of any window.

In the case of predicate-based windowing via `windowUntil` and `windowWhile`, having subsequent source elements that do not match the predicate can also lead to *empty windows*, as demonstrated in the following example:

```
StepVerifier.create(
    Flux.just(1, 3, 5, 2, 4, 6, 11, 12, 13)
        .windowWhile(i -> i % 2 == 0)
        .concatMap(g -> g.defaultIfEmpty(-1))
    )
        .expectNext(-1, -1, -1) //respectively triggered by odd 1 3 5
        .expectNext(2, 4, 6) // triggered by 11
        .expectNext(12) // triggered by 13
        // however, no empty completion window is emitted (would contain extra
matching elements)
        .verifyComplete();
```

8.4.3. Buffering with `Flux<List<T>>`

Buffering is similar to windowing, with the following twist: instead of emitting *windows* (which are each a `Flux<T>`), it emits *buffers* (which are `Collection<T>` - by default, `List<T>`).

The operators for buffering mirror those for windowing: `buffer`, `bufferTimeout`, `bufferUntil`, `bufferWhile`, and `bufferWhen`.

Where the corresponding windowing operator opens a window, a buffering operator creates a new collection and start adding elements to it. Where a window closes, the buffering operator emits the collection.

Buffering can also lead to dropping source elements or having overlapping buffers, as shown here:

```
StepVerifier.create(
    Flux.range(1, 10)
        .buffer(5, 3) //overlapping buffers
    )
        .expectNext(Arrays.asList(1, 2, 3, 4, 5))
        .expectNext(Arrays.asList(4, 5, 6, 7, 8))
        .expectNext(Arrays.asList(7, 8, 9, 10))
        .expectNext(Collections.singletonList(10))
        .verifyComplete();
```

Unlike in windowing, `bufferUntil` and `bufferWhile` do not emit an empty buffer, as shown in the following example:

```
StepVerifier.create(
    Flux.just(1, 3, 5, 2, 4, 6, 11, 12, 13)
        .bufferWhile(i -> i % 2 == 0)
    )
    .expectNext(Arrays.asList(2, 4, 6)) // triggered by 11
    .expectNext(Collections.singletonList(12)) // triggered by 13
    .verifyComplete();
```

8.5. Parallelizing Work with `ParallelFlux`

With multi-core architectures being a commodity nowadays, being able to easily parallelize work is important. Reactor helps with that by providing a special type, `ParallelFlux`, that exposes operators that are optimized for parallelized work.

To obtain a `ParallelFlux`, you can use the `parallel()` operator on any `Flux`. **By itself, this method does not parallelize the work**. Rather, it divides the workload into "rails" (by default, as many rails as there are CPU cores).

In order to tell the resulting ParallelFlux where to execute each rail (and, by extension, to execute rails in parallel) you have to use `runOn(Scheduler)`. Note that there is a recommended dedicated Scheduler for parallel work: `Schedulers.parallel()`.

Compare the next two examples, the first of which is shown in the following code:

```
Flux.range(1, 10)
    .parallel(2) ①
    .subscribe(i -> System.out.println(Thread.currentThread().getName() + " -> " +
i));
```

① We force a number of rails instead of relying on the number of CPU cores.

The following code shows the second example:

```
Flux.range(1, 10)
    .parallel(2)
    .runOn(Schedulers.parallel())
    .subscribe(i -> System.out.println(Thread.currentThread().getName() + " -> " +
i));
```

The first example produces the following output:

```
main -> 1
main -> 2
main -> 3
main -> 4
main -> 5
main -> 6
main -> 7
main -> 8
main -> 9
main -> 10
```

The second correctly parallelizes on two threads, as shown in the following output:

```
parallel-1 -> 1
parallel-2 -> 2
parallel-1 -> 3
parallel-2 -> 4
parallel-1 -> 5
parallel-2 -> 6
parallel-1 -> 7
parallel-1 -> 9
parallel-2 -> 8
parallel-2 -> 10
```

If, once you process your sequence in parallel, you want to revert back to a "normal" `Flux` and apply the rest of the operator chain in a sequential manner, you can use the `sequential()` method on `ParallelFlux`.

Note that `sequential()` is implicitly applied if you `subscribe` to the ParallelFlux with a `Subscriber` but not when using the lambda-based variants of `subscribe`.

Note also that `subscribe(Subscriber<T>)` merges all the rails, while `subscribe(Consumer<T>)` runs all the rails. If the `subscribe()` method has a lambda, each lambda is executed as many times as there are rails.

You can also access individual rails or "groups" as a `Flux<GroupedFlux<T>>` through the `groups()` method and apply additional operators to them through the `composeGroup()` method.

8.6. Replacing Default `Schedulers`

As we have seen in the Threading and Schedulers section, Reactor Core comes with several `Scheduler` implementations. While you can always create new instances through the `new*` factory methods, each `Scheduler` flavor also has a default singleton instance that is accessible through the direct factory method (such as `Schedulers.elastic()` versus `Schedulers.newElastic()`).

These default instances are the ones used by operators that need a `Scheduler` to work when you do not explicitly specify one. For example, `Flux#delayElements(Duration)` uses the `Schedulers.parallel()` instance.

In some cases, however, you might need to change these default instances with something else in a cross-cutting way, without having to make sure every single operator you call has your specific `Scheduler` as a parameter. An example is measuring the time every single scheduled task takes by wrapping the real schedulers, for instrumentation purposes. In other words, you might want to **change the default `Schedulers`**.

Changing the default schedulers is possible through the `Schedulers.Factory` class. By default, a `Factory` creates all the standard `Scheduler` through similarly named methods. Each of these can be overridden with your custom implementation.

Additionally, the `Factory` exposes one additional customization method: `decorateExecutorService`. It is invoked during the creation of every reactor-core `Scheduler` that is backed by a `ScheduledExecutorService` (even non-default instances, such as those created by calls to `Schedulers.newParallel()`).

This lets you tune the `ScheduledExecutorService` to be used: The default one is exposed as a `Supplier` and, depending on the type of `Scheduler` being configured, you can choose to entirely bypass that supplier and return your own instance or you can `get()` the default instance and wrap it.

Once you create a `Factory` that fits your needs, you must install it via `Schedulers.setFactory(Factory)`.

Finally, there is a last customizable hook in `Schedulers`: `onHandleError`. This hook is invoked whenever a `Runnable` task submitted to a `Scheduler` throws an `Exception` (note that if there is an `UncaughtExceptionHandler` set for the `Thread` that ran the task, both the handler and the hook will be invoked).

8.7. Using Global Hooks

Reactor has another category of configurable callbacks that are invoked by Reactor operators in various situations. They are all set in the `Hooks` class, and fall into three categories:

- Dropping Hooks
- Internal Error Hook
- Assembly Hooks

8.7.1. Dropping Hooks

Dropping hooks are invoked when the source of an operator does not comply with the Reactive Streams specification. These kind of errors are outside of the normal execution path (that is, they cannot be propagated through `onError`).

Typically, a `Publisher` calls `onNext` on the operator despite having already called `onCompleted` on it previously. In that case, the `onNext` value is *dropped*. The same is true for an extraneous `onError` signal.

The corresponding hooks, `onNextDropped` and `onErrorDropped`, let you provide a global `Consumer` for these drops. For example, you can use it to log the drop and cleanup resources associated with a value if needed (as it never makes it to the rest of the reactive chain).

Setting the hooks twice in a row is additive: every consumer you provide is invoked. The hooks can be fully reset to their defaults by using `Hooks.resetOn*Dropped()` methods.

8.7.2. Internal Error Hook

One hook, `onOperatorError`, is invoked by operators when an unexpected `Exception` is thrown during the execution of their `onNext`, `onError` and `onComplete` methods.

Unlike the previous category, this is still within the normal execution path. A typical example is the `map` operator with a map function that throws an `Exception` (such as division by zero). It is still possible at this point to go through the usual channel of `onError`, and that is what the operator does.

First, it passes the `Exception` through `onOperatorError`. The hook lets you inspect the error (and the incriminating value, if relevant) and *change* the `Exception`. Of course, you can also do something on the side, such as log and return the original Exception.

Note that the `onOperatorError` hook can be set multiple times: you can provide a `String` identifier for a particular `BiFunction`, and subsequent calls with different keys concatenates the functions, which are all executed. On the other hand, reusing the same key twice lets you replace a function you previously set.

As a consequence, the default hook behavior can be both fully reset (using `Hooks.resetOnOperatorError()`) or partially reset for a specific `key` only (by using `Hooks.resetOnOperatorError(String)`).

8.7.3. Assembly Hooks

These hooks tie in the lifecycle of operators. They are invoked when a chain of operators is assembled (that is, instantiated). `onEachOperator` lets you dynamically change each operator as it is assembled in the chain, by returning a different `Publisher`. `onLastOperator` is similar, except that it is only invoked on the last operator in the chain before the `subscribe` call.

If you want to decorate all operators with a cross-cutting `Subscriber` implementation, you can look into `Operators#lift*` methods to help you deal with the various types of Reactor `Publishers` out there (Flux, Mono, ParallelFlux, GroupedFlux, ConnectableFlux), as well as their `Fuseable` version.

Like `onOperatorError`, these hooks are cumulative and can be identified with a key. They can also be reset partially or totally.

8.7.4. Hook Presets

The `Hooks` utility class provides a couple of preset hooks. These are alternatives to the default behaviors that you can use by calling their corresponding method, rather than coming up with the hook yourself:

- `onNextDroppedFail()`: `onNextDropped` used to throw a `Exceptions.failWithCancel()` exception. It now defaults to logging the dropped value at the DEBUG level. To go back to the old default behavior of throwing, use `onNextDroppedFail()`.
- `onOperatorDebug()`: This method activates debug mode. It ties into the `onOperatorError` hook, so calling `resetOnOperatorError()` also resets it. It can be independently reset via `resetOnOperatorDebug()` as it uses a specific key internally.

8.8. Adding a Context to a Reactive Sequence

One of the big technical challenges encountered when switching from an imperative programming perspective to a reactive programming mindset lies in how you deal with threading.

Contrary to what you might be used to, in reactive programming, a `Thread` can be used to process several asynchronous sequences that run roughly at the same time (actually, in non-blocking locksteps). The execution can also easily and often jump from one thread to another.

This arrangement is especially hard for developers that use features dependent on the threading model being more "stable", such as `ThreadLocal`. As it lets you associate data with a **thread**, it becomes tricky to use in a reactive context. As a result, libraries that rely on `ThreadLocal` at least introduce new challenges when used with Reactor. At worst, they work badly or even fail. Using the MDC of Logback to store and log correlation IDs is a prime example of such a situation.

The usual workaround for `ThreadLocal` usage is to move the contextual data, `C`, along your business data, `T`, in the sequence, by using `Tuple2<T, C>` for instance. This does not look good and leaks an orthogonal concern (the contextual data) into your method and `Flux` signatures.

Since version `3.1.0`, Reactor comes with an advanced feature that is somewhat comparable to `ThreadLocal` but applied to a `Flux` or a `Mono` instead of a `Thread`: the `Context`.

As an illustration of how it looks like, here is a very simple example of both writing to the `Context` and reading from it:

```
String key = "message";
Mono<String> r = Mono.just("Hello")
                .flatMap( s -> Mono.subscriberContext()
                                   .map( ctx -> s + " " + ctx.get(key)))
                .subscriberContext(ctx -> ctx.put(key, "World"));

StepVerifier.create(r)
            .expectNext("Hello World")
            .verifyComplete();
```

In the following sections, we'll learn about the `Context` and how to use it, so that you will eventually understand the example above.

> This is an advanced feature that is more targeted at library developers. It requires good understanding of the lifecycle of a `Subscription` and is intended for libraries that are responsible for the subscriptions.

8.8.1. The `Context` API

A `Context` is an interface reminiscent of `Map`: it stores key-value pairs and lets you fetch a value you stored by its key. More specifically:

- Both key and values are of type `Object`, so a `Context` can contain any number of highly divergent values from different libraries and sources.
- A `Context` is **immutable**.
- Use `put(Object key, Object value)` to store a key-value pair, returning a new `Context` instance. You can also merge two contexts into a new one by using `putAll(Context)`.
- You can check if the key is present with `hasKey(Object key)`.
- Use `getOrDefault(Object key, T defaultValue)` to retrieve a value (cast to a `T`) or fall back to a default one if the Context does not have that key.
- Use `getOrEmpty(Object key)` to get an `Optional<T>` (the context attempts to cast the stored value to `T`).
- Use `delete(Object key)` to remove the value associated to a key, returning a new `Context`.

> When **creating a** `Context`, you can create pre-valued contexts with up to five key-value pairs by using the static `Context.of` methods. They take 2, 4, 6, 8 or 10 `Object` instances, each couple of `Object` instances being a key-value pair to add to the `Context`.
>
> Alternatively you can also create an empty `Context` by using `Context.empty()`.

8.8.2. Tying the `Context` to a `Flux` and Writing

To make the context useful, it must be tied to a specific sequence and be accessible by each operator in a chain. Note that the operator must be a Reactor native operator, as `Context` is specific to

Reactor.

Actually, a `Context` is tied to each `Subscriber` to a chain. It uses the `Subscription` propagation mechanism to make itself available to each operator, starting with the final `subscribe` and moving up the chain.

In order to populate the `Context`, which can only be done at subscription time, you need to use the `subscriberContext` operator.

Use `subscriberContext(Context)`, which merges the `Context` you provide and the `Context` from downstream (remember, the `Context` is propagated from the bottom of the chain towards the top). This is done through a call to `putAll`, resulting in a new `Context` for upstream.

> You can also use the more advanced `subscriberContext(Function<Context, Context>)`. It receives the state of the `Context` from downstream and lets you put or delete values as you see fit, returning the new `Context` to use. You can even decide to return a completely different instance, although it is really not recommended (doing so might impact 3rd-party libraries that depend on the `Context`).

8.8.3. Reading the Context

Populating the `Context` is one aspect, but retrieving that data from it is equally important. Most of the time, the responsibility of putting information into the `Context` is on the end user's side, while exploiting that information is on the 3rd-party library's side, as such libraries are usually upstream of the client code.

The tool for reading data from the context is the static `Mono.subscriberContext()` method.

8.8.4. Simple Examples

The examples in this section are meant as ways to better understand some of the caveats of using a `Context`.

Let's first look back at our simple example from the introduction in a bit more details:

```
String key = "message";
Mono<String> r = Mono.just("Hello")
                .flatMap( s -> Mono.subscriberContext() ②
                                   .map( ctx -> s + " " + ctx.get(key))) ③
                .subscriberContext(ctx -> ctx.put(key, "World")); ①

StepVerifier.create(r)
            .expectNext("Hello World") ④
            .verifyComplete();
```

① The chain of operators ends with a call to `subscriberContext(Function)` that puts `"World"` into the `Context` under the key `"message"`.

② We `flatMap` on the source element, materializing the `Context` with `Mono.subscriberContext()`.

③ We then use `map` to extract the data associated to `"message"` and concatenate that with the original word.

④ The resulting `Mono<String>` indeed emits `"Hello World"`.

> The numbering above vs the actual line order is not a mistake: it represents the execution order. Even though `subscriberContext` is the last piece of the chain, it is the one that gets executed first (due to its subscription time nature, and the fact that the subscription signal flows from bottom to top).

Note that in your chain of operators, the **relative positions** of where you **write** to the `Context` and where you **read** from it matters: the `Context` is immutable and its content can only be seen by operators above it, as demonstrated in the following code example:

```
String key = "message";
Mono<String> r = Mono.just("Hello")
                     .subscriberContext(ctx -> ctx.put(key, "World")) ①
                     .flatMap( s -> Mono.subscriberContext()
                                       .map( ctx -> s + " " + ctx.getOrDefault(key,
"Stranger")));  ②

StepVerifier.create(r)
            .expectNext("Hello Stranger") ③
            .verifyComplete();
```

① The `Context` is written to too high in the chain…

② As a result, in the `flatMap`, there's no value associated to our key. A default value is used instead.

③ The resulting `Mono<String>` thus emits `"Hello Stranger"`.

The following example also demonstrates the immutable nature of the `Context`, and how `Mono.subscriberContext()` always returns the `Context` set by `subscriberContext` calls:

```
String key = "message";

Mono<String> r = Mono.subscriberContext() ①
    .map( ctx -> ctx.put(key, "Hello")) ②
    .flatMap( ctx -> Mono.subscriberContext()) ③
    .map( ctx -> ctx.getOrDefault(key,"Default")); ④

StepVerifier.create(r)
    .expectNext("Default") ⑤
    .verifyComplete();
```

① We materialize the `Context`

② In a `map` we attempt to mutate it

③ We re-materialize the `Context` in a `flatMap`

④ We read the attempted key in the `Context`

⑤ The key was never set to "Hello".

Similarly, in case of several attempts to write the same key to the Context, the **relative order of the writes** matters too: operators reading the Context will see the value that was set closest to under them, as demonstrated in the following example:

```
String key = "message";
Mono<String> r = Mono.just("Hello")
                .flatMap( s -> Mono.subscriberContext()
                                   .map( ctx -> s + " " + ctx.get(key)))
                .subscriberContext(ctx -> ctx.put(key, "Reactor")) ①
                .subscriberContext(ctx -> ctx.put(key, "World")); ②

StepVerifier.create(r)
            .expectNext("Hello Reactor") ③
            .verifyComplete();
```

① A write attempt on key "message".

② Another write attempt on key "message".

③ The map only saw the value set closest to it (and below it): "Reactor".

Here what happens is that the Context is populated during subscription with "World". Then the subscription signal moves upstream, and another write happens. This produces a second immutable Context with a value of "Reactor". After that, data starts flowing. The flatMap sees the Context closest to it, which is our second Context with the "Reactor" value.

You might wonder if the Context is propagated along with the data signal. If that was the case, putting another flatMap between these two writes would use the value from the top Context. But this is not the case, as demonstrated by the following example:

```
String key = "message";
Mono<String> r = Mono.just("Hello")
                     .flatMap( s -> Mono.subscriberContext()
                                        .map( ctx -> s + " " + ctx.get(key))) ③
                     .subscriberContext(ctx -> ctx.put(key, "Reactor")) ②
                     .flatMap( s -> Mono.subscriberContext()
                                        .map( ctx -> s + " " + ctx.get(key))) ④
                     .subscriberContext(ctx -> ctx.put(key, "World")); ①

StepVerifier.create(r)
            .expectNext("Hello Reactor World") ⑤
            .verifyComplete();
```

① This is the first write to happen.

② This is the second write to happen.

③ First flatMap sees second write.

④ Second flatMap concatenates result from first one with the value from **first write**.

⑤ The Mono emits "Hello Reactor World".

The reason is that the Context is associated to the Subscriber and each operator accesses the Context by requesting it from its downstream Subscriber.

One last interesting propagation case is the one where the Context is also written to **inside** a flatMap, as in the following example:

```
String key = "message";
Mono<String> r =
        Mono.just("Hello")
            .flatMap( s -> Mono.subscriberContext()
                               .map( ctx -> s + " " + ctx.get(key))
            )
            .flatMap( s -> Mono.subscriberContext()
                               .map( ctx -> s + " " + ctx.get(key))
                               .subscriberContext(ctx -> ctx.put(key, "Reactor")) ①
            )
            .subscriberContext(ctx -> ctx.put(key, "World")); ②

StepVerifier.create(r)
            .expectNext("Hello World Reactor")
            .verifyComplete();
```

① This subscriberContext does not impact anything outside of its flatMap

② This subscriberContext impacts the main sequence's Context

In the example above, the final emitted value is "Hello World Reactor" and not "Hello Reactor World", because the subscriberContext that writes "Reactor" does so as part of the inner sequence of the second flatMap. As a consequence, it is not visible / propagated through the main sequence and the first flatMap doesn't see it. Propagation + immutability isolate the Context in operators that create intermediate inner sequences like flatMap.

8.8.5. Full Example

Let's consider a more real life example of a library reading information from the Context: A reactive HTTP client that takes a Mono<String> as the source of data for a PUT but also looks for a particular Context key to add a correlation ID to the request's headers.

From the user perspective, it is called as follows:

```
doPut("www.example.com", Mono.just("Walter"))
```

In order to propagate a correlation ID, it would be called as follows:

```
doPut("www.example.com", Mono.just("Walter"))
    .subscriberContext(Context.of(HTTP_CORRELATION_ID, "2-j3r9afaf92j-afkaf"))
```

As you can see in the snippets above, the user code uses `subscriberContext` to populate a `Context` with an `HTTP_CORRELATION_ID` key-value pair. The upstream of the operator is a `Mono<Tuple2<Integer, String>>` (a simplistic representation of an HTTP response) returned by the HTTP client library. So it is effectively passing information from the user code to the library code.

The following example shows mock code from the library's perspective that reads the context and "augments the request" if it can find the correlation ID:

```
static final String HTTP_CORRELATION_ID = "reactive.http.library.correlationId";

Mono<Tuple2<Integer, String>> doPut(String url, Mono<String> data) {
    Mono<Tuple2<String, Optional<Object>>> dataAndContext =
            data.zipWith(Mono.subscriberContext() ①
                             .map(c -> c.getOrEmpty(HTTP_CORRELATION_ID))); ②

    return dataAndContext
            .<String>handle((dac, sink) -> {
                if (dac.getT2().isPresent()) { ③
                    sink.next("PUT <" + dac.getT1() + "> sent to " + url + " with
header X-Correlation-ID = " + dac.getT2().get());
                }
                else {
                    sink.next("PUT <" + dac.getT1() + "> sent to " + url);
                }
                sink.complete();
            })
            .map(msg -> Tuples.of(200, msg));
}
```

① Materialize the `Context` through `Mono.subscriberContext()`.

② Extract a value for a the correlation ID key, as an `Optional`.

③ If the key was present in the context, use the correlation ID as a header.

In the library snippet, you can see how it zips the data `Mono` with `Mono.subscriberContext()`. This gives the library a `Tuple2<String, Context>`, and that context contains the `HTTP_CORRELATION_ID` entry from downstream (as it is on the direct path to the subscriber).

The library code then uses `map` to extract an `Optional<String>` for that key, and, if the entry is present, it uses the passed correlation ID as a `X-Correlation-ID` header. That last part is simulated by the `handle` above.

The whole test that validates the library code used the correlation ID can be written as follows:

```
@Test
public void contextForLibraryReactivePut() {
    Mono<String> put = doPut("www.example.com", Mono.just("Walter"))
            .subscriberContext(Context.of(HTTP_CORRELATION_ID, "2-j3r9afaf92j-afkaf"))
            .filter(t -> t.getT1() < 300)
            .map(Tuple2::getT2);

    StepVerifier.create(put)
                .expectNext("PUT <Walter> sent to www.example.com with header X-
Correlation-ID = 2-j3r9afaf92j-afkaf")
                .verifyComplete();
}
```

8.9. Dealing with Objects that need cleanup

In very specific cases, your application may deal with types that necessitate some form of cleanup once they're not in use anymore. This is an advanced scenario, for example when you have *reference counted* objects or when you're dealing with *off heap* objects. Netty's `ByteBuf` is a prime example of both.

In order to ensure proper cleanup of such objects, you need to accommodate for it on a `Flux`-by-`Flux` basis, as well as in several of the global hooks (see Using Global Hooks):

- The `doOnDiscard` `Flux`/`Mono` operator
- The `onOperatorError` hook
- The `onNextDropped` hook
- Operator-specific handlers

This is needed because each hook is made with a specific subset of cleanup in mind, and users might want ie. to implement specific error handling logic in addition to cleanup logic within `onOperatorError`.

Note that some operators are less adapted to dealing with objects that need cleanup. For example, `bufferWhen` can introduce overlapping buffers, and that means that the discard "local hook" above might see a first buffer as being discarded and cleanup an element in it that is in a second buffer *which is still valid.*

❗ For the purpose of cleaning up, **all these hooks MUST be IDEMPOTENT**. They might on some occasions get applied several times to the same object. Unlike the `doOnDiscard` operator, which performs a class `instanceOf` check, the global hooks are also dealing with instances that can be any `Object` and it is up to the user's implementation to distinguish between which instances need cleanup and which don't.

8.9.1. The `doOnDiscard` operator / local hook

This hook has been specifically put in place for cleanup of objects that would otherwise never be exposed to user code. It is intended as a cleanup hook for flows that operate under normal circumstances (ie. not malformed sources that push too many items, which is covered by `onNextDropped`).

It is local, in the sense that it is activated through an operator and only applies to a given `Flux` or `Mono`.

Obvious cases include operators that filter elements from upstream. These elements never reach the next operator (or final subscriber), but this is part of the normal path of execution. As such, they are passed to the `doOnDiscard` hook. For example:

- `filter`: items that don't match the filter are considered "discarded"
- `skip`: items skipped are discarded
- `buffer(maxSize, skip)` with `maxSize < skip`: "dropping buffer", items in between buffers are discarded
- …

But `doOnDiscard` is not limited to filtering operators, and is also used by operators that internally queue data for backpressure purposes. More specifically, most of the time this is important during cancellation: an operator that prefetches data from its source and later drains to its subscriber upon demand could have un-emitted data when it gets cancelled. Such operators use the `doOnDiscard` hook during cancellation to clear up their internal backpressure `Queue`.

> ⚠ Each call to `doOnDiscard(Class, Consumer)` is additive with the others, to the extent that it is only visible and used by operators upstream of it.

8.9.2. The `onOperatorError` hook

The `onOperatorError` hook is intended to modify errors in a transverse manner (similar to an AOP catch-and-rethrow).

When the error happens during the processing of an `onNext` signal, the element that was being emitted is passed to `onOperatorError`.

If that type of element needs cleanup you need to implement it in the `onOperatorError` hook, possibly on top of error-rewriting code.

8.9.3. The `onNextDropped` hook

With malformed `Publishers`, there could be cases where an operator receives an element when it expected none (typically, after having received the `onError` or `onComplete` signals). In such cases, the unexpected element is "dropped", passed to the `onNextDropped` hook. If you have types that need cleanup, you must detect these in the `onNextDropped` hook and implement cleanup code there as well.

8.9.4. Operator-specific handlers

Some operators that deal with buffers and/or collect values as part of their operations have specific handlers for cases where collected data isn't propagated downstream. If you use such operators with the type(s) that need cleanup, you need to perform cleanup in these handlers.

For example, `distinct` has such a callback that is invoked when the operator terminates (or is cancelled) in order to clear the collection it uses to judge if an element is distinct or not. By default, the collection is a `HashSet` and the cleanup callback is simply a `Hashet::clear`. But if you deal with reference counted objects, you might want to change that to a more involved handler that would `release` each element in the set before `clear()`ing it.

8.10. Null-safety

Although Java does not allow expressing null-safety with its type system, Reactor now provides annotations to declare nullability of APIs, similar to those provided by Spring Framework 5.

Reactor leverages these annotations, but they can also be used in any Reactor-based Java project to declare null-safe APIs. Nullability of types used inside method bodies is outside of the scope of this feature.

These annotations are meta-annotated with JSR 305 annotations (a dormant JSR that is supported by tools like IntelliJ IDEA) to provide useful warnings to Java developers related to null-safety in order to avoid `NullPointerException` at runtime. JSR 305 meta-annotations allows tooling vendors to provide null-safety support in a generic way, without having to hard-code support for Reactor annotations.

It is not necessary nor recommended with Kotlin 1.1.5+ to have a dependency on JSR 305 in your project classpath.

They are also used by Kotlin which natively supports null-safety. See this dedicated section for more details.

The following annotations are provided in the `reactor.util.annotation` package:

- `@NonNull` indicates that a specific parameter, return value, or field cannot be `null`. (It is not needed on parameters and return value where `@NonNullApi` applies) .
- `@Nullable` indicates that a parameter, return value, or field can be `null`.
- `@NonNullApi` is a package level annotation that indicates non-null is the default behavior for parameters and return values.

Nullability for generic type arguments, varargs, and array elements is not supported yet. See issue #878 for up-to-date information.

Appendix A: Which operator do I need?

In this section, if an operator is specific to `Flux` or `Mono` it is prefixed accordingly. Common operators have no prefix. When a specific use case is covered by a combination of operators, it is presented as a method call, with leading dot and parameters in parentheses, like this: `.methodCall(parameter)`.

I want to deal with:

- Creating a New Sequence…
- Transforming an Existing Sequence
- Filtering a Sequence
- Peeking into a Sequence
- Handling Errors
- Working with Time
- Splitting a `Flux`
- Going Back to the Synchronous World
- Multicasting a `Flux` to several `Subscribers`

A.1. Creating a New Sequence…

- that emits a `T`, and I already have: `just`
  - …from an `Optional<T>`: `Mono#justOrEmpty(Optional<T>)`
  - …from a potentially `null` T: `Mono#justOrEmpty(T)`
- that emits a `T` returned by a method: `just` as well
  - …but lazily captured: use `Mono#fromSupplier` or wrap `just` inside `defer`
- that emits several `T` I can explicitly enumerate: `Flux#just(T…)`
- that iterates over:
  - an array: `Flux#fromArray`
  - a collection or iterable: `Flux#fromIterable`
  - a range of integers: `Flux#range`
  - a `Stream` supplied for each Subscription: `Flux#fromStream(Supplier<Stream>)`
- that emits from various single-valued sources such as:
  - a `Supplier<T>`: `Mono#fromSupplier`
  - a task: `Mono#fromCallable`, `Mono#fromRunnable`
  - a `CompletableFuture<T>`: `Mono#fromFuture`
- that completes: `empty`
- that errors immediately: `error`

    - …but lazily build the `Throwable`: `error(Supplier<Throwable>)`
- that never does anything: `never`
- that is decided at subscription: `defer`
- that depends on a disposable resource: `using`
- that generates events programmatically (can use state):
    - synchronously and one-by-one: `Flux#generate`
    - asynchronously (can also be sync), multiple emissions possible in one pass: `Flux#create` (`Mono#create` as well, without the multiple emission aspect)

A.2. Transforming an Existing Sequence

- I want to transform existing data:
    - on a 1-to-1 basis (eg. strings to their length): `map`
        - …by just casting it: `cast`
        - …in order to materialize each source value's index: `Flux#index`
    - on a 1-to-n basis (eg. strings to their characters): `flatMap` + use a factory method
    - on a 1-to-n basis with programmatic behavior for each source element and/or state: `handle`
    - running an asynchronous task for each source item (eg. urls to http request): `flatMap` + an async `Publisher`-returning method
        - …ignoring some data: conditionally return a `Mono.empty()` in the flatMap lambda
        - …retaining the original sequence order: `Flux#flatMapSequential` (this triggers the async processes immediately but reorders the results)
        - …where the async task can return multiple values, from a `Mono` source: `Mono#flatMapMany`
- I want to add pre-set elements to an existing sequence:
    - at the start: `Flux#startWith(T…)`
    - at the end: `Flux#concatWith(T…)`
- I want to aggregate a `Flux`: (the `Flux#` prefix is assumed below)
    - into a List: `collectList`, `collectSortedList`
    - into a Map: `collectMap`, `collectMultiMap`
    - into an arbitrary container: `collect`
    - into the size of the sequence: `count`
    - by applying a function between each element (eg. running sum): `reduce`
        - …but emitting each intermediary value: `scan`
    - into a boolean value from a predicate:
        - applied to all values (AND): `all`
        - applied to at least one value (OR): `any`

        - testing the presence of any value: `hasElements`
        - testing the presence of a specific value: `hasElement`
- I want to combine publishers…
    - in sequential order: `Flux#concat` or `.concatWith(other)`
        - …but delaying any error until remaining publishers have been emitted: `Flux#concatDelayError`
        - …but eagerly subscribing to subsequent publishers: `Flux#mergeSequential`
    - in emission order (combined items emitted as they come): `Flux#merge` / `.mergeWith(other)`
        - …with different types (transforming merge): `Flux#zip` / `Flux#zipWith`
    - by pairing values:
        - from 2 Monos into a `Tuple2`: `Mono#zipWith`
        - from n Monos when they all completed: `Mono#zip`
    - by coordinating their termination:
        - from 1 Mono and any source into a `Mono<Void>`: `Mono#and`
        - from n sources when they all completed: `Mono#when`
        - into an arbitrary container type:
            - each time all sides have emitted: `Flux#zip` (up to the smallest cardinality)
            - each time a new value arrives at either side: `Flux#combineLatest`
    - only considering the sequence that emits first: `Flux#first`, `Mono#first`, `mono.or(otherMono).or(thirdMono)`, `flux.or(otherFlux).or(thirdFlux)`
    - triggered by the elements in a source sequence: `switchMap` (each source element is mapped to a Publisher)
    - triggered by the start of the next publisher in a sequence of publishers: `switchOnNext`
- I want to repeat an existing sequence: `repeat`
    - …but at time intervals: `Flux.interval(duration).flatMap(tick → myExistingPublisher)`
- I have an empty sequence but…
    - I want a value instead: `defaultIfEmpty`
    - I want another sequence instead: `switchIfEmpty`
- I have a sequence but I am not interested in values: `ignoreElements`
    - …and I want the completion represented as a `Mono`: `then`
    - …and I want to wait for another task to complete at the end: `thenEmpty`
    - …and I want to switch to another `Mono` at the end: `Mono#then(mono)`
    - …and I want to emit a single value at the end: `Mono#thenReturn(T)`
    - …and I want to switch to a `Flux` at the end: `thenMany`
- I have a Mono for which I want to defer completion…
    - …until another publisher, which is derived from this value, has completed:

    `Mono#delayUntil(Function)`

- I want to expand elements recursively into a graph of sequences and emit the combination…
  - …expanding the graph breadth first: `expand(Function)`
  - …expanding the graph depth first: `expandDeep(Function)`

A.3. Peeking into a Sequence

- Without modifying the final sequence, I want to:
  - get notified of / execute additional behavior [1: sometimes referred to as "side-effects"] on:
    - emissions: `doOnNext`
    - completion: `Flux#doOnComplete`, `Mono#doOnSuccess` (includes the result if any)
    - error termination: `doOnError`
    - cancellation: `doOnCancel`
    - "start" of the sequence: `doFirst`
      - this is tied to `Publisher#subscribe(Subscriber)`
    - subscription (as in `Subscription` acknowledgment after `subscribe`): `doOnSubscribe` **(tied to `Subscriber#onSubscribe(Subscription)`)
    - request: `doOnRequest`
    - completion or error: `doOnTerminate` (Mono version includes the result if any)
      - but **after** it has been propagated downstream: `doAfterTerminate`
    - any type of signal, represented as a `Signal`: `Flux#doOnEach`
    - any terminating condition (complete, error, cancel): `doFinally`
  - log what happens internally: `log`
- I want to know of all events:
  - each represented as `Signal` object:
    - in a callback outside the sequence: `doOnEach`
    - instead of the original onNext emissions: `materialize`
      - …and get back to the onNexts: `dematerialize`
  - as a line in a log: `log`

A.4. Filtering a Sequence

- I want to filter a sequence:
  - based on an arbitrary criteria: `filter`
    - …that is asynchronously computed: `filterWhen`
  - restricting on the type of the emitted objects: `ofType`
  - by ignoring the values altogether: `ignoreElements`

        - by ignoring duplicates:
            - in the whole sequence (logical set): `Flux#distinct`
            - between subsequently emitted items (deduplication): `Flux#distinctUntilChanged`
- I want to keep only a subset of the sequence:
    - by taking N elements:
        - at the beginning of the sequence: `Flux#take(long)`
            - …based on a duration: `Flux#take(Duration)`
            - …only the first element, as a `Mono`: `Flux#next()`
            - …using `request(N)` rather than cancellation: `Flux#limitRequest(long)`
        - at the end of the sequence: `Flux#takeLast`
        - until a criteria is met (inclusive): `Flux#takeUntil` (predicate-based), `Flux#takeUntilOther` (companion publisher-based)
        - while a criteria is met (exclusive): `Flux#takeWhile`
    - by taking at most 1 element:
        - at a specific position: `Flux#elementAt`
        - at the end: `.takeLast(1)`
            - …and emit an error if empty: `Flux#last()`
            - …and emit a default value if empty: `Flux#last(T)`
    - by skipping elements:
        - at the beginning of the sequence: `Flux#skip(long)`
            - …based on a duration: `Flux#skip(Duration)`
        - at the end of the sequence: `Flux#skipLast`
        - until a criteria is met (inclusive): `Flux#skipUntil` (predicate-based), `Flux#skipUntilOther` (companion publisher-based)
        - while a criteria is met (exclusive): `Flux#skipWhile`
    - by sampling items:
        - by duration: `Flux#sample(Duration)`
            - but keeping the first element in the sampling window instead of the last: `sampleFirst`
        - by a publisher-based window: `Flux#sample(Publisher)`
        - based on a publisher "timing out": `Flux#sampleTimeout` (each element triggers a publisher, and is emitted if that publisher does not overlap with the next)
- I expect at most 1 element (error if more than one)…
    - and I want an error if the sequence is empty: `Flux#single()`
    - and I want a default value if the sequence is empty: `Flux#single(T)`
    - and I accept an empty sequence as well: `Flux#singleOrEmpty`

A.5. Handling Errors

- I want to create an erroring sequence: `error`…
  - …to replace the completion of a successful `Flux`: `.concat(Flux.error(e))`
  - …to replace the **emission** of a successful `Mono`: `.then(Mono.error(e))`
  - …if too much time elapses between onNexts: `timeout`
  - …lazily: `error(Supplier<Throwable>)`
- I want the try/catch equivalent of:
  - throwing: `error`
  - catching an exception:
    - and falling back to a default value: `onErrorReturn`
    - and falling back to another `Flux` or `Mono`: `onErrorResume`
    - and wrapping and re-throwing: `.onErrorMap(t → new RuntimeException(t))`
  - the finally block: `doFinally`
  - the using pattern from Java 7: `using` factory method
- I want to recover from errors…
  - by falling back:
    - to a value: `onErrorReturn`
    - to a `Publisher` or `Mono`, possibly different ones depending on the error: `Flux#onErrorResume` and `Mono#onErrorResume`
  - by retrying: `retry`
    - …triggered by a companion control Flux: `retryWhen`
    - … using a standard backoff strategy (exponential backoff with jitter): `retryBackoff`
- I want to deal with backpressure "errors" [2: request max from upstream and apply the strategy when downstream does not produce enough request]…
  - by throwing a special `IllegalStateException`: `Flux#onBackpressureError`
  - by dropping excess values: `Flux#onBackpressureDrop`
    - …except the last one seen: `Flux#onBackpressureLatest`
  - by buffering excess values (bounded or unbounded): `Flux#onBackpressureBuffer`
    - …and applying a strategy when bounded buffer also overflows: `Flux#onBackpressureBuffer` with a `BufferOverflowStrategy`

A.6. Working with Time

- I want to associate emissions with a timing (`Tuple2<Long, T>`) measured…
  - since subscription: `elapsed`
  - since the dawn of time (well, computer time): `timestamp`

- I want my sequence to be interrupted if there is too much delay between emissions: `timeout`
- I want to get ticks from a clock, regular time intervals: `Flux#interval`
- I want to emit a single `0` after an initial delay: static `Mono.delay`.
- I want to introduce a delay:
  - between each onNext signal: `Mono#delayElement`, `Flux#delayElements`
  - before the subscription happens: `delaySubscription`

A.7. Splitting a `Flux`

- I want to split a `Flux<T>` into a `Flux<Flux<T>>`, by a boundary criteria:
  - of size: `window(int)`
    - ...with overlapping or dropping windows: `window(int, int)`
  - of time `window(Duration)`
    - ...with overlapping or dropping windows: `window(Duration, Duration)`
  - of size OR time (window closes when count is reached or timeout elapsed): `windowTimeout(int, Duration)`
  - based on a predicate on elements: `windowUntil`
    - ......emitting the element that triggered the boundary in the next window (`cutBefore` variant): `.windowUntil(predicate, true)`
    - ...keeping the window open while elements match a predicate: `windowWhile` (non-matching elements are not emitted)
  - driven by an arbitrary boundary represented by onNexts in a control Publisher: `window(Publisher)`, `windowWhen`
- I want to split a `Flux<T>` and buffer elements within boundaries together...
  - into `List`:
    - by a size boundary: `buffer(int)`
      - ...with overlapping or dropping buffers: `buffer(int, int)`
    - by a duration boundary: `buffer(Duration)`
      - ...with overlapping or dropping buffers: `buffer(Duration, Duration)`
    - by a size OR duration boundary: `bufferTimeout(int, Duration)`
    - by an arbitrary criteria boundary: `bufferUntil(Predicate)`
      - ...putting the element that triggered the boundary in the next buffer: `.bufferUntil(predicate, true)`
      - ...buffering while predicate matches and dropping the element that triggered the boundary: `bufferWhile(Predicate)`
    - driven by an arbitrary boundary represented by onNexts in a control Publisher: `buffer(Publisher)`, `bufferWhen`
  - into an arbitrary "collection" type `C`: use variants like `buffer(int, Supplier<C>)`

- I want to split a `Flux<T>` so that element that share a characteristic end up in the same sub-flux: `groupBy(Function<T,K>)` TIP: Note that this returns a `Flux<GroupedFlux<K, T>>`, each inner `GroupedFlux` shares the same `K` key accessible through `key()`.

A.8. Going Back to the Synchronous World

Note: all of these methods except `Mono#toFuture` will throw an `UnsupportedOperatorException` if called from within a `Scheduler` marked as "non-blocking only" (by default `parallel()` and `single()`).

- I have a `Flux<T>` and I want to:
  - block until I can get the first element: `Flux#blockFirst`
    - ...with a timeout: `Flux#blockFirst(Duration)`
  - block until I can get the last element (or null if empty): `Flux#blockLast`
    - ...with a timeout: `Flux#blockLast(Duration)`
  - synchronously switch to an `Iterable<T>`: `Flux#toIterable`
  - synchronously switch to a Java 8 `Stream<T>`: `Flux#toStream`
- I have a `Mono<T>` and I want:
  - to block until I can get the value: `Mono#block`
    - ...with a timeout: `Mono#block(Duration)`
  - a `CompletableFuture<T>`: `Mono#toFuture`

A.9. Multicasting a `Flux` to several `Subscribers`

- I want to connect multiple `Subscriber` to a `Flux`:
  - and decide when to trigger the source with `connect()`: `publish()` (returns a `ConnectableFlux`)
  - and trigger the source immediately (late subscribers see later data): `share()`
  - and permanently connect the source when enough subscribers have registered: `.publish().autoConnect(n)`
  - and automatically connect and cancel the source when subscribers go above/below the threshold: `.publish().refCount(n)`
    - ...but giving a chance for new subscribers to come in before cancelling: `.publish().refCountGrace(n, Duration)`
- I want to cache data from a `Publisher` and replay it to later subscribers:
  - up to `n` elements: `cache(int)`
  - caching latest elements seen within a `Duration` (Time-To-Live): `cache(Duration)`
    - ...but retain no more than `n` elements: `cache(int, Duration)`
  - but without immediately triggering the source: `Flux#replay` (returns a `ConnectableFlux`)

Appendix B: FAQ, Best Practices, and "How do I…?"

B.1. How Do I Wrap a Synchronous, Blocking Call?

It's often the case that a source of information is synchronous and blocking. To deal with such sources in your Reactor applications, apply the following pattern:

```
Mono blockingWrapper = Mono.fromCallable(() -> { ①
    return /* make a remote synchronous call */ ②
});
blockingWrapper = blockingWrapper.subscribeOn(Schedulers.elastic()); ③
```

① Create a new `Mono` by using `fromCallable`.

② Return the asynchronous, blocking resource.

③ Ensure each subscription will happen on a dedicated single-threaded worker from `Schedulers.elastic()`.

You should use a Mono because the source returns one value. You should use `Schedulers.elastic` because it creates a dedicated thread to wait for the blocking resource without tying up some other resource.

Note that `subscribeOn` does not subscribe to the `Mono`. It specifies what kind of `Scheduler` to use when a subscribe call happens.

B.2. I Used an Operator on my `Flux` but it Doesn't Seem to Apply. What Gives?

Make sure that the variable you `.subscribe()` to has been affected by the operators you think should have been applied to it.

Reactor operators are decorators. They return a different instance that wraps the source sequence and add behavior. That is why the preferred way of using operators is to **chain** the calls.

Compare the following two examples:

without chaining (incorrect)

```
Flux<String> flux = Flux.just("foo", "chain");
flux.map(secret -> secret.replaceAll(".", "*")); ①
flux.subscribe(next -> System.out.println("Received: " + next));
```

① The mistake is here. The result isn't attached to the `flux` variable.

without chaining (correct)

```
Flux<String> flux = Flux.just("foo", "chain");
flux = flux.map(secret -> secret.replaceAll(".", "*"));
flux.subscribe(next -> System.out.println("Received: " + next));
```

This sample is even better (because it's simpler):

with chaining (best)

```
Flux<String> secrets = Flux
  .just("foo", "chain")
  .map(secret -> secret.replaceAll(".", "*"))
  .subscribe(next -> System.out.println("Received: " + next));
```

The first version will output:

```
Received: foo
Received: chain
```

Whereas the two other versions will output the expected:

```
Received: ***
Received: *****
```

B.3. My `Mono` `zipWith`/`zipWhen` Is Never Called

example

```
myMethod.process("a") // this method returns Mono<Void>
        .zipWith(myMethod.process("b"), combinator) //this is never called
        .subscribe();
```

If the source `Mono` is either `empty` or a `Mono<Void>` (a `Mono<Void>` is empty for all intents and purposes), some combinations will never be called.

This is the typical case for any transformer like the `zip` static method or `zipWith`/`zipWhen` operators, which by definition need an element from each source to produce their output.

Using data-suppressing operators on sources of `zip` is thus problematic. Examples of data-suppressing operators include `then()`, `thenEmpty(Publisher<Void>)`, `ignoreElements()` and `ignoreElement()`, `when(Publisher…)`.

Similarly, operators that use a `Function<T,?>` to tune their behavior, like `flatMap`, do need at least one element to be emitted for the `Function` to have a chance to apply. Applying these on an empty (or `<Void>`) sequence will never produce an element.

You can use `.defaultIfEmpty(T)` and `.switchIfEmpty(Publisher<T>)` to respectively replace an **empty** sequence of `T` with a default value or a fallback `Publisher<T>`, which could help avoid some of these situations. Note that this doesn't apply to `Flux<Void>`/`Mono<Void>` sources, as you can only switch to another `Publisher<Void>`, which is still guaranteed to be empty. Here's an example of `defaultIfEmpty`:

use `defaultIfEmpty` before `zipWhen`

```
myMethod.emptySequenceForKey("a") // this method returns empty Mono<String>
        .defaultIfEmpty("") // this converts empty sequence to just the empty
String
        .zipWhen(aString -> myMethod.process("b")) //this is called with the empty
String
        .subscribe();
```

B.4. How to Use `retryWhen` to Emulate `retry(3)`?

The `retryWhen` operator can be quite complex. Hopefully this snippet of code can help you understand how it works by attempting to emulate a simpler `retry(3)`:

```
Flux<String> flux =
Flux.<String>error(new IllegalArgumentException())
    .retryWhen(companion -> companion
    .zipWith(Flux.range(1, 4), ①
          (error, index) -> { ②
            if (index < 4) return index; ③
            else throw Exceptions.propagate(error); ④
          })
    );
```

① Trick one: use `zip` and a `range` of "number of acceptable retries + 1".

② The `zip` function lets you count the retries while keeping track of the original error.

③ To allow for three retries, indexes before 4 return a value to emit.

④ In order to terminate the sequence in error, we throw the original exception after these three retries.

B.5. How to Use `retryWhen` for Exponential Backoff?

Exponential backoff produces retry attempts with a growing delay between each of the attempts, so as not to overload the source systems and risk an all out crash. The rationale is that if the source produces an error, it is already in an unstable state and not likely to immediately recover from it. So blindly retrying immediately is likely to produce yet another error and add to the instability.

Since `3.2.0.RELEASE`, Reactor comes with such a retry baked in: `Flux.retryBackoff`.

For the curious, here is how to implement an exponential backoff with `retryWhen` that delays retries and increase the delay between each attempt (pseudocode: delay = attempt number * 100 milliseconds):

```
Flux<String> flux =
Flux.<String>error(new IllegalArgumentException())
    .retryWhen(companion -> companion
        .doOnNext(s -> System.out.println(s + " at " + LocalTime.now())) ①
        .zipWith(Flux.range(1, 4), (error, index) -> { ②
          if (index < 4) return index;
          else throw Exceptions.propagate(error);
        })
        .flatMap(index -> Mono.delay(Duration.ofMillis(index * 100))) ③
        .doOnNext(s -> System.out.println("retried at " + LocalTime.now())) ④
    );
```

① We log the time of errors.

② We use the `retryWhen` + `zipWith` trick to propagate the error after 3 retries.

③ Through `flatMap`, we cause a delay that depends on the attempt's index.

④ We also log the time at which the retry happens.

When subscribed to, this fails and terminates after printing out:

```
java.lang.IllegalArgumentException at 18:02:29.338
retried at 18:02:29.459 ①
java.lang.IllegalArgumentException at 18:02:29.460
retried at 18:02:29.663 ②
java.lang.IllegalArgumentException at 18:02:29.663
retried at 18:02:29.964 ③
java.lang.IllegalArgumentException at 18:02:29.964
```

① first retry after about 100ms

② second retry after about 200ms

③ third retry after about 300ms

B.6. How Do I Ensure Thread Affinity Using `publishOn()`?

As described in Schedulers, `publishOn()` can be used to switch execution contexts. The `publishOn` operator influences the threading context where the rest of the operators in the chain below it will execute, up to a new occurrence of `publishOn`. So the placement of `publishOn` is significant.

For instance, in the example below, the `transform` function in `map()` is executed on a worker of `scheduler1` and the `processNext` method in `doOnNext()` is executed on a worker of `scheduler2`. Each **subscription** gets its own worker, so all elements pushed to the corresponding subscriber are published on the same `Thread`.

Single threaded schedulers may be used to ensure thread affinity for different stages in the chain or for different subscribers.

```
EmitterProcessor<Integer> processor = EmitterProcessor.create();
processor.publishOn(scheduler1)
         .map(i -> transform(i))
         .publishOn(scheduler2)
         .doOnNext(i -> processNext(i))
         .subscribe();
```

B.7. What Is a Good Pattern for Contextual Logging? (MDC)

Most logging frameworks allow contextual logging, letting users store variables that are reflected in the logging pattern, generally by way of a `Map` called the MDC ("Mapped Diagnostic Context"). This is one of the most recurring use of `ThreadLocal` in Java, and as a consequence this pattern assumes that the code being logged is tied in a one-to-one relationship with a `Thread`.

That might have been a safe assumption to make before Java 8, but with the advent of functional programming elements in the Java language things have changed a bit…

Let’s take the example of a API that was imperative and used the template method pattern, then switches to a more functional style. With the template method pattern, inheritance was at play. Now in its more functional approach, higher order functions are passed to define the "steps" of the algorithm. Things are now more declarative than imperative, and that frees the library to make decisions about where each step should run. For instance, knowing which steps of the underlying algorithm can be parallelized, the library can use an `ExecutorService` to execute some of the steps in parallel.

One concrete example of such a functional API is the `Stream` API introduced in Java 8 and its `parallel()` flavor. Logging with a MDC in a parallel `Stream` is not a free lunch: one need to ensure the MDC is captured and reapplied in each step.

The functional style enables such optimizations, because each step is thread-agnostic and referentially transparent, but it can break the MDC assumption of a single `Thread`. The most idiomatic way of ensuring any kind of contextual information is accessible to all stages would be to pass that context around through the composition chain. During the development of Reactor we encountered the same general class of problem, and we wanted to avoid this very hands-down and explicit approach. This is why the `Context` was introduced: it propagates through the execution chain as long as `Flux` and `Mono` are used as the return value, by letting stages (operators) peek at the `Context` of their downstream stage. So instead of using `ThreadLocal`, Reactor offers this map-like object that is tied to a `Subscription` and not a `Thread`.

Now that we’ve established that MDC "just working" is not the best assumption to make in a declarative API, how can we perform contextualized log statements in relation to events in a Reactive Stream (`onNext`, `onError`, and `onComplete`)?

This entry of the FAQ offers a possible intermediate solution when one wants to log in relation to these signals in a straightforward and explicit manner. Make sure to read the Adding a Context to a Reactive Sequence section beforehand, and especially how a write must happen towards the bottom of the operator chain for operators above it to see it.

To get contextual information from the `Context` to the MDC, the simplest way is to wrap logging statements in a `doOnEach` operator with a little bit of boilerplate code. This boilerplate depends on both the logging framework/abstraction of your choice and the information you want to put in the MDC, so it has to be in your codebase.

The following is an example of such a helper function around a single MDC variable and focused on logging `onNext` events, using Java 9 enhanced `Optional` API:

```
public static <T> Consumer<Signal<T>> logOnNext(Consumer<T> logStatement) {
    return signal -> {
        if (!signal.isOnNext()) return; ①
        Optional<String> toPutInMdc =
signal.getContext().getOrEmpty("CONTEXT_KEY"); ②

        toPutInMdc.ifPresentOrElse(tpim -> {
            try (MDC.MDCCloseable cMdc = MDC.putCloseable("MDC_KEY", tpim)) { ③
                logStatement.accept(signal.get()); ④
            }
        },
        () -> logStatement.accept(signal.get())); ⑤
    };
}
```

① `doOnEach` signals include `onComplete` and `onError`. In this example we're only interested in logging `onNext`

② We will extract one interesting value from the Reactor `Context` (see the The `Context` API section)

③ We use the `MDCCloseable` from SLF4J 2 in this example, allowing try-with-resource syntax for automatic cleanup of the MDC after the log statement is executed

④ Proper log statement is provided by the caller as a `Consumer<T>` (consumer of the onNext value)

⑤ In case the expected key wasn't set in the `Context` we use the alternative path where nothing is put in the MDC

Using this boilerplate code ensures that we are good citizens with the MDC: we set a key right before we execute a logging statement and remove it immediately after. There is no risk of polluting the MDC for subsequent logging statements.

Of course, this is a suggestion. You might be interested in extracting multiple values from the `Context` or in logging things in case of `onError`. You might want to create additional helper methods for these cases or craft a single method that makes use of additional lambdas to cover more ground.

In any case, the usage of the preceding helper method could look like the following reactive web controller:

```
@GetMapping("/byPrice")
public Flux<Restaurant> byPrice(@RequestParam Double maxPrice,
@RequestHeader(required = false, name = "X-UserId") String userId) {
    String apiId = userId == null ? "" : userId; ①

    return restaurantService.byPrice(maxPrice))
               .doOnEach(logOnNext(r -> LOG.debug("found restaurant {} for ${}",
②
                    r.getName(), r.getPricePerPerson())))
               .subscriberContext(Context.of("CONTEXT_KEY", apiId)); ③
}
```

① We need to get the contextual information from the request header to put it in the `Context`

② Here we apply our helper method to the `Flux`, using `doOnEach`. Remember: operators see `Context` values defined below them.

③ We write the value from the header to the `Context` using the chosen key `CONTEXT_KEY`.

In this configuration, the `restaurantService` can emit its data on a shared thread, yet the logs will still reference the correct `X-UserId` for each request.

For completeness, we can also see what an error-logging helper could look like:

```
public static Consumer<Signal<?>> logOnError(Consumer<Throwable>
errorLogStatement) {
    return signal -> {
        if (!signal.isOnError()) return;
        Optional<String> toPutInMdc =
signal.getContext().getOrEmpty("CONTEXT_KEY");

        toPutInMdc.ifPresentOrElse(tpim -> {
            try (MDC.MDCCloseable cMdc = MDC.putCloseable("MDC_KEY", tpim)) {
                errorLogStatement.accept(signal.getThrowable());
            }
        },
        () -> errorLogStatement.accept(signal.getThrowable()));
    };
}
```

Nothing much has changed, except for the fact that we check that the `Signal` is effectively an `onError`, and that we provide said error (a `Throwable`) to the log statement lambda.

Applying this helper in the controller is very similar to what we’ve done before:

```
@GetMapping("/byPrice")
public Flux<Restaurant> byPrice(@RequestParam Double maxPrice,
@RequestHeader(required = false, name = "X-UserId") String userId) {
    String apiId = userId == null ? "" : userId;

    return restaurantService.byPrice(maxPrice))
               .doOnEach(logOnNext(v -> LOG.info("found restaurant {}", v))
               .doOnEach(logOnError(e -> LOG.error("error when searching
restaurants", e)) ①
               .subscriberContext(Context.of("CONTEXT_KEY", apiId));
}
```

① In case the `restaurantService` emits an error, it will be logged with MDC context here

Appendix C: Reactor-Extra

The `reactor-extra` artifact contains additional operators and utilities that are for users of `reactor-core` with advanced needs.

As this is a separate artifact, you need to explicitly add it to your build:

```
dependencies {
     compile 'io.projectreactor:reactor-core'
     compile 'io.projectreactor.addons:reactor-extra' ①
}
```

① Add the reactor extra artifact in addition to core. See Getting Reactor for details about why you don't need to specify a version if you use the BOM, usage in Maven, etc...

C.1. `TupleUtils` and Functional Interfaces

The `reactor.function` package contains functional interfaces that complement the Java 8 `Function`, `Predicate` and `Consumer` interfaces, for 3 to 8 values.

`TupleUtils` offers static methods that act as a bridge between lambdas of these functional interfaces to a similar interface on the corresponding `Tuple`.

This allows to easily work with independent parts of any `Tuple`, for instance:

```
.map(tuple -> {
  String firstName = tuple.getT1();
  String lastName = tuple.getT2();
  String address = tuple.getT3();

  return new Customer(firstName, lastName, address);
});
```

Can be instead written as

```
.map(TupleUtils.function(Customer::new)); ①
```

① (as `Customer` constructor conforms to `Consumer3` functional interface signature)

C.2. Math Operators With `MathFlux`

The `reactor.math` package contains a `MathFlux` specialized version of `Flux` that offers mathematical operators, like `max`, `min`, `sumInt`, `averageDouble`...

C.3. Repeat and Retry Utilities

The `reactor.retry` package contains utilities to help in writing `Flux#repeatWhen` and `Flux#retryWhen` functions. The entry points are factory methods in respectively `Repeat` and `Retry` interfaces.

Both interfaces can be used as a mutative builder AND are implementing the correct `Function` signature to be used in their counterpart operators.

Since 3.2.0, one of the most advanced retry strategies offered by these utilities is also part of the `reactor-core` main artifact directly: exponential backoff is available as the `Flux#retryBackoff` operator.

C.4. Schedulers

Reactor-extra comes with several specialized schedulers:

- `ForkJoinPoolScheduler` in package `reactor.scheduler.forkjoin`: uses the Java `ForkJoinPool` to execute tasks.
- `SwingScheduler` in package `reactor.swing` executes tasks in the Swing UI event loop thread, the `EDT`.
- `SwtScheduler` in package `reactor.swing` executes tasks in the SWT UI event loop thread.