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

Curso académico 2021/2022

Trabajo de Fin de Máster

Saga coreografiada con eventos y consumidores

Autor: Miguel García Sanguino
Tutor: Micael Gallego Carrillo

Universidad
Rey Juan Carlos

1.1.1 Índice de contenidos

1.1.2 Índice de ilustraciones

2 Introducción

Cuando tenemos una transacción con microservicios, tanto en middleware como en frontend, se puede complicar mucho una transacción muy sencilla. En el frontend se suele devolver poca información, o usar métodos lentos y costosos para actualizarla. En el middleware se suele complicar y acabar con servicios muy acoplados y complejos para algo que en un monolito podían ser pocas líneas de código.

El patrón saga[1] es un patrón para asegurar la consistencia de una transacción distribuida entre microservicios. El patrón en sí se basa en crear estructuras de los servicios para los caminos en los que todo va bien y también para cada posible acción en la que la transacción falla y se debe dar marcha atrás a las acciones que ya se hayan ejecutado por servicios anteriores. Existen varias formas de implementar las sagas. Una de las más usadas es que un servicio orqueste los pasos de la transacción y controle el estado, decidiendo las acciones siguientes o las operaciones de marcha atrás en caso de fallo. Es cierto que globalmente la hace más sencilla y controlable, pero acopla mucho los servicios unos con otros.

En general el uso de microservicios tiene más sentido cuando estamos en proyectos de un tamaño considerable, en el que tenemos un número de equipos importante donde el gobierno entre ellos requiere de cierta independencia para poder ser ágiles. De ahí que piense que la saga con orquestación pueda traernos una dificultad de gobierno extra, aunque globalmente sea más sencilla.

Además, por experiencia, este tipo de transacciones dejan una mala experiencia de usuario. Se suele contestar con la primera parte de la saga y el cliente no tiene la respuesta completa. Se va a investigar la mejor forma de iniciar y consumir las actualizaciones de la transacción.

3 Objetivos

Como se ha comentado en la introducción, se quiere buscar una manera de simplificar las sagas y su consumo. Para ello se va a profundizar en las herramientas y técnicas vistas en el máster, teniendo en cuenta dos objetivos principales:

- Profundizar sobre la construcción de transacciones con microservicios coreografiados.
- Investigar y plantear la conexión de consumidores a procesos asíncronos largos como el de una transacción con microservicios.

Cuando nos plateamos una transacción en la que interviene más de un microservicio, tenemos un problema principal: en qué momento podemos fijar en base de datos la transacción. No vamos a poder sincronizarlos, por lo que se va a ir guardando pasos de la transacción y en caso de cancelar la transacción, tener la seguridad de que se compensan las operaciones que ya se hubiesen realizado.

Por todo esto, la primera restricción que me he autoimpuesto es que los servicios estén coreografiados y se comuniquen por eventos. Así pueden ser 100% independientes. Cada servicio recibirá eventos con un contrato y emitirá eventos con otro contrato. Cada uno de los servicios

[1] https://www.baeldung.com/cs/saga-pattern-microservices

desconoce los servicios anteriores y posteriores, y desconoce la transacción en sí. De esta manera cada servicio tiene una responsabilidad única y en caso de que fuera necesario añadir o eliminar pasos en nuestra saga, no vamos a necesitar tocar los demás servicios. Así reduciremos la dependencia entre los equipos que estarían manteniendo los servicios y por tanto la necesidad de gobierno entre ellos.

Para el consumidor imagino el mismo escenario: habrá un equipo que mantenga al consumidor y no queremos que esté fuertemente acoplado a la transacción en sí. Por ello, proponemos que este equipo tenga dos piezas, el frontend y un servicio "backend for frontend" (BFF en adelante). El servicio se encargará de dos cosas: iniciar la transacción y escuchar e informar debidamente al usuario de las actualizaciones que realicen los servicios. De esta manera la responsabilidad de conexión front – middleware la tiene el mismo equipo que desarrolla el frontend.

El patrón BFF intenta simplificar la relación entre el front y el middle, y generalmente se propone un servicio BFF por cada consumidor. Aunque solo vamos a tener un consumidor, la parte que nos interesa es en la que el patrón se adapta a los casos de uso específicos de cada consumidor y que es el consumidor el que con total libertad adecúa su API a sus necesidades, no a las del middleware. De esta manera es el BFF el que consumirá los mismos eventos que el middleware y los adaptará para que lleguen al front de la forma que más se ajuste al front.

Este punto me parece muy interesante ya que siempre es un punto doloroso en los proyectos. Por ejemplo, suele ocurrir que no existe un modelo de datos óptimo para ambas capas. Al introducir este servicio intermediario entre front y middleware, permitimos que el front reciba los datos como le venga mejor, y a su vez no modificamos los contratos de los servicios. Esto nos ayudará a cumplir el objetivo de independencia entre capas, ayudando a que los equipos no necesiten constantemente ponerse de acuerdo. Muy probablemente en este punto querríamos poner un test de contrato. Una vez más, al estar el BFF en medio, el test de contrato sería entre servicios, que es más cómodo de montar que con un front como consumidor.

Nos fijamos además varios objetivos extra: que las necesidades del consumidor impacten lo mínimo en el desarrollo de los servicios, que los servicios estén lo más desacoplados entre ellos y que las necesidades de los servicios a su vez también impacten lo mínimo en el desarrollo de los consumidores. Independencia y desacoplamiento en todos los actores.

Por supuesto la escalabilidad, la resiliencia, la mantenibilidad y todas las buenas prácticas aprendidas en el máster están entre los objetivos implícitamente.

3.1 Caso de uso

Como caso de uso se ha elegido una aplicación para pedir comida a domicilio. Nuestra transacción se inicia cuando un usuario realiza un pedido. La transacción debe reservar la comida en el restaurante, reservar un rider para que lo reparta y hacer el pago de comida. Si el restaurante no puede realizar el pedido, no hay riders disponibles o el cliente no dispone de saldo, la transacción debe cancelarse y hacer rollback de las reservas que ya se hayan realizado.

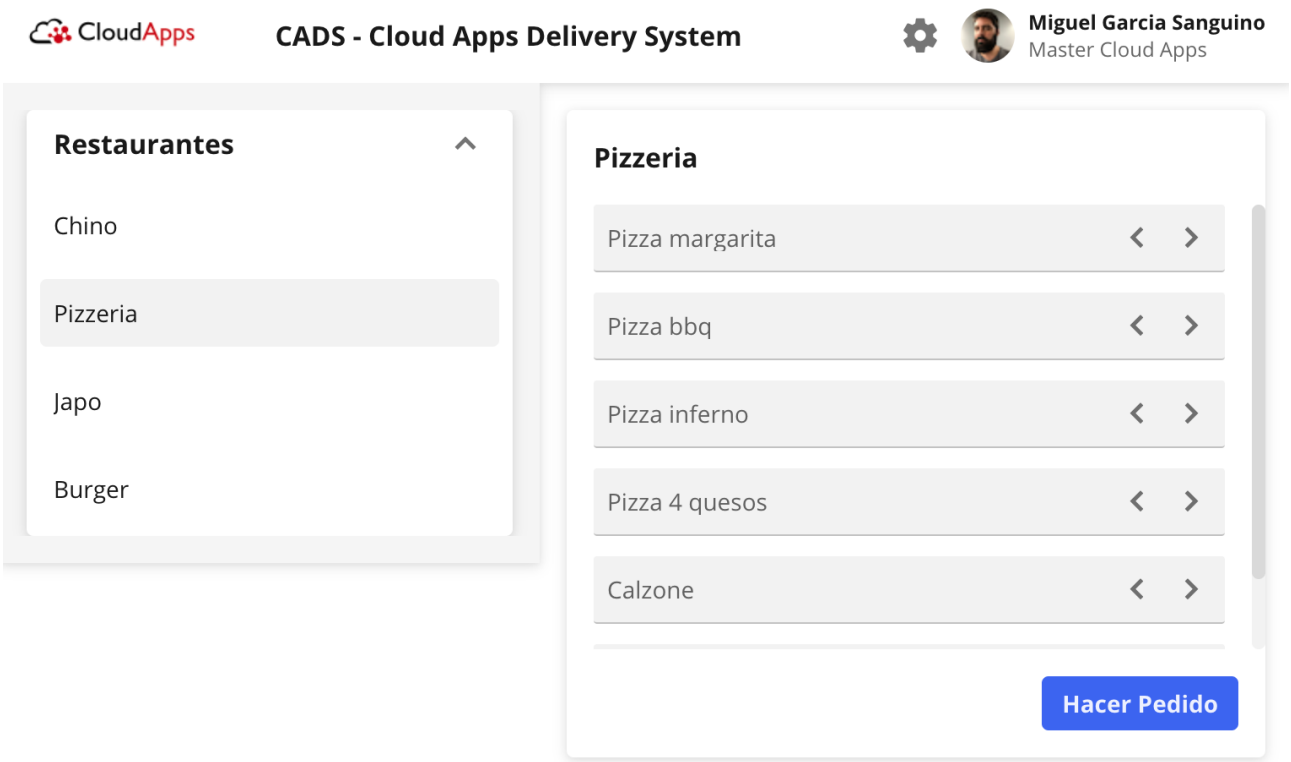


Ilustración 1 - Aplicación que usarán los clientes

Para cada uno de estos pasos vamos a crear un servicio: **Restaurant**, **Rider** y **Payment**. Como intentamos acercarnos lo más posible a un escenario empresarial, vamos a crear un servicio **Order** que solo almacenará los datos del pedido para su explotación, pero realmente no formará parte de la propia saga. Técnicamente podríamos prescindir de él.

Estos servicios son completamente independientes de sus consumidores, y **Order** será quien inicie el proceso y hará de auditor de los cambios con fines de negocio. Cada servicio tendrá la responsabilidad de informar sobre qué ha ocurrido en cada paso de la saga, y como decíamos será el servicio BFF el responsable de la comunicación con el front.

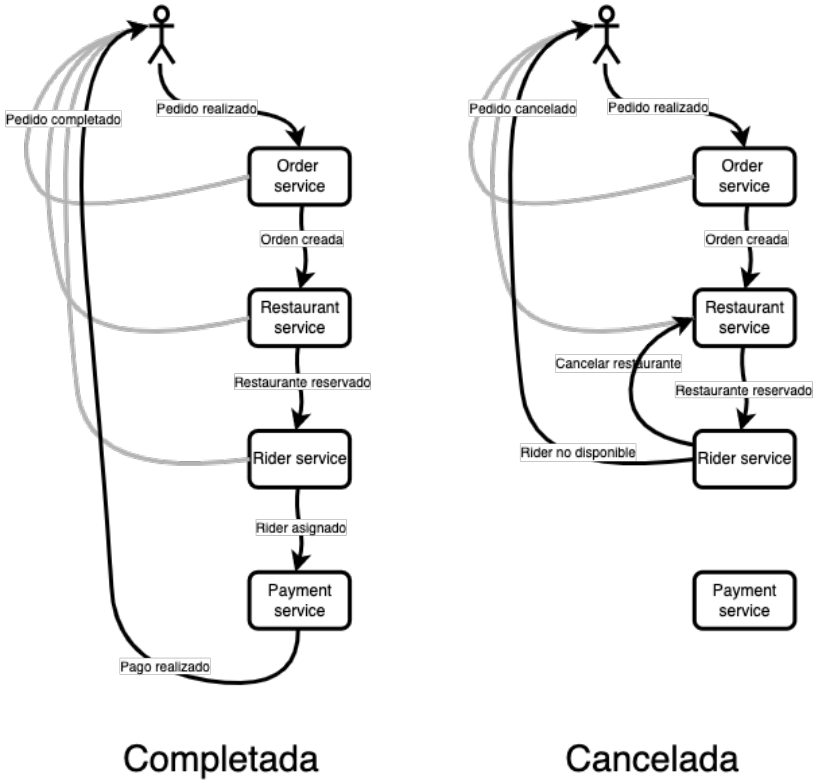
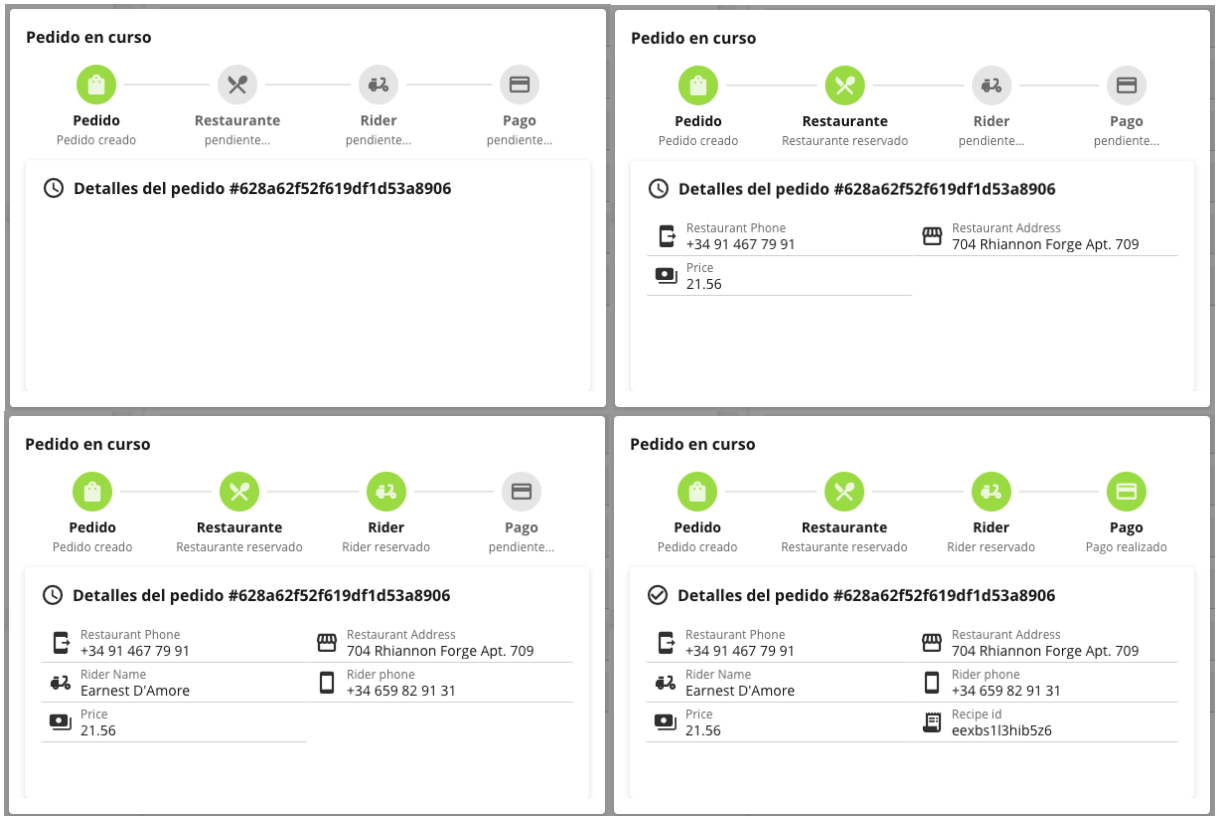


Ilustración 2 - Flujo de transacción completada y cancelada

En la ilustración 2 podemos ver el flujo de una transacción completada y otra cancelada. En el caso de que no se encuentre un rider disponible, se informará al usuario y al servicio de restaurante para que se deshaga la reserva del restaurante. El front en vez de recibir una respuesta única en una llamada REST, recibe cada una de las actualizaciones que van ocurriendo a lo largo de la saga, teniendo actualizado al usuario que será informado en tiempo real.



4 Stack tecnológico

El stack tecnológico se ha elegido con el criterio de cumplir los requisitos y facilitar la implementación. En el máster hemos visto Node y Java aunque se ha utilizado mucho más Java, por lo que se van a implementar los servicios con Node para practicar más esta tecnología.

4.1 Middleware

Ilustración 3 - Stack middleware

Kubernetes: Es el orquestador de contenedores estándar más usado en la industria.

Kafka: Para los eventos se ha decidido usar Kafka porque la forma en la que se conectan los consumidores y se actualizan los offset nos puede ayudar en la coreografía y en el escalado de los servicios.

Node: Como comentábamos, se ha decidido que se implementen los servicios con nodejs.

MongoDB: cada servicio tendrá los datos de su parte de la transacción, únicamente relacionables por el id de la transacción. Al no haber una necesidad de varias tablas ni relaciones entre ella, una base de datos no relacional se nos hacía más sencilla. Además, al igual que node vs java, se ha visto menos en el máster que bases de datos sql.

Kafka.js: se han probado 3 clientes de kafka[2] kafka-node[3] y kafka.js[4]. De los tres el último parece el más estable, el que está más mantenido y el que ofrece más opciones para la tarea que vamos a realizar.

Express: es un estándar bien afianzado y muy conocido. Además, para la parte front no solo es compatible con api rest.

Mongoose: al igual que express, muy utilizado en la industria, funciona bien, y es cómodo para el caso de uso que tenemos.

[2] https://www.npmjs.com/package/kafka
[3] https://www.npmjs.com/package/kafka-node
[4] https://www.npmjs.com/package/kafkajs

4.2 Frontend

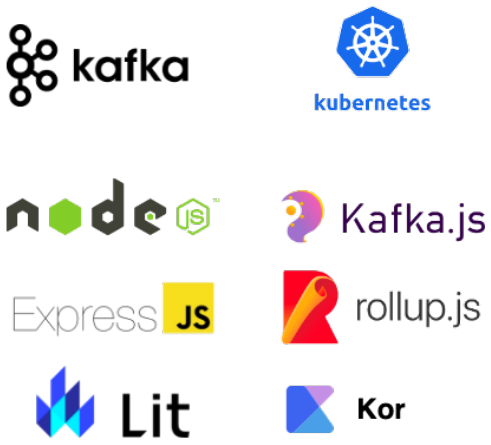


Ilustración 4 - Stack frontend

Kubernetes y Kafka: El servicio BFF aunque está incluido como parte del front porque es mantenido por los desarrolladores front, es un servicio más. Por tanto, el stack de front es en parte compartido con el de middleware, como es el caso de Kubernetes y Kafka.

Node: En el caso de middle podría usarse cualquier otro lenguaje, pero en el caso del BFF es importante que sea node para facilitar a los desarrolladores front su creación y mantenimiento.

Express y Kafka.js: aunque queramos independencia también queremos tener unos estándares, por lo que usaremos en lo más posible el mismo stack que en middle.

Rollup.js: de entre todos los builders que se han estimado (rollup[5], webpack[6] y esbuild[7]), rollup es el que nos ofrecía más facilidad.

Lit: para el frontend se ha decidido realizar usando lo más posible vanilla js, y se ha decantado por el uso de webcomponents. En vez de ir con webcomponents nativos, se ha decidido el uso de Lit porque es una librería muy sencilla y nos evitamos el uso de frameworks como React, Vue o Angular.

Kor: el frontend en sí no es objetivo del presente TFM, por lo que tenía sentido usar un catálogo de componentes que ya tienen estilos. Entre todos los catálogos [8]que hay para Lit, Kor nos ayudará a crear un layout fácilmente y sin complicaciones.

[5] https://www.rollupjs.org/
[6] https://webpack.js.org/
[7] https://esbuild.github.io/
[8] https://github.com/web-padawan/awesome-lit#design-systems

5 Arquitectura

5.1 Middleware

Ya hemos definido bastante los requisitos para nuestros servicios, pero no cómo vamos a desarrollarlos. Tenemos claro que los servicios deben ser coreografiados, y al no tener un mecanismo que controle por dónde pasa una transacción, en caso de caídas, es muy importante que nuestros servicios sean completamente escalables, resilientes por sí mismos y completamente independientes.

Para poder conseguir esto y no tener un código complejo, se ha decidido que los servicios sean idempotentes.

5.1.1 Flujo middleware

El flujo en la capa middleware quedaría como vemos en la ilustración 5. Tan solo tenemos un punto de entrada desde fuera de nuestro flujo: una petición REST al servicio de Order.

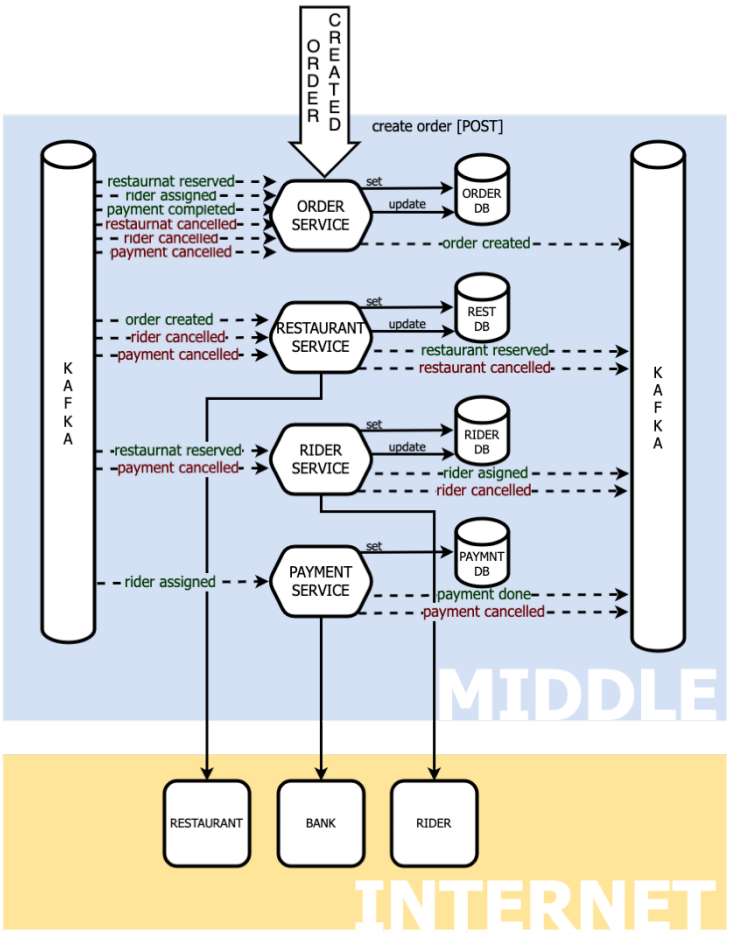


Ilustración 5 - Flujo middleware

(Nótese que en la ilustración 5 se ha pintado dos veces Kafka para que se entienda mejor el flujo de eventos de entrada y de salida)

- Ningún servicio conecta con otro directamente, todos envían y reciben eventos de Kafka.
- Los servicios Restaurant, Rider y Payment reciben un evento de entrada, hacen peticiones externas a los restaurantes, riders o bancos y dependiendo de la respuesta, publican un evento para continuar o hacer rollback, según proceda.
- Restaurant y Rider pueden recibir rollbacks en caso de cancelación de Payments.
- Restaurant puede recibir rollback en caso de cancelación de Rider.
- Order genera el orderId y registra en su base de datos el estado completo de la transacción.

Para simplificar el proyecto y centrarnos en los patrones, no se implementa en los servicios ninguna lógica. Por ejemplo, el servicio Rider debería buscar el rider más cercano y disponible, que tenga capacidad para el pedido y entonces realizar la petición y esperar a que se acepte el envío.

5.1.2 Resiliencia e idempotencia

Para poder desacoplar totalmente los servicios unos de otros, y evitar tener un orquestador que guarde un estado, se ha decidido que los servicios deben ser completamente idempotentes para asegurar que nada se deja de procesar y que nada se procesa más de una vez. En la ilustración 6 podemos ver el flujo interno de un servicio para cumplir estos requisitos.

Cuando un evento es consumido, lo primero que realizamos es comprobar si con ese orderId ya se encuentra en base de datos. En Anexos podemos ver los estados de un pedido, así como los contratos de los eventos que se han usado.

Si no existe en BBDD, se realiza la lógica de negocio correspondiente, se envía el evento de salida con los datos que hemos persistido y por último se marca el evento consumido como leído.

Si ya existe en BBDD lo que hacemos es enviar el evento de salida con los datos que tenemos en base de datos y se marca el evento consumido como leído.

Kafka para esto es bastante cómodo. Se ha creado una base de servicio[9], de la que extienden todos los servicios, para no replicar esta lógica en cada servicio. Lo que hace es guardar el offset actual y lo incrementa según se van consumiendo eventos. Si el offset actual es actualizado, se envía a Kafka para persistirlo y que en caso de caída se retome repitiendo el menor número posible de eventos. En el apartado *5.1.4 Base-service* se comenta más sobre se realiza este proceso.

[9] https://github.com/MasterCloudApps-Projects/Choreographed-Saga-with-Events-and-Consumers/tree/main/base-service

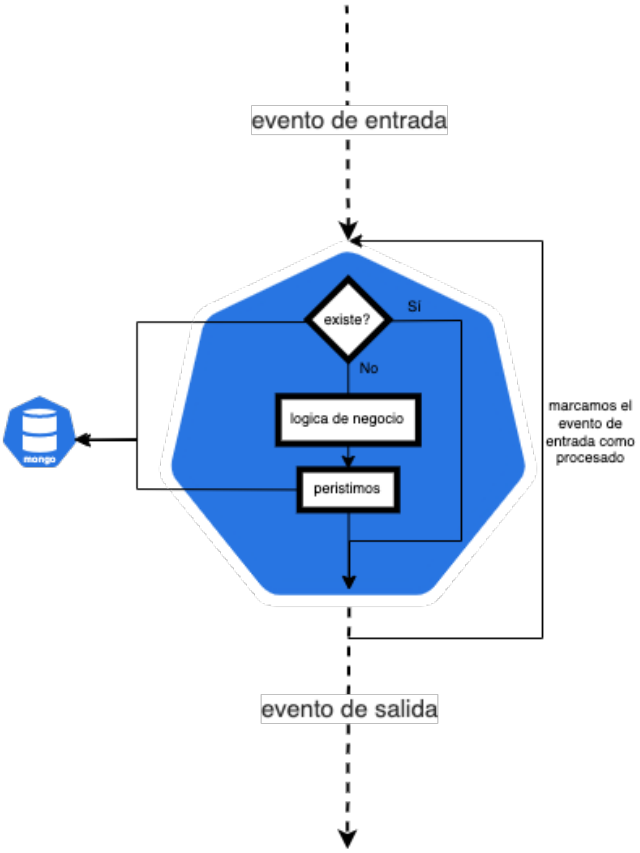


Ilustración 6 - Flujo idempotencia y resiliencia

De esta manera los servicios pueden preocuparse solo de definir los tópicos y la lógica de negocio, abstrayéndose de marcar los eventos como leídos.

La idempotencia tiene otras ventajas además de permitirnos el consumo de eventos con coreografía sin tener una máquina de estados, ni tener que guardar en la BBDD lo que se va procesando. Además de esto, nos ayuda a que los servicios sean más fácilmente resilientes, escalables e independientes:

- Resilientes porque en cualquier momento un servicio puede pararse por un error o manualmente, que al iniciarse otra instancia no tendremos que preocuparnos por cómo se ha parado ni cómo tiene que iniciarse.
- Escalables porque, aunque por escalar acabase consumiéndose varias veces el mismo evento, no pasaría nada. Independientemente de que el escalado sea aumentando o disminuyendo el número de instancias.
- Independientes, realmente cada servicio tiene su responsabilidad y puede abstraerse totalmente del anterior y de los siguientes.

Veamos cómo se cumple la resiliencia analizando en cada momento dónde un servicio puede caerse en medio de una transacción:

- Antes de leer el evento: en este caso no habría problema, aún no se ha hecho nada.
- Durante la lógica de negocio: aún no se ha persistido nada, aunque pueden haberse realizado peticiones externas, por lo que trasladamos la necesidad de idempotencia a los servicios

externos que consumimos. Si se repite la misma operación, no habría problema ya que no hemos persistido nada.

- Después de persistir: si después de persistir se cae el servicio, es porque la lógica ha finalizado y hemos persistido el resultado, por lo que podemos continuar desde este punto. La segunda vez no procesamos la lógica de nuevo pero sí publicamos el evento de salida.
- Después de enviar el evento de salida: en este caso se enviaría dos veces el evento de salida, pero como nuestros servicios son idempotentes, el siguiente no tendrá problema en repetir el mismo evento dos veces.
- Después de marcar el evento de entrada como consumido: en este caso, al volver a levantar el servicio, no se volverá a consumir el evento.

En todos los casos o se continúa donde se quedó o se repite la operación, pero al ser todos los servicios idempotentes no es un problema. Como vemos el servicio es bastante simple y cumple con todos los requisitos que nos habíamos propuesto.

5.1.3 Kafka Mongo connect

Una de las pruebas que se ha realizado es utilizar **Kafka Mongo connect**, un extractor de BBDD que nos permite olvidarnos de enviar los eventos desde el servicio y tener que preocuparnos de si se ha enviado o no después de persistirlo en base de datos.

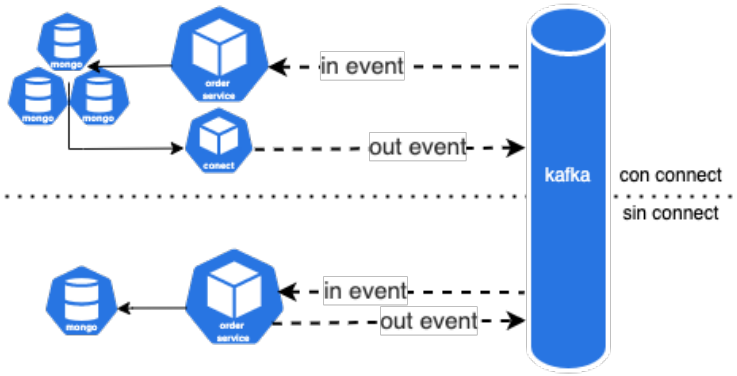


Ilustración 7 - Comparativa con o sin extractor de kafka

A priori nos simplifica la arquitectura para conseguir la idempotencia. Una vez persistido en base de datos solo tengo que marcar el evento de entrada como leído, y si un evento ya lo tenemos en base de datos, no tenemos que hacer nada, ya se habrá encargado o se encargará el extractor de enviar el evento de salida.

Se realizaron pruebas y se tuvo el proyecto funcionando así. En github se puede ver la configuración del extractor utilizado[10] pero finalmente fue descartado y se optó por dejar al servicio que realizase esa lógica.

[10] https://github.com/MasterCloudApps-Projects/Choreographed-Saga-with-Events-and-Consumers/tree/main/kafkaConnectMongoDb

La decisión principalmente se toma por simplificar la infraestructura sin aumentar la complejidad de los servicios. El extractor nos obliga a que las bases de datos Mongo estén montadas con un replica set de al menos 3 instancias. Si a esto le sumamos que cada servicio requiere al menos un extractor, estamos hablando de sumar al menos 3 pods a cada servicio. En cualquier caso, si el proyecto ya se quisiera usar replica set, seguramente habría que volver a valorar el uso de extractores.

Por otro lado, los extractores no han sido tan flexibles como se pensaba en un primer momento. Se han tenido que configurar para que ejecuten cada muy poco tiempo, lo que hace que quizá esta lectura activa haga aún más ineficiente el uso de los recursos, no solo por el número de elementos sino por el consumo de recursos. A priori parece que los extractores tienen más sentido en procesos más offline en los que no se tenga al usuario esperando una respuesta.

También nos hace pensar que deberíamos tener varios extractores corriendo a la vez por cada servicio, por la forma en la que se puede definir qué extraemos y de qué modo lo extraemos. No parece que podamos generar una configuración solo por servicio, sino que seguramente deberíamos tener uno por cada evento o por cada estado. Aunque se consiguiera que fueran en un solo pod por servicio, no deja de ser un hilo por tópico. Aunque según se investigó, es posible que el conector de debezium[11] simplifique varios de estos puntos, pero se decidió no avanzar más en esta dirección.

En cualquier caso, la realidad es que implementar la publicación en el servicio apenas supone esfuerzo, ni le aumenta complejidad, al tener que ser idempotentes ya de por sí, el extractor no aporta valor en este caso de uso. Incluso estando toda la lógica en el servicio, se simplifica la lectura y deja en el servicio toda la lógica y la responsabilidad.

5.1.4 Base-service

Para reutilizar y abstraer la lógica del servicio de la lógica de las comunicaciones y la resiliencia, se ha creado base-service[12]. Se ha creado como dependencia al resto de servicios y tiene el cometido de encargarse de la conexión a Kafka y Mongo. En la base además de la conexión en sí, se realizan las altas de consumidores y productores de Kafka. La base se publica como paquete npm privado en github y es consumido por los servicios como dependencia npm.

Una de las tareas que realiza esta base es la gestión del offset de cada tópico. El offset no deja de ser un puntero al siguiente evento que debe ser leído. Como hemos comentado se marcará el offset después de haber ejecutado toda la lógica del servicio. Normalmente esto no es un problema si los eventos son marcados automáticamente al consumirse. Pero al no saber cuánto tardará cada evento en ser procesado, ni tener la seguridad de que se procesarán en orden, debemos tener cuidado del orden en el que van terminando y no subir a un offset sin que todos los eventos anteriores hayan sido procesados. Por ejemplo, si llegan dos eventos y el segundo termina antes que el primero, no podemos subir el offset al segundo o si se cae en ese momento el servicio, el primer evento nunca se terminaría de procesar.

[11] https://debezium.io/documentation/reference/stable/connectors/mongodb.html
[12] https://github.com/MasterCloudApps-Projects/Choreographed-Saga-with-Events-and-Consumers/tree/main/base-service

Para solucionar esto lo que se hace es gestionar el offset[13], guardando en memoria los offset actuales y asegurando qué offset hay que marcar en cada momento. En el ejemplo que poníamos, el segundo evento quedaría marcado en memoria, y cuando el primero termine, se marcará el offset al segundo.

Puede ocurrir que tengamos eventos completamente procesados antes de que uno anterior, por lo que no se habrá subido aún el offset. Si en ese momento tuviéramos una caída del servicio, al levantarse una nueva instancia se volvería a consumir todos los eventos ya procesados, pero como hemos dicho la idempotencia nos permite despreocuparnos de esta situación.

5.1.5 Conclusiones Middleware

Como resumen, nuestro middleware estará desacoplado de sus consumidores y al mismo tiempo los servicios estarán desacoplados entre ellos. Con ello conseguiremos nuestro objetivo en el que en un supuesto entorno empresarial suficientemente grande pueda gobernarse de forma independiente cada una de las piezas que constituyen nuestra saga y sus consumidores.

Se planteó la necesidad de servicio de Order. Funcionalmente solo genera un orderId, y hace de auditor de los updates en la saga. Por lo que se podría pensar en prescindir de este servicio, pero se ha decidido dejar por varias razones.

La primera es quién se quedaría la responsabilidad de crear el orderId. Podemos dársela al servicio Restaurant, pero ya tendría que exponer un api rest y en caso de que se decida que la saga cambie y se reserve primero el Rider, los cambios serían importantes. Order ayuda a abstraer el resto de la saga del exterior, ya que el resto de los servicios solo se comunican por eventos, sin exponer un api rest. Podemos dársela entonces al BFF, pero aparte de que es una responsabilidad que no le corresponde, debería tener base de datos y pierde totalmente el sentido de BFF. Si además tuviéramos dos frontales diferentes con su BFF cada uno, tendríamos otro problema más grande aún.

La segunda razón es porque en un caso de uso completo es muy posible que el usuario desee ver pedidos pasados. Incluso al abrir una notificación de actualización del pedido, querrá abrirlo y ver el detalle del pedido. También un usuario interno, por ejemplo, en un servicio de atención al cliente, se va a necesitar acceder al estado de los pedidos. Por lo tanto, el servicio Order puede tener todas esas funciones, liberando así de llamadas a los servicios de la saga.

Por todo esto, se decide que el servicio de Order es necesario.

[13] https://github.com/MasterCloudApps-Projects/Choreographed-Saga-with-Events-and-Consumers/blob/main/base-service/src/kafka/offsetManager.js

5.2 Frontend

5.2.1 Estáticos y BFF

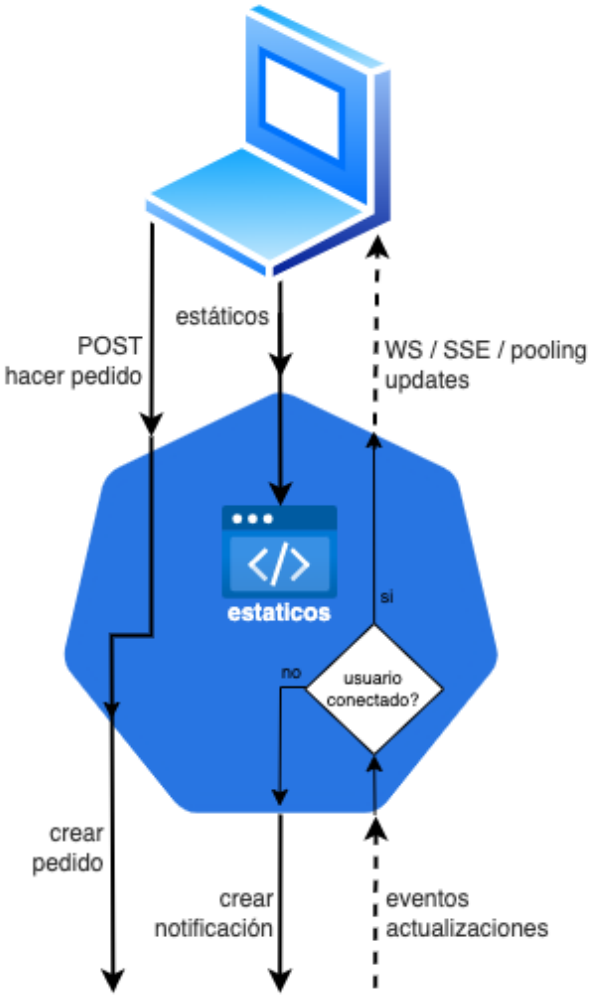


Ilustración 8 - Contenedor Frontend

El mismo equipo que implementará el frontend implementará el BFF. Esto nos ayudará a simplificar el proceso de desarrollo y mantenimiento. Para simplificar también el CI/CD se ha planteado que el mismo artefacto sea el que despliegue el front y el BFF, así no hay problemas de versionado, si se despliega uno se despliega el otro. En el proyecto se ha usado el mismo pod, pero puede ocurrir que por escalabilidad necesitemos más instancias de uno que de otro, por lo que se podrían generar dos pods diferentes sin problemas.

Lo que hacemos es en el express del servicio BFF exponemos los estáticos en la carpeta que habremos construido con rollup. Express además persistirá una conexión desde el BFF con cada usuario conectado para recibir los updates de los eventos.

En caso de recibir un evento para un usuario que no tiene conexión abierta, si es una notificación final, o bien de cancelación o bien ok al pago, se enviará una notificación a través del servicio de notificaciones.

De esta manera, si un usuario se desconecta, se queda sin internet, o se cansa de esperar el resultado, siempre va a recibir una notificación del estado de su pedido. Las notificaciones no se van a implementar, pero si se ha creado el servicio y se generan como sms, email o push de forma aleatoria para simular una aplicación real.

Para construir el frontend se utiliza rollup. Los estáticos del frontend se despliegan junto al BFF. Se usa la misma estrategia que con el resto de los servicios con Docker multistage, lanzando la build de rollup en el stage de build[14] copiando los assets generados en la imagen final.

5.2.2 Flujo Frontend

En el frontend el flujo es mucho más sencillo, ya que tan solo tenemos dos elementos: el front que se ejecuta en el dispositivo del cliente y el BFF.

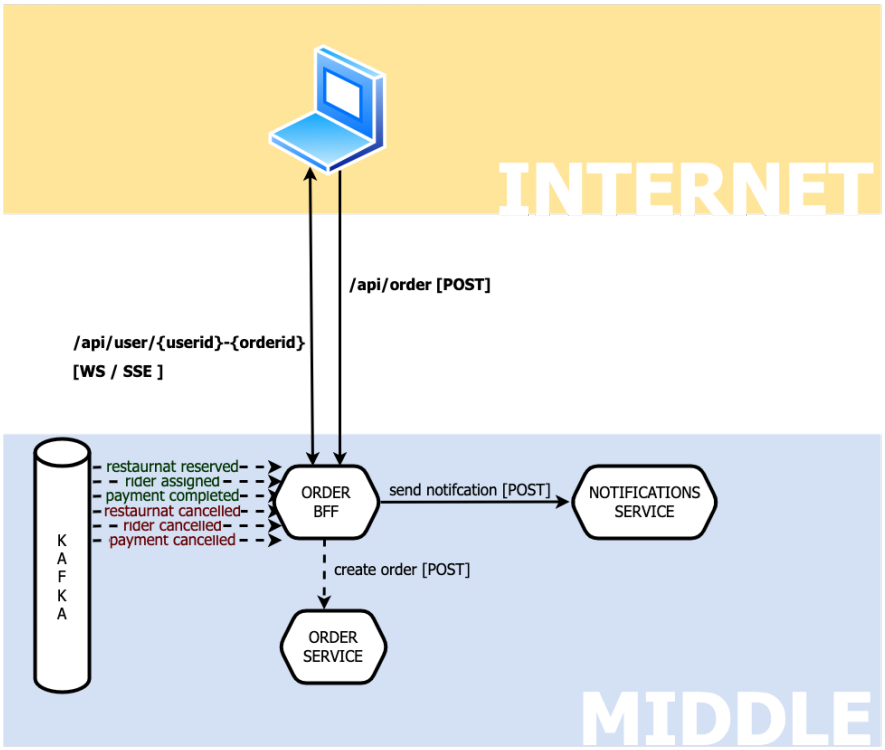


Ilustración 9 - Flujo frontend

- Una vez que se realiza el post a BFF con los datos del pedido se establece la conexión asíncrona entre front y BFF.
- Cada uno de los eventos que recibe BFF se informa al usuario correspondiente.
- BFF no requiere de base de datos, no necesita persistir nada.
- En la ilustración 9 podemos ver también el servicio de notificaciones, para que en caso de que el usuario haya desconectado se le informe por esta vía.

[14] https://github.com/MasterCloudApps-Projects/Choreographed-Saga-with-Events-and-Consumers/blob/main/front/Dockerfile - L22

5.2.3 Conexión asíncrona

Para la conexión asíncrona entre front y BFF se han probado 3 tecnologías diferentes: Web Sockets, Server Sent Events y Long Pooling. No se han encontrado grandes diferencias, pero estas son las que se podrían sacar las siguientes conclusiones:

- En los 3 casos se queda una conexión abierta.
- Pooling lo descartaría por tener un hilo del servidor corriendo permanentemente y porque cada X segundos se repite la petición pudiendo recibir la respuesta en el momento de regenerar la conexión, haciendo perder el tiempo al usuario. Además, genera más tráfico y no ofrece ningún beneficio con respecto a las otras dos alternativas.
- Server Sent Events es más sencillo de implementar y cumple con el estándar http.
- Web Sockets permite bidireccionalidad por el mismo canal y datos más complejos.

Para el caso de uso que tenemos Server Sent Events parece la opción óptima, pero si se necesitarán enviar datos más complejos o bidireccionalidad en la comunicación, seguramente, se cambiaría a Web Sockets.

El proyecto se ha dejado configurable para cambiar el tipo de conexión, cambiando la propiedad **FRONT_CONNECTION_TYPE** en el fichero **.env**[15], poniendo el valor **WS** o **SSE** y haciendo docker build de nuevo.

[15] https://github.com/MasterCloudApps-Projects/Choreographed-Saga-with-Events-and-Consumers/blob/main/front/.env

5.3 Arquitectura final en kubernetes

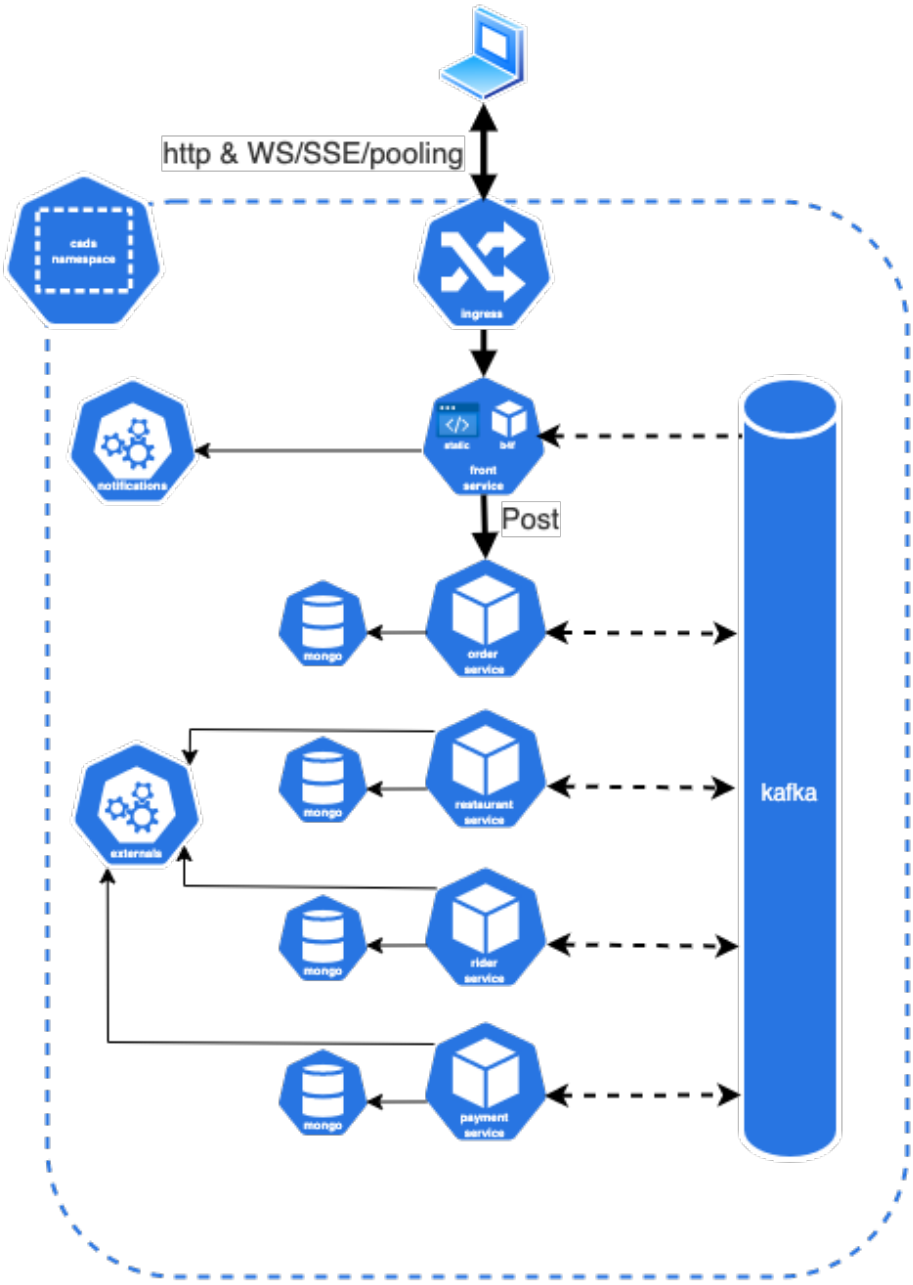


Ilustración 10 - Arquitectura en kubernetes

Además de los servicios y el front que ya se han descrito, se han implementado también el ingress y un api externals, que emula los restaurantes, riders y bancos.

El api externals dispone de un endpoint especial /api/config que permite que configuremos tanto la respuesta como el delay para poder hacer pruebas de todos los posibles casos.

También están incluidos kowl[16] y Kafka-ui[17], para facilitar la depuración, ya que permite visualizar y generar eventos desde una interfaz web. No son necesarios para el proyecto, pero ayudan para realizar pruebas y comprobar que todo funciona correctamente.

[16] https://github.com/redpanda-data/kowl
[17] https://github.com/provectus/kafka-ui

6 Testing e2e

Para comprobar que todo funciona correctamente y todos los casos de uso que se han implementado, se realizan test e2e con cypress y gherkin[18].

Todos los test lo primero que hacen es configurar el api de externals para el caso que queramos ejecutar. Por ejemplo, configuramos externals Rider para que responda con un 404 y comprobamos que al front llega la actualización y que a la orden en el restaurante se le ha realizado el rollback.

El api de notifications también se ha implementado con propósito de testing: si recibe una petición de enviar una notificación, guarda en memoria la notificación que enviaría. Se implementa a su vez el endpoint /api/notifications/:id que devolverá la notificación en caso de existir. De esta manera los tests pueden comprobar si se ha enviado o no una notificación al usuario para una compra.

Por último, los tests e2e comprueban los estados del pedido en las tres bases de datos, para asegurarnos que el rollback se ha realizado de forma correcta.

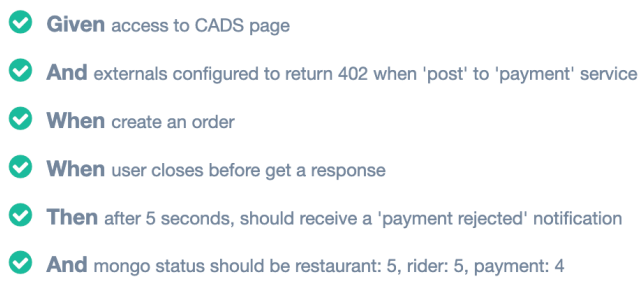


Ilustración 11 - Ejemplo e2e gherkin

En la ilustración 11 podemos ver el gherkin de uno de los escenarios que se ejecutan.

- Se abre la página.
- Se configura el api externals para que devuelva un 402 en el proceso de pago.
- Se crea la orden a través del front.
- Se cierra la página antes de obtener la respuesta.
- Se espera 5 segundos y se comprueba que el servicio de notificaciones tiene la notificación para nuestro pedido en el estado correcto.
- Se comprueba en las 3 bases de datos los estados (véase el Anexo de estados).

Se han generado dos reportes, uno realizado con mokawesome con screenshots y vídeos[19], y otro realizado con el reporte estandar de cucumber[20]. Este último aporta con respecto al anterior que en el informe podemos ver los gherkin ejecutados y en caso de fallo en que punto falló.

Además, se ha probado a borrar pods durante la ejecución de los tests e2e, comprobando que una vez vuelve a levantarse el servicio en cuestión, todo continúa como se esperaba y los tests e2e siguen saliendo en verde. Para estas pruebas hay que aumentar los timeouts de los tests que hay actualmente en el repositorio, o si no daría como fallido el test.

[18] https://raw.githubusercontent.com/MasterCloudApps-Projects/Choreographed-Saga-with-Events-and-Consumers/main/docs/e2e.mp4
[19] http://tfm.sanguino.io/mochawesome/
[20] http://tfm.sanguino.io/cucumber/

7 Conclusiones y trabajos futuros

7.1 Conclusiones

Se han conseguido todos los objetivos que se han planteado:

- Se ha desarrollado una saga coreografiada con eventos en kafka, haciendo los servicios idempotentes, resilientes e independientes.
- El frontend consume actualizaciones de la saga en tiempo real sin que los servicios tengan que hacer nada para informar a sus consumidores, y a la vez se informa al usuario perfectamente, estando en línea o no.

Aparte, hay varios puntos interesantes como conclusiones destacables:

- A pesar de la complejidad que tiene realizar una transacción con un sistema de microservicios, se ha realizado un sistema que a priori es bastante simple y no añade mucha complejidad a los propios servicios, no dificulta la escalabilidad, resiliencia ni el mantenimiento. Todo esto a su vez simplifica el gobierno de los equipos de desarrollo que desarrollarían los servicios. El uso de coreografía usando eventos para comunicar servicios idempotentes es lo que permite esta simplicidad.

- El uso de BFF es fundamental para este desarrollo. Con un servicio tan sencillo nos estamos quitando una responsabilidad muy grande del middleware y a la vez desacoplándolo y facilitando el frontend. El middleware no hace nada extra para que el front pueda consumir los eventos. Además, agiliza y simplifica el propio frontend al ser el mismo equipo el que desarrolla ambas piezas.

- Aunque a priori se creía que iba a haber mucha diferencia ente los sistemas de conexión asíncronas entre front y middleware, después de probar los tres planteamientos, se ha visto que apenas hay diferencias entre ellos. En el siguiente punto, *6.4 trabajos futuros*, se planteará que además de medir los tiempos, se podrían crear unos tests de performance que midan en el BFF los posibles bloqueos y consumos para decidir mejor.

En general ha sido un trabajo muy enriquecedor, pudiendo practicar muchas de las partes que vimos en el máster, no todas reflejadas en esta memoria, pero sí presentes.

7.2 Trabajos futuros

Hay varias tareas que se han quedado en el tintero por no poder abordarlas. A futuro me gustaría poder hacer las siguientes mejoras y evoluciones:

Refactor: El trabajo ha sido muy de investigación, en modo prueba de concepto, sin tests unitarios, cambio de ideas y probando muchas alternativas. Hay mucho código que llegados al punto en el que estamos no se habría escrito y organizado como está. Algún refactor ya se ha realizado, pero harían falta más iteraciones para llegar a un código limpio, escalable y mantenible.

Performance tests de la conexión asíncrona: se ha visto que en tiempos de front – BFF no hay diferencias significativas, pero ¿qué ocurre con el BFF? La idea sería hacer el mismo análisis, pero desde el punto de vista del BFF, analizando en los 3 casos de conexión asíncrona:

- En qué casos se bloquean hilos y en cuáles no.
- Cuál consume más recursos.
- Cuántas conexiones en paralelo puede soportar.

Para ello se propondrían unos tests con artillery o gatling que sean capaces de analizar estos puntos conflictivos y poder determinar desde el lado BFF que conexión es más óptima.

Escalabilidad: En principio está bastante preparado para ser escalable. Solo tendríamos que poner más particiones a los tópicos y configurar el escalado en kubernetes. Alguna prueba he realizado, pero me gustaría profundizar más.

Testing: solo se han implementado los tests e2e que aseguran el funcionamiento de todo. Lo primero que haría sería añadir tests unitarios junto al refactor. También se plantean hacer tests e2e de los que se han hecho manualmente en los que se pare un servicio durante una transacción para comprobar que, aunque se tarde más, todo sigue funcionando. Hasta ahora esto se ha probado a mano. Para comprobar la escalabilidad de la que hablábamos en el punto anterior, se deberían hacer los tests pertinentes también. Aparte se planearían añadir tests de contrato, tests de performance, monkey testing, etc…

CI/CD: siempre se pensó en montar un mono-repo o varios repos con github actions, pero no se ha llegado a realizar, no es muy costoso y aportaría mucho valor.

Observabilidad: Siempre estuvo en el tintero la observabilidad pero se quedó fuera por falta de tiempo. Se estudiaron varios sistemas y el que a futuro se puede implementar es APM de elastic [21]. Como principal ventaja está el hecho de que una forma sencilla te permite conectar las distintas capas, front, middle y bases de datos, para poder obtener un dashboard donde visualizar todo desde el punto de vista del cliente hasta las querys a base de datos. Además de obtener todas las métricas y alertas de elastic habituales.

Seguridad: Como userId se está usando un uuid generado y guardado en local storage, me gustaría aplicar todo lo aprendido en el máster relacionado con autenticación, validación, etc.

Conciliaciones: Deberíamos crear un sistema para el consumo de datos, conciliaciones que comprueben que los estados de las bases de datos son coherentes. Es posible que este sistema se pueda crear con extractores, ya que aquí no necesitamos inmediatez.

Escalabilidad BFF: aunque es un servicio muy simple y que no consume muchos recursos, me gustaría plantear la necesidad de escalarlo. Sobre todo, por estudiar como escalar un servicio en el que a cualquiera de las instancias le puede llegar un evento de actualización de cualquier cliente y a su vez esa instancia no sea la que tenga la conexión persistente con el usuario. Una posibilidad es hacer que todas las instancias reciban todos los eventos, y solo aquella que tenga la conexión con el cliente hará algo con el evento. Pero para el caso en el que el cliente se haya desconectado tendríamos que persistir clientes que han desconectado o los que están conectados en ese momento. En realidad, es un dato que solo interesa almacenar por minutos, por lo que en vez de utilizar una base de datos pensaría en algo más tipo Hazelcast[22], que comparta clientes conectados/desconectados.

[21] https://www.elastic.co/observability/application-performance-monitoring
[22] https://hazelcast.com/clients/node-js/

Frontend – UX: se hubiese querido añadir en el front un componente reactivo, que si se tarda más de unos segundos en recibir respuesta le mostrase mensajes al cliente disculpando la tardanza y ofreciéndole la opción de desconectarse y recibir la respuesta vía notificación. Algo tipo: “vaya, estamos tardando un poco en confirmar tu pedido, si lo deseas puedes irte y te mandaremos una notificación/email/sms, saludos!”

8 Bibliografía

- Apache Kafka:

  https://kafka.apache.org/24/documentation.html

- Saga Pattern in Microservices, Baeldung:

  https://www.baeldung.com/cs/saga-pattern-microservices

- A Brief Intro to KafkaJS:

  https://kafka.js.org/docs/introduction

- Kafka Acks Explained, Stanislav Kozlovski, Better Programming:

  https://betterprogramming.pub/kafka-acks-explained-c0515b3b707e

- MongoDB Kafka Connector:

  https://www.mongodb.com/docs/kafka-connector/current/

- Mongoose v6.3.3:

  https://mongoosejs.com/docs/guide.html

- Kubernetes Documentation:

  https://kubernetes.io/docs/home/

9 Anexos

9.1 Estados de una petición

Para marcar y consultar el estado de una transacción o de parte de una de las transacciones dentro de las bases de datos de cada servicio, se han definido los siguiente estados: Creado = 0, Actualizado = 1, Completado = 2, Rechazado = 4, Cancelado = 5

9.2 Tópicos

Sirva como referencia los siguientes contratos de cada uno de los posibles eventos generados durante el transcurso de la transacción:

order_created

```
{
 "orderId": "627767df9d5cbd88d1b5d631",
 "userId": "l2wxpcu6e1ea1hqhyzk",
 "cart": [{ "itemId": "p01", "amount": 3 }, { "itemId": "p02", "amount": 2 }, … ],
 "status": 0
}
```

restaurant_reserved

```
{
 "orderId": "627767e19d5cbd88d1b5d636",
 "cart": [ {…}, {…}, … ],
 "price": 16.84,
 "restaurantPhone": "+34 91 801 32 09",
 "restaurantAddress": "226 Hills Brook Apt. 844",
 "status": 0
}
```

restaurant_cancelled

```
{
 "orderId": "627767df9d5cbd88d1b5d631",
 "restaurantCancellationReason": "restaurant busy",
 "status": 4
}
```

rider_assigned

```
{
 "orderId": "627767e19d5cbd88d1b5d636",
 "riderId": "40599w1l2wxpjid",
 "riderPhone": "+34 625 91 22 32",
 "riderName": "Jon Moen",
 "status": 0
}
```

rider_cancelled

```
{
 "orderId": "627768a99d5cbd88d1b5d65c",
 "riderCancellationReason": "no rider available",
 "status": 4
}
```

payment_done

```
{
 "orderId": "6277689c9d5cbd88d1b5d653",
 "receiptId": "40599w1l2wxtl9l",
 "status": 0
}
```

payment_cancelled

```
{
 "orderId": "627767e19d5cbd88d1b5d636",
 "paymentCancellationReason": "generic error",
 "status": 4
}
```

Todos llevan la misma cabecera con el correlation id para poder trazar a qué pedido corresponden:

```
{
 "correlation-id": "{userId}-{orderId}"
}
```