Reactive Programming

Use cases:

1.Capital One redesigned its auto loan application around Reactive principles to simplify online car shopping and financing. Customers can browse more than four million cars from over 12,000 dealers and pre-qualify for financing in seconds, without impacting credit scores.

2.LinkedIn turned to Reactive principles to build real-time presence indicators (online indicators) for the half billion users on its social network.

3.Verizon Wireless, operators of the largest 4G LTE network in the United States, slashed response times in half using Reactive principles in the upgrade of its e-commerce website that supports 146 million subscribers handling 2.5 billion transactions a year.

4.Walmart Canada rebuilt its entire Web application and mobile stack as a Reactive system and saw a 20 percent increase in conversion to sales from web traffic and a 98 percent increase in mobile orders while cutting page load times by more than a third.

Blocking request processing

In traditional MVC applications, when a request come to server, a servlet thread is created. It delegates the request to worker threads for I/O operations such as database access etc.

During the time worker threads are busy, servlet thread (request thread) remain in waiting status and thus it is blocked. It is also called synchronous request processing.
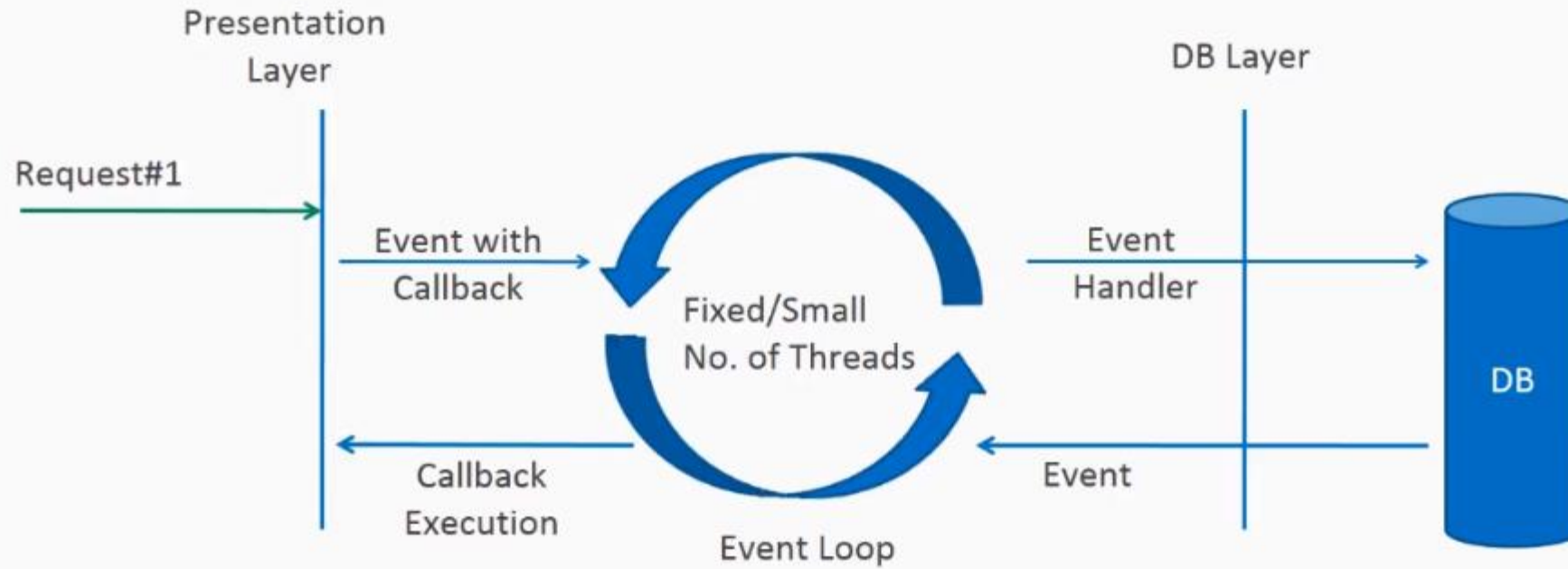
Blocking request processing
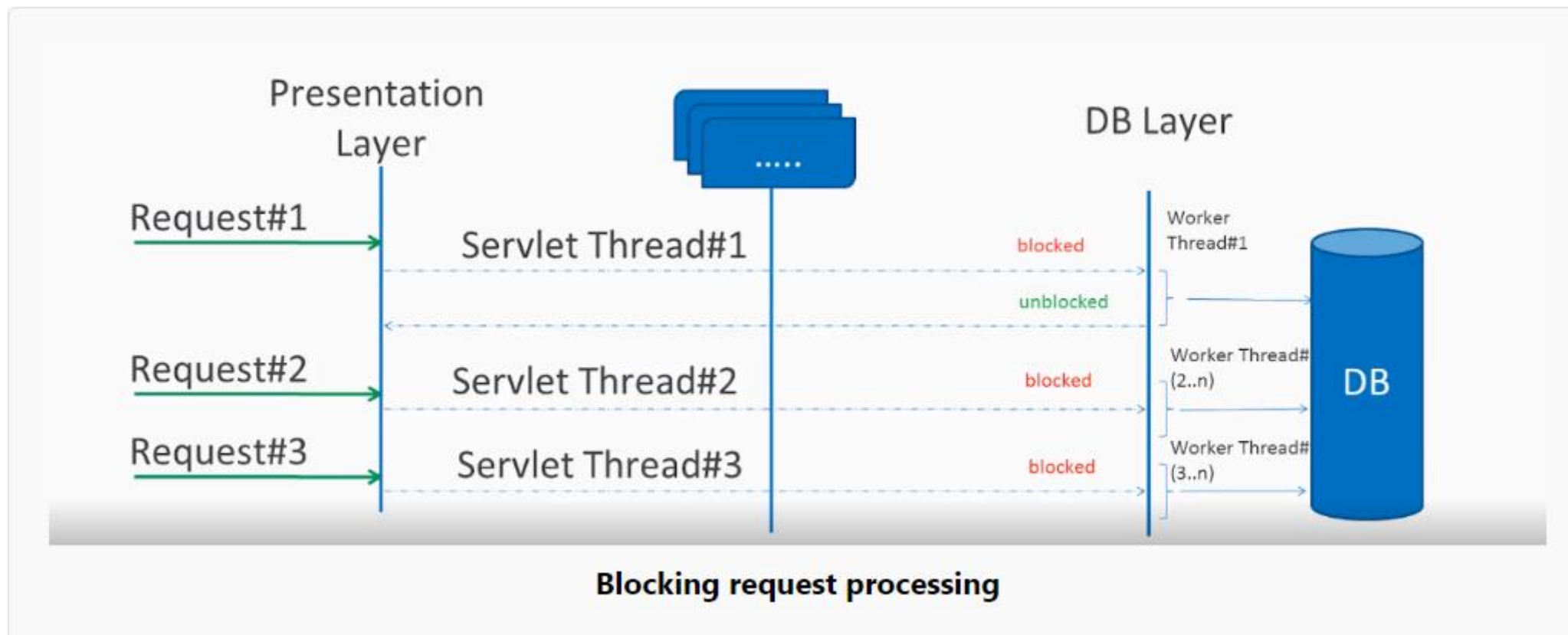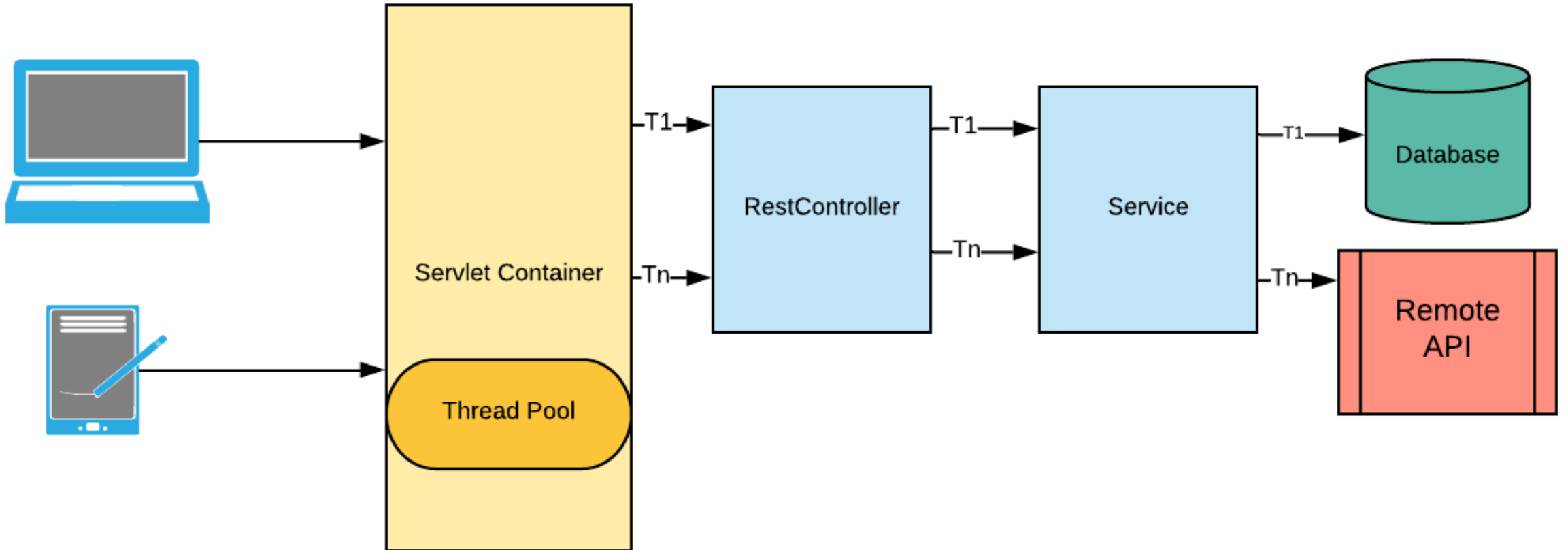
Non-blocking request processing

In non-blocking or asynchronous request processing, no thread is in waiting state. There is generally only one request thread receiving the request.

All incoming requests come with a event handler and call back information. Request thread delegates the incoming requests to a thread pool (generally small number of threads) which delegate the request to it's handler function and immediately start processing other incoming requests from request thread.
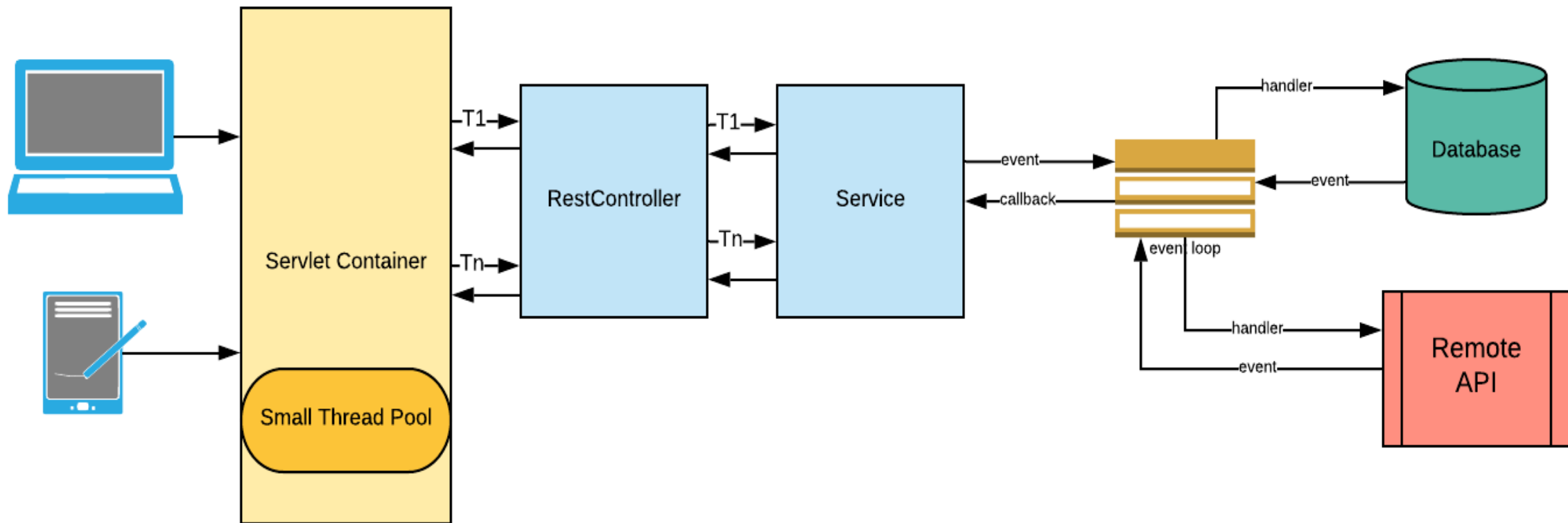
When the handler function is complete, one of thread from pool collect the response and pass it to the call back function.

Non-blocking request processing

Spring MVC Request/Response Flow
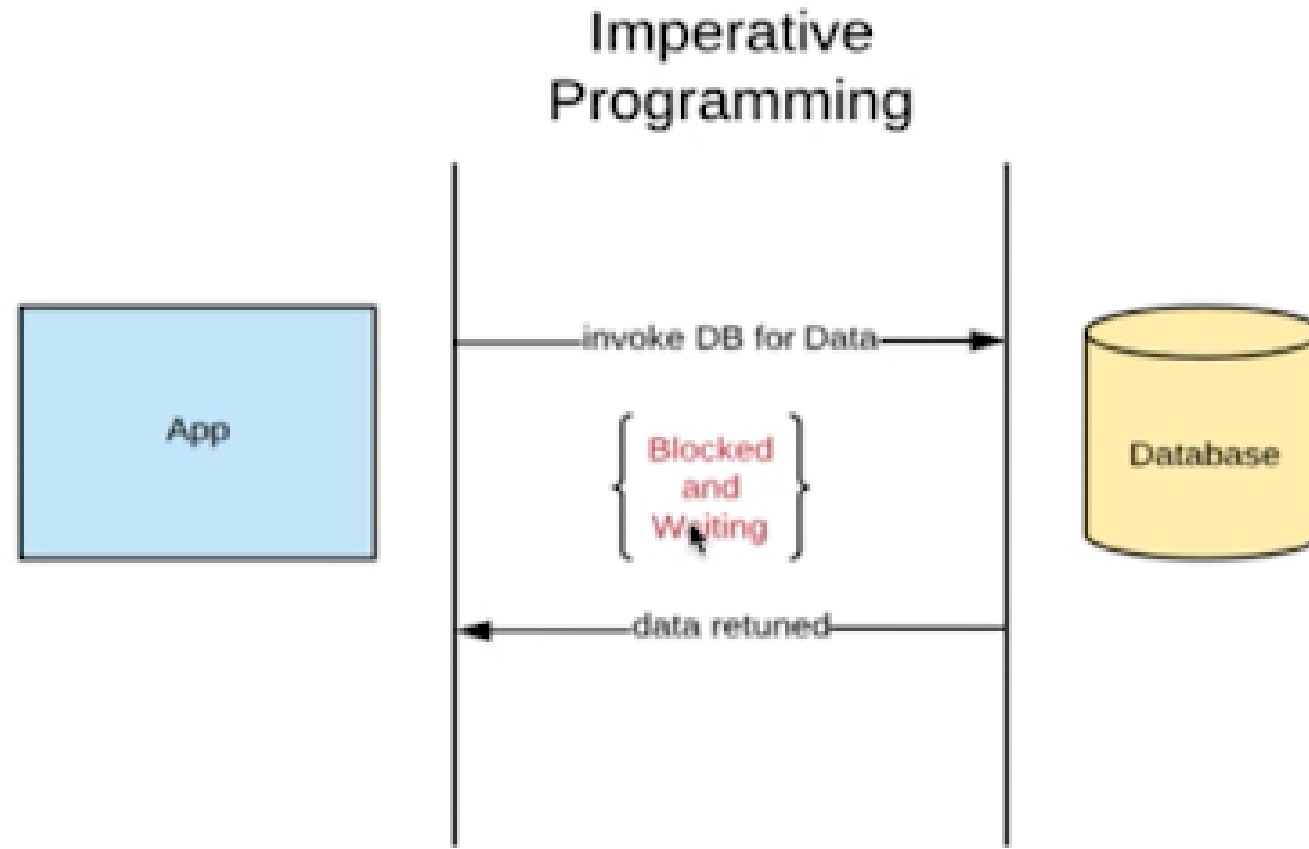
Reactive Request Processing

Imperative Programming

Blocking, Synchronous & No Backpressure

Imperative Programming, Blocking & synchronous:

Imperative
Programming

App

invoke DB for Data

Blocked
and
Waiting

data retuned

Database

```java
@GetMapping("/v1/items/{id}")
public ResponseEntity<Item> getItemFromExternalServices(@PathVariable Integer id){

    //1st db call
    Price price = priceRepository.findById(id).get();  // 1.db-call  2.blocking

    //2nd rest call
    ResponseEntity<Location> locationEntity =
            restTemplate.getForEntity(locationUrl,Location.class);  // 1.rest call synchronous  2.blocking

    Item item = buildItem(price,locationEntity.getBody());

    return ResponseEntity.ok(item);

}
```

The first a DB call is made (proceRepository.finById(id) method call), which is
blocking and then a rest API call is made (restTemplate.getForEntity() method call)
and then Item is built.

If you look at the code, it is in sequential mode and it is imperative style API - where
the flow is Top-Down approach. So, the above code is blocking and synchronous.

What is reactive programming?

The term, "reactive," refers to programming models that are built around reacting to changes.

It is build around publisher-subscriber pattern (observer pattern).

In reactive style of programming, we make a request for resource and start performing other things. When the data is available, we get the notification along with data inform of call back function.

In callback function, we handle the response as per application/user needs.

No Backpressure

```java
@GetMapping("/v1/items")
public ResponseEntity<List<Item>> getAllItems(){

    List<Item> items = itemRepository.getAllItems();

    return ResponseEntity.ok(items);

}
```

Application → getAllItems() → Database

Database → Data Returned → Application

Let's imagine the getAllItems() is trying to fetch a huge set of the records, then the application will crash with "Out of memory" error.

Also, the client will be overwhelmed with huge data. It would be nice if there is a possibility for the client to say "Slow Down" by applying backpressure, then things will slow down momentarily and will pick up from where they left.

In this approach, there is no back pressure possibility for the client to say slow down.

One important thing to remember is back pressure. In non-blocking code, it becomes important to control the rate of events so that a fast producer does not overwhelm its destination.

Reactive web programming is great for applications that have streaming data, and clients that consume it and stream it to their users. It is not great for developing traditional CRUD applications. If you're developing the next Facebook or Twitter with lots of data, a reactive API might be just what you're looking for.

Asynchronous in Java

It is possible to make the logic asynchronous in Java with the following concepts, but they have their own limitations.

Callbacks

- Complex
- No Return Value
- Code is hard to read and maintain

Futures

- Returns Future instance
- Hard to compose multiple async operations

Completable Future

- Introduced in Java 8
- Supports functional style API
- Not a great fit asynchronous call with multiple items

Java CompletableFuture API and reactive streams — relate to each other?

The main difference is that a CompletableFuture represents one result of an asynchronous call, while Reactive Streams is a pattern for pushing N messages asynchronously through a system.

CompletableFuture doesn't (and couldn't) address the elasticity requirement of the Reactive Manifesto by providing backpressure, as it operates on the level of a single call. For any reactive streams implementation, on the other hand, backpressure is a central concept.

Scenario 1: Single Calls to Microservices

When your code calls a remote microservice or a slow database to retrieve a specific result to fulfill the current request, it is probably NOT the place to use a reactive stream.

One result is all you ever asked for. You are much better off using the methods in the CompletableFuture API to add a custom callback, specifying what to do with the result once it's ready. In other words, this is a perfect use case for the CompletableFuture API.
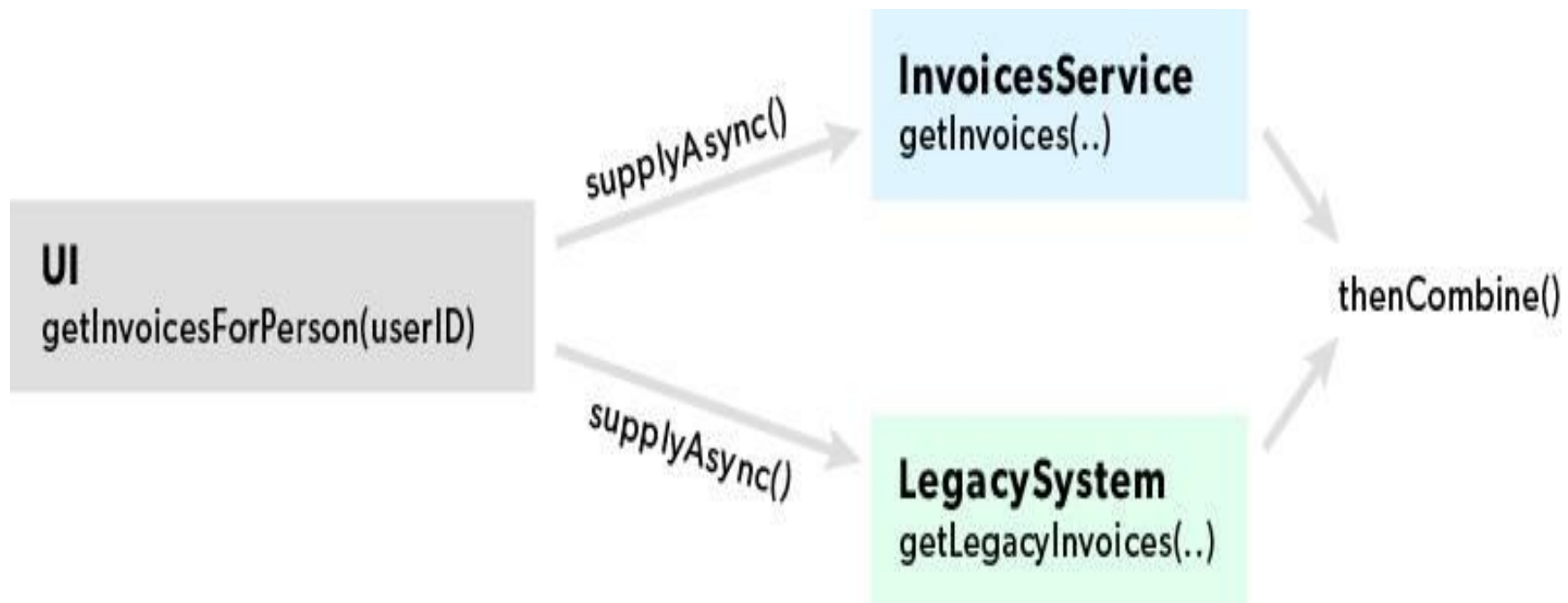
Scenario 2: A Stream of Asynchronously Processed Data

Imagine a scenario where you are truly operating on a stream of messages. Let's say you have an SMSGateway that receives SMS messages as they come in from users. Each message needs to be processed by SMSProcessor (a remote microservice) and contains a command that takes variable time to execute.

You could implement this scenario based on CompletableFuture API.

SMSGateway
Publisher<SMS>

stream of SMS objects

SMSProcessor
Subscriber<SMS>

onNext(SMS)

incoming SMS-s
from users

What if SMSProcessor is overwhelmed by previous messages, and not yet ready to receive more data? Such streaming scenarios are much better taken care of by the Reactive Streams approach, implemented by frameworks like RxJava or Project Reactor.

Single asynchronous operations where CompletableFuture helps to keep things simple, and true streaming scenarios where we use reactive streams.

We can say that reactive streams take the same concept and generalize it one step further.

What is Reactive Programming?

The term, "reactive," refers to programming models that are built around reacting to changes.

It is a new programming paradigm, built around publisher-subscriber pattern (observer pattern), asynchronous and Non Blocking, where the Data flows as an Event/Message Driven stream, Functional Style code (not Imperative) and Back Pressure on Data Streams.

What is Reactive Programming ?

- Reactive Programming is a new programming paradigm

- Asynchronous and non blocking

- Data flows as an **Event/Message** driven stream

- Functional Style Code

- BackPressure on Data Streams

Data flow as Event Driven Stream in Reactive Programming

In Reactive programming, there will always one event or message for every result item from Data Source.

The possible Data sources are Database, External File Service, File etc. There will be always one Event or Message emitted for completion or error.

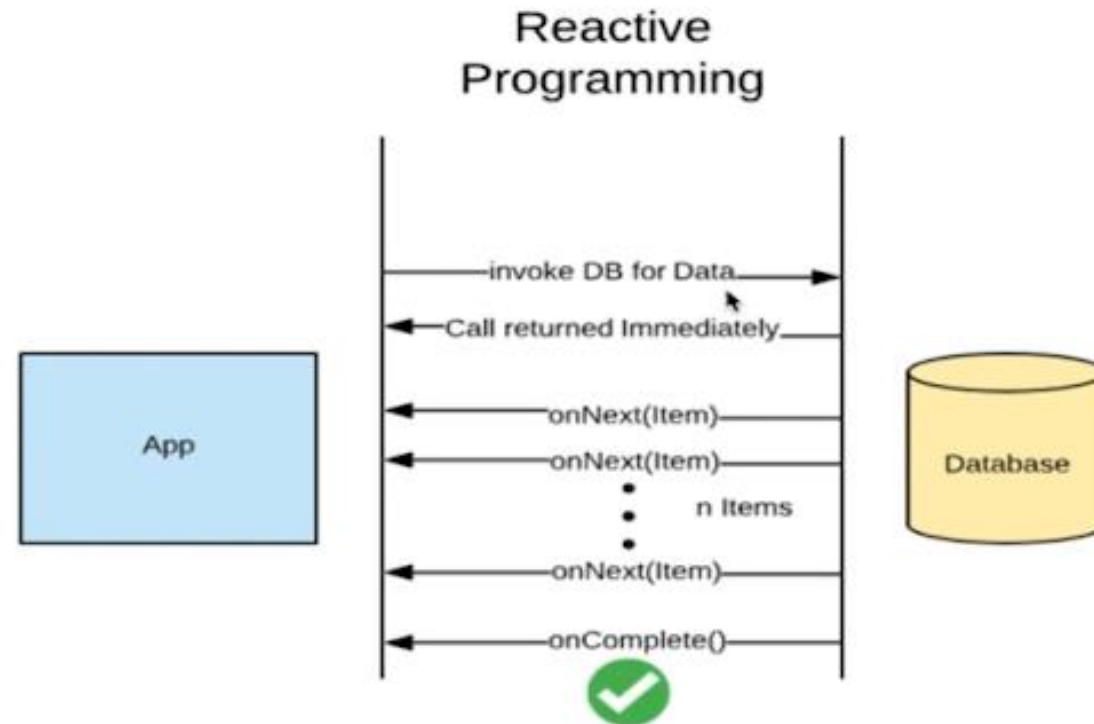Following are the 3 methods that will form the life cycle:

OnNext(item) – Data Stream Events

OnComplete() – Completion or Success Event

OnError() – Error Event

**Happy path - where there is no Error:**

In Reactive world, the call to getAllItems() will invoke the data from Database, but the response call will be returned immediately to the client, whereas the data will be sent one after the other using onNext(Item) call.
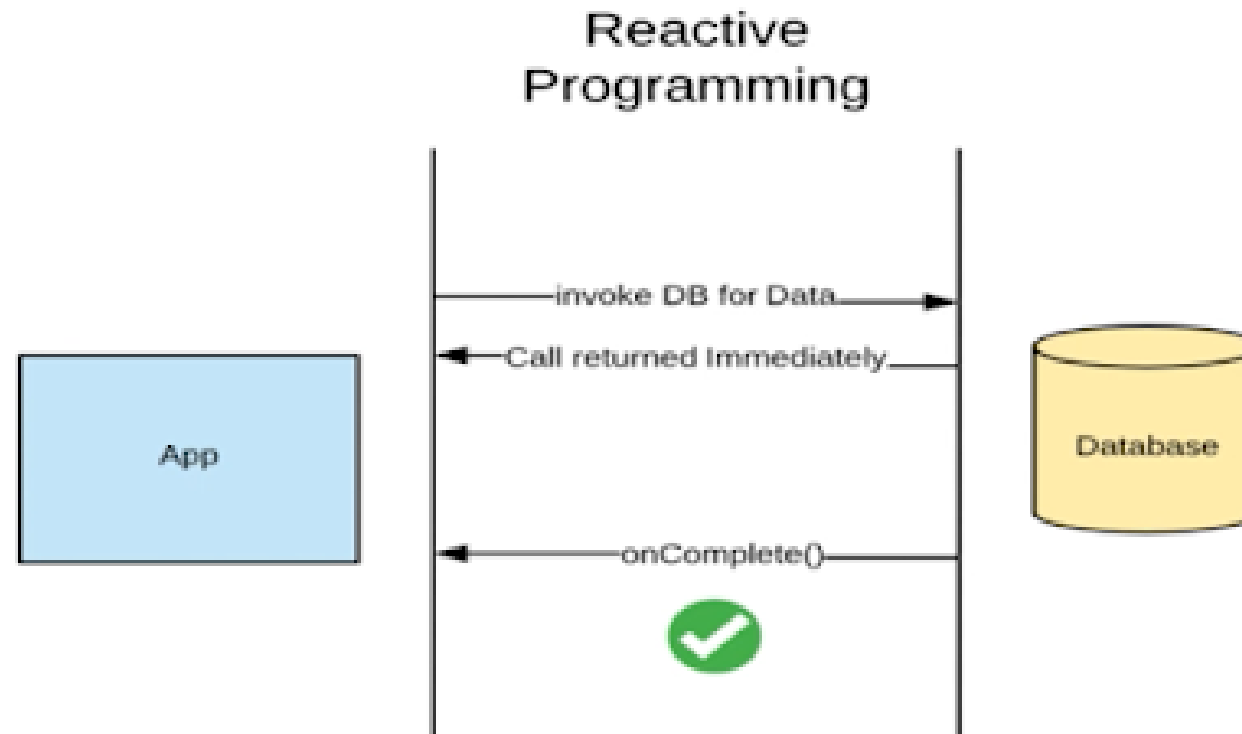
Once all the data is pushed to the client (via onNext(Item)), the onComplete() method will be invoked to say that the transaction is completed.

Error case:
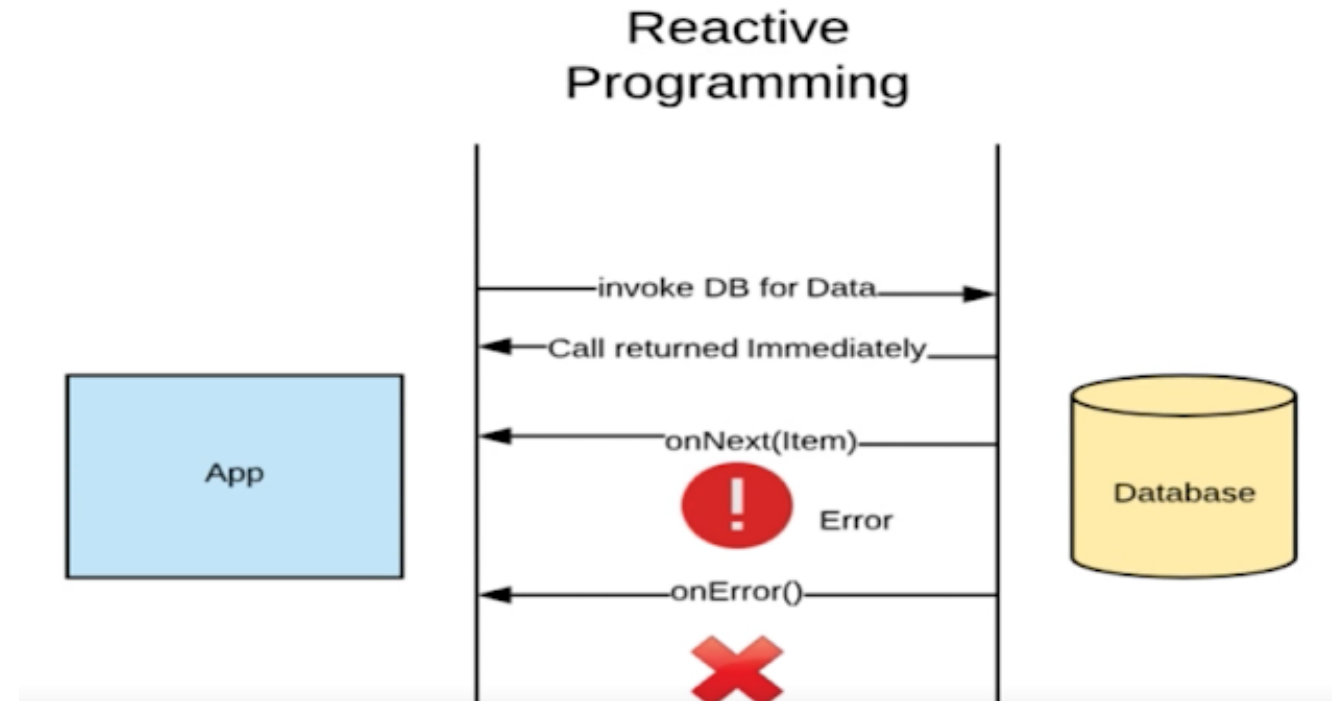
Incase of any error during while passing the data to the client in onNext(Item) method call, the onError() method will be triggered and will communicate the client about the error.

No-data flow:

Incase of no data found, then the transaction will be made complete with onComplete() call.

What is Reactive Streams Specification?

The new Reactive Streams Specification is the specification or Rules for Reactive Streams. It was created by engineers from Netflix, Pivotal, Lightbend, RedHat, Twitter, and Oracle, among others and is now part of Java 9.

It defines four interfaces:

- ✓ Publisher
- ✓ Subscriber
- ✓ Subscription
- ✓ Processor

Publisher: Emits a sequence of events to subscribers according to the demand received from its subscribers. A publisher can serve multiple subscribers.

It has a single method:

```
public interface Publisher<T>
{
    public void subscribe(Subscriber<? super T> s);
}
```

Publisher represents the DataSource.
  -> Database
  -> External Service

Subscriber: Receives and processes events emitted by a Publisher. Please note that no notifications will be received until Subscription#request(long) is called to signal the demand.

It has four methods to handle various kind of responses received.

```
public interface Subscriber<T>
{
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

**Note: Subscriber is always attached to the publisher**

Subscription: Defines a one-to-one relationship between a Publisher and a Subscriber. It can only be used once by a single Subscriber. It is used to both signal desire for data and cancel demand (and allow resource cleanup).

```
public interface Subscription<T>
{
    public void request(long n);
    public void cancel();
}
```

Processor: Represents a processing stage consisting of both a Subscriber and a Publisher and obeys the contracts of both.

```
public interface Processor<T, R> extends Subscriber<T>,
Publisher<R>
{
}
```

Publisher/Subscriber Event Flow
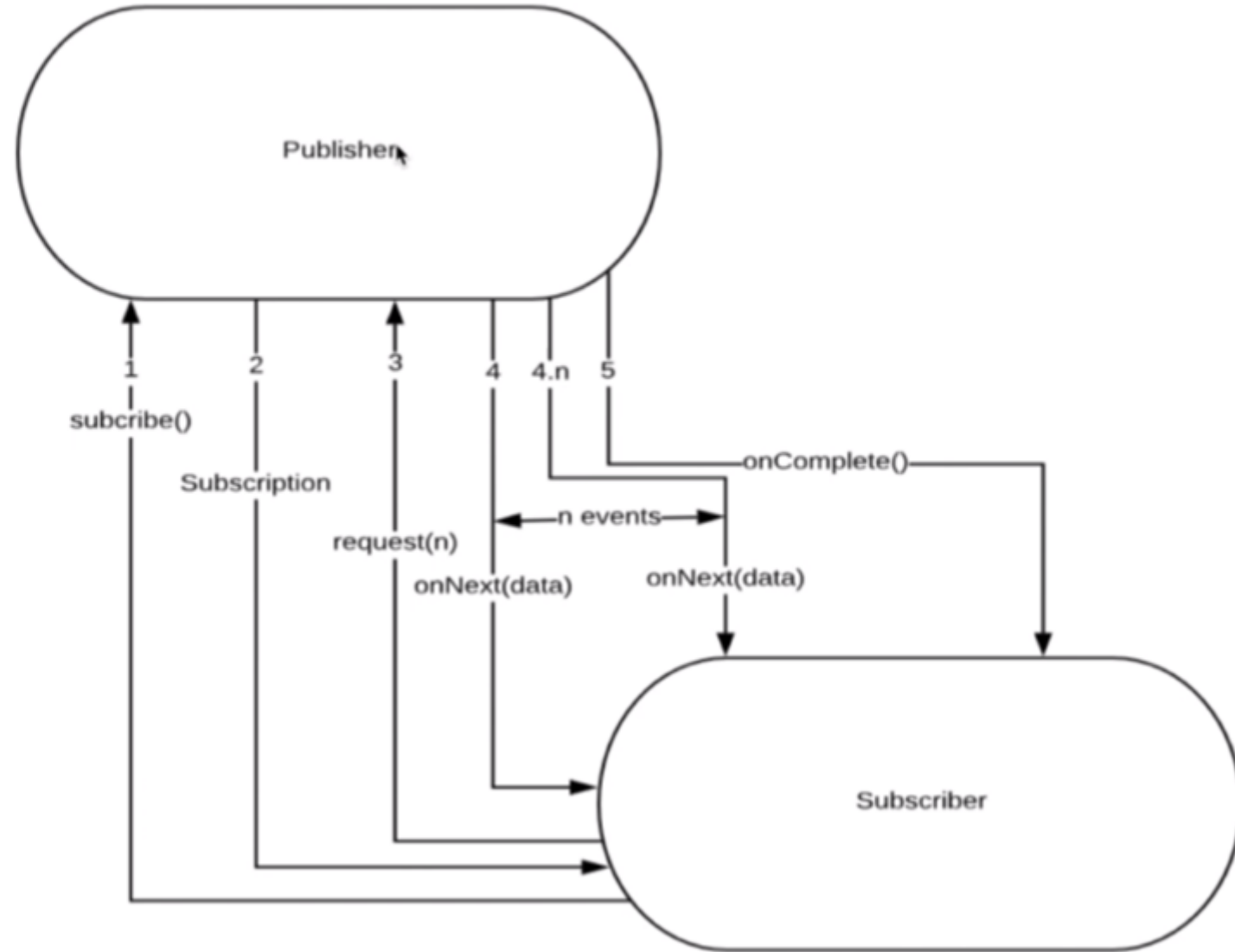
What is Reactive Library?

Reactive Library is the implementation of the above Reactive Stream Specification. The following are the 3 popular Reactive libraries in Java.

RxJava
Project Reactor
Flow Class in JDK 9

Project Reactor was built and maintained by Pivotal.
  It is the recommended library for Spring Boot.

https://projectreactor.io/

reactor-core

- **Flux** and **Mono**

- Reactive Types of project reactor.

- **Flux** – Represents 0 to N elements

- **Mono** – Represents 0 to 1 element.

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 |

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.

operator

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.

Top - Marble diagram (represents the data)
Middle – Operator represents filtering, transformation, enrich etc.,
Bottom – Marble diagram (final output sent to the subscriber)

Examples : More than one data from the database (or) rest endpoint

Flux - 0 to N elements

```
Flux.just("Spring","Boot","Reactive)
        .map(s -> s.concat("Flux")
        .subscribe("System.out::println");
```

reactor.core.publisher

Class Flux<T>

java.lang.Object
    reactor.core.publisher.Flux<T>

Type Parameters:

T - the element type of this Reactive Streams `Publisher`

All Implemented Interfaces:

Publisher<T>

Direct Known Subclasses:

ConnectableFlux, FluxOperator, FluxProcessor, GroupedFlux

```
public abstract class Flux<T>
extends Object
implements Publisher<T>
```

A Reactive Streams `Publisher` with rx operators that emits 0 to N elements, and then completes (successfully or with an error).

The recommended way to learn about the `Flux` API and discover new operators is through the reference documentation, rather than through this javadoc (as opposed to learning more about individual operators). See the "which operator do I need?" appendix.
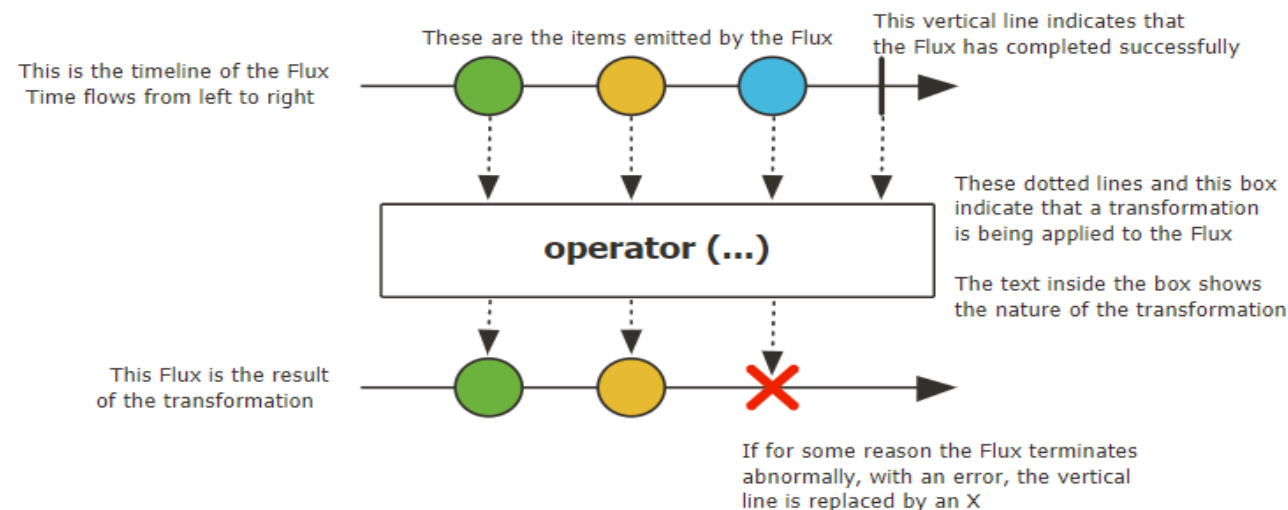
Modifier and Type	Method and Description
Mono<Boolean>	**all**(Predicate<? super T> predicate) Emit a single boolean true if all values of this sequence match the Predicate.
Mono<Boolean>	**any**(Predicate<? super T> predicate) Emit a single boolean true if any of the values of this Flux sequence match the predicate.
<P> P	**as**(Function<? super Flux<T>,P> transformer) Transform this Flux into a target type.
T	**blockFirst**() Subscribe to this Flux and **block indefinitely** until the upstream signals its first value or completes.
T	**blockFirst**(Duration timeout) Subscribe to this Flux and **block** until the upstream signals its first value, completes or a timeout expires.
T	**blockLast**() Subscribe to this Flux and **block indefinitely** until the upstream signals its last value or completes.
T	**blockLast**(Duration timeout) Subscribe to this Flux and **block** until the upstream signals its last value, completes or a timeout expires.
Flux<List<T>>	**buffer**() Collect all incoming values into a single List buffer that will be emitted by the returned Flux once this Flux completes.
Flux<List<T>>	**buffer**(Duration bufferingTimespan) Collect incoming values into multiple List buffers that will be emitted by the returned Flux every bufferingTimespan.
Flux<List<T>>	**buffer**(Duration bufferingTimespan, Duration openBufferEvery) Collect incoming values into multiple List buffers created at a given openBufferEvery period.
Flux<List<T>>	**buffer**(Duration bufferingTimespan, Duration openBufferEvery, Scheduler timer) Collect incoming values into multiple List buffers created at a given openBufferEvery period, as measured on the provided Scheduler.
Flux<List<T>>	**buffer**(Duration bufferingTimespan, Scheduler timer) Collect incoming values into multiple List buffers that will be emitted by the returned Flux every bufferingTimespan, as measured on the provided Scheduler.

buffer

```
public final Flux<List<T>> buffer(int maxSize)
```

Collect incoming values into multiple `List` buffers that will be emitted by the returned `Flux` each time the given max size is reached or once this Flux completes.
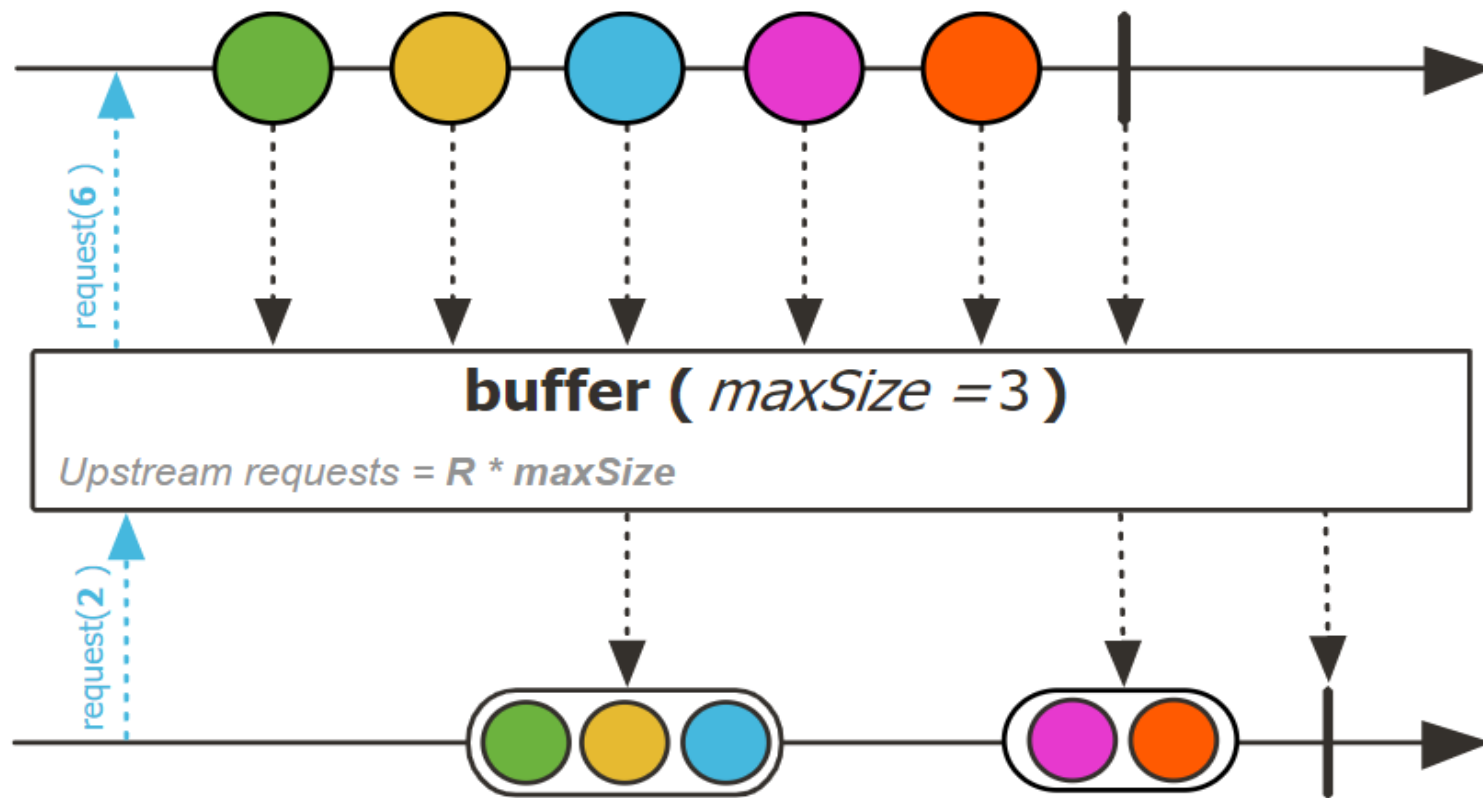
Parameters:

`maxSize` - the maximum collected size

Returns:

a microbatched `Flux` of `List`

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.

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.

operator

This Mono is the result of the transformation.

If for some reason the Mono terminates abnormally, with an error, the vertical line is replaced by an X.

Examples : Only one data from the database (or) rest endpoint

Mono – 0 to 1 element

```
Mono.just("Spring")
        .map(s -> s.concat("Mono"))
        .subscribe("System.out::println");
```

Note : Mono contains only one element, if more elements are provided it will through compile-time exception.

Project Reactor

The following are the 3 popular libraries used in Project Reactor:

a) Reactor-core
b) Reactor-test
c) Reactor-netty

Reactor-core:

It is the core library for Project Reactor, implementation of Reactive Stream Specification and it requires Java 8 minimum.

The following are the various Reactor Types of Project Reactor:

- Flux: Represents 0 to N elements
- Mono: Represents 0 to 1 eleme

Flux:

```
Flux.just("A", "B", "C").
        .map(s -> s.concat("flux"))
        .subscribe(System.out::println);
```

The above example where Flux is accepting 3 elements
(A, B and C) and it is using map() function to concat "flux"
so that subscribe() method will the output as:

Aflux
Bflux
Cflux

Mono:

```
Mono.just("A").
        .map(s -> s.concat("mono"))
        .subscribe(System.out::println);
```

The above example where Mono is accepting 1 element (A) and it is using map() function to concat "flux" so that subscribe() method will the output as:

Amono

Reactive systems are:

Responsive: respond in a timely manner if at all possible, responsiveness means that problems can be detected quickly and dealt with accordingly.

Resilient: remain responsive in the event of failure, failures are contained with each component isolating components from each other.

Elastic: stay responsive under varying workload, reactive systems can react to changes in the input rate by increasing or decreasing the resources allocated to services.

Message Driven: rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency.

The Reactive Manifesto

AngularJS

| Component with AsyncPipe | Observable \<Quote[]\> | EventSource |

Spring Boot

| Controller: Flux\<Quote\> | Repository: Flux\<Quote\> |

MongoDB

| Data Source Quotes |
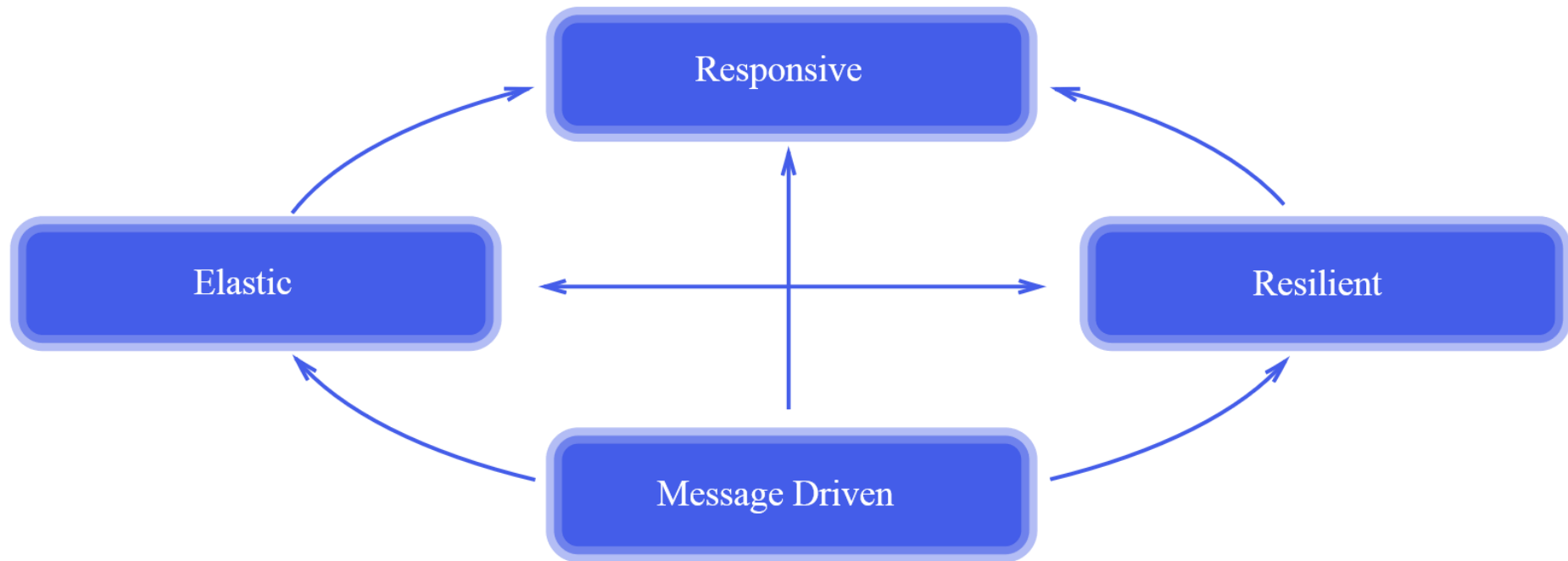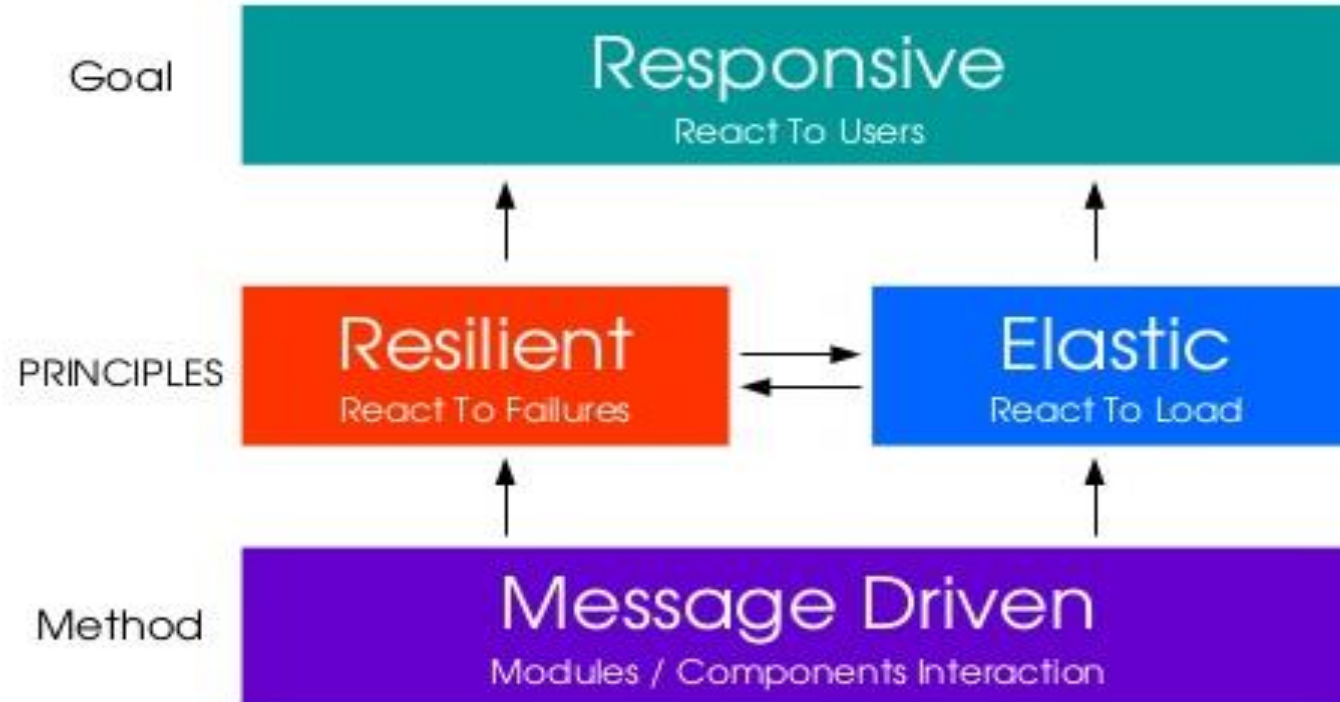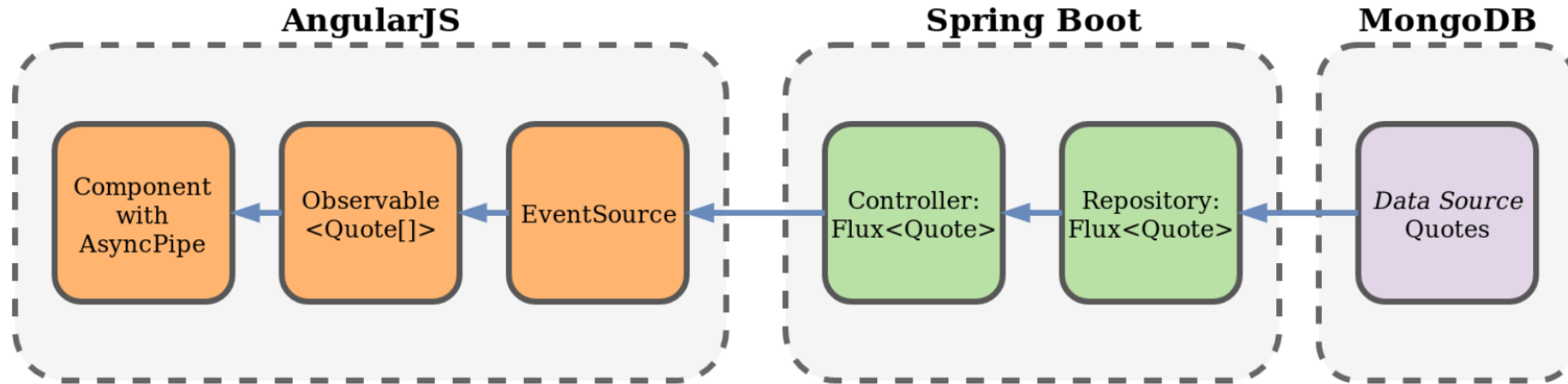
Spring Boot Reactive Web

This is a Spring Boot 2.0 application that retrieves data using Spring Reactive Web (WebFlux), instead of using the standard synchronous MVC framework. It connects to a MongoDB database in a reactive way too.

Angular Reactive

This simple Angular application consumes the controller on the backend side using a reactive approach, Server-Sent Events, so data is loaded on screen as soon as it's available.

Reactive programming is about processing an asynchronous stream of data items, where applications react to the data items as they occur.

A stream of data is essentially a sequence of data items occurring over time.

This model is more memory efficient because the data is processed as streams, as compared to iterating over the in-memory data.

RxJava is a Java VM implementation of ReactiveX a library for composing asynchronous and event-based programs by using observable sequences.

The building blocks of RxJava are Observables and Subscribers. Observable is used for emitting items and Subscriber is used for consuming those items.

Reactive programming is about building asynchronous, non-blocking, and event-driven applications that can easily scale.

Reactor is a Reactive library for building non-blocking applications. It is based on the Reactive Streams Specification in Java 9.

Reactive Streams are push-based. It is the Publisher that notifies the Subscriber of newly available values as they come, and this push aspect is key to being reactive.

The process of restricting the number of items that a subscriber is willing to accept (as judged by the subscriber itself) is called **backpressure** and is essential in prohibiting the overloading of the subscriber (pushing more items that the subscriber can handle).

Reactive Programming

It is an asynchronous programming paradigm focused on streams of data.

Reactive programs also maintain a continuous interaction with their environment, but at a speed which is determined by the environment, not the program itself. Interactive programs work at their own pace and mostly deal with communication, while reactive programs only work in response to external demands and mostly deal with accurate interrupt handling. Real-time programs are usually reactive.

Common Use Cases

> External Service Calls

> Highly Concurrent Message Consumers

> Abstraction Over Asynchronous Processing

Features of Reactive Programming

Data Streams

Asynchronous

Non-blocking

Backpressure

Failures as Messages

Data Streams

> Data Streams can be just about anything.

> Mouse clicks, or other user interactions

➢ JMS Messages, RESTful Service calls, Twitter feed, Stock Trades, list of data from a database.

> A Stream is a sequence of events ordered in time

> Events you want to listen to

Asynchronous

> Events are captured asynchronously

> A function is defined to execute when an event is emitted.

> Another function is defined if an error is emitted.

> Another function is defined when complete is emitted

GoF Observer Pattern

Observer

+notify()

Subject

+observerCollection

+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()

notifyObservers()
   for observer in observerCollection
      call observer.notify()

ConcreteObserverA

+notify()

ConcreteObserverB

+notify()
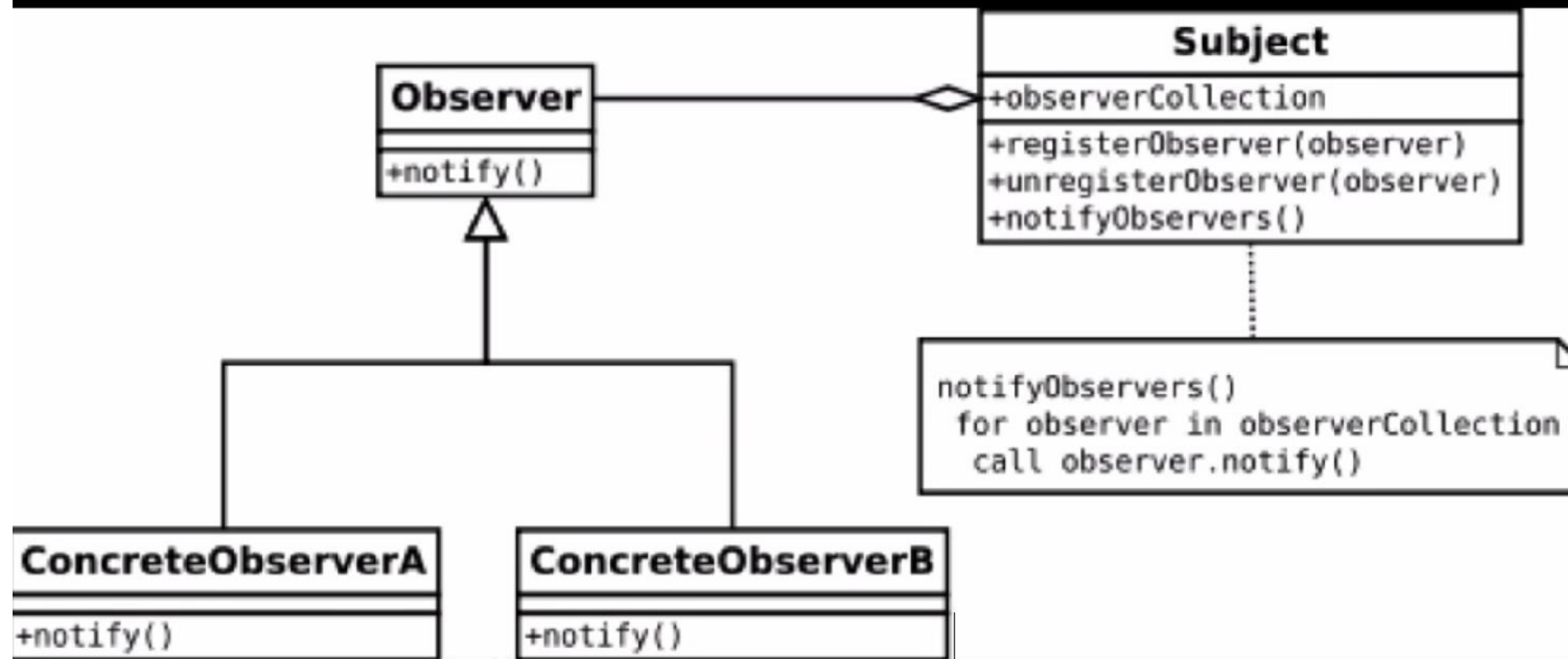
Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure.

This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols.

JDK9 java.util.concurrent.Flow

JDK 9 Flow API

The Flow API (and the Reactive Streams API), in some ways, is a combination of ideas from Iterator and Observer patterns.

The Iterator is a pull model, where the application pulls items from the source.

The Observer is a push model, where the items from the source are pushed to the application.

Using the Flow API, the application initially requests for N items, and then the publisher pushes at most N items to the Subscriber.

So its a mix of Pull and Push programming models.

```
                          Request N
                          data items
┌─────────────────┐  ◄─────────────────────  ┌─────────────────┐
│                 │                           │                 │
│                 │  ─────────────────────►   │                 │
│    Publisher    │                           │   Subscriber    │
│                 │       Push stream of      │                 │
│                 │       N data items        │                 │
└─────────────────┘                           └─────────────────┘
```
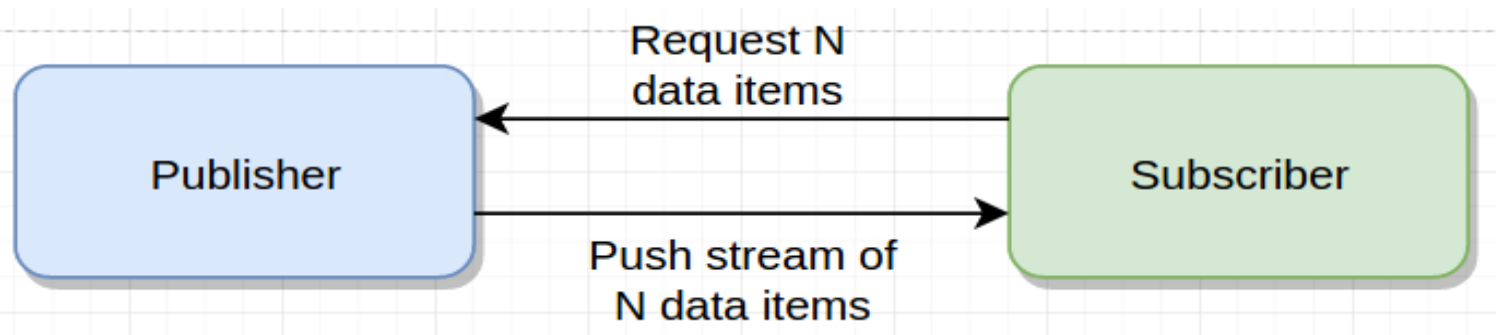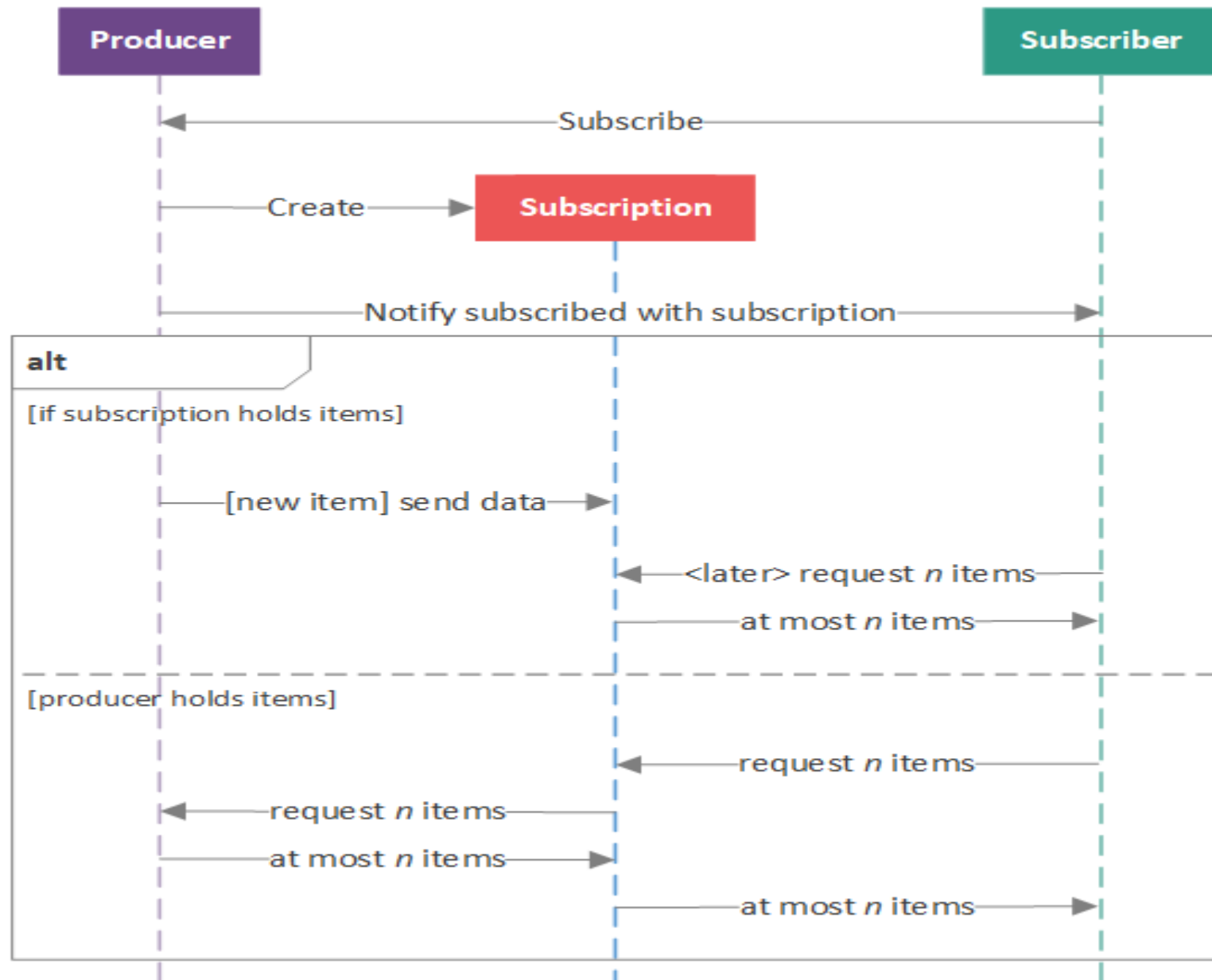
The Flow API Interfaces

```java
@FunctionalInterface
public static interface Flow.Publisher<T> {
    public void    subscribe(Flow.Subscriber<? super T> subscriber);
}

public static interface Flow.Subscriber<T> {
    public void    onSubscribe(Flow.Subscription subscription);
    public void    onNext(T item) ;
    public void    onError(Throwable throwable) ;
    public void    onComplete() ;
}

public static interface Flow.Subscription {
    public void    request(long n);
    public void    cancel() ;
}

public static interface Flow.Processor<T,R>  extends
Flow.Subscriber<T>, Flow.Publisher<R> {
}
```

The Subscriber

The Subscriber subscribes to the Publisher for the callbacks.

Data items are not pushed to the Subscriber unless requested, but multiple items may be requested.

Subscriber method invocations for a given Subscription are strictly ordered.

The application can react to the following callbacks, which are available on the subscriber.

Callback	Description
onSubscribe	Method invoked prior to invoking any other Subscriber methods for the given Subscription.
onNext	Method invoked with a Subscription's next item.
onError	Method invoked upon an unrecoverable error encountered by a Publisher or Subscription, after which no other Subscriber methods are invoked by the Subscription. If a Publisher encounters an error that does not allow items to be issued to a Subscriber, that Subscriber receives onError, and then receives no further messages.
onComplete	Method invoked when it is known that no additional Subscriber method invocations will occur for a Subscription that is not already terminated by error, after which no other Subscriber methods are invoked by the Subscription. When it is known that no further messages will be issued to it, a subscriber receives onComplete.

Sample Subscriber

```java
import java.util.concurrent.Flow.*;
...

public class MySubscriber<T> implements Subscriber<T> {
  private Subscription subscription;

  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1); //a value of  Long.MAX_VALUE may be considered as effectively unbounded
  }

  @Override
  public void onNext(T item) {
    System.out.println("Got : " + item);
    subscription.request(1); //a value of  Long.MAX_VALUE may be considered as effectively unbounded
  }

  @Override
  public void onError(Throwable t) {
    t.printStackTrace();
  }

  @Override
  public void onComplete() {
    System.out.println("Done");
  }
}
```

The Publisher

```java
import java.util.concurrent.SubmissionPublisher;

...

    //Create Publisher
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

    //Register Subscriber
    MySubscriber<String> subscriber = new MySubscriber<>();
    publisher.subscribe(subscriber);

    //Publish items
    System.out.println("Publishing Items...");
    String[] items = {"1", "x", "2", "x", "3", "x"};
    Arrays.asList(items).stream().forEach(i -> publisher.submit(i));
    publisher.close();
```

The Subscription

Links a Flow.Publisher and Flow.Subscriber. Subscribers receive items only when requested, and may cancel at any time, via the Subscription.

Method	Description
`request`	Adds the given number of n items to the current unfulfilled demand for this subscription.
`cancel`	Causes the Subscriber to (eventually) stop receiving messages.

Subscription is a connection between Subscriber and Publisher. Basically, Publisher will create a subscription for every Subscriber which will try to subscribe to it, and this subscription will handle requests from the subscriber. Publisher act as the storage of data and subscription will obtain data from it.

Sample Processor to transform String to Integer

```java
import java.util.concurrent.Flow.*;
import java.util.concurrent.SubmissionPublisher;
...


public class MyTransformProcessor<T,R> extends SubmissionPublisher<R> implements Processor<T, R> {

  private Function function;
  private Subscription subscription;

  public MyTransformProcessor(Function<? super T, ? extends R> function) {
    super();
    this.function = function;
  }

  @Override
  public void onSubscribe(Subscription subscription) {
    this.subscription = subscription;
    subscription.request(1);
  }

  @Override
  public void onNext(T item) {
    submit((R) function.apply(item));
    subscription.request(1);
  }

  @Override
  public void onError(Throwable t) {
    t.printStackTrace();
  }

  @Override
  public void onComplete() {
    close();
  }
}
```

Sample code to transform data stream using processor

```java
import java.util.concurrent.SubmissionPublisher;
...

    //Create Publisher
    SubmissionPublisher<String> publisher = new SubmissionPublisher<>();

    //Create Processor and Subscriber
    MyFilterProcessor<String, String> filterProcessor = new MyFilterProcessor<>(s -> s.equals("x"));

    MyTransformProcessor<String, Integer> transformProcessor = new MyTransformProcessor<>(s -> Integer.parseInt(s));

    MySubscriber<Integer> subscriber = new MySubscriber<>();

    //Chain Processor and Subscriber
    publisher.subscribe(filterProcessor);
    filterProcessor.subscribe(transformProcessor);
    transformProcessor.subscribe(subscriber);

    System.out.println("Publishing Items...");
    String[] items = {"1", "x", "2", "x", "3", "x"};
    Arrays.asList(items).stream().forEach(i -> publisher.submit(i));
    publisher.close();
```

Spring WebFlux

Spring WebFlux Module

Spring Framework 5 includes a new spring-webflux module.

The module contains support for reactive HTTP and WebSocket clients as well as for reactive server web applications including REST, HTML browser, and WebSocket style interactions.

Server Side

On the server-side WebFlux supports 2 distinct programming models:

❑ Annotation-based with @Controller and the other annotations supported also with Spring MVC

❑ Functional, Java 8 lambda style routing and handling Both programming models are executed on the same reactive foundation that adapts non-blocking HTTP runtimes to the Reactive Streams API.

Reactor's types

Reactor's two main types are the Flux<T> and Mono<T>. A Flux is the equivalent of an RxJava Observable, capable of emitting 0 or more items, and then optionally either completing or erroring.

A Mono on the other hand can emit at most once. It corresponds to both Single and Maybe types on the RxJava side. Thus an asynchronous task that just wants to signal completion can use a Mono<Void>.

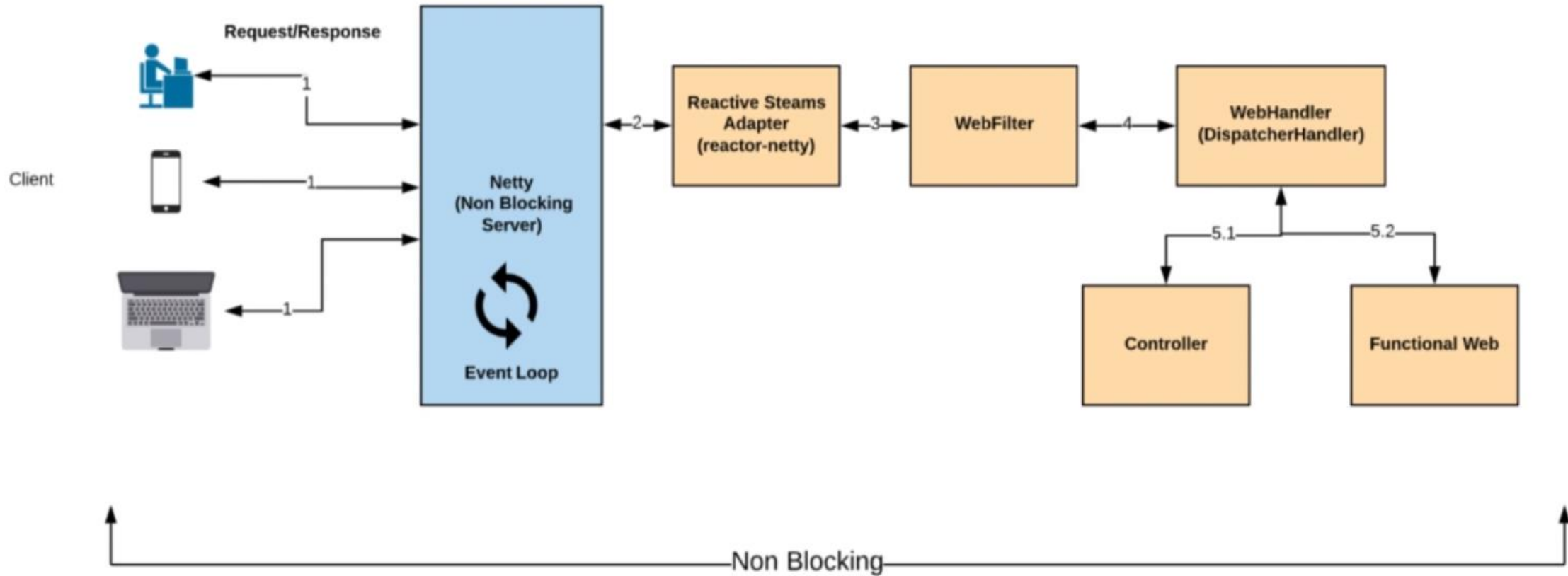| @Controller, @RequestMapping	Router Functions
spring-webmvc	spring-webflux
Servlet API	HTTP / Reactive Streams
Servlet Container	Tomcat, Jetty, Netty, Undertow

Spring WebFlux - Non Blocking Request/Response

Request/Response

Client

Netty (Non Blocking Server)

Event Loop

1

1

1

2

Reactive Steams Adapter (reactor-netty)

3

WebFilter

4

WebHandler (DispatcherHandler)

5.1

5.2

Controller

Functional Web

Non Blocking

Spring WebFlux vs Spring MVC

Spring Webflux + Netty

Spring MVC + Tomcat

Non-blocking Client

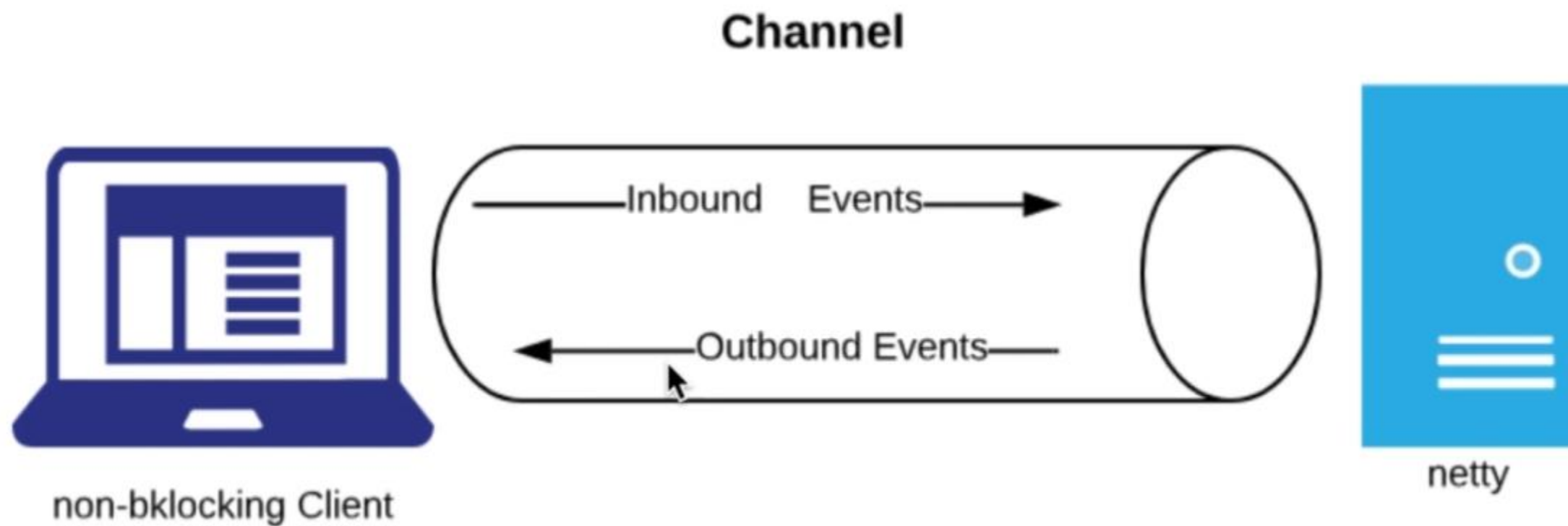Request →

← Future

netty

Blocking Client

Request →

← Response
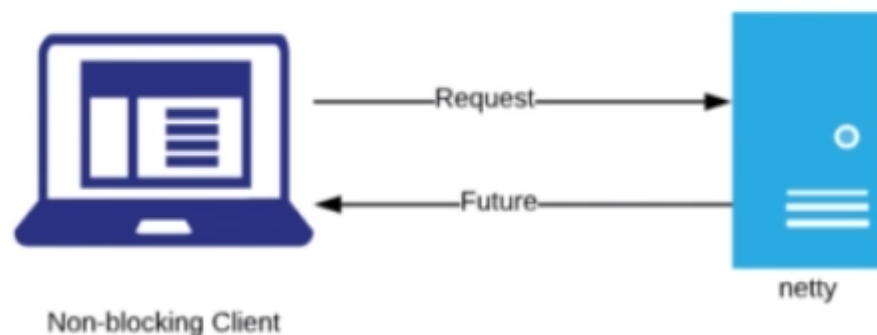
Tomcat
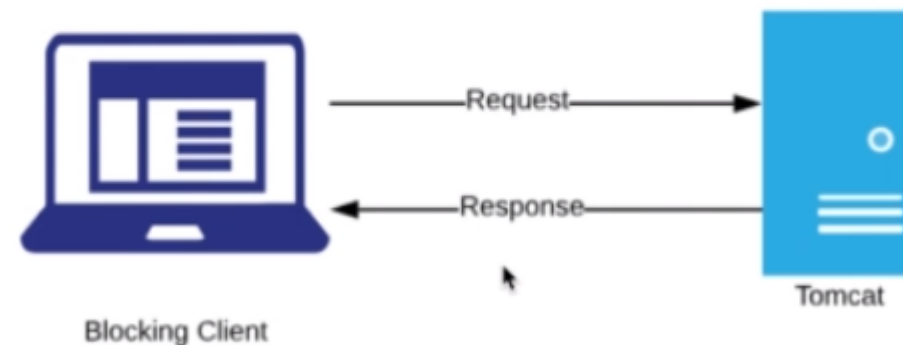
Events in Netty

• Client requesting for a new connection is treated as an event.

• Client requesting for data is treated as an event.

• Client posting for data is treated as an event.

• Errors are treated as event.

Netty – Channel + Events

Netty – Channel + Events

- Inbound events:
  - Requesting for Data
  - Posting Data and etc.,

- Outbound events
  - Opening or Closing a connection.
  - Sending response to the client.
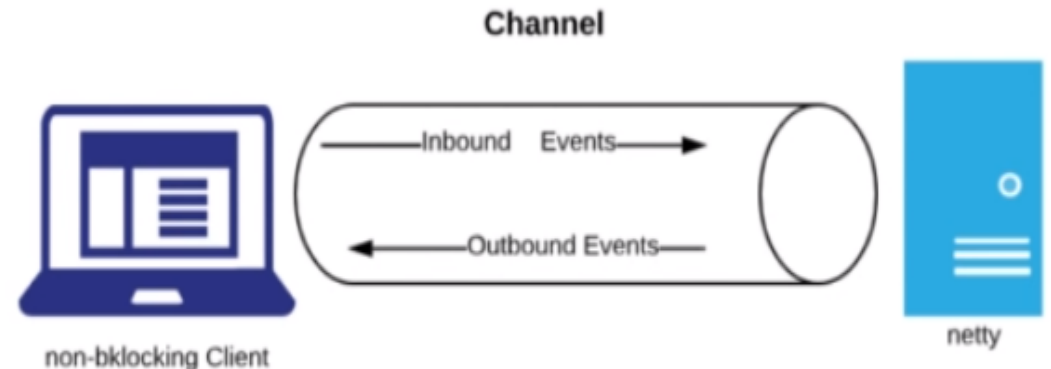
Netty - Event Loop

- Loop the looks for events.

- EventLoop is registered with a **single** dedicated thread.

Event Queue

event 2
event 3
• • •
event n

Event Loop

Channel LifeCycle

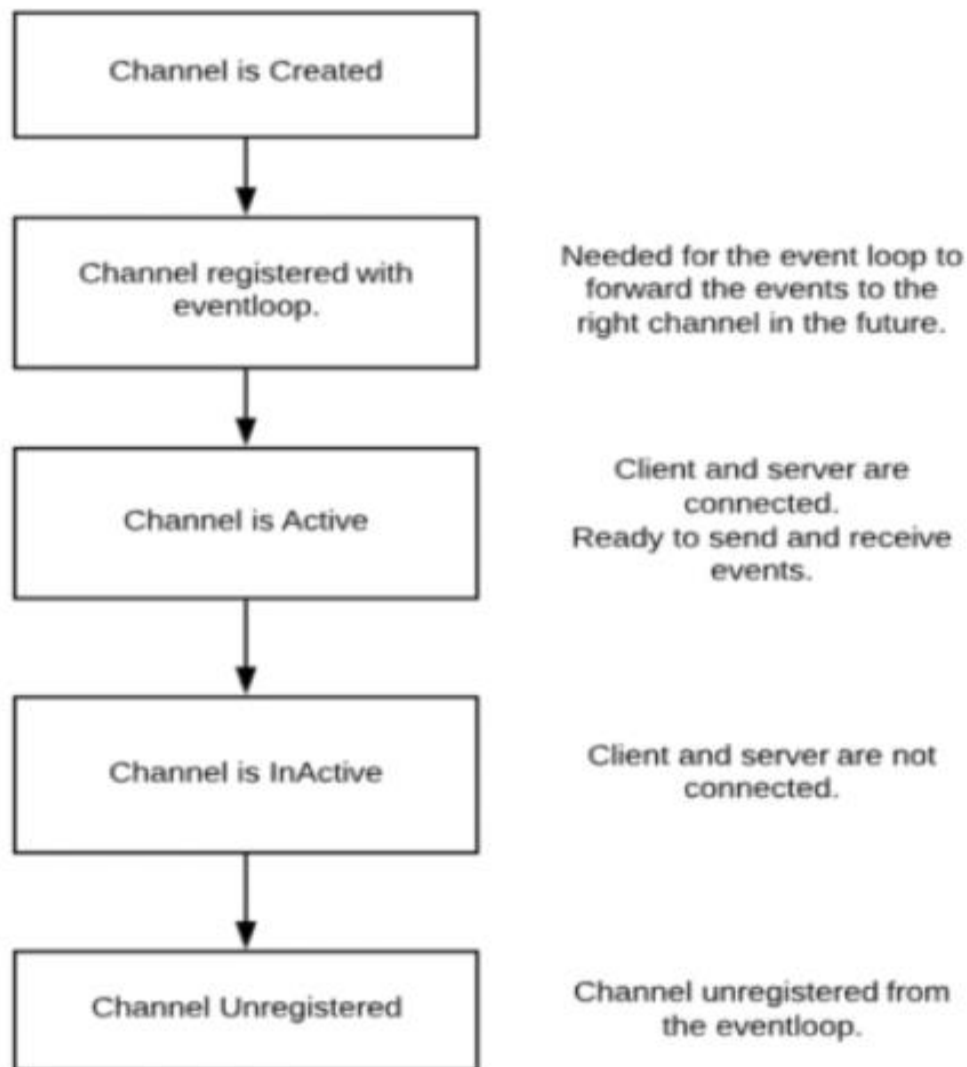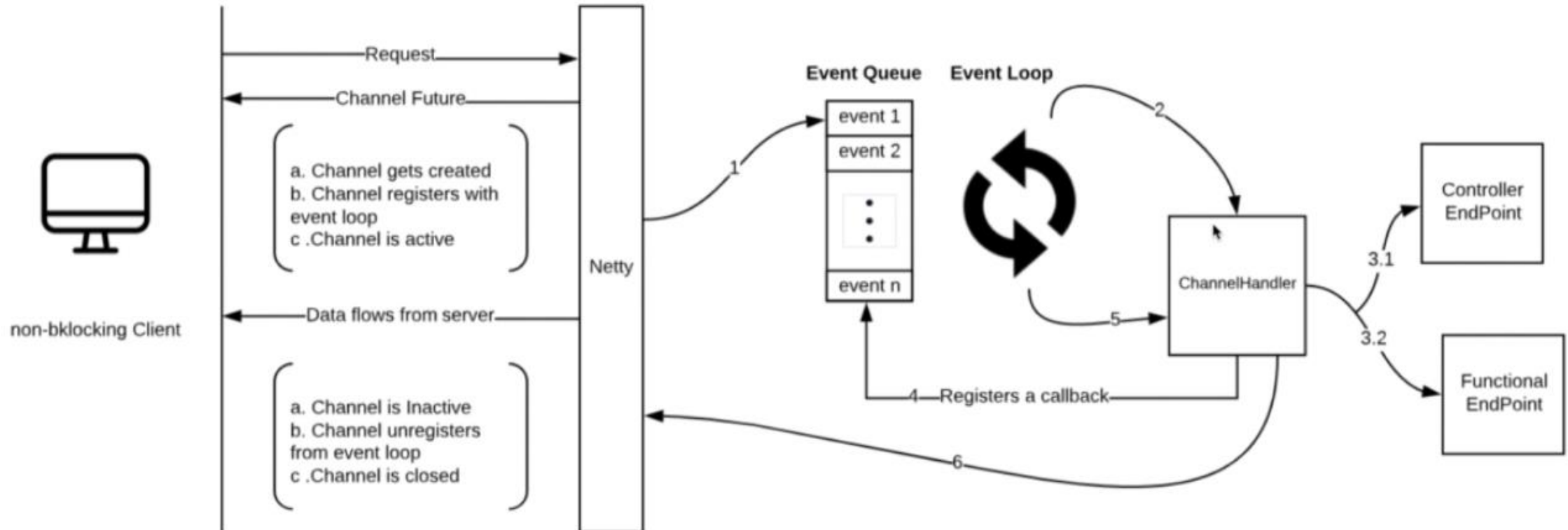| Channel is Created	
↓	
Channel registered with eventloop.	Needed for the event loop to forward the events to the right channel in the future.
↓	
Channel is Active	Client and server are connected. Ready to send and receive events.
↓	
Channel is InActive	Client and server are not connected.
↓	
Channel Unregistered	Channel unregistered from the eventloop.

EventLoop + Channel

Both Spring MVC and Spring WebFlux support client-server architecture but there is a key difference in the concurrency model and the default behavior for blocking nature and threads. In Spring MVC, it is assumed that applications can block the current thread while in webflux, threads are non-blocking by default. It is the main difference between spring webflux vs mvc.

Reactive and non-blocking generally do not make applications run faster. The expected benefit of reactive and non-blocking is the ability to scale the application with a small, fixed number of threads and lesser memory requirements. It makes applications more resilient under load, because they scale in a more predictable manner.

Package org.springframework.data.repository.reactive

Support for reactive repository.

Interface Summary

Interface	Description
ReactiveCrudRepository<T,ID>	Interface for generic CRUD operations on a repository for a specific type.
ReactiveSortingRepository<T,ID>	Extension of ReactiveCrudRepository to provide additional methods to retrieve entities using the sorting abstraction.
RxJava2CrudRepository<T,ID>	Interface for generic CRUD operations on a repository for a specific type.
RxJava2SortingRepository<T,ID>	Extension of RxJava2CrudRepository to provide additional methods to retrieve entities using the sorting abstraction.

RestTemplate Blocking Client

RestTemplate uses the Java Servlet API, which is based on the thread-per-request model.

This means that the thread will block until the web client receives the response. The problem with the blocking code is due to each thread consuming some amount of memory and CPU cycles.

Let's consider having a lot of incoming requests, which are waiting for some slow service needed to produce the result.

Sooner or later, the requests waiting for the results will pile up. Consequently, the application will create many threads, which will exhaust the thread pool or occupy all the available memory. We can also experience performance degradation because of the frequent CPU context (thread) switching.

WebClient Non-Blocking Client

On the other side, WebClient uses an asynchronous, non-blocking solution provided by the Spring Reactive framework.

While RestTemplate uses the caller thread for each event (HTTP call), WebClient will create something like a "task" for each event.

Behind the scenes, the Reactive framework will queue those "tasks" and execute them only when the appropriate response is available.

The Reactive framework uses an event-driven architecture.

It provides means to compose asynchronous logic through the Reactive Streams API.

As a result, the reactive approach can process more logic while using fewer threads and system resources, compared to the synchronous/blocking method.

WebClient is part of the Spring WebFlux library.

Therefore, we can additionally write client code using a functional, fluent API with reactive types (Mono and Flux) as a declarative composition.

Fluent API means to build an API in such way so that it meets the following criteria:

❑ The API user can understand the API very easily.

❑ The API can perform a series of actions in order to finish a task. In Java, we can do it with a series of method calls (chaining of methods).

❑ Each method's name should be domain-specific terminology.

❑ The API should be suggestive enough to guide API users on what to do next and what possible operations users can take at a particular moment.

```java
Mono<Pizza> getPizzaReactive(int id)
{
    return webClient
        .get()
        .uri("http://localhost:8080/pizza/" + id)
        .retrieve()
        .bodyToMono(Pizza.class)
        .onErrorMap(PizzaException::new);
}
```