Reactive Spring

```
$ git clone https://github.com/fvasco/reactive-spring.git

$ cd reactive-spring

$ ./gradlew bootRun &

$ time curl http://localhost:8080/configuration
```

Reactor 3.1 - Spring Boot 2

Reactive Spring

Francesco Vasco

Spring Boot

- Permette di create una applicazione web eseguibile basata su Spring Framework
- Incorpora Tomcat, Jetto o Undertow (non serve creare il WAR)
- Non richiede generazione del codice
- Si personalizza con annotazioni o con un DSL specifico per Kotlin (non richiede XML)

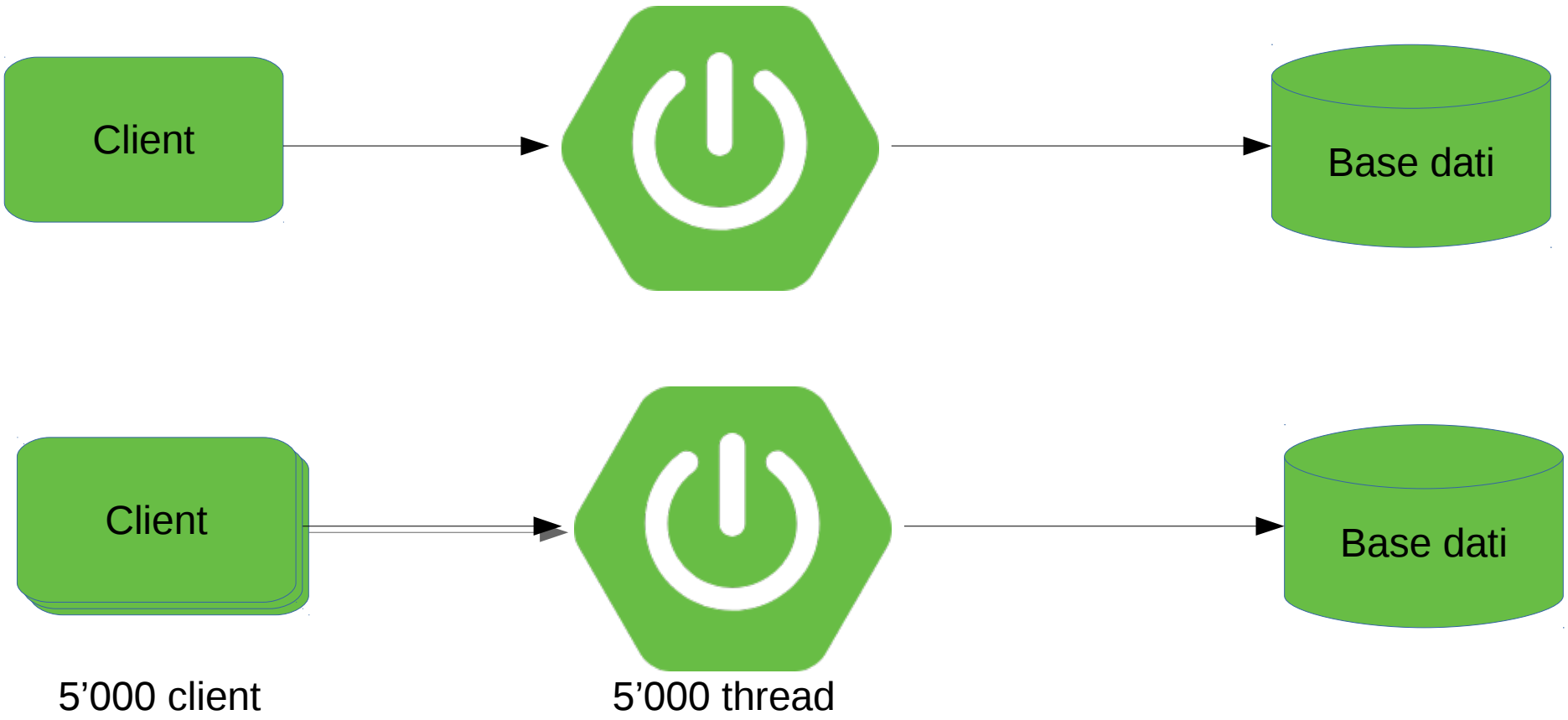
Spring Boot

```
@Controller
@EnableAutoConfiguration
class SampleController {

    @RequestMapping( …value: "/")
    @ResponseBody
    fun home() = "Hello World!"
}

fun main(args: Array<String>) {
    SpringApplication.run(SampleController::class.java, *args)
}
```

Spring Boot

```
class SimpleService {

    fun now() = Instant.now()

    fun optional() = Optional.of( value: "hello")

    fun list() = Arrays.asList("java", "util", "List")

    fun future() = CompletableFuture.completedFuture( value: "hello")
}
```

Multi thread

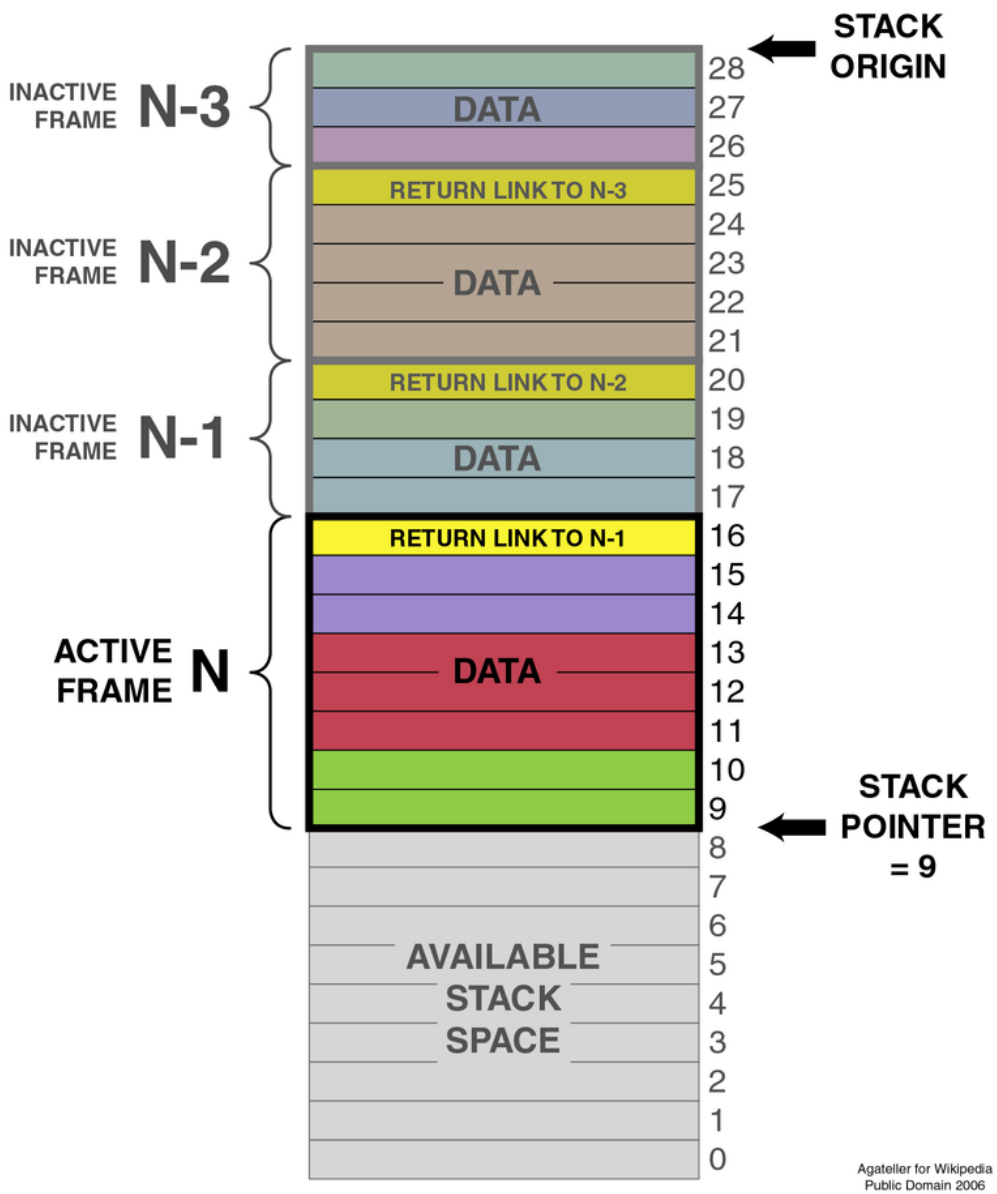


Ma anche

NO

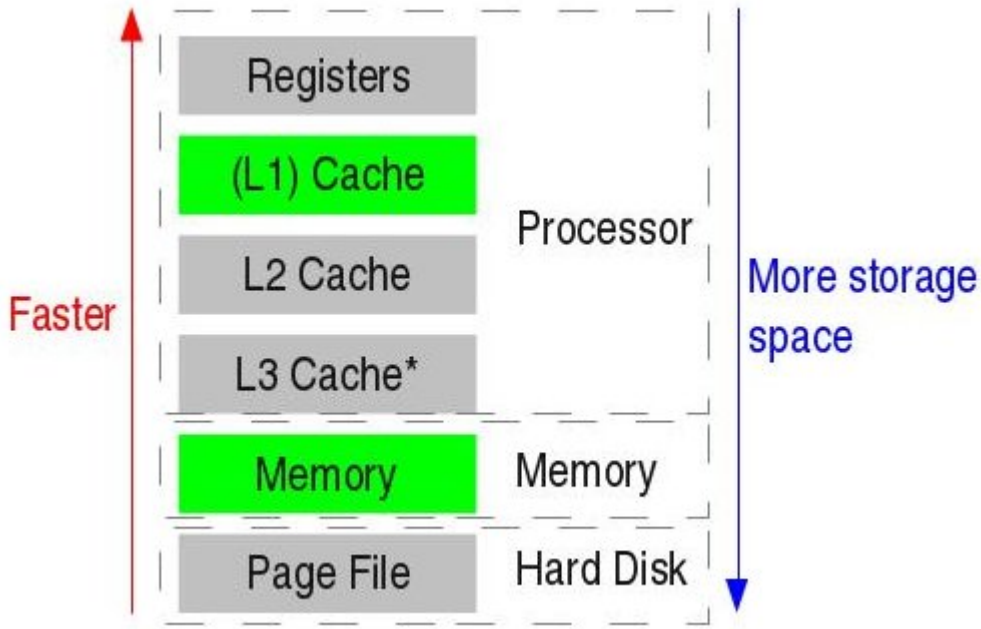
Stack size

- Ogni thread richiede un proprio stack
- La dimensione predefinita dello stack è di 1024kB



Context switch

- Per gestire più thread il processore deve, periodicamente, cambiare il contesto di esecuzione
- L'elaborazione del nuovo thread potrebbe richiedere aree di memoria non presenti nella cache del processore



Context switch

How long does it take to make a context switch? (tl;dr answer: very expensive)
http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html

Not Your Father's Complexity - Maurice Naftalin

Soluzione

Eseguire tutte l’elaborazione in parallelo

1) utilizzando pochi thread

2) senza forzare il context switch

ForkJoinPool

- Ogni thread possiede una propria coda di task da eseguire
- Ogni thread accoda i nuovi task sulla propria coda
- Un thread senza task può *rubare* dei task da un altro thread

Programmazione bloccante / non bloccante

	Bloccante	Non bloccante
Stile	Diretto	Passaggio delle continuazione (CPS)
Concorrenza	Multi-Thread	*Multiplexed* Thread
Sincronizzazione	Lock Spinlock	Coda di task
Notifica eventi	Polling	Listener
Comunicazione	Coda bloccante	Flux (*)

Valori di ritorno	Object	CompletableFuture Deferred (**)
	Optional	Mono (*)
	Collection	Flux (*)

* Progetto Reactor
** kotlinx.coroutines

Valori

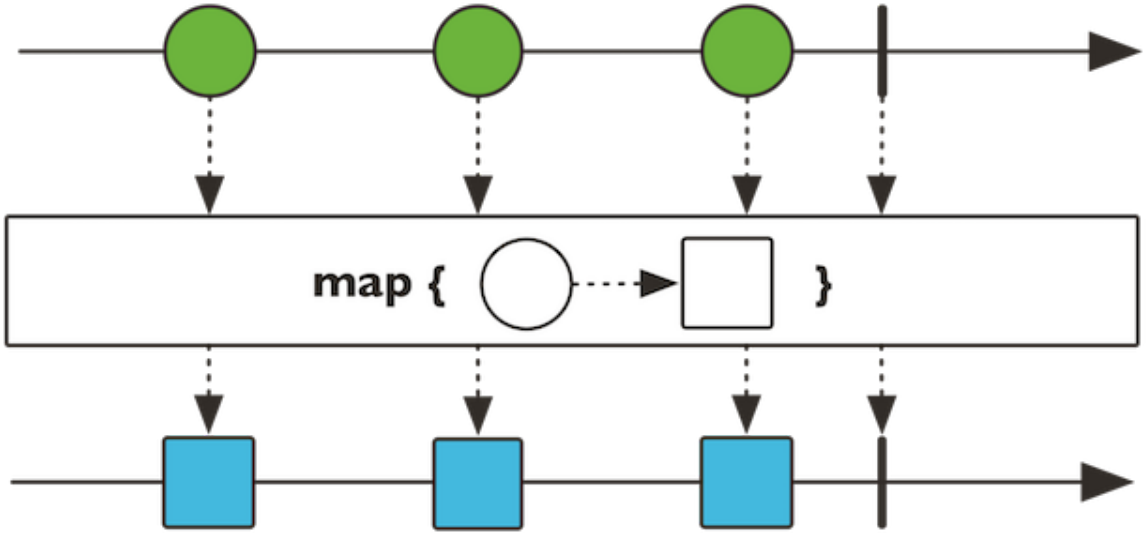
```
class MonoService {

    fun single() = "hello"

    fun optional() = Optional.of( value: "hello")

    fun future() = CompletableFuture.completedFuture( value: "hello")

    fun emptyMono() = Mono.empty<String>()

    fun mono() = Mono.just( data: "hello")

    fun futureGenerator() = future { "hello" }

    fun emptyMonoGenerator() = mono { null }

    fun monoGenerator() = mono { "hello" }
}
```

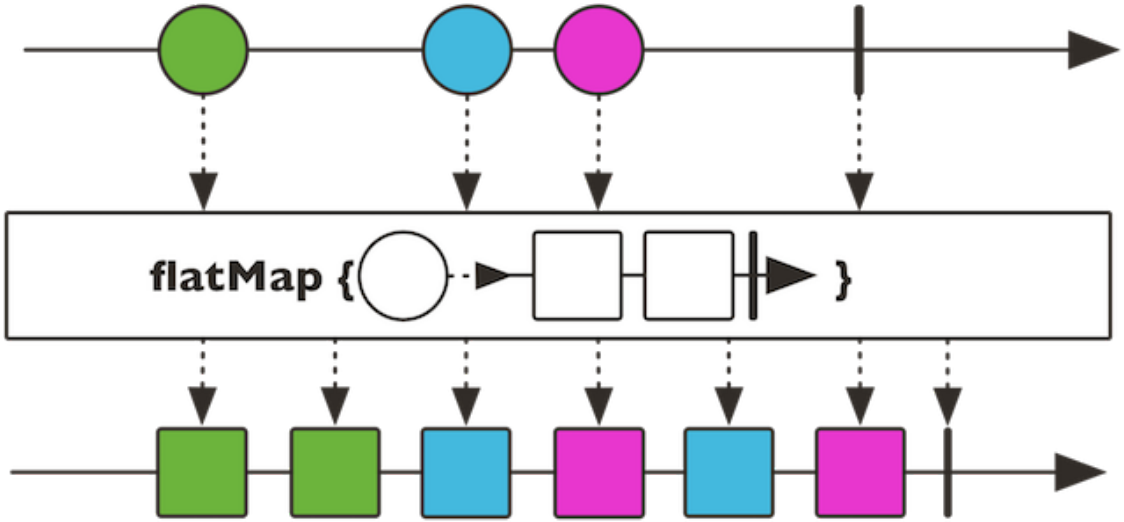
Collezioni

```
class FluxService {

    fun collection() = Arrays.asList(1, 2, 3)

    fun stream() = Stream.of( ...values: 1, 2, 3)

    fun flux() = Flux.just( ...data: 1, 2, 3)

    fun emptyFlux() = Flux.empty<Int>()

    fun fluxGenerator() = flux {
        for (i:Int in 1..3)
            send(i)
    }
}
```


CAUTION
SLIPPERY
WHEN
WET

Operatori di Flux





Stack trace in asincrono

```kotlin
fun main(args: Array<String>) {
    val mono: Mono<Unit> = mono {
        System.err.println("--> mono")
        Thread.dumpStack()
        subroutine()
    }
    mono.block()
}

suspend fun subroutine() {
    System.err.println("--> subroutine1")
    Thread.dumpStack()
    yield()
    System.err.println("--> subroutine2")
    Thread.dumpStack()
}
```

```
--> mono
java.lang.Exception: Stack trace
    at java.lang.Thread.dumpStack(Thread.java:1336)
    at example.MonoStackTraceKt$main$mono$1.doResume(MonoStackTrace.kt:9)
    at kotlin.coroutines.experimental.jvm.internal.CoroutineImpl.resume(CoroutineImpl.kt:54)
    at kotlinx.coroutines.experimental.DispatchTask.run(CoroutineDispatcher.kt:123)
    at java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1402)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
    at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1056)
    at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1692)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:157)
--> subroutine1
java.lang.Exception: Stack trace
    at java.lang.Thread.dumpStack(Thread.java:1336)
    at example.MonoStackTraceKt.subroutine(MonoStackTrace.kt:17)
    at example.MonoStackTraceKt$main$mono$1.doResume(MonoStackTrace.kt:10)
    at kotlin.coroutines.experimental.jvm.internal.CoroutineImpl.resume(CoroutineImpl.kt:54)
    at kotlinx.coroutines.experimental.DispatchTask.run(CoroutineDispatcher.kt:123)
    at java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1402)
    at java.util.concurrent.ForkJoinTask.doExec(ForkJoinTask.java:289)
    at java.util.concurrent.ForkJoinPool$WorkQueue.runTask(ForkJoinPool.java:1056)
    at java.util.concurrent.ForkJoinPool.runWorker(ForkJoinPool.java:1692)
    at java.util.concurrent.ForkJoinWorkerThread.run(ForkJoinWorkerThread.java:157)
--> subroutine2
java.lang.Exception: Stack trace
    at java.lang.Thread.dumpStack(Thread.java:1336)
    at example.MonoStackTraceKt.subroutine(MonoStackTrace.kt:20)
    at example.MonoStackTraceKt$subroutine$1.doResume(MonoStackTrace.kt)
    at kotlin.coroutines.experimental.jvm.internal.CoroutineImpl.resume(CoroutineImpl.kt:54)
    at kotlin.coroutines.experimental.jvm.internal.CoroutineImpl.resume(CoroutineImpl.kt:53)
    at kotlinx.coroutines.experimental.DispatchTask.run(CoroutineDispatcher.kt:123)
    at java.util.concurrent.ForkJoinTask$RunnableExecuteAction.exec(ForkJoinTask.java:1402)
```

Parallelismo in asincrono

```
@RestController
class FlatMap {
    @GetMapping( ...value: "example/get1")
    fun get1() : Mono<Int> =
            getValue( id: 1)
```

200 GET get1 localhos... docu... vnd.m... 129 B 1 B → 1010 ms

```
    @GetMapping( ...value: "example/getAll")
    fun getAll() : ParallelFlux<Int!> =
            getIds()
                    .parallel()
                    .flatMap { id -> getValue(id) }
}
```

200 GET getAll localhos... docu... vnd.m... 3,92 kB 3,80 kB → 1066 ms

```
fun getValue(id: Int) : Mono<Int> = mono {
    delay( time: 1, TimeUnit.SECONDS)
    return@mono id
}

fun getIds() : Flux<Int> = flux {
    repeat( times: 1000) { i ->
        send(i)
    }
}
```

Combinazione asincrona

```
@RestController
class Combine {
    @GetMapping( ...value: "example/helloWorld")
    fun helloWorld(): Mono<String> {
        val hello : Mono<String>  = getHello()
        return getWorld()
                .flatMap { world ->
                    mono {
                        hello.awaitSingle() + ' ' + world
                    }
                }
    }
}

fun getHello() : Mono<String>  = mono {
    delay( time: 1, TimeUnit.SECONDS)
    return@mono "Hello"
}

fun getWorld() : Mono<String>  = mono {
    delay( time: 1, TimeUnit.SECONDS)
    return@mono "world"
}
```

Combinazione asincrona

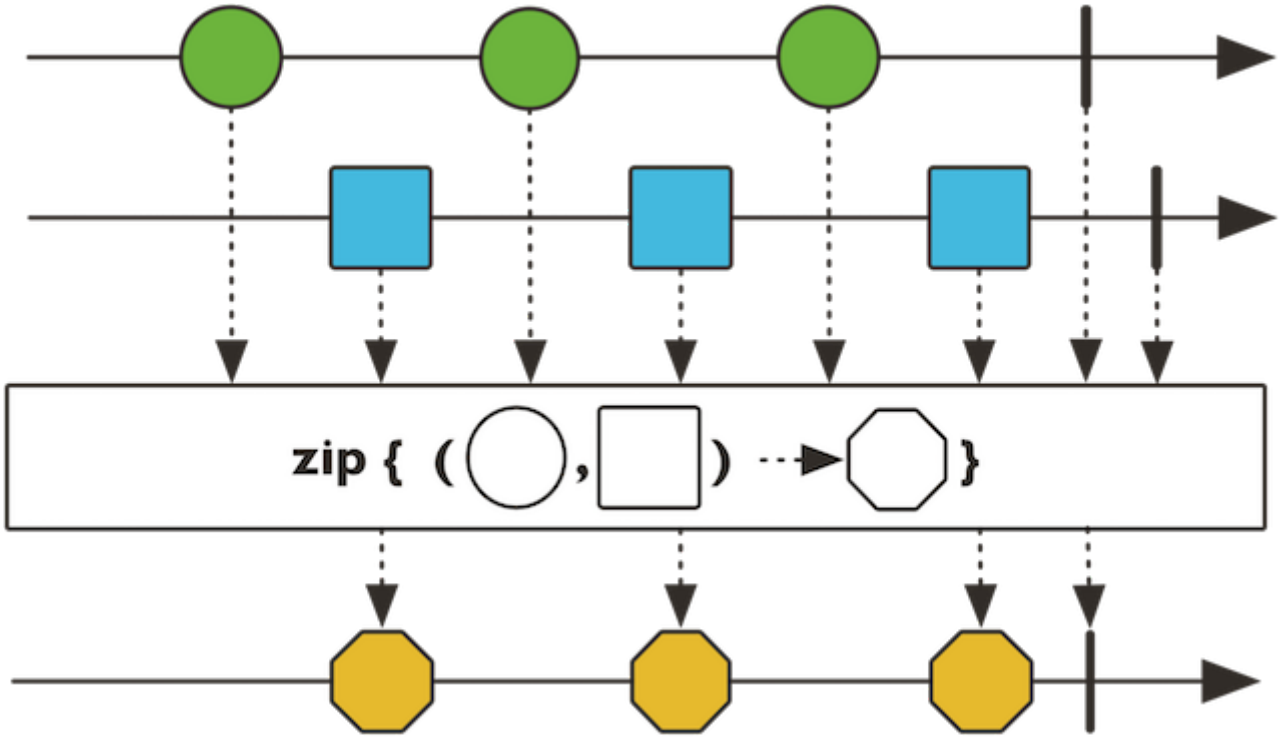
```
@RestController
class Combine2 {
    @GetMapping( ...value: "example/helloWorld2")
    fun helloWorld(): Mono<String> {
        val hello = getHello().toFuture()
        return getWorld().mapNotNull { world ->
            hello.await() + ' ' + world
        }
    }
}
```

```
fun <T : Any, R : Any> ParallelFlux<T>.mapNotNull(mapper: suspend (T) -> R?) =
    flatMap { elem -> mono { mapper(elem) } }
```

Reactor Kore - A Cross-Platform Non-Blocking Reactive Foundation

https://github.com/reactor/reactor-core/issues/979

zip



Conclusioni

- Executor per ridurre il costo dei Thread
- ForkJoinPool per la computazione asincrona non bloccante
- Valutare la programmazione asincrona per ottimizzare l’utilizzo di risorse
- Gli operatori reattivi possono avere particolarità: consultare la documentazione
- Un buon log può essere più utile di uno stack trace

Bubble Bobble

https://github.com/fvasco/reactive-spring

Java™

Matteo Mortari

Kotlin

Francesco Vasco

ReacThing

- Azienda leader nel IoT per acquari
- Facile installazione grazie alla tecnologia Plug and Pray
- Auto-diagnosi dei sensori (valore NaN)
- Configurazione dei sensori centralizzata
- Cruscotto per monitoraggio real-time

https://github.com/fvasco/reactive-spring

Sensore

Latenza: 1000ms

https://github.com/fvasco/reactive-spring

Centralina

Latenza: 600ms

https://github.com/fvasco/reactive-spring

Base dati

Latenza: 400ms

https://github.com/fvasco/reactive-spring

http://localhost:8080/sensorsKt

- Elenco di tutti i sensori disponibili
- SLA 700ms

```
flux.parallel() → parallelFlux

[1, 2, 3, 4, 5].take(3) → [1, 2, 3]

[1, 2, 3, 4, 5].filter { n -> n % 2 == 0 } → [2, 4]

[1, 2, 3, 4].average() → 2.5

[1, 2, 3, 4, 5].mapNotNull { n-> if(n <= 3) n*2 else null } → [2, 4, 6]

mono { “value” }.await() → “value”

flux[1, 2, 3].collectList() → Mono([1, 2, 3])

flux[1, 2, 3].collectMap ({ it }, { it*10 } → Mono(Map{1→10, 2→20, 3→30})
```

http://localhost:8080/sensorsKt/active

- Elenco di tutti i sensori disponibili
- SLA 700ms

- Sensori attivi (escludere i sensori che restituiscono NaN come primo valore)
- SLA 1750ms

http://localhost:8080/sensorsKt/check

- Elenco di tutti i sensori disponibili
- SLA 700ms

- Sensori attivi (escludere i sensori che restituiscono NaN come primo valore)
- SLA 1750ms

- Sensori con valori fuori soglia, preso come media delle misurazioni valide su 3.
  Sensori con tre misurazioni consecutive NaN.
- SLA 3800ms.