1.
Vert.x,
Asynchronous programming,
and reactive systems
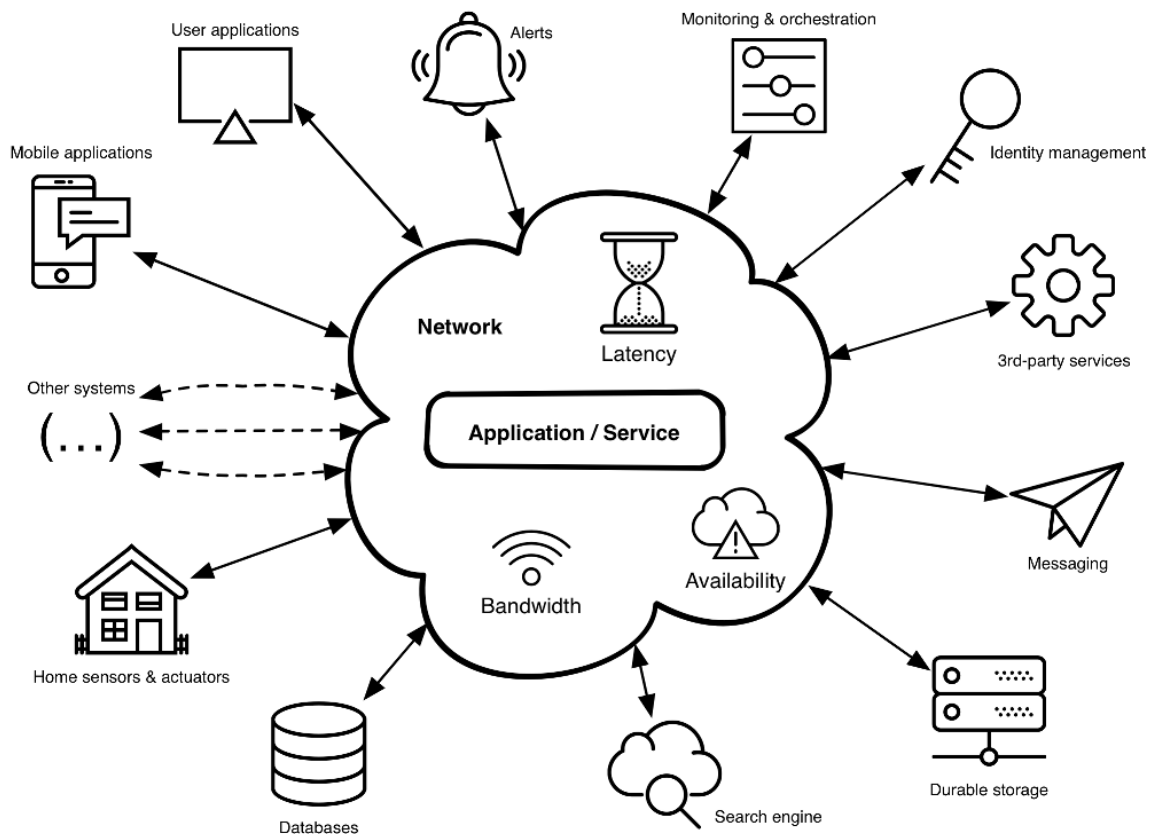
Being distributed and networked is the norm

💡 Today, an application is more naturally exposed to end users through **web** and **mobile** interfaces.

This naturally brings the -into play, and hence **distributed systems.**

Not living on an isolated island

A networked application/service

There is no free lunch on the network

- **The bandwidth can fluctuate a lot**
  so data-intensive interactions between services may suffer

- **The latency fluctuates a lot**
  and because services need to talk to services that talk to additional
  services to process a given request, all network-induced latency adds to
  the overall request-processing times.

- **Availability should not be taken for granted**
  Networks fail. Routers fail. Proxies fail.

Modern applications are made of distributed and networked services.

They are accessed over networks that introduce some problems on their own, and

each service needs to maintain several incoming and outgoing connections.

The simplicity of blocking APIs

- Services need to manage connections to other services and requesters
- The traditional and widespread model for managing concurrent network connections is to allocate a thread for each connection.

  - Servlets in JavaEE (before additions in version 3),
  - Spring Framework (before additions in version 5),
  - Ruby on Rails, Python Flask, and many more.

This model has the advantage of simplicity as it is **synchronous**.

▼ demo1: **TCPSynchronousEchoServer**

```java
import java.io.*;

public class TCPSynchronousEchoServer {
  public static void main(String[] args) throws Throwable {
    ServerSocket server = new ServerSocket();
    server.bind(new InetSocketAddress(3000));
    while (true) {
      Socket socket = server.accept();
      new Thread(clientHandler(socket)).start();
    }
  }

  private static Runnable clientHandler(Socket socket) {
    return () -> {
      try (
        BufferedReader reader = new BufferedReader(
```

```java
        new InputStreamReader(socket.getInputStream()));
      PrintWriter writer = new PrintWriter(
        new OutputStreamWriter(socket.getOutputStream()))) {
      String line = "";
      while (!"/quit".equals(line)) {
        line = reader.readLine();
        System.out.println("~ " + line);
        writer.write(line + "\n");
        writer.flush();
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  };
  }
}
```

💡 Above server code uses the main thread for accepting connections, and each connection is allocated a new thread for processing I/O.

The I/O operations are synchronous, so threads may **block** on I/O operations.

▼ 💻

In the beginning, there were threads...

The classic approach to concurrency is to use **threads**. Multiple threads can live within a single **process**, perform **concurrent** work, and **share** the same memory space.
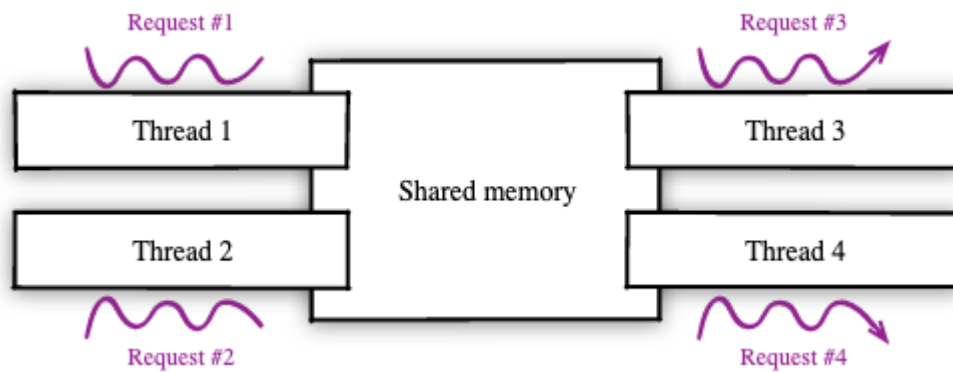
Most application and service development frameworks are based on multi-threading.

On the surface, the model of having 1 thread per connection is reassuring because developers can rely on traditional **imperative style** code.

This is fine, especially if you forget about those silly mistakes you can make with multi-threading and memory access ...

## Blocking APIs waste resources, increase costs
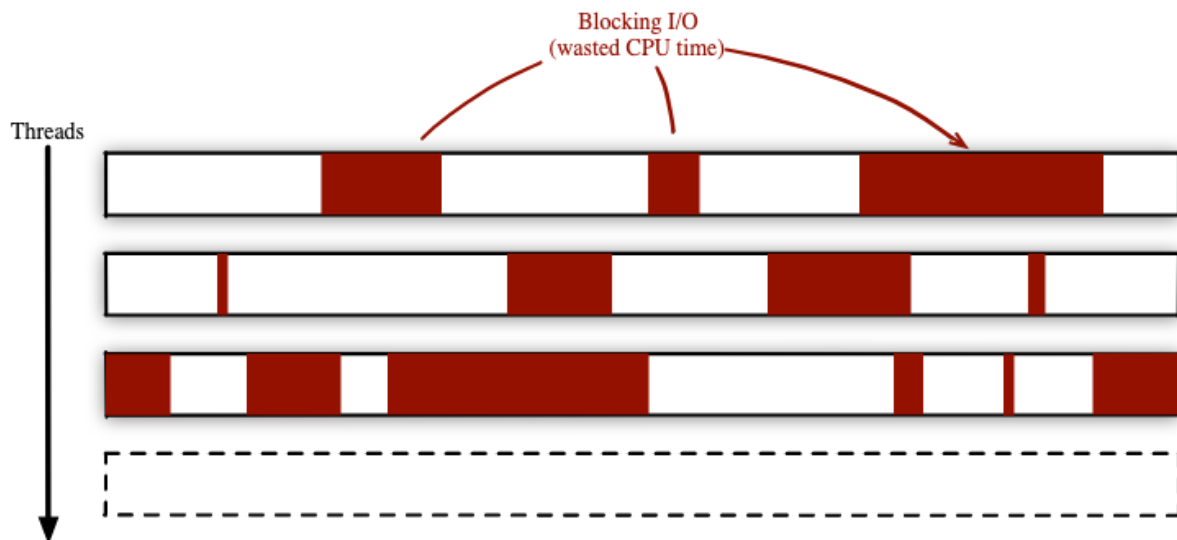
Input/output operations such as read and write may block the thread, meaning that it is being parked by the operating system.

This happens for two reasons:

1. A read operation may be waiting for data to arrive from the network / any external sources.

2. A write operation may have to wait for buffers to be drained if they are full from a previous write operation.

💡 It is also important to recall that we often need more threads than incoming network connections
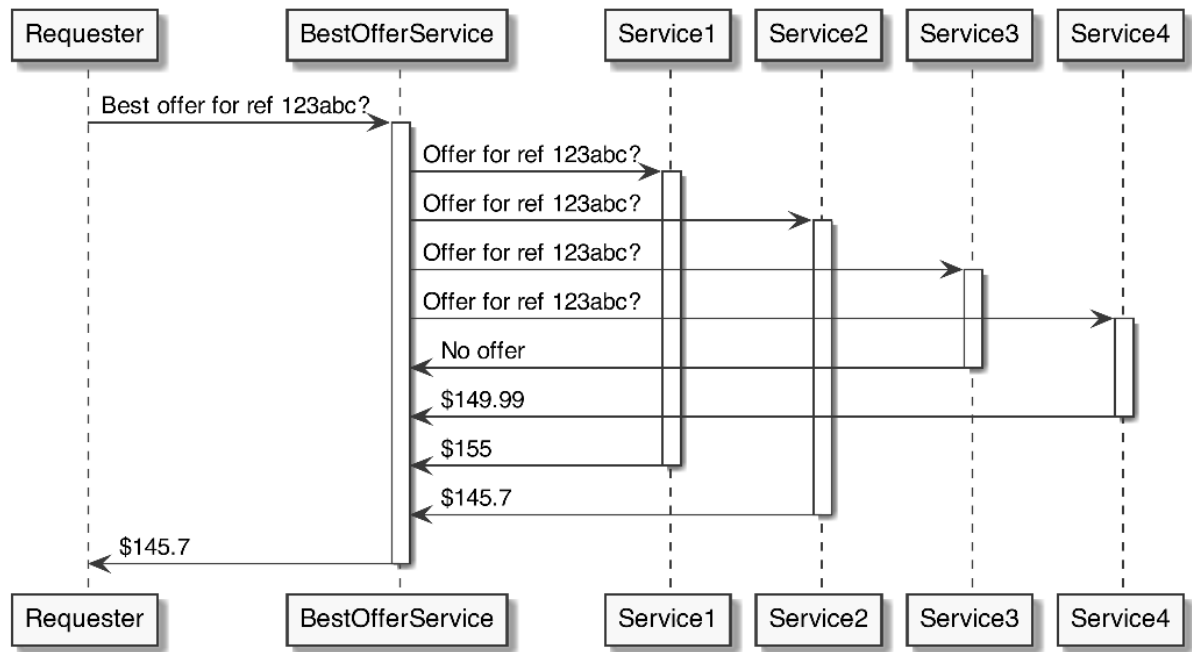
💡 so if all applications use **blocking I/O APIs**, there can quickly be too many threads to manage and schedule, which requires starting more server/container instances as traffic ramps up.
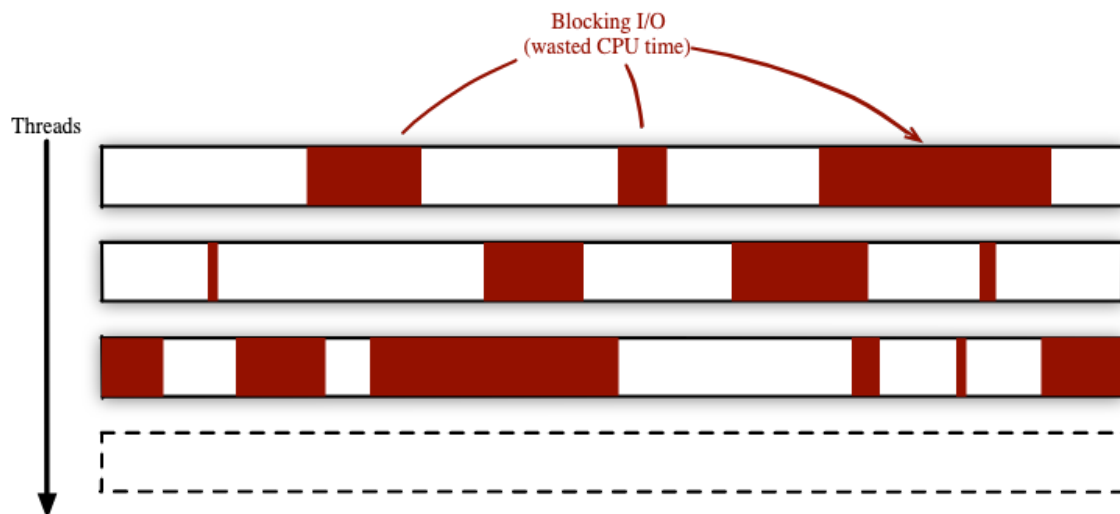
💡 This translates directly to increased operating costs. 😔

▼ 💻

Multi-threading is "simple" but limited

What happens as the workload grows beyond moderate workloads? (see the C10k problem)

The answer is simple: you start making your operating system kernel **suffer** because there is too much **context switching** work with in-flight requests.

Some of your threads will be **blocked** because they are waiting on I/O operations to complete, some will be **ready** to handle I/O results, and some will be in the middle of doing CPU-intensive tasks.

Modern kernels have very good schedulers, but you cannot expect them to deal with 50 000 threads as easily as they would do with 5 000. Also, threads aren't cheap: creating a thread takes a few milliseconds, and a new thread eats about 1MB of memory.

Asynchronous programming with non-blocking I/O

- Instead of waiting for I/O operations to complete, we can shift to **non-blocking I/O.**

- The idea behind non-blocking I/O is to request a (blocking) operation, and move on to doing other tasks until the operation is ready.

  - For example, a non-blocking read may ask for up to 256 bytes over a network socket, and the execution thread does other things (like dealing with another connection) until data has been put into the buffers, ready for consumption in memory.

- In this model, many concurrent connections can be **multiplexed** on a **single thread**, as network latency typically exceeds the CPU time it takes to read incoming bytes.

👉 Java has long had the java.nio (Java NIO) package, which offers non-blocking I/O APIs over files and networks.

▼ demo-2: **TCPAsynchronousEchoServer**

```java
import java.nio.*;

public class TCPAsynchronousEchoServer {
  public static void main(String[] args) throws IOException {

    Selector selector = Selector.open();

    ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
    serverSocketChannel.bind(new InetSocketAddress(3000));
    serverSocketChannel.configureBlocking(false);
    serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

    while (true) {
      selector.select();
      Iterator<SelectionKey> it = selector.selectedKeys().iterator();
      while (it.hasNext()) {
        SelectionKey key = it.next();
        if (key.isAcceptable()) {
          newConnection(selector, key);
        } else if (key.isReadable()) {
          echo(key);
        } else if (key.isWritable()) {
          continueEcho(selector, key);
```

```
        }
        it.remove();
      }
    }
  }

  //...
```

👉 Above example shows, non-blocking I/O is doable, but it significantly increases the code complexity compared to the initial version that used blocking APIs.

👉 **java.nio** focuses solely on what it does (here, I/O APIs). It does not provide higher-level protocol-specific helpers

👉 Also, **java.nio** does not prescribe a threading model, which is still important to properly utilize CPU cores, handle asynchronous I/O events, and articulate the application processing logic.

👉 This is why, in practice, developers rarely deal with Java NIO. Networking libraries like **Netty** and **Apache Mina** solve the shortcomings of Java NIO, and many toolkits and frameworks are built on top of them. As you will soon discover, **Eclipse Vert.x** is one of them.

Multiplexing event-driven processing: the case of the event loop

imp-note : reactor/event-loop pattern

A popular threading model for processing asynchronous events is that of the event loop.

Instead of polling for events that may have arrived, as we did in the previous Java NIO example, events are pushed to an event loop.
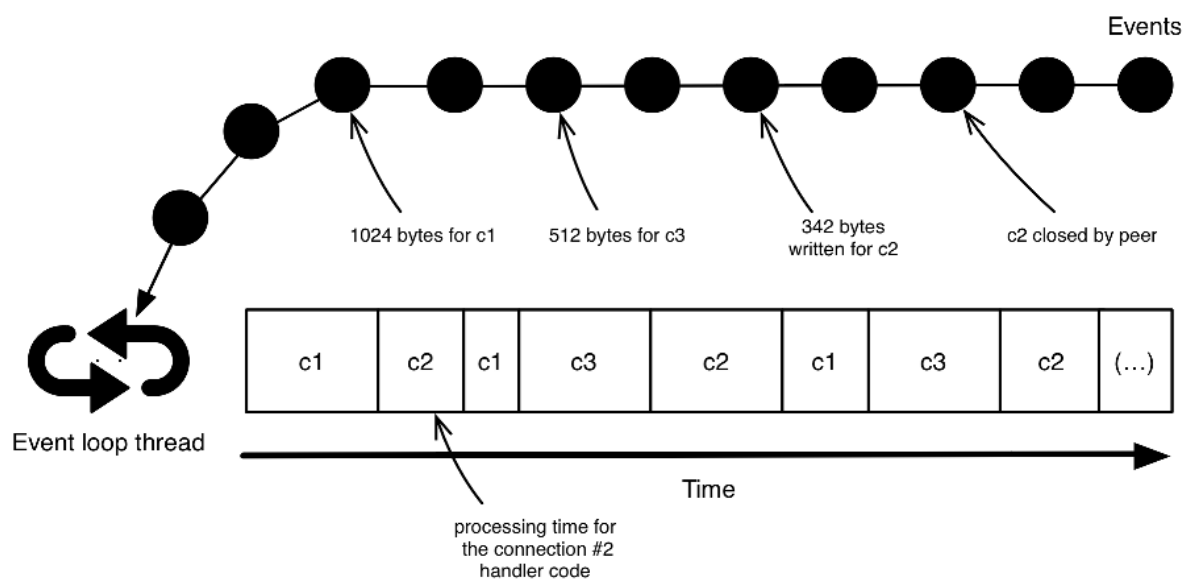
👉 A single thread is assigned to an event loop, and processing events shouldn't perform any blocking or long-running operation. Otherwise the thread blocks, defeating the purpose of using an event loop.

👉 Event loops are quite popular: JavaScript code running in web browsers runs on top of an event loop.

http://latentflip.com/loupe/

▼ 💻

Asynchronous programming: scalability and resource efficiency

Processing more concurrent connections with less threads is possible when you use **asynchronous I/O**. Instead of blocking a thread when an I/O operation occurs, we move on to another task which is ready to progress, and resume the initial task later when it is ready.

Vert.x multiplexes concurrent workloads using **event loops**.

Events

Time
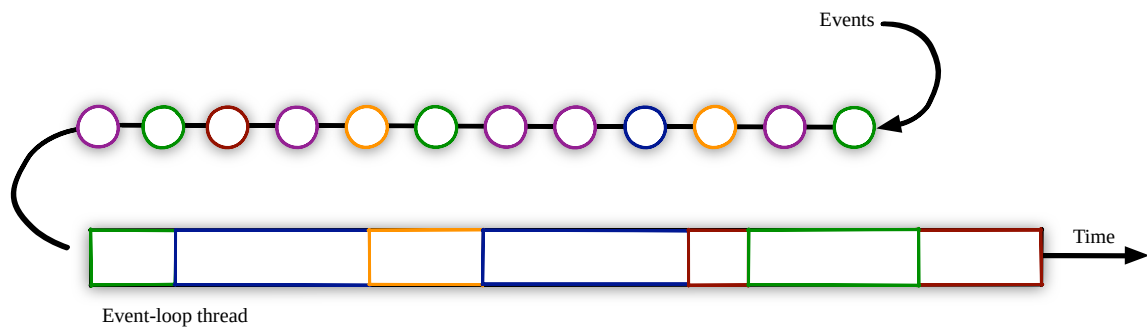
Event-loop thread

Code that runs on event loops should not perform blocking I/O or lengthy processing. But don't worry if you have such code: Vert.x has worker threads and APIs to process events back on an event loop.

Summary

- Leverage asynchronous programming and non-blocking I/O to handle more concurrent connections and use less threads

- Use one threading model for asynchronous event processing (the event loop)

By combining these two techniques, we can build scalable and resource-efficient applications

What is a reactive system?

Reactive applications are both **scalable** as workloads grow, and **resilient** when failures arise.

A reactive application is **responsive** as it keeps latency under control by making efficient usage of system resources, and by protecting itself from errors.

The four properties of reactive systems are exposed in The Reactive Manifesto:

responsive, resilient, elastic, and message-driven

The Reactive Manifesto

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more

R http://www.reactivemanifesto.org/

Why should we be reactive? 🙋‍♂️

The easiest way for us to answer this question is to think about the requirements
we have while building applications these days.

- Today everything should be online 24/7 and
  should respond with lightning speed;
- If it's slow or down, users would prefer an alternative service.
- Today slow means unusable or broken.

- We are working with greater volumes of data
  that we need to serve and process fast.

These are the new requirements,

we have to fulfill if we want our software to be competitive.

So in other words we have to be:

1. 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.

2. Scalable

   This way, we are going to be able to handle
   a huge amount of data
   or large numbers of user requests.

3. Fault-tolerant

   This way, the system will appear stable to its users.

4. Responsive

   This means fast and available.

Let's think about how to accomplish this:

👉 We can become modular if our system is event-driven.
We can divide the system into multiple micro-services or modules or components, that are going to communicate with each other using notifications / events / messages.
This way, we are going to react to the data flow of the system, represented by notifications.

👉 To be scalable means to react to the ever-growing data, to react to load without falling apart.

👉 Reacting to failures/errors will make the system more fault-tolerant.

👉 To be responsive means reacting to user activity in a timely manner

```
If the application is event-driven,
it can be decoupled into multiple self-contained components.

This helps us become more scalable, because we can always add new components
or remove old ones without stopping or breaking the system.
```

```
If errors and failures are passed to the right component,
which can handle them as notifications, as Events

the application can become more fault-tolerant or resilient.
```

```
So if we build our system to be event-driven,
we can more easily achieve scalability and failure tolerance,
and a scalable, decoupled, and error-proof application
is fast and responsive to users.
```
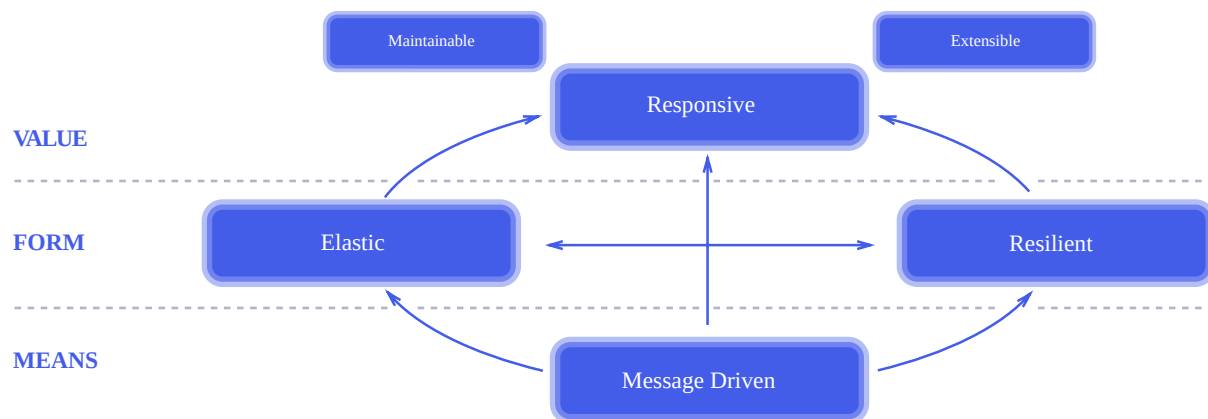

Changes in the application data can be modeled with notifications/messages/events, which can be propagated to the right handlers. So, writing applications using reactive programming is the easiest way to comply with the Manifesto.

Summary

Elastic — Elasticity is the ability for the application to work with a variable number of instances.

Resilient — Resiliency is partially the flip side of elasticity. When one instance crashes in a group of elastic instances, resiliency is naturally achieved by

redirecting traffic to other instances, and a new instance can be started if necessary.

Responsive — Responsivity is the result of combining elasticity and resiliency.

Message-driven — Using asynchronous message passing rather than blocking paradigms like remote procedure calls is the key enabler to elasticity and resiliency, which lead to responsiveness.

What else does reactive mean?

Reactive System:

Dependable applications that are message-driven, resilient, elastic, and responsive.

Reactive/Async Programming:

A means for reacting to changes and events

Spec: Reactive Extensions

RxJava, a popular reactive extensions API for Java that greatly helps coordinate asynchronous event and data processing

Reactive Streams:

When systems exchange continuous streams of data, the classical producer/consumer problems arise. It is especially important to provide back-pressure mechanisms so that a consumer can notify a producer when it is

emitting too fast. With reactive streams, the main goal is to reach the best throughput between systems.

Reactive Microservices
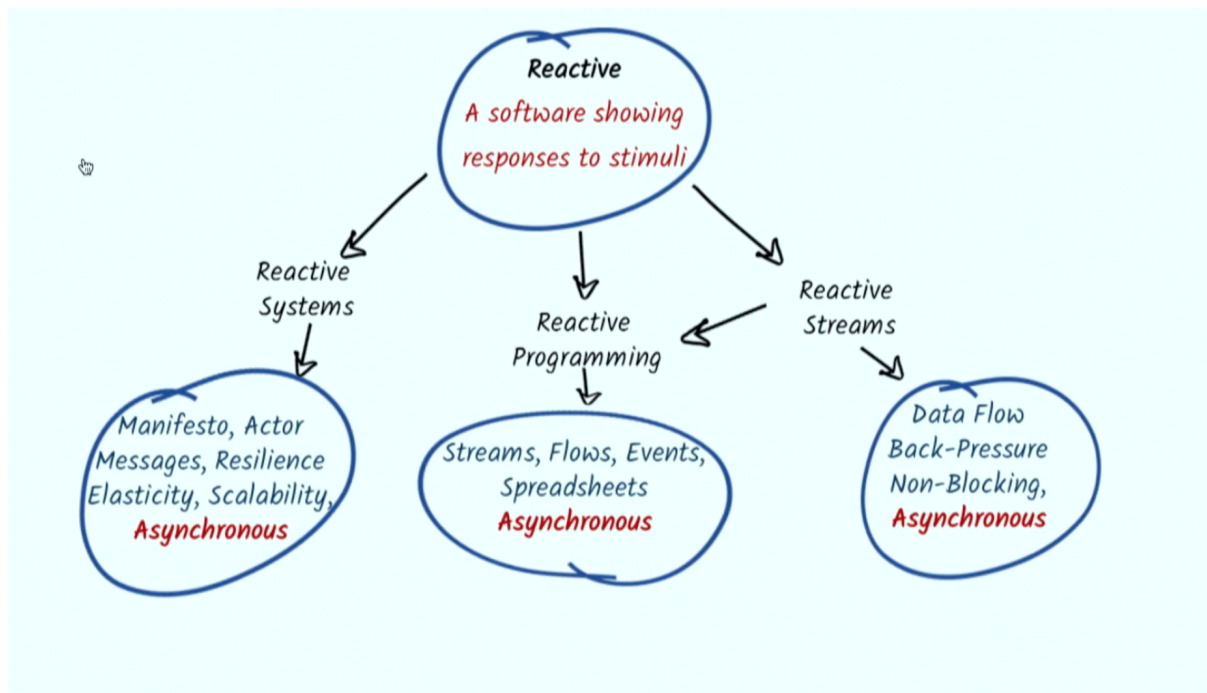
What is Vert.x?

According to the Vert.x website (https://vertx.io/),

"Eclipse Vert.x is a tool-kit for building reactive applications on the JVM."

Initiated by Tim Fox in 2012,

Vert.x is a project now fostered at the vendor-neutral Eclipse Foundation.

While the first project iterations were aimed at being a **"Node.js for the JVM,"**

Vert.x has since significantly deviated toward providing an asynchronous programming foundation tailored for the specifics of the JVM.
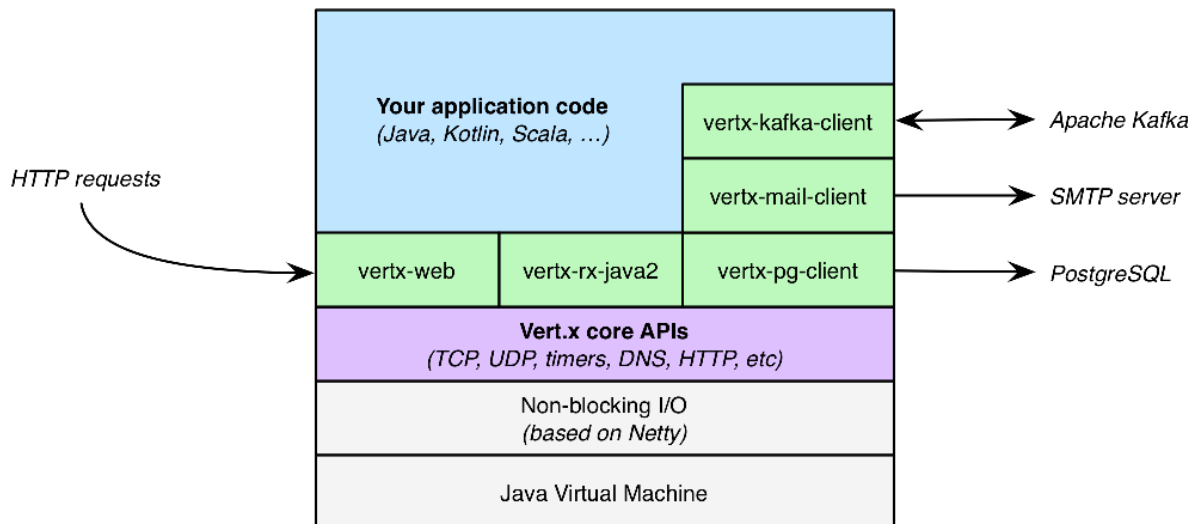
💡 The essence of Vert.x

 The focus of Vert.x is processing asynchronous events, mostly coming from non-blocking I/O, and the threading model processes events in an event loop.

💡 It is very important to understand that Vert.x is a toolkit and not a framework:

Overview of the structure of a Vert.x application

The Vert.x project is organized in composable modules, with figure showing the structure of a random Vert.x application:

> 💡 **A core project**, called vertx-core,
> provides the APIs for asynchronous programming, non-blocking I/O, streaming, and convenient access to networked protocols such as TCP, UDP, DNS, HTTP, or WebSockets.

> 💡 A set of modules that are part of the community-supported Vert.x stack, such as a better web API (**vertx-web**) or databases (**vertx-kafka-client, vertx-redis, vertx-mongo, etc.**) provide functionality for building all kinds of applications.

💡 A wider ecosystem of projects provides even more functionality, such as connecting with **Apache Cassandra**, **non-blocking I/O to communicate between system processes**, etc.

Vert.x is polyglot as it supports most of the popular JVM languages: **JavaScript, Ruby, Kotlin, Scala, Groovy, and more**

Your first Vert.x application

It's finally time for us to write a Vert.x application!

Let's continue with the echo TCP protocol, It will still expose a TCP server on port 3000, where any data is sent back to the client. We will add two other features:

1. The number of open connections will be displayed every 5 seconds.
2. An HTTP server on port 8080 will respond with a string giving the current number of open connections.

▼ demo-3 : TCPEchoServer

```java
package example;

import io.vertx.core.Vertx;
import io.vertx.core.net.NetSocket;

public class TCPVertxEchoServer {

    private static int numberOfConnections = 0;

    public static void main(String[] args) {

        Vertx vertx = Vertx.vertx();

        vertx.createNetServer()
                .connectHandler(Ex::handleNewClient)
                .listen(3000);

        vertx.setPeriodic(5000, id -> System.out.println(howMany()));

        vertx.createHttpServer()
                .requestHandler(request -> request.response().end(howMany()))
                .listen(8080);
    }

    private static void handleNewClient(NetSocket socket) {
        numberOfConnections++;
        socket.handler(buffer -> {
            socket.write(buffer);
            if (buffer.toString().endsWith("/quit\n")) {
                socket.close();
            }
        });
        socket.closeHandler(v -> numberOfConnections--);
    }

    private static String howMany() {
        return "We now have " + numberOfConnections + " connections";
    }

}
```

```
An event loop is managing the processing of events,
be it a new TCP connection,
the arrival of a buffer,
a new HTTP request, or
a periodic task that is being fired.

every event handler is being executed on the same (event-loop) thread
```

▼ So is this a reactive application?

The short answer is that no, this application is not reactive

The culprit for not being reactive clearly is elasticity

▼ What are the alternatives to Vert.x?

- Node.js. : always single-threaded process
- Akka   from lightbend
- Spring Framework : version 5. (if spring modules are based projectreactor)
- Quarkus
- Netty
- Scripting languages.   like Kotlin
- Native languages. like Go

Summary

- Asynchronous programming allows you to multiplex multiple networked connections on a single thread

- Managing non-blocking I/O is more complex than the equivalent imperative code based on blocking I/O, even for simple protocols.

- The event loop and the reactor pattern simplify asynchronous event processing.

- A reactive system is both scalable and resilient, producing responses with consistent latencies despite demanding workloads and failures.

- Vert.x is an approachable, efficient toolkit for writing asynchronous and reactive applications on the JVM.

Introduction to Vert.x and Reactive | Eclipse Vert.x

Eclipse Vert.x is a tool-kit for building reactive applications on the JVM. Reactive applications are both scalable as workloads grow, and resilient when failures arise. A reactive application is responsive as it keeps latency under control by making efficient usage of system resources, and by pro-

https://vertx.io/introduction-to-vertx-and-reactive/