1. **Spring Web Reactive Framework**

**Reactive Programming –**

* Reactive programming is about non-blocking applications that are asynchronous and event-driven and require a small number of threads to scale. A key aspect of that definition is the concept of **backpressure** which is a mechanism to ensure producers don’t overwhelm consumers. For example, in a pipeline of reactive components that extends from the database to the HTTP socket when the HTTP client is slow the data repository slows down or stops until capacity frees up. From a programming model perspective reactive programming involves a major shift from imperative style logic to a declarative composition of async logic. It is comparable to using CompletableFuture in Java 8 and composing follow-up actions via lambda expressions.
* **What is Backpressure concept** – Due to the non-blocking nature of Reactive Programming, the server doesn't send the complete stream at once. It can push the data concurrently as soon as it is available. Thus, the client waits less time to receive and process the events. But, there are issues to overcome. **Backpressure in software systems is the capability to overload the traffic communication**. In other words, emitters of information overwhelm consumers with data they are not able to process. Eventually, people also apply this term as the mechanism to control and handle it. It is the protective actions taken by systems to control downstream forces.

A reactive system meets these demands by being **responsive, resilient, elastic and message driven**. While reactive systems refer to the architecture or high-level design of a system, reactive programming refers to an implementation technique that can be used in a reactive architecture. Reactive programming is driven by events and focuses on the flow of data in a non-blocking, asynchronous way. Reactive programming is the foundation of Spring WebFlux, an alternative way of developing web applications. Spring WebFlux makes it possible to build reactive applications on the HTTP layer. It is a reactive fully non-blocking, annotation-based web framework built on Project Reactor that supports reactive streams back pressure and runs on non-blocking servers such as Netty, Undertow and Servlet 3.1+ containers. Non-blocking servers are generally based on the event loop model which uses only a small number of threads handling requests. When talking about non-blocking or asynchronous request processing, it means no thread is in a waiting state. Essentially, threads are able to complete their task without waiting for previous tasks to be completed.

**Responsive systems** focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. [Note: In microservice environment this can be improved using the non-blocking/asynchronous communication/ Reactive Programming Libraries like: RxJava, Reactor Framework]

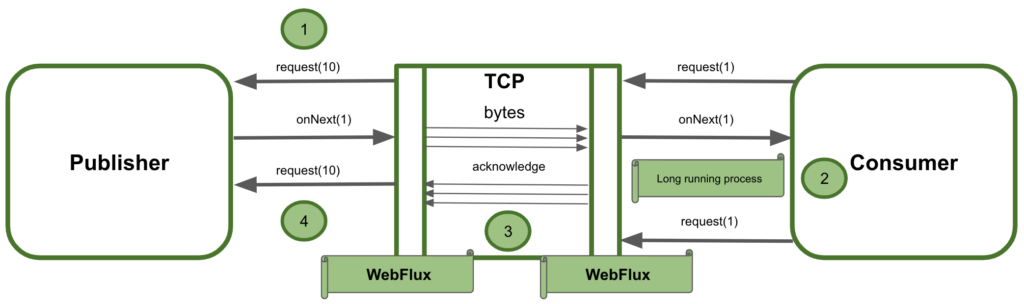
**Resilient**: The system stays responsive in the case of failure. Failures should be handled proactively. [Note: In microservice environment it can be improved using the circuit breaker pattern, load balancing, retry and timeouts]

**Elastic**: The system stays responsive under varying workload. [Note: In microservice environment it can be done using docker, Kubernetes and respective cloud platforms like AKS (azure Kubernetes service), AWS’s EKS (Elastic Kubernetes service), AWS’s ECS (Elastic Container Service) ].

**Message Driven**: The components of the system should be loosely coupled. These components communicate with each other using the asynchronous message-passing technique. [Note: In microservices this can be achieved using Messaging Systems like ActiveMQ, RabbitMQ or Kafka].

* **Handling Backpressure in Spring WebFlux: -**

[Spring WebFlux](https://www.baeldung.com/spring-webflux)**provides an asynchronous non-blocking flow of reactive streams**. The responsible for backpressure within Spring WebFlux is the [Project Reactor](https://projectreactor.io/docs/core/release/reference/#reactive.backpressure). It internally uses [Flux functionalities](https://projectreactor.io/docs/core/release/reference/#_on_backpressure_and_ways_to_reshape_requests) to apply the mechanisms to control the events produced by the emitter. WebFlux uses TCP flow control to regulate the backpressure in bytes. But it does not handle the logical elements the consumer can receive. Let's see the interaction flow happening under the hood. WebFlux framework is responsible for the conversion of events to bytes in order to transfer/receive them through TCP. It may happen that the consumer starts and long-running job before requesting the next logical element While the receiver is processing the events, WebFlux enqueue bytes without acknowledgment because there is no demand for new events Due to the nature of the TCP protocol, if there are new events the publisher will continue sending them to the network

[](https://www.baeldung.com/wp-content/uploads/2021/04/Screenshot-2021-03-19-at-16.40.30-1024x304-1.png)

In conclusion, the diagram above shows that the demand in logical elements could be different for the consumer and the publisher. Spring WebFlux does not ideally manage backpressure between the services interacting as a whole system. It handles it with the consumer independently and then with the publisher in the same way. But it is not considering the logical demand between the two services.

**Reactive streams and Project Reactor**

Spring WebFlux supports the Reactive Stream API, which is a standardized tool for processing asynchronous streams with non-blocking backpressure. Backpressure is a way of dealing with a data stream that may be too large to be reliably processed. In other words, backpressure refers to the ability to request data when the consumer is ready to process them. Reactive streams have a publisher (producer) — subscriber (consumer) model. The publisher emits an event, and a subscriber will read it. In the Reactive Streams API, there are four main interfaces:

* Publisher — Emits events to subscribers based on the demands received from its subscribers. A publisher can serve multiple subscribers and it has only one method: subscribe
* Subscriber — Receives events emitted by the Publisher. The subscribe has four methods to deal with the events received: onSubscribe, onNext, onError and onComplete
* Subscription — Represents the relationship between the subscriber and the publisher. It has methods that allow requesting for data request(long n) and to cancel the demand of events cancel()
* Processor — Publisher and subscriber at the same time; rarely used.

Spring WebFlux internally uses Project Reactor and its publisher implementations, Flux and Mono.

* Mono — A publisher that can emit 0 or 1 element.
* Flux — A publisher that can emit 0..N elements.

Mono and Flux offer simple ways of creating streams of data:

Reactor offers several operators for working with Flux and Mono objects. Most commonly used are:

* Map — Used to transform the publisher elements to another elements
* FlatMap — Similar to map, but transformation is asynchronous
* FlatMapMany — Mono operator used to transform a Mono into a Flux
* DelayElements — Delays the publishing of each element by a given duration
* Concat — Used to combine publishers’ elements by keeping the sequence of the publishers
* Merge — Used to combine publishers without keeping the publishers’ sequence, instead it interleaves the values
* Zip — Used to combine two or more publishers by waiting on all the sources to emit one element and combining these elements into an output value.

**More detail can be found** [**here**](https://myoffice.accenture.com/personal/ram_prakash_singh_accenture_com/Documents/Microsoft%20Teams%20Chat%20Files/ReactiveServer%20(1).pptx?web=1)

**Here1**

**Advantage of using Reactive Programming:**

* Better performance i.e., better use of CPU as opposed to single-threaded execution. Netty server uses an event loop model where a request is handled by a thread
* Maximum throughput can be achieved as calls are non-blocking.
* Better user experience.
* All the expensive operations like database calls and REST API calls are taken care of by the worker thread and our application is not blocked.

**Cons of using Reactive Programming**:

* Limited documentation and resource availability.
* Doesnt supports the ORM framework which is widely used in many applications.
* There is no strong community for problem resolution.
* Proper mono and flux creation should be done otherwise we can end up executing the code outside the reactive pipeline.

**Use Case:**

**Basket:**

The existing ecommerce application was unable to scale and support the high load of data and the number of users.

To overcome this issue and to handle high load and produce maximum throughput, we use reactive programming in the basket use case.

To convert basket to order we need an asynchronous and nonblocking setup since the number of users making orders is generally high, and our application is supporting multiple channels meaning data is coming from various sources in parallel.

The flow is as follows:

* Checkout will call basket service to post the order
* Basket service performs business logic, inserts the data in DB, and publishes a message to the order topic (which further talks to OMS to create actual order)
* Performs transformation and prepares response

In this flow, Basket is not waiting for a response and all the tasks are executed in parallel

**Fulfillment:**

In Fulfillment service the requirement is to display all the fulfillment options based on availability. Currently, there are four fulfillment types supported Home Delivery, Click and Collect, PUDO, and digital. In this use case also since there are multiple clients and an asynchronous setup is required, the calls are made in parallel to check for all four fulfillment types and then all the responses are aggregated and returned to consumers.

**Use Case for Backpressure:**

The downstream OMS system was not able to consume messages at a rate, by which we were publishing the basket object to be converted into an order. We leveraged the backpressure feature of Reactive programming where we controlled the rate of publishing the message to OMS, hence saving the downstream system from crashing.

**When not to use Reactive Programming**

Fulfillment service had a requirement where in business users can configure different fulfillment types, shipping methods, and rules around the shipping methods. Those endpoints were also provided in the same codebase where we were using reactive programming.

In this particular use case, we realized that we are incurring extra costs by using the reactive database as the data is only configuration data and there will be a smaller number of calls. After analyzing we separated out the codebase for those endpoints and used spring boot and hibernate instead of reactive programming.