UNIVERSIDAD DE ANTIOQUIA

ARQUITECTURA REACTIVA APLICADA AL DESARROLLO DE SOFTWARE

Autor(es)

Juan Alberto Zapata Zapata

Universidad de Antioquia
Facultad de ingeniería,
Ingeniería de Sistemas
Medellín, Colombia

2019

ARQUITECTURA REACTIVA APLICADA AL DESARROLLO DE SOFTWARE

Juan Alberto Zapata Zapata

Asesores (a) o Director(a) o Co- Directores(a).

Luis Hernando Silva

Universidad de Antioquia
Facultad de Ingeniería, Ingeniería de sistemas.
Medellín, Colombia
2019.

Resumen
Era necesario para la empresa Sofka llevar una trazabilidad de cada uno de sus empleados, con trazabilidad nos referimos al hecho de llevar un control de las hora laborales, los proyectos donde se desempeñaba y los respectivos detalles de cada proyecto. Para suplir dicha necesidad se implementó un sistema de Gestión de proyectos interno de la empresa el cual proporciona la información conveniente y necesaria con la cual llevar un informe e historial de cada uno de los empleados y del proyecto al que pertenecían.

Introducción
El problema principal se debía principalmente a la falta de un sistema que gestionara la trazabilidad de los empleados en los diferentes proyectos en los que desempeñaba. La práctica consistió en la realización de un trabajo investigativo en el ámbito de desarrollo de aplicaciones bajo el paradigma reactivo, el cual posee como base, la programación asíncrona y funcional. Las bondades de este paradigma se demostraron al aplicarlas desarrollando un sistema para la gestión de clientes y proyectos de la empresa, la metodología fue bajo el marco de trabajo ágil SCRUM, sin embargo hubo un limitante en concreto que impidió que el total desarrollo de la plataforma y este fue la falta de estructuración y atención requerida. La plataforma puede considerarse con un mínimo producto viable sin embargo cuenta con falencias funcionales.

Objetivos
El objetivo general es la profundización y estudio de la arquitectura reactiva y para la aplicación de su concepto se realizará un sistema para la gestión de los clientes y proyectos de la empresa Sofka, la aplicación será un sistema propio de la empresa y se desarrollará bajo el stack de tecnologías Angular, Java Spring Boot y MongoDB.

Objetivos específicos:

- Realizar un documento que evidencie las bondades, modelamiento, arquitectura y problemas del paradigma reactivo.
- Desarrollar una plataforma de gestión y seguimiento de proyectos para la empresa Sofka Technologies utilizando la metodología del paradigma reactivo.
- Trabajar bajo el estándar de código limpio según el código ISO/IEC 25000.

Marco Teórico
La necesidad de crear sistemas cuyo funcionamiento sea mejor en rendimiento y que proporcionen una mejoría en el tiempo de respuesta y el consumo adecuado de los recursos del sistema se ha convertido en el pan de cada dia de los programadores,“Los usuarios esperan que los tiempos de respuesta sean de milisegundos y que sus sistemas estén operativos el 100% del tiempo. Los datos son medidos en Petabytes. Las demandas de hoy simplemente no están siendo satisfechas por las arquitecturas software de ayer.”[1]. La programacion orientada hacia el ámbito asíncrono ha ido tomando una relevancia digna de una gran atención y podría decirse que los sistemas que actualmente no cuenten con

alguna técnica de programación asíncrona, hoy en día, cuentan con una gran desventaja frente aquellos que sí. Gracias a que esta tendencia de desarrollo ha

ido creciendo, diferentes procedimientos han sido creados, tales como los callbacks, promesas, entre otras, [2]cuyo próximo ciclo, tome lo mejor de los anteriores y les agregue mejoras al diseño del próximo. De estas técnicas ha surgido una cuyo uso está en pleno apogeo y está ganando gran fama por su simplicidad de uso pero compleja y vasta cobertura sobre el modelo asíncrono; esta técnica es llamada programación reactiva. Este tipo de programación establece en su manifiesto los cuatro principios fundamentales del porqué y el para qué fue desarrollada. "Creemos que se necesita un enfoque coherente para la arquitectura de sistemas y creemos que todos los aspectos necesarios ya han sido identificados por separado: queremos sistemas Responsivos, Resilientes, Elásticos y Orientados a Mensajes. Nosotros les llamamos Sistemas Reactivos."[1].

Metodología

la metodología para alcanzar los objetivos planteados se sustenta bajo el desarrollo de sesiones (Dojos) de aprendizajes sobre las distintas tecnologías full-stack previstas, tales como frameworks back-end y front-end. Concurrentemente se realizó un proyecto donde se puso en práctica los conocimientos que se fueron adquiriendo y con ellos se apoyó la realización del sistema para la empresa Sofka Technologies. El proyecto se llevó a cabo bajo una metodología ágil, más específicamente Scrum.

Cronograma de Actividades

La práctica académica constó de 24 semanas, y dada la metodología de trabajo Scrum que se adoptó para el desarrollo del mismo, esas 24 semanas se repartieron en sprints de dos semanas cada uno, para un total de 12 sprints.

Fig 1 Tabla de actividades

#Sprint	Actividad
0	Inception: Introducción a las actividades a desarrollar durante el proceso de prácticas
1	Estudio de vanilla JavaScript y conceptos sobre asincronicidad
2	Realización de un mini proyecto front, con las operaciones CRUD para la gestión de proyecto solo con Javascript
3	Estudio de Spring core y algunas de sus distribuciones más importante. También se estudiarán conceptos sobre Angular y React y se estudiará la arquitectura REST
4	Reunión inicial con los implicados en la iniciativa del desarrollo de la plataforma que gestionará los proyectos de la empresa Sofka. Estudio de las diferentes metodologías de pruebas BDD y TDD
5	Estudio del paradigma de programación funcional y concurrentemente, comienzo en el desarrollo de la plataforma
6	Continuación del estudio de programación funcional, introducción al paradigma reactivo y desarrollo de los servicios en Spring-boot para la plataforma de gestión de proyectos
7	Continuación en el desarrollo de la plataforma aplicando los conceptos que se van adquiriendo de programación reactiva
8	Realización de la documentación de los servicios y continuación en el estudio del paradigma reactivo
9	Se comienza a escribir el documento que se espera como resultado de la práctica
10	Finalización de la primera entrega de la plataforma de gestión de proyectos
11	Refinamiento del documento escrito y exposición de prácticas académicas.

Resultados y análisis

- Se realizó un documento en el que se muestren las bondades y problemas del objeto de estudio, así mismo, proporcionar un modelo de la arquitectura bajo la cual trabaja el modelo reactivo. Ver apéndice A

- Se desarrolló una primera entrega de la plataforma que gestionará los clientes y proyectos de la empresa Sofka. Primer release ambiente desarrollo disponible BackEnd en https://github.com/jazz1228/oursofka, FrontEnd en https://github.com/jazz1228/oursofka-front.git

Conclusiones

En resumen podemos concluir que el presente documento pretende ser una guía y un punto de partida para la construcción de sistemas de negocio robustos que puedan evolucionar en el tiempo según las distintas necesidades y tecnologías marquen la evolución de dichos sistemas; más no puede decirse que se cubren de forma exhaustiva todos los aspectos y detalles en la construcción de dichos sistemas, ya que esto no sería práctico ni posible. A pesar de ello podemos encontrar ciertos pilares que nos guíen hacia la construcción de mejores sistemas, los cuales son:

- Estructura, ampliamente basada en el principio de inversión de dependencias y su realización a nivel de arquitectura: Arquitectura limpia, arquitectura hexagonal o de puertos y adaptadores. (Separación de lo esencial y lo accidental)
- Modelado, hacemos un muy fuerte énfasis en la necesidad de un modelado, análisis y un lenguaje adecuado, a través de los principios del diseño dirigido por el dominio DDD. (Sin entendimiento de dominio no hay diseño correcto posible).
- Estilo y paradigma, Hacer uso de un adecuado nivel de abstracción en el corazón mismo del software nos lleva a un nivel diferente de desarrollo y productividad (programación funcional y reactiva).

Referencias Bibliográficas

[1] Reactivemanifesto.org. (2019). El Manifiesto de Sistemas Reactivos. [online] Available at: https://www.reactivemanifesto.org/es [Accessed 7 Jul. 2019].

[2]Función Callback. (2019). Retrieved from https://developer.mozilla.org/es/docs/Glossary/Callback_function

[3] Evans, E. (2014). Domain-driven design. Boston, Mass.: Addison-Wesley.

Anexos

Apéndice A : Programación y sistemas reactivos

Estructura y modelado

Es necesario comprender el cómo y el para qué del desarrollo de la arquitectura de un sistema cualquiera, esto con el fin de ayudar en su evolución a través del tiempo y en facilitar y promover un alto desacoplamiento entre los diferentes módulos que componen un sistema. Los sistemas reactivos, bajo sus principios de responsividad y elasticidad, presuponen arquitecturas que den soporte a los mismos y por tal motivo se estudiarán esquemas arquitecturales que demuestran un apoyo a los sistemas reactivos.

Propósito y motivación

La arquitectura de un sistema debe entenderse como el conjunto de reglas que definen un sistema o dicho de otro modo los lineamientos a seguir para lograr sistemas bien construidos. En palabras más coloquiales, una arquitectura es una estructura predefinida que contará a manera de historia lo que el sistema va a realizar. (Clean Architecture By Robert C. Martin). Para lograr aquel cometido debe primero responderse la pregunta cuya respuesta tiene mayor pertinencia y relevancia al momento de diseñar una arquitectura ¿ Que hará el sistema a desarrollar?

Teniendo una respuesta a lo anterior se poseerá un conocimiento en particular de lo que hará nuestro sistema y el para qué de su construcción o dicho de manera más precisa, tendremos definido un dominio. Para lograr precisar un dominio se debe contar con personas especializadas en el mismo, pues definido bien este, se tendrá la mayor parte del porcentaje de la arquitectura desarrollada.

Ahora la cuestión es, ¿Cómo lograr que todas las personas involucradas en el desarrollo se entiendan entre sí?; de la anterior pregunta nace un término mencionado por el autor **Eric Evans** en su libro **Domain Driven Desing** [9]: "The Ubiquitous Language" o en español un lenguaje único el cual será comprensible por cada una de las personas.

Pero ahora, ¿cómo medir si la arquitectura cumple con lo **esencial**? De la pregunta podemos sacar dos aspectos importantes: Medición y Esencial . De lo segundo podemos basarnos para medir los primero y es de hecho, en lo que más nos centraremos en este texto.

Cómo la estructura propuesta logrará hacer **claro y accesible lo esencial?**
Cómo se explicará en este documento la arquitectura propuesta tiene sus bases en el concepto general de arquitecturas centradas en el dominio, como lo son la arquitectura limpia, arquitectura de cebolla ó arquitectura de puertos y

adaptadores; el concepto fundamental y central en cada una de éstas es el de aislar el dominio de la solución (**lo esencial**). En este punto se debe aclarar ¿a qué nos referimos con aislar el dominio de la solución y aislarlo de qué?

"Cuando la **complejidad** de este dominio no se trata en el diseño, no importará que la tecnología de infraestructura esté bien concebida. Un diseño exitoso debe abordar sistemáticamente este aspecto central [dominio] del software". — what is ddd, DDD community.

Ahora nace un concepto el cual es la complejidad, pero ésta puede ser dividida en 2 tipos: Complejidad **esencial** y complejidad **accidental**.

Complejidad esencial

Es aquella que es inherente al problema que queremos solucionar, refleja la intención y el propósito por el cual fue creado el sistema y está expresada en los términos del dominio del problema. No es variable a no ser que cambie el dominio y el propósito del sistema.

Complejidad accidental

Es aquella que existe y es inyectada por la solución misma, es decir por los detalles y los medios por los cuales se materializa la solución, es altamente variable y depende en gran medida de detalles y decisiones tomadas en el momento de crear la solución, se ve altamente afectada por detalles y elecciones tomados por todos los que participan en la construcción de la solución y por el entorno tecnológico mismo en el cual se desarrolla. Es muy cambiante ya que un nuevo, framework, librería, tecnología de comunicación, base de datos ó dispositivos la afectan y pueden aumentarla.

¿Qué pasa cuando el código que tiene que ver con la complejidad **accidental** y el que tiene que ver con la complejidad **esencial** no están claramente separados?

El sistemas se torna engorroso y difícilmente desacoplable, pues los componentes se encuentran mezclados entre sí en una arquitectura monolítica que hace confuso y poco claro el fin y para lo que ha sido construido el sistema. Además, la escalabilidad del sistema se hace difícil y el proceso para adaptarse a cambios futuros más costoso y complejo. La complejidad esencial pierde su naturaleza final provocando resultados inesperados. Sin embargo existen métodos destinados a solventar dicho problema. En el actual informe se tratara una arquitectura la cual tiene como principio desacoplar el dominio del negocio (Complejidad esencial) de las tecnologías o frameworks (Complejidad accidental) que se usen para la construcción de la misma.

Modelamiento

El concepto DDD expresa al dominio del negocio como el centro del sistema, las capas externas deben girar entorno al dominio y este debe a su vez ser independiente de la tecnología que sea adaptada al mismo. Mírese (Fig 1) este comportamiento como el de un adaptador.

Fig 1

Ahora expresando lo anterior de manera formal :
Las políticas de alto nivel no deben dependender de las políticas de bajo nivel o lo que se llamarán detalles de implementación tecnológicamente concretos. [9]

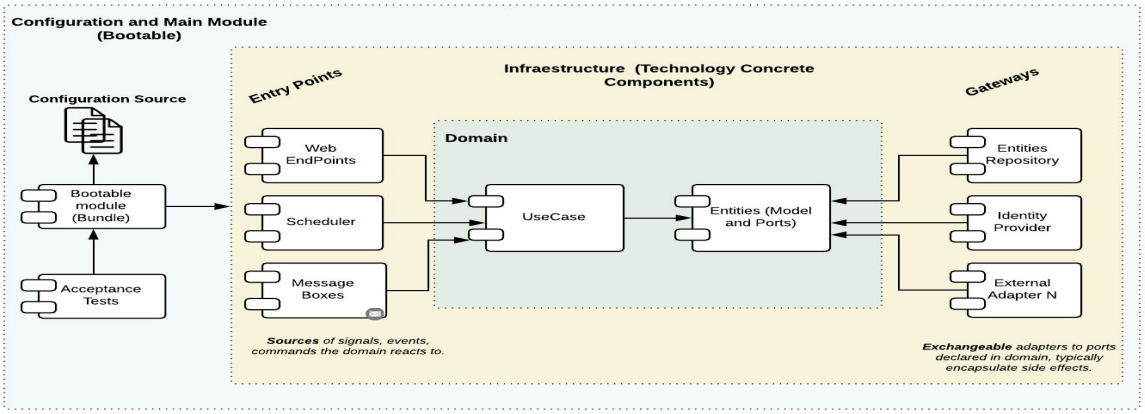


Fig 2

- Los módulos que residen en la capa de dominio no deben tener dependencias hacia los componentes que se encuentran en las capas más externas ya que dichos componentes representan adaptadores a tecnologías y subsistemas externos, los cuales por su naturaleza son volátiles y deben poder ser fácilmente intercambiables.
- Los adaptadores externos pertenecen a una de las siguientes categorías, o son puntos de entrada al sistema, es decir disparan señales, eventos o comandos a los cuales el dominio reacciona o son adaptadores a necesidades de interacción con subsistemas, bases de datos o tecnologías externas. Los módulos en esta capa pueden depender de diversas tecnologías externas. Según sea el caso las reglas de dependencias de estos módulos varía:
    - **EntryPoints**: También conocidos cómo **Driving adapters** Dependen del módulo domain:useCases y su comportamiento está limitado a adaptar y transformar los datos de entrada a un formato que sea compatible con el lenguaje de dominio para posteriormente disparar la ejecución del proceso de dominio, no deben tener ninguna dependencia con ningún otro

EntryPoint ni mucho menos depender directamente de ningún Gateway ya que esto abre la puerta a que se implemente lógica de dominio en las capas externas.
    - Deben delegar la ejecución completamente en los useCases
    - No deben depender de Gateways ni interfaces definidas en el dominio
    - No deben implementar interfaces que serán llamadas por el dominio (ya que este comportamiento es propio de los **Driven adapters**)
- **Driven adapters:** Dependen del módulo domain:model que es allí donde se encuentran las interfaces que este adaptador implementará, su responsabilidad es adaptar y traducir al lenguaje de dominio, interacciones con subsistemas o tecnologías ajenas al dominio, estos módulos son pasivos es decir no llaman al dominio, sino que materializan de una forma tecnológicamente concreta las necesidades expresadas por este en su propio lenguaje.

Estilo de programación: Funcional y Reactivo

Propósito y motivación

Ahora la que procede es definir conceptos claves que ayudarán a la construcción de sistemas reactivos.

Programación Funcional

Si bien el arquitectura reactiva no es estrictamente elaborada bajo el paradigma funcional, pues esta puede a su vez coexistir con la POO (programación orientada a objetos), en este documento ahondaremos en los conceptos básicos de la programación funcional y el cómo facilita el proceso para establecer sistemas reactivos.

Segun el paper "Why Functional Programming Matters [8]:
"Los programas funcionales no contienen sentencias de (re)asignación, es decir una vez las variables obtienen un valor, este nunca cambia (constantes en lugar de variables). De forma más general decimos que los programas funcionales puros no contienen efectos secundarios en absoluto. Una llamada a una función no puede tener otro efecto más que computar su resultado. Esto elimina la mayor fuente de bugs y también hace el orden de ejecución irrelevante — dado que ningún efecto secundario puede cambiar el valor de una expresión, esta puede ser evaluada en cualquier momento. Esto libera al programador de la carga de tener que prescribir

el flujo de control. Ya que las expresiones pueden ser evaluadas en cualquier momento, se pueden reemplazar las expresiones por sus valores que pueden estar pre-computados — esto significa que los programas son "Referencialmente

transparentes". Esto hace a los programas funcionales más "tratables" matemáticamente que sus contrapartes convencionales."

Datos inmutables

Este debería considerarse uno de los conceptos más importantes del paradigma funcional pues representa la esencia del mismo bajo la premisa que establece que una vez una variable se le asigna un valor, esta se mantendrá constante a través del tiempo lo cual facilitará el proceso de debugging (Obviamente manteniendo la tasa de bugs casi ínfima). Dado que se estableció desde un principio trabajar con una arquitectura limpia, encontramos en nuestro sistema un dominio el cual es "el centro de todo" y para especial caso tratado en el documento, la inmutabilidad de las datos (constantes o final en Java) desempeña un papel importante al definir el que será el modelo del dominio.

```java
@Data
@Builder(toBuilder = true)
@AllArgsConstructor
public class ActivitySofkiano {

    private final String id;
    private final String sofkianoId;
    private final String projectId;
    private final String projectName;
    private final Date date;
    private final Date creationDate;
    private final int hours;
    private final String activityDescription;
    private final boolean billable;
    private final String type;


}
```

Funciones puras

"Las variables pueden ser cambiadas por sus valores y viceversa" Según el paper Why Functional Programming Matters [8], la transparencia referencial puede verse en su máximo esplendor en la llamadas funciones puras. Las funciones puras son aquellas que se limitan a su contexto solamente evitando producir cambios en el exterior. A continuación se ilustran ejemplos de las mismas.

- Operaciones básicas aritméticas

```java
public int suma(int value1, int value2){
    return value1+value2;
}
```

- Funciones de utilidad

```
public static Date getJustMonthAndYear() throws ParseException {
    SimpleDateFormat formatter = new SimpleDateFormat(
            pattern: "yyyy-MM");
    Date today = null;

    return today = formatter.parse(formatter.format(new Date()));


}
```

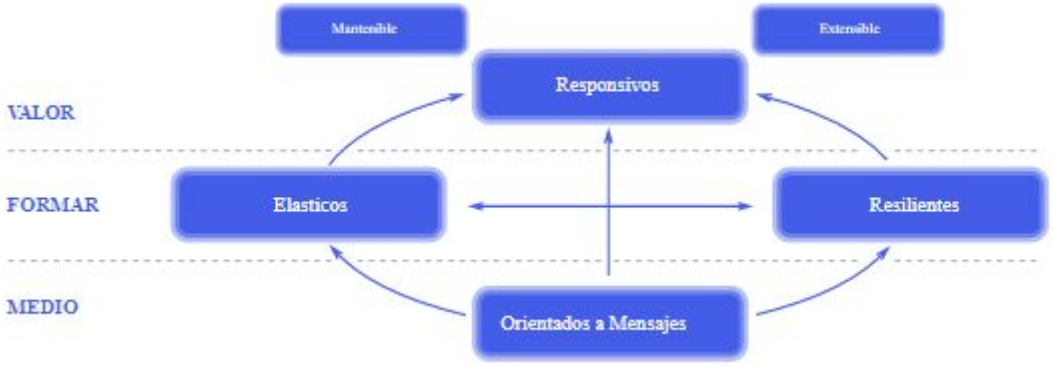
- Factories

```
static Mono<Technology> createTechnology(String id, String name, String urlIcon) {
    return isEmpty(id, name, urlIcon) ? error(BusinessException.Type.INVALID_CREATION.build())
            : just(Technology.builder().id(id).name(name).urlIcon(urlIcon).build());
}
```

Programación Reactiva

Ahora, habiendo hablado de programación funcional se da inicio al concepto fundamental de este documento, reactividad en la programación. Al hablar de sistemas reactivos se debe referir al concepto de asicronicidad y flujo de datos (Streams), y siendo aún más específicos flujos reactivos "Reactive streams"[3]. Los flujos reactivos son aquellos que soportan dicha asincronicidad y conceptos llevados a la práctica como lo son backpressure y operaciones no bloqueantes, lo cual convierte al sistema que lo implemente en un sistema altamente responsivo.

Pero para alcanzar dicha responsividad el sistema debe cumplir ciertos requisitos que apoyen dicho cometido. Esto presupone un cambio a nivel de pensamiento arquitectónico de todo un sistema de software que desemboca en el alineamiento de todos los componentes del mismo hacia el cumpliento de ciertas reglas que dictan resiliencia, elasticidad y orientación a mensajes [1].



[1] Gráfica de Sistema Reactivo.

Existen herramientas para los diferentes lenguajes que soportan dicha estructura, sin embargo en este documento se usará Project Reactor[4], una propia del lenguaje de programación Java y desarrollada por Pivotal y soportada por Spring.

En Reactor existen 2 tipos de datos fundamentales los cuales son Mono[5] y Flux[6] y representan de forma correspondiente un resultado asíncrono de 0..1 ó una secuencia asíncrona de 0..n ítems.

Mono

Un flujo de tipo Mono en Reactor según su documentación, produce resultados que vas de 0..1 a un elemento y su comportamiento de inicio a fin se controla por medio de señales de iniciación y finalización. La manera más común convertir cualquier estructura de datos a un flujo reactivo es por medio del método just(). Ejemplo:

```
Mono.just(T_sofkiano);
```

De ahí en adelante lo que pase con el flujo será controlado mediante señales y operadores.

Las señales que pueden finalizar un flujo son onError() u onComplete(), para que este continue se usa la señal onNext().

En la siguiente imagen se ilustra el proceso de flujos en Mono

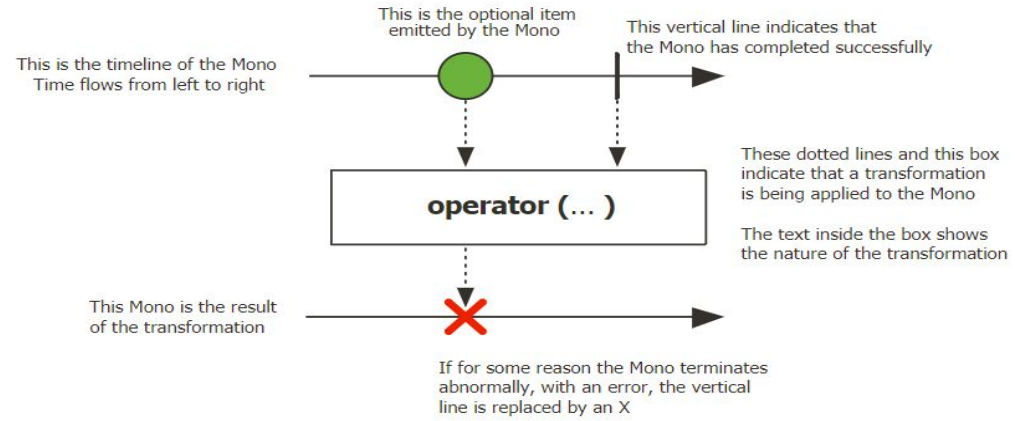


Fig 7 [5]

Flux

Por su parte Flux es una estructura un algo más compleja en su comportamiento pues esta da resultados 0..N lo cual indica que puede manejar más de un ítem en su flujo reactivo. La manera de iniciarlo es similar a un Mono y sus señales de iniciación y finalización son similares, unos cuantos operadores son distintos.

El siguiente diagrama ilustra el comportamiento de un Flux

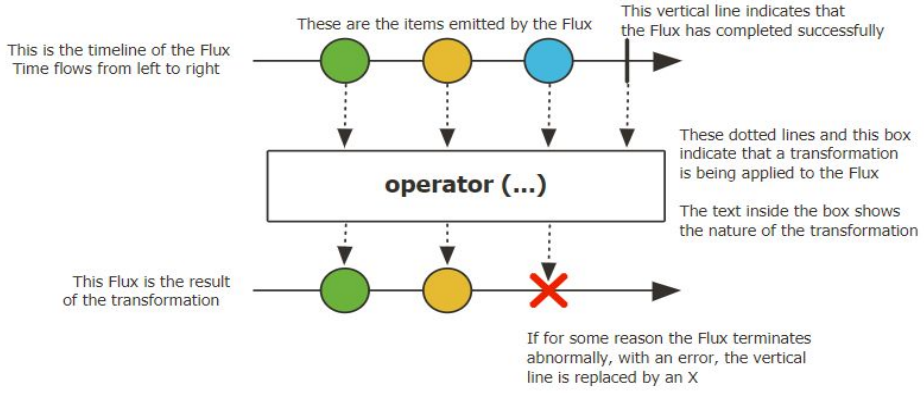


Fig 8 [6]

Como se puede apreciar, es un comportamiento similar al de un Mono con la diferencia de que este produce más elementos que un Mono.

Disminución de latencia

Una operación bloqueante convencional aumenta la latencia pues hace uso de diferentes hilos de ejecución cuando se realiza una petición a un sistema construido bajo este modelo. La latencia de la misma es la suma de la latencia de cada uno de los hilos de ejecución.

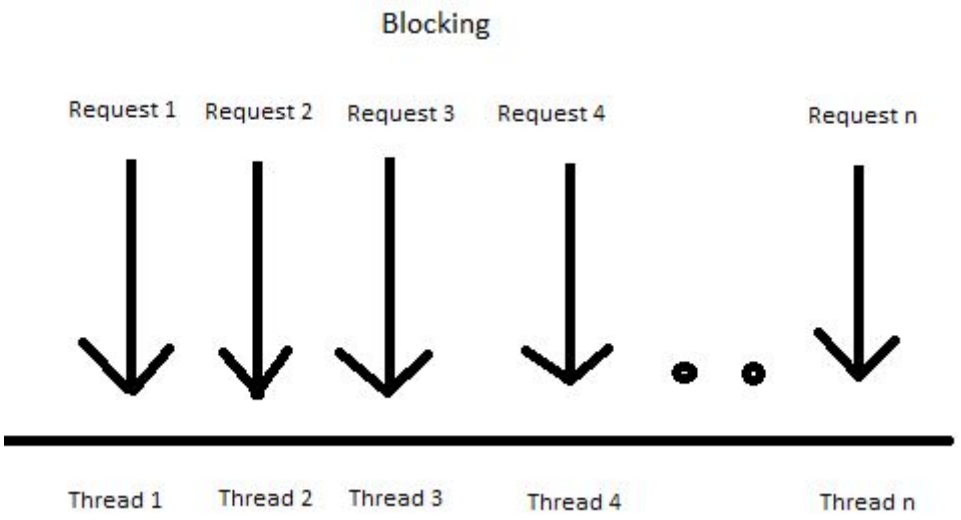


Fig 9 Sistema bloqueante

$$LatTotal = (Lat(Req1) + Lat(Req2) + Lat(Req3) + Lat(Req4) + .. + Lat(Reqn))$$

Ahora como se aprecia en la imagen anterior la latencia es realmente alta y en cualquier momento los recursos del sistema podrían llegar a agotarse. Para ello se tiene como opción la implementación de un sistema Non-Blocking o no bloqueante como se muestra a continuación en la siguiente figura :

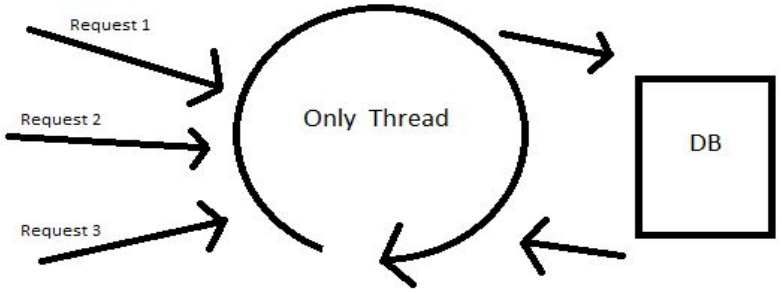


Fig 10 Sistema no bloqueante

$$LatTotal = Max(Lat(Req1) + Lat(Req2) + Lat(Req3))$$

En este caso se logra disminuir la latencia total de la petición a la latencia máxima de las llamadas internas y no a la suma de estas, además de que a los recurso del

sistema se le dan mejor uso, aprovechando así la capacidad que ofrece la memoria.

Ejemplo

```
public Mono<HistorySofkiano> createHistorySofkianoAndSofkianoInProject(String idProject, SofkianoIdAndHourlyRate sofkianoInProject, Date startDate, Date endDate) {

    Mono.zip(manageSofkianoInProjectUseCase.createNewSofkianosInProject(sofkianoInProject.getIdSofkiano(), idProject, startDate, endDate),
            manageSofkianoUseCase.findById(sofkianoInProject.getIdSofkiano()), uuid()).flatMap(mergeStreams ->

            firstProjectInSofka(mergeStreams.getT3(), mergeStreams.getT2(), idProject, sofkianoInProject)


    ).then();

    return Mono.empty();

}
```

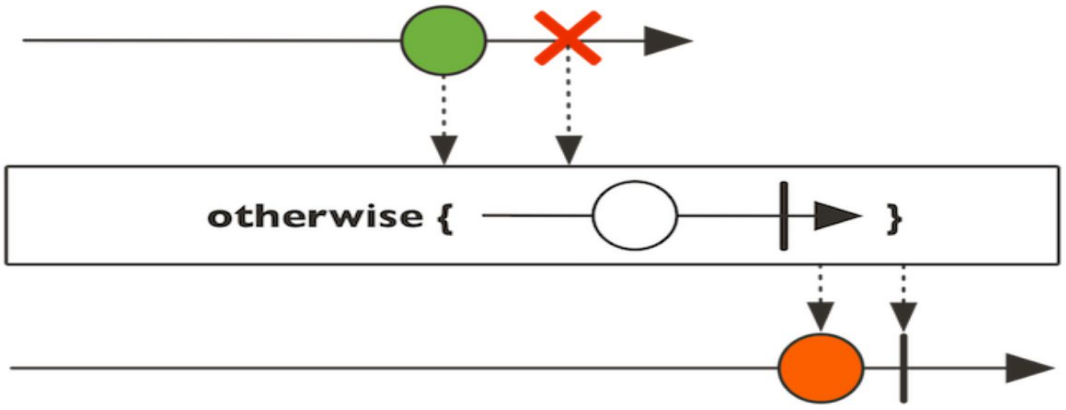
Como se aprecia en el código, gracias a la programación reactiva y la herramienta Reactor, se simplifica muchísimo el manejo de Stream reactivos y los beneficios que este representa para el sistema.

Aumentar la resiliencia a errores

Cuando una determinada acción puede fallar y se desea que el sistema sea resiliente a esa falla concreta (aunque hay casos en que el enfoque es simplemente dejar que otro componente de más alto nivel maneje el fallo) podemos hacer uso de:

- Flujos alternativos (fallback)
- Valores por defecto en caso de error.

Para la solución con flujos alternativos usamos el operador onErrorResume() para tener una alternativa y soportar fallos que se pueden dar en el proceso.



Ejemplo:

```
Flux.just("a", "b", "c")
        .flatMap { s ->
            if (s == "b")
                Mono.error<RuntimeException>(RuntimeException())
            else
                Flux.just(s + "1", s + "2")
        }.onErrorResume { throwable -> Mono.just("d") }.log()
        .subscribe { println(it) }
```

En el ejemplo anterior se usa un flujo alterno cuando se obtiene como resultado una excepción genérica y se procede a realizar una operación alterna en consecuencia.

Orientación a mensajes

Cuando hablamos de arquitectura de microservicios nos referimos a un aspecto de la arquitectura del sistema a alto nivel que define el sistema como un conjunto de módulos **independientes** que colaboran para lograr un objetivo de negocio de un alcance mucho mayor, así como las distintas áreas de una compañía colaboran para lograr los objetivos globales.

Un sistema reactivo es un sistema distribuido robusto, es decir da ciertas garantías fundamentales sobre su adecuado funcionamiento [1]. Para hacer posibles dichas garantías deben aceptarse las limitaciones y condiciones de un sistema distribuido y hacer frente a estas de forma adecuada, para lo cual el enfoque de comunicación asíncrona dirigida por mensajes se hace fundamental, teniendo como beneficios adicionales el bajo acoplamiento, el aislamiento (independencia) y transparencia en la localización.

A continuación se muestra una comparación entre comunicación entre microservicios sincrona y asincrona :

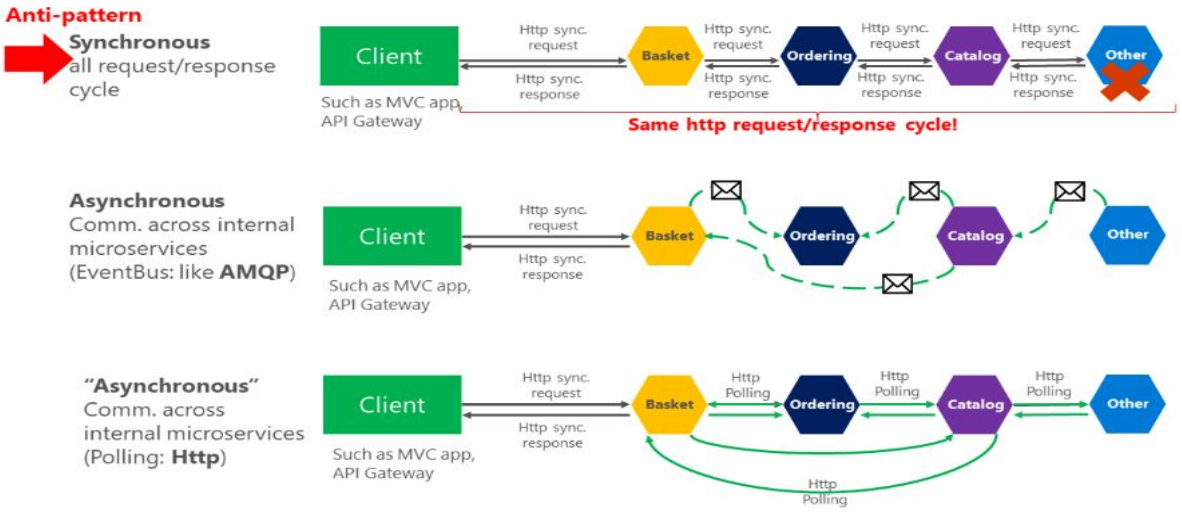


Fig 11 [10] patrones de comunicación entre microservicios

Tipos y/o patrones de mensajes - Semántica de comandos y eventos

Cuando hablamos de comunicación asíncrona dirigida por mensajes, nos podemos dar cuenta que dichos mensajes pueden ser usados de diversas formas y con distintas semánticas las cuales detallaremos a continuación.

Eventos (Event notification - Pub/Sub)

Los eventos representan un suceso que ha ocurrido en el dominio, la representación de una decisión y/o cambio de estado, los eventos pueden verse como notificaciones que emiten los diferentes microservicios para informar a todos los posibles interesados de una decisión, suceso o cambio de estado que ha tenido lugar en su correspondiente contexto delimitado. Es importante notar que cuando se emite un evento, el emisor de dicho evento es completamente autónomo en el sentido de aquello que está notificando, es decir el evento representa un suceso ocurrido en tiempo pasado por ejemplo: "direcciónDelClienteActualizada", "pagoAProvedorRealizado", "polizaRevocada". Es muy importante entender las implicaciones de esto, ya que los eventos no representan intenciones ni peticiones sino sucesos ocurridos que no pueden ser cambiados, por lo tanto es muy importante que sean emitidos solamente por el subdominio que tenga la autonomía de tomar dicha decisión.

Los eventos deben ser emitidos para el contexto global del dominio y deben estar disponibles para cualquier posible interesado, es decir, el sistema emisor no debe conocer qué sistemas están interesados en dicho evento ni debe ser de su interés si existen o nó interesados en dicho evento.

Al ser emitido un evento, los mecanismos subyacentes deben garantizar que este sea entregado a todos los interesados que se han subscrito a su ocurrencia, dando garantía de entrega con la semántica de entrega "al menos una" (**At-Least-Once**).

Comandos

Los comandos representan la intención de realizar una acción en concreto; una acción que debe ser realizada por el contexto que tenga la responsabilidad de realizarlas, es decir puede verse (al contrario de los eventos), como una petición concreta de ejecutar un proceso o acción por parte de un sistema externo, por ejemplo: "notificarPagoAProvedor", "agregarClienteAListaNegra".

Los comandos representan acciones que deben ser ejecutadas a petición de un contexto que tiene la autonomía de solicitar la realización de dichas acciones como parte de la realización de sus propios procesos, aunque no tenga la responsabilidad directa de la ejecución de estas acciones sí tiene la autonomía de solicitar la realización de éstas.

Los comandos pueden verse como la forma asíncrona y desacoplada del consumo de servicios expuestos desde otro microservicio.

De manera gráfica a nivel abstracto se representa de una mejor manera la forma como interaccionan tanto los eventos y los comandos en el sistema a continuación:

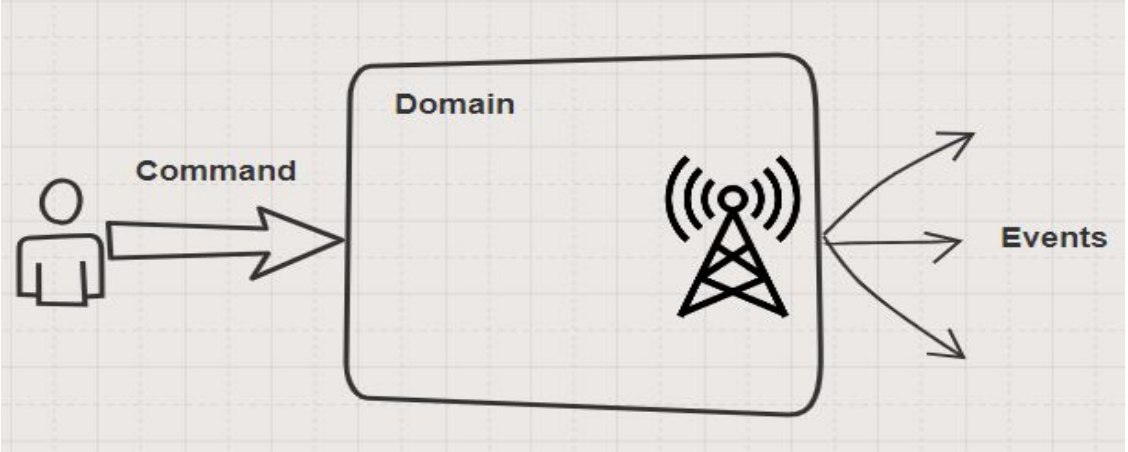


Fig 12 [7]

Broker de Mensajeria

Para lograr la comunicación existen diferentes herramientas de utilidad que ayudan a lograr dicho fin, estas son broker de mensajería y podemos encontrar tales como Apache Kafka, Apache ActiveMQ, RabbitMQ, entre otros. Este documento hará uso de RabbitMQ pero orientado al entorno reactivo. Project Reactor[11] adaptó el mismo.

Una breve introducción a RabbitMQ puede dar a través de su arquitectura

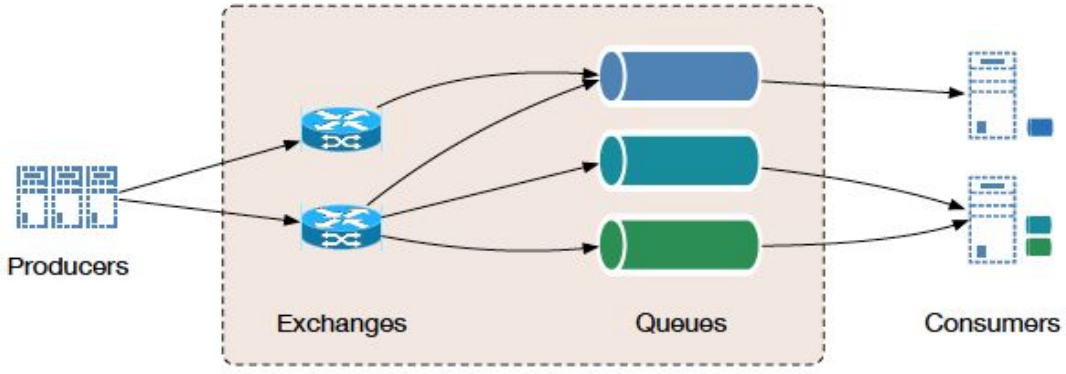


Fig 13

De la figura 13 se puede extraer los conceptos básicos bajo el funcionamiento del broker.

Productores: Son aquellos que inicia la comunicación enviando los mensajes a los Exchanges o direcciones que serán las encargadas de enrutar los mismo a las diferentes colas que sean definidas.

Exchanges: Como se decía anteriormente, estos son los encargados de dirigir el mensaje enviado del productor a las colas donde se almacenarán para ir siendo procesados bajo la filosofía FIFO.

Queues o colas: El mensaje será almacenado aquí para luego ser consumido por los diferentes "suscriptores".

Consumidores o Suscriptores: Como el nombre lo indica, estos serán los diferentes clientes suscriptos y los cuales, dependiendo de la configuración que se les sea suministrada, consumen los mensajes de las colas.

Para profundizar más en los conceptos se recomienda acceder la página principal de RabbitMQ [12]

Glosario

Comunicación no bloqueante HTTP

En el contexto que se trabaja en el documento es necesario usar una herramienta que se adapte a nuestros requerimientos y funcione de manera asíncrona y haga o consuma peticiones HTTP no bloqueantes. Para el caso de Spring se puede usar WebFlux o WebClient como se verá a continuación :

```
public Mono<Details> someRestCall(String name) {
        return this.webClient.get().uri("/{name}/details", name)
                                        .retrieve().bodyToMono(Details.class);
}
```

En el anterior ejemplo se consume un servicio por medio de su URI.

Backpressure

Cuando se desarrolla una aplicación reactiva es esencial manejar adecuadamente los conceptos de control de flujo de datos y backpressure, para ello tomaremos como base los conceptos expresados en la especificación "*Reactive Streams*" (ver http://www.reactive-streams.org/). Segun el manifiesto reactivo el mecanismo BackPressure se define así "Cuando un componente está luchando para mantenerse, el sistema en su conjunto debe responder de manera sensata. Es inaceptable que el componente bajo estrés falla catastróficamente o que deje caer los mensajes de manera incontrolada. Ya que no puede hacer frente y no puede fallar, debe comunicar el hecho de que está bajo tensión para los componentes en sentido ascendente y así lograr que reduzcan la carga."

Referencias y bibliografía

[1] Reactivemanifesto.org. (2019). *El Manifiesto de Sistemas Reactivos*. [online] Available at: https://www.reactivemanifesto.org/es [Accessed 7 Jul. 2019].

[2] Daniel Bustamante Ospina. International Operations At Sofka Technologies.

[3] Reactive-streams.org. (2019). [online] Available at: https://www.reactive-streams.org/ [Accessed 7 Jul. 2019].

[4] Projectreactor.io. (2019). *Project Reactor*. [online] Available at: https://projectreactor.io/ [Accessed 7 Jul. 2019].

[5] Projectreactor.io. (2019). *Mono (Reactor Core 3.2.10.RELEASE)*. [online] Available at: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Mono.html [Accessed 7 Jul. 2019].

[6] Projectreactor.io. (2019). *Flux (Reactor Core 3.2.10.RELEASE)*. [online] Available at: https://projectreactor.io/docs/core/release/api/reactor/core/publisher/Flux.html [Accessed 7 Jul. 2019].

[7] Medium. (2019). *Command vs. Event in Domain Driven Design - Ingeniously Simple - Medium*. [online] Available at: https://medium.com/ingeniouslysimple/command-vs-event-in-domain-driven-design-be6c45be52a9 [Accessed 7 Jul. 2019].

[8] Hughes, J. (1989). Why Functional Programming Matters. *The Computer Journal*, 32(2), pp.98-107.

[9] Evans, E. (2014). *Domain-driven design*. Boston, Mass.: Addison-Wesley.

[10] Docs.microsoft.com. (2019). Comunicación en una arquitectura de microservicio. [online] Available at: https://docs.microsoft.com/es-es/dotnet/standard/microservices-architecture/architect-microservice-container-applications/communication-in-microservice-architecture [Accessed 8 Jul. 2019].

[11] Arnaud Cogoluègnes, P. (2019). *Reactor RabbitMQ Reference Guide*. [online] Projectreactor.io. Available at: https://projectreactor.io/docs/rabbitmq/snapshot/reference/ [Accessed 8 Jul. 2019].

[12] Rabbitmq.com. (2019). *Messaging that just works — RabbitMQ*. [online] Available at: https://www.rabbitmq.com/ [Accessed 8 Jul. 2019].