

Application Delivery Fundamentals 2.0 B: Java

Vert.x



consulting | technology | outsourcing

Goals



- Reactive Programming
- Reactive Extensions
- Vertx Installation
- Verticles
- Testing
- Vertx web stack
- Vertx Microservices
-

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.

What is reactive programming?



- 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.



- Reactive programming is a paradigm that revolves around the propagation of change.
- In other words, if a program propagates all the changes that modify its data to all the interested parties (users, other programs, components, and subparts), then this program can be called reactive.
- A simple example of this is Microsoft Excel.
- If you set a number in cell A1 and another number in cell 'B1', and set cell 'C1' to SUM(A1, B1); whenever 'A1' or 'B1' changes, 'C1' will be updated to be their sum.

Reactive java programming



- What is the difference between assigning a simple variable `c` to be equal to the sum of the `a` and `b` variables and the reactive sum approach?
- In a normal Java program, when we change `'a'` or `'b'`, we will have to update `'c'` ourselves.
- In other words, the change in the flow of the data represented by `'a'` and `'b'`, is not propagated to `'c'`.
- Here is this illustrated through source code:
 - `int a = 4;`
 - `int b = 5;`
 - `int c = a + b;`
 - `System.out.println(c); // 9`
 - `a = 6;`
 - `System.out.println(c);`
 - `// 9 again, but if 'c' was tracking the changes of 'a' and 'b',`
 - `// it would've been 6 + 5 = 11`



Why should we be reactive?

- While 10-15 years ago it was normal for websites to go through maintenance or to have a slow response time.
- Today everything should be online 24/7 and should respond with lightning speed; if it's slow or down, users would prefer an alternativeservice.
- Today slow means unusable or broken.

Why Reactive java programming



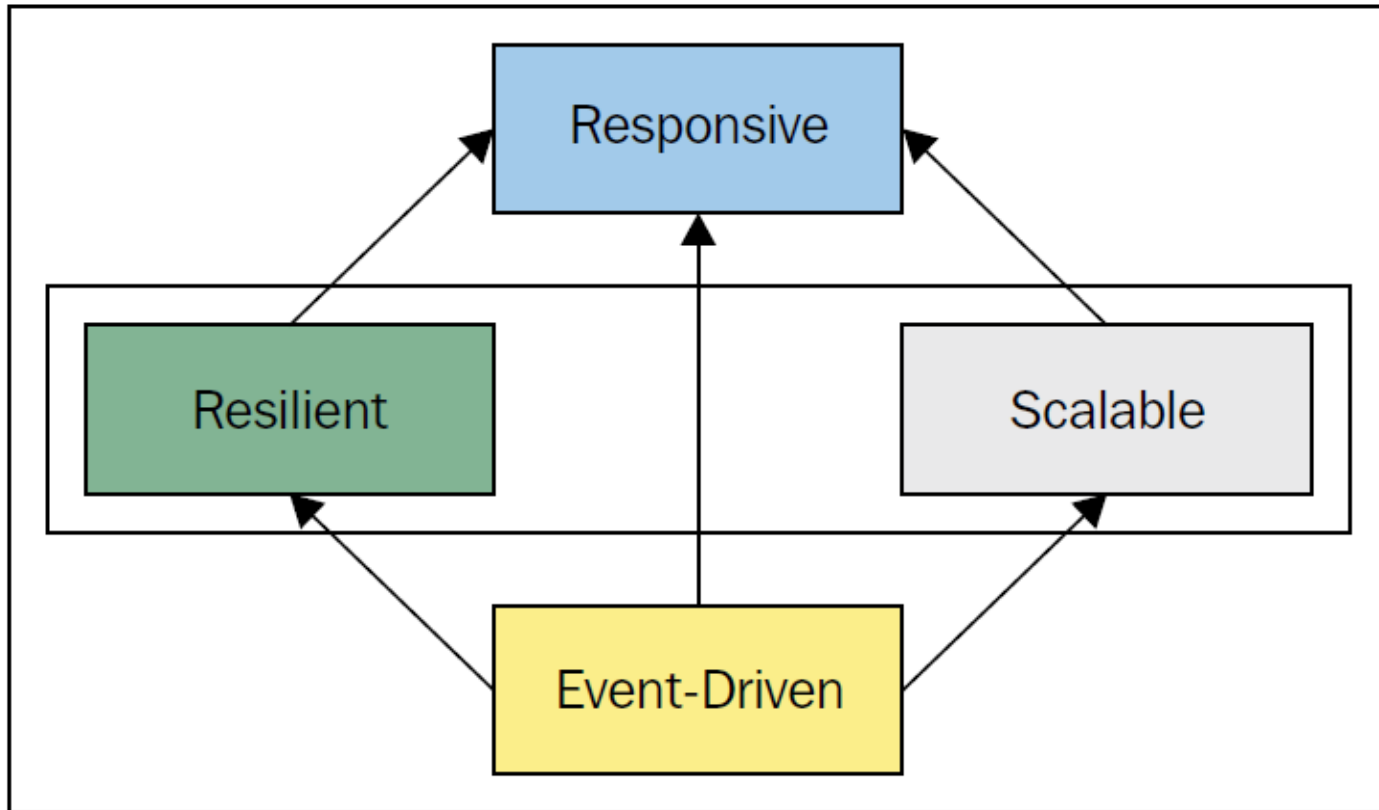
- Evolving changes/demands in the Computing Ecosystem
 - Hardware level
 - Virtualization and cloud strategies
 - Software System Level
 - Software Application Level
 - The impatient human beings!

Why Reactive java programming



- These are the new requirements we have to fulfill if we want our software to be competitive.
- **Modular/dynamic**: This way, we will be able to have 24/7 systems, because modules can go offline and come online without breaking or halting the entire system.
- Additionally, this helps us better structure our applications as they grow larger and manage their code base.
- **Scalable**: This way, we are going to be able to handle a huge amount of data or large numbers of user requests.
- **Fault-tolerant**: This way, the system will appear stable to its users.
- **Responsive**: This means fast and available.

Why Reactive java programming



Creating and Connecting Observables, Observers, and Subjects



- Observable factory methods—just, from, create, and others
- Observers and subscribers
- Hot and cold observables; connectable observables
- What subjects are and when to use them
- Observable creation



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) — and extensively implements the Reactive Streams specification.



Introducing Reactor

- 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.



- Reactive Extensions, also known as ReactiveX, enable us to express the asynchronous events in an application as a set of observable sequences.
- Other applications can subscribe to these observables, in order to receive notifications of events that are occurring.
- A producer can then push these notification events to a consumer as they arrive.
- Alternatively, if a consumer is slow, it can pull notification events according to its own consumption rate.
- The end-to-end system of a producer and its consumers is known as a pipeline.



The ReactiveX API

- **Observables**: Observables represent the core concept of ReactiveX.
- They represent the sequences of emitted items, and they generate events that are propagated to the intended subscribers.



- Types of Observables
 - Observable
 - Flowable
 - Single
 - Maybe
 - Completable



- Observable: emit a stream elements (endlessly)
- Flowable: emit a stream of elements (endlessly, with backpressure)
- Single: emits exactly one element
- Maybe: emits zero or one elements
- Completable: emits a “complete” event, without emitting any data type, just a success/failure

Reactivex API



Type	Description
<code>Flowable<T></code>	Emits 0 or n items and terminates with an success or an error event. Supports backpressure, which allows to control how fast a source emits items.
<code>Observable<T></code>	Emits 0 or n items and terminates with an success or an error event.
<code>Single<T></code>	Emits either a single item or an error event. The reactive version of a method call.
<code>Maybe<T></code>	Succeeds with an item, or no item, or errors. The reactive version of an <code>Optional</code> .
<code>Completable</code>	Either completes with an success or with an error event. It never emits items. The reactive version of a <code>Runnable</code> .

Reactivex API



From / To	Flowable	Observable	Maybe	Single	Completable
Flowable		toObservable()	reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrError() last()/lastOrError() single()/singleOrError() all()/any()/count() (and more...)	ignoreElements()

Reactivex API



Observable	toFlowable()		reduce() elementAt() firstElement() lastElement() singleElement()	scan() elementAt() first()/firstOrErr or() last()/lastOrErr or() single()/singleO rError() all()/any()/coun t() (and more...)	ignoreElements ()
Maybe	toFlowable()	toObservable()		toSingle() sequenceEqual ()	toCompletable()
Single	toFlowable()	toObservable()	toMaybe()		toCompletable()
Completable	toFlowable()	toObservable()	toMaybe()	toSingle() toSingleDefault ()	



The ReactiveX API

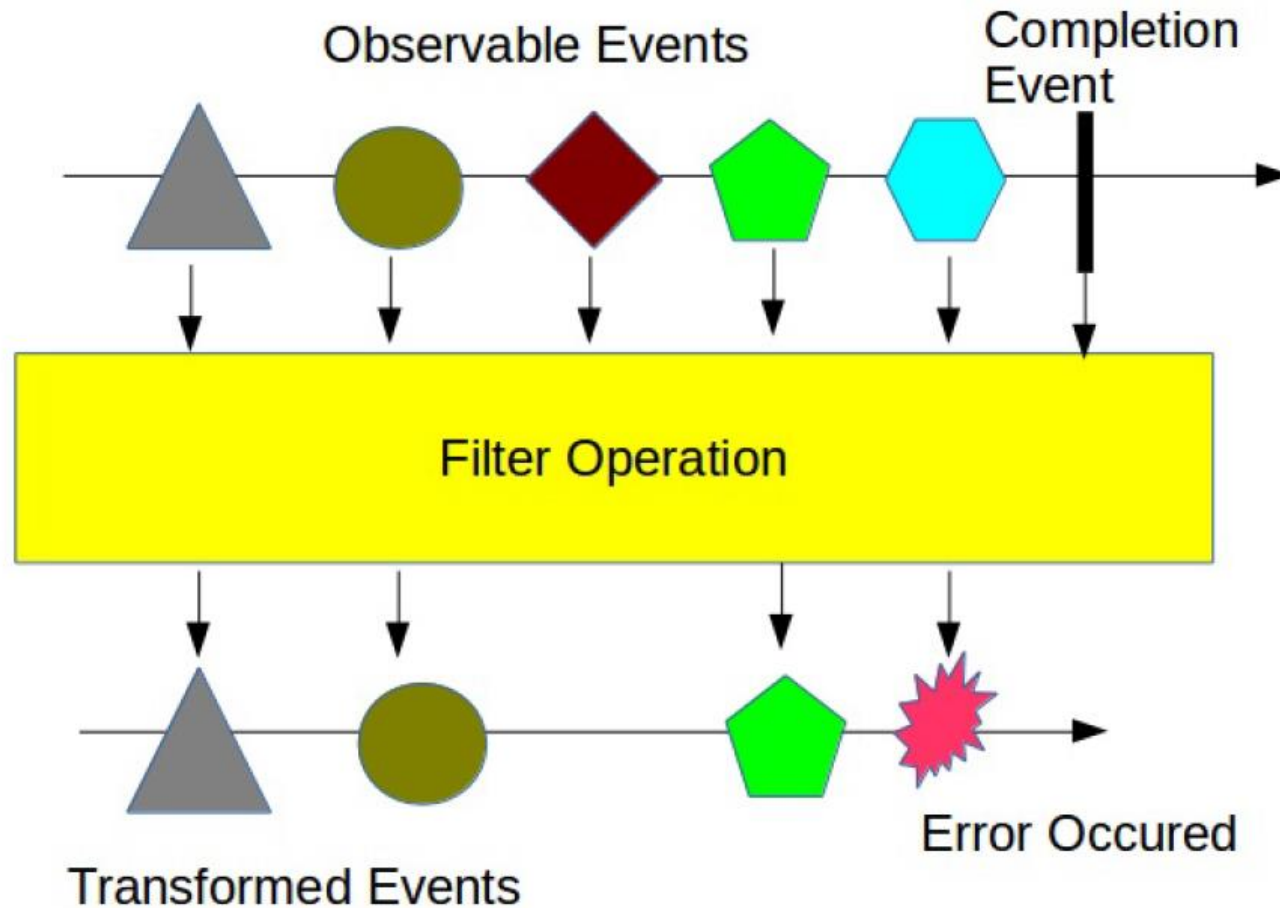
- **Observer:** Any application can express its intent for events published by an observable by creating an observer and subscribing to the respective observable.
- The intent is expressed in terms of the `OnNext`, `OnCompleted`, and `OnError` methods.
- Each observable sends a stream of events, followed by a completion event, which executes these methods.



The ReactiveX API

- **Operators:** Operators enable us to transform, combine, and manipulate the sequences of items emitted by observables.
- The operators on an observable provide a new observable, and thus, they can be tied together.
- They do not work independently on the original observable; instead, they work on the observable generated by the previous operator to generate a new observable.
- The complete operator chain is lazy. It is not evaluated until an observer is subscribed to it.

The ReactiveX API





The ReactiveX API

- ReactiveX provides the architecture design to build reactive applications.
- Individual libraries were built around it in different imperative languages to enable its use.
- These abstractions allow us to build asynchronous, non-blocking applications.



- **Composite streams**

- In software design, composition refers to grouping different entities and treating each group as a single entity.
- Additionally, the single entity exhibits the same behavior as the type it refers to.
- ReactiveX streams are composite in nature.
- They make it possible to combine existing data streams, add transformations, and generate new data streams.



- **Flexible operators**
 - The libraries offer a range of operators for all kinds of functions.
 - Each of the operators accomplishes its tasks similarly to that of a workstation on an assembly line.
 - It takes input from the previous workstation and provides input to the next workstation.
 - These operators offer all kinds of data transformation, stream orchestration, and error handlers.

The ReactiveX API



- **Flexible operators**
 - ReactiveX makes it easier to build event-based applications.
 - However, the framework does not present the ways in which different event-driven applications should interact with each other.
 - In a microservice architecture consisting of numerous event-driven services, the gains made are often offset by the workarounds required for inter-process communication.

Reactive Streams



- The API consists of the following four interfaces:
 - Publisher
 - Subscriber
 - Subscription
 - Process

Reactive Streams Publisher



- **Publisher:** The publisher is responsible for the generation of an unbounded number of asynchronous events and pushing those events to the associated subscribers.



Reactive Streams Subscriber

- **Subscriber:** The subscriber is a consumer of the events published by a publisher.
- The subscriber gets events for subscription, data, completion, and error.
- It can choose to perform actions on any of them.



Reactive Streams Subscriber

- **Subscription**: A subscription is a shared context between the publisher and subscriber, for the purpose of mediating the data exchange between the two.
- The subscription is available with the subscriber only, and enables it to control the flow of events from the publisher.
- The subscription becomes invalid if there is an error or a completion.
- A subscriber can also cancel the subscriptions, in order to close its stream.



Reactive Streams Subscriber

- **Processor**: The processor represents a stage of data processing between a subscriber and a publisher.
- Consequently, it is bound by both of them.
- The processor has to obey the contract between the publisher and the subscriber.
- If there is an error, it must propagate it back to the subscriber



- **Asynchronous processing**
 - Asynchronous execution refers to the ability to execute tasks without having to wait to finish previously executed tasks first.
 - The execution model decouples tasks, so that each of them can be performed simultaneously, utilizing the available hardware.



- **Asynchronous processing**
 - The Reactive Streams API delivers events in an asynchronous manner.
 - A publisher can generate event data in a synchronous blocking manner.
 - On the other hand, each of the on-event handlers can process the events in a synchronously blocking manner.
 - However, event publishing must occur asynchronously.
 - It must not be blocked by the subscriber while processing events.



- **Subscriber Backpressure**
 - A subscriber can control events in its queue to avoid any overruns.
 - It can also request more events if there is additional capacity.
 - Backpressure enforces the publisher to bound the event queues according to the subscriber.
 - Furthermore, a subscriber can ask to receive one element at a time, building a stop-and-wait protocol.
 - It can also ask for multiple elements.
 - On the other hand, a publisher can apply the appropriate buffers to hold non-delivered events, or it can just start to drop events if the production rate is more than the consumption rate.



- **Subscriber Backpressure**
 - It is important to note that the Reactive Streams API is aimed at the flow of events between different systems.
 - Unlike ReactiveX, it does not provide any operators to perform transformations.
 - The API has been adopted as a part of the `java.util.concurrent.Flow` package in JDK 9



David Karnok's classification

- David Karnok, a veteran of various reactive projects like Rxjava and Reactor, has categorized the evolution of reactive libraries into the following generations.
 - Zero
 - First
 - Second
 - Third
 - Fourth
 - Fifth



Zero generation

- The zero generation revolves around the `java.util.observable` interface and the related callbacks.
- It essentially uses the observable design pattern for reactive development.
- It lacks the necessary support of composition, operators, and backpressure.



First generation

- The first generation represents Erik Mejer's attempt to address reactive issues by building Rx.NET.
- This referred to implementations in the form of the IObservable and IObservable interfaces.
- The overall design was synchronous and lacked backpressure.



Second generation

- The first generation deficiencies of backpressure and synchronous handling were handled in the second generation APIs.
- This generation refers to the first implementations of Reactive Extensions, such as RxJava 1.X and Akka.



Third generation

- The third generation refers to the Reactive Streams specification, which enables library implementors to be compatible with each other and compose sequences, cancellations, and backpressure across boundaries.
- It also enables an end user to switch between implementations at their own will.



Fourth generation

- The fourth generation refers to the fact that reactive operators can be combined in an external or internal fashion, leading to performance optimization.
- A fourth generation reactive API looks like a third generation, but internally, the operators have changed significantly to yield intended benefits.
- Reactor 3.0 and RxJava 2.x belong to this generation.

Fifth generation



- The fifth generation refers to a future work, in which there will be a need for bidirectional reactive I/O operations over the streams.



- Reactor is an implementation completed by the Pivotal Open Source team, conforming to the Reactive Streams API.
- The framework enables us to build reactive applications, taking care of backpressure and request handling.



- The library offers the following features.
 - Infinite Data Streams
 - Push Pull Model
 - Concurrency Agnostic
 - Operator vocabulary



Infinite Data Streams

- Reactor offers implementations for generating infinite sequences of data.
- At the same time, it offers an API for publishing a single data entry.
- This is suited to the request-response model.
- Each API offers methods aimed at handling the specific data cardinality.
- Rather than waiting for the entire data collection to arrive, subscribers to each data stream can process items as they arrive.
- This yields optimized data processing, in terms of space and time.
- The memory requirement is limited to a subset of items arriving at the same time, rather than the entire collection.
- In terms of time, results start to arrive as soon as the first element is received, rather than waiting for the entire dataset.

Push Pull Stream



- Reactor is a push-pull system. A fast producer raises events and waits for the slower subscriber to pull them.
- In the case of a slow publisher and a fast subscriber, the subscriber waits for events to be pushed from the producer.
- The Reactive Streams API allows this data flow to be dynamic in nature.
- It only depends on the real-time rate of production and the rate of consumption.



Concurrency Agnostic

- The Reactor execution model is a concurrency agnostic.
- It does not cover how different streams should be processed.
- The library facilitates different execution models, which can be used at a developer's discretion.
- All transformations are thread safe.
- There are various operators that can influence the execution model by combining different synchronous streams.



Operator Vocabulary

- Reactor provides a wide range of operators.
- These operators allow us to select, filter, transform, and combine streams.
- The operations are performed as a workstation in a pipeline.
- They can be combined with each other to build high level, easy-to-reason data pipelines.



Operator Vocabulary

- Reactor has been adopted in Spring Framework 5.0 to provide reactive features.
- The complete project consists of the following sub-projects:
- **Reactor-Core**: This project provides the implementation for the Reactive Streams API. The project is also the foundation for Spring Framework 5.0 Reactive Extensions.
- **Reactor-Extra**: This project complements the Reactor-Core project. It provides the necessary operators to work on top of the Reactive Streams API.
- **Reactor-Tests**: This project contains utilities for test verification.
- **Reactor-IPC**: This project provides non-blocking, inter-process communication. It also provides backpressure-ready network engines for HTTP (including WebSockets), TCP, and UDP. The module can also be used to build microservices.

Getting Reactor



Maven natively supports the BOM concept. First, you need to import the BOM by adding the following snippet to your `pom.xml`:

```
<dependencyManagement> 1
  <dependencies>
    <dependency>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-bom</artifactId>
      <version>2020.0.7</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

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

Getting Reactor

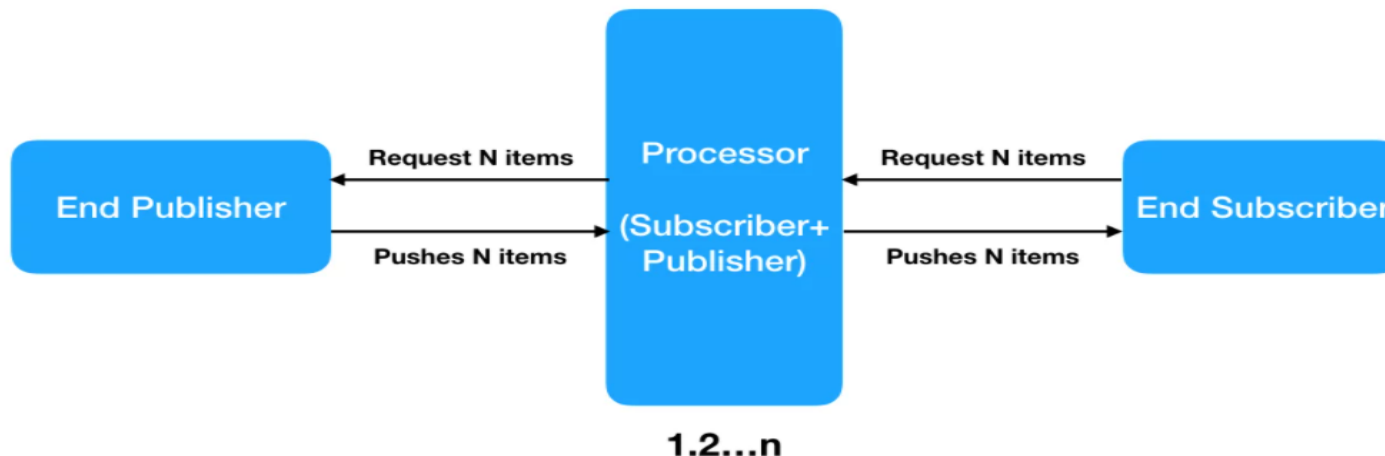


```
<dependencies>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-core</artifactId> 1
  </dependency>
  <dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId> 3
    <scope>test</scope>
  </dependency>
</dependencies>
```

The Publisher and Subscriber APIs in a Reactor



- Publisher Stream Handson
- Subscriber Stream





Reactive Streams API

- It defines four interfaces:
- 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:
- Publisher.java
- `public interface Publisher<T>`
- `{`
- `public void subscribe(Subscriber<? super T> s);`
- `}`



Reactive Streams API

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



Reactive Streams API

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



Reactive Streams API

- Processor: Represents a processing stage consisting of both a Subscriber and a Publisher and obeys the contracts of both.
- Processor.java
- `public interface Processor<T, R> extends Subscriber<T>, Publisher<R>`
- `{`
- `}`

Reactive Streams API



- Two popular implementations of reactive streams are RxJava and **Project Reactor**



Observable Interface

- The `java.util.Observable` interface implements the Observer pattern, which can be co-related here.
- However, all similarities end here. If we look at the `Observable` interface, we have the following methods:

```
public class Observable {  
    void addObserver (Observer o);  
    void deleteObserver (Observer o);  
    void deleteObservers();  
    void notifyObservers();  
    void notifyObserver(int arg);  
    int countObservers();  
    boolean hasChanged();  
}
```



The Observable.just method

Available in: ☒ Flowable, ☒ Observable, ☒ Maybe, ☒ Single, ☐ Completable

- Constructs a reactive type by taking a pre-existing object and emitting that specific object to the downstream consumer upon subscription.
- For every call to the subscribe() method, the whole collection is emitted from the beginning, element by element:

```
listObservable.subscribe(  
    color -> System.out.print(color + "|"),  
    System.out::println,  
    System.out::println  
);  
listObservable.subscribe(color -> System.out.print(color + "/"));  
Observable<Integer> arrayObservable =  
    Observable.from(new Integer[] {3, 5, 8});  
arrayObservable.subscribe(System.out::println);
```



The Observable.from method

- Constructs a sequence from a pre-existing source or generator type.
- This piece of code creates an Observable instance from a List instance.
- When the subscribe method is called on the Observable instance, all of the elements contained in the source list are emitted to the subscribing method

```
List<String> list = Arrays.asList(  
    "blue", "red", "green", "yellow", "orange", "cyan", "purple"  
);  
Observable<String> listObservable = Observable.from(list);  
listObservable.subscribe(System.out::println);
```



The Observable.fromIterable method

Available in: ☒ Flowable, ☒ Observable, ☐ Maybe, ☐ Single, ☐ Completable

- Signals the items from a `java.lang.Iterable` source (such as Lists, Sets or Collections or custom Iterables) and then completes the sequence.

fromIterable example:

```
List<Integer> list = new ArrayList<>(Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8));

Observable<Integer> observable = Observable.fromIterable(list);

observable.subscribe(item -> System.out.println(item), error -> error.printStackTrace(),
    () -> System.out.println("Done"));
```



The Observable.fromArray method

fromArray

Available in: ☒ Flowable, ☒ Observable, ☐ Maybe, ☐ Single, ☐ Completable

Signals the elements of the given array and then completes the sequence.

fromArray example:

```
Integer[] array = new Integer[10];
for (int i = 0; i < array.length; i++) {
    array[i] = i;
}






Observable<Integer> observable = Observable.fromArray(array);

observable.subscribe(item -> System.out.println(item), error -> error.printStackTrace(),
    () -> System.out.println("Done"));
```



The Observable.fromCallable method

fromCallable

Available in:  Flowable,  Observable,  Maybe,  Single,  Completable

When a consumer subscribes, the given `java.util.concurrent.Callable` is invoked and its returned value (or thrown exception) is relayed to that consumer.

fromCallable example:

```
Callable<String> callable = () -> {  
    System.out.println("Hello World!");  
    return "Hello World!";  
}  
  
Observable<String> observable = Observable.fromCallable(callable);  
  
observable.subscribe(item -> System.out.println(item), error -> error.printStackTrace(),  
    () -> System.out.println("Done"));
```

Remark: In `Completable`, the actual returned value is ignored and the `Completable` simply completes.



The Observable. fromAction method

Available in: ☐ Flowable, ☐ Observable, ☒ Maybe, ☐ Single, ☒ Completable

When a consumer subscribes, the given `io.reactivex.function.Action` is invoked and the consumer completes or receives the exception the `Action` threw.

fromAction example:

```
Action action = () -> System.out.println("Hello World!");

Completable completable = Completable.fromAction(action);

completable.subscribe(() -> System.out.println("Done"), error -> error.printStackTrace());
```

Note: the difference between `fromAction` and `fromRunnable` is that the `Action` interface allows throwing a checked exception while the `java.lang.Runnable` does not.



The Observable.fromRunnable method

Available in: ☐ Flowable, ☐ Observable, ☒ Maybe, ☐ Single, ☒ Completable

When a consumer subscribes, the given `io.reactivex.function.Action` is invoked and the consumer completes or receives the exception the `Action` threw.

fromRunnable example:

```
Runnable runnable = () -> System.out.println("Hello World!");






Completable completable = Completable.fromRunnable(runnable);

completable.subscribe(() -> System.out.println("Done"), error -> error.printStackTrace());
```

Note: the difference between `fromAction` and `fromRunnable` is that the `Action` interface allows throwing a checked exception while the `java.lang.Runnable` does not.



The Observable.fromFuture method

Available in:  Flowable,  Observable,  Maybe,  Single,  Completable

Given a pre-existing, already running or already completed `java.util.concurrent.Future`, wait for the `Future` to complete normally or with an exception in a blocking fashion and relay the produced value or exception to the consumers.

fromFuture example:

```
ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();

Future<String> future = executor.schedule(() -> "Hello world!", 1, TimeUnit.SECONDS);

Observable<String> observable = Observable.fromFuture(future);

observable.subscribe(
    item -> System.out.println(item),
    error -> error.printStackTrace(),
    () -> System.out.println("Done"));

executor.shutdown();
```



The Observable.from{reactive type} method

Wraps or converts another reactive type to the target reactive type.

The following combinations are available in the various reactive types with the following signature pattern:

```
targetType.from{sourceType}()
```

Available in:

targetType \ sourceType	Publisher	Observable	Maybe	Single	Completable
Flowable	✓				
Observable	✓				
Maybe				✓	✓
Single	✓	✓			
Completable	✓	✓	✓	✓	

*Note: not all possible conversion is implemented via the `from{reactive type}` method families. Check out the `to{reactive type}` method families for further conversion possibilities.

Observable.Create



Available in: Flowable, Observable, Maybe, Single, Completable

ReactiveX documentation: <http://reactivex.io/documentation/operators/create.html>

Construct a **safe** reactive type instance which when subscribed to by a consumer, runs an user-provided function and provides a type-specific `Emitter` for this function to generate the signal(s) the designated business logic requires. This method allows bridging the non-reactive, usually listener/callback-style world, with the reactive world.

create example:

```
ScheduledExecutorService executor = Executors.newSingleThreadedScheduledExecutor();

ObservableOnSubscribe<String> handler = emitter -> {

    Future<Object> future = executor.schedule(() -> {
        emitter.onNext("Hello");
        emitter.onNext("World");
        emitter.onComplete();
        return null;
    }, 1, TimeUnit.SECONDS);

    emitter.setCancellable(() -> future.cancel(false));
};

Observable<String> observable = Observable.create(handler);

observable.subscribe(item -> System.out.println(item), error -> error.printStackTrace(),
    () -> System.out.println("Done"));

Thread.sleep(2000);
executor.shutdown();
```

Note: `Flowable.create()` must also specify the backpressure behavior to be applied when the user-provided function generates more items than the downstream consumer has requested.



Hot and Cold Observables

- Observables are broadly categorized as Hot or Cold, depending on their emission behavior.
- A Cold Observable is one which starts emitting upon request(subscription), whereas a Hot Observable is one that emits regardless of subscriptions.
- /* Demonstration of a Cold Observable */
- `Observable<Long> cold = Observable.interval(500, TimeUnit.MILLISECONDS); // emits a long every 500 milliseconds`
- `cold.subscribe(l -> System.out.println("sub1, " + l)); // subscriber1`
- `Thread.sleep(1000); // interval between the two subscribes`
- `cold.subscribe(l -> System.out.println("sub2, " + l)); // subscriber2`



Hot and Cold Observables

- Hot observables emit values independent of individual subscriptions.
- They have their own timeline and events occur whether someone is listening or not.



BackPressure

- BackPressure Drop
- BackPressure Latest
- BackPressure Missing
- BackPressure Error
- BackPressure Buffer

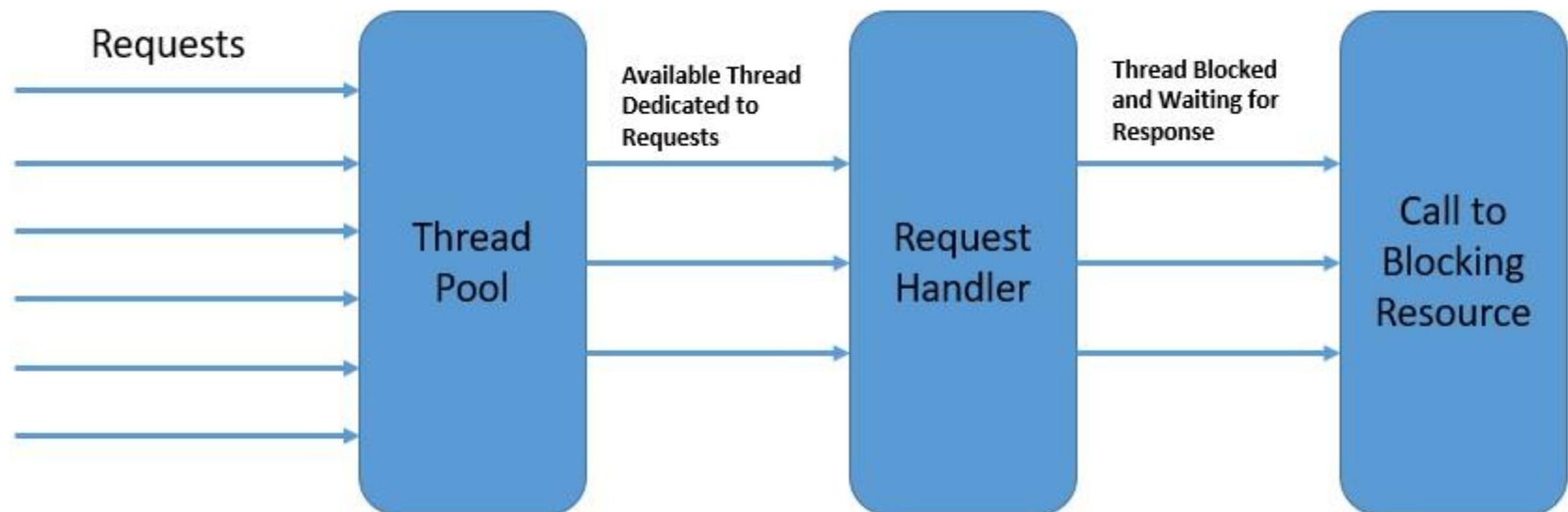


Blocking Request Processing

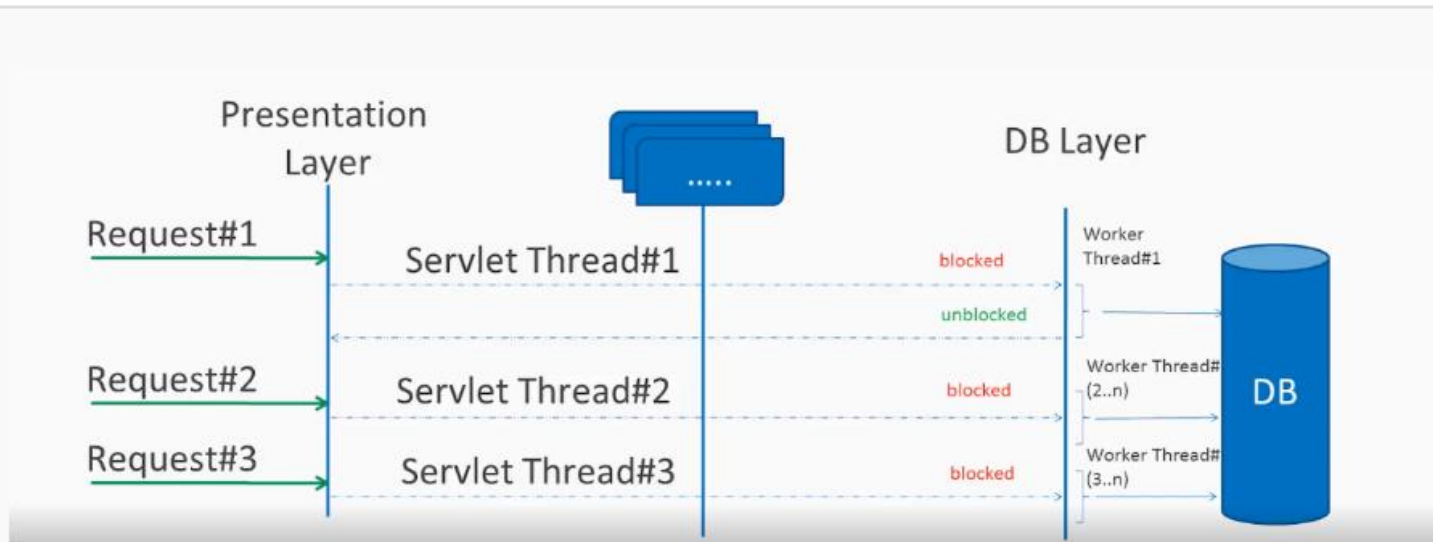
- In traditional MVC applications, when a request come to server, a servlet thread is created.
- It delegates the request to worker threads for I/O operations such as database access etc.
- During the time worker threads are busy, servlet thread (request thread) remain in waiting status and thus it is blocked.
- It is also called synchronous request.
- As server can have some finite number of request threads, it limits the server capability to process that number of requests at maximum server load.
- It may hamper the performance and limit the full utilization of server capability.



Blocking request processing



Blocking request processing



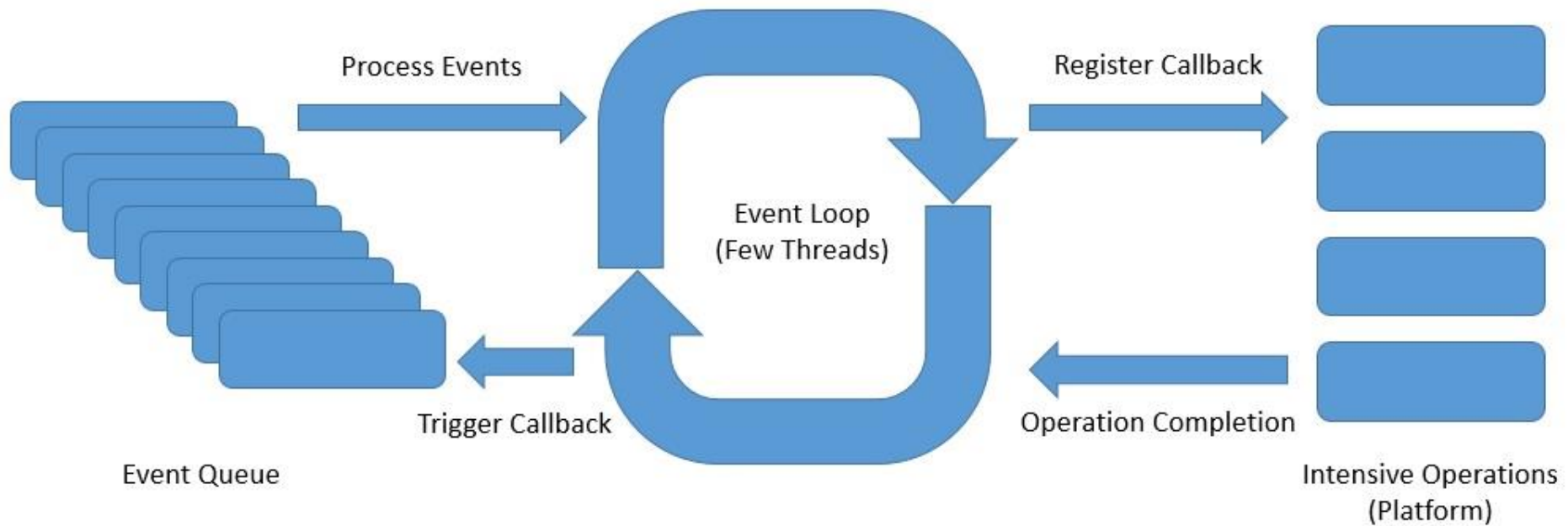
Blocking request processing



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.

Event Loop asynchronous Non blocking request processing





Event Loop

- The event loop runs continuously in a single thread, although we can have as many event loops as the number of available cores
- The event loop process the events from an event queue sequentially and returns immediately after registering the callback with the platform
- The platform can trigger the completion of an operation like a database call or an external service invocation
- The event loop can trigger the callback on the operation completion notification and send back the result to the original caller

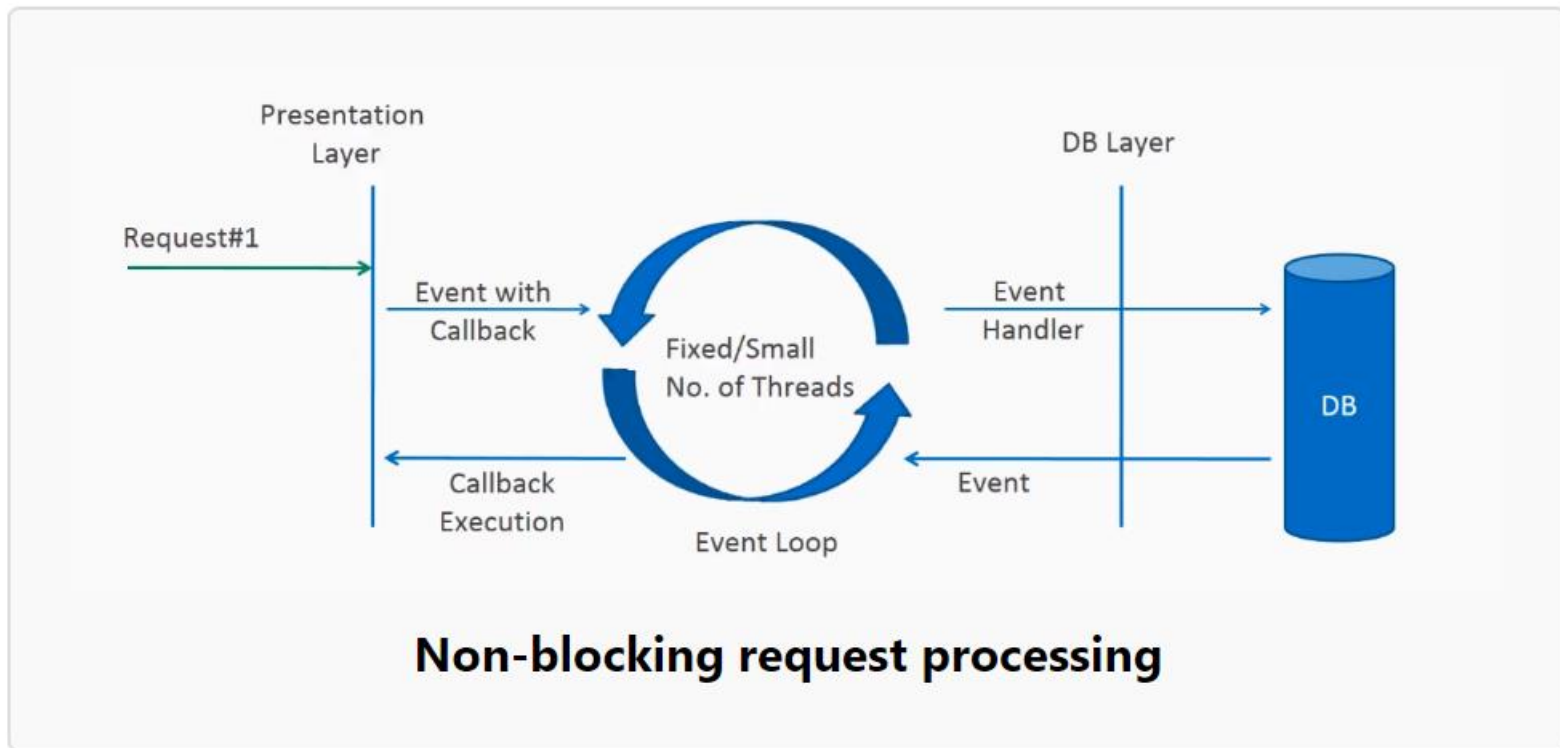
Event Loop



- The event loop model is implemented in a number of platforms including Node.js, Netty, and Ngnix.
- They offer much better scalability than traditional platforms like Apache HTTP Server, Tomcat, or JBoss.



Non-blocking request processing

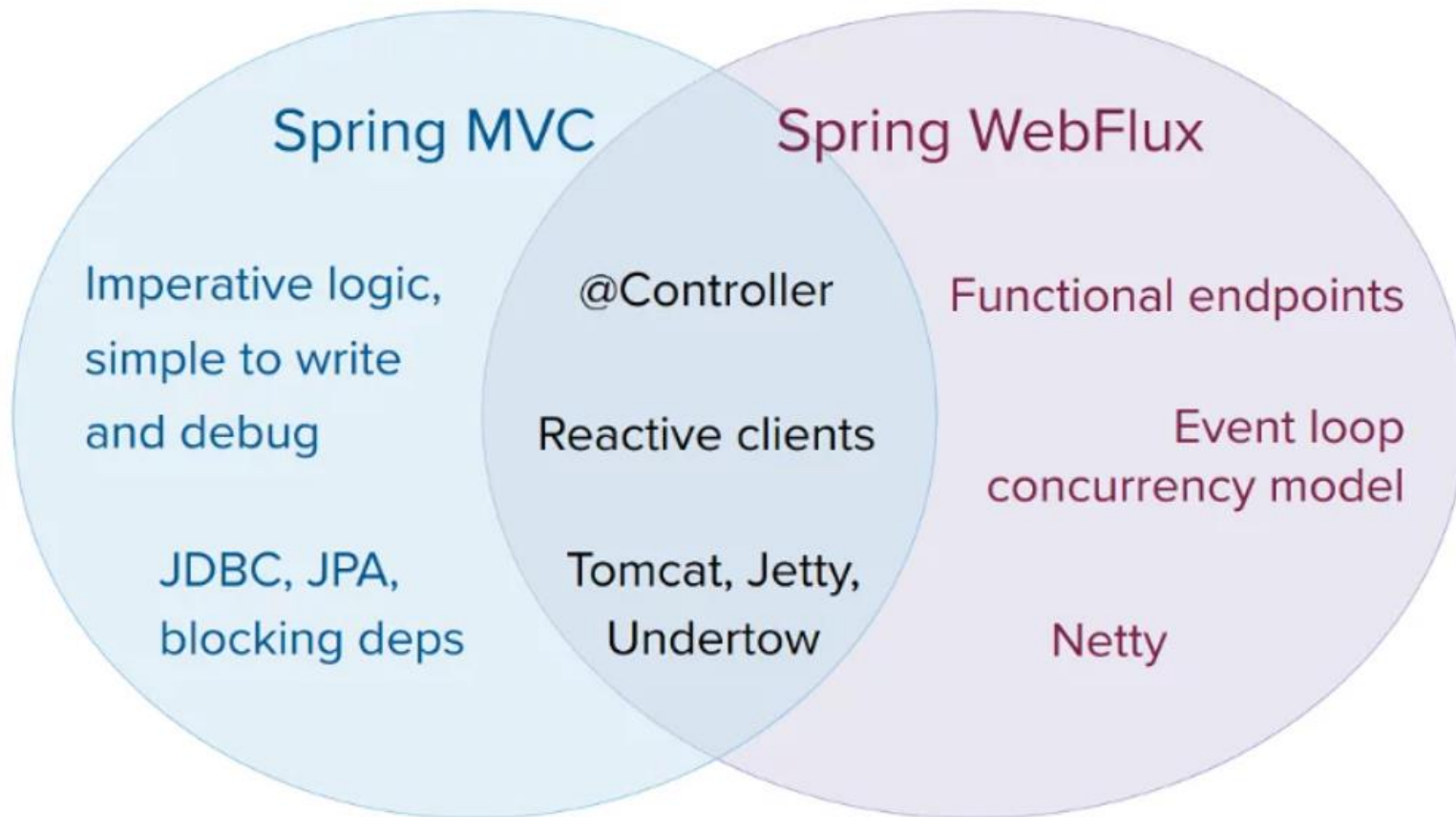




Non Blocking Request Processing

- Non-blocking nature of threads helps in scaling the performance of the application.
- Small number of threads means less memory utilization and also less context switching as well.

Spring Web Flux





- Spring WebFlux is built on Project Reactor.
- Project Reactor is the implementation of Reactive Streams specification. Reactor provides two types:
 - Mono: implements Publisher and returns 0 or 1 elements
 - Flux: implements Publisher and returns N elements.



- Spring WebFlux is parallel version of Spring MVC and supports fully non-blocking reactive streams.
- It supports the back pressure concept and uses Netty as inbuilt server to run reactive applications.
- Spring webflux uses project reactor as reactive library.
- Reactor is a Reactive Streams library and therefore, all of its operators support non-blocking back pressure.
- It is developed in close collaboration with Spring.



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.

Spring WebFlux



Spring MVC

Servlet Stack

Blocking

Supported Servers

Tomcat

Jetty

Undertow

Servlet Container

@Controller, @RequestMapping

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Spring WebFlux

Reactive Stack

Non-blocking

Supported Servers

Tomcat

Jetty

Netty

Undertow

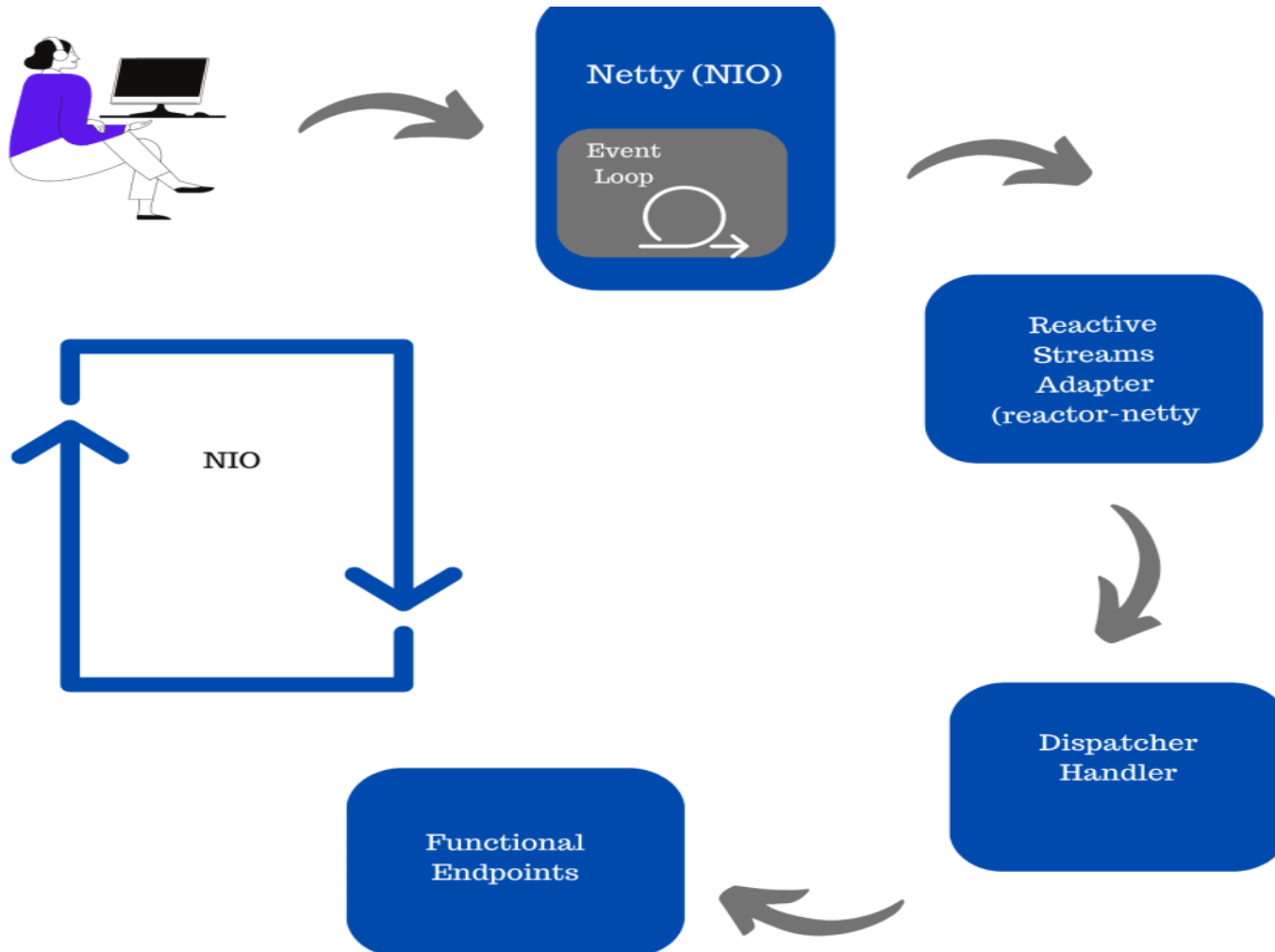
Servlet 3.1 Container

@Controller, @RequestMapping

Handler & Router Function

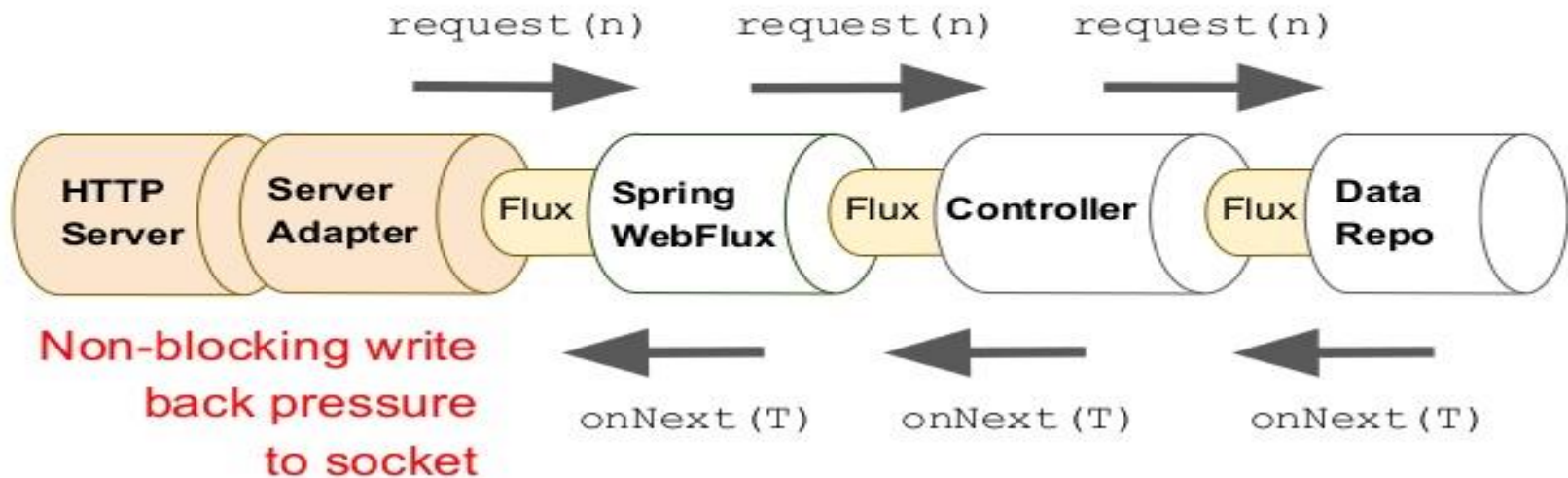
```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Spring WebFlux





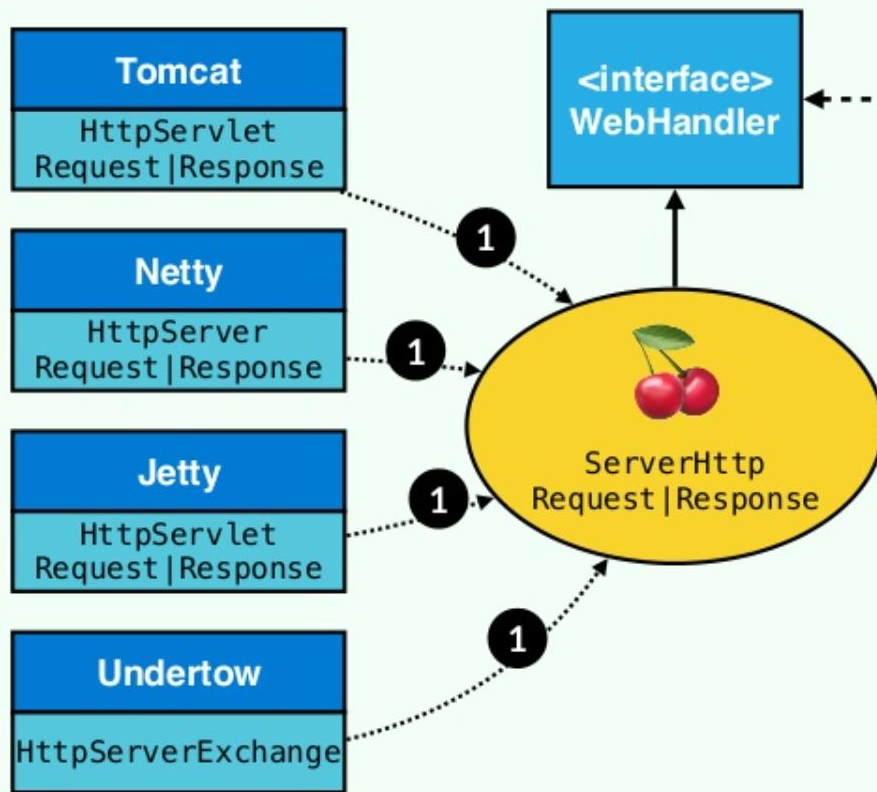
Response streaming on **reactive** stack



Spring WebFlux

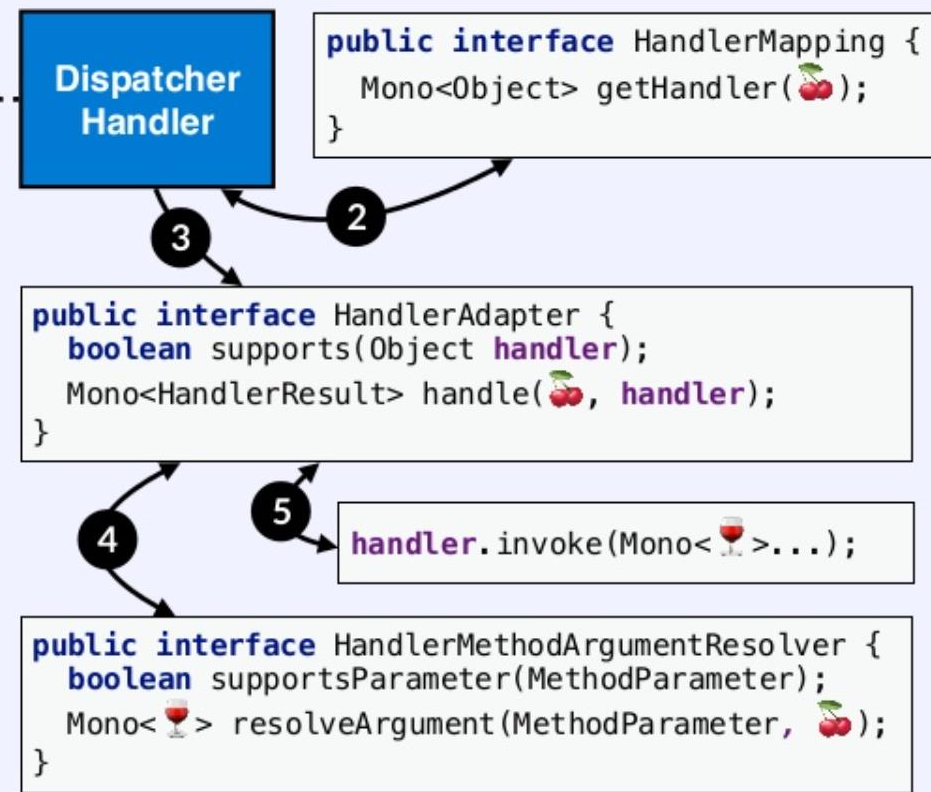


spring-web



spring-webflux

Clip slide





Flux against the world

	composable	lazy	reusable	async	cached	push	back-pressure	operator fusion
CompletableFuture	✓	✗	✓	✓	✓	✓	✗	✗
Stream	✓	✓	✗	✗	✗	✗	✗	✗
Optional	✓	✗	✓	✗	✓	✗	✗	✗
Observable (RxJava 1)	✓	✓	✓	✓	✓ *.cache()	✓	✓ *.onBackpressure*()	✗
Observable (RxJava 2)	✓	✓	✓	✓	✓ *.cache()	✓	✗	✓
Flowable (RxJava 2)	✓	✓	✓	✓	✓ *.cache()	✓	✓	✓
Flux	✓	✓	✓	✓	✓ *.cache()	✓	✓	✓

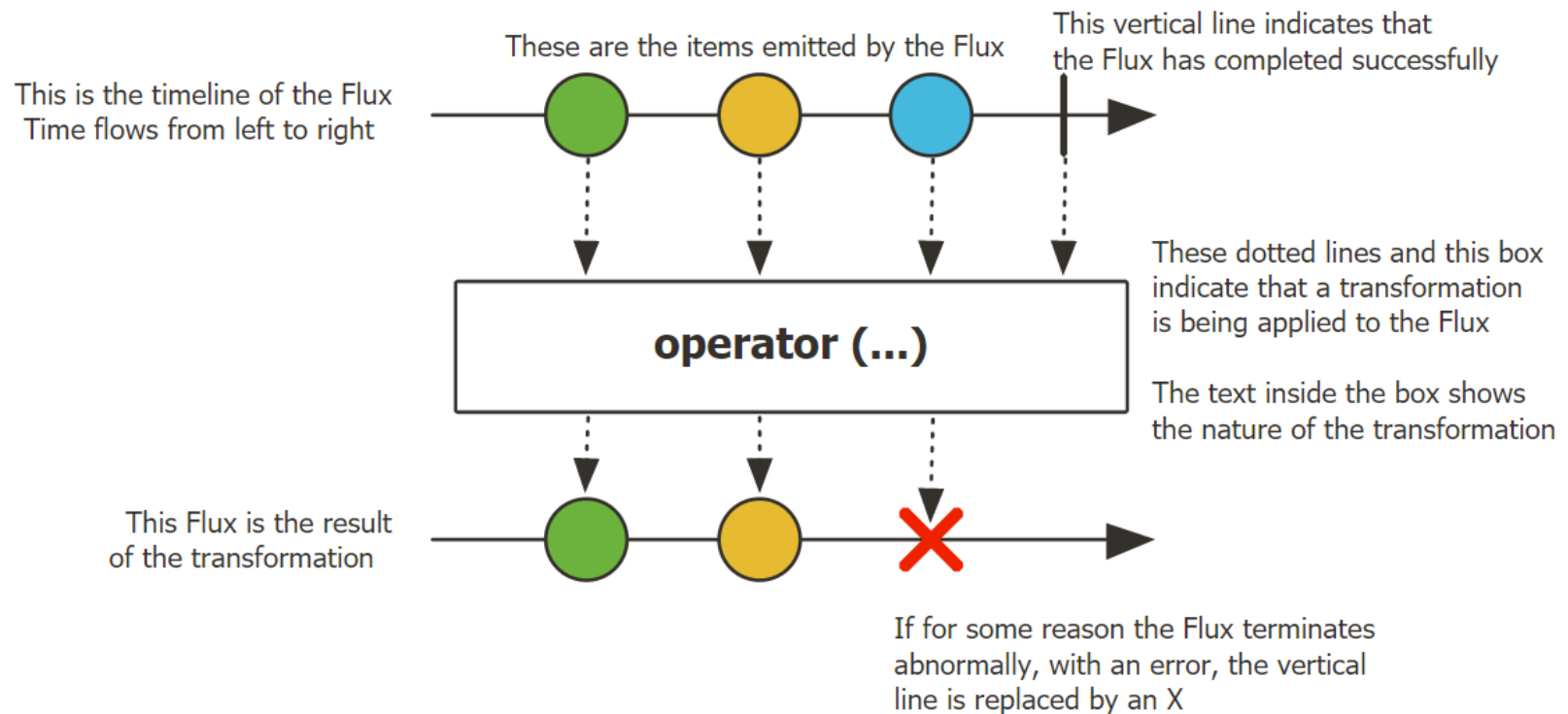


- A $\text{Flux}\langle T \rangle$ is a standard $\text{Publisher}\langle T \rangle$ that represents an asynchronous sequence of 0 to N emitted items, optionally terminated by either a completion signal or an error.
- As in the Reactive Streams spec, these three types of signal translate to calls to a downstream Subscriber's `onNext`, `onComplete`, and `onError` methods.

Flux, an Asynchronous Sequence of 0-N Items



The following image shows how a Flux transforms items:





The Flux.from methods

- The From methods can be used to generate a Flux from various sources, such as arrays, collections, and so on.
- In this case, all of the values are identified as multi-valued datasets beforehand.
- The generated Flux publishes the value events for each value in the original dataset, followed by a completion event.



The Flux.from methods

- The offered methods have the following variants:
 - Flux.fromArray: This is used to build a stream from an array of a type.
 - Flux.fromIterable: This is used to build a stream from collections.
 - All collections are of the Iterable<T> type, which can be passed to this to generate the intended stream.



The Flux Utility methods

- Flux offers methods to generate infinite streams and empty streams, or to convert an existing Reactive Stream publisher to Flux.
- These methods are required to generate streams that can be combined with other streams, using the available operators.



The Flux Utility methods

- Available operators, as follows:
 - `Flux.empty`: This method generates an empty stream with no values and only completion.
 - `Flux.error`: This method generates an error stream with no values and only specified errors.
 - `Flux.never`: This method generates a stream with no events at all. It does not generate events of any type.
 - `Flux.from`: This method takes an existing reactive publisher and generates a Flux from it.
 - `Flux.defer`: This method is used to build a lazy reactive publisher. The method takes a Java 8 supplier to instantiate a subscription-specific Reactive Stream publisher. The publisher instance is only generated when a subscriber makes a subscription to the Flux.



The Flux.generate method

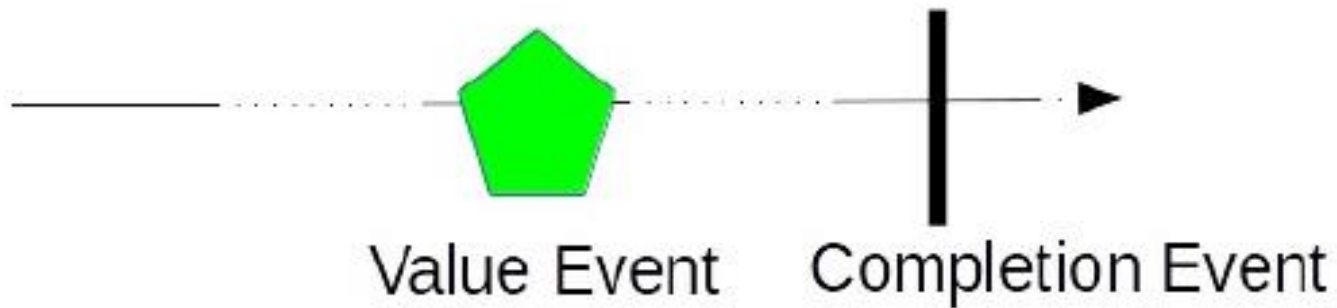
- Flux supports programmatic event generation.
- This is an advanced usage method of the API, and it involves some more components.



- Mono is capable of generating a maximum of one event.
- This is a specific use case for Flux, capable of handling one response model, such as data aggregation, HTTP requestresponse, service invocation response, and so on.
- It is important to note that a Mono emits the following three events:
 - Value refers to the single value generated by the publisher
 - Completion refers to a normal termination of the stream
 - Error refers to an erroneous termination of the stream
 - Since Mono is a subset of Flux, it supports a subset of Flux operators.



Mono Events Timeline



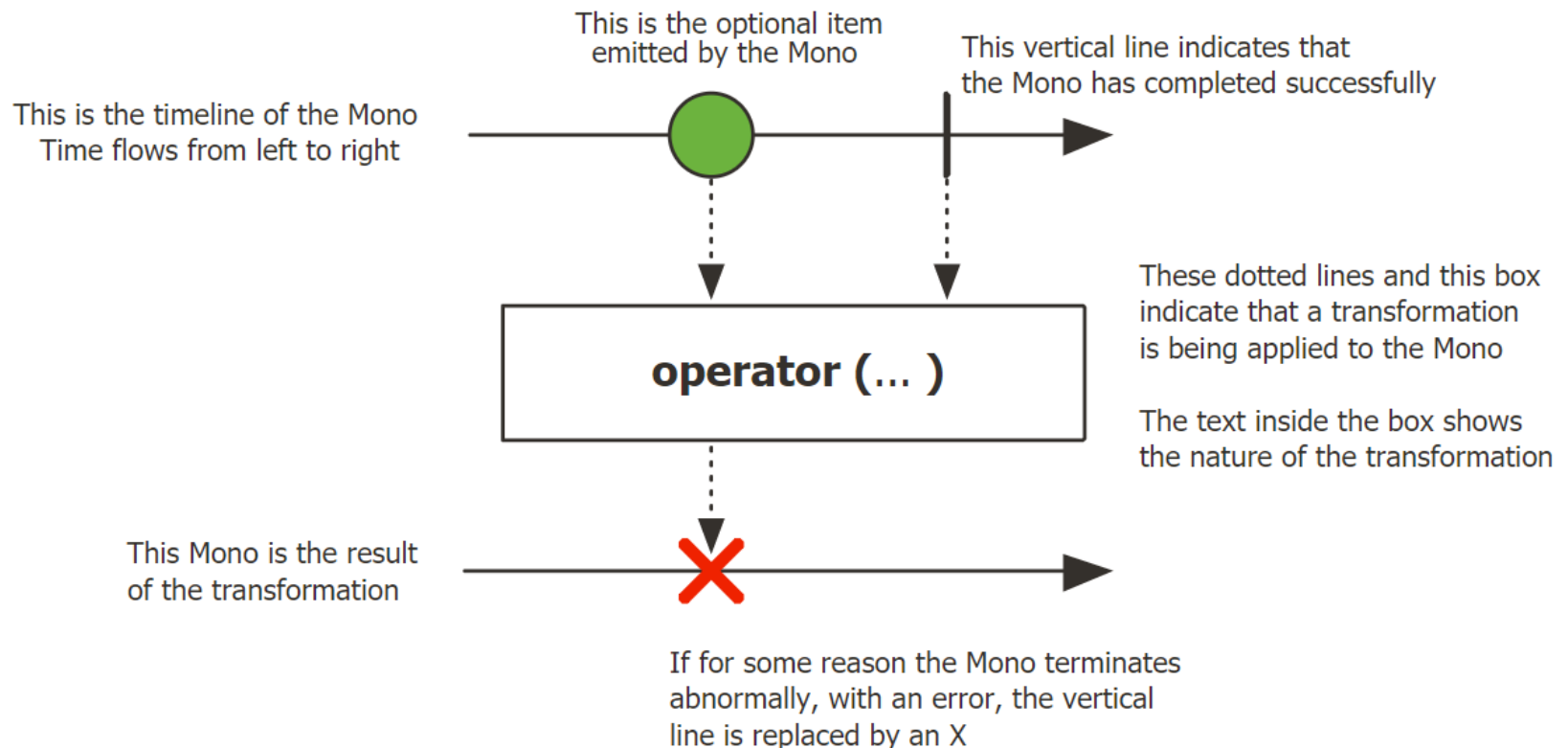


- A `Mono<T>` is a specialized `Publisher<T>` that emits at most one item via the `onNext` signal then terminates with an `onComplete` signal (successful `Mono`, with or without value), or only emits a single `onError` signal (failed `Mono`).
- Most `Mono` implementations are expected to immediately call `onComplete` on their `Subscriber` after having called `onNext`.
- `Mono.never()` is an outlier: it doesn't emit any signal, which is not technically forbidden although not terribly useful outside of tests.
- On the other hand, a combination of `onNext` and `onError` is explicitly forbidden.
- `Mono` 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`.



Mono, an Asynchronous 0-1 Result

The following image shows how a **Mono** transforms an item:





The Mono.just method

- The Mono.just method is the simplest method for Mono generation.
- It takes a single value and generates a finite Mono stream from it.
- A completion event is published after publishing the specified value:
 - `Mono.just("Red");`
 - `Mono.justOrEmpty(value);`
 - `Mono.justOrEmpty(Optional.empty());`



The Mono.from method

- The From methods are used to build a Flux when the value can be determined from an existing source.
- Unlike the Flux methods, where the sources are multivalued, the sources for Mono are single-valued.
- These methods are offered in the following variants:
 - fromCallable: This method generates Mono with one value, followed by the completion event. If multi-valued datasets, like arrays or collections, are returned from Callable, then the complete dataset is pushed as an object in the single event.



The Mono.from method

- fromFuture: This method generates Mono with one value, followed by the completion event.
- fromSupplier: This method generates Mono with one value, followed by the completion event.
- fromRunnable: This method generates Mono with no value and only a completion event.
- This can be explained by using the following code:
- `Mono.fromSupplier(() -> 1);`
- `Mono.fromCallable(() -> new String[]{"color"}).subscribe(t -> System.out.println("received " + Mono.fromRunnable(() -> System.out.println(" ")).subscribe(t -> System.out.println("")))`



The Mono Utility methods

- Mono offers methods to generate empty/error streams or to convert an existing Reactive Stream publisher to Mono.
- These methods are required to generate streams that can be combined with others by using the available operators, as follows:
- `Mono.empty`: Generates a stream with no value and only a completion.
- `Mono.error`: Generates a stream with no value and only a specified error.
- `Mono.never`: Generates a stream with no events at all. It does not generate an event of any type.
- `Mono.from`: Generates a Mono stream from an existing stream publisher. `Mono.defer`: This method is used to build a lazy reactive publisher. It also takes a Java 8 supplier to instantiate a subscription-specific Reactive Stream publisher. The publisher instance is only generated when a subscriber makes a subscription to the Mono.



- In addition to the Flux.create methods, there is a Mono.create method.
- This method provides a MonoSink, which can be used to generate a value, completion, or error event. Unlike the Flux methods, where we are generating N events, if we generate more events in Mono, they are dropped.
- There is also no handling for backpressure, as there is only one event.



- Vert.x is a toolkit or platform for implementing reactive applications on the Java Virtual Machine (JVM).
- An open-source polyglot platform or toolkit is referred to as Vert.x in Java.
- We can say that it is an alternative to the JEE.
- It comes with a different approach in the market to solve the issues such as developing networked and highly concurrent applications.
- It shares common functionalities among all the supported languages like Java, Ruby, Kotlin, Scala, and JavaScript.



- Vert.x is a toolkit, so we can embed it into our standalone Java application.
- We can use it by instantiating the object of Vert.x and calling the method on it.
- In toolkit mode, our code control vert.x.
- It also acts as a platform, so we can set up it using the command line and tell the components to it which want to run.



Metrics & Ops.

Hawkular

Shell

Dropwizard

Bridges

TCP bridge

SockJS bridge

Integration

Stomp

Apache Camel

RabbitMQ

AMQP

Language

Groovy

JavaScript

Ceylon

Ruby

Data Access

MySQL / Postgres

Mongo

JDBC

Redis

Cluster Mngt.

Hazelcast

Ignite

Zookeeper

Web Applications

Templating

Auth and Oauth

Web

Microservice Toolbox

Service Discovery

Circuit Breaker

Vert.x Core

HTTP 1 & 2

UDP

EventBus

Shared data

TCP



Vert.x is Reactive

- Vert.x calls itself a reactive toolkit.
- Reactive applications consists of components that send messages or events to each other.
- This is quite a different internal design than Java EE.
- This internal design makes Vert.x suitable for different types of applications than Java EE (for instance chat and game servers).
- In fact, I would risk the bold claim that Vert.x is suitable for more types of applications than Java EE.



Vert.x is Polyglot

- Vert.x is polyglot meaning you can implement the components you want Vert.x to execute (called "Verticles") in many different languages.
- Vert.x can execute Java, JavaScript, Ruby, Python and Groovy. Support for Scala and Clojure should be arriving soon (it was originally planned for v. 3.0 but I am not sure if they got it in).
- Vert.x can even deploy verticles written in different languages into the same application.
- This gives you the freedom to choose the most suitable language for each job.
- Some tasks might be easier to implement using a functional language like Scala.
- Others easier in a more traditionally imperative language like Java.



Core Vert.x concepts

- what a verticle is, and
- how the event bus allows verticles to communicate.



Threading and programming models

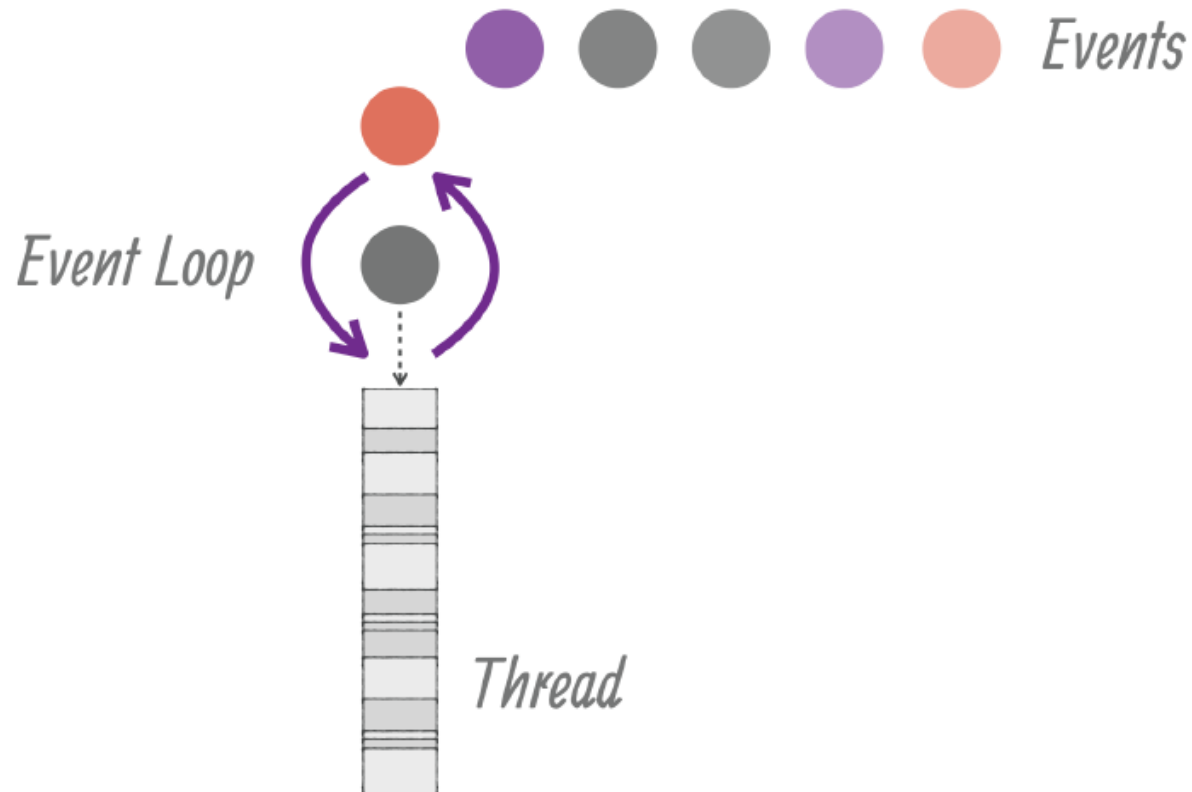
- Many networking libraries and frameworks rely on a simple threading strategy: each network client is being assigned a thread upon connection, and this thread deals with the client until it disconnects.
- This is the case with Servlet or networking code written with the `java.io` and `java.net` packages.
- While this "synchronous I/O" threading model has the advantage of remaining simple to comprehend, it hurts scalability with too many concurrent connections as system threads are not cheap.
- Under heavy loads an operating system kernel spends significant time just on thread scheduling management.



Threading and programming models

- In such cases we need to move to "asynchronous I/O" for which Vert.x provides a solid foundation.
- The unit of deployment in Vert.x is called a Verticle.
- A verticle processes incoming events over an event-loop, where events can be anything like receiving network buffers, timing events, or messages sent by other verticles.
- Event-loops are typical in asynchronous programming models:

Event Loop





Event Loop

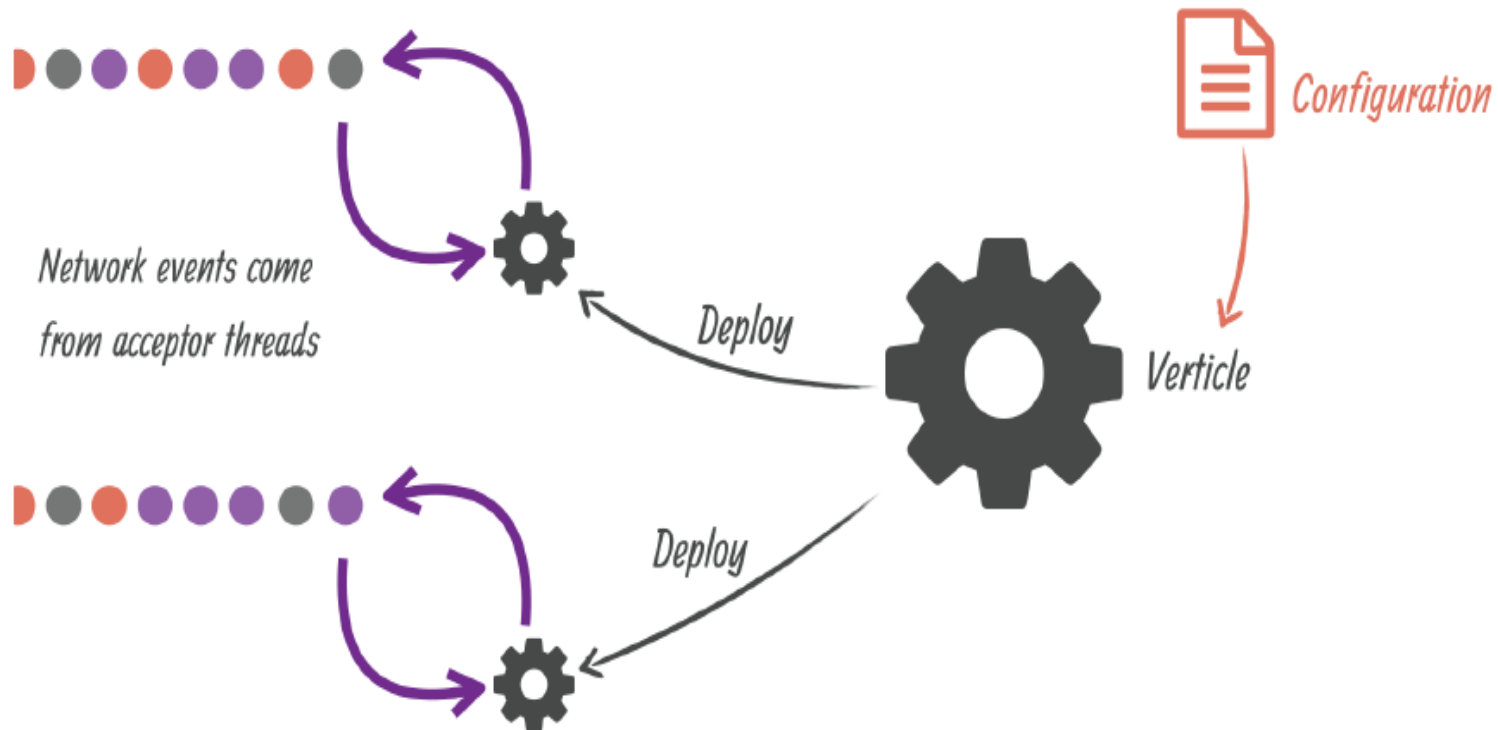
- Each event shall be processed in a reasonable amount of time to not block the event loop.
- This means that thread blocking operations shall not be performed while executed on the event loop, exactly like processing events in a graphical user interface (e.g., freezing a Java / Swing interface by doing a slow network request).
- Vert.x offers mechanisms to deal with blocking operations outside of the event loop.
- In any case Vert.x emits warnings in logs when the event loop has been processing an event for too long, which is also configurable to match application-specific requirements (e.g., when working on slower IoT ARM boards).

Event Loop



- Every event loop is attached to a thread.
- By default Vert.x attaches 2 event loops per CPU core thread.
- The direct consequence is that a regular verticle always processes events on the same thread, so there is no need to use thread coordination mechanisms to manipulate a verticle state (e.g, Java class fields).
- A verticle can be passed some configuration (e.g., credentials, network addresses, etc) and a vertical can be deployed several times:

Event Loop





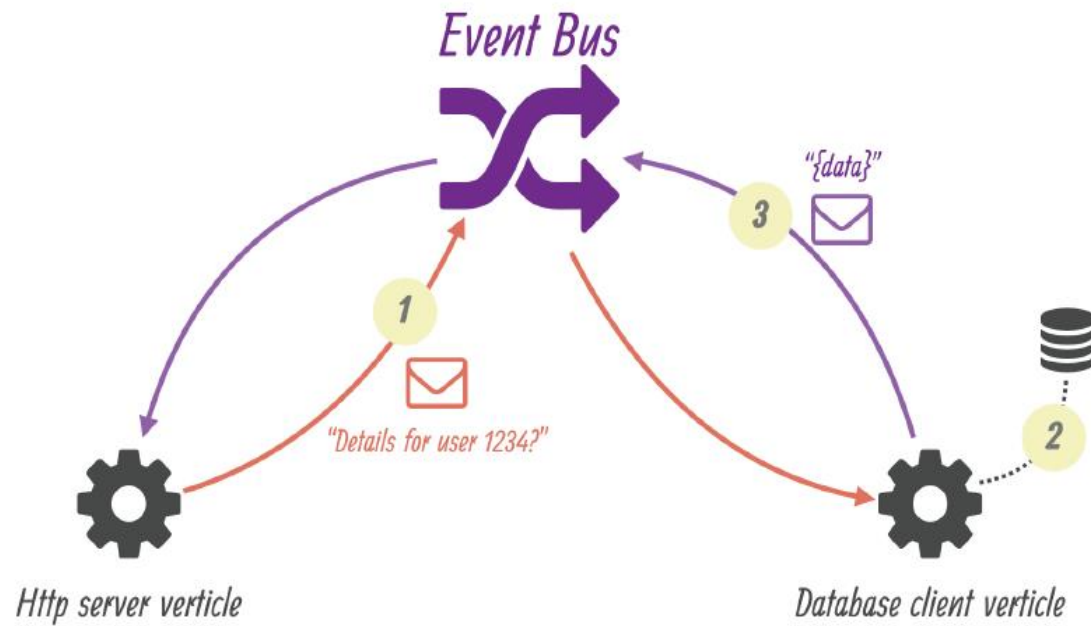
Event Loop

- Incoming network data are being received from accepting threads then passed as events to the corresponding verticles.
- When a verticle opens a network server and is deployed more than once, then the events are being distributed to the verticle instances in a round-robin fashion.
- It is very useful for maximizing CPU usage with lots of concurrent networked requests.
- Finally, verticles have a simple start / stop life-cycle, and verticles can deploy other verticles.



- Verticles form technical units of deployments of code in Vert.x.
- The Vert.x event bus is the main tool for different verticles to communicate through asynchronous message passing.
- For instance suppose that we have a verticle for dealing with HTTP requests, and a verticle for managing access to the database.
- The event bus allows the HTTP verticle to send a request to the database vertical that performs a SQL query, and responds back to the HTTP verticle

Event Bus



Event Bus



- The event-bus allows passing any kind of data, although JSON is the preferred exchange format since it allows verticles written in different languages to communicate.
- In general JSON is a popular general-purpose semi-structured data marshaling text format.
- Message can be sent to destinations which are free-form strings.
- The event bus supports the following communication patterns:
 - 1. point-to-point messaging, and
 - 2. request-response messaging and
 - 3. publish / subscribe for broadcasting messages



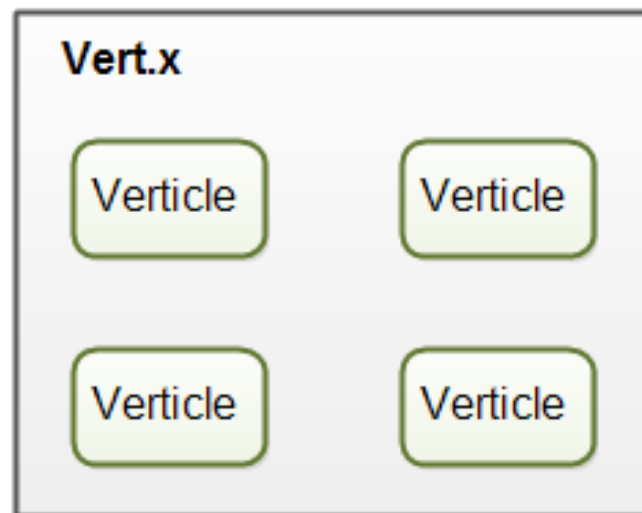
Event Bus

- The event bus allows verticles to transparently communicate not just within the same JVM process:
- when network clustering is activated, the event bus is distributed so that messages can be sent to verticles running on other application nodes.
- The event-bus can be accessed through a simple TCP protocol for third-party applications to communicate.
- The event-bus can also be exposed over general-purpose messaging bridges (e.g, AMQP, Stomp), a SockJS bridge.
- It allows web applications to seamlessly communicate over the event bus from JavaScript running in the browser by receiving and publishing messages just like any vertical would do.



Verticles

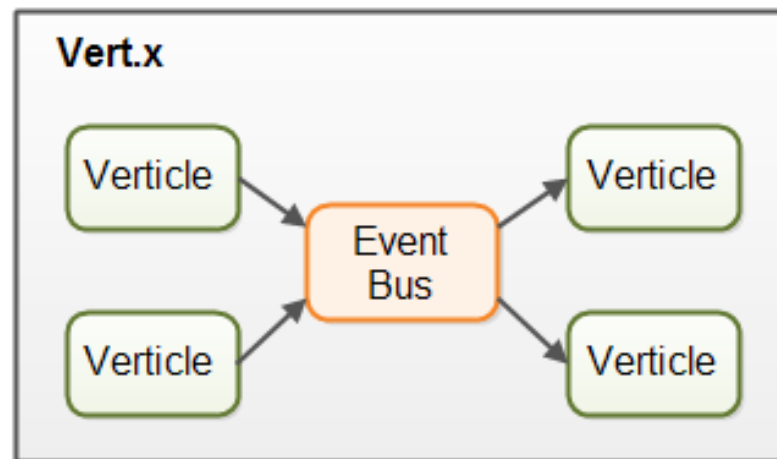
- Vert.x can deploy and execute components called Verticles.
- You can think of verticles as being similar to servlets or message driven EJBs in the Java Enterprise Edition model.





Verticles Event Bus

- Verticles are event driven, meaning they do not run unless they receive a message.
- Until then they remain dormant.
- Verticles can communicate with each other via the Vert.x event bus.





Vert.x Event Bus

- Messages can be simple objects (e.g. Java objects), strings, CSV, JSON, binary data or whatever else you need.
- Verticles can send and listen to addresses.
- An address is like a named channel.
- When a message is sent to a given address, all verticles that listen on that address receive the message.
- Verticles can subscribe and unsubscribe to addresses without the senders knowing.
- This results in a very loose coupling between message senders and message receivers.
- All message handling is asynchronous.
- If a verticle sends a message to another verticle, that message is first put on the event bus, and control returned to the sending verticle.
- Later, the message is dequeued and given to the verticles listening on the address the message was sent to.



The Vert.x Thread Model

- Verticles run in single thread mode.
- That means, that a verticle is only ever executed by a single thread, and always by the same thread.
- That means that you will never have to think about multithreading inside your verticle (unless you yourself start other threads which your verticle communicates with etc).
- Vert.x is capable of using all the CPUs in your machine, or cores in your CPU.
- Vert.x does this by creating one thread per CPU.
- Each thread can send messages to multiple verticles.
- Remember, verticles are event driven and are only running when receiving messages, so a verticle does not need to have its own exclusive thread.
- A single thread can distribute messages to multiple verticles.

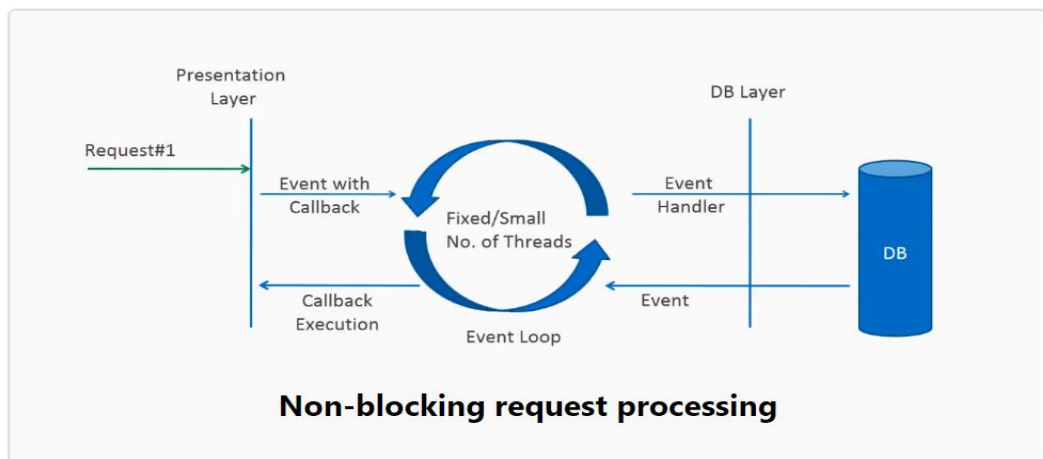
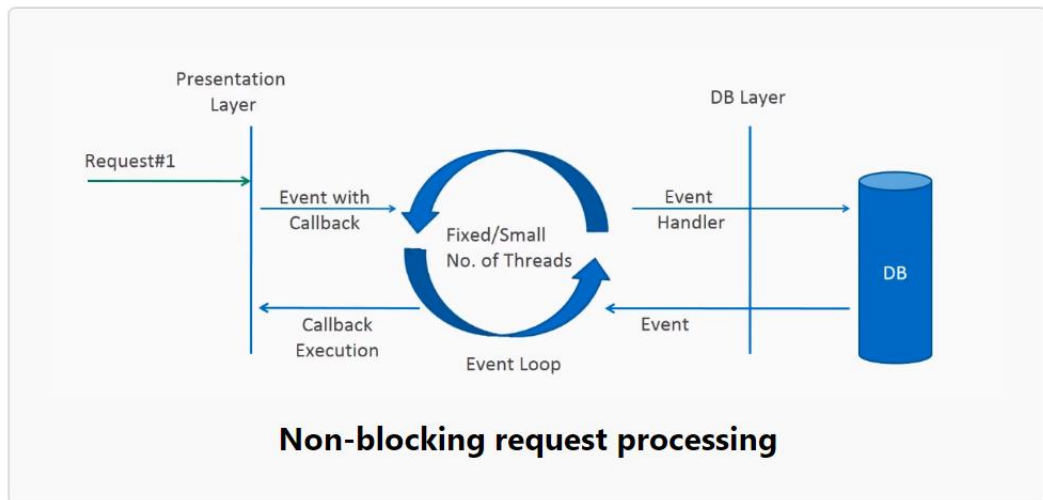
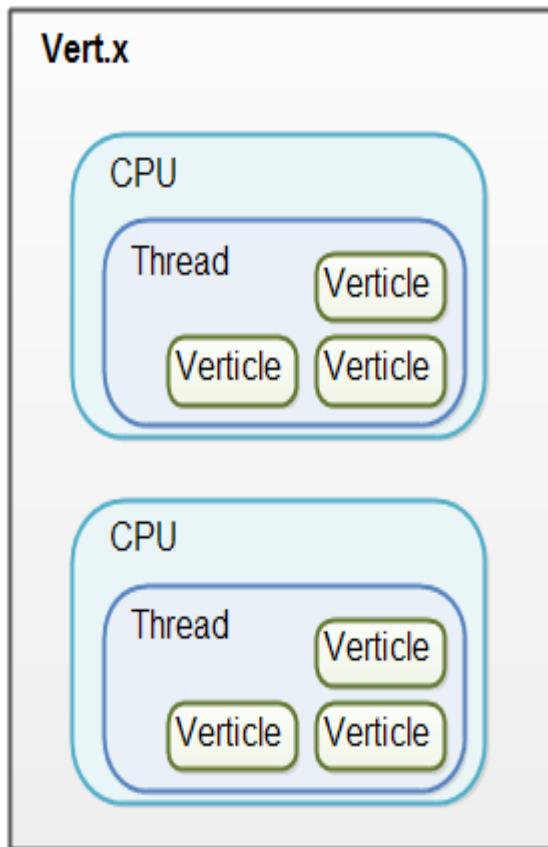


The Vert.x Thread Model

- When a thread delivers a message to a verticle, the message handling code of that verticle is executed by the thread.
- The message delivery and message handling logic is executed by calling a method in a handler (listener object) registered by the verticle.
- Once the verticle's message handling logic finishes, the thread can deliver a message to another verticle.



The Vert.x Thread Model



Vert.x Services



- Vert.x comes with a set of built-in services(functionality). Some of these services are:
 - HTTP server
 - JDBC connector
 - MongoDB connector
 - SMTP Mail
 - Message queue connectors

Installation



- It is distributed in a zip file having a bunch of Jar files.
- We simply unzip the file and add all the JAR files to the classpath of our Java application.
- After adding all the JAR files, we are ready to use Vert.x in our application.
- We can download the zip file from Maven Central or from the Bintray.
- Let's start with a new vert.x web project, but before that, we have to make sure that our system has the following things:
 - JDK 1.8 or higher.
 - An IDE or text editor.
 - Maven 3 or higher.
 - A browser for performing HTTP requests.

Installation



- In order to create the first web project using Vert.x, we should have to follow the following steps:

Create a new Vert.x application

Version

3.9.7 4.0.3 4.1.0.Beta1 4.1.0-SNAPSHOT

Language

Java Kotlin

Build

Maven Gradle

Group Id

com.examplecom.rps

Artifact Id

starter

Dependencies (1/72)

Web, MQTT, etc.

[+ Show dependencies panel](#)

[Advanced options +](#)

Selected dependencies (1)

Vert.x Web x

Generate Project alt + ⌘ >_

Deployment



```
C:\Windows\System32\cmd.exe - mvn exec:java
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. All rights reserved.

G:\Local disk\vertx\newbie>mvn exec:java
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.rps:newbie >-----
[INFO] Building newbie 1.0.0-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] --- exec-maven-plugin:3.0.0:java (default-cli) @ newbie ---
HTTP server started on port 8888
May 16, 2021 10:40:19 PM io.vertx.core.impl.launcher.commands.VertxIsolatedDeployer
INFO: Succeeded in deploying verticle
-
```



Implementing a Verticle

- You implement a verticle by creating a class that extends `io.vertx.core.AbstractVerticle`.
- Here is an example verticle class:

```
import io.vertx.core.AbstractVerticle;  
public class BasicVerticle extends AbstractVerticle {  
  
}
```



- The AbstractVerticle class contains a start() method which you can override in your verticle class.
- The start() method is called by Vert.x when the verticle is deployed and ready to start.
- Here is how implementing the start() method looks:

```
public class BasicVerticle extends AbstractVerticle {  
  
    @Override  
    public void start() throws Exception {  
        System.out.println("BasicVerticle started");  
    }  
}
```



- The `start()` method is where you initialize your verticle.
- Inside the `start()` method you will normally create e.g. HTTP or TCP server, register event handlers on the event bus, deploy other verticles, or whatever else your verticle needs to do its work.
- The `AbstractVerticle` class also contains another version of `start()` which takes a `Future` as parameter.
- This `Future` can be used to asynchronously tell `Vert.x` if the `Verticle` was deployed successfully.

Stop()



- The AbstractVerticle class also contains a stop() method you can override.
- The stop() method is called when Vert.x shuts down and your verticle needs to stop.
- Here is an example of overriding the stop() method in your own verticle:

```
public class BasicVerticle extends AbstractVerticle {  
  
    @Override  
    public void start() throws Exception {  
        System.out.println("BasicVerticle started");  
    }  
  
    @Override  
    public void stop() throws Exception {  
        System.out.println("BasicVerticle stopped");  
    }  
}
```



Vertical Deploy()

- First a Vertx instance is created.
- Second, the `deployVerticle()` method is called on the Vertx instance, with an instance of your verticle (BasicVerticle in this example) as parameter.
- Vert.x will now deploy the verticle internally.
- Once Vert.x deploys the verticle, the verticle's `start()` method is called.
- The verticle will be deployed asynchronously, so the verticle may not be deployed by the time the `deployVerticle()` method returns.
- If you need to know exactly when a verticle is fully deployed, you can provide a Handler implementation to the `deployVerticle()`.



Vertical Deploy()

```
public class VertxVerticleMain {  
    public static void main(String[] args) throws InterruptedException  
    {  
        Vertx vertx = Vertx.vertx();  
  
        vertx.deployVerticle(new BasicVerticle());  
    }  
}
```



Vertical Deploy()

```
vertx.deployVerticle(new BasicVerticle(), new
Handler<AsyncResult<String>>() {
    @Override
    public void handle(AsyncResult<String> stringAsyncResult) {
        System.out.println("BasicVerticle deployment complete");
    }
});
```

Or Lambda Implementation

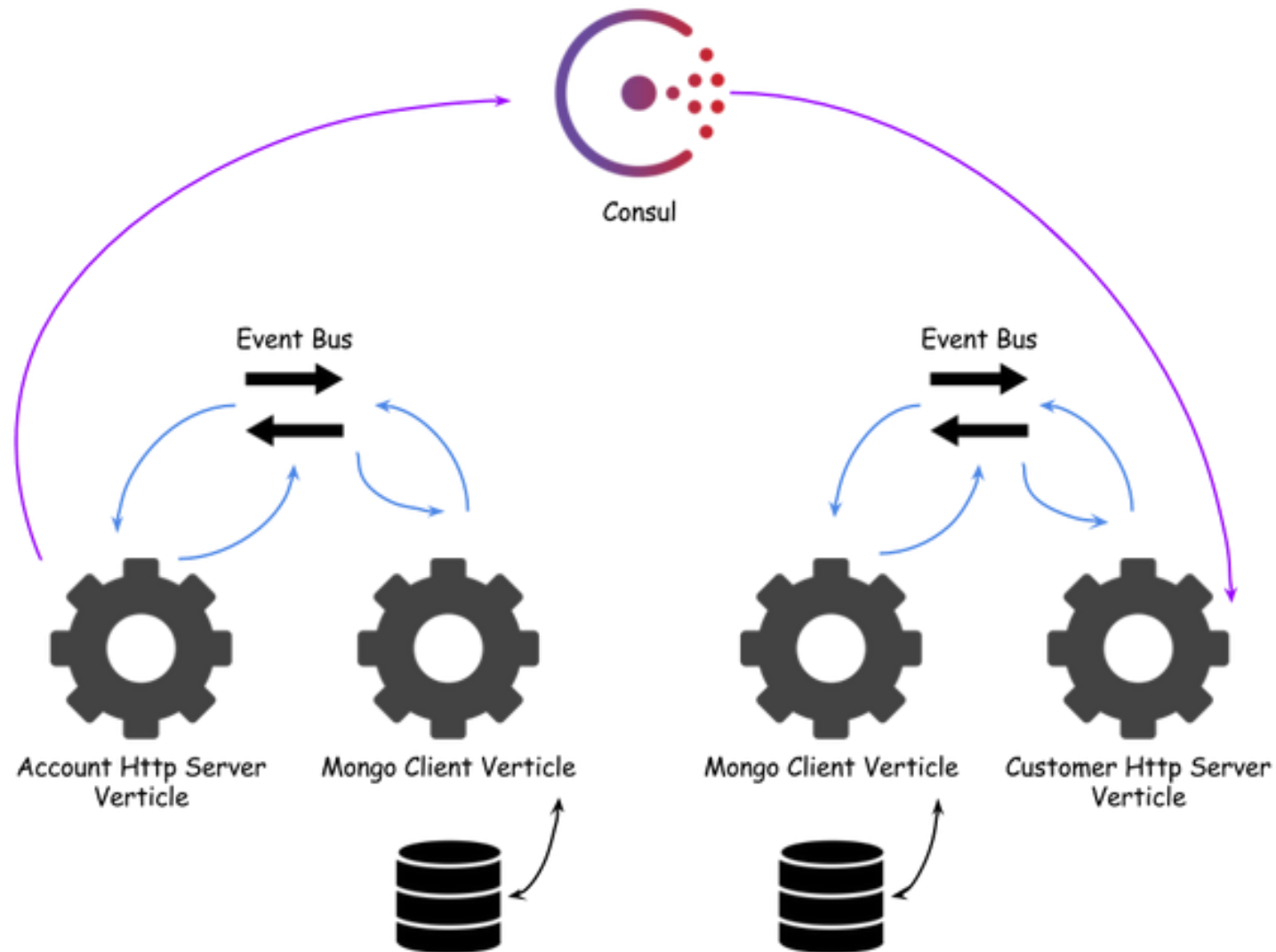
```
vertx.deployVerticle(new BasicVerticle(), stringAsyncResult -> {
    System.out.println("BasicVerticle deployment complete");
});
```



Deploying a Verticle From Another Verticle

```
public class BasicVerticle extends AbstractVerticle {  
  
    @Override  
    public void start() throws Exception {  
        System.out.println("BasicVerticle started");  
  
        vertx.deployVerticle(new SecondVerticle());  
    }  
  
    @Override  
    public void stop() throws Exception {  
        System.out.println("BasicVerticle stopped");  
    }  
}
```

Vert.x Microservice



Questions



Module Summary



- Spring Integration Framework.
- Message, Channel and Adapter
- Understood the different Component Integration
- Understood the Event-Driven Architecture

