Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2020/2021

Trabajo de Fin de Máster

Comparativa de frameworks de sagas en microservicios

Autores:
Stefano Lagatolla Struggia y Ana González Santamaría

Tutor:
Micael Gallego Carrillo

ÍNDICE

1. RESUMEN

Este trabajo ha consistido en un estudio del patrón de sagas y varios frameworks diseñados para la implementación del mismo. En la parte teórica, hemos comparado cómo se implementan, cómo se conectan con sus respectivos microservicios y curva de aprendizaje entre otras cosas. En cuanto a la parte práctica, hemos desarrollado una serie de ejemplos con cada uno de estos frameworks para el mismo caso de uso, de forma que se puedan comparar más fácilmente entre sí.

Las comparaciones realizadas a lo largo de este proyecto pretenden ampliar la documentación de unas tecnologías aún muy novedosas y ayudar en la elección de la herramienta que más se adapte a las necesidades de tu aplicación a la hora de implementar sagas y otras metodologías de comunicación entre diferentes microservicios.

A lo largo de esta memoria, hablaremos de la motivación y los objetivos de este proyecto, explicaremos las bases teóricas de los conceptos más importantes, estudiaremos tres de los frameworks más relevantes en cuanto a la comunicación de microservicios, veremos las implementaciones del patrón saga en estos frameworks y compararemos las principales características de estas tecnologías, remarcando sus ventajas e inconvenientes. Estas tecnologías en las que se centrará nuestro proyecto son Axon, Eventuate y Cadence.

2. INTRODUCCIÓN Y OBJETIVOS

2.1 MOTIVACIÓN

Cuando trabajamos con monolitos, la capa de la interfaz de usuario y la de acceso a todos los datos están combinados en la misma plataforma, por lo que las operaciones se realizan mediante transacciones de la misma base de datos y si hay errores la propia transacción deshace los cambios.

Aunque esta arquitectura es más sencilla de implementar, a medida que crece la aplicación el monolito se vuelve más difícil de mantener. Es por esto por lo que muchos deciden pasarse a arquitecturas microservicios. Los microservicios son un enfoque arquitectónico organizativo que consiste en separar la aplicación en servicios independientes y comunicarlos entre sí.

La complejidad de los microservicios se debe precisamente a esta comunicación. Puesto que todos los microservicios tienen su propia base de datos, a la hora de deshacer los cambios en caso de error nos vemos obligados a acceder de forma individual a cada uno de los microservicios implicados.

Es este problema el que provoca que surjan frameworks para facilitar la gestión de estos procesos de comunicación. Existen diversas metodologías en estos frameworks, tales como APIs, Event Sourcing, Sagas, etc. Entre todas estas, nosotros nos hemos centrado en el Patrón de Sagas.

La razón principal de realizar este trabajo es la aparición de varias de estas herramientas de comunicación muy similares entre sí, la falta de documentación y de ejemplos que nos hemos encontrado, una comunidad muy pequeña y otros inconvenientes que nos llevan a preguntarnos cuál es la más adecuada y a realizar una comparación entre los más destacados.

2. 2 OBJETIVOS

Como hemos dicho anteriormente, los objetivos de este trabajo son investigar, comparar y documentar las diferentes tecnologías que existen para la comunicación entre microservicios y, específicamente, la implementación de sagas. A continuación, vamos a desarrollar cada uno de dichos objetivos.

En lo referente a la investigación, hemos estudiado las distintas tecnologías, analizado los ejemplos existentes y aprendido las diferentes sintaxis que utiliza cada una. Con esto hemos pretendido tener una base teórica antes de profundizar en lo referente a los siguientes objetivos.

En cuanto a la comparación de tecnologías, hemos implementado el mismo ejemplo en java con los distintos frameworks a estudiar para así poder resaltar los pros y los contras de cada uno. Con esto buscamos ayudar a encontrar la opción que más se ajuste a las necesidades de cada caso de uso.

Por último, como resultado de los objetivos anteriores, hemos podido generar una documentación adicional a la ya existente de cada herramienta. Esto cumple con el objetivo de ampliar la documentación de estos frameworks, que en ocasiones es muy escasa debido a su novedad y a una comunidad muy reducida.

Durante los siguientes capítulos de esta memoria, iremos explicando cómo se han desarrollado cada uno de estos objetivos.

3. BASES TEÓRICAS

3.1 MICROSERVICIOS

Los microservicios son un sistema de desarrollo software que ha estado ganando popularidad en los últimos años. Lo que distingue esta arquitectura de los enfoques monolíticos más tradicionales es que está compuesta por un conjunto de pequeños servicios que funcionan de manera autónoma. La razón por la que este tipo de arquitectura no está tan arraigada es porque es laboriosa de implementar y comunicar, aunque también presenta una serie de ventajas con respecto a la arquitectura monolítica precedente. [1]
A continuación, vemos las principales ventajas e inconvenientes de los microservicios. [2]

Ventajas

- Funcionalidad modular, ya que está compuesta por módulos independientes.
- Es escalable.
- Uso de contenedores permitiendo el despliegue y desarrollo de una aplicación rápidamente.
- Fáciles de mantener y testear.

Desventajas

- Dificultad para fragmentar el monolito en distintos microservicios.
- Complejidad de gestión que se incrementa con el aumento del número de servicios.
- Hay que lidiar con la problemática de los sistemas distribuidos: comunicación interna, manejo de dependencias, consistencia de los datos almacenados, entre otros.

Como acabamos de ver, uno de los principales inconvenientes a la hora de implementar una arquitectura microservicios es la gestión de la comunicación entre los distintos servicios que componen una aplicación. Esta comunicación se puede realizar por medio de APIS con https, colas de mensajes, eventos o sagas.

En este trabajo nos centraremos en el patrón de sagas, del cual hablaremos detalladamente a continuación.

3.2 SAGAS

El patrón de arquitectura “Saga” proporciona una gestión de transacciones mediante una secuencia de transacciones locales. Una transacción local es la unidad de trabajo ejecutada por cada participante de la saga. Cada una de estas actualiza la base de datos y publica un mensaje, un evento o una petición para avanzar a la siguiente transacción de la saga. [3] De esta manera la aplicación mantiene la consistencia de sus datos en múltiples servicios.

Toda operación que forme parte de la saga puede ser revertida por transacciones de compensación. Esto garantiza que, o bien todas las operaciones se completan con éxito, o bien se ejecutan las correspondientes transacciones de compensación para deshacer el trabajo previamente realizado. Como contrapartida, el modelo de programación es más complejo, pues se deben diseñar transacciones de compensación específicas para cada una de las transacciones previas.

En el patrón saga, una transacción de compensación debe ser idempotente y reintentable. Estos dos principios garantizan que una transacción pueda ser gestionada sin ninguna intervención manual. [4]

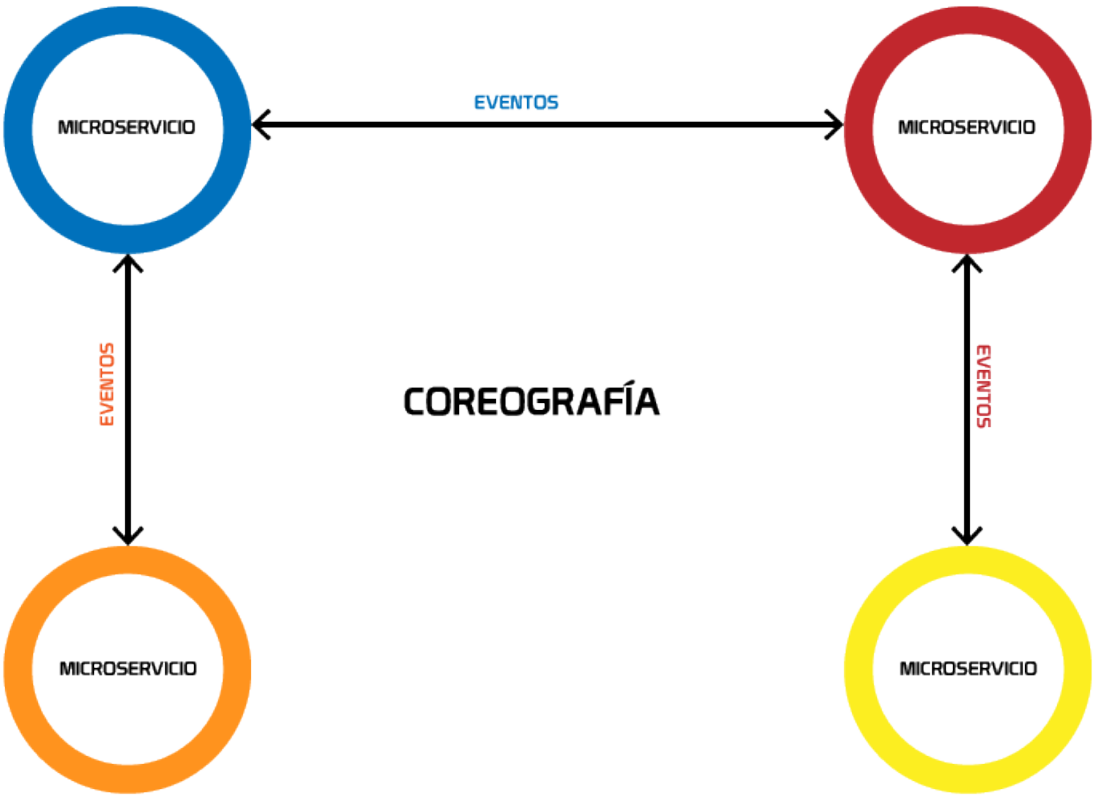
Hay dos maneras de coordinar las sagas:

- Coreografía: cada transacción publica eventos de dominio que “llaman” a otras transacciones de otros microservicios.

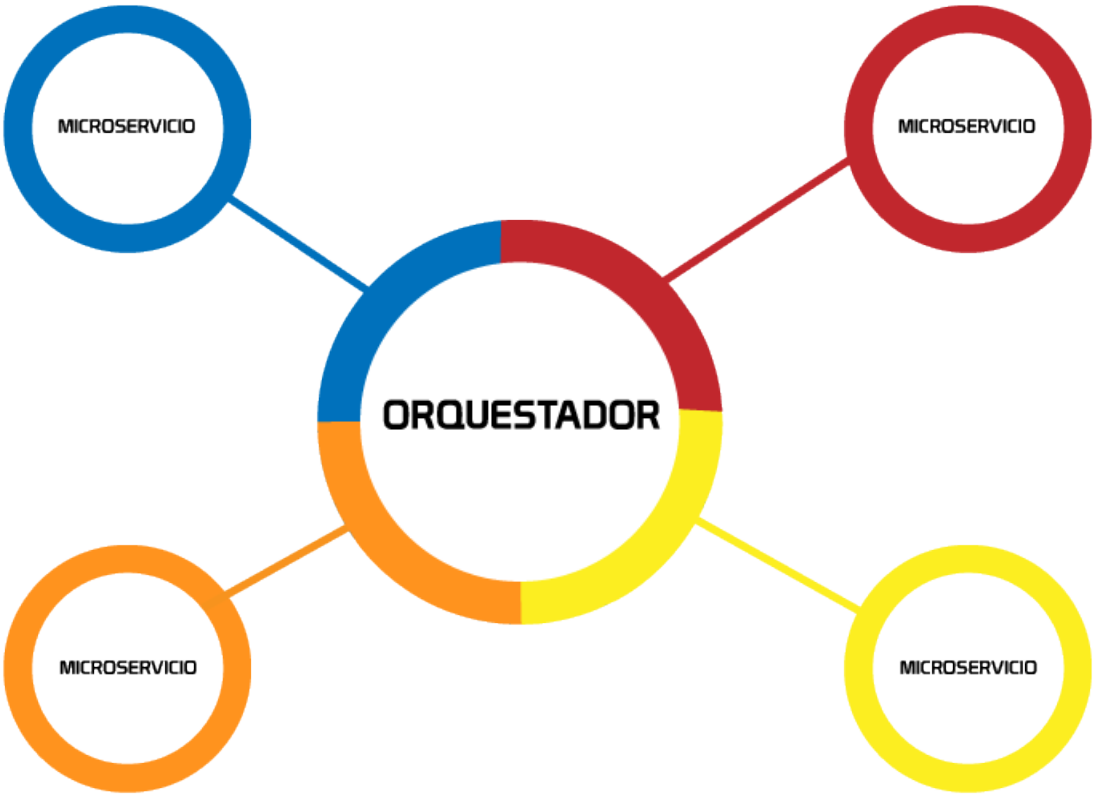
- Orquestación: creamos un objeto orquestador, que es el encargado de decirle a los participantes qué transacción ejecutar.

3.2.1 Orquestación vs. Coreografía

La coreografía es un enfoque basado en eventos. Cuando usamos una coreografía, los participantes se suscriben a cada uno de los eventos del resto de participantes que necesiten y responden de forma acorde. Cuando un servicio actualiza sus datos emite un evento del dominio anunciando que se ha hecho. En los servicios suscritos a estos eventos se "llamarán" a nuevos updates que a su vez generarán nuevos eventos. [5]



La orquestación usa un objeto orquestador que le dice a cada uno de los participantes de la saga qué hacer. El orquestador se comunica con los participantes usando interacción request/response. Para ejecutar un paso de la saga, se manda un comando al participante indicándole qué operación ejecutar. Después de que el participante ejecute la acción, envía una respuesta de vuelta al orquestador. El orquestador procesa la respuesta y determina qué paso de la saga ejecutar a continuación. [5]



El patrón **orquestador** nos permite conocer de una manera fácil el flujo de cada transacción, ya que cada servicio se comunica con una clase central llamada orquestador. La complejidad se va a mantener de forma lineal, pues todo se conecta a una sola clase. Por otra parte, el patrón **coreografía** no está centralizado, sino que debemos comprobar qué eventos emite y recibe cada servicio. Debido a esto, es mucho más complejo seguir la traza de las transacciones y aplicar sus respectivas compensaciones. La complejidad será mayor cuantos más servicios y transacciones tenga nuestra aplicación.

Sin embargo, el patrón **coreografía** es mucho más sencillo de implementar, ya que cada servicio se comunica directamente con aquel del que dependa sin ningún intermediario. Por su parte, el patrón **orquestador** puede contener mucha lógica, lo que complica la implementación y mantenimiento de la saga.

Después de ver esto, podemos concluir que ambos patrones son igualmente válidos y que la elección de uno u otro dependerá del escenario, influyendo el número de servicios, la cantidad de transacciones o el número de pasos de las mismas. Para casos de uso con pocos participantes y transacciones es más recomendable usar el patrón **coreografía** para mantener la sencillez; mientras que cuando aumenta el tamaño, es preferible implementar el patrón de **orquestación** para centralizar el flujo de las comunicaciones y hacerlo más comprensible y por tanto facilitar su mantenimiento.

3.3 FRAMEWORKS

Un framework, también conocido o marco de trabajo, es un código de software ya desarrollado que proporciona estructura y funcionalidades para desarrollar un software mayor. Este código se puede usar como base para facilitar el desarrollo, abstrayendo al programador de funcionalidades triviales, para poder enfocarse en implementaciones más complejas. [6]

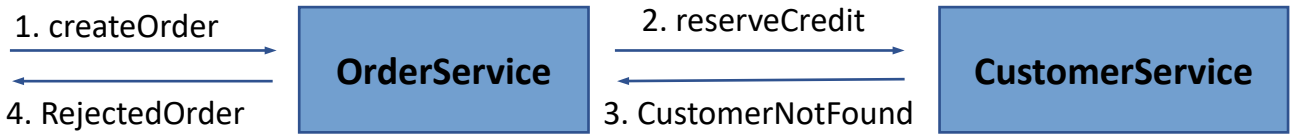
En el siguiente capítulo vamos a ver una serie de frameworks que nos ayudan a la hora de implementar el patrón Saga y comunicar microservicios entre sí.

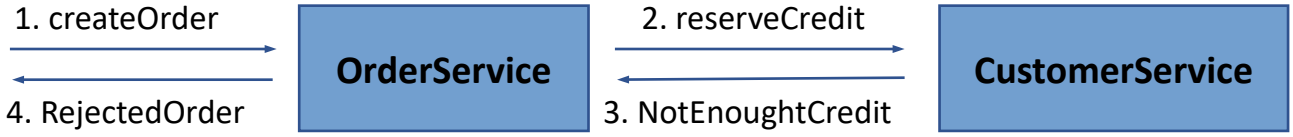
4. FRAMEWORKS DE SAGAS

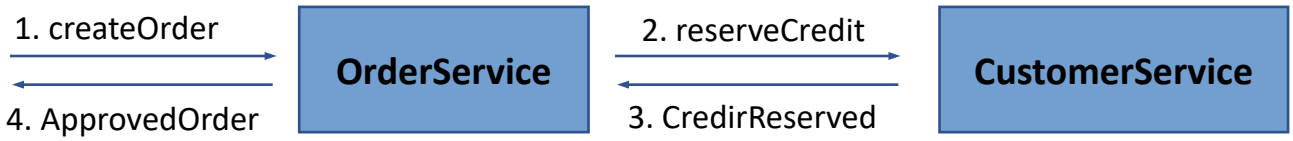
Como hemos visto anteriormente, los frameworks nos ayudan a desarrollar software de forma más sencilla. Para ayudarnos en la implementación del patrón Saga, hemos utilizado tres frameworks diferentes: Axon, Eventuate y Cadence.

A continuación, vamos a ver el funcionamiento de cada una de estas tecnologías y cómo se implementa una saga en ellas. Para ello vamos a basarnos en el ejemplo de la aplicación de pedidos y clientes que se explica en la página de descripción del patrón saga [3] y que hemos implementado con cada uno de los frameworks.

La lógica de esta aplicación es sencilla: consiste en la creación de pedidos *Order,* por un cliente determinado *Customer.* El pedido tiene un coste determinado, por lo que si el cliente no tiene suficiente dinero el pedido se rechazará. La arquitectura de esta aplicación está constituida por dos microservicios: *OrderService*, que crea los pedidos en estado pendiente y asociados a un cliente y a un precio, y *CustomerService,* que es el encargado de reservar el dinero necesario para el pedido o enviar un error en caso de que el dinero sea insuficiente.







En los anteriores diagramas podemos ver los tres posibles casos que pueden ocurrir a la hora de crear un pedido: que se rechace porque no existe el cliente asociado al pedido (primer caso), que se rechace porque el cliente asociado al pedido no tiene dinero suficiente (segundo caso) o que el pedido se apruebe porque el cliente reserve el crédito con éxito (tercer caso).
Para coordinar estos dos microservicios implementamos una saga, utilizando para ello los tres frameworks que veremos a continuación.

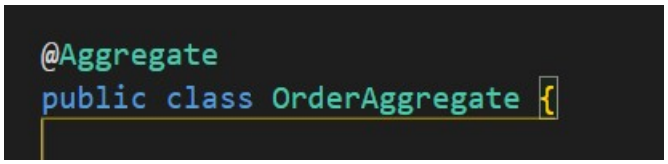
Se puede consultar el código de cada uno de estos ejemplos en este enlace [7].

4.1 AXON

Axon [8] es una tecnología creada en marzo de 2010 por la compañía AxonIQ. Es un framework desarrollado para la construcción de sistemas de microservicios controlados por eventos, basado en las arquitecturas DDD, CQRS y Event Sourcing [9]. Esta tecnología proporciona las herramientas necesarias para poder implementar estos tipos de arquitecturas, tales como agregadores, repositorios, comandos, eventos, colas de peticiones y almacenamiento de eventos.

La implementación de sagas con esta tecnología requiere de un cliente, que sería el propio framework, y un Axon-Server. Este es un servicio que centraliza y gestiona los eventos, estados y comunicaciones entre los participantes.

Para empezar a implementar una saga en una aplicación, en primer lugar debemos definir un agregador, que es un objeto que contiene un estado y un método para modificar el mismo.



Para definir una saga necesitamos tener un método handle que espera la llegada del evento especificado por la anotación **@SagaEventHandler.** La saga se inicia cuando se recibe un evento

anotado con **@StartSaga**. Cuando se recibe una excepción se lanza un comando de compensación con su respectivo evento, el cual será recibido por otro servicio que deshará los cambios. En nuestro caso, definimos este método de inicio en el *OrderService*, de manera que la saga comienza cuando se recibe el evento que indica que una orden ha sido creada. Desde este servicio se manda el comando *ValidateCustomerPaymentCommand* al *CustomerService* y se espera su respuesta.

```
@Saga
public class OrderSaga {

    @StartSaga
    @SagaEventHandler(associationProperty = "orderId")
    public void handle(OrderCreatedEvent orderCreatedEvent){
        String customerId = orderCreatedEvent.getCustomerId();
        //Asociar la saga al siguiente associationProperty
        SagaLifecycle.associateWith("customerId", customerId);
        //Enviar el nuevo commando para accionar
        try {
            commandGateway.sendAndWait(new ValidateCustomerPaymentCommand(customerId,
            orderCreatedEvent.getPrice(), orderCreatedEvent.getOrderId()));
          } catch (RuntimeException e) {
            commandGateway.send(new OrderRejectedCommand(orderCreatedEvent.getOrderId(),
            OrderStatus.REJECTED, "Customer Not Found"));
            SagaLifecycle.end();
          }
    }
```

Para recibir un comando y actualizar el estado del agregador tenemos que definir un método con la anotación de **@CommandHandler** el cual recibe un comando para resolverlo emitiendo un evento. El evento es recibido tanto por el servicio que va a realizar el siguiente paso de la Saga como por el propio agregador para actualizar su estado.

En esta imagen vemos el método definido en *CustomerService* para recibir el comando lanzado. Al recibir el comando lanzado en la imagen anterior comprueba si se cumplen las condiciones necesarias y lanza un evento que indica qué respuesta debe darle el *CustomerService* al *OrderService.*

```
@CommandHandler
public void handle(ValidateCustomerPaymentCommand validateCustomerPaymentCommand) {
    if(validateCustomerPaymentCommand.getPrice().compareTo(balance) == -1 ) {
        AggregateLifecycle.apply(new ValidatedCustomerPaymentEvent(
                validateCustomerPaymentCommand.getOrderId(),validateCustomerPaymentCommand.getPrice()
                , validateCustomerPaymentCommand.getCustomerId()
                )
        );
    }else {
        AggregateLifecycle.apply(new InsufficientMoneyEvent(validateCustomerPaymentCommand.getOrderId
        (), validateCustomerPaymentCommand.getCustomerId()));
    }
}

@EventSourcingHandler
public void on(ValidatedCustomerPaymentEvent validatedCustomerPaymentEvent) {
    balance = balance.subtract(validatedCustomerPaymentEvent.getPrice());
}
```

4.2 EVENTUATE

Eventuate [10] es una tecnología creada por Chris Richardson en agosto de 2013. Es una plataforma que permite resolver los problemas generados por la gestión distribuida de los datos en arquitecturas microservicios para que el desarrollador pueda centrarse en la lógica de negocio. Esta tecnología a su vez está constituida por tres frameworks: Eventutate Local, basado en Event Sourcing; Eventuate Tram, basado en mensajes transaccionales y Eventuate Tram Sagas, un framework basado en sagas por orquestación. [11] En este trabajo nos hemos centrado en Eventuate Tram Sagas.

Eventuate Tram Sagas es un framework para implementar sagas en microservicios Java que usen SpringBoot, Micronaut o Quarkus. Está basado en el framework Eventuate Tram, que funciona mediante el envío de mensajes asíncronos entre los distintos participantes de la saga. Esto permite a los microservicios actualizar de forma automática su estado y publicar esta información como mensajes o eventos a otros servicios. [12]

Eventuate Tram está conformado por cuatro servicios: Apache Zookeeper, Apache Kafka, una base de datos MySQL y un componente CDC. Todos estos servicios se despliegan con docker-compose de forma distribuida.

Para implementar una saga con Eventuate, lo primero que debemos hacer es crear una clase que implemente la interfaz **SimpleSaga** y dentro de la misma definir un orquestador con **SagaDefinition**. El orquestador será la parte más importante de la aplicación, pues divide el funcionamiento de la misma en pasos de ejecución.

Cada uno de los pasos declara un método que se corresponde con la acción que tiene que hacer alguno de los participantes cuando se alcanza ese paso. También están definidas las compensaciones de los pasos que las necesiten, las cuales se ejecutarán si se produce un fallo en el resto de la saga. Así mismo, se incluye el manejo de las posibles excepciones lanzadas por los métodos declarados en cada paso.
En nuestro ejemplo, la saga se encuentra definida en el *OrderService* .

```
public class CreateOrderSaga implements SimpleSaga<CreateOrderSagaData> {

  private SagaDefinition<CreateOrderSagaData> sagaDefinition =
          step()
            .invokeLocal(this::create)
            .withCompensation(this::reject)
          .step()
            .invokeParticipant(this::reserveCredit)
            .onReply(CustomerNotFound.class, this::handleCustomerNotFound)
            .onReply(CustomerCreditLimitExceeded.class, this::handleCustomerCreditLimitExceeded)
          .step()
            .invokeLocal(this::approve)
          .build();
```

El método *reserveCredit()* del orquestador, definido también en *OrderService*, debe comunicarse con el *CustomerService,* pues es el que guardará en su base de datos la información de la reserva de crédito. Para ello, el método lanza un mensaje con un **Command** desde el *OrderService*, el cual será recibido por un método **Handler** en el *CustomerService*, estableciendo así la comunicación entre los dos microservicios.

```
private CommandWithDestination reserveCredit(CreateOrderSagaData data) {
  long orderId = data.getOrderId();
  Long customerId = data.getOrderDetails().getCustomerId();
  Money orderTotal = data.getOrderDetails().getOrderTotal();
  return send(new ReserveCreditCommand(customerId, orderId, orderTotal))
            .to("customerService")
            .build();
}
```

```
public CommandHandlers commandHandlerDefinitions() {
  return SagaCommandHandlersBuilder
            .fromChannel("customerService")
            .onMessage(ReserveCreditCommand.class, this::reserveCredit)
            .build();
}
```

4.3 CADENCE

Cadence [13] es un framework desarrollado por la empresa Uber para gestionar los distintos microservicios que componen su plataforma. No estuvo disponible de forma libre hasta el año 2017, cuando convirtieron Cadence en un proyecto open source. [14]

Cadence consiste en un framework de programación (cliente) y un servicio de gestión (backend). El framework permite al desarrollador crear y coordinar funciones y soporta los lenguajes Go, Java, Python y Ruby (aunque los dos últimos no tienen soporte oficial). El backend es un servicio stateless y depende de un almacenamiento persistente, con Cassandra, MySQL o PostgreSQL. Este servicio backend maneja el historial de los workflows, coordina las actividades de cada participante, redirige las señales al worker correcto, etc. [15]

En Cadence, usamos el término **Workflow** para referirnos a una abstracción con estado que engloba toda la transacción y la lógica de negocio. El workflow no puede hacer llamadas al exterior, por lo que organiza su ejecución en **Activities**, que son funciones propias de cada participante y que sí pueden comunicarse con servicios externos (APIs, mensajes Kafka, etc). [16]

Para crear una saga en Cadence, debemos definir actividades en cada uno de los servicios que participen en la misma e implementar un workflow que orqueste todas estas actividades.

Las actividades se definen como métodos de una interfaz. Debemos colocar estas interfaces en un paquete común, ya que las usaremos tanto en el workflow como en el servicio donde se implementen. En nuestro caso, las usaremos tanto en *CustomerService* como en *OrderService*.

```
package es.codeurjc.example_cadence_common.activities;

public interface CustomerActivities {
    void reserveCredit(Long customerId, Double amount);
}
```

El workflow también se define en una interfaz con métodos y, en nuestro caso, se encuentra definida en *OrderService*. Estos métodos contienen anotaciones para indicar que reaccionan a señales o queries. La anotación principal es **@WorkflowMethod**, que indica que el método se ejecuta cuando se inicia el workflow. Cuando este método termina se considera que el workflow ha finalizado.

```
public interface CreateOrderWorkflow {
    @WorkflowMethod
    Long createOrder(Long customerId, Double totalMoney);
}
```

La implementación de actividades se realiza en otro servicio distinto al del Workflow, en este caso *CustomerService*. Este método comprueba si existe el cliente y si tiene crédito suficiente y, si es así, se reserva el dinero. En caso contrario se lanza una excepción que deberá ser manejada por el orquestador para que termine la saga y se inicien las compensaciones.

```java
@Override
public void reserveCredit(Long customerId, Double money) {
    Optional<Customer> customer = service.findById(customerId);
    if(customer.isPresent()) {
        if (Double.compare(customer.get().getMoney(), money) > 0) {
            Customer optCustomer = customer.get();
            optCustomer.setMoney(optCustomer.getMoney() - money);
        }else{
            throw new CreditLimitExceededException();
        }
    } else {
        throw new CustomerNotFoundException();
    }
}
```

La implementación del Workflow también se hace en *OrderService.* Para empezar, instanciamos un objeto **Saga**. A continuación, llamamos al método *createOrder()* y añadimos su compensación *rejectOrder()*. El siguiente paso será llamar a la actividad del *CustomerService*, a través del método *reserveCredit*. Si todo va bien continuará la ejecución de la saga y se ejecutará el *approveOrder.* En caso contrario, si el *reserveCredit* falla, se lanzará una excepción y en el catch se llamará a las actividades de compensación de la saga.

```java
@Override
public Long createOrder(Long customerId, Double amount) {
    Saga.Options sagaOptions = new Saga.Options.Builder().build();
    Saga saga = new Saga(sagaOptions);
    try {
        Long orderId = orderActivities.createOrder(customerId, amount);
        saga.addCompensation(orderActivities::rejectOrder, orderId);
        customerActivities.reserveCredit(customerId, amount);
        orderActivities.approveOrder(orderId);
        return orderId;
    } catch (ActivityFailureException e) {
        saga.compensate();
        throw e;
    }
}
```

5. COMPARACIÓN DE FRAMEWORKS DE SAGAS

Una vez que hemos estudiado y definido los frameworks de Axon, Eventuate y Cadence, centrándonos en la manera que tienen de implementar las sagas, vamos a hacer una comparación de los mismos. Para ello vamos a enfocarnos en una serie de propiedades que dividiremos en tres grupos, resaltando sus pros y sus contras y explicando cada una de ellas.

5.1 FICHA TÉCNICA

	Axon	**Eventuate**	**Cadence**
Año de creación	2010	2013	2017
Tamaño de la comunidad	Grande	Mediana	Pequeña
Lenguajes soportados	Java (Spring Boot)	Java (Spring Boot, Micronaut y Quarkus)	Java, Go (oficial) Python, Ruby (comunidad)

En este primer apartado vamos a comparar tres características generales de los frameworks: antigüedad, comunidad y lenguajes.

Axon, al ser el framework más antiguo de los tres, es también el que mayor comunidad tiene. Cuenta con gran número de ejemplos y vídeos explicativos además de dudas resueltas en distintos foros. A continuación le sigue Eventuate, con una comunidad algo menor, aunque con gran número de ejemplos y dudas resueltas. Por último, tenemos a Cadence. Realmente fue creado antes de 2017, aunque fue en ese año cuando se hizo público, por lo que no podemos hablar de comunidad hasta ese momento. Es por esto por lo que decimos que tiene una comunidad pequeña, con un número reducido de ejemplos y pocas dudas resueltas.

Con respecto a los lenguajes, Axon y Eventuate usan únicamente Java, por lo que se asemejan bastante, mientras que Cadence originalmente se creó para Go y luego se amplió a Java, lo que influye notablemente en la cantidad de ejemplos que tiene.

Con estos aspectos, podemos determinar que Axon y Eventuate son tecnologías más consolidadas mientras que Cadence está empezando a sentar sus bases.

5.2 SOFTWARE

	Axon	**Eventuate**	**Cadence**
Curva de aprendizaje	Rápida	Media	Lenta
Complejidad del código	Media - Alta	Media	Alta
Server	Integrado	No integrado	Integrado
Server DB	MySQL, PostgreSQL	MySQL	Cassandra, MySQL, PostgreSQL
Invasividad en el código	Baja	Alta	Alta

En este segundo apartado vamos a estudiar características más específicas de los frameworks, centrándonos en su funcionamiento y en su implantación en nuestra aplicación.

En primer lugar, vemos la curva de aprendizaje, es decir, la facilidad que tiene el desarrollador de familiarizarse con el funcionamiento del framework. Podemos observar que está directamente relacionado con el tamaño de la comunidad que vimos en el apartado anterior. También podemos ver la relación con la complejidad del código, aunque con algún matiz que comentaremos a continuación.

A nivel de complejidad, Axon es un framework sencillo, es bastante intuitivo y cuenta con numerosos ejemplos como ya vimos en el apartado anterior. Sin embargo, lo consideramos de dificultad media alta porque su complejidad crece progresivamente a medida que aumenta el número de participantes que se deben coordinar. Por su parte, Eventuate puede resultar complicado inicialmente, ya que se debe configurar la clase orquestador, pero resulta sencillo leer el código y escalar la aplicación a un número mayor de participantes. Por último, Cadence cuenta también con la ventaja de la escalabilidad que tiene Eventuate. Sin embargo, la configuración del orquestador (lo que llamamos workflow en este framework) y la poca información que hay al respecto nos han dado muchos problemas a la hora de trabajar con esta tecnología, llevándonos a colocarla en primer lugar en cuanto a dificultad.

En cuanto al server, mientras que Cadence y Axon tienen un servidor integrado que se debe levantar para gestionar la información de los estados, Eventuate gestiona sus propios estados valiéndose para ello de cuatro servicios: un apache zookeeper, un apache kafka y un componente CDC. Además, los tres almacenan la información de los estados en base de datos.

Por último, hablaremos de invasividad del código. Con esto nos referimos a cuánto debemos modificar nuestra aplicación inicial para conseguir integrar uno de estos framework y cuánto dependen nuestras clases de dicho framework. Así, vemos que Eventuate y Cadence son altamente invasivos, pues todo debe depender de un orquestador central; mientras que Axon es menos invasivo, ya que las clases ejecutan su funcionalidad normal solo que lanzando eventos para llamar al siguiente participante una vez finalizada la acción.

5.3 CONSTRUCCIÓN DE SAGAS

	Axon	**Eventuate**	**Cadence**
CQRS	Sí	Opcional	Opcional
Asíncrono por defecto	Sí	No	Sí
Tipo de saga	Coreografía	Orquestación	Orquestación
Manejo de errores	Manual	Automático	Manual
Compensaciones	Manual	Automático	Automático

En este último apartado nos centraremos en los aspectos más relacionados con la construcción de las sagas en cada uno de los tres frameworks.

El aspecto más importante de este apartado es el tipo de patrón saga que implementa cada uno de los frameworks. Axon utiliza sagas con coreografía, lo que provoca una menor invasividad pero una complejidad creciente para gran número de participantes, como hemos visto anteriormente. Por su parte, Eventuate y Cadence implementan sagas con orquestación, lo que hace que sean más invasivos pero a su vez más fácilmente escalables y más sencillos para aplicaciones con un gran número de participantes.

También es importante destacar cómo se comportan estos frameworks ante los errores y qué procesos se llevan a cabo para compensar los cambios en la base de datos que han generado las transacciones anteriores.

En este aspecto el menos automático de los frameworks es Axon, ya que al ser un patrón de sagas por coreografía debe ser el encargado de manejar las excepciones con *try* y *catch* para detectar el error y lanzar el comando correspondiente para que se ejecute la compensación.

El siguiente en cuanto a nivel de automatización sería Cadence. Este framework tiene orquestador, por lo que se define exactamente qué acción se debe ejecutar para compensar una determinada transacción. Sin embargo, el orquestador debe manejar las excepciones con *try* y *catch* y, en el caso de que entre en el *catch*, debe llamar a la acción *compensate*, como podemos ver en la imagen. Es por eso que decimos que las compensaciones son automáticas mientras que el manejo de errores se hace de forma manual.

```
try {
    //Acción
    Long orderId = orderActivities.createOrder(customerId, amount);
    //Compensación de la acción
    saga.addCompensation(orderActivities::rejectOrder, orderId, rejectedReason);
    customerActivities.reserveCredit(customerId, amount);
    orderActivities.approveOrder(orderId);
    return orderId;
} catch (ActivityFailureException e) {
    //En caso de error lanzamos la compensación
    saga.compensate();
    throw e;
}
```

Por último, el framework más automático en cuanto al manejo de errores sería Eventuate. En este framework se define en la propia saga tanto las compensaciones a realizar como las excepciones que se pueden recibir. De esta manera, en cuanto una acción lanza una excepción la saga la detecta y pone en marcha el proceso de compensación. Podemos verlo en la imagen.

```
private SagaDefinition<CreateOrderSagaData> sagaDefinition =
        step()
          .invokeLocal(this::create) //Acción
          .withCompensation(this::reject) //Compensación
        .step()
          .invokeParticipant(this::reserveCredit)
          //Posibles excepciones que puede lanzar la acción reserveCredit
          .onReply(CustomerNotFound.class, this::handleCustomerNotFound)
          .onReply(CustomerCreditLimitExceeded.class, this::handleCustomerCreditLimitExceeded)
        .step()
          .invokeLocal(this::approve)
        .build();
```

6. CONCLUSIÓN

Como dijimos en los primeros capítulos de esta memoria, uno de los objetivos de este proyecto era ayudar a determinar qué framework es el que mejor se adapta a las necesidades de nuestra aplicación. Así, después de estudiar cada uno de los tres frameworks tratados en este proyecto y de comparar las características de unos y otros, en este apartado hablaremos de las conclusiones que hemos extraído durante el desarrollo de este trabajo, enfocándonos principalmente en la elección de una tecnología para la implementación de sagas en nuestra aplicación.

En primer lugar, debemos tener en cuenta el tamaño de nuestra aplicación y la complejidad de su lógica de negocio. Como ya hemos visto anteriormente, para aplicaciones pequeñas o medianas, que cuentan con un número reducido de participantes es más recomendable utilizar Axon, ya que es más sencillo de implementar en estos casos al utilizar sagas con coreografía. Sin embargo, para aplicaciones con gran número de microservicios implicados o con previsión de crecimiento en un futuro, es preferible utilizar Eventuate o Cadence porque el patrón de sagas por orquestación hace que sean escalables sin aumentar por ello su complejidad. Además, entre Eventuate y Cadence, es más aconsejable decantarse por Eventuate ya que dispone de mayor documentacion.

El lenguaje de la aplicación también es un factor a tener en cuenta. Axon y Eventuate son tecnologías firmemente asentadas en el lenguaje Java, tienen grandes comunidades que les respaldan pero limitadas únicamente a dicho lenguaje. Por su parte, Cadence a pesar de ser muy nueva y tener una comunidad reducida, está disponible en distintos lenguajes como Go, Python, Ruby o el propio Java lo cual le da más flexibilidad frente a Axon y Eventuate.

También, para los apasionados de las nuevas tecnologías, Cadence es un framework muy nuevo, que aún está en crecimiento y que tiene muy pocos ejemplos. Esto abre la posibilidad de aportar y desarrollar nuevas ideas, contribuir con la comunidad, por ejemplo, la versión de Cadence en Python y Ruby están mantenidas por la comunidad, etc. Así, esto puede ser una motivación para elegir una tecnología más novedosa como Cadence por delante de otras más asentadas como Axon o Eventuate.

Con todo esto que hemos visto, concluimos que ninguno de estos frameworks es mejor que los otros y que no hay una respuesta definitiva sobre cuál escoger en todas las situaciones. Por eso, la elección del framework que implementará las sagas en nuestra aplicación deberá basarse en el caso de uso de esta, de los recursos y tiempo disponibles y de las preferencias del equipo de desarrollo, siendo igualmente válidos Axon, Eventuate y Cadence.

BIBLIOGRAFÍA

[1] Brian Atkinson, (4 mayo 2017), The truth about microservices. Red Hat Developer. https://www.redhat.com/es/topics/microservices/what-are-microservices

[2] Chackray. ¿Qué son los microservicios? Definición, características y ventajas y desventajas. https://www.chakray.com/es/que-son-los-microservicios-definicion-caracteristicas-y-ventajas-y-desventajas/

[3] Chris Richardson, (27 octubre 2018), Microservices Pattern. Saga Pattern. https://microservices.io/patterns/data/saga.html

[4] Saga Pattern in Microservices, Baeldung. https://www.baeldung.com/cs/saga-pattern-microservices

[5] Chris Richardson, (4 agosto 2019), Managing data consistency in a microservice architecture using Sagas part 2 – coordinating sagas. https://chrisrichardson.net/post/sagas/2019/08/04/developing-sagas-part-2.html

[6] Framework de software, (22 abril 2021), TicPortal. https://www.ticportal.es/glosario-tic/framework-software

[7] https://github.com/MasterCloudApps-Projects/microservices-frameworks

[8] https://axoniq.io/product-overview/axon-framework

[9] https://github.com/AxonFramework/AxonFramework

[10] https://eventuate.io/

[11] Eventuate explained using microservices patterns, (24 febrero 2020), Eventuate. https://eventuate.io/post/eventuate/2020/02/24/why-eventuate.html

[12] https://github.com/eventuate-tram/eventuate-tram-sagas

[13] https://cadenceworkflow.io/

[14] Bea Schuster, (12 noviembre 2019), Conducting Better Business with Uber's Open Source Orchestration Tool, Cadende. Cadence Engineering. https://eng.uber.com/open-source-orchestration-tool-cadence-overview/

[15] https://github.com/uber/cadence

[16] Anar Sultanov, (5 mayo 2021), Orchestration-based saga using Cadence workflow, SULTANOV.DEV. https://sultanov.dev/blog/orchestration-based-saga-using-cadence/