Vert.x

Imperative vs Reactive Programming

Imperative programming is based on a *step-by-step execution model*. The program explicitly defines **how** something should be done — executing instructions sequentially, often in a thread-per-request model. When a blocking operation occurs (e.g., database call), the thread waits until the result is returned before continuing.

It is:

- Synchronous by default
- Blocking by nature
- Control-flow driven (call stack based)
- Easier to reason about linearly

Reactive programming is based on an *event-driven, non-blocking model*. Instead of waiting for operations to complete, the system reacts to events when results become available. Execution is composed around asynchronous streams or callbacks, typically using a small number of event loop threads.

It is:

- Asynchronous and non-blocking
- Event-driven
- Push-based (data flows through streams)
- Designed for high concurrency and IO-bound workloads

Imperative Programming (Traditional Spring MVC)

Model:

- Thread per request
- Blocking calls
- Sequential execution
- Direct control flow

```java
@GetMapping("/user/{id}")
public User getUser(@PathVariable Long id) {
    User user = userRepository.findById(id); // blocking DB call
    return user;
}
```

What happens internally:

1. Thread assigned from pool
2. Thread blocks while waiting for DB
3. Thread released only after response returned

Reactive Programming

Model:

- Event-driven
- Non-blocking IO
- Callback / Future / Promise / Stream-based
- Small number of event loop threads

java

```java
userRepository.findById(id)
    .onSuccess(user -> {
        response.end(user.toJson());
    });
```

Key difference:

- Instead of: "Wait for result, then continue"
- We say: "When result is ready, notify me"

Why Thread-per-Request Breaks at Scale

- Example: 2000 concurrent users
- Each DB call takes 200ms
- Thread pool = 200 threads

Explain:

- Threads are blocked most of the time
- Increasing thread pool → more memory + context switching
- System scales vertically, not efficiently

Note:
Imperative model scales by adding threads.

Reactive model scales by reducing waiting.

What is Vert.x?

Vert.x is a toolkit for building event-driven, non-blocking applications on the JVM.

Important:

- Not a framework like Spring
- No heavy abstraction
- No annotation magic
- You control everything
- Event-loop architecture
- Lightweight
- Polyglot
- Built-in event bus
- Designed for high concurrency

Vert.x Core Architecture

Event Loop Threads:

- Small number (usually 2 × CPU cores)
- Handle all I/O
- Must NEVER block

Worker Threads

- For blocking operations
- Used explicitly

Event Bus

- Async communication mechanism
- Local or clustered

Vert.x VS Spring

```
HTTP Request
     ↓
Event Loop
     ↓
Non-blocking Handler
     ↓
Async DB Call
     ↓
Callback
     ↓
Response
```

```
HTTP Request → Thread → Blocking DB → Response
```

What is a Verticle?

Verticle = lightweight component (similar to service bean, but more explicit)

Types:

- Standard verticle (event loop based)
- Worker verticle (blocking allowed)

```java
public class HttpVerticle extends AbstractVerticle {
  @Override
  public void start() {
    vertx.createHttpServer()
      .requestHandler(req -> req.response().end("Hello Vert.x"))
      .listen(8080);
  }
}
```

Event Bus – Async Communication

- Publish/Subscribe
- Request/Reply
- Clustered communication
- Loose coupling

```
vertx.eventBus().consumer("orders", msg -> {
  msg.reply("Processed");
});
```

```
vertx.eventBus().request("orders", "Order1", reply -> {
  System.out.println(reply.result().body());
});
```

This is like having internal Kafka-like messaging inside your application.

Futures and Async Flow

```
getUser(id)
  .compose(user -> getOrders(user))
  .onSuccess(result -> handle(result))
  .onFailure(err -> handleError(err));
```

Spring WebFlux:

- Mono
- Flux

Vert.x:

- Future
- Promise

Vert.x vs Spring WebFlux

	Spring WebFlux	Vert.x
Abstraction	High	Lower
Opinionated	Yes	No
Event Bus	No	Yes
Configuration	Annotation	Code
Learning curve	Medium	Higher
Control	Less	More

Where Vert.x Shines

- High I/O microservices
- API gateway
- Real-time systems
- WebSockets
- Event-driven architecture
- Lightweight edge services

Not ideal for:

- Heavy blocking legacy systems
- CRUD-heavy monoliths

Vert.x Clustering

Vert.x can run multiple instances (nodes) and automatically cluster them.

When clustered:

- Event Bus works across JVM instances
- Services can communicate transparently
- No code changes needed

Supported Cluster Managers

- Hazelcast
- Infinispan
- Zookeeper

Backpressure & Flow Control

Problem

If:

- Service A produces events faster than Service B can consume,
- You risk memory explosion.

Solution in Vert.x

Vert.x streams support:

- Pause / Resume
- Flow control
- WriteQueueFull handling

Vert.x prevents uncontrolled memory growth by:

- Signaling when consumer is overloaded
- Allowing producer to slow down

Vert.x + Kafka (Event-Driven Microservices)

Why this is powerful:

- Non-blocking Kafka
- High throughput
- No thread pool explosion
- Naturally async

Reactive Database Drivers

Vert.x supports:

- Reactive PostgreSQL client
- Reactive MySQL client
- Reactive Mongo client

Spring JDBC:

- Blocking
- Thread-per-request

Vert.x SQL Client:

- Async
- Event-loop friendly

Vert.x + Kubernetes

Vert.x characteristics:

- Small memory footprint
- Fast startup
- Stateless by design
- Cluster-aware

Ideal for:

- Horizontal scaling
- Microservices in Kubernetes
- API gateway pods

Because:

- Few threads
- Efficient CPU usage
- High concurrency with low memory