

# Why Reactive ?

---

During past years, the requirements for modern applications have increased

- from gigabytes of data to terabytes,
- from thousands of requests per second to millions,
- from seconds of response time to milliseconds,

and this list can be continued.

Until now, most of these problems have been solved using the microservices architecture,

but what's next? What next approach should we apply where microservices fail?

---

**Being distributed and networked is the norm**

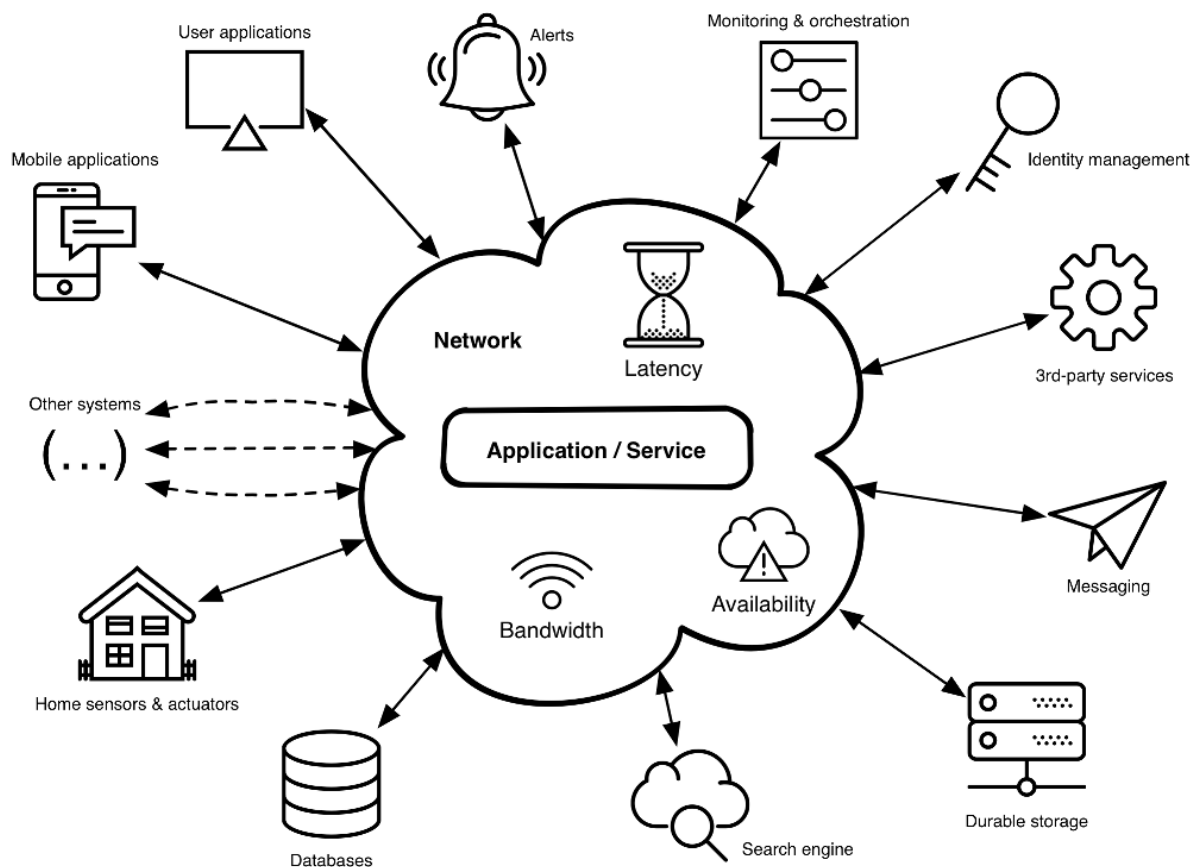


Today, an application is more naturally exposed to end users through **web** and **mobile** interfaces.

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 with Flask, and many more.

### thread per connection model

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

#### ▼ demo1: **TCPsynchronousEchoServer**

```
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(
                    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();
}
};
}
```

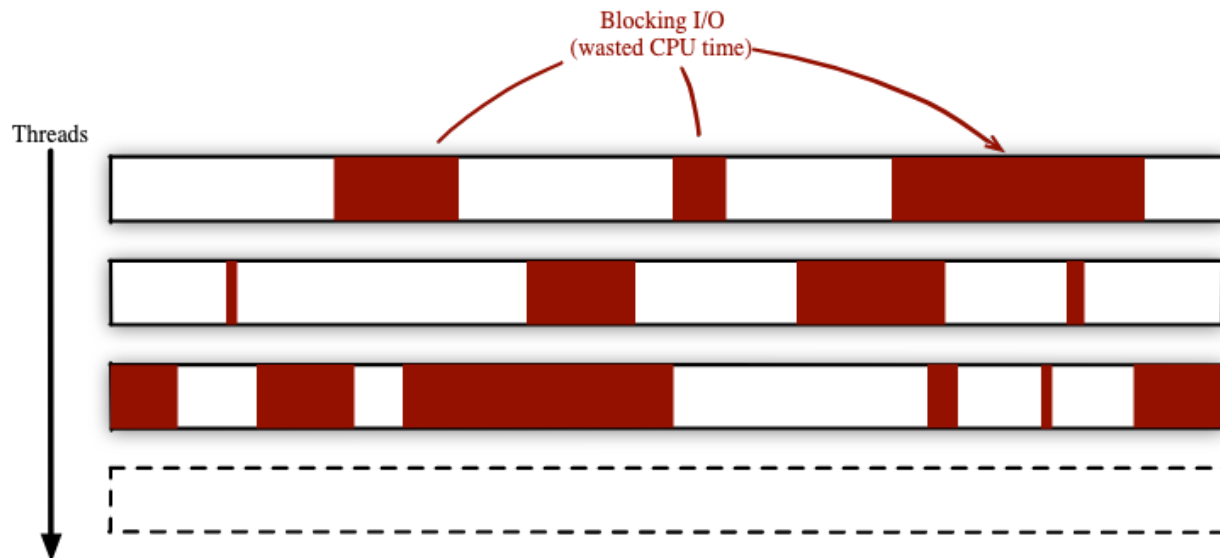


Above server code uses the main thread for accepting connections, and each connection is allocated a new thread for processing I/O.

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

---

## Blocking APIs waste resources, increase costs



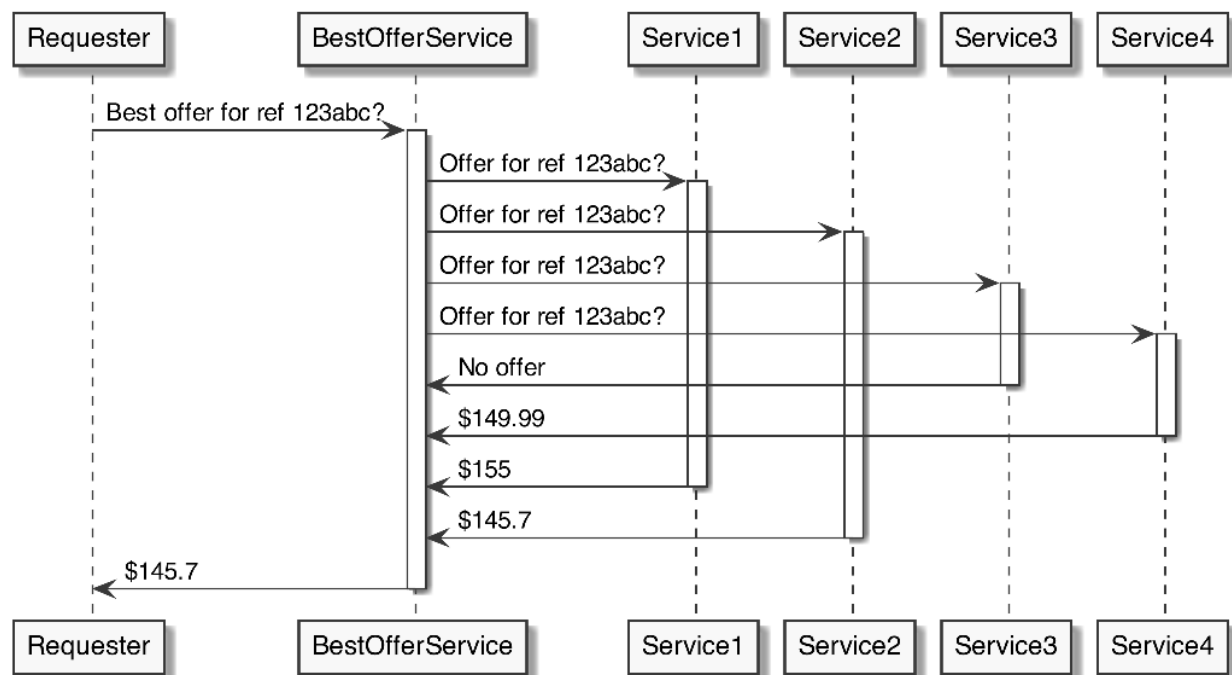
Input/output operations such as [read](#) and [write](#) may block the thread, meaning that it is being parked by the operating system.

This happens for two reasons:

1. A read operation may be waiting for data to arrive from the network / any external sources. ( file,db )
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.



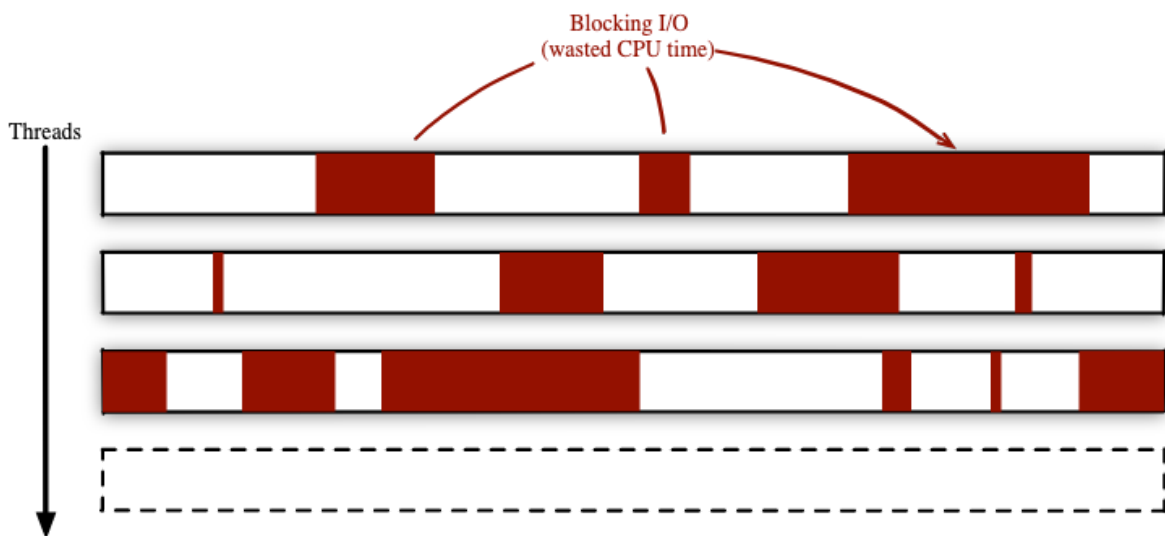
This translates directly to **increased operating costs**. 😞



## 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

```
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 like HTTP , .....

☞ 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.

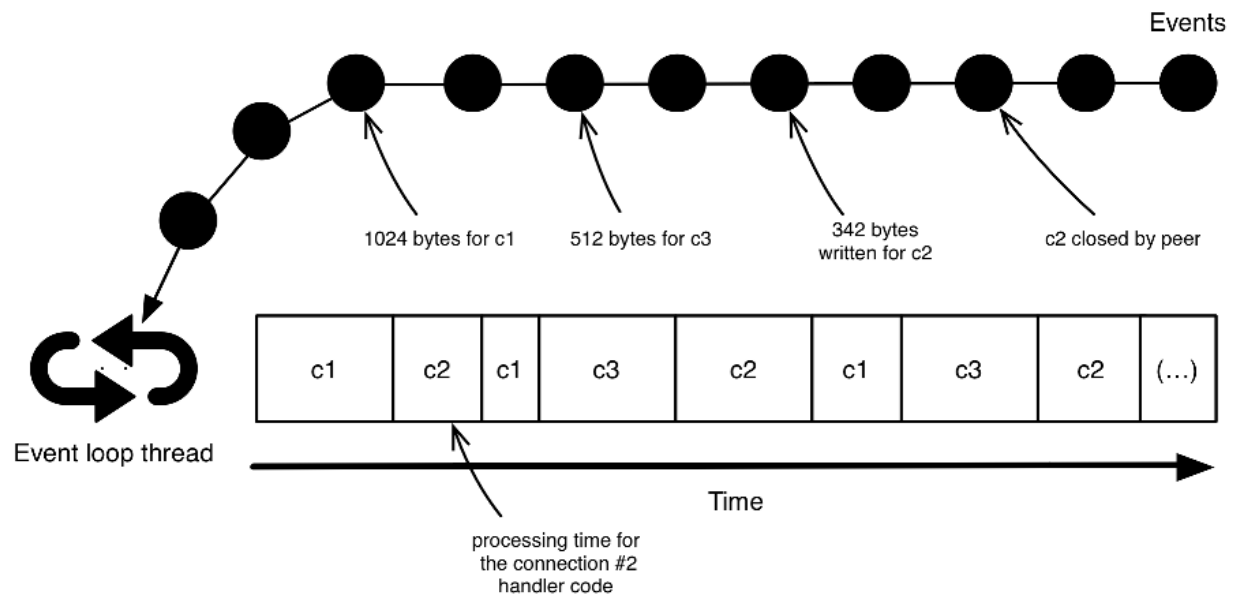
---

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

a.k.a, **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 & Node.js 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.

---

### 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 java Applications/Microservices

---

## What is a reactive system/application/service?


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/event/data-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 resilient, more flexible

 <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/verticle, 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



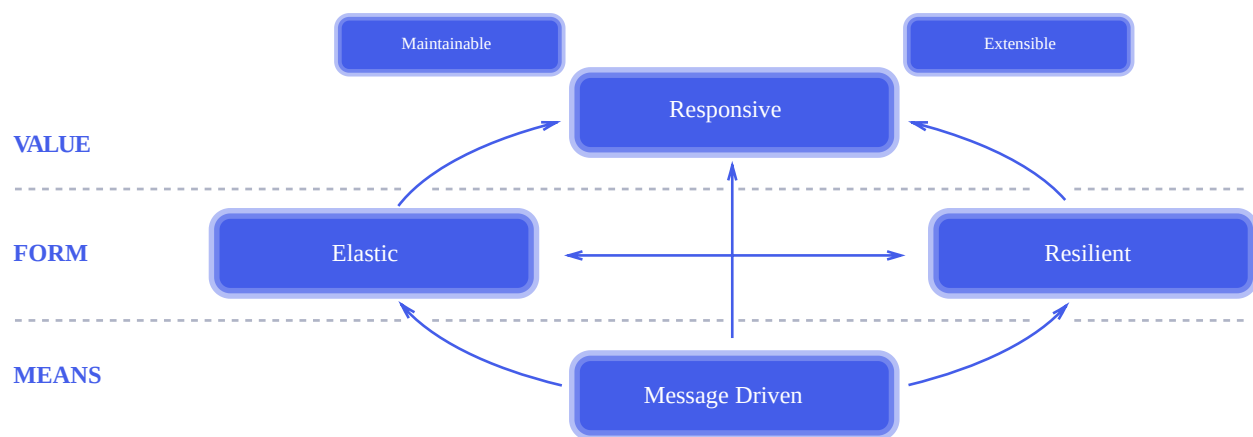
If the application is event-driven,  
it can be decoupled into multiple self-contained components.

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 [Async/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.

---

Reactive/Async programming != Reactive Systems

Reactive/Async programming + Reactive Principles = Reactive Systems/Applications/Microservices

---

## What else does reactive mean?

### Reactive System / Reactive MicroService:

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

### Reactive/Async Programming:

A means for reacting to changes or events

default options : Using **callbacks**

Spec: Reactive eXtensions ( Rx ) - <http://reactivex.io/>

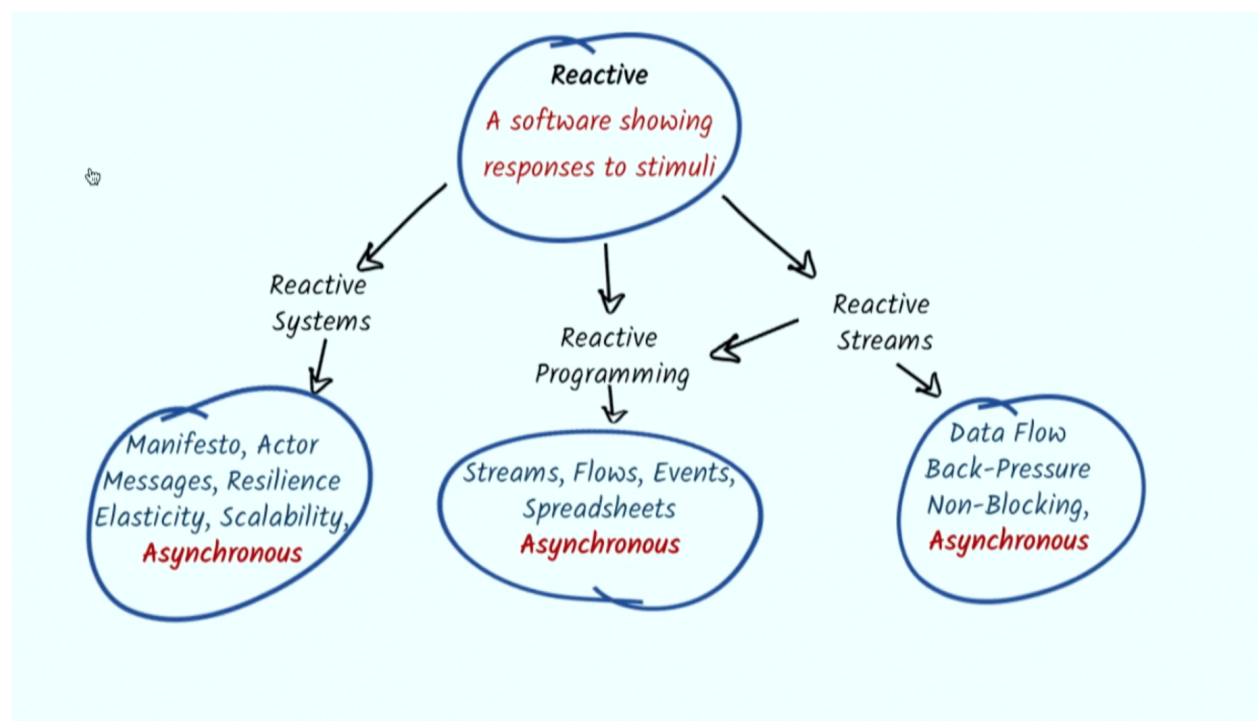
### RxJava

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

### Mutiny

## Reactive/Async 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.



- Async programming And Non-Blocking
- Reactive

- Programming
  - Systems
  - Streams
- 

## 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.
- 

For Java Developers

3 choices to develop reactive applications

Vertx

Quarkus

Akka

Spring - Reactor ( our focus )

---