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

Curso académico 2019/2020

Trabajo de Fin de Máster

Saga orquestada con eventos y consumidores

Autor: Miguel García Sanguino
Tutor: Micael Gallego Carrillo

Universidad Rey Juan Carlos

Table of Contents

Objetivos

El presente trabajo de fin de máster tiene como objetivo profundizar en las herramientas y técnicas vistas en el máster. En concreto se plantean dos objetivos principales.

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

En definitiva, se quiere crear una transacción con microservicios completa, punto a punto.

Cuando nos plateamos una transacción en la que intervienen más de un microservicio tenemos un problema principal: en qué momento podemos fijar en base de datos la transacción y en caso de tener que realizar una operación de compensación, tener la seguridad de que se compensan las operaciones que ya se hubiesen realizado.

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.

Nos fijamos además un objetivo 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 el 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 master están entre los objetivos implícitamente.

Introducción

Una de las herramientas que vimos en durante el máster era el uso de eventos para comunicar los servicios y el uso del patrón saga para controlar las transacciones. Uno de los patrones que vimos fue una saga orquestada y con máquina de estado. Es cierto que globalmente la hace más sencilla y controlable, pero acopla mucho los servicios unos con otros.

Tal y como hemos visto, los microservicios empiezan a tener sentido cuando estamos en proyectos de tamaño considerable, en el que tenemos un numero de equipos importante en el que el gobierno entre ellos requiere de cierta independencia para poder ser ágiles. De ahí que piense que la orquestación pueda traernos una dificulta de gobierno extra, aunque globalmente sea más sencilla.

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 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 servicios en sí. Reduciendo 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 estén fuertemente acoplados a la transacción en si, por lo que 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 – middle la tiene el mismo equipo.

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 middle, permitimos que el front reciba los datos como le vienen mejor, y a su vez no modificamos los contratos de los servicios. Esto nos ayudará a cumplir el objetivo de independencia entre capas y 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 extar el BFF en medio, el test de contrato sería entre servicios, que es más cómodo de montar que con un front como consumidor.

Caso de uso

Como caso de uso se va realizar 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.

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 de que ha ocurrido en cada paso de la saga, y como decíamos será el servicio BFF el responsable de la comunicación con el front.

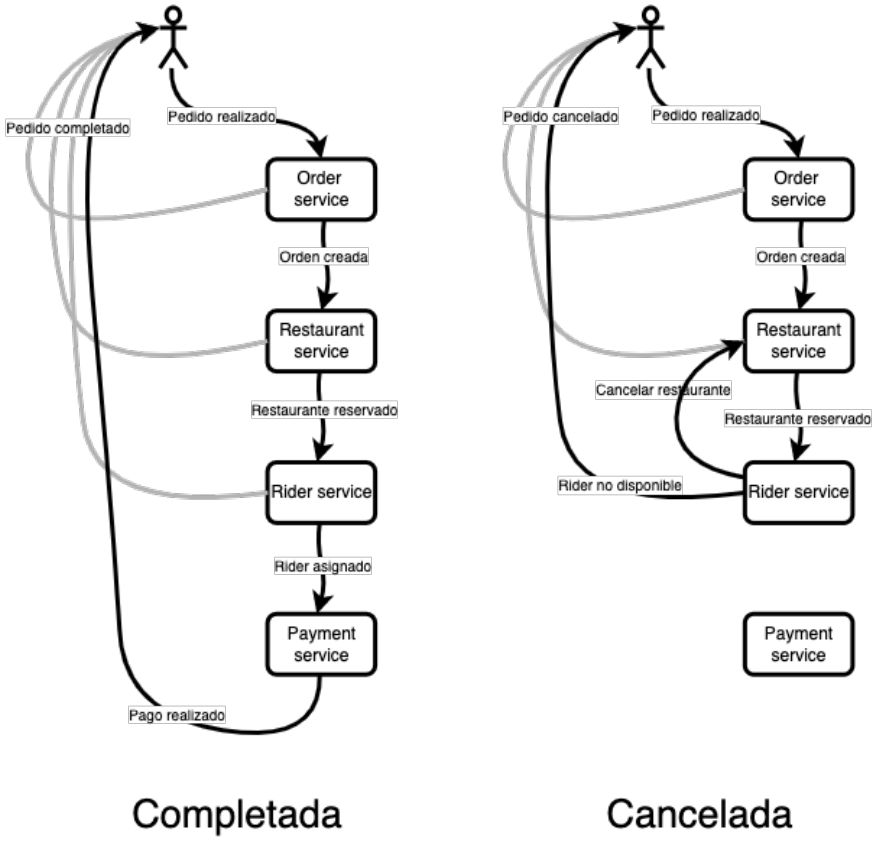

Ilustración 1 - Flujo de transacción completa y cancelada

En la ilustración 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.

Stack tecnológico

Middleware

Ilustración 2 - Stack middleware

Kubernetes: Lo hemos visto mucho en el master y es ya un estándar cuando hablamos de microservicios.

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 orquestación y en el escalado de los servicios.

Node: Aunque el master hemos visto Node y Java, se ha utilizado mucho más Java, por lo que se va a implementar los servicios con node. Aun así, se pretendía migrar alguno de los servicios a java, para comprobar que se puede cambiar perfectamente de tecnología sin que el resto de los servicios tengan si quiera que saberlo.

MongoDB: cada servicio tendrá los datos de su parte de la transacción, únicamente relacionables por el id de la transacción. Al internamente 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 master que bases de datos sql.

Kafka.js: se han probado 3 clientes de kafka[1] kafka-node[2] y kafka.js[3]. De los tres el ultimo parece el más estable, el que está más mantenido y el que ofrece más opciones para la tarea que vamos a realizar.

Express: no se pretender innovar en todo, lo hemos usado mucho en el master y es compatible con api rest, sse y websockets. No se han buscado alternativas.

Mongoose: al igual que express, se ha utilizado en el máster, funciona bien, no se han buscado alternativas.

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

Frontend

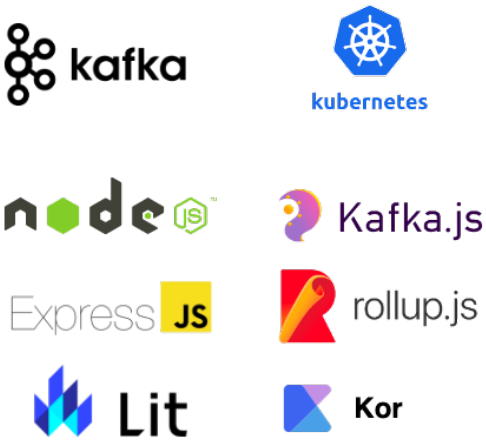

Ilustración 3 - Stack frontend

Kubernetes y Kafka: El servicio BFF lo hemos tratado en todo el TFM como si fuera parte del front, por tanto el stack de front es en parte compartido con el de middleware.

Node: En el caso del BFF vamos a usar nodejs seguro, este servicio lo desarrolla y mantiene el mismo equipo que el frontend, por lo que que en ambos se use javascript facilitará mucho las cosas.

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[4], webpack[5] y esbuild[6], 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 Rect, Vue o Angular.

Kor: el frontend en si no es objetivo del presente TFM, por lo que tenía sentido usar un catálogo de componentes que ya tienen estilos. Entre todos los catalogos [7]que hay para Lit, Kor nos ayudará a crear un layout fácilmente y sin complicaciones.

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

Arquitectura

Middleware

Ya hemos definido bastante los requisitos para nuestros servicios, pero no como vamos a desarrollarlos. Tenemos claro que los servicios deben ser coreografiados, que no queremos guardar una máquina de estados, y queremos que sean escalables, resilientes e independientes.

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

Resiliencia

Cuando un evento es consumido lo primero que realizamos es comprobar si con ese orderId ya se encuentra en base de datos.

Si no existe, 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 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[8] 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 es actualizándose envía a Kafka la actualización. De esta manera los servicios pueden preocuparse solo de definir los tópicos y la lógica de negocio, abstrayéndose de marcar los eventos como leídos.

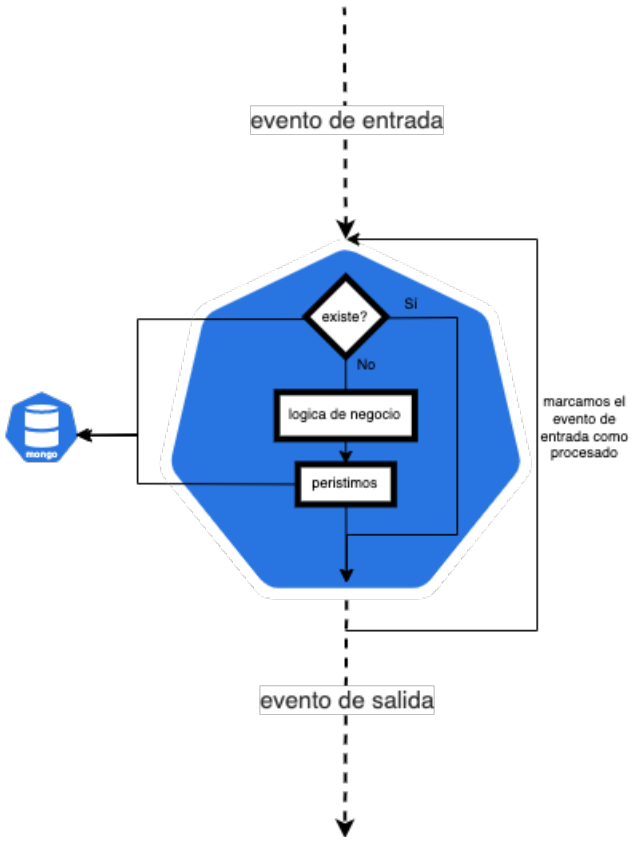

Ilustración 4 - Flujo idempotencia

La idempotencia tiene más ventajas además de permitirnos el consumo de eventos con coreografía sin deber tener una máquina de estados ni tener que guardar que se ha procesado. Además de esto, nos ayuda a que los servicios sean más fácilmente resilientes, escalables e independientes. Resilientes porque da igual cuando caiga que podemos tumbar el servicio, levantar otro y no tendremos que preocuparnos por cómo se ha caído ni como tiene que levantarse. Escalables porque, aunque por escalar acabase consumiéndose varias veces el mismo evento, no pasaría nada. O incluso si se desescala. Independientes, realmente cada servicio tiene su responsabilidad y puede abstraerse totalmente del anterior y de los siguientes.

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

Veamos como cumplimos la resiliencia. Un servicio puede caerse en medio de una transacción en varios momentos:

- 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, podemos continuar desde este punto
- 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 continua donde se quedó o se repite la operación, pero al ser todos idempotentes no es un problema. Como vemos el servicio es bastante simple y cumple con todos los requisitos que nos habíamos propuesto.

Construción

Todos los servicios utilizan docker multistage para construirse, y están pensados para configurarse e instalar dependencias lo primero y por último copiar el src. Esto ayuda mucho al proceso de desarrollo para evitar tener que instalar las dependencias cada vez que editamos el src.

Para instalar las dependencias se copia el fichero **.npmrc** dentro de stage de build, se instalan las dependencias y se borra el **.npmrc**. Al usarse y eliminarse en el stage de build evitamos que nuestras credenciales queden en algunas de las capas de docker.

```
FROM node:16.14.2-alpine3.15 AS builder
...
COPY package*.json /usr/src/app/
COPY .npmrc /usr/src/app/
RUN npm ci --only=production
RUN rm -rf .npmrc
...

FROM node:16.14.2-alpine3.15
WORKDIR /usr/src/app
COPY --from=builder /usr/src/app/ /usr/src/app/
...
```

De esta manera, al copiar el código y las dependencias al stage final, evitamos que acaben nuestras credenciales en la imagen de docker.

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.

A priori no 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[9] pero finalmente fue descartado y se optó por dejar al servicio que realizase esa lógica.

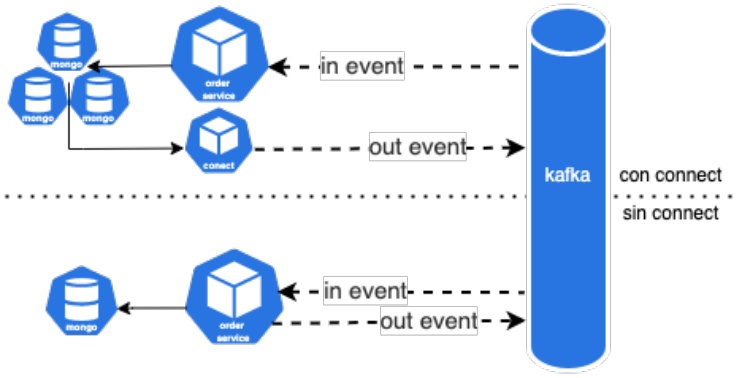

Ilustración 5 - Comparativa con o sin extractor de kafka

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 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 mas ineficiente el uso de los recursos, no solo por el número de elementos si no por el consumo de recursos.

También los pipelines que definen que extraemos y de qué modo lo hace, nos hace pensar que deberíamos tener varios corriendo a la vez, uno por cada topic y por cada estado. Aunque se consiguiera que fueran en un solo pod por servicio, no deja de ser un hilo por topic. Aunque según se investigó, es posible que el conector de debezium[10] simplificara varios de estos puntos, la realidad es que apenas supone esfuerzo para el servicio ni le aumenta complejidad ser el servicio el que se encarga de publicar los eventos.

Por último, estamos sacando parte de la responsabilidad del servicio fuera, algo que no termina de convencer en una arquitectura como la planteada.

[9] https://github.com/MasterCloudApps-Projects/Orchestrated-Saga-with-Events-and-Consumers/tree/main/kafkaConnectMongoDb
[10] https://debezium.io/documentation/reference/stable/connectors/mongodb.html

Base-service

Para reutilizar y abstraer la lógica del servicio de la lógica de las comunicaciones y la resiliencia, se ha creado base-service[11]. Se ha creado como depencia 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 si, 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 topic. El offset no deja de ser un puntero al siguiente evento que debe ser leido, por lo que, si llegan dos eventos, el evento 5 tarda 1 segundo en ser procesado, y el evento 6 tarda 300ms, si se sube el offset a 6 antes de acabar el evento 5, si en esos 700ms que hay antes de que se procese se cae el servicio, el evento 5 no habría sido totalmente procesado. Lo que se hace es gestionar el offset[12], guardando en en memoria los offset actuales y asegura que offset hay que setear en cada momento. En el ejemplo el offet estaría en 5, se guardaría el evento 6 como procesado, y al procesar el 5 se subiría el offset a 7.

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

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

Flujo middleware

Con todo lo que hemos visto hasta ahora el flujo en la capa middleware quedaría como vemos en la ilustración. Tan solo tenemos un punto de entrada desde fuera de nuestro flujo, una petición al servicio de Order. (Nótese que en la ilustración 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 puden 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.

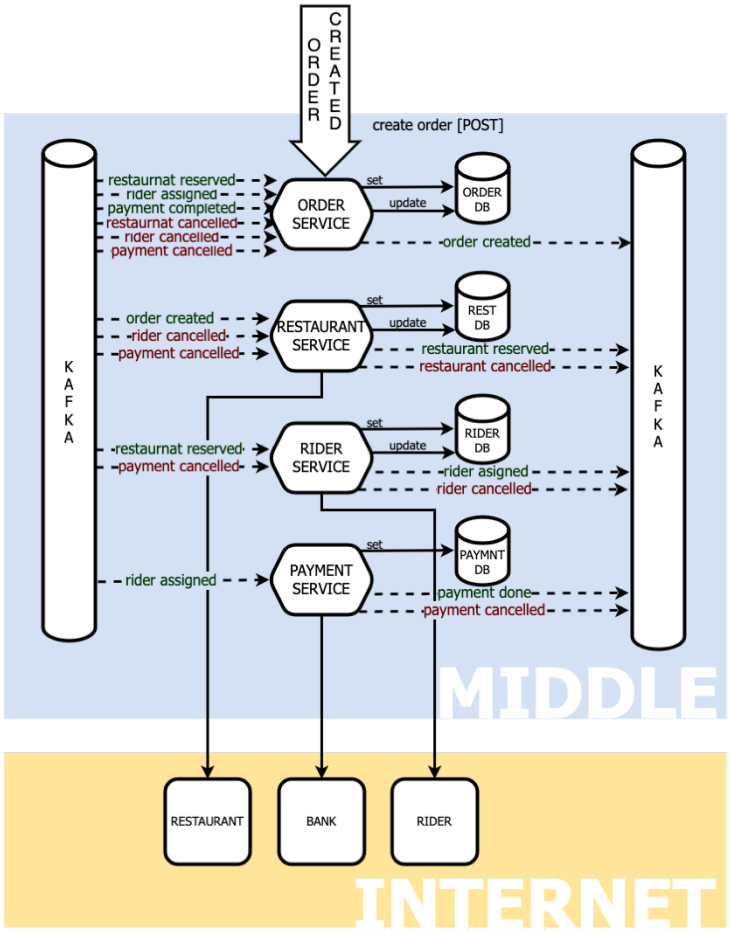

Ilustración 6 - Flujo middleware

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.

Como resumen, nuestro middleware estará desacoplado de sus consumidores y al mismo tiempo los servicios estén desacoplados entre ellos. Con ello conseguiremos nuestro objetivo en el que 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 dos razones.

La primera es quien 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, por ejemplo, que la saga cambie y se reserve primero el Rider, los cambios son importantes. Order ayuda a abstraer el resto de la saga del exterior, ya que el resto de 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 no tener una opción mejor clara y por saber que en un escenario más real tendrá más responsabilidades, se decide que el servicio de Order es necesario.

Frontend

Estaticos y BFF

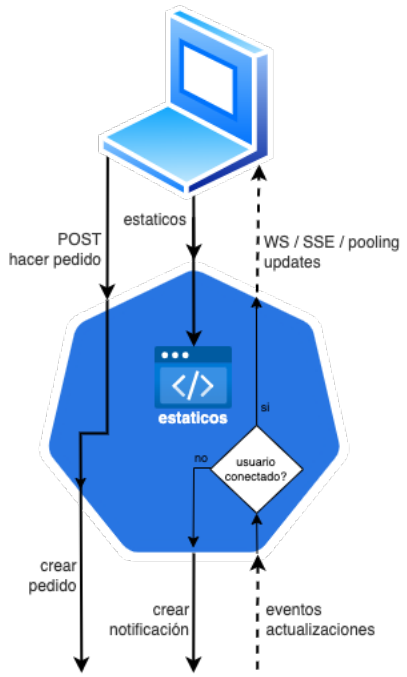

Ilustración 7 - Contenedor Frontend

Como hemos comentado, 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.

El frontend persiste una conexión con el BFF 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 desconecta, porque se queda sin internet, o porque 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, pero se lanza la build de rollup en el stage de build y se copian los assets generados en la imagen final.

```
FROM node:16.14.2-alpine3.15  AS builder
...
WORKDIR /usr/src/web
RUN npm ci
RUN rm -rf .npmrc
RUN npm run build

FROM node:16.14.2-alpine3.15
...
COPY --from=builder /usr/src/web/public /usr/src/app/public
...
```

Flujo Frontend

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

- 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 el diagrama podemos ver también el servicio de notificaciones, para que en caso de que el usuario haya desconectado se le inform por esta vía.

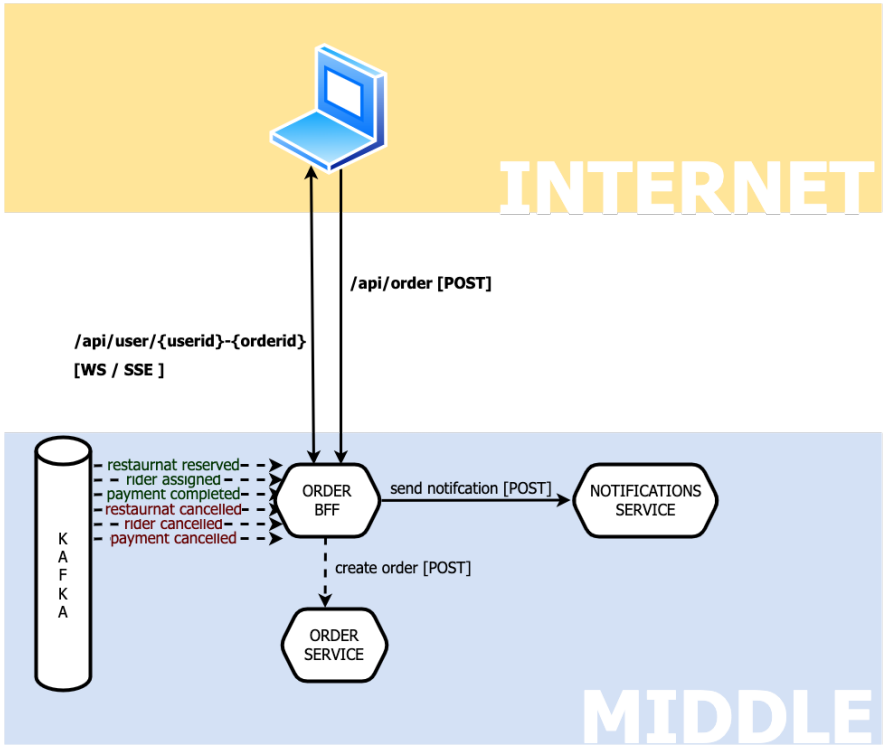

Ilustración 8 - Flujo frontend

Conexión asíncrona

Para la conexión asíncrona entre front y BFF se han probado 3 tecnologías diferentes, websockets, server sent events y long pooling. No se han encontrado grandes diferencias, pero estas son las que podría destacar:

- En los 3 casos se queda una conexión abierta.

- pooling lo descartaría por dejar 1 hilo siempre pillado y porque cada X segundos se repite la petición, genera más tráfico y no ofrece ningún beneficio.

- server sent events es más sencillo de implementar.

- web sockets permite bidireccionalidad.

Para el caso de uso que tenemos me quedaría con Server Sent Events, pero si se necesitarán enviar datos más complejos o bidireccionalidad, cambiaria 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**[13], poniendo el valor **WS** o **SSE** y haciendo docker build de nuevo.

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

Arquitectura final en kubernetes

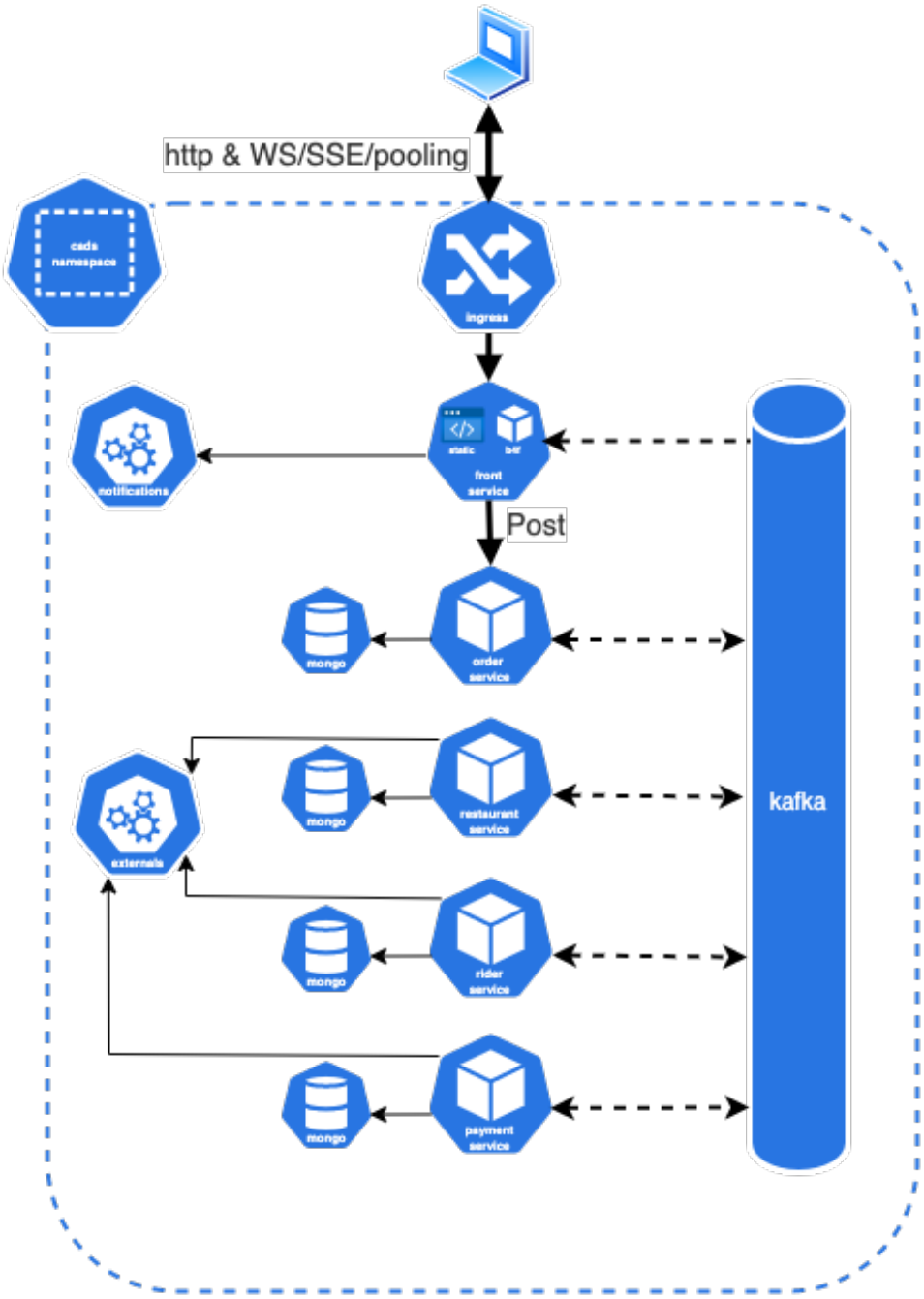

Ilustración 9 - Arquitectura en kubernetes

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 externas 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[14] y Kafka-ui[15], para facilitar la depuración. 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.

[14] https://github.com/redpanda-data/kowl
[15] https://github.com/provectus/kafka-ui

Testing e2e

Para comprobar que todo funciona correctamente y todos los casos de usos se ha implementado test e2e con cypress y gherkin.

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 test pueden comprobar si se ha enviado o no una notificación al usuario para una compra.

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

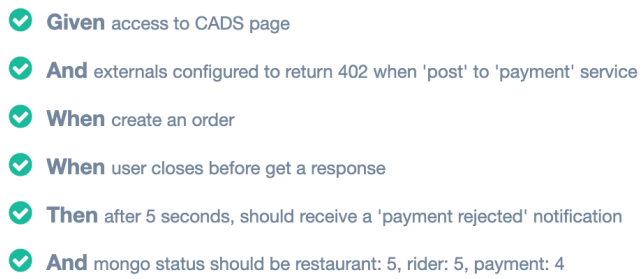

Ilustración 10 - Ejemplo e2e gherkin

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

- Se abre la pagina,
- 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 (5 es rollback realizado, 4 es rechazado, cuando el order no ha llegado a ese servicio se marca como -1)

Se han generado dos reportes para los los test E2E, uno realizado con mokawesome con screenshots y videos[16]. Y otro realizado con cucumber[17], 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 manualmente pods durante la ejecución de los test e2e, comprobando que una vez vuelve a levantarse el servicio en cuestión, todo continua como se esperaba y los test e2e siguen saliendo en verde. Para estas pruebas hay que aumentar los timeouts de los test que hay actualmente en el repositorio, o si no daría como fallido el test.

[16] http://tfm.sanguino.io/mochawesome/
[17] http://tfm.sanguino.io/cucumber/

Conclusiones y trabajos futuros

Conclusiones

Se ha conseguido todos los objetivos que se han planteado, aunque como veremos en trabajos futuros, se hubiese querido abordar muchas más cuestiones.

Hay 3 puntos interesantes como conclusiones que se quieren destacar:

- 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 y en no dificulta la escalabilidad, resiliencia ni el mantenimiento. El uso de coreografía usando eventos para comunicar servicios idempotentes es lo que permite esta simplicidad.

- El uso de BFF es fundamental en el trabajo. Un servicio tan sencillo nos está quitando una responsabilidad muy grande del middleware y a la vez aislándolo del frontend. El middleware no hace nada extra para que front pueda consumir los eventos. Además agiliza y simplifica el propio frontend al ser el mismo equipo el que desarrolla ambas piezas.

- Se planteaban varios sistemas de conexión asíncronas entre front y el middleware, pero se ha visto que apenas hay diferencias entre ellos, a nivel frontend, algo que a priori parecía fundamental se ha visto que no ha sido tanto. En trabajos futuros lo ampliamos, pero faltarían unos test de performance, analizando los posibles bloqueos y consumos del BFF 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 si presentes.

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 test 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 haría falta más iteraciones para llegar a un código limpio, escalable y mantenible.

Performance test de async connection: aunque 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.
- Cual consumen más recursos cpu/ram
- Cuantas conexiones en paralelo puede soportar

Además, unos buenos test con artillery o gatling que sean capaces de analizar estos pains pueden ser muy interesantes de realizar.

Escalabilidad: En principio está bastante preparado para ser escalable, solo tendríamos que poner más particiones a los topics y configurar el escalado en kubernetes. Alguna prueba he realizado, pero me gustaría profundizar más en un futuro. Para el sistema de marcado como leído de los eventos, es posible que tengamos que realizar algún cambio, ya que en caso de recalcularse los consumidores el offset ya no sería uno a uno, si no que tendría que ir de n en n.

Testing: solo se han implementado los test e2e que aseguran el funcionamiento de todo. Lo primero que haría seria añadir test unitarios junto al refactor. También se plantean hacer test e2e en los que se tumbe un servicio durante una transacción para comprobar que, aunque se tarde más, todo sigue funcionando. Hasta ahora esto se ha probado a mano. Para hacer check de la escalabilidad de la que hablábamos en el punto anterior, se deberían hacer los test pertinentes también. Aparte se planearían añadir test de contrato, test 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 cierra un poco el circulo.

Observabilidad: Siempre estuvo en el tintero la observabilidad pero se quedó fuera por no poder abordar tantos puntos. Se estudiaron varios sistemas y el que a futuro se puede implementar es APM de elastic [18]. Como principal ventaja está el hecho de que una forma sencilla te permite conectar las distintas capas, front y middle, y los distintos servicios, para poder obtener un dashboard donde visualizar desde el punto de vista del cliente. Ademas de obtener todas las métricas y alertas de elastic que ya conocemos

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

[18] https://www.elastic.co/observability/application-performance-monitoring

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 ser la que tiene la conexión persistente. Una posibilidad es hacer que todas las instancias reciban todos los eventos, 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. 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[19], que comparta clientes desconectados por unos minutos.

Frontend – UX: se hubiese querido añadir en el front un componente reactivo, que si se tarda más de unos segundos en recibir respuesta le diera mostrase textos 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!”

[19] https://hazelcast.com/clients/node-js/

Bibliografia

- Apache Kafka
- A Brief Intro to Kafka · KafkaJS
- Kafka Acks Explained | by Stanislav Kozlovski | Better Programming
- MongoDB Kafka Connector — MongoDB Kafka Connector
- Mongoose v6.3.3: (mongoosejs.com)
- Kubernetes Documentation | Kubernetes

Anexos

Topics

Posibles estados: Creado = 0, Actualizado = 1, Completado = 2, Rechazado = 4, Cancelado = 5

Todos los topic llevan como cabecera: { "correlation-id": "{userId}-{orderId}" }

order_created


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


restaurant_reserved


```
{
 "orderId": "627767e19d5cbd88d1b5d636",
 "cart": [{
    "itemId": "p01",
    "amount": 3
  }
 ],
 "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
}
```