Technical Paper

Digital Architectures

Microservices

Transaction and query management in microservices architectures

Autor:
Santiago García-Bonacho Rico
Corrector:
José Manuel López Doña

Contenidos

1. Introducción 3
2. Contexto y necesidades 4
3. Caso de uso 5
4. Enfoque tradicional 7
4.1. Rediseño hacia el monolito. (marcha atrás) 8
4.2. Sistemas de compensación de transacciones 8
4.3. Transacciones REST-TA 9
5. Enfoque orientado a dominio 10
5.1. Patrón Saga 10
6. Arquitectura propuesta 11
6.1. Arquitectura Reactiva 11
6.1.1. Capa de transformación: API → Event Layer 11
6.1.2. Capa de dominio: Domain Layer 12
6.1.3. Capa de consulta: CQRS opcional 13
6.1.4. Event Gateway, Event Sourcing y Replay Service 14
6.1.5. Eventos Inmutables: El único elemento de integración 14
6.2. Arquitectura Imperativa 15
6.3. Implementación de la Saga - Consistencia Eventual 15
6.3.1. Saga: Coreografía de microservicios 15
6.3.2. Saga: Orquestación de microservicios 16
6.4. Transacciones ACID con Sagas 18
6.5. Consistencia de consulta usando CQRS 18
6.6. Consulta de datos agregados 20
6.6.1. Microservicios Composite 20
6.6.2. Vistas materializadas 20
6.6.3. Procesos Batch 21
6.7. Event Sourcing 21
6.8. Transacciones In-Service (ACID) 22
7. Referencias 23

1. Introducción

El uso de **arquitectura de microservicios** permite la implementación de soluciones con un alto grado modularidad y autonomía en el ciclo de vida del software lo que facilita, por ejemplo:

- Componentes de software con menor acople entre dominios funcionales
- Uso de las tecnologías más adecuadas para la implementación de cada funcionalidad
- Optimización del uso de infraestructuras
- Disminución del time-to-market en el desarrollo nuevas funcionalidades
- Orientación de la implementación a dominios/productos

Por el contrario, cuando mayor autonomía en el diseño, implementación, evolución y ejecución de los microservicios mayor impacto se tiene en la consistencia del sistema, lo que sea refleja principalmente en dos áreas, las relativas a la transaccionalidad y la agregación de datos.

En este documento técnico se revisan en detalle la problemática y principales soluciones propuestas para reducir el impacto de pérdida de consistencia en la **transaccionalidad** de los datos.

2. Contexto y necesidades

Cuando se enfrenta el proceso de implantación de una arquitectura basada en microservicios, ya sea desde cero o rompiendo una aplicación monolítica ya existente, es frecuente hacerse preguntas del tipo:

- **¿Qué pasa con las transacciones entre microservicios?**
- **¿Cómo se orquestan los procesos en los que intervienen microservicios de varios dominios funcionales?**

Este problema ha sido tratado extensamente en el pasado. En el capítulo 4. Enfoque tradicional se hace un estudio de las distintas respuestas convencionales a estas preguntas y en los capítulos 5. Enfoque orientado a dominio y 6. Arquitectura propuesta se realiza un análisis de diferentes alternativas con un enfoque más alineado con los principios de una arquitectura de microservicios.

Con independencia de la solución empleada para la implementación de las transacciones, es fundamental realizar un diseño de los microservicios mediante el uso de técnicas **DDD** (Domain Driven Design) (5), porque un correcto modelado orientado a dominio, va a permitir reducir el acoplamiento y necesidades de manejar transacciones de mayor complejidad, muchas veces resultantes de un mal diseño de los microservicios.

En este sentido, suele ser de gran utilidad:

- Establecer correctamente fronteras de contexto (**context boundaries**) para cada microservicio.
- Agrupar las entidades (**entities**) pertenecientes a un dominio en torno a sus agregados raíz (**aggregate roots**) correspondientes.
- Asegurar que un agregado raíz y todas las entidades asociadas pertenecen únicamente a un microservicio.

Por tanto, una de las primeras preguntas que debemos hacernos cuando nos vemos en la necesidad de implementar una transacción y/o orquestación entre varios microservicios es: **¿He diseñado bien mis microservicios?** Un buen diseño DDD nos ayudará a evitar algunas orquestaciones innecesarias.

Dicho esto, en ciertos casos de uso, existen procesos complejos que involucran varios microservicios, aun estando bien diseñados, y no hay más remedio que plantearse los conceptos de **transacción**, **orquestación** y **coreografía**. En este documento veremos estos casos de uso mediante ejemplos reales y explicaremos que en buena parte de las situaciones, lo más conveniente puede ser mantener una consistencia eventual (1, 2, 3, 4 y 5).

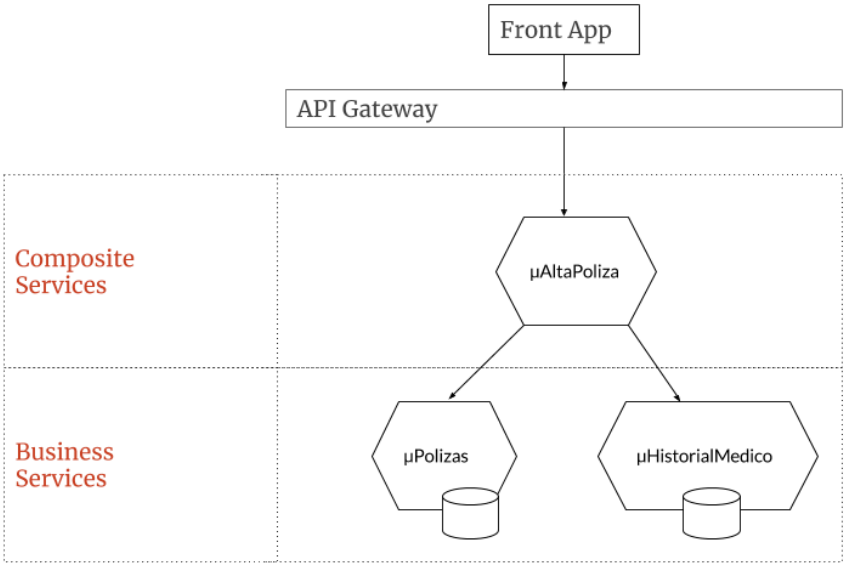
En general, la recomendación es intentar minimizar la orquestación entre microservicios, pero cuando es necesario, se propone el uso del patrón **Saga** para realizar esta orquestación (1 y 5), como se describirá en más detalle en el documento.

3. Caso de uso

Supongamos que somos una compañía del sector asegurador y tenemos los microservicios de **µPolizas** e **µHistorialMedico**, pertenecientes a dos dominios funcionales diferentes.

En este escenario, cuando queremos dar de alta un Póliza se tienen que cumplir dos condiciones de dos dominios funcionales distintos. Por un lado, todos los documentos y datos legales tienen que ser correctos y por otro lado el informe médico del asegurado tiene que estar guardado en su historial médico. Supongamos que µPolizas pertenece al dominio funcional "producción" e µHistorialMedico al dominio funcional “prestaciones”.

Por encima de estos dos microservicios tenemos un **API Gateway** que permite a las aplicaciones frontales acceder a las API de estos servicios. En el proceso de **alta** de una **nueva Póliza** será necesario orquestar una operación de negocio compleja, para ello creamos un nuevo microservicio llamado **µAltaPoliza** que realizará las peticiones necesarias en una capa de servicios que llamaremos composite.



Cuando se da de alta una póliza, estos son los **escenarios** que **pueden suceder** en nuestro ejemplo:

1. La creación la póliza falla en µPolizas por lo tanto el microservicio µAltaPoliza devuelve un error. Si dentro de µPolizas estamos usando transacciones SQL los datos de la póliza nunca se persistirán en nuestro sistema, **¡todo correcto!**
2. La póliza se crea correctamente en µPolizas pero el historial médico no es posible ser creado en µHistorialMedico. µAltaPoliza devuelve un error, pero tenemos una póliza en el sistema que no tiene ningún historial médico asociado: **Inconsistencia de Datos.**
3. La póliza se crea correctamente, pero al crear el historial médico la conexión HTTP con µHistorialMedico se cae repentinamente. La creación del historial médico puede haber sucedido o no, no lo sabemos.

Es un ejemplo simulado de un caso de negocio de una empresa de seguros. No se trata de un modelo de negocio real, es sólo un ejemplo para explicar un caso concreto donde es necesario orquestar varios microservicios de diferentes dominios funcionales.

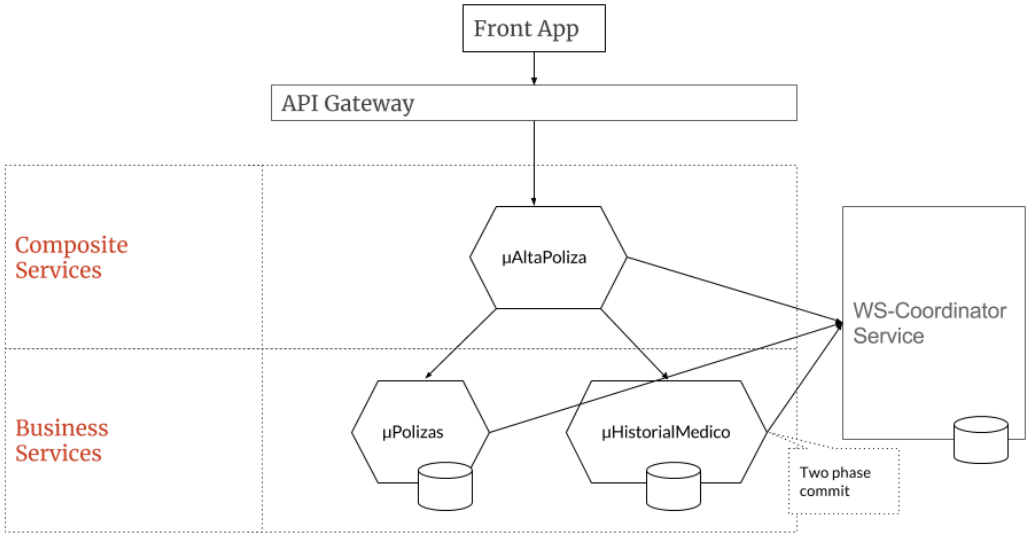
Analizando el problema vemos que estamos ante un ejemplo típico de transacción que involucra varios (micro) servicios, que plateamos resolver desde diferentes puntos de vista en las siguientes secciones del documento.

4. Enfoque tradicional

Tradicionalmente las transacciones se han definido en base al concepto **ACID**:

- **Atómico**: todos los cambios son llevados a cabo en la transacción se realizan o no se realiza ninguno.
- **Consistente**: los datos son transformados de un estado correcto a otro.
- **Aislado**: se impide que varios actualizadores simultáneos interfieran entre sí.
- **Durabilidad**: los cambios entregados (committed) se persistirán incluso en el caso de ocurrir un fallo. En este caso todos los cambios **permanecerán bloqueados** hasta que la transacción no los guarde o haga rollback de todos ellos.

Esto no es un problema nuevo y se ha tratado desde múltiples puntos de vista, desde el Object Transaction Service (OTS) de CORBA pasando por el Transaction API de J2EE y el Java Transaction Service (JTA/JTS) o las especificaciones del estándar XA que definen un estándar para el manejo de transacciones distribuidas (DTP), generalmente para gestores de bases de datos, y por último WS-AT (Web Services Atomic Transactions) de SOAP basado en los tres protocolos: (completion, volatile two-phase commit 2PC, and durable two-phase commit) implementados dentro de un WS-coordinador que actúa como gestor de contexto de transacción emulando el concepto ACID dentro de la transacción global.



Un posible planteamiento podría ser usar un gestor de transacciones distribuidas, WS-Coordinator, que articula el protocolo WS-AT, similar a XA, e implementar una estrategia de commit en dos fases 2PC en cada uno de los servicios intervinientes y si es necesario mediante BPEL articular un protocolo de compensación de transacciones a través de un ESB.

Actualmente sabemos que este tipo de soluciones que proponía la arquitectura SOA basada en SOAP y WS no ha tenido demasiado éxito e incluso en aquella época los especialistas no daban mucho crédito a esta nueva y moderna solución. Según un miembro veterano del comité OASIS, encargado de establecer las especificaciones de los componentes del protocolo WS-TX para las transacciones distribuidas SOAP decía **"I will rely on XA rather that an immature, albeit, modern solution."** (Confiaré antes en XA en lugar de una solución inmadura, aunque moderna) (9) y propone mantener el protocolo XA argumentando que los servidores de aplicaciones Oracle, WebSphere y WebLogic admiten transacciones XA, IBM AIX, Sun Solaris y HP-UX junto con las siguientes bases de

datos empresariales que son compatibles con XA: IBM DB2 Universal Database V8.2 y Oracle 9i y 10g.

Pero, **¿no estaremos matando moscas a cañonazos?** Nuestro caso es **mucho más pequeño** que todo esto y se va a repetir frecuentemente dentro de un mismo sistema en una arquitectura de microservicios.

Este tipo de soluciones tradicionales pueden provocar una bajada drástica del rendimiento global del sistema al poder producirse bloqueos que en algunos casos extremos podrían ser graves llegando a paralizar completamente un sistema, con la complejidad añadida de tener el sistema dividido en multitud de microservicios de diferente procedencia. Esto hacen que estas soluciones old-style puedan considerarse inadecuadas o al menos difíciles de implementar en una arquitectura de microservicios.

4.1. Rediseño hacia el monolito. (marcha atrás)

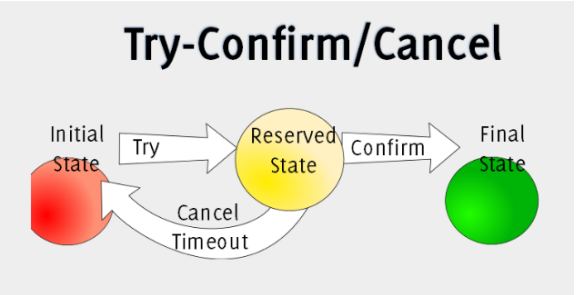
Tras la complejidad que ofrecen técnicas tradicionales para la gestión de la transaccionalidad, se puede tener la tentación, en nuestro ejemplo, de dejar Polizas e HistorialMedicos bajo la misma base datos SQL y así las transacciones serán automáticamente ACID sin tener que andar con complicados diseños distribuidos pero claro, eso sería volver al **monolito** y lo que queremos es conseguir las ventajas que ofrece una arquitectura basada en microservicios, Además, podría ser el caso que alguno de los microservicios persistan sus datos en bases de datos no SQL.

Por lo tanto, no hay marcha atrás posible, hay que continuar analizando más alternativas.

4.2. Sistemas de compensación de transacciones

Existen múltiples frameworks que usan **BPEL** (Business Process Execution Language) que soportan el diseño de flujos de trabajo (workflows) para implementar procesos complejos que, sin ser transacciones ACID, simulan una transacción realizando compensaciones a posteriori, si es necesario. Estos workflows no son fácilmente escalables y su complejidad va aumentando según se incrementa el número de intervinientes. Los caminos de error suelen ser complejos y difíciles de reproducir, aumentando exponencialmente los errores heurísticos posibles en cualquier transacción.

En este sentido atomikos propone una vuelta de tuerca más con su patrón **TCC** (Try Cancel Commit: (10) (11)) que hace más sencillo el trato de estos flujos de compensaciones. Es un patrón basado en el protocolo Two Phase Commit en el cual se obliga a implementar a los servicios intervinientes un estado preparado o reservado que pasará a un estado definitivo cuando se ejecute el cancel o el commit según corresponda. Mediante este protocolo se evita parte de los bloqueos del 2PC puesto que el estado preparado no es bloqueante. Eso sí, hay que tener en cuenta que es un sistema de compensaciones por lo tanto no se cumple el paradigma ACID. Pero, usando TCC los flujos de compensación son más sencillos y no hay tantos caminos de error.



4.3. Transacciones REST-TA

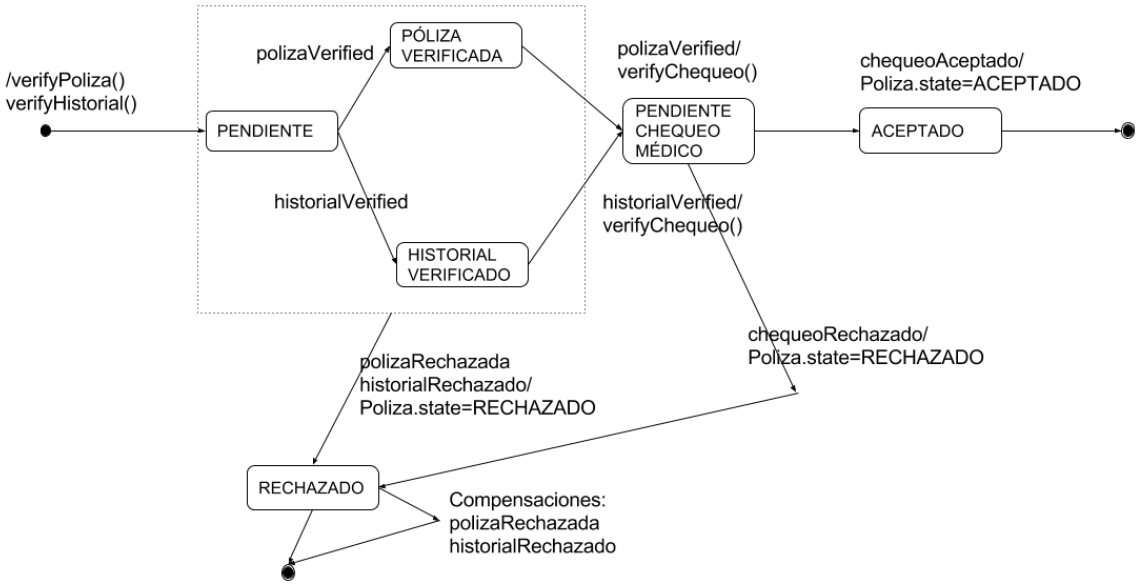
Ha sido desarrollada por RedHat, pero aún no ha salido de la fase de borrador. Sin embargo, es compatible con el servidor de aplicaciones WildFly y está listo para ser usado. Este estándar permite usar el servidor de aplicaciones como un coordinador de transacciones con una API REST específica para crear y unirse a las transacciones distribuidas. Los servicios web RESTful que deseen participar en la transacción en dos fases 2PC también deben admitir una API REST específica. Desafortunadamente, para unir una transacción distribuida a los recursos locales del microservicio, todavía tendríamos que implementarlos en una sola plataforma JTA o escribir este puente nosotros mismos, una tarea no trivial.

La compañía Atomikos propone un modelo de orquestación de transacciones para servicios REST en la que no se realizan compensaciones, pero en la que cada uno de los servicios intervinientes tienen que tener bases de datos capaces de ejecutar transacciones ACID (12). Proponen un gestor transaccional distribuido que es el encargado de mantener los enlaces a la transacción ACID global. En el artículo referenciado prometen que el acoplamiento es muy ligero y que el producto empleado (**ExtremeTransactions**) es escalable y preparado para el Cloud.

5. Enfoque orientado a dominio

Como hemos comentado previamente, es importante realizar un buen diseño de nuestros microservicios, porque muchas transacciones y/o orquestaciones realmente no son necesarias cuando nuestros microservicios están correctamente diseñados (5).

En nuestro ejemplo, es posible separar la inserción del informe médico del alta de la póliza, pero supongamos que dependiendo del contenido del informe médico la póliza tiene un valor u otro. Desde un punto de vista puramente funcional existe una interrelación entre la póliza y el informe médico. Y se establece una máquina de estados para la entidad póliza que va cambiando en función de la presencia o no del informe médico y de su contenido.



5.1. Patrón Saga

Cuando en una misma transacción de negocio están involucrados varios microservicios de distintos dominios funcionales es necesaria cierta orquestación. En nuestro caso de uso supongamos que se permite la entrega de la documentación firmada para dar de alta la póliza, pero el chequeo médico se realiza de manera diferida y en función de los datos obtenidos, el valor final de la póliza y las coberturas se ven afectados. En el proceso global intervienen varias acciones asíncronas, la firma de los documentos y la realización del informe médico. Cómo hemos hablado anteriormente este proceso se puede expresar en forma de máquina de estados. Para orquestar esta transacción completa introducimos el patrón Saga.

"*La idea básica es **dividir** la transacción general en **múltiples pasos** o actividades. Solo los pasos internos se pueden realizar en transacciones atómicas, pero la Saga se encarga de la consistencia general. La Saga tiene la responsabilidad de completar la transacción de negocio general o de dejar el sistema en un estado conocido de finalización. Por lo tanto, en caso de errores, se aplica un procedimiento de negocio opuesto que se realiza llamando a pasos o actividades de **compensación** en **orden inverso***." (1). Es un buen resumen, de este concepto propuesto ya hace mucho tiempo por Héctor García-Molina y Kenneth Salem en (8) y muy común en DDD.

Cómo veremos más adelante las Sagas se pueden implementar como una coreografía o una orquestación de microservicios.

6. Arquitectura propuesta

En esta sesión se proponen varios diseños de arquitectura para resolver la transaccionalidad en soluciones de microservicios, considerando enfoques imperativo y reactivos.

6.1. Arquitectura Reactiva

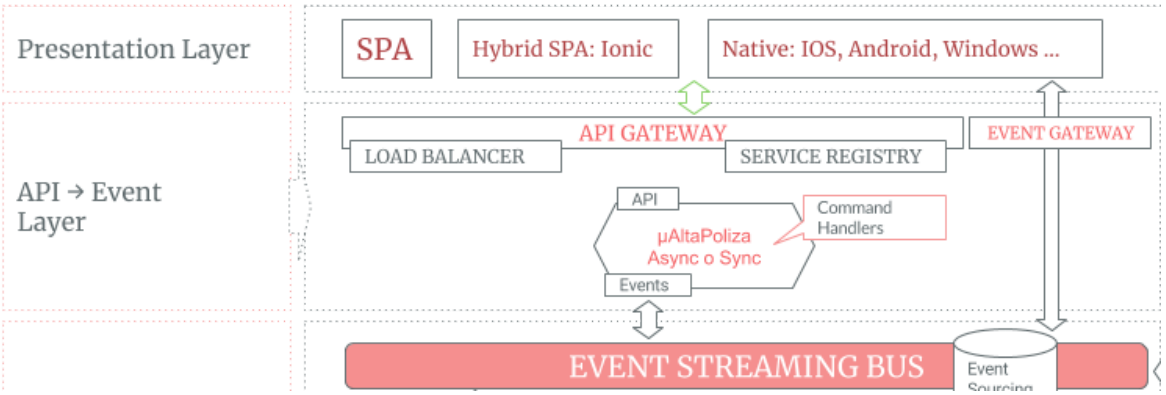
Proponemos una arquitectura **Event Driven**, basada en la forma de afrontar los problemas de diseño orientado a dominio con **DDD**, con las siguientes tres capas:

- **Capa de transformación** de API a eventos donde residen los manejadores de acciones (command handler**)**, que pueden ser **síncronos** y **asíncronos.**
- **Capa de dominio** donde residen los servicios de los distintos dominios y las Sagas que manejan las transacciones. La integración de los servicios dentro de esta capa se realizará de manera reactiva al 100% aunque dentro de cada microservicio la implementación es completamente libre, pudiendo ser reactiva, imperativa, etc.
- **Capa de consulta,** encargada de almacenar la información de la manera más parecida a como va a ser consultada. Escucha eventos de la capa de dominio y expone un API de consulta único. Puede implementar una arquitectura CQRS o no, y será una decisión en función de la necesidad del dominio funcional en el que nos encontremos.

6.1.1. Capa de transformación: API → Event Layer

La capa de transformación incluye un **API Gateway** que se encarga de la seguridad y de enrutar las peticiones de las aplicaciones frontales al servicio adecuado a través del **Service Registry**. Junto con el API Gateway (o por separado) hay un balanceador de carga, **Load Balancer,** que detecta la carga de los microservicios orquestando las peticiones o escalando horizontalmente, si es necesario, un microservicio sobrecargado.

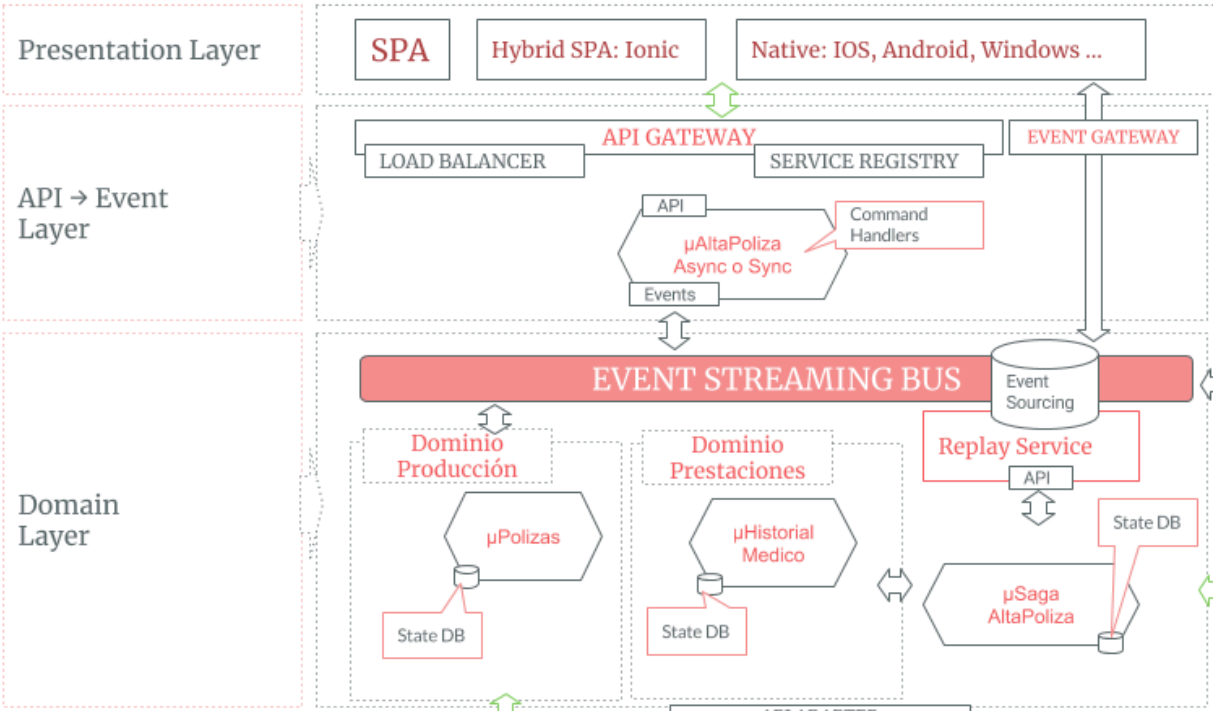
*"El d**espliegue en horizontal** es la replicación de servicios exactamente iguales que se despliegan en caso de un aumento de la carga"*



Los microservicios de esta capa se encargará de una funcionalidad concreta**,** serán los encargados de generar los eventos necesarios para la capa de dominio. Habrá un microservicio por cada uno de las funcionalidades de nuestro sistema y podrán ser **asíncronos** o **síncronos.** En el caso de nuestro ejemplo crearemos un servicio llamado **µAltaPoliza**. Vamos a explicar la solución **Async** como la **Sync** de nuestro ejemplo.

6.1.2. Capa de dominio: Domain Layer

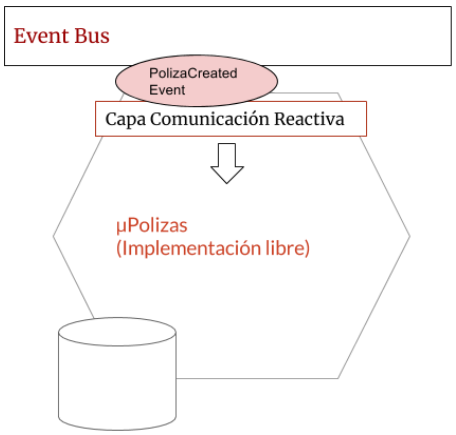
Esta capa incluye un **Event Streaming Bus,** encargado de "orquestar" la comunicación entre todos los microservicios de esta capa. Se trata de una orquestación muy débil, y solamente hace de tubería (**pipeline**) para comunicar el emisor (**publisher**) con los **subscriptores**. También realiza la función de equilibrado de carga (**Load Balancer)** para detectar qué servicio tiene menos carga y si es necesario desplegar más o menos servicios.



En esta capa se definirán los dominios funcionales del sistema. Y según la metodología **DDD,** se establecerán las fronteras de contexto, **context boundaries.** En nuestro ejemplo hemos creado dos dominios uno llamado **producción**, donde estará todo el seguimiento de clientes, siniestros y pólizas, y otro llamado **prestaciones**, donde estará toda la información sobre los médicos y prestaciones que pueden recibir los clientes.

NOTA: E*n un análisis más detallado puede que estos dominios no tengan sentido, o quizá sí, pero valen para ilustrar nuestro ejemplo*. En estos dominios lógicos residirán cada uno de los servicios de nuestro ejemplo µPolizas y µHistorialMedico.

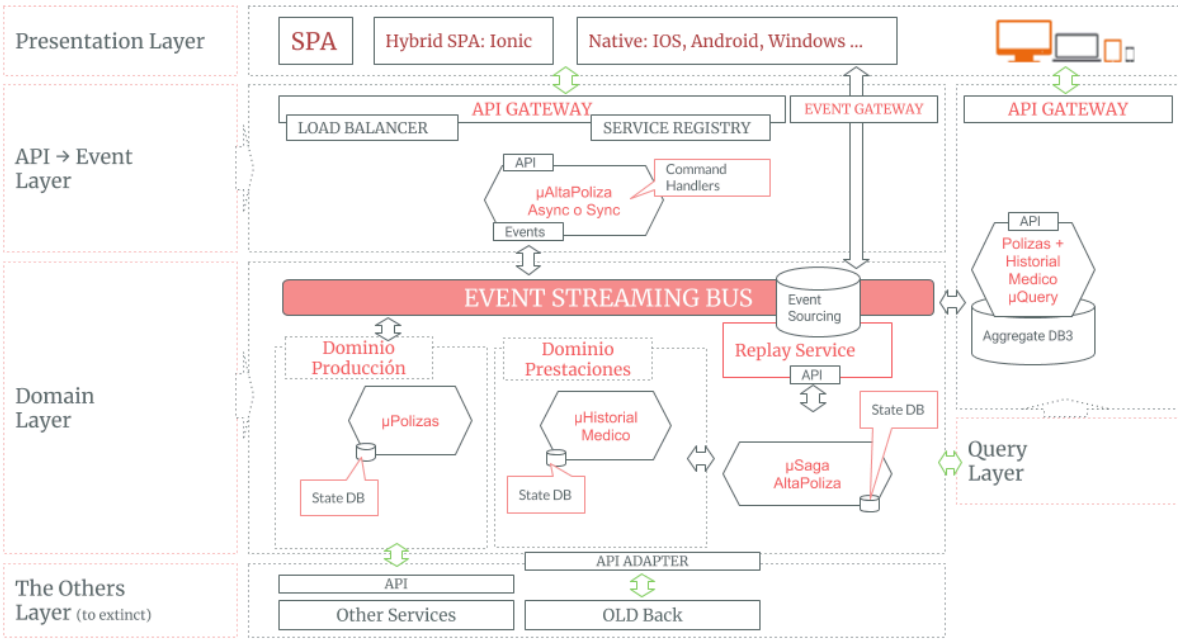
Los servicios en esta capa cumplen el **manifiesto reactivo** en su **capa de comunicación**. Es importante hacer notar que, dentro, los microservicios, pueden estar desarrollados de la manera más conveniente para cumplir con sus funcionalidades, es decir, de manera reactiva o no. Esto sólo significa que los microservicios solo reaccionan a eventos a los que se encuentren subscritos, es decir, no pueden ser invocados directamente a través de ninguna interfaz.



6.1.3. Capa de consulta: CQRS opcional

No es obligatorio que exista un sistema diferenciado de consulta para todos los microservicios de todos dominios funcionales. Los microservicios de dominio podrían exponer un API únicamente de consulta que se indexaría en el Service Registry del API Gateway. Pero, es importante resaltar que este API es solamente de consulta.

En el caso de una capa de consulta (Query Layer) con un enfoque **CQRS** (Command Query Responsibility Segregation), ésta recibe los eventos y almacena la información preparada para ser consultada de manera eficiente. De esta forma, los datos de estado y los de consulta están en sistemas diferenciados y se actualizan a través de la escucha de eventos de dominio y de aplicación. Esta sería la visión global de la arquitectura lógica propuesta.



Los microservicios de esta capa recibirán los eventos ya procesados y persistirán los datos preparados y optimizados para las consultas necesarias. Las bases de datos más convenientes para esta capa son bases de datos noSQL optimizadas para la consulta como mongoDB o Cassandra.

6.1.4. Event Gateway, Event Sourcing y Replay Service

Son elementos opcionales de la arquitectura:

- **Event Gateway:** permite la emisión e inyección de eventos directamente desde las aplicaciones frontales en el bus de eventos de la aplicación. Es un elemento lógico que puede estar implementado en la misma infraestructura que el API Gateway. No es más que una pasarela que aporta seguridad y ofrece un canal para publicar y emitir eventos desde el Event Bus de la aplicación. También podría enviar correos, SMS o notificaciones push para realizar las comunicaciones asíncronas.
- **Event Sourcing:** almacena la totalidad de los eventos emitidos y recibidos dentro del sistema permitiendo reproducir el estado en cualquier momento. Más adelante, en el documento, se profundizará más sobre este concepto.
- **Replay Service:** es un servicio de arquitectura que ofrece utilidades sobre la base de datos de event sourcing a los distintos microservicios o elementos de la arquitectura.

6.1.5. Eventos Inmutables: El único elemento de integración

Los eventos serán la única ligadura entre los microservicios. Por lo tanto, hay que tener mucho cuidado a la hora de diseñar los eventos de una aplicación. Mediante la metodología Event Storming propuesta en (13) se establecen todos los principios de un enfoque DDD. En un primer lugar es necesario establecer un **Ubiquitous Language** (UL) entre el negocio y el desarrollo de una aplicación. Los eventos son parte de este lenguaje ubicuo.

A continuación, se incluyen recomendaciones para el diseño e implementación de los eventos.

Diseño de los eventos:

1. Los eventos son verbos en pasado.
2. Los **eventos** son **inmutables.** Es mejor crear un evento nuevo que modificar un campo de un evento ya existente.
3. Se pueden crear eventos libremente, pero es conveniente mantener un **catálogo** bien documentado.
4. Es conveniente garantizar la compatibilidad hacia atrás.
    a. Se pueden añadir nuevos campos a un evento ya existente.
    b. Nunca eliminar un campo en un evento ya existente.
5. Al leer eventos usar el paradigma Tolerant Reader (4), es decir, solo leer los datos de los eventos que necesitamos. No comprobar la estructura de datos completa de los eventos.

Todos los eventos deberán ser **idempotentes** debido a que en ocasiones será necesario volver a lanzar los eventos en un proceso de reintento. Idempotente significa que si falla la operación esta podrá repetirse de manera segura en el futuro, es decir, el hecho de hacer una cosa dos veces no altera el resultado final.

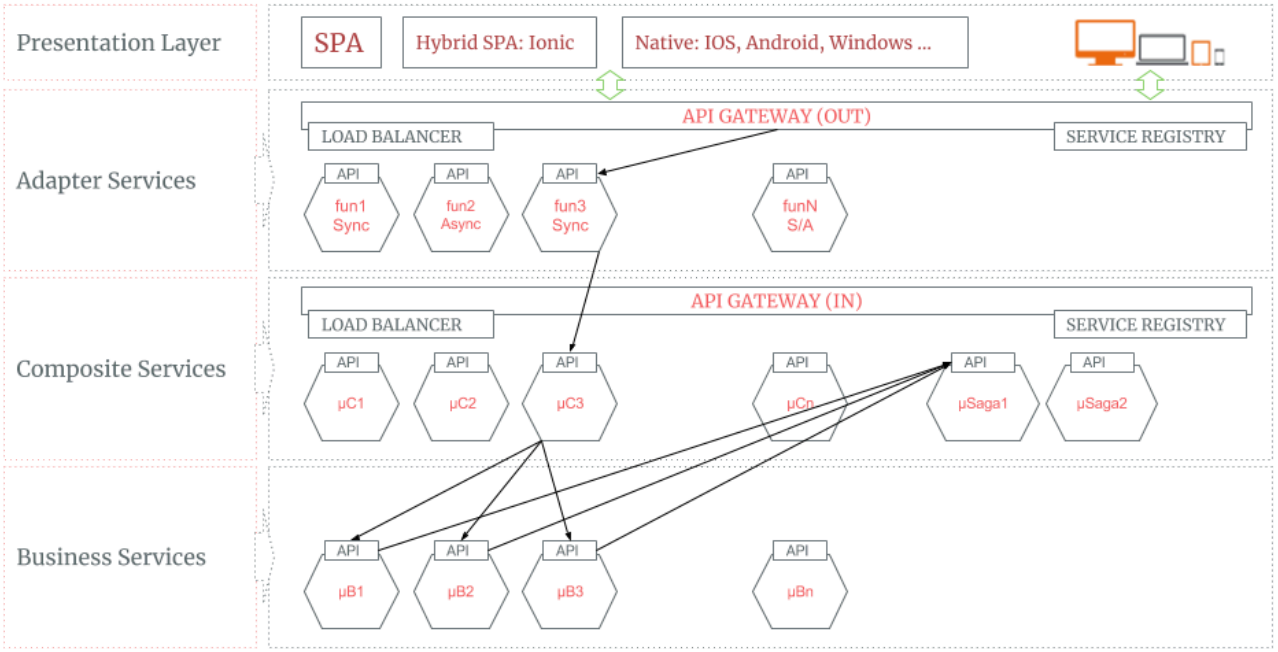
Implementación de los eventos:

La implementación más sencilla para estos eventos podría ser mediante JSON. En cualquier caso, lo más conveniente es que estén implementados en texto plano para no dependen de una librería de serialización y deserialización. Tampoco es conveniente acoplar los eventos a un lenguaje determinado.

6.2. Arquitectura Imperativa

En esta visión de la arquitectura las operaciones tienen que estar dirigidas a un microservicio en concreto, y a continuación se enumeran las principales diferencias, respecto al enfoque reactivo:

- Los servicios deberán tener en su configuración las dependencias concretas a los servicios que necesiten. Estas dependencias se resolverán con ayuda de un Service Registry.
- Es necesario usar un Service Registry interno donde los servicios se inscriban con sus operaciones (endpoints) correspondientes para poder ser accedidos.
- Las sagas actuarán de manera proactiva almacenando la información de las dependencias que tienen con todos los microservicios del negocio que manejan.
- No es una arquitectura que admita CQRS per se. Se podrá implementar en un microservicio concreto si este emite eventos locales.
- Pull en lugar de push. Los cambios de estado de los microservicios tienen que ser consultados, y esto fuerza a que algunos procesos tienen que ser síncronos.



6.3. Implementación de la Saga - Consistencia Eventual

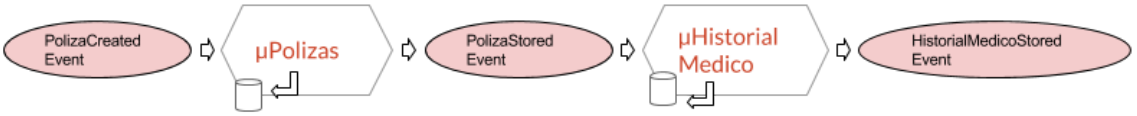
A continuación, se detalla la implementación del patrón Saga, para la gestión de transacciones, diferenciando entre un enfoque de coreografía y de orquestación de servicios.

6.3.1. Saga: Coreografía de microservicios

En este caso no existe un microservicio concreto encargado de la funcionalidad de la Saga. Esta se reparte entre todos los microservicios intervinientes en el proceso. En este caso se establece una correlación entre los distintos eventos de los microservicios

El servicio μPolizas escucha el evento PolizaCreated y procede a almacenar en su base de datos de estado el Póliza emitiendo al terminar un evento **PolizaStored**. El servicio

µHistorialMedico escuchará el evento PolizaStored y almacenará o actualizará su base de datos de HistorialMedicos con los datos de la Póliza, emitiendo al finalizar un evento **HistorialMedicoStored**.



El evento final es HistorialMedicoStored, y una vez emitido la saga concluye la operación completa.

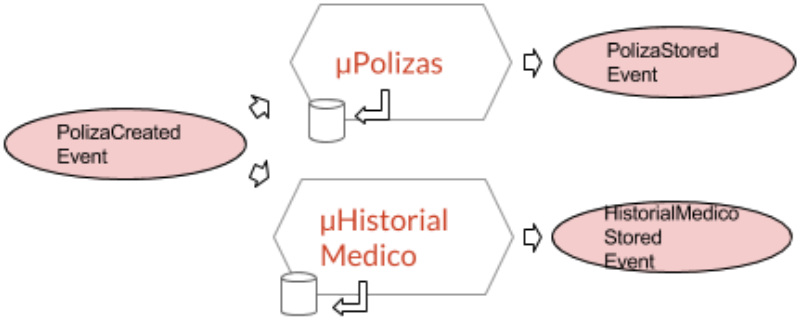
Problemas de implementación de tipo coreografiada (5, pag 99):

- Introducción de dependencias cíclicas entre los microservicios. Un microservicio tiene que estar subscrito a eventos de otro microservicio y viceversa.
- Los objetos de negocio serán más complejos, porque que tienen que almacenar información del estado de la transacción global. Y la lógica de la transacción se difumina entre todos los microservicios intervinientes.

6.3.2. Saga: Orquestación de microservicios

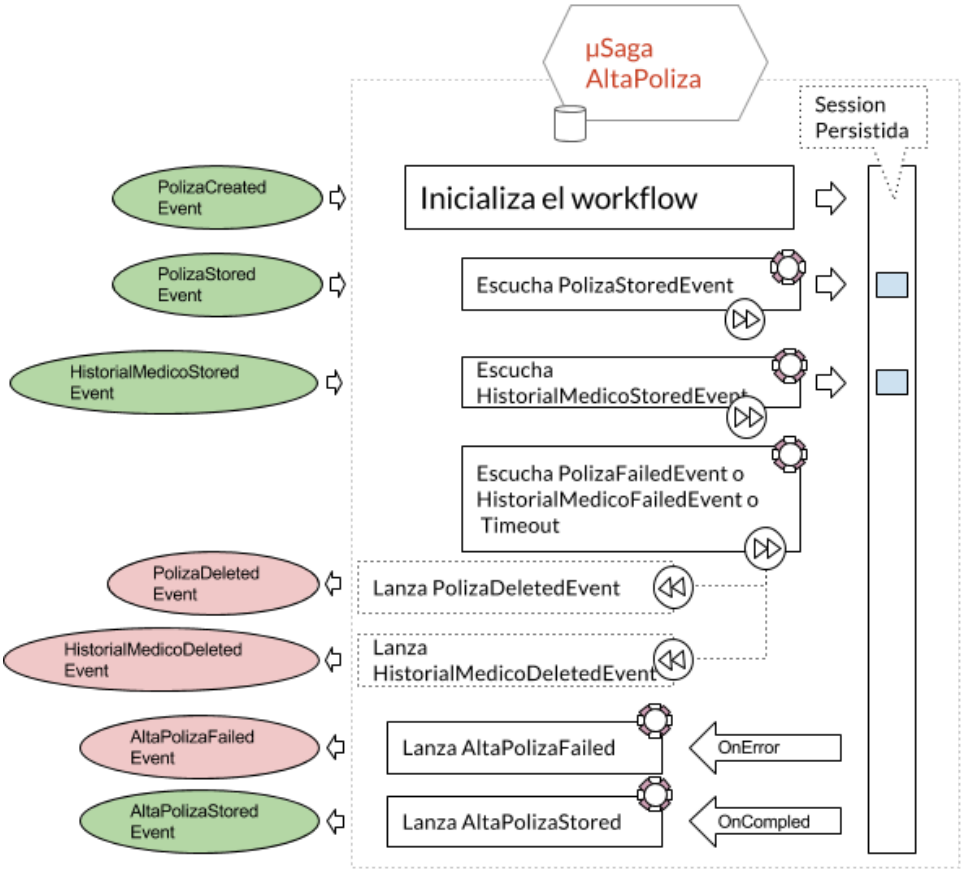
Existe un microservicio encargado de la orquestación completa de la transacción. Este servicio recibe los eventos de todos los servicios intervinientes y emite los eventos necesarios para compensar las operaciones que no queden resueltas.

En nuestro ejemplo los dos servicios escucharán el evento **PolizaCreated,** que les indicará almacenar una Póliza y un HistorialMedico en sus correspondientes subsistemas. Al terminar el microservicio µPolizas emitirá un evento **PolizaStored** y µHistorialMedico un evento **HistorialMedicoStored**.



Destacar que **PolizaCreated** es un **evento de aplicación** puesto que sobrepasa las fronteras de los dominios, en contraposición de los **eventos de dominio** que se mantienen dentro de las fronteras de un dominio específico.

La saga sería un microservicio con el flujo de negocio a nivel de aplicación, e iría emitiendo eventos según fuera recibiendo los eventos correspondientes. Un posible diagrama sería el siguiente:



Los eventos que lanza la Saga deben de ser escuchados por los microservicios µPolizas, µHistorialMedico y µAltaPoliza. Cada uno deberá actuar al recibirlos haciendo la acción de compensación correspondiente:

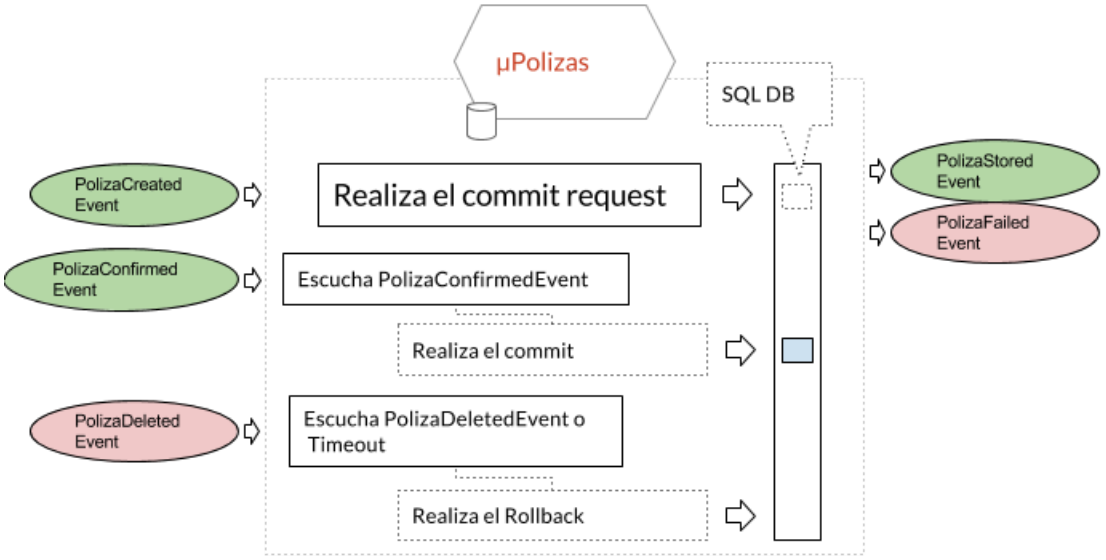
- **PolizaDeletedEvent**: escuchado por µPolizas. Debería eliminar la Póliza
- **HistorialMedicoDeletedEvent:** escuchado por µHistorialMedico. Debe eliminar la HistorialMedico
- **AltaPolizaFailedEvent**: escuchado por µAltaPoliza Sync. Debe informar que el alta de la Póliza ha fallado
- **AltaPolizaFailedEvent**: lanzado a través del Event Gateway a la aplicación frontal que realizó la acción (µAltaPoliza Async). Debe informar al usuario que el alta ha fallado.
- **AltaPolizaStoredEvent:** escuchado por µAltaPoliza Sync. Debe informar que el alta de la Póliza ha funcionado correctamente.
- **AltaPolizaStoredEvent**: lanzado a través del Event Gateway a la aplicación frontal que realizó la acción (µAltaPoliza Async). Debe informar al usuario que el alta ha funcionado correctamente.

Los microservicios Saga pueden estar implementadas por herramientas **BPMN** (Business Process Model and Notation). Por ejemplo, en (11) sugieren el motor de flujos Camuda. Destacar que todavía no se ha evaluado aún esta solución, pero sería interesante incluir este tipo de herramientas ligeras, o similares, para el control de los flujos de Saga. Siempre y cuando no se ponga en riesgo la autonomía e independencia de los equipos de desarrollo dentro de sus dominios, ni implique que este tipo de microservicios Saga tenga que realizarse sólo por equipos especializados.

6.4. Transacciones ACID con Sagas

Si todos los servicios que intervienen en una Saga soportan transacciones e implementan **Two Phase Commit (2PC),** sería posible implementar transacciones ACID, manteniendo la implementación de la saga prácticamente igual. Simplemente la Saga tendría que emitir al finalizar la acción completa eventos del tipo PolizaConfirmedEvent para que los puedan escuchar todos los servicios involucrados en la Saga. Esta solución es muy útil si, por ejemplo, uno de los microservicios es una interfaz con un sistema legado.

El flujo de eventos en uno de los microservicios sería podría ser así:

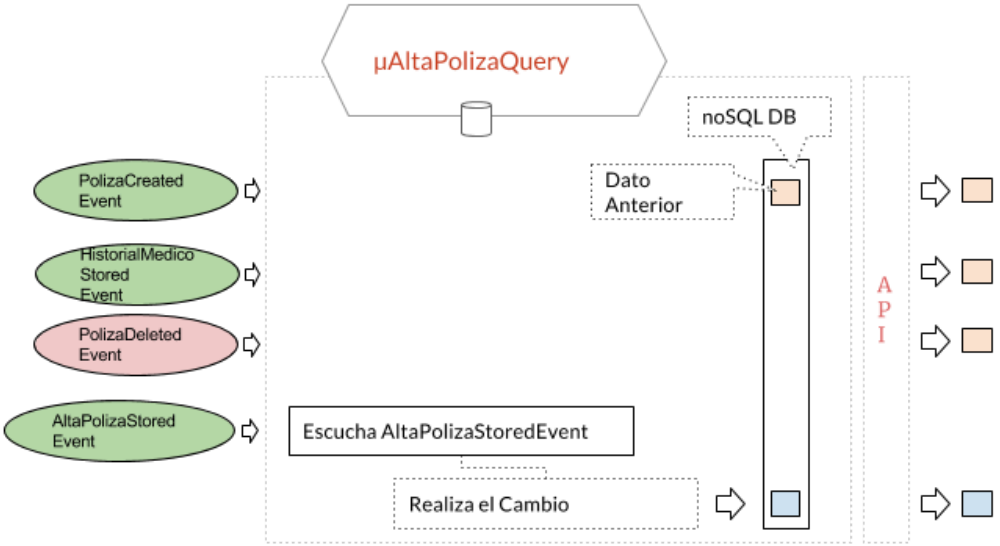


Es importante notar que, aunque esta esta solución es posible, no es conveniente en una arquitectura de microservicios. Como ya hemos comentado previamente, el protocolo 2PC puede originar bloqueos: mientras un dato se encuentra en el estado commit request, el sistema tiene que guardar el estado creando una especie de sesión en la base de datos del microservicio. Este comportamiento no es fácilmente escalable y genera una dependencia del sistema sobre los recursos del gestor de base de datos que realiza estos bloqueos.

En este sentido, la recomendación es tratar con precaución, y a ser posible, evitar, soluciones ACID en las implementaciones de las Sagas, si se quiere una arquitectura escalable basada en microservicios. Se ha reproducido aquí para ilustrar la **convivencia** de sistemas **legados** con la arquitectura propuesta.

6.5. Consistencia de consulta usando CQRS

La base de datos de estado de cada uno de los microservicios intervinientes en nuestro ejemplo podría ser consultada independientemente a través de su API de consulta. Si introducimos una pequeña modificación en la arquitectura y separamos los sistemas de consulta y los sistemas de comando (insert/update) podríamos incluir una funcionalidad nueva a la solución, haciendo que los datos consultados sean consistentes en todo momento.



Destacar que ***toda la información que sale del sistema debe de proceder de la capa de consulta (Query Layer).*** *Por ejemplo,* si un microservicio o una saga necesita consultar un dato lo hace a través del **API Gateway** de la capa de consulta, accediendo al API del microservicio de consulta correspondiente que contenga la información requerida. Esta circunstancia hace que desde un punto de vista global los datos siempre serán **consistentes**. Es decir, los datos consultados siempre son consistentes dado que hasta que la Saga no lanza el último evento de la transacción los datos de consulta no se cambiarán. Mientras dura toda la negociación de la saga el estado de la consulta de Pólizas y de Historial médico será coherente y no será posible encontrar Pólizas sin historial médico y viceversa.

A continuación, se enumera las ventajas e inconvenientes de este enfoque.

Ventajas:

- Consistencia de los datos de consulta en todo momento. Independientemente de cómo esté el estado interno de un microservicio la consulta siempre será consistente puesto que solo se actualiza cuando la saga haya terminado completamente.
- Los dos sistemas se pueden dimensionar de manera independiente.
- Si un microservicio accede a los datos de otros a través de una vista materializada versionable, su nivel de dependencia será más ligero que si se realiza a través de un API que ataca directamente contra los datos internos del microservicio. Las vistas materializadas se pueden versionar y archivar más fácilmente que la base de estado de un microservicio.

Inconvenientes:

1. El mantenimiento de las vistas materializadas puede ser costoso.
2. Los programadores tienen que trabajar más creando un sistema de consulta independiente del sistema de inserción.
3. La actualización de las vistas materializadas se hace a través de eventos, si estos eventos no se lanzan las vistas materializadas no se actualizan.
4. La carga inicial de un sistema separado de consulta puede ser costosa sobre todo si convivimos con sistemas legados.

Posibles soluciones a los inconvenientes planteados pueden ser:

1. Un buen diseño DDD y un buen modelado de eventos mitigaría estos inconvenientes. Metodologías de modelado de eventos como Event Storming podría ayudar en este punto (13)
2. Sería posible la construcción de automatismos y utilidades de arquitectura que mitigara estos problemas. La automatización de la generación de un entorno de consulta separado es posible y sería de ayuda.
3. Si los eventos y las inserciones en la base de datos de estado del microservicio están dentro de la misma transacción la posibilidad de que los eventos no se lancen bajan drásticamente mitigando mucho este inconveniente. (7)
4. El volcado de información de los sistemas legados podría realizarse de una manera gradual, creando sistemas proxy de consulta que realicen consultas a los sistemas legados y almacenen la respuesta a modo de caché en la parte Query según se vaya obteniendo. Esta solución es suficientemente compleja y requiere de un análisis detallado.

6.6. Consulta de datos agregados

Este documento se ha centrado en la inserción y la modificación de datos, y no se ha tratado la consulta de datos agregados. Este apartado sólo tiene como objetivo ser una introducción a las diferentes alternativas de cómo poder abordar este tipo de consultas. Un análisis más detallado se abordará en otro documento.

A continuación, se enumeran opciones para afrontar esta necesidad en arquitectura de microservicios.

6.6.1. Microservicios Composite

- Los microservicios de esta capa hacen llamadas a las API de consulta de varios microservicios de negocio.
- Se agrega la información y se muestra de manera conveniente para ser consumida por las aplicaciones frontales.
- La consulta se realiza en el momento y los datos se generan al vuelo.
    - El uso de caché de consulta no es fácil. Esto es debido a que generalmente los servicios de inserción y actualización son individuales por cada microservicio de negocio. Los servicios agregados de consulta se generan cuando una aplicación necesita agregar cierta información por lo tanto están separados y desacoplados los unos de los otros.
- Cuando los datos agregados son de varios dominios funcionales el dueño de estos microservicios es el equipo que desarrolla la funcionalidad de la aplicación.
- Más indicado para un modelo de arquitectura imperativa, pero también se pude emplear en una arquitectura orientado a eventos.

6.6.2. Vistas materializadas

- Generalización del modelo CQRS en el que se pueden guardar datos compuestos a partir de datos de varios dominios funcionales.
- Se agrega la información, que se muestra de una manera adecuada para ser consumida por las aplicaciones frontales.
- Solo son posibles en un modelo de arquitectura orientado a eventos.
- Se actualizan en tiempo real escuchando los eventos del dominio.

- Cuando los datos agregados son de varios dominios funcionales, el dueño de los microservicios que actualizan las vistas materializadas es el equipo que desarrolla la funcionalidad completa de la aplicación.
- El modelo y el sistema de consulta es completamente independiente de los sistemas y modelos de inserción/modificación.

6.6.3. Procesos Batch

- Actualizan los lagos de datos de una manera periódica.
- La información no se actualiza en tiempo real.
- Analítica de datos: procesado de grandes cantidades de información.
- Se agrega la información y se muestra de una manera adecuada para ser consumida por las aplicaciones frontales.

6.7. Event Sourcing

Martin Fowler: "*Event Sourcing garantiza que todos los cambios en el estado de la aplicación se almacenan como una secuencia de eventos. No solo podemos consultar estos eventos, también podemos usar el registro de eventos para reconstruir estados pasados, y como una base para ajustar automáticamente el estado para hacer frente a los cambios retroactivos*." (14)

Event sourcing se basa en la persistencia de un diario o "log" de todos los eventos ocurridos en una aplicación ordenados en la posición exacta en la que han ido ocurriendo. El estado de la aplicación puede ser calculado en cualquier momento recorriendo la sucesión completa de eventos que han tenido lugar hasta entonces. Es un protocolo con **cero pérdidas**. Esto significa que la recuperación es simple y eficiente, ya que se basa completamente en un diario, o en un registro ordenado.

¿Qué aporta Event Sourcing?

- ***Recuperación en caso de desastre.*** Ningún sistema perfecto y además del happy path existen gran cantidad de casuísticas que pueden hacer fallar un sistema. El almacén de eventos de Event Sourcing podría hacer recuperar el estado de nuestro sistema justo antes de un desastre y poder aplicar las medidas adecuadas para solucionarlo (15)(16).
- ***Análisis retroactivo.*** Al poner en producción un nuevo microservicio con una nueva funcionalidad será fácilmente posible volver a procesar todos los eventos ocurridos antes de haber puesto el servicio en producción (14).
- ***Monitorización de todo lo que ocurre en una aplicación.*** Todas las acciones que ocurran en una aplicación estarán auditadas y monitorizadas. (14)
- ***Pruebas de escenarios reales.*** Hace posible la creación de escenarios reales de prueba que podrán reproducirse todas las veces que sea necesario para probar nuevos servicios o actualizaciones.
- ***Posibilita los modelos paralelos.*** Se trata de recrear el estado de una aplicación en un momento concreto del pasado o del futuro y comprobar cómo reaccionó o reaccionaría nuestro sistema (17)

- ***Reprocesado para consulta***. En una arquitectura CQRS el Event Sourcing no es totalmente necesario, pero si es muy aconsejable haciendo posible la creación de nuevas vistas en función de eventos ya ocurridos.

Requerimientos de para su implementación:

- ***Resiliencia****: Prueba de fallos. La caída de un nodo del event bus no puede suponer la pérdida de ningún evento.*
- ***Escalabilidad****: Tiene que ser escalable horizontalmente.*
- ***Rendimiento:*** *El número de eventos puede ser enorme la herramienta tiene que soportar un flujo muy alto de eventos por segundo.*
- ***Consultable:*** *Los eventos persistidos tienen que ser fácilmente consultables.*

Kafka es la herramienta que más se ajusta a estos requerimientos. Además en las últimas versiones viene de forma nativa un método de persistencia distribuida que permite el almacén y posterior consulta de cualquier evento procesado en cualquiera de sus colas (**18**). Con Kafka es posible realizar consultas para obtener cualquier evento o conjunto de eventos que haya ocurrido en el pasado. A través de un servicio de arquitectura que llamaremos **Replay Service,** cualquier Saga o microservicio puede volver a calcular su estado en un momento determinado del tiempo mediante el procesamiento ordenado de todos los eventos ocurridos.

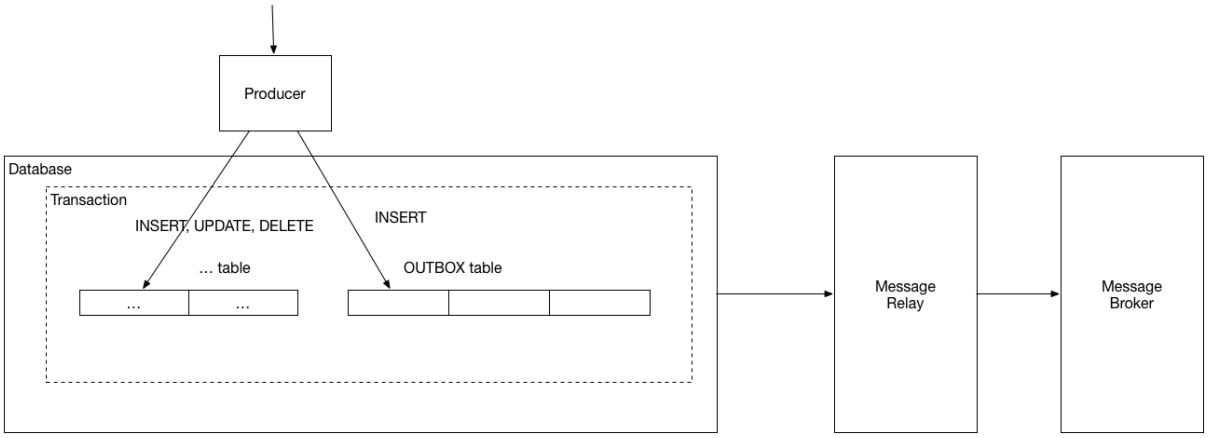
6.8. Transacciones In-Service (ACID)

Dos o más acciones concretas dentro de una misma operación de negocio se tienen que poder realizar completamente o no realizarse ninguna de ellas. Es el concepto tradicional de transacción en el mundo del software. El ejemplo más claro es la inserción de un dato en una tabla y el update de una tercera dentro del contexto de una misma transacción SQL:

- Se realizan en un corto periodo de tiempo.
- Se tienen que garantizar la completitud de la operación.
- Es posible hacer Rollback de todas las acciones que componen la operación completa.

Este tipo de transacciones seguirán existiendo en una arquitectura de microservicio dentro de un mismo microservicio. Y seguirá siendo necesario realizar operaciones compuestas de varias acciones que tengan que cumplir una transacción ACID al 100%.

Por ejemplo, si implementamos una arquitectura orientada a eventos, la inserción en la base de datos y el lanzamiento del evento tiene que formar parte de una transacción (5).



No se va a profundizar mucho más en este concepto y simplemente se recoge aquí para que destacar que las transacciones ACID dentro de los procesos van a seguir siendo útiles dentro de la implementación de un microservicio.

7. Referencias

A continuación, se incluye el listado de referencias que se han tenido en cuenta durante el análisis y diseño de la arquitectura detallada en este documento:

- (1) https://blog.bernd-ruecker.com/saga-how-to-implement-complex-business-transactions-without-two-phase-commit-e00aa41a1b1b
- (2) http://www.grahamlea.com/2016/08/distributed-transactions-microservices-icebergs/
- (3) http://www.baeldung.com/transactions-across-microservices
- (4) https://martinfowler.com/articles/microservices.html
- (5) https://www.manning.com/books/microservices-patterns
- (6) https://stackoverflow.com/questions/30213456/transactions-across-rest-microservices
- (7) https://github.com/eventuate-tram/eventuate-tram-core
- (8) https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf
- (9) http://searchmicroservices.techtarget.com/tip/Using-Web-services-as-distributed-transactions-and-the-role-of-XA
- (10) https://www.atomikos.com/Blog/TransactionsForTheRESTOfUs
- (11) https://www.infoq.com/presentations/Transactions-HTTP-REST
- (12) https://www.atomikos.com/Blog/ACIDTransactionsAcrossMicroservices
- (13) https://techbeacon.com/introduction-event-storming-easy-way-achieve-domain-driven-design
- (14) https://martinfowler.com/eaaDev/EventSourcing.html
- (15) https://martinfowler.com/eaaDev/RetroactiveEvent.html
- (16) http://www.grahamlea.com/2016/08/distributed-transactions-microservices-icebergs/
- (17) https://martinfowler.com/eaaDev/ParallelModel.html
- (18) https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/