Vert.x 3.4 - Kotlin 1.1

Fuga dal callback-hell

Come programmare un multi-reattore e vivere felici

Francesco Vasco

Eclipse Vert.x

- Componenti: vertici
- Event Bus
- Multi Reactor Pattern
- Poliglotto
  - Java
  - JavaScript
  - Groovy
  - Ruby
  - Ceylon
  - Kotlin
  - Scala

Core
Web
Web Client
Data access
Integration
Event Bus bridges
Authentication and Authorisation
Reactive
Microservices
IoT
Devops
Testing
Clustering
Services
Cloud

Eclipse Vert.x

Vert.x Rx

Don't like callback-style APIs? Vert.x provides *Rx-ified* (using RxJava) versions for most of its APIs so you can use those if you prefer.

RxJava is a great choice when you want to perform complex operations on multiple asynchronous streams of data.

Reactive streams

Vert.x supports reactive streams so your applications can interoperate with other reactive systems such as Akka or Project Reactor.

Java Manual API

Source

Vert.x Sync

Vertx-sync allows you to deploy verticles that run using fibers. Fibers are very lightweight threads that can be blocked without blocking a kernel thread. This enables you to write your verticle code in a familiar synchronous style.

Java Manual API

Source | Examples

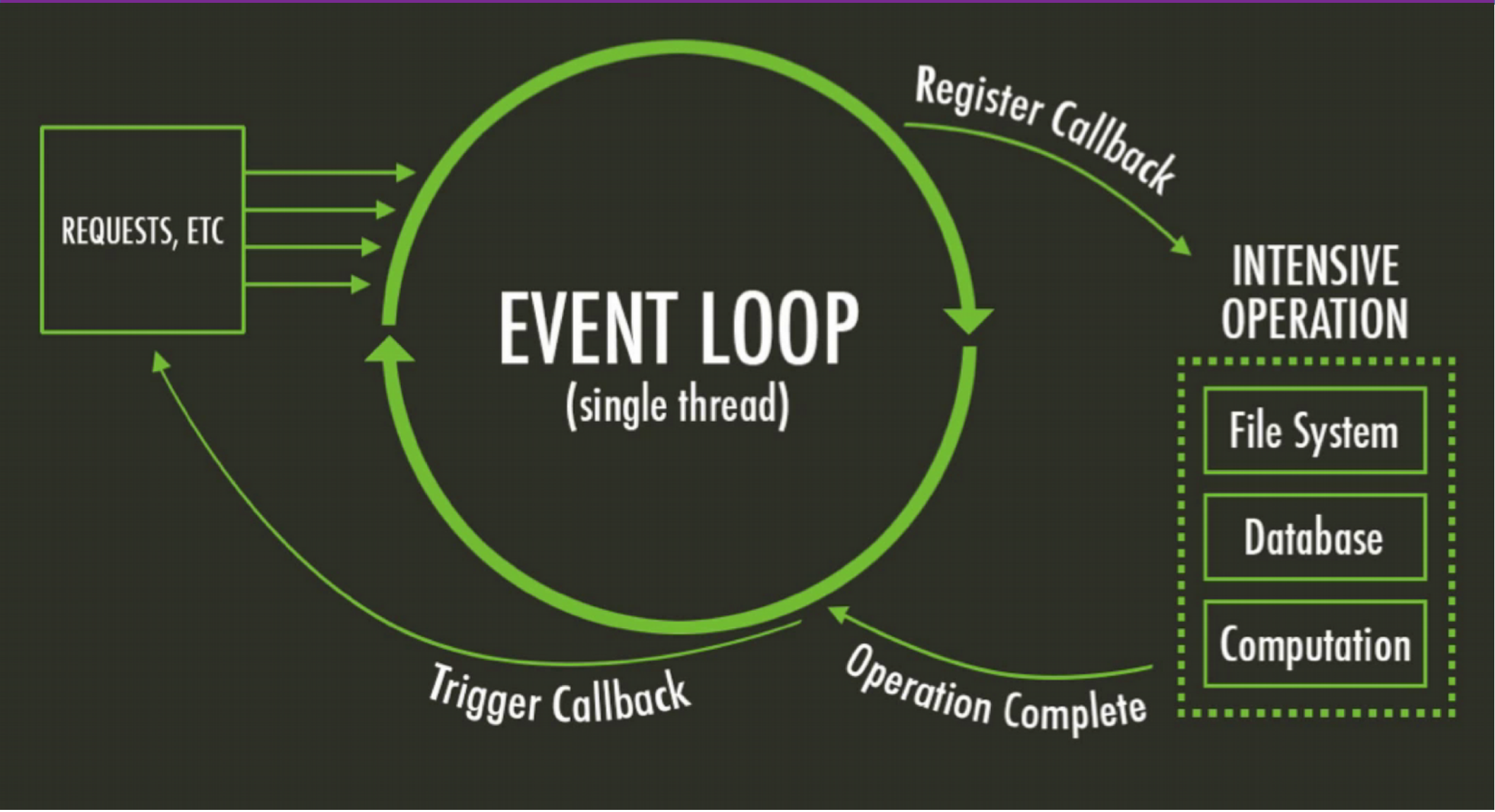
Event Bus

- Consumer
- Publish / Send

```
vertx
        .eventBus()
        .consumer<String>("messageBus")
        .bodyStream()
        .handler(::println)

vertx
        .eventBus()
        .publish("messageBus", "hello")
```

Multi Reactor Pattern



Multi Reactor Pattern

The Golden Rule - Don’t Block the Event Loop

```
print("Working...")
Thread.sleep(1000)
println(" completed")
```

```
long timerID = vertx.setTimer(1000, id -> {
  System.out.println("And one second later this is printed");
});

System.out.println("First this is printed");
```

10 million lines in 8132 repositories

Kotlin is equally strong with server-side and Android developers (roughly 50/50 divide). Spring Framework 5.0 has introduced Kotlin support, so did vert.x 3.4. Gradle and TeamCity are using Kotlin for build scripts. More projects using Kotlin can be found at kotlin.link.

Many well-known companies are using Kotlin: Pinterest, Coursera, Netflix, Uber, Square, Trello, Basecamp, amongst others. Corda, a distributed ledger developed by a consortium of well-known banks (such as Goldman Sachs, Wells Fargo, J.P. Morgan, Deutsche Bank, UBS, HSBC, BNP Paribas, Société Générale), has over 90% Kotlin in its codebase.

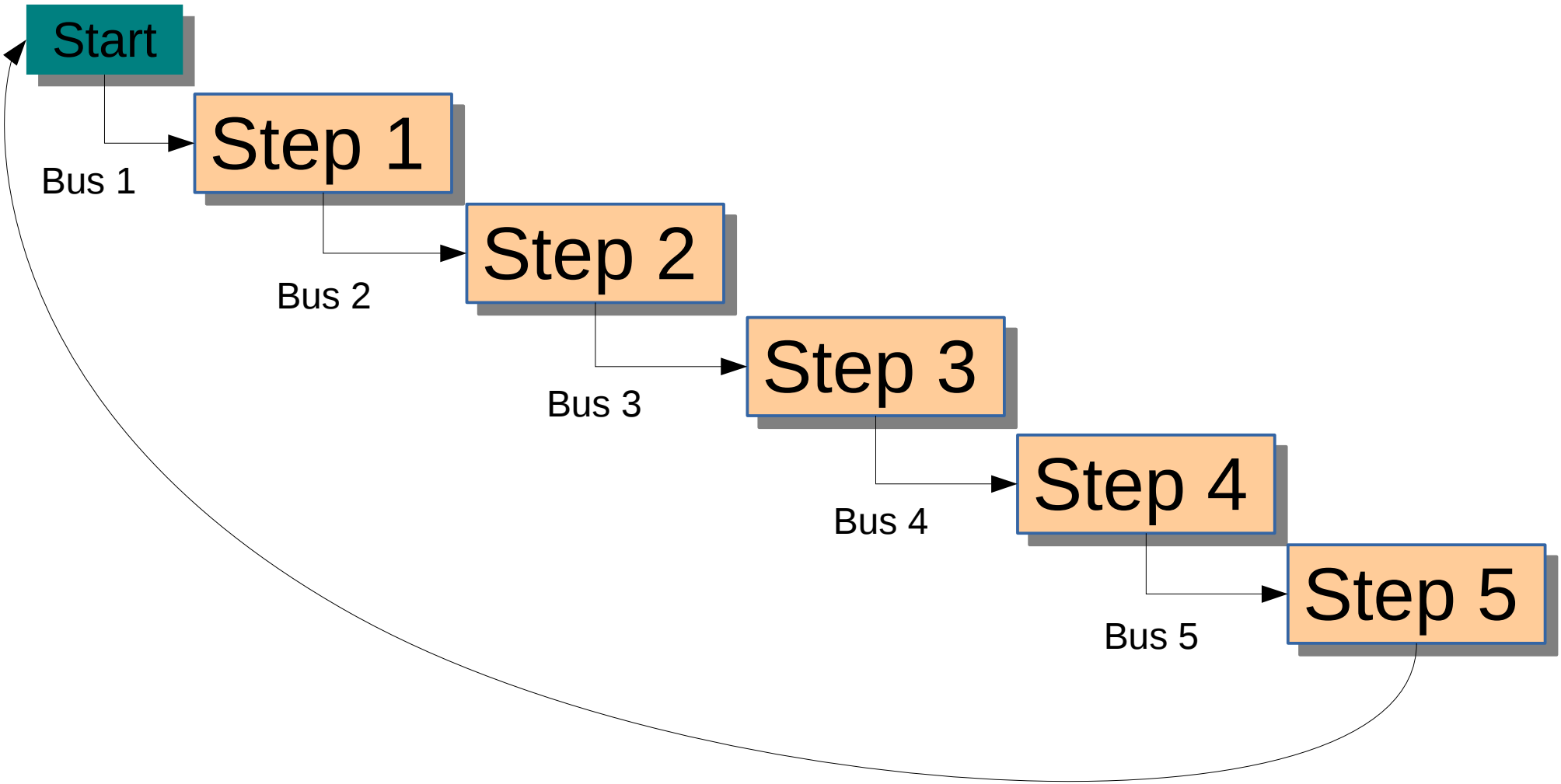
JavaScript support

All language features in Kotlin 1.1
work on both JVM/Android and JavaScript

Coroutine experimental support

Kotlin supports suspending functions
on both JVM/Android and JavaScript

Workflow


Start
Step 1
Bus 1
Step 2
Bus 2
Step 3
Bus 3
Step 4
Bus 4
Step 5
Bus 5

Step

```
fun doWork(workStep: WorkStep,
           description: String,
           vertx: Vertx,
           handler: Handler<String>) {
    val millis = ThreadLocalRandom.current().nextLong(2000)
    vertx.setTimer(millis) {
        val result = "$description ($workStep $millis ms)"
        handler.handle(result)
    }
}
```

Step 1

1)On *message*

2)Do *work*

3)Send *work result* to *next step bus*

4)Return *next step result*

```
final EventBus eventBus = vertx.eventBus();
eventBus
        .<String>consumer(UtilKt.getBusName(WorkStep.STEP1)).handler(
                workRequestMessage ->
                        UtilKt.doWork(WorkStep.STEP1, workRequestMessage.body(), vertx,
                                workResult ->
                                        eventBus.send(UtilKt.getBusName(WorkStep.STEP2), workResult,
                                                asyncMessageResult ->
                                                        workRequestMessage.reply(asyncMessageResult.result().body())
                                        )
                        )
);
```

1 2 3 4

Step 2

```
override fun start() {
    vertx.eventBus()
            .consumer<String>(WorkStep.STEP2.busName)
            .handler(WorkRequestHandler(vertx))
}

private class WorkRequestHandler(val vertx: Vertx) : Handler<Message<String>> {
    override fun handle(requestMessage: Message<String>) {
        doWork(WorkStep.STEP2, requestMessage.body(), vertx, WorkResultHandler(vertx, requestMessage))
    }
}

private class WorkResultHandler(val vertx: Vertx,
                                val requestMessage: Message<String>) : Handler<String> {
    override fun handle(result: String) {
        vertx.eventBus().send(WorkStep.STEP3.busName, result, NextStepResultHandler(requestMessage))
    }
}

private class NextStepResultHandler(val requestMessage: Message<String>) : Handler<AsyncResult<Message<String>>> {
    override fun handle(event: AsyncResult<Message<String>>) {
        val result = event.result().body()
        requestMessage.reply(result)
    }
}
```

Step 3

```
override fun start() {
    val eventBus = vertx.eventBus()
    val messageConsumer = eventBus.consumer<String>(STEP3.busName)
    messageConsumer.handler(this::handleWorkRequest)
}

private fun handleWorkRequest(requestMessage: Message<String>) {
    val eventBus = vertx.eventBus()
    launch {
        try {
            val jobDescription = requestMessage.body()

            val workMessage = handle<String> { handler ->
                doWork(STEP3, jobDescription, vertx, handler)
            }

            val result = handleResult<Message<String>> { handler ->
                eventBus.send(STEP4.busName, workMessage, handler)
            }

            requestMessage.reply(result.body())
        } catch(e: Exception) {
            requestMessage.fail(0, e.message)
        }
    }
}
```

Delimited continuation

In programming languages,
a delimited continuation,
composable continuation
or partial continuation,
is a "slice" of a continuation frame
that has been reified into a function.

Unlike regular continuations,
delimited continuations return a value,
and thus may be reused and composed.

https://en.wikipedia.org/wiki/Delimited_continuation

Continuation

```
try {
  // … (1)
  val result = doWork(…)
  // … (2)
} catch (exception) {
  // … (3)
}
```

```
interface Continuation<in T> {

    fun resume(result: T)

    fun resumeWithException(exception: Throwable)
}
```

Coroutine

1)Avvio (*startCoroutine*)

2)Suspensione/ripresa (suspend/resume)

3)Terminazione

```
final EventBus eventBus = vertx.eventBus();
eventBus
        .<String>consumer(UtilKt.getBusName(WorkStep.STEP1)).handler(

        workRequestMessage ->
                UtilKt.doWork(WorkStep.STEP1, workRequestMessage.body(), vertx,

                        workResult ->
                                eventBus.send(UtilKt.getBusName(WorkStep.STEP2), workResult,

                                        asyncMessageResult ->
                                                workRequestMessage.reply(asyncMessageResult.result().body())
                                )
                )
);
```

Creare una coroutine

```
val future = launch {
    // body
}
```

```
private class FutureContinuation<T>(private val future: Future<T>,
                                    override val context: CoroutineContext) : Continuation<T> {

    override fun resume(value: T) = future.complete(value)

    override fun resumeWithException(exception: Throwable) = future.fail(exception)
}


fun <T> launch(block: suspend () -> T): Future<T> {
    val future = Future.future<T>()
    val futureContinuation = FutureContinuation<T>(future, EmptyCoroutineContext)
    block.startCoroutine(completion = futureContinuation)
    return future
}
```

Sospendere una coroutine

handle { handler → delay(1000, handler) }

```
suspend fun <T> handle(block: (Handler<T>) -> Unit): T =
        suspendCoroutine { cont: Continuation<T> ->
            // handler calls `resume`
            val handler = Handler<T> { cont.resume(it) }
            block(handler)
        }
```

val user = **handleResult** { handler → loadUser(id, handler) }

```
suspend fun <T> handleResult(block: (Handler<AsyncResult<T>>) -> Unit): T =
        suspendCoroutine { cont: Continuation<T> ->
            val handler =
                    Handler<AsyncResult<T>> { asyncResult ->
                        if (asyncResult.succeeded())
                            cont.resume(asyncResult.result())
                        else
                            cont.resumeWithException(asyncResult.cause())
                    }
            block(handler)
        }
```

Step 3

```
override fun start() {
    val eventBus = vertx.eventBus()
    val messageConsumer = eventBus.consumer<String>(STEP3.busName)
    messageConsumer.handler(this::handleWorkRequest)
}

private fun handleWorkRequest(requestMessage: Message<String>) {
    val eventBus = vertx.eventBus()
    launch {
        try {
            val jobDescription = requestMessage.body()

            val workMessage = handle<String> { handler ->
                doWork(STEP3, jobDescription, vertx, handler)
            }

            val result = handleResult<Message<String>> { handler ->
                eventBus.send(STEP4.busName, workMessage, handler)
            }

            requestMessage.reply(result.body())
        } catch(e: Exception) {
            requestMessage.fail(0, e.message)
        }
    }
}
```

Step 4

```
class Verticle4 : AbstractVerticle() {
    override fun start() {
        val eventBus = vertx.eventBus()
        launch {
            val messageConsumer = eventBus.consumer<String>(WorkStep.STEP4.busName)
            messageConsumer.forEach { requestMessage ->
                try {
                    val description = requestMessage.body()

                    val workMessage = handle <String> { handler ->
                        doWork(WorkStep.STEP4, description, vertx, handler)
                    }

                    val result = handleResult<Message<String>> { handler ->
                        eventBus.send(WorkStep.STEP5.busName, workMessage, handler)
                    }

                    requestMessage.reply(result.body())
                } catch(e: Exception) {
                    requestMessage.fail(0, e.message)
                }
            }
        }
    }
}
```

For Each

```
suspend fun <T> ReadStream<T>.forEach(block: suspend (T) -> Unit) {
    val stream = this
    suspendCoroutine { cont: Continuation<Unit> ->

        stream.endHandler { cont.resume(Unit) }
        stream.exceptionHandler { cont.resumeWithException(it) }

        stream.handler { handler ->
            pause()
            val future = launch { block(handler) }
            future.setHandler { asyncResult ->
                resume()

                if (!asyncResult.succeeded()) {
                    // on error remove handlers
                    stream.handler(null)
                    stream.exceptionHandler(null)
                    stream.endHandler(null)
                    cont.resumeWithException(asyncResult.cause())
                }
            }
        }
```

Step 5

```
override fun start() {
    val eventBus = vertx.eventBus()
    eventBus.launchConsumer<String, String>(WorkStep.STEP5.busName) { requestMessage ->
        val description = requestMessage.body()
        val result: String = handle { handler ->
            doWork(WorkStep.STEP5, description, vertx, handler)
        }

//          val result = handleResult<Message<String>> { handler ->
//              eventBus.send(WorkStep.STEP6.busName, workMessage, handler)
//          }

        return@launchConsumer result
    }
}
```

Suspending Consumer

```
fun <Req, Res> EventBus.launchConsumer(address: String,
                                       block: suspend (Message<Req>) -> Res): MessageConsumer<Req> {
    val consumer = consumer<Req>(address)
    launch {
        consumer.forEach { message ->
            try {
                val res = block(message)
                message.reply(res)
            } catch(e: Exception) {
                message.fail(0, e.message)
            }
        }
    }
    return consumer
}
```

kotlinx.coroutines

- Job
- Future
- Generatori
- Mutex
- Pipeline
- Channel
- Selector
- JDK8 / NIO / Swing / JavaFX
  Reactive / RxJava1 / RxJava2

Job, mutex

```
var count = 0

val mutex = Mutex()

fun main(vararg args: String) = runBlocking {
    val jobs = mutableListOf<Job>()
    repeat(10) {
        jobs += async(CommonPool) {
            delay(ThreadLocalRandom.current().nextLong(1000), TimeUnit.MILLISECONDS)
            mutex.withMutex {
                count++
            }
        }
    }

    repeat(10) {
        jobs += async(CommonPool) {
            delay(ThreadLocalRandom.current().nextLong(1000), TimeUnit.MILLISECONDS)
            mutex.withMutex {
                count--
            }
        }
    }

    jobs.forEach { it.join() }
    println("Result $count")
}
```

Actor, producer, pipeline, channel

```
val printer = actor<String>(CommonPool) {
    for (text in channel)
        println(text)
}

fun numbers(from: Int) = produce<Int>(CommonPool) {
    for (n in from..Integer.MAX_VALUE) {
        send(n)
    }
}

fun filterNotDivisibleBy(divisor: Int, numbers: ReceiveChannel<Int>) = produce<Int>(CommonPool) {
    for (n in numbers) {
        if (n % divisor != 0) send(n)
    }
}

fun primes() = produce<Int>(CommonPool) {
    var primes = numbers(2)
    while (true) {
        val prime = primes.receive()
        send(prime)
        primes = filterNotDivisibleBy(prime, primes)
    }
}

fun main(vararg args: String) = runBlocking {
    for (n in primes()) {
        printer.send("Prime $n")
    }
}
```

Grazie

http://vertx.io/docs/vertx-core/kotlin/

https://github.com/Kotlin/kotlin-coroutines/blob/master/kotlin-coroutines-informal.md

https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md