Battle of Java Frameworks:

Case study of Quarkus and Spring WebFlux

by Vodafone Greece

Devstaff Greece
13 July 2023

C1


WELCOME

Hello

Roi Arapoglou
API Dev Chapter Leader

Alex Argyriou
Backend Developer

Stelios Tsiakalos
API Engineer

George Sorotos
Backend Developer

Agenda

01. Vodafone Digital eXperience Layer History

02. How did we get Reactive

03. SmallRye Mutiny intro

04. Developing with SmallRye Mutiny

05. Reactive in Spring Universe

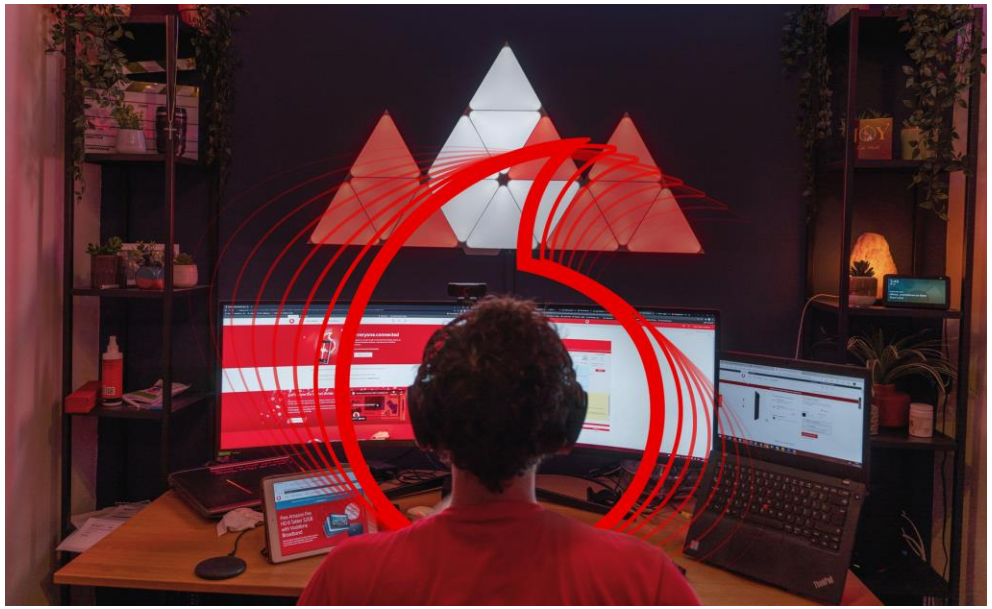
06. Developing with Spring WebFlux

07. Q&A

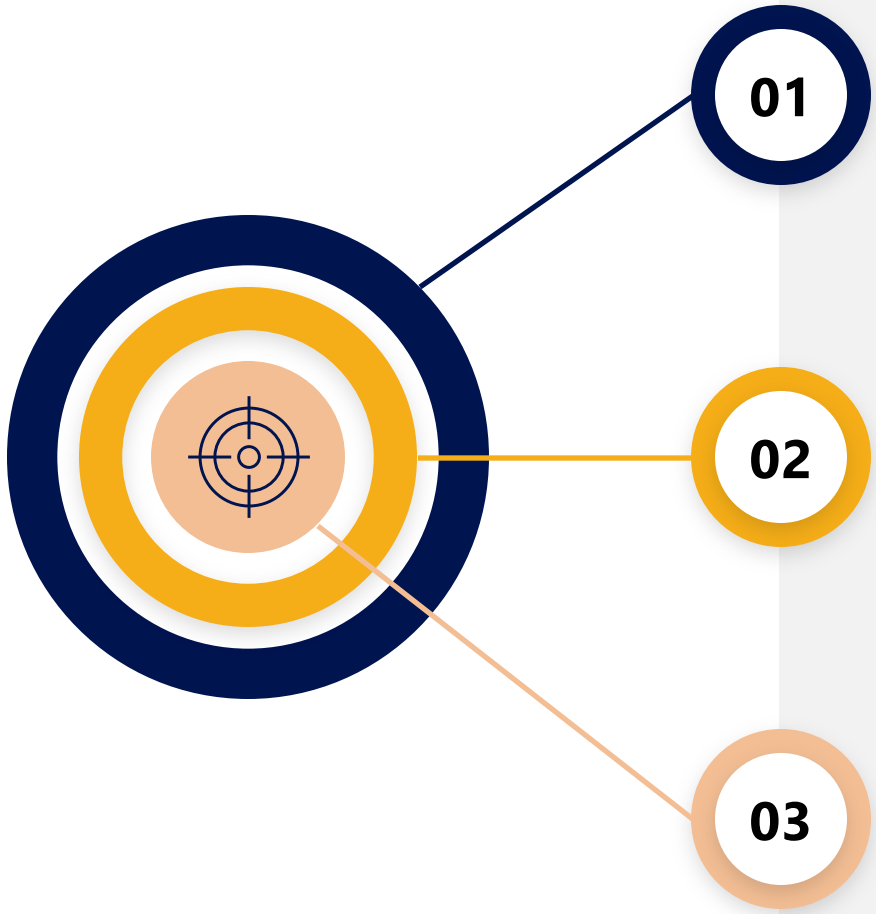
Digital eXperience Layer

Vodafone Greece: Digital Transformation

- **Vodafone** operates on 21 countries and serves millions of mobile and fixed users
- **Digital technology** can now collect so much more data about customer behavior and turn data into something great.
- **Digital transformation** delivers a modern and digital architecture to increase performance in terms of quality and omni channel experience
  - Web Channels (VFAPP, VFWEB, TOBI)
  - Store (Physical and eKiosks)
  - Call center agents
- **Digitalization of customer interaction** has been accelerated with COVID-19
- **Digital and agile initiatives transform Vodafone from a TelCo to a TechCo company**



What makes a good frontend user experience?



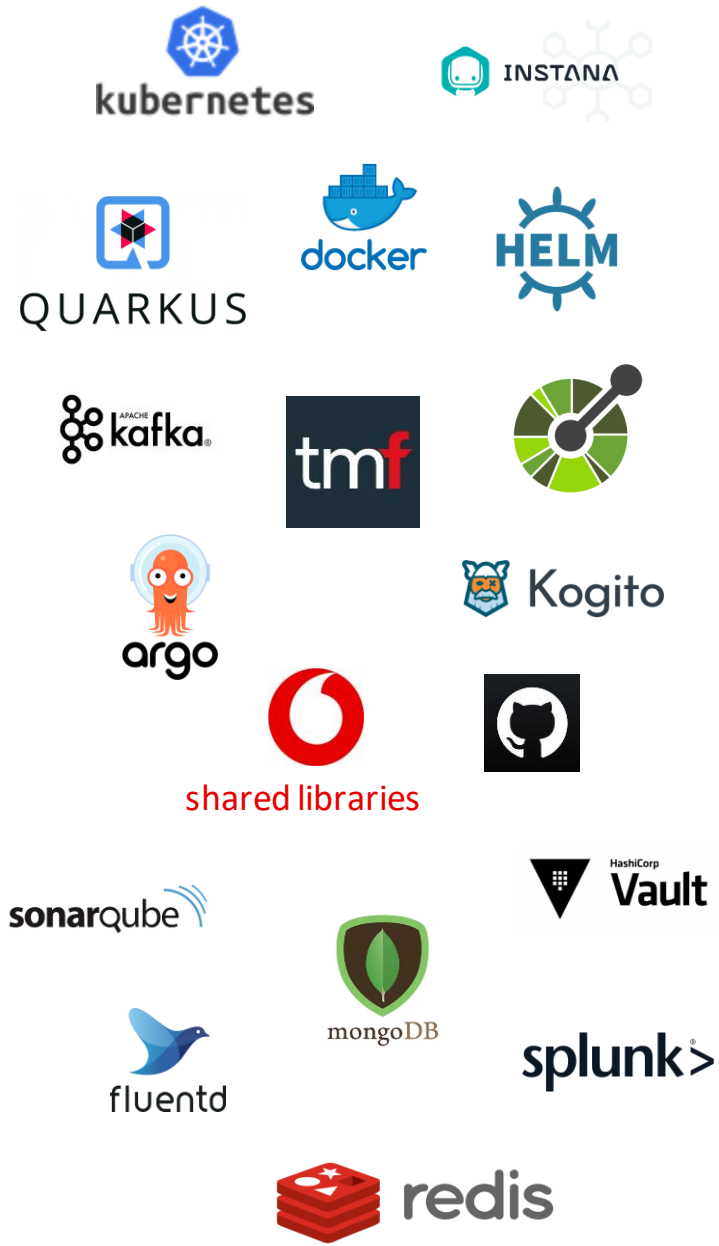
API Performance
A frontend application spends most of its fetching data. Fast APIs are key to the performance of an application.

Intuitive features
Mobile applications rely on feature-rich backends to offer intuitive capabilities

Developer performance Developers have a greater throughput and output when they are working with intuitive backends

API Development Chapter

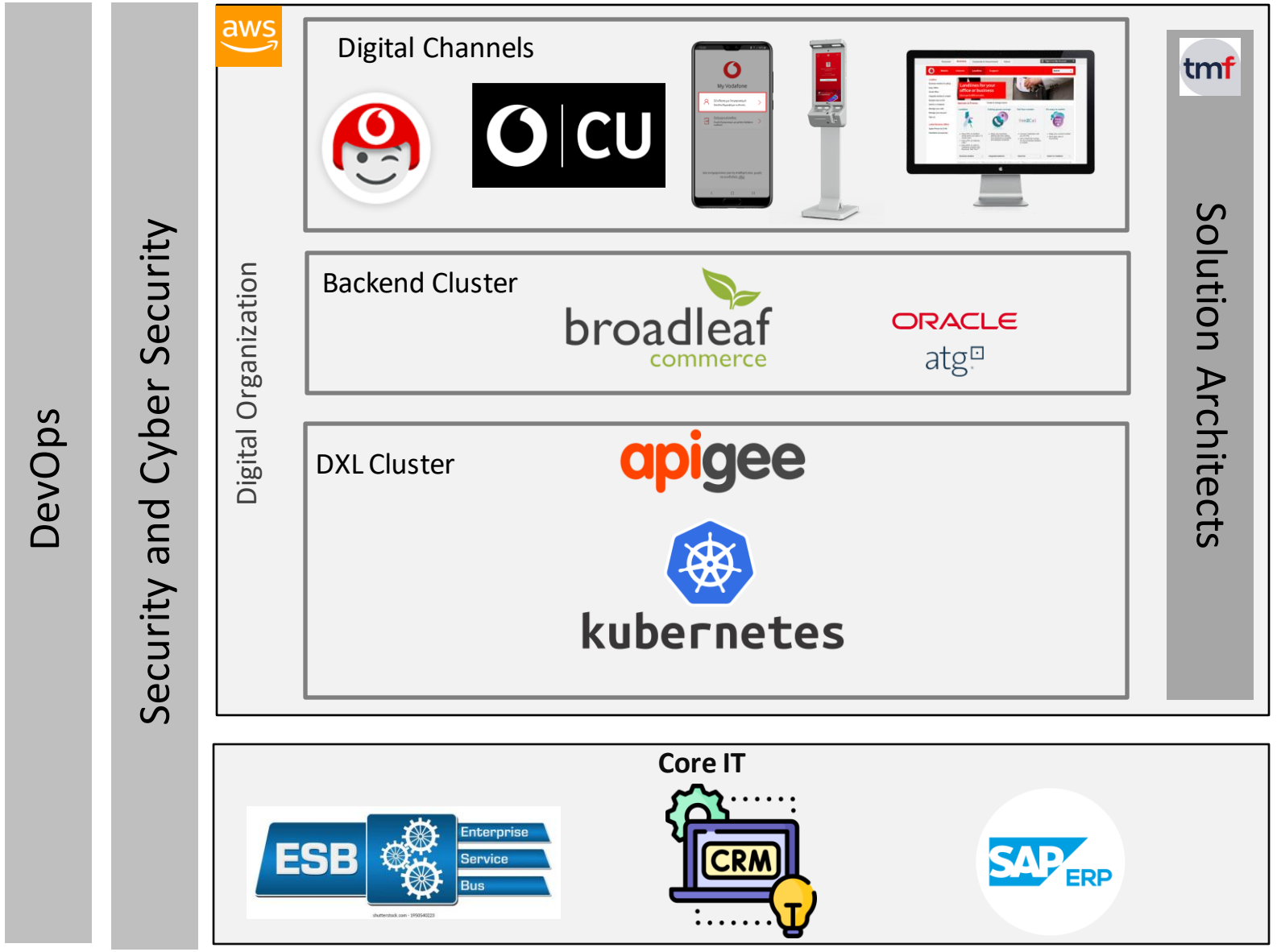
- 30 Team members do **R&D and in-house development**
- Decommission legacy backend systems and adoption of modern microservice SOA
- 170 microservices on production
- **~5,075,524** total calls per day or ~58.74 req/sec
- **198ms** mean latency and **0.17%** error rate
- Adoption of Quarkus framework since 2019
- Experience in DevOps tools



Backend Chapter

- 12 Team members do **R&D and in-house development.**
- Decommission legacy monolith backend systems and adoption of modern microservices architectures.
- 39 microservices will be used as the main backend ecommerce platform.

Digital eXperience Layer



Microservice architecture: Spatial and Temporal decoupling

- Spatial decoupling enables location transparency:
  - Virtual address rather than physical address
  - Replicas across different locations
  - Load balancers, service registry, cluster orchestration etc
- Temporal Decoupling enables do things independent of time:
  - When interactions are coupled with time, components must be always available
  - Can't always predict availability and reachability.
  - Asynchronous (non-blocking) communication
    - CQRS design pattern
    - Messaging systems e.g. Kafka, RabbitMQ
    - **Reactive programming paradigm**

How did we get Reactive

Technology Advancements

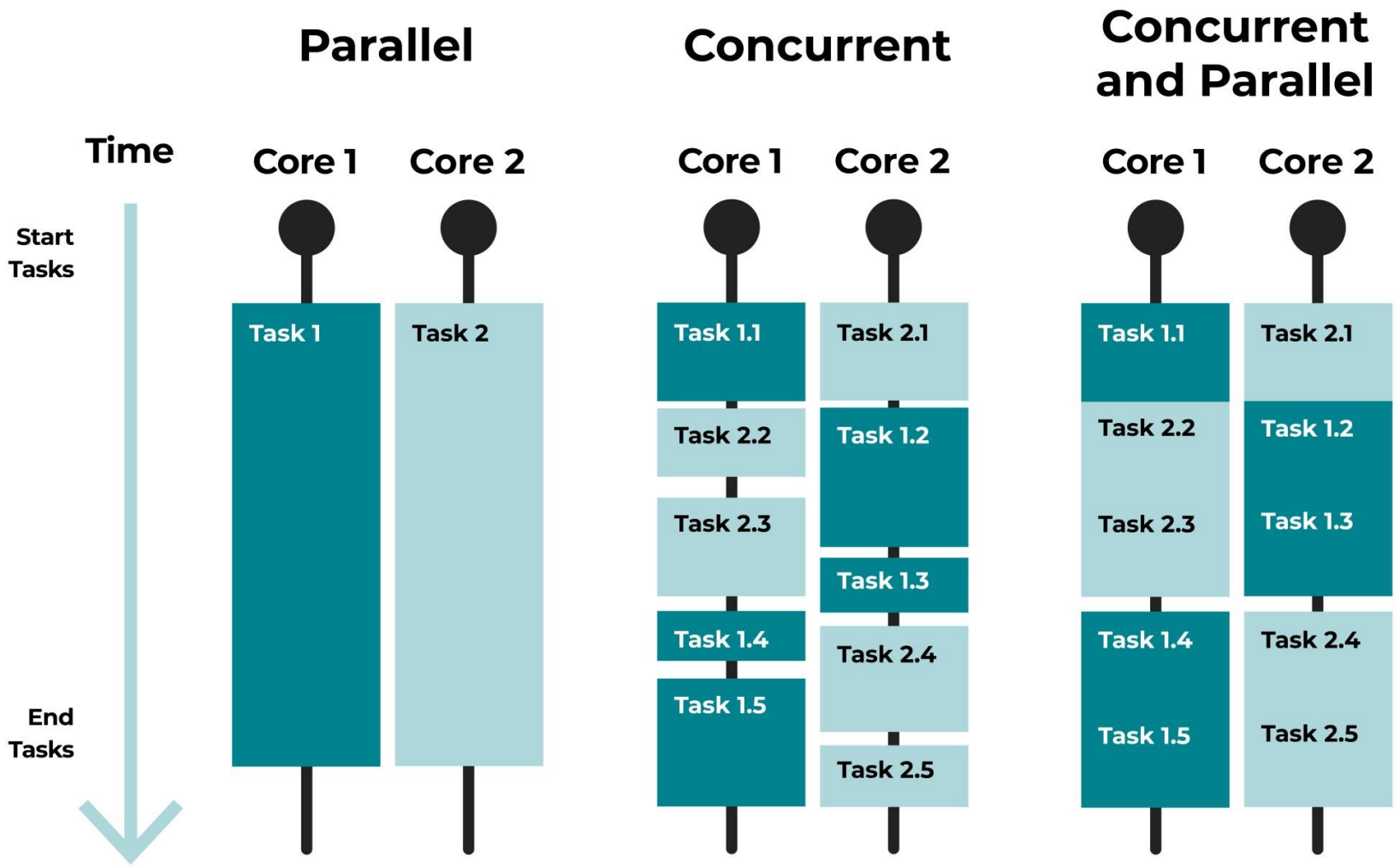
Hardware

- Devices or computers comes up with multiple cores
- Developers need to learn programming patterns to maximize the use of multiple cores
- Parallel Programming concepts
- Parallel Streams
- Threads

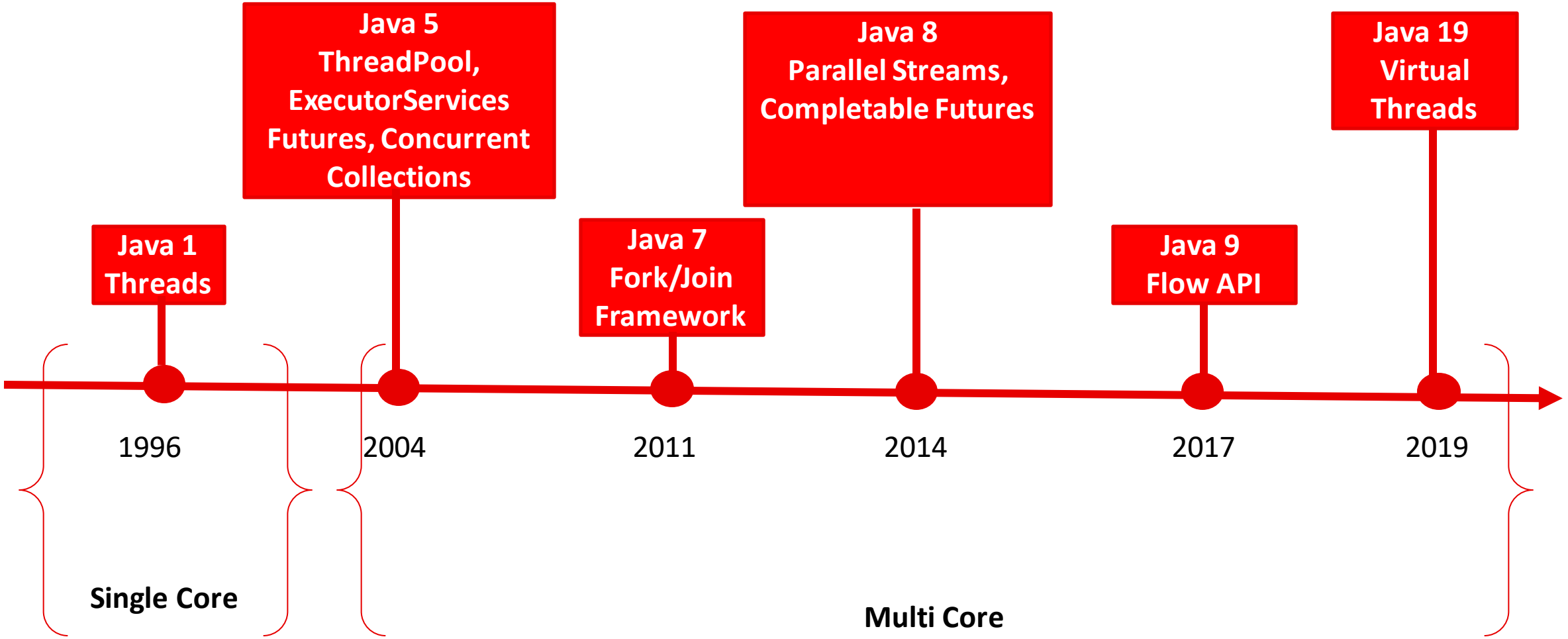
Software

- Microservices Architecture Style
- Blocking I/O calls were common in synchronous calls and impact the latency of application
- Apply Asynchronous Programming concepts
- CompletableFuture
- Functional Style Programming

Can you explain concurrency and parallelism?



Evolution of Concurrency and Parallelism in Java



Reactive Programming

- It’s a paradigm, not a language feature.
- Focuses on facilitating working with asynchronous and non-blocking components
- Solves issues that come with asynchronous programming
  - Solves callback hell
  - Simplifies concurrency in asynchronous systems
  - Abstracts low-level mechanics
- Many languages have support for it.
- Java frameworks (based on Reactive Streams specification)
  - Java 9
  - Project Reactor used by Spring framework
  - Netflix RxJava (extension of reactive extensions for C#)
  - Smallrye Mutiny used by Quarkus
  - Helidon
  - MongoDB Reactive Streams Driver
  - …etc

Reactive Programming concepts

- React-ive → The program reacts to an **event** containing a value received from somewhere.
- What can this event be?
  - The result of an HTTP request
  - A Kafka event
  - A database result
  - An item from a stream of items produced by a source
- A **stream**<> is a continuous flow of **events** that you can subscribe to and react to in different ways.
- A **subscriber** is someone who listens to events produced by a stream.
- Streams can be Hot or Cold!
  - Hot streams: events will be emitted even if there is no subscriber. Subscribers typically only get items emitted after their subscription.
  - Cold streams: events will be emitted only if there is a subscriber to a stream. Each subscriber typically receives all events separately.

Dissolving the confusion

- Asynchronous != Parallel
  - Asynchronous -> operations take place in a non-sequential fashion.
  - Parallel -> operations run at the same time in different threads.
- Asynchronous != Multithreaded
  - You can have asynchronous code running in one thread.
- Asynchronous != non-blocking
  - A method executed asynchronously **can still block a thread.**
- Multithreading comes at a cost
  - For creating and sharing memory amongst threads.
- We use Reactive interfaces because
  - Easier for us write asynchronous methods.
  - Use non-blocking i/o without resulting in callback hell!

Remember..

Reactive and non-blocking do not necessarily make applications faster.

- Non-blocking approach requires more effort but may slightly increase processing time.

Key benefit of reactive and non-blocking: scalability with fewer threads and less memory.

- Strengths of reactive stack become evident in the presence of latency.

SmallRye Mutiny

SmallRye Mutiny - Overview

- Intuitive Event-Driven Reactive Programming Library for Java.
- Conforms to Reactive Streams specification: non-blocking I/O and backpressure.
- Can be used as a standalone library, or integrated as part of other frameworks such as Quarkus.
- Supports two main reactive data types:
  - Uni (single item or failure)
  - Multi (stream of data)
  - Both are lazy! Nothing happens if no one subscribes to the results!
- Compact set of methods/ operators leading to improved readability and navigability on the API.
- Supports Uni <-> Multi transformations, and has converters for interoperability with various other reactive libraries:
  - Project Reactor
  - RxJava

Events

Event	Uni / Multi	Direction	Note
item	Uni + Multi	upstream -> downstream	The upstream sent an item.
failure	Uni + Multi	upstream -> downstream	The upstream failed.
completion	Multi	upstream -> downstream	The upstream completed.
subscribe	Uni and Multi	downstream -> upstream	A downstream subscriber is interested in the data.
subscription	Uni and Multi	upstream -> downstream	Event happening after a `subscribe` event to indicate that the upstream acknowledged the subscription.
cancellation	Uni and Multi	downstream -> upstream	A downstream subscriber does not want any more events.
overflow	Multi	upstream -> downstream	The upstream has emitted more than the downstream can handle.
request	Multi	downstream -> upstream	The downstream indicates its capacity to handle `n` items.

- Typically when programming reactive flows with Quarkus, we are mostly explicitly interacting with 'item and 'failure' events.

Typical scenario (mostly under the hood):

1. Subscriber sends subscription request to upstream/publisher, upstream/publisher responds with acknowledgement.
2. Subscriber sends 'request' event indicating its current capacity to handle incoming data.
3. Publisher starts emitting at the requested capacity.
4. Subscriber may cancel the subscription at any time, or change its indicated capacity.
5. If all possible items have been emitted, publisher issues a 'completion' event, signaling the end of the current data stream.

Some examples!

```java
Uni<String> request = networkRequest(parameters);

request
        .onItem().transform(item -> doSomeTransformation(item))
        .onFailure().transform(technicalException -> new BusinessException(technicalException))
        .subscribe()
        .with(
                item -> logger.info(item),
                failure -> logger.error("Failed with: ", failure)
        );
```

[1] Handling an async operation which produces a Uni

```java
Multi<String> anotherRequest = dataStream(parameters);

Multi<String> results = anotherRequest
        .onItem().transform(item -> doSomeTransformation(item))
        .onFailure().recoverWithCompletion();

results.subscribe().with(
        item -> logger.info(item),
        () -> logger.info("stream completed!")
);
```

[2] Handling an async operation which produces a Multi

```java
Uni<ResponseObjectA> uniA = invokeHttpServiceA();
Uni<ResponseObjectB> uniB = invokeHttpServiceB();

Uni.combine()
        .all().unis(uniA, uniB).asTuple()
        .subscribe().with(tuple -> {
                logger.info("Response from A: " + tuple.getItem1());
                logger.info("Response from B: " + tuple.getItem2());
        });
```

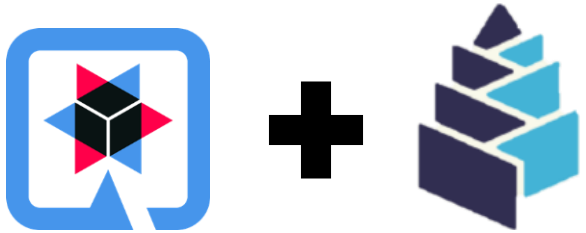
[3] Concurrent async operations via combining Unis

```java
return httpCallNumberOne()
        .onItem().transformToUni(resultOfHttpCallNo1 -> dependentHttpCallNumberTwo())
        .onFailure().recoverWithItem(r -> fallbackLogic());
```

[4] Chaining two operations
(second is dependent on completion of first)

Mutiny in Quarkus

- Mutiny is well-integrated with Quarkus
  - Included by default as a dependency / extension.
  - Quarkus utilizes the Mutiny library as part of its Reactive APIs.



https://mvnrepository.com/artifact/io.smallrye.reactive/mutiny

https://mvnrepository.com/artifact/io.quarkus/quarkus-mutiny

GETTING HELP FROM THE COMMUNITY

Documentation

We have a lot of documentation. Be sure to check our Getting started page, and all our guides. Also check out our FAQ section and Quarkus Tips Playlist

Stack Overflow

Ask your questions on Stack Overflow. After the documentation, it's probably the best place to look for answers. We actively monitor the Quarkus tag.

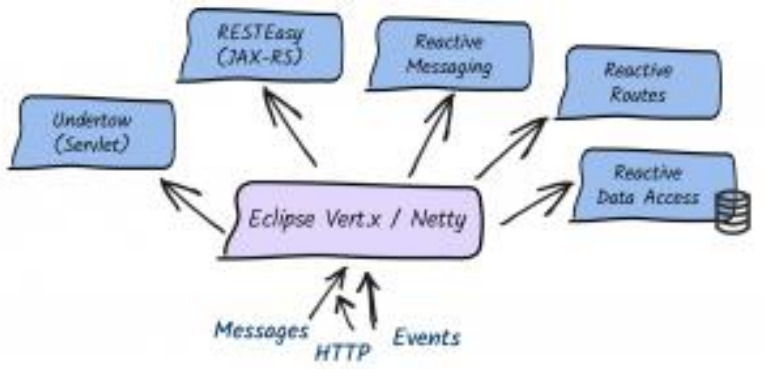
Discussions and Collaboration

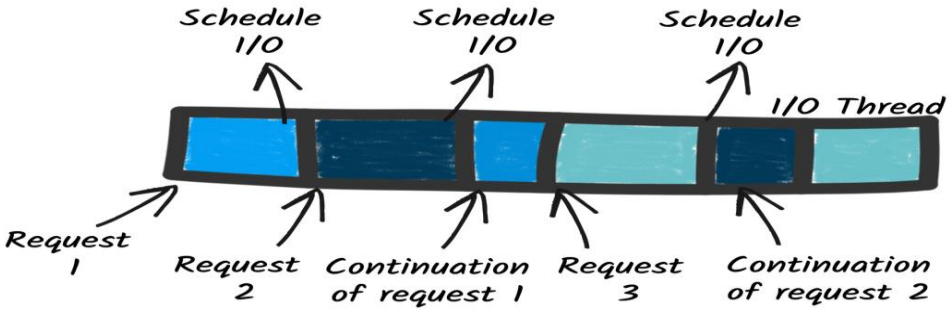
Check out our GitHub Discussions collaboration area to interact with other Quarkus users and developers.

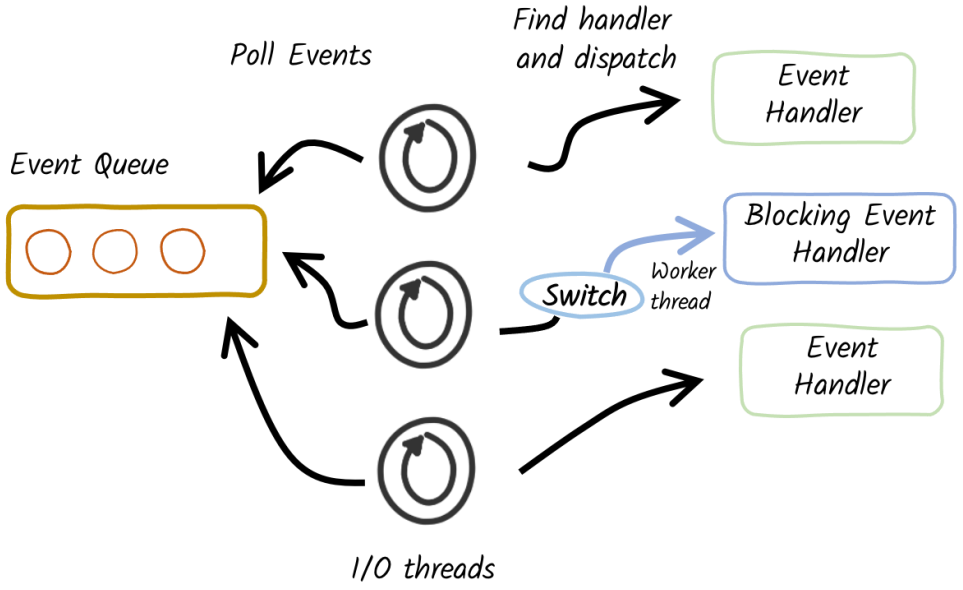
Quarkus Development

Discuss the development of Quarkus with the team in the development mailing list or by Zulip Chat.

Quarkus – Reactive in action



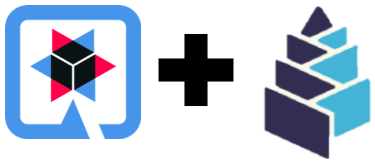




Quarkus has a reactive engine powered by Eclipse Vert.x and Netty, which handles the non-blocking I/O interactions. Quarkus uses the event-loop mechanism by default.

However, Quarkus can identify when the application logic is blocking or non-blocking, so it can switch between models of execution.

Override the default event loop! Quarkus with worker threads



```java
@Blocking
@GET
public Uni<Response> myBlockingMethod() {
    return someService.doStuff();
}
```

1. Use 'Blocking' annotation

```java
@GET
public ModelObject myBlockingMethod() {
    return someService.doImperativeBlockingStuff();
}
```

2. Avoid reactive data types & operations

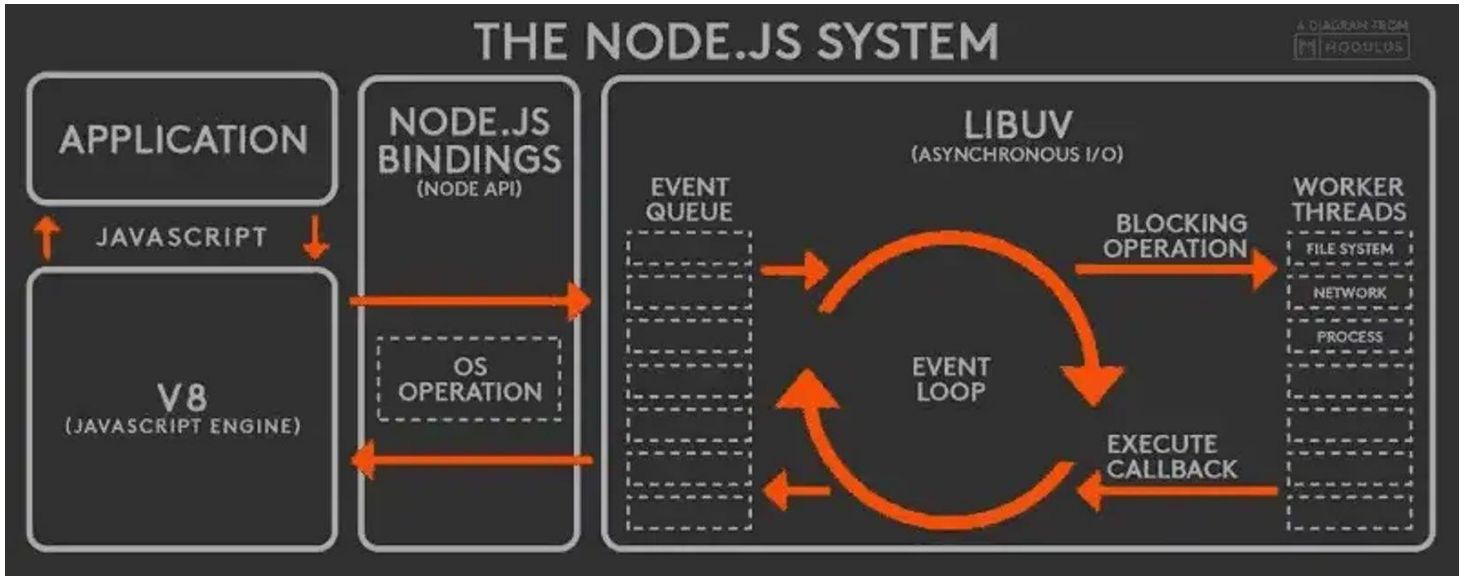
```java
return methodThatReturnsUni()
        .emitOn(Infrastructure.getDefaultWorkerPool())
        .onItem().transform(this::applySomeTransformation);
```

```java
return methodThatReturnsUni()
        .onItem().transform(this::applySomeTransformation)
        .runSubscriptionOn(Infrastructure.getDefaultWorkerPool())
        .subscribe().with(
                item -> logger.info("Item: " + item),
                Throwable::printStackTrace
        );
```

3. Use special mutiny operators (emitOn, runSubscriptionOn)

Event loop? Rings a bell...!

- Released in 2009
- Extremely fast and lightweight
- Non-blocking by default
  - Asynchronous by default
- 1 Thread can serve multiple requests
- Took the programming world by storm
- Blew Java/C# out of the water in terms of performance, and cost savings



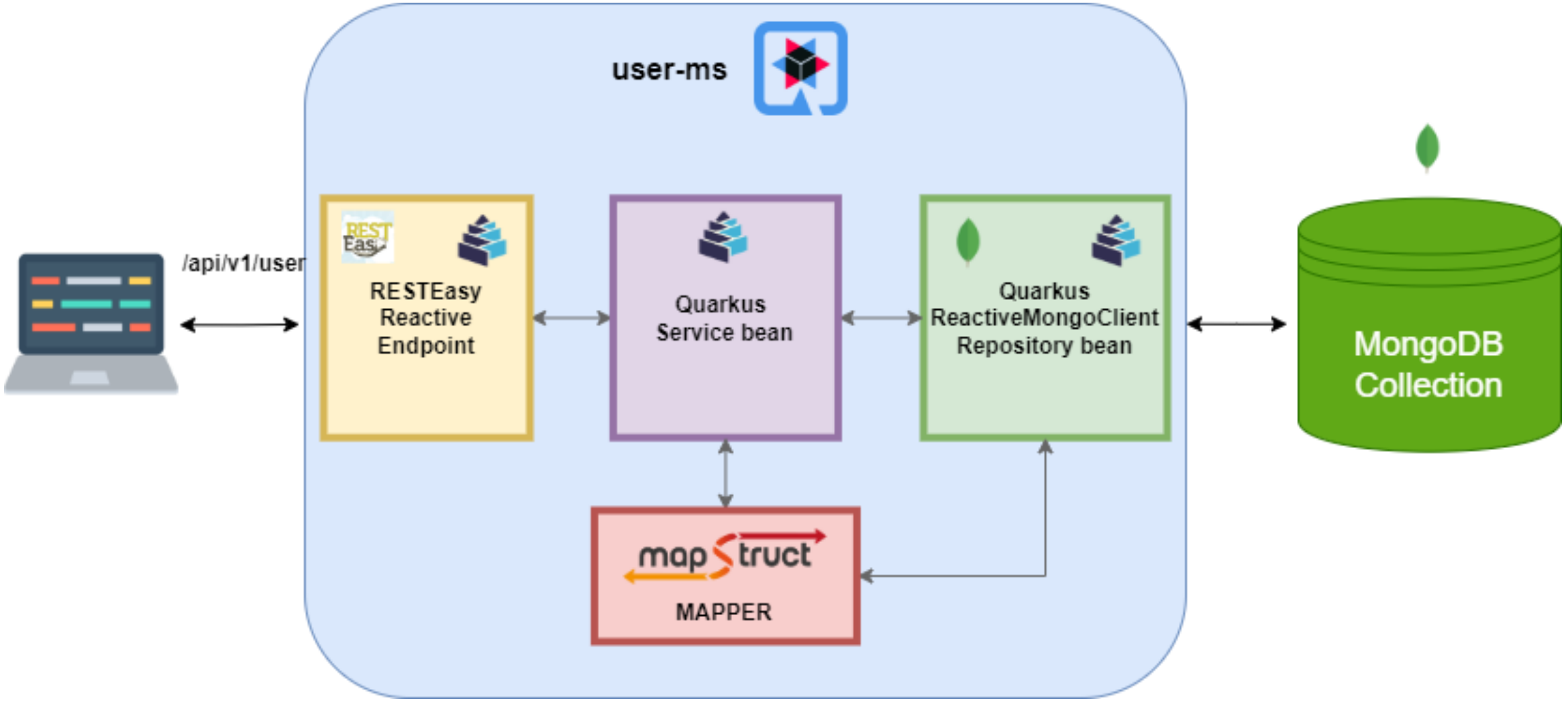
Quarkus Demo!

Quarkus Demo

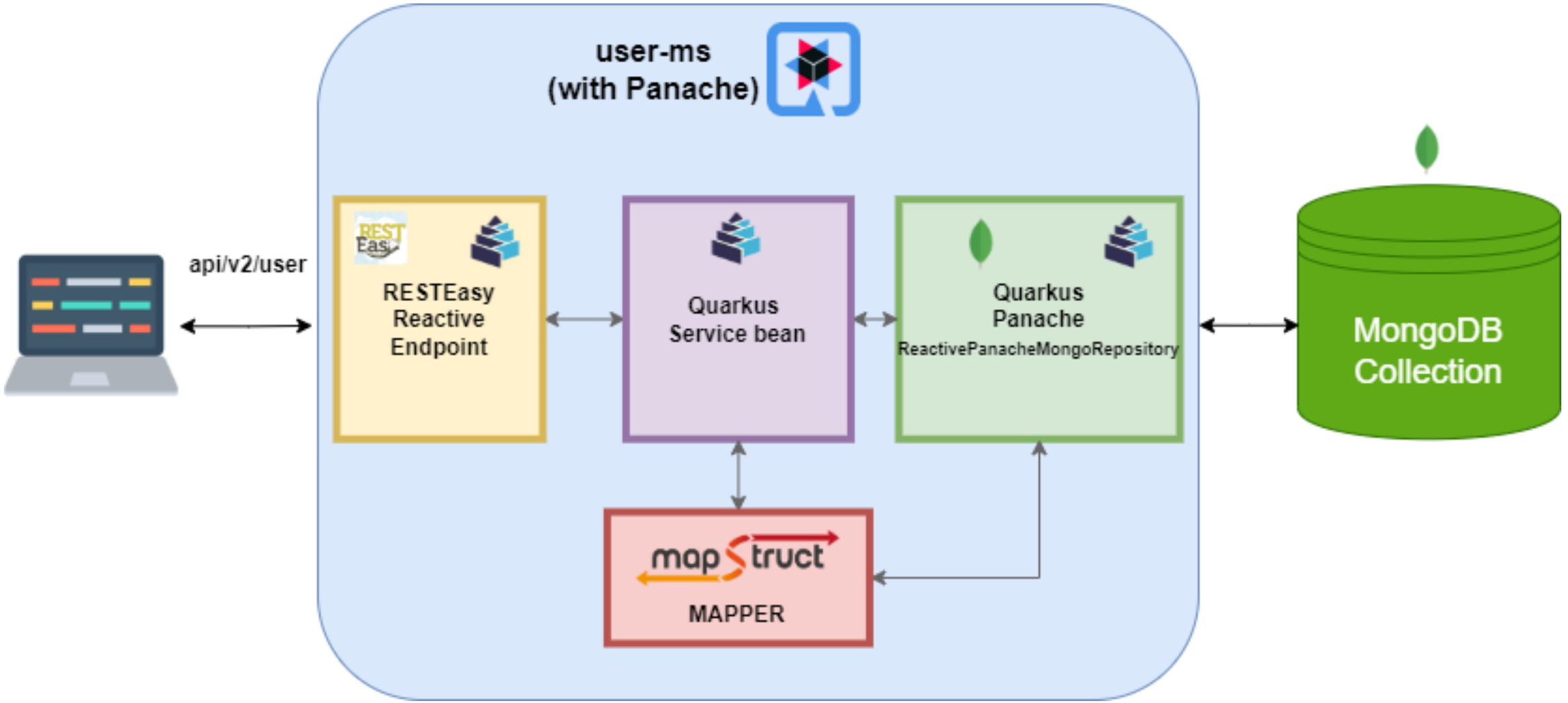
- Hypothetical microservice called **user-ms** that receives HTTP requests -> performs MongoDB operations. Data represents information of hypothetical users.
- Libraries/tools used in this Quarkus application:
  - RESTEasy Reactive
  - MapStruct
  - ReactiveMongoClient
  - Panache (ReactivePanacheMongoRepository)
  - SmallRye OpenAPI
  - and of course, SmallRye Mutiny!

GitHub Repo ***quarkus-microservice-poc***

Quarkus Demo Topology with ReactiveMongoClient



Quarkus Demo Topology with Panache



Data model

- User:
  - username
  - email
  - firstname
  - lastname
- Every distinct user is stored with a unique id in the mongodb collection (called UserInfo)
- The MongoDB documents of UserInfo also contain some helpful metadata date fields.

LINK TO DEMO REPO:

github.com/steliost03/quarkus-microservice-poc

Reactive in Spring Universe

Project Reactor Overview

- Project Reactor is a fully non-blocking reactive library with back-pressure support included.
- It’s the foundation of the reactive stack in the Spring ecosystem.
- It is based on the Reactive Streams specification.
- Simple vocabulary, through well defined operators.
- Operator-fusion aware.
- Offers 2 reactive and composable APIs
  - Flux 0 – N element(s)
  - Mono 0 – 1 element

Project Reactor in Spring Framework

- Featured in projects such as :
  - Spring WebFlux
  - Spring Data etc.
- WebFlux is a Reactor based framework.
- Netty under the hood.
- Spring base reactive core.
- Non-blocking and supports Reactive Streams back pressure.
- Includes a client to perform HTTP requests with (WebClient).

Observer Pattern

Meet the Observer Pattern

You know how newspaper or magazine subscriptions work:

1. A newspaper publisher goes into business and begins publishing newspapers.
2. You subscribe to a particular publisher, and every time there's a new edition it gets delivered to you. As long as you remain a subscriber, you get new newspapers.
3. You unsubscribe when you don't want papers anymore, and they stop being delivered.
4. While the publisher remains in business, people, hotels, airlines and other businesses constantly subscribe and unsubscribe to the newspaper.

The need...

- Efficient multicore CPU resource usage.
- Handling streams of non-finite and maybe "live" data.
- Such as:
  - Social media platforms
  - Financial trading systems
  - Internet of Things (IoT) applications
  - Real-time analytics platforms
  - Streaming media services

Reactive Streams – Publisher and Subscriber

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

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

Imperative vs Reactive [Data aspect]

A fancy balloon

```
public Item[] balloon() {
    return Stream.generate(Item::new)
            .limit( maxSize: 10)
            .toArray(Item[]::new);
}
```

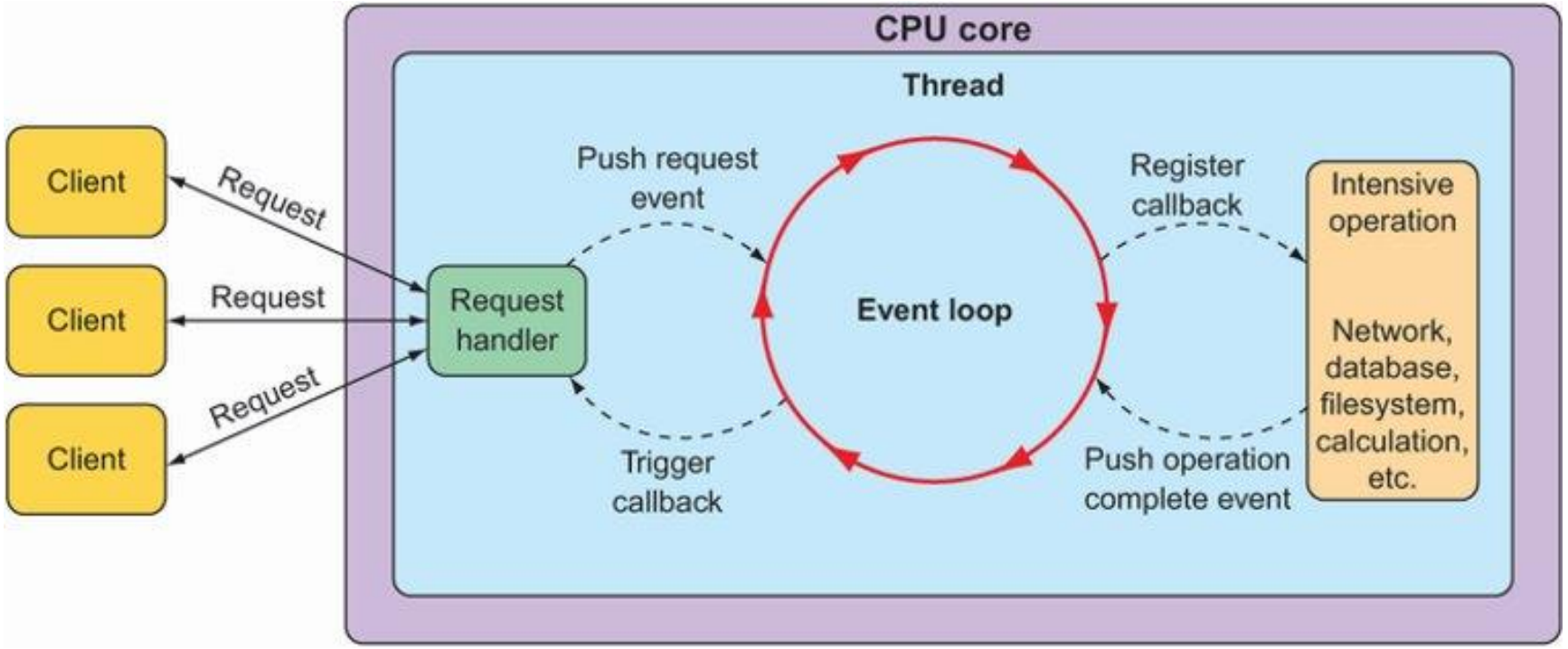
A fearless hose

```
public Flux<Item> gardenHose() {
    return Flux.fromStream(
            Stream.generate(Item::new)
            .limit( maxSize: 10)
    ).delayElements(Duration
            .of( amount: 1, ChronoUnit.SECONDS));
}
```

Spring WebFlux – Concurrency Model

- Use of Netty. Netty is a non-blocking I/O client-server framework

- Non-blocking servers use a small, fixed-size thread pool (event loop workers) to handle requests.



Reactive [Request Aspect]

`Schedulers` method	Description
`.immediate()`	Executes the subscription in the current thread.
`.single()`	Executes the subscription in a single, reusable thread. Reuses the same thread for all callers.
`.newSingle()`	Executes the subscription in a per-call dedicated thread.
`.elastic()`	Executes the subscription in a worker pulled from an un-bounded, elastic pool. New worker threads are created as needed, and idle workers are disposed of (by default, after 60 seconds).
`.parallel()`	Executes the subscription in a worker pulled from a fixed-size pool, sized to the number of CPU cores.

LINK TO DEMO REPO:

user-mongo repo

simple example

simple example consumer

Coding session

Q&A


thank you!