Reactive programming is a programming pattern designed for

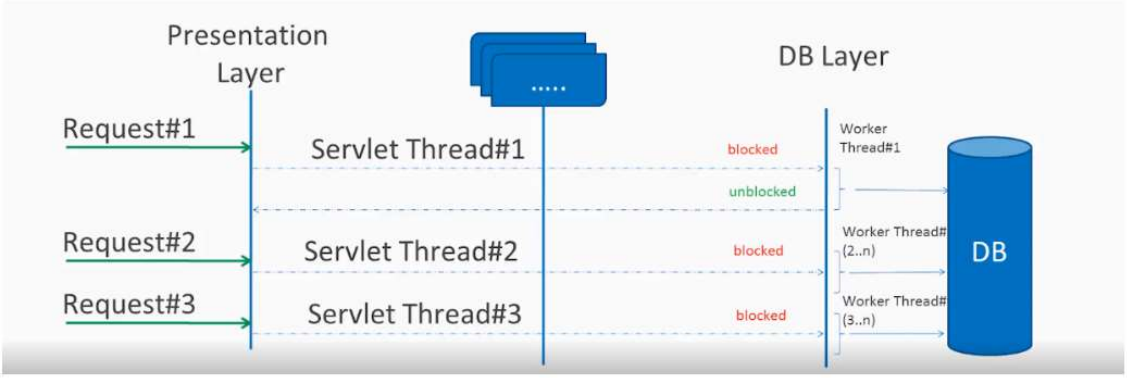
asynchronous, non-blocking and event-driven data processing.

Asynchronous and Non-Blocking

Reactive programming is designed to handle streams of data and events, which makes it perfect for handling asynchronous and non-blocking tasks. Asynchronous tasks are tasks that run independently of the main thread, which means that they can be executed without blocking the main thread. Non-blocking tasks are tasks that do not block the main thread, which means that they can be executed in parallel with other tasks.

Blocking Request Processing

In traditional MVC applications, **a new servlet thread is created (or obtained from the thread pool) when a request comes to the server**. It delegates the request to worker threads for I/O operations such as database access etc. During the time worker threads are busy, the servlet thread (request thread) remains in waiting status, and thus it is blocked. It is also called **synchronous request processing**.



Blocking request processing

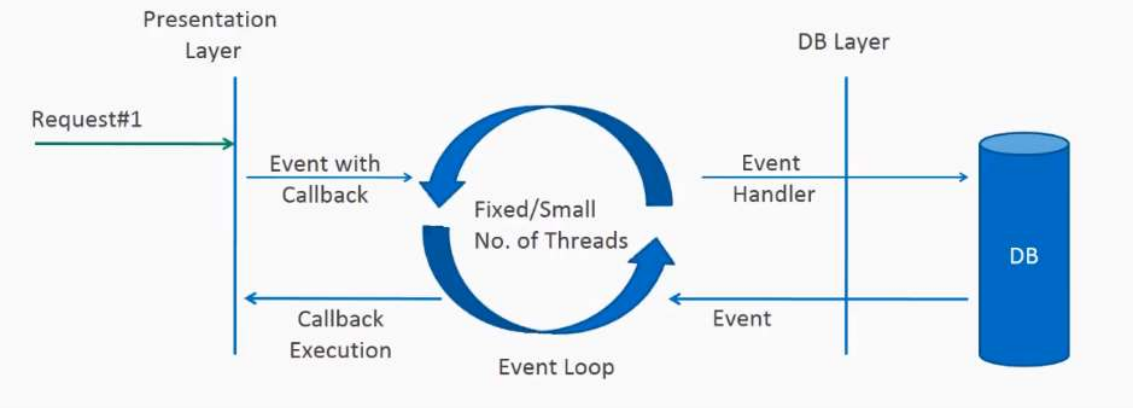
As a server can have some finite number of request threads, it limits the server's capability to process that number of requests at maximum server load. It may hamper the performance and limit the full utilization of server capability.

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 an event handler and callback information. Request thread delegates the incoming requests to a thread pool (generally a small number of threads) which delegates the request to its handler function and immediately starts processing other incoming requests from the request thread.

When the handler function is complete, one thread from the pool collects the response and passes it to the call back function.



Non-blocking request processing

Non-blocking nature of threads helps in scaling the performance of the application. A small number of threads means less memory utilization and less context switching.

it is built around the publisher-subscriber pattern. In the reactive style of programming, we make a request for resources and start performing other things. When the data is available, **we get the notification along with data in the callback function**. The **callback function handles the response** as per application/user needs.

If you are developing the next *Facebook* or *Twitter* with lots of data, real-time analytics applications, chat applications, or live update websites, a reactive API might be just what you are looking for.

Reactive Streams API

The new Reactive Streams API 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

The publisher emits a sequence of events to subscribers according to the demand received from its subscribers. A publisher can serve multiple subscribers.

```
Publisher.java

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

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 kinds of responses received.

```
Subscriber.java

public interface Subscriber<T> {

  public void onSubscribe(Subscription s);
  public void onNext(T t);
  public void onError(Throwable t);
  public void onComplete();
}
```

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 cancels demand (and allow resource cleanup).

Subscription.java

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

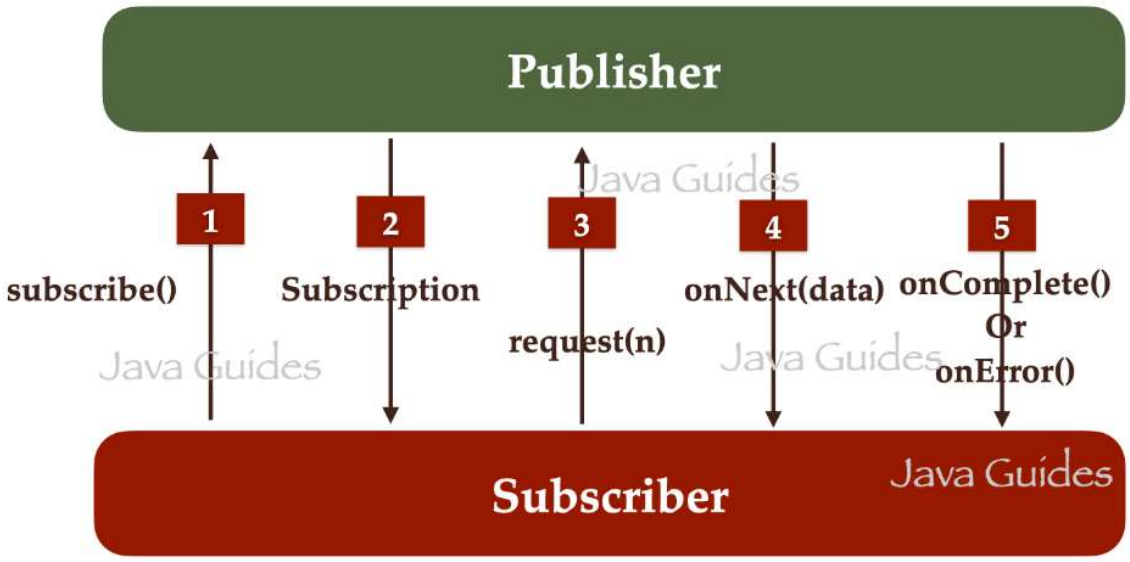
Processor

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

Processor.java

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

Reactive Stream Workflow



1. The *Subscriber* will call *subscribe()* method of the *Publisher* to subscribe or register with the *Publisher*.

2. The Publisher creates an instance of *Subscription* and sends it to *Subscriber* saying that your subscription is successful.

3. Next, the Subscriber will call the *request(n)* method of Subscription to request data from the *Publisher*.

4. Next, *Publisher* call *onNext(data)* method to send data to the *Subscriber*. *Publisher* call *onNext(data)* n times. It means if there are 10 items then the *Publisher* call *onNext(data)* method 10 times.

5. Once the Publisher sends all the data to *Subscriber*, the next *Publisher* call *onComplete()* method to notify *Subscriber* that all the data has been sent. If there are any errors while sending the data then the *Publisher* call *onError()* method to send error details to the *Subscriber*.

Reactive Programming Libraries

A reactive library is nothing but the implementation of reactive specification interfaces. Here are some reactive libraries that are available to us:

- Project Reactor
- RxJava
- JDK 9 Flow Reactive Stream

Project Reactor

The Project Reactor is a fourth-generation reactive library, based on the Reactive Streams specification, for building non-blocking applications on the JVM.

Mono and Flux Implementations

Project reactor libraries provide two implementations of the *Publisher* interface:

1. Mono
2. Flux

Mono: Returns 0 or 1 element.
The Mono API allows producing only one value.

Flux: Returns 0...N elements.
The Flux can be endless; it can produce multiple values.

Mono vs Flux

Mono and Flux are both implementations of the `Publisher` interface. In simple terms, we can say that when we're doing something like a computation or making a request to a

database or an external service, and expecting a maximum of one result, then we should use Mono.

When we're expecting multiple results from our computation, database, or external service call, then we should use Flux.

To use *Mono* and *Flux*, make sure that you add Project Reactor Core dependency:

```
<dependency>
	<groupId>io.projectreactor</groupId>
	<artifactId>reactor-core</artifactId>
	<version>3.6.5</version>
</dependency>
```

Flux Examples:

```
Flux<Integer> intFlux = Flux.range(1,10).delayElements(Duration.ofSeconds(1));
intFlux.subscribe(System.out::println);
```

```
Flux<String> stringFlux = Flux.just("Apple", "Banana", "Orange",
Mango").delayElements(Duration.ofSeconds(1));
stringFlux.subscribe(System.out::println);
```

```
Flux<String> stringFlux = Flux.just("Apple", "Banana", "Orange", "Mango").log();

stringFlux.subscribe((element) -> {
		System.out.println(element);
	});
```

```
Flux<Integer> flux = Flux.range(1, 10).delayElements(Duration.ofSeconds(1));
flux.filter(i -> i % 2 == 0).map(i -> i * 2)
	.subscribe(System.out::println);
```

```
System.out.println("Press any key to end");
System.in.read();
```

```
Flux<User> userFlux = Flux.just(
				new User(1,"raj",30),
				new User(2,"sam",40),
				new User(3,"tom",60),
				new User(4,"kim",30),
				new User(5,"jim",20)
				);
```

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

```
Flux<Integer> intFlux = Flux.range(1,10).delayElements(Duration.ofSeconds(1))
        .map(e-> {
            if(e==5) throw new RuntimeException("An error occured in the Flux");
            return e;
        });

   intFlux.subscribe((element) -> {
        System.out.println(element);
    }, throwable -> System.out.println(throwable.getMessage()));
```

```
System.out.println("Press a key to end");
System.in.read();
```

Mono Example:

```
Mono<Integer> intMono = Mono.just(50).delayElement(Duration.ofSeconds(1));
intMono.subscribe(System.out::println);
```

```
Mono<?> stringMono = Mono.just("reactive java").log();
             stringMono.subscribe((element) -> {
                    System.out.println(element);
             });
```

Spring WebFlux

Spring WebFlux is a non-blocking, reactive web framework for building reactive, scalable web applications. It is part of the Spring Framework, and it is built on top of Project Reactor, which is a reactive programming library for building asynchronous, non-blocking applications.

Programming models supported by Spring WebFlux:

Spring WebFlux supports two types of programming models:

- The traditional annotation-based model with *@Controller*, *@RestController*, *@RequestMapping*, and other annotations that you have been using in Spring MVC.
- A brand-new Functional style model based on Java 8 lambdas for routing and handling requests.

What Is *R2DBC*? (Reactive Relational Database Connectivity)

- database access in the Java/JVM world remains basically synchronous.
- we can use R2DBC to perform database operations in a reactive way.