Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA COMPUTADORES

Trabajo Fin de Grado

Arquitectura de Microservicios en Kubernetes

Autor: Sergio López Rico

Tutor/es: Salvador Otón Tortosa

2022

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Grado

ARQUITECTURA DE MICROSERVICIOS EN KUBERNETES

Sergio López Rico

07 / 2022

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Grado en Ingeniería de Computadores

Trabajo Fin de Carrera

Arquitectura de Microservicios en Kubernetes

Autor: **Sergio López rico**

Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: _____

Vocal 1º: _____

Vocal 2º: _____

Calificación: _____

Alcalá de Henares a de de 2022

Quiero agradecer a mis padres por mi primer ordenador, un Pentium III 700Mhz, con el que empecé a teclear mis primeras líneas de código. También, por su apoyo incondicional en todas mis decisiones vitales, lo cual, me ha llevado a descubrir mi pasión y que hoy en día me pueda dedicar a ella, la informática.

Índice detallado

Resumen

El desarrollo de aplicaciones online está en constante crecimiento. Cada vez hay un mayor número de usuarios, lo que se traduce en necesitar más recursos de hardware para atender sus peticiones. Además, con el tiempo las aplicaciones van teniendo más complejidad de software, cosa que complica los despliegues en producción y con el crecimiento de la aplicación crece el número de desarrolladores trabajando en ella, lo que dificulta la organización del proyecto.

Por eso surgen, entre otros, los siguientes problemas que trataré en este trabajo:

- Satisfacer la demanda en momentos de carga elevada.
- Uso de metodología que facilite el desarrollo de estas aplicaciones.

Palabras clave

Kubernetes, Docker, Microservicios, Saga Pattern, CICD.

Abstract

The development of online applications is constantly growing. As time goes by, there are more users, and therefore, we need more hardware resources in order to process their requests. Furthermore, the application's software complexity increases, so releasing them to production becomes harder. Finally, as more complex applications, the more developers we are going to need working on it, so, also, the project management gains complexity.

For that reason, I am going to discuss about the surging problems during this project:

- Satisfy demand in heavy load times.
- Methodologies that ease the development of these applications.

Keywords

Kubernetes, Docker, Microservicios, Saga Pattern, CICD.

1. Introducción y Objetivos

1.1. Introducción

El desarrollo de aplicaciones online está en constante crecimiento. Cada vez hay un mayor número de usuarios, lo que se traduce en necesitar más recursos de hardware para atender sus peticiones. Además, con el tiempo las aplicaciones van teniendo más complejidad de software, cosa que complica los despliegues en producción y con el crecimiento de la aplicación crece el número de desarrolladores trabajando en ella, lo que dificulta la organización del proyecto.

Por eso surgen, entre otros, los siguientes problemas que trataré en este trabajo:

- Satisfacer la demanda en momentos de carga elevada.
- Uso de metodología que facilite el desarrollo de estas aplicaciones.

1.2. Objetivos

1.2.1. Objetivo general

Construir una arquitectura de microservicios escalables en la nube, que de una manera óptima pueda ajustarse a la demanda de los usuarios de una manera automática y libre de fallos.

1.2.2. Objetivos específicos

Para poder conseguir el objetivo propuesto hay que plantearse los siguientes objetivos:

- ❏ Segmentar nuestra aplicación en microservicios.
- ❏ Guiar el desarrollo de las aplicaciones con tests (TDD, Test Driven Development).
- ❏ Utilizar la metodología de desarrollo TDD (Test Driven Development) y CI/CD (Continuous Integration/Continuous Delivery).
- ❏ Servir las aplicaciones en contenedores de Docker.
- ❏ Configurar despliegues de aplicaciones en Kubernetes.

1.3 Metodología

En la actualidad existen muchas metodologías, tecnologías y herramientas para el desarrollo de aplicaciones en la nube. A lo largo del TFG, se explicarán las que van a ser utilizadas en el desarrollo de este proyecto, contando sus antecedentes y la razón por la que es utilizada para el desarrollo de aplicaciones basadas en microservicios. Van a ser las siguientes:

- Docker: Para la creación de una aplicación portable.
- Kubernetes: Para añadir escalabilidad a nuestra aplicación.
- Github Actions: Herramienta para aplicar técnicas de CI/CD
- Testing: Para la comprobación del correcto funcionamiento de la aplicación en un entorno CI/CD

Tras la introducción se utilizará como objeto de estudio una aplicación en la que ejecutarán las fases de Planificación, Diseño, Implementación y Pruebas típicos del ciclo de vida de un software. La aplicación consistirá en un *ECommerce* cuyo propósito será recopilar pedidos de una tienda online y gestionar sus pagos y envíos.

2. Microservicios

2.1. ¿Qué es un Monolito?

Para empezar a hablar sobre microservicios primero hay que entender la arquitectura monolítica, que era la más común y que dio lugar a la aparición de la arquitectura de microservicios.

Se puede describir brevemente una arquitectura monolítica como aquella en la que todos los servicios están acoplados bajo una misma aplicación y todos sus componentes están en la misma base de código, necesitándose todas entre ellas para su correcto funcionamiento. Esta arquitectura tiene algunas ventajas:

- ❏ El desarrollo, ya que se pueden aplicar cambios globales en la lógica de negocio, lo cual puede ser muy habitual al principio, de manera sencilla, y cuando entren nuevas personas en el equipo de desarrollo podrán entender más fácilmente toda la aplicación.
- ❏ Los despliegues son más fáciles y rápidos de hacer ya que solo tenemos que desplegar un único contenedor o aplicación.
- ❏ El escalado también se simplifica ya que basta con arrancar copias de la misma aplicación tras un balanceador.

En resumidas cuentas podemos decir que un monolito, en primera instancia, es más sencillo de desarrollar, escalar y desplegar. Pero cuando crece la aplicación, estas ventajas desaparecen, y se convierten en aspectos negativos:

- ❏ El desarrollo se complica ya que cuanto más código fuente hay más intimidados se van a sentir los desarrolladores nuevos por la dificultad para entender la aplicación y llegar a ser productivos, además de dificultar encontrar las implicaciones que puede tener en otra parte de la aplicación incluso con el cambio más, aparentemente, sencillo.
- ❏ A medida que crece la aplicación va a consumir más recursos y podría saturar nuestro entorno de desarrollo tanto por la ejecución como por la propia herramienta por las tareas que realiza de análisis, indexaciones, etc.
- ❏ Los despliegues continuos se complican ya que los test y las compilaciones tardan más en ejecutarse. Además, se necesita, en la mayoría de los casos, lanzar test de regresión sobre toda la aplicación ya que al estar todos los componentes ligados no podemos asegurarnos de que un cambio afecte a otra parte no esperada.
- ❏ El escalamiento del equipo de desarrollo se ve reducido ya que cuando sea necesario separar a los equipos en cada área de la aplicación (Frontend, Servicio A, Servicio B….), éstos van a tener que coordinarse con el resto para que los cambios que hacen no interfieran de manera negativa, o peor, de manera inesperada a los demás equipos.

- ❏ El escalamiento de los servicios porque los distintos apartados funcionales de nuestra aplicación pueden requerir recursos distintos, imaginemos que tenemos una aplicación que tiene una parte con procesos intensivos en CPU y memoria, y otros en GPU para procesamiento de imágenes. Cada vez que escalemos creando una nueva instancia de la aplicación, ésta tendrá que tener estos recursos necesarios disponibles.
- ❏ Esta arquitectura no permite tener aislamiento de fallos ya que si un servicio consume, por ejemplo, toda la memoria, hará que se colapse todo el sistema en lugar de dejar inactivo solo al servicio responsable.

2.2. ¿Qué es un Microservicio?

El término microservicio hace referencia a un principio organizacional que consiste en construir una aplicación separando los componentes de negocio en distintos servicios autónomos que pueden trabajar, o no, juntos.

Aunque construir un monolito tiene sus desventajas a largo plazo, adoptar esta arquitectura al comienzo de un desarrollo no es una mala decisión ya que al principio es complicado saber en que microservicios se puede separar, y además, con los monolitos se puede pivotar con mayor facilidad. Eso sí, hay que estar preparado para identificar el momento de pasar a microservicios y llevarlo a cabo.

Enfocándonos en los aspectos negativos de un monolito podemos enumerar las ventajas principales de una arquitectura de microservicios.

- ❏ Cada equipo se puede encargar de un grupo de microservicios. Esto simplifica la integración y los despliegues continuos ya que no deberían afectar a otras partes de la aplicación, siempre que no se cambien los contratos de las APIs y los canales de comunicación. Además, el desarrollo se simplifica ya que, aunque sacrificando visión global, el repositorio de código fuente no crece de manera abusiva dificultando el entendimiento de cómo funciona un microservicio.
- ❏ Cada microservicio puede tener unos requisitos de hardware distintos, y con esta organización podemos escalar distintos tipos de máquinas según se necesite más CPU, GPU o memoria.
- ❏ El aislamiento de fallos en este caso sí está presente, ya que si un servicio se cae permite que el resto de servicios no dependientes puedan seguir operando con normalidad y generar un menor impacto en la disponibilidad de la aplicación.

Pero como cualquier solución, los microservicios también tienen sus desventajas. Encontrar la separación adecuada es un reto, de hecho, si no se hace bien, se podría acabar desarrollando un monolito distribuido. Además, los sistemas distribuidos son bastante más complejos de desarrollar, testear y desplegar. También se encuentran complicaciones a la hora de realizar transacciones ACID.

2.3. Características principales

2.3.1. Escalabilidad

Existen numerosas descripciones acerca de cómo debe ser un microservicio. Algunas de ellas hacen alusión a Domain Driven Design, otras a su tamaño por tener en su nombre la palabra “micro” y dicen que debería de poder ser desarrollado o refactorizado en dos semanas. Hay otra descripción mencionada en el libro libro *The Art of Scalability* (Addison-Wesley, 2015) describe un modelo de escalabilidad que cuadra bastante bien con esta arquitectura.

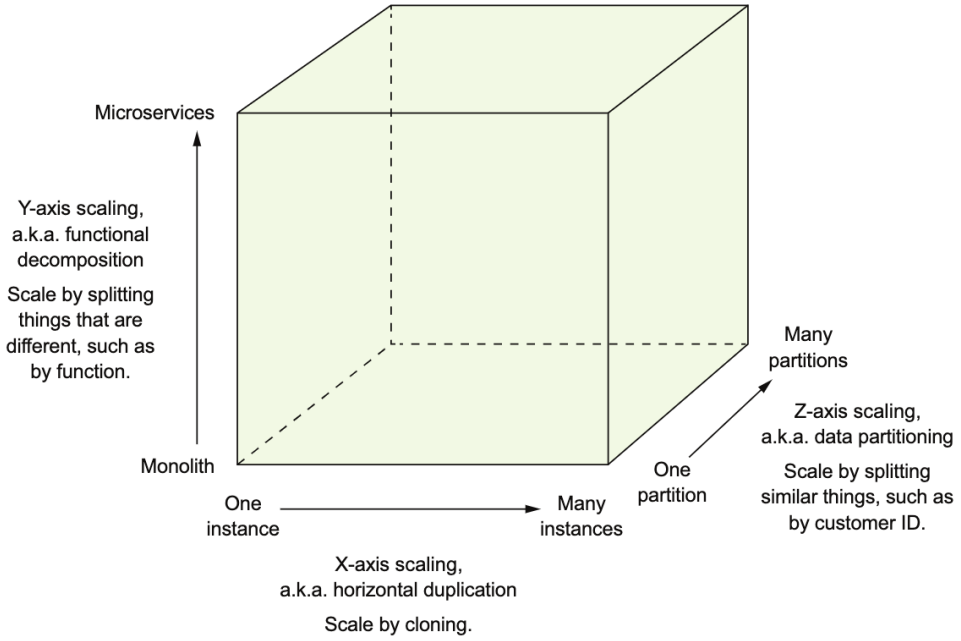

Figura 1. Escalabilidad en microservicios

Este modelo está formado por tres ejes X, Y, Z, que se describen a continuación:

- **Eje X:** Escalamiento horizontal de la aplicación, tras un balanceador de carga que reparte las peticiones entre N instancias idénticas del mismo servicio. Con este escalamiento ganamos capacidad y disponibilidad.
- **Eje Y:** Escalamiento por descomposición en diferentes microservicios. Dividiendo la funcionalidad, reduciendo la complejidad y alejándonos de una arquitectura monolítica.
- **Eje Z:** Escalamiento por partición de los datos en función de un discriminador, como puede ser el país, permitiendo a un router dirigir las peticiones a distintos grupos de microservicios. Este escalamiento es útil para distribuir las peticiones, pero para mejorar la escalabilidad funcional hace falta escalar en el eje y.

La definición de microservicio con este modelo no hace alusión al tamaño del microservicio, sino sobre cómo enfocar la responsabilidad de cada uno de ellos.

2.3.2. Modularidad

En el desarrollo de una aplicación grande la modularidad es un aspecto muy importante ya que es difícil que una persona o equipo conozca y entienda todos los aspectos técnicos y funcionales de ella. En cambio, separando la aplicación en distintos módulos según el área de negocio u otros aspectos funcionales podemos tener equipos o individuos trabajando en cada uno de ellos con un conocimiento casi absoluto de su contenido al poder abstraerse del resto de la aplicación.

En la arquitectura de microservicios se usa a un servicio como unidad de módulo. Además de la simplificación también mejora la encapsulación tras una API, ya que en un monolito puedes hacer un bypass funcional accediendo a clases o funciones internas, o incluso, acceder a la base de datos directamente y tener distintos puntos en el que se puede modificar el estado de la aplicación, lo cual dificulta entender el flujo del programa y puede ocasionar comportamientos inesperados.

2.3.3. Única base de datos

Los microservicios tienen que estar lo más desacoplados posibles y comunicarse únicamente mediante APIs. Si cada uno de ellos tiene su propia base de datos el aislamiento es mayor y el acoplamiento disminuye. Además mejora la disponibilidad del servicio ya que un lock por escritura en la base de datos de otro microservicio no afectará a los demás. Ésto, sin embargo, trae complicaciones de cara a hacer transacciones a nivel global y se explicará cómo solucionarlo más adelante.

2.3. ¿Cuándo pasar a Microservicios?

Viendo las ventajas y desventajas de cada arquitectura podemos ver que hay dos factores principales que perjudican a la arquitectura monolítica: el tamaño del equipo y de la aplicación. A medida que crezca una de las variables, o ambas, empezará a aparecer los síntomas que serán indicadores de que hay que iniciar esa migración:

El desarrollo es lento

Debido al crecimiento de la aplicación la aplicación va a tardar más en levantarse en el entorno de desarrollo, consumirá más recursos y la máquina sobre la que se trabaja se verá afectada. Además, crecerá la complejidad y el desarrollador tardará más en aplicar los cambios al tener que ser más meticuloso por la interconexión de todas las piezas.

El escalado se complica

Como se ha mencionado, los distintos componentes consumen distintos recursos. Por lo tanto, otro síntoma será que tengamos servidores levantados con recursos sin utilizar constantemente. Pongamos por ejemplo una aplicación en la que tenemos dos servicios:

- Servicio A: utilizado el 80% del tiempo y es intensivo en memoria.
- Servicio B: utilizado solo un 20% del tiempo, intensivo en GPU.

Con esta configuración en una arquitectura monolítica tendríamos que tener en todos nuestros servidores potencia tanto en Memoria como en GPU, y estaremos desperdiciando el recurso de la GPU por falta de uso un 80% del tiempo, problema que se resuelve si tenemos un escalado de servicios para un tipo de servicio y otro.

Despliegues lentos y/o poco fiables

A medida que la aplicación crece es más difícil hacer pruebas automatizadas y se harán muchas de ellas a mano, lo que afectará en el tiempo que se tarda en iniciar el despliegue de un cambio desde que se ha terminado su desarrollo.

Stack tecnológico anticuado

A medida que pasa el tiempo, además de crecer la aplicación, salen nuevas tecnologías y otras suben de versión. En una arquitectura monolítica, aplicar un cambio de versión puede resultar en romper el correcto funcionamiento, y dificulta añadir nuevas tecnologías por posibles problemas de compatibilidad.

2.3. Descomposición en microservicios

Un servicio es una pieza independiente de software que ofrece una funcionalidad. Cada servicio puede tener su propia arquitectura y stack tecnológico, aunque la típica sea hexagonal. Toda interacción con éste debe ser mediante una API que encapsule su implementación al completo, y no permita acceder a sus datos de forma directa. Para ello, esta API debe exponer las operaciones disponibles que pueden ser de dos tipos:

- Command (Comando): Es una operación que genera un cambio en la aplicación.
- Query (Consulta): Es una operación que sirve para extraer y consultar información.

Además de esas operaciones un servicio puede publicar eventos para notificar a los servicios que estén suscritos y poder reaccionar ante estas operaciones, normalmente mediante colas.

Una vez hemos descrito que es un microservicio toca abordar el cómo se crea esta arquitectura y como se separan los servicios.

2.3.1. Cuidado con las librerías compartidas

Con el fin de reutilizar código en las aplicaciones, los desarrolladores tienen que crear paquetes. Es una muy buena práctica, pero hay que tener cuidado porque a veces se tiende a poner en esas librerías lógica de negocio, que está sujeto a cambios, y si cambia algún requisito de negocio tendríamos que reconstruir y desplegar todos los servicios de nuevo. Este caso de uso de una librería es carne de cañón para crear un servicio.
En cambio, sí puede ser un paquete compartido si este contiene funcionalidades más genéricas, como por ejemplo una librería para operaciones matemáticas o para realizar validaciones, que no las reglas de validación en sí mismas.

2.3.2. ¿De qué tamaño es un microservicio?

El número de líneas de un servicio no es relevante, a pesar de que la arquitectura de microservicios contenga la palabra “micro”. La principal regla para saber que un servicio para saber si hemos diseñado bien un microservicio es que pueda ser mantenido por un único equipo. Si este servicio requiere más de un equipo, las baterías de pruebas requieren mucho tiempo si son un indicador de que este servicio tiene más tamaño o responsabilidad de la necesaria. Además, si un cambio en un microservicio provoca cambios en otros, o viceversa, significa que esos microservicios puedan estar acoplados en mayor o menor medida.

2.3.5. Definición de una arquitectura de microservicios

Una vez tenemos definidos los requisitos de negocio podemos empezar a definir el microservicio, esto se puede hacer en tres pasos. Pero no es un proceso puramente mecánico y puede necesitar varias iteraciones. Esos tres pasos son:

1. Identificar modelos y operaciones del sistema
2. Identificar los servicios
3. Definir las APIs de cada servicio y la relación entre ellos.

Identificar modelos y operaciones del sistema

Sin entrar en profundidad sobre cómo se van a ejecutar esas operaciones (HTTP, RPC, Broker) debemos identificar y definir de manera abstracta que queremos que nuestro sistema entero sea capaz de hacer extrayendo esas operaciones de los requisitos de negocio y los casos de uso definidos, preferiblemente, en Gherkin. Estas operaciones como ya hemos mencionado, se clasifican en comandos (commands) y consultas (queries) para, respectivamente, alterar u obtener el estado de nuestro sistema. En este punto no hay que centrarse en la implementación de esas operaciones ya que estamos definiendo nuestra arquitectura.

Para poder definir las operaciones hay que definir primero los modelos de nuestro dominio de alto nivel para tener un vocabulario sobre el que definir las operaciones. Cada servicio tendrá su dominio, y estos modelos serán más especializados.

Para extraer los modelos hay que buscar los sustantivos principales en nuestras historias de usuario, y los verbos para las operaciones.

Historia de usuario	**Modelos**	**Operaciones**
Given un **usuario registrado**	Usuario	Registro de usuario
And **añade** al **carrito productos**	Carrito, Producto	Añadir al carrito
When el usuario hace un **pago**	Pago	Pagar pedido
Then el **pedido** se crea	Pedido	Crear pedido

Una vez se identifican las operaciones hay que establecerlas definiendo una serie de puntos:

- **Operación**: nombre de la operación.
- **Retorno**: el valor devuelto por la operación.
- **Precondición**: situación que se tiene que dar antes de ejecutar la operación, se extrae de los “Given”.
- **Postcondición**: cambio que se provoca en nuestro sistema, se extrae de los “Then”.

Mostramos a continuación con el ejemplo de la operación “Crear pedido”

Operación	crearPedido()
Retorno	id_pedido
Precondición	● Hay un usuario registrado ● El usuario ha añadido productos al carrito ● El usuario ha pagado
Postcondición	Un pedido es creado

Identificar los servicios

Una vez se han identificado las operaciones podemos separar los servicios por áreas de negocio, para ello hay que saber a **qué** se dedica una empresa, y en base a eso definir los servicios, ya que es la parte más estable e invariante de una empresa a lo largo del tiempo, y este es un aspecto importante a la hora de definir la arquitectura de nuestro sistema.

También es importante saber el **cómo** hace una empresa sus negocios, pero al contrario que el **que**, la manera de hacer el negocio sí que puede cambiar con bastante frecuencia, y es lo que definirá la implementación de cada microservicio, y es precisamente una de las razones por las que adoptar esta arquitectura.

No es necesario complicarse demasiado al principio y se pueden dividir los servicios a grandes rasgos, ya que lo ideal es iterar a lo largo del tiempo ya que veremos como algunos servicios se vuelven ineficientes al tener que intercomunicarse entre sí constantemente, siendo esto una señal de que quizá tendrían que estar combinados en uno, o al contrario, que un servicio se vuelva muy complejo y nos está exigiendo que lo separemos en dos o más.

Para identificar los servicios está bien tener en mente estos dos principios de desarrollo de POO que están ligeramente ligados:

- **Single Responsibility Principle (SRP)**: Cada servicio debe tener una única responsabilidad, y por tanto una única razón de cambio, si un servicio tiene varias responsabilidades (o compartidas con otro servicio) supondrá que ese servicio va a estar sujeto a cambios con mayor frecuencia
- **Common Closure Principle (CCP)**: Cada paquete o servicio debe estar cerrado con todos los elementos que se verían afectados por el cambio de cualquier clase dentro de este paquete, evitando a toda costa que un cambio dentro de una clase afecte a otros paquetes.

Definir las APIs de cada servicio y la relación entre ellos.

Ahora toca asignar a cada servicio sus operaciones. Algunas de estas operaciones podrán ser autónomas y realizar los cambios necesarios sin llamar a operaciones de otros servicios. En cambio, hay otras operaciones que seguramente tengan que comunicarse con uno o más servicios.

Llegado a este punto, hemos definido nuestros servicios y sus apis de una forma muy abstracta, pero también hay que definir el protocolo de comunicación que se va a utilizar, y si es síncrono, que reduce disponibilidad o asíncrono, que trae otras complicaciones.

En la arquitectura de microservicios hay que considerar, además de estos servicios con lógica de negocio, otros servicios que servirán para agilizar las consultas ya que, si no, nuestros frontales tendrían que estar haciendo peticiones a varios sistemas, con sus métodos de autenticación, etc. Para solucionar esto se pueden crear API gateways, que son unos servicios que expondrán operaciones globales en nuestro sistema y que internamente se encargará de recopilar esta información de cada microservicio que tenga la información solicitada.

Este tipo de operativa hace que a medida que nuestro ecosistema de servicios crece, incrementa la latencia y la disponibilidad de la información si alguno de ellos no está disponible. Aquí entra en juego el patrón de microservicios CQRS que explicaremos más adelante, que consiste en emitir eventos desde nuestros microservicios cuando una operación tiene lugar para que vaya creando de manera pasiva una foto del estado de nuestra aplicación en lugar de hacerlo activamente cada vez que se requiera.

A continuación vamos a enumerar algunos de los problemas que conlleva la descomposición de un sistema en microservicios.

2.3.6. Obstáculos al descomponer una aplicación

Definir microservicios en base a las áreas de negocio parece sencillo si lo piensas de manera abstracta, pero conlleva ciertos problemas que mencionamos a continuación.

Latencia de red

Este es un problema habitual en sistemas distribuidos. Las conexiones entre los distintos servicios añaden un retardo de red, y se puede optimizar combinando servicios que detectamos que trabajan conjuntamente siempre, o haciendo operaciones por lotes en lugar de manera individual, pero siempre estará ahí.

Disponibilidad reducida por comunicaciones síncronas

Si tenemos dos (o más) servicios colaborando de manera síncrona, la caída de uno de ellos supondrá la caída del resto. Esto se puede solucionar implementando comunicaciones asíncronas mediante mensajería para que cada sistema ejecute las operaciones necesarias cuando estén disponibles sin afectar al resto.

Inconsistencia de datos

Otro obstáculo que tenemos es que al tener cada microservicio su propia base de datos, tenemos que encargarnos de mantener una consistencia entre todos los microservicios ya que no podemos hacer uso de las transacciones ACID de una base de datos, por lo tanto surge la

consistencia eventual. En comunicaciones síncronas se puede resolver mediante lo que se conoce Two-Phase Commit, pero preferimos optar por un sistema que, a cambio de complejidad, funciona mejor en más situaciones, un sistema de transacciones en sistemas distribuidos conocido como SAGAS.

God Classes

En todos los sistemas existe al menos una clase, que rompiendo con el SRP, lleva consigo propiedades y métodos que involucran distintos dominios de nuestra aplicación, y al estar involucrada en todo el sistema complica cuando queremos migrar un monolito a microservicios.

2.4. Comunicación entre microservicios

2.4.1. Tipos de comunicación

En el apartado anterior hablamos de cómo se decide una arquitectura de microservicios. Ahora toca hablar de los distintos métodos de comunicación entre ellos. Podemos dividirlo en dos tipos: sincronos y asincronos. Y a su vez, dividirlos según la cardinalidad de sus participantes: uno a uno (1-1) o uno a varios (1-N).

Tipo	**1-1**	**1-N**
Síncrono	Request/Response	No disponible
Asíncrono	async Request/Response One-Way notifications	Publish/Subscribe (pubsub) Publish/async Responses

Comunicación 1-1:

- Request/Response: Este tipo de comunicación hace una petición a un servicio y espera su respuesta. Aunque se espera una respuesta temprana, esto podría no ocurrir y bloquearía el servicio.
- Async Request/Response: Igual que el modelo anterior, pero el servicio no se queda bloqueado por la respuesta.
- One-Way notifications: Un servicio envía una petición, o mensaje, pero no espera ningún tipo de respuesta.

Comunicación 1-N:

- Publish/Subscribe: Un cliente publica un mensaje que puede ser consumido por ninguno o varios servicios.
- Publish/async Responses: Un cliente publica un mensaje y espera la respuesta de los servicios, normalmente mediante otro mensaje.

2.4.2. Definición de las APIs

Las APIs o interfaces es algo muy habitual en el desarrollo de software. Son una herramienta muy útil ya que permiten exponer las funcionalidades que están disponibles, y permiten cambiar la implementación de un módulo sin afectar a los clientes que lo utilizan.

Aún así hay que tener cuidado con los cambios de implementación, ya que como expone Hyrum, un ingeniero de Google, ante un contrato utilizado por muchos usuarios siempre habrá algún comportamiento en la implementación del que dependa uno de los usuarios de esa API o interfaz.

También hay que tener en cuenta que una interfaz que no se cumple en el desarrollo de software dará un fallo de compilación, mientras que un microservicio, completamente desacoplado podría cambiar su contrato y que lance el error en tiempo de ejecución, lo cual es algo no deseable. Por tanto, uno de los principales retos es definir una API correctamente, para ello hay herramientas como OpenAPI (https://www.openapis.org) que generan documentación sobre los servicios. Además de eso, hay herramientas como Pact Broker (https://pact.io) que es un repositorio de contratos para los servicios, y permiten testear que los servicios cumplen con su especificación, y al mismo tiempo, servir de mocks para los distintos clientes que necesiten esa API, para poder trabajar en desarrollo sin depender de un servicio de terceros.

Siendo cuidadosos en el proceso del apartado anterior, podemos hacer un desarrollo API-first, que consiste en definir bien nuestras APIs para que el resto de equipos puedan trabajar sin necesidad de que el servicio esté terminado.

2.4.3. Evolución de las APIs

Las aplicaciones evolucionan a lo largo del tiempo. Cuando realizamos cambios sobre un monolito la aplicación se compila de nuevo, en cambio cuando tenemos microservicios hay que tener más control sobre los cambios que hacemos sobre la API porque implica que el resto de equipos tendrán que incorporar en sus servicios que consuman esa API los cambios. Eso en el caso de APIs de uso interno, pero si éstas son utilizadas por otras empresas es aún más complejo, ya que no quieres tener que molestar a todos los clientes con el cambio.

Por esto, en el mundo de las APIs es muy importante el versionado de los servicios, y en la mayoría de los casos podrán convivir varias versiones del servicio al mismo tiempo.

Versionado semántico (semver)

Semver es una especificación que sirve de guía para versionar nuestro software mediante una serie de reglas con la que la mayoría de nosotros estamos familiarizados, su formato es MAJOR.MINOR.PATCH (p.e. 3.14.16), incrementar cada uno de estos números implican lo siguiente:

- Major: Cambio es incompatible con la versión anterior.
- Minor: Cambio o mejora compatible con la versión anterior (retrocompatible).
- Patch: Cambio retrocompatible para solucionar un error o bug.

Cambios Minor (retrocompatibles)

Siguiendo el principio de la robustez, que dice: “Sea conservador en lo que envía, sea liberal en lo que acepta”. Esto significa que los cambios que podemos hacer sobre una API deberían ser del siguiente tipo:

- Añadir atributos opcionales en una petición, añadiendo un valor por defecto para no romper las implementaciones existentes.
- Añadir atributos a una respuesta.
- Añadir nuevas operaciones.

Para que las integraciones sigan este principio significa que los clientes que usen nuestra API también deben aceptar valores adicionales en la respuesta.

Cambios Major (breaking changes)

En un mundo ideal, todos los cambios que hagamos sobre nuestro software serían retrocompatibles, pero a veces es necesario hacer cambios drásticos sobre nuestra interfaz o contratos, cuando esto ocurre hay que dar tiempo para que los consumidores de nuestras APIs puedan aplicar los cambios necesarios. Para ello hay que permitir que varias versiones de nuestro producto convivan, por ejemplo, añadiendo un prefijo en nuestra url (api/v1/, api/v2). También se puede hacer mediante la cabecera http de negociación de contenido (Accept: version=x), pero esto es menos habitual.

2.4.4. Formato del mensaje

La esencia de la comunicación entre servicios son los mensajes, ya sean sincronos, mediante HTTP o gRPC, o mediante colas. Hay dos tipos: de texto y binarios.

Mensajes de texto

Este tipo de mensajes son legibles por humanos, los formatos habituales suelen ser JSON y XML, los cuales se autodescriben al venir en formato de llave-valor. Este formato suele ser más aparatoso, o como se dice en inglés, *verbose*. Por lo tanto, los mensajes van a ser, dependiendo del caso, muy grandes y si tu sistema requiere optimizaciones extremas en el consumo de ancho de banda deberías plantearte los mensajes binarios.

XML	JSON

`<note>`
`<to>Tove</to>`
`<from>Jani</from>`
`<heading>Reminder</heading>`
`<body>Don't forget me this weekend!</body>`
`</note>`
	`{`
	`  "note":{`
	`  "to": "Tove",`
	`  "from": "Jani",`
	`  "heading": "Reminder",`
	`  "body": "Don't forget me!"`
	`  }`
	`}`

Mensajes binarios

Hay distintos formatos para mensajes binarios, uno de los más populares es Protocol buffer, de Google. Este formato consiste en diseñar un contrato, y la librería se encargará de generar el código que hará la serialización/deserialización del mensaje, permitiendo transmitir de manera más óptima el mensaje sin enviar datos innecesarios como la clave de las propiedades, o caracteres delimitadores como pueden ser “{}[],” para el formato JSON o “<>?” para XML.

Contrato	Mensaje original	Binario (HEX)

message User {
  required string email = 1;
  required int32 id = 2;
  optional string name = 3;
}
	{
	  "id": 2,
	  "email": "foo@bar.com",
	  "name": "Foo Bar"
	}
		0A 0B 66 6F 6F 40 62 61 72 2E 63 6F 6D 10 02 1A 07 46 6F 6F 20 42 61 72

Conversión realizada con la herramienta: http://yura415.github.io/js-protobuf-encode-decode/		

Mientras que el mensaje original tiene 47 bytes (uno por carácter, sin contar espacios y saltos de línea), el binario tiene 24 bytes, como vemos, la optimización es considerable, y se notará más en mensajes más largos, con arrays que incrementarían con la recurrencia del nombre de los campos, etc.

Otra ventaja de los mensajes binarios, es que si hacemos algún cambio del contrato nos daremos cuenta en tiempo de compilación.

2.4.5. Comunicación síncrona

Cuando se usa este mecanismo el cliente realiza una petición y espera una respuesta, dependiendo del cliente, éste podría quedar bloqueado hasta que reciba la respuesta, pero a diferencia que una comunicación asíncrona, este mecanismo implica que esperamos una respuesta casi inmediata.

2.5. Patrón SAGA

2.5.1. Introducción

Se ha hablado sobre cómo descomponer una arquitectura monolítica en microservicios y sus principales ventajas y desventajas. Uno de los problemas más importantes cuando hablamos de microservicios es la inconsistencia de datos.

Transacciones ACID

Cuando se tiene un monolito, operando sobre una base de datos podemos hacer uso de las transacciones ACID, las cuales proporcionan confianza e integridad sobre nuestros datos ya que nos protege contra fallos en nuestras operaciones de negocio que puedan quedar incompletas reflejando un estado erróneo en la base de datos.

Es un acrónimo que se refiere a cuatro propiedades que definen una transacción (del inglés **A**tomicity, **C**onsitency, **I**solation and **D**urability), que significan lo siguiente:

- **Atomicidad**: Cada operación de lectura y escritura es tratada como una única unidad, o se ejecutan todas o ninguna.
- **Consistencia**: Esta propiedad asegura que solo se empieza aquello que puede terminar y que cada operación de escritura va a llevar la base de datos de un estado válido a otro válido.
- **Aislamiento**: Asegura que una operación no va a afectar a otra aunque intenten escribir sobre el mismo dato controlando cuando un cambio es visible para el resto de operaciones.
- **Durabilidad**: Asegura que al terminar una operación se persiste el cambio y tenemos la seguridad de que no se va a perder ese dato.

Para conseguir transacciones distribuidas se podría hacer uso del patrón de commit en dos fases o 2PC (Two-Phase Commit), que consiste en que el microservicio que inicia la transacción hace una especie de orquestación, llamando a todos los microservicios involucrados y esperando su respuesta. Si uno de los microservicios falla, hará una llamada al resto para revertir la operación y no guardar nada en su estado. Este patrón tiene el gran inconveniente de usar una comunicación síncrona, reduciendo la disponibilidad total del sistema.

Disponibilidad del sistema

Es una métrica de mantenimiento usada para medir cuánto tiempo, en forma de porcentaje, que servicio está caído, calculando la probabilidad de que un sistema esté caído cuando se necesita. Su fórmula es la siguiente:

$$Disponibilidad(\%) = \frac{Tiempoactivo}{Tiempoactivo + Tiempoinactivo} * 100$$

Para considerarse tiempo en activo tiene que cumplir con tres características para considerarse en estado disponible:

- **Funcionalidad**: No está fuera de servicio para inspección o reparación.
- **Normalidad**: Funciona de manera ideal con el rendimiento esperado.
- **Disponibilidad**: Está disponible para uso en producción sin interrumpir programas.

En microservicios, la disponibilidad global del sistema se mide como el producto de la disponibilidad de cada servicio ya que si uno de ellos no está disponible afectará al resto en alguna medida.

$$DisponibilidadGlobal = \prod_{n}^{i} Disponibilidad_i$$

2.5.3. Descripción de Saga

Cuándo trabajamos en microservicios solo disponemos de transacciones ACID a nivel local, dentro de cada microservicio. Para poder realizar transacciones habrá que hacer uso del patrón Saga, que consiste en una secuencia de transacciones locales a nivel de microservicio, que se van propagando por mensajes al resto.

El mayor desafío con este patrón es que son transacciones ACD, sin la I de aislamiento.

Cuándo una de las operaciones falle, ese microservicio mandará un mensaje para que los microservicios hagan un rollback de sus transacciones. Cómo cada una de las transacciones ejecutadas satisfactoriamente han hecho cambios en la base de datos visibles por los otros microservicios, no se puede simplemente deshacer el cambio, generando inconsistencia con el resto de microservicios, en su lugar habrá que hacer lo que se conoce como transacción de compensación.

2.5.3 Transacciones y compensación

La gran ventaja de las transacciones ACID es que podemos deshacer los cambios si alguna de la lógica de negocio ha fallado. Ésto no puede ocurrir en SAGAs, por lo tanto, hay que aplicar la metodología de mensajes transaccionales, en los que cada transacción local puede ser compensada. Nótese que se habla de compensación y no de reversión, ya que, por la falta de aislamiento, el resto de participantes pueden haber iniciado otras operaciones irreversibles, por tanto, toca iniciar una secuencia que revierta las acciones tomadas.

Suponiendo una saga con *N* transacciones, si la transacción *n+1*, el resto de transacciones desde n hasta 0 deben ser revertidas.

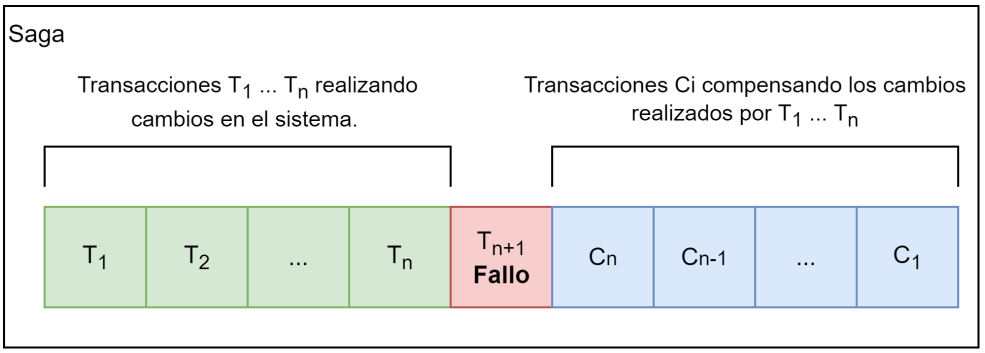

Figura 2. Transacciones de compensación en saga.

Cabe mencionar que en un saga no todas las operaciones son de escritura, por lo tanto, una operación de lectura, cómo puede ser una autorización de una tarjeta, en la cual no se ha hecho ningún cargo. Sin embargo, si se ha hecho un cargo en la tarjeta, haría falta una transacción compensatoria que iniciaría una devolución.

2.5.4 Tipos de coordinación

La implementación del patrón saga consiste en la lógica que coordina los distintos pasos de la saga. Cuando una saga es iniciada, la lógica debe encargarse de invocar al primer involucrado y hacer un seguimiento de la secuencia de transacciones hasta que termine el proceso. Si alguna transacción falla, es el encargado de emitir las transacciones que compensarán esos cambios.Hay dos tipos de coordinación a la hora de implementar esta lógica: Coreografía y Orquestación.

Coreografía

En este tipo de organización la toma de decisiones y la secuenciación de las acciones se distribuyen entre todos los participantes de la saga, comunicándose entre ellos principalmente con eventos. Cómo no hay un coordinador central, cada uno de los participantes de la saga se suscriben a los eventos de los otros y publican sus propios eventos en respuesta, desencadenando una cadena de mensajes. A continuación, se muestra un ejemplo:

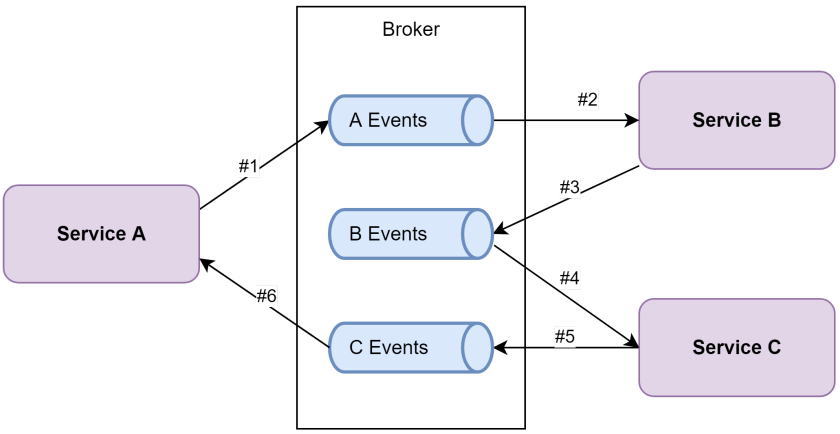

Figura 3. Patrón saga coordinado mediante coreografía.

Esta coordinación tiene dos beneficios principales:

- **Simplicidad**: Los servicios se encargan simplemente de publicar mensajes cuando un objeto de negocio cambia.
- **Desacoplamiento**: Los distintos integrantes se suscriben a eventos y no tienen mucho conocimiento de que hacen los demás.

En cambio, tiene las siguientes desventajas:

- **Dificultad**: A diferencia de la orquestación, al no haber un coordinador principal, para poder entender todo el flujo de transacciones que transcurren en la saga. Para poder entender todo el flujo se tiene que explorar cada uno de los servicios y mirar los eventos que publican y a los que están suscritos.
- **Riesgo de acoplamiento**: Una de sus principales ventajas puede convertirse en una contra ya que cada participante de la saga tiene que estar suscrito a todos los eventos que le afecten, y que si hay un cambio en un flujo de trabajo de la orquestación uno o más participantes tengan que cambiar.
- **Dependencias cíclicas**: Pueden existir este tipo de dependencias en las que dos servicios se invoquen de manera cíclica. En principio no tiene porqué suponer un problema, pero es un Design Smell.

Este tipo de coordinación funciona bien en sagas simples, pero si el caso es más complejo puede ser más interesante coordinar por orquestación ya que si el flujo de la saga es muy complicado y largo, hacer un cambio sobre este puede suponer el cambio de múltiples participantes.

Orquestación

A diferencia de en la coreografía, en la orquestación hay un integrante de la saga que conoce toda la secuencia, cuya responsabilidad será indicar al resto qué operaciones realizar. Éstos responderán con un mensaje al terminar dichas acciones y el orquestador es el que sabe cual es la siguiente operación a ejecutar. Así continuamente hasta terminar la secuencia.

Una buena forma de diseñar una orquestación es como una máquina de estados, la cual consiste de un set de estados y de transiciones entre estos que son disparadas por eventos de cada participante. Cada transición puede tener asociada una acción que termina con la finalización de una transacción local por uno de los participantes. A continuación se muestra un diagrama mostrando cómo funciona la orquestación:

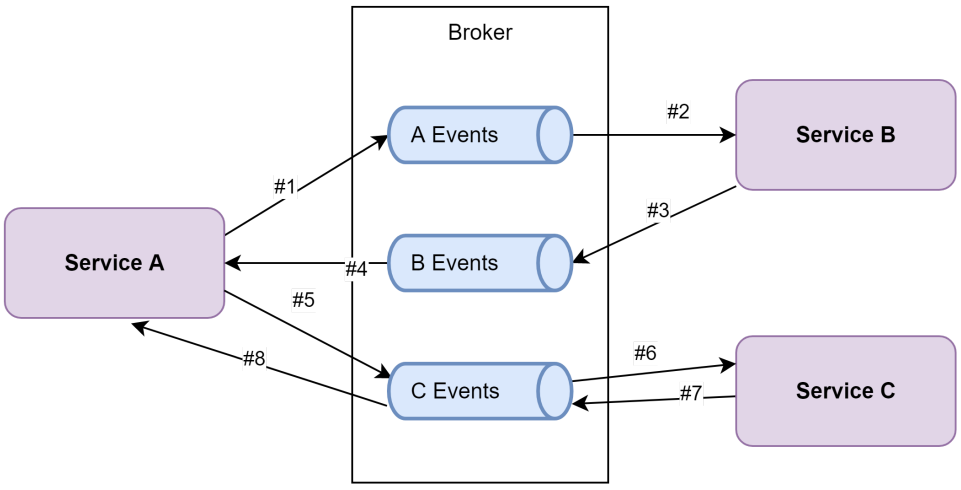

Figura 4. Patrón saga coordinado mediante orquestación.

Este modelo presenta las siguientes ventajas:

- **Dependencias simplificadas**: Se eliminan las dependencias cíclicas, el orquestador depende de los participantes pero los participantes son independientes y no tienen que preocuparse del resto de integrantes.
- **Acoplamiento reducido:** Cada participante es invocado por el orquestador, por lo tanto este no tiene que tener conocimiento del resto.

En cambio este modelo trae las siguientes desventajas:

- **Único punto de fallo o SPOF**: del inglés Single Point of Failure, si el orquestador no está disponible el resto de servicios van a quedar a la espera para poder continuar la secuencia.
- **Lógica centralizada:** Si no se tiene cuidado, este modelo puede convertir a los participantes en servicios “tontos” controlados por el orquestador. Esto se puede evitar si el orquestador se encarga únicamente de dirigir la secuencia y no almacena lógica de negocio.

Híbrido

No existe una buena y única solución que resuelva todos los problemas, por eso se pueden combinar ambos sistemas entre ellos e incluso dividir en subsistemas y aplicar coreografías u orquestaciones de manera aislada y que formen parte de una orquestación mayor.

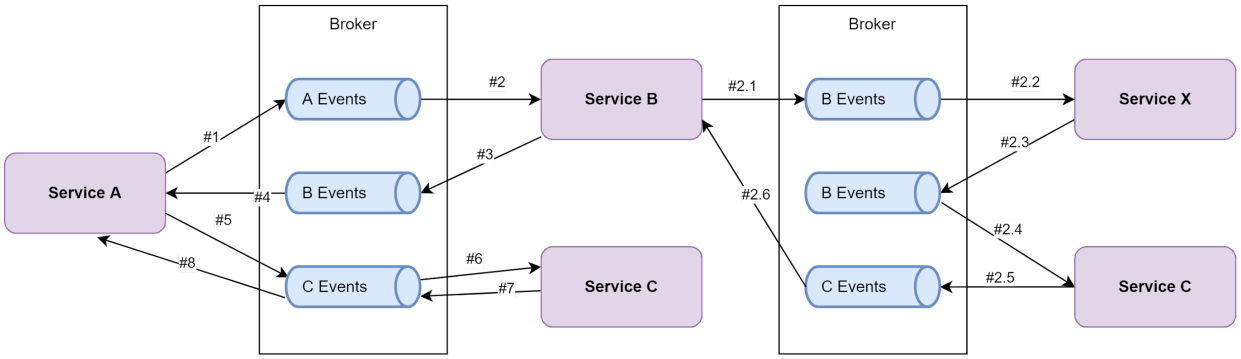

Figura 5. Coordinación híbrida en sagas.

2.5.5. Manejando la falta de aislamiento

Debido a la falta de aislamiento en sagas con las transacciones locales, un cambio realizado en un participante de la saga es inmediatamente visible por otras participantes una vez la transacción local es realizada, esto puede provocar que otros servicios lean y modifiquen un estado en medio de una transacción saga y ocurran lo que se conoce como anomalías.

Anomalías

Actualizaciones perdidas

Esta anomalía ocurre cuando un saga sobrescribe una actualización realizada por otro, por ejemplo, si un pedido es cancelado justo cuando se ha procesado un pago, sobreescribiendo el estado de cancelado por el de pagado.

Conflicto de escritura-lectura (aka Dirty Reads)

Consiste en la lectura de información en medio de una transacción saga, en la que se ha persistido una transacción local en uno de los participantes pero no ha terminado la transacción saga, por tanto, otro participante podría leer esta información y tomar acciones en base a ésta. Posteriormente, si hay un rollback de la transacción saga, se haría una compensación sobre esa transacción local previa, resultando en un estado inconsistente en el otro participante que utilizó esa información.

Medidas para manejar la falta de aislamiento

Para prevenir este tipo de anomalías se puede recurrir a las siguientes medidas:

- **Lock semántico**: Bloqueo a nivel de aplicación.
- **Actualizaciones conmutativas**: Diseñar las operaciones para poder ser ejecutadas en cualquier orden.
- **Visión pesimista**: Reorganizar las transacciones de una saga para minimizar los riesgos.
- **Versionado**: Grabar los resultados en base de datos de tal manera que puedan ser reordenados.

Estructura de una transacción saga

Antes de proceder a como aplicar las medidas aquí tenemos los tres tipos de transacciones que componen una transacción saga.

- **Transacciones compensables**: Son aquellas que realizan cambios en el estado y que pueden ser compensadas.
- **Transacciones pivote:** Son puntos “go/no-go” en la saga. Si una transacción pivote se guarda, la saga continuará hasta completarse. Además, esta transacción no se puede compensar ni recuperar. Esta puede ser la última transacción compensable o la primera recuperable.
- **Transacción recuperable**: Transacciones que siguen a la pivotal y que su éxito está asegurado.

2.6. Command Query Responsibility Segregation (CQRS)

Cuando se adopta una arquitectura de microservicios se complica la labor de consultar datos. Mientras que en un monolito podíamos hacer una consulta accediendo a todas las tablas y crear una API que exponga toda la información necesaria por parte del cliente consumidor, en microservicios no se puede ya que cada uno tiene su base de datos. Podría usarse una misma base de datos, pero rompería con una de las principales ventajas de los microservicios: la encapsulación.

Existen dos soluciones para este problema: Patrón de composición de APIs y CQRS.

2.6.1. Patrón de composición de APIs

Es la solución más simple, y debería ser usada siempre que sea posible. Consiste en crear un componente que se encarga de hacer las consultas a los distintos servicios. Ésto podría ser en el propio cliente, como una aplicación web, pero esto puede ser problemático ya que puede ser que los servicios se encuentren en una red privada. Se muestra a continuación cómo sería.

Cómo alternativa, se puede crear otro servicio, una API Gateway, que sea responsable de invocar a las APIs que sean necesarias y componer sus resultados y devolverlos al cliente consumidor como se muestra a continuación.

Puede hacerse un API Gateway por cliente si es necesario, y conveniente. Una desventaja es que en algunos escenarios puede resultar a tener un consumo de memoria excesivo si tiene que estar juntando grandes conjuntos de datos.

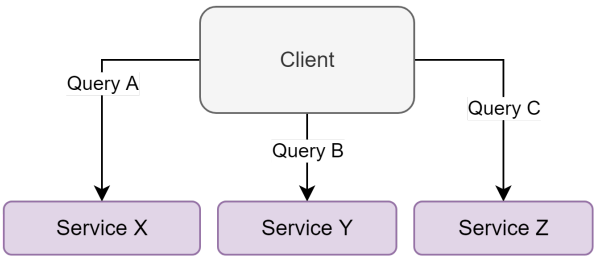

Figura 6. Composición en Cliente.

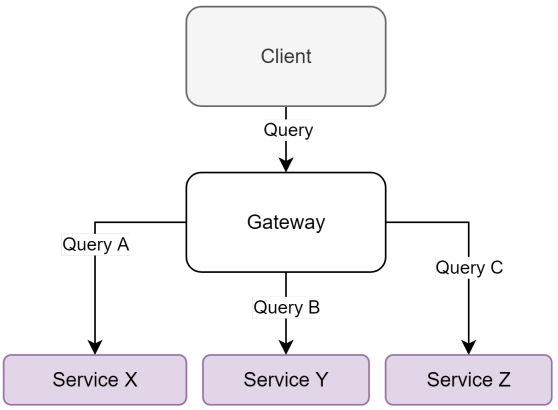

Figura 7. Api Gateway.

Beneficios y desventajas

El principal beneficio de este patrón es que es simple e intuitivo, pero tiene varias desventajas, que son las siguientes:

Sobrecarga

Es el equivalente a un monolito en el que se unifica toda la información en una consulta, pero claro, en este caso supone hacer múltiples llamadas a distintos servicios, lo que conlleva más gasto de computación, tráfico de red y latencia, por lo tanto incrementa el gasto de ejecutar esta aplicación.

Riesgo de disponibilidad reducida

Al seleccionar una arquitectura de microservicios se trata de incrementar la disponibilidad de los servicios al separar las distintas áreas de negocio con sus correspondientes recursos necesarios para eliminar la dependencia y poder seguir operando lo mejor posible si uno de ellos falla. Al aplicar este patrón, se tiene que invocar a todos los servicios involucrados y la aplicación vuelve a ser dependiente de ellos.

Inconsistencia de datos

Se ha descrito previamente en el patrón saga como funcionan los mensajes transaccionales. Si hacemos la composición de APIs de esta manera puede darse el caso que el API Gateway llame a un Servicio X con un resultado tras realizar una transacción y al mismo tiempo hacer una consulta al Servicio Y en el que no se ha propagado todavía la transacción del otro. A esto se le conoce como consistencia eventual.

Sabiendo ésto se propone a continuación una solución muy extendida conocida como Command Query Responsibility Segregation, en adelante CQRS.

2.6.2. Introducción al parón CQRS.

En arquitecturas tradicionales se utiliza el modelo de datos para consultar y actualizar una base de datos, es sencillo y es válido para hacer operaciones CRUD básicas. Pero cuando las aplicaciones se hacen más complejas, este modelo puede complicar las cosas.

El patrón CQRS permite desacoplar las lecturas de las escrituras, ya que en una app pueden haber muchos requisitos sobre cómo se quiere acceder a los datos, y se pueden construir distintos DTO (Objetos de transferencia de datos) según la necesidad del consumidor. Además del modelo de datos, puede que haga falta distintos requisitos de rendimiento y escalabilidad.

Para separar las lecturas de las escrituras, se separan las operaciones en dos tipos:

- **Comandos:** que son las operaciones de escritura. Deben de centrarse en tareas o casos de uso, por ejemplo: *CancelarPedido* en lugar de SetStatusTo(Cancelado). Los comandos además, se pueden encolar para procesar de manera asíncrona.
- **Consultas**: operaciones de lectura.

Además de separar la manera en la que se interactúa, también se puede cambiar el almacenamiento, y utilizar, por ejemplo, una base de datos relacional para la escritura y una no relacional para la lectura. El principal beneficio es la flexibilidad y escalabilidad, en cambio, las desventajas son complejidad adicional y que los datos de lectura pueden estar más tiempo desactualizados tras una actualización realizada por un comando. Por tanto, no se recomienda si la aplicación va a tener un único interfaz de usuario o si las reglas de negocio son simples.

3. Docker

3.1. Virtualización

Cuando trabajamos en un proyecto de software hace falta preparar un entorno en el que nuestra aplicación pueda ser ejecutada, esto conlleva instalar dependencias tanto a nivel de sistema como de aplicación. Esto supone que cada vez que haya que preparar un entorno de desarrollo nuevo, tanto para un desarrollador nuevo en el equipo como para un despliegue en una máquina nueva habrá que dedicarle tiempo a la preparación de todas estas dependencias del sistema, además, una mínima actualización del sistema operativo podría hacer dejar de funcionar la aplicación por una dependencia, o peor aún, hacer que algo funcione en nuestro entorno de desarrollo y no en el de producción. Por tanto, también es necesario que los entornos de desarrollo y producción estén alineados.

Para solucionar este problema existe la **virtualización**, que nos permite, dicho de manera sencilla, correr uno o varios sistemas invitados (*Guest*) dentro de otro sistema anfitrión (*Host*) en forma de máquina virtual (VM). Ésta nos abstrae del hardware y por tanto nos permite aislar piezas de software del entorno en el que se ejecuten, haciendo su funcionamiento consistente entre los distintos posibles entornos desde local a producción.

Un concepto importante para entender la virtualización, y los beneficios de **Docker,** es el ***Hypervisor***, que es un software sobre el que se ejecutan las VM, como intermediario entre el Host y el Guest. Este software se encarga del encendido y apagado, de reservar los recursos necesarios (CPU, RAM…), adaptadores de red y de traducir las órdenes de éstas al host, entre otras tantas.

Existen dos tipos de Hypervisor:

Type 1

También conocido como *bare-metal*, este Hypervisor se ejecuta directamente sobre el hardware físico de la máquina. No tiene que ejecutar un sistema operativo entre medias y traduce directamente todas las instrucciones de las VM al hardware donde se ejecutan.

Este tipo, además de tener un buen rendimiento, es también conocido por ser muy seguro, al estar las máquinas virtuales en ejecución completamente aisladas.

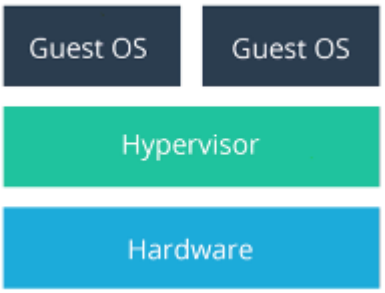

Figura 8. Virtualización con Hypervirsor Type 1.

Type 2

La principal diferencia con el primero es que estos Hypervisor son instalados en un sistema operativo, el cual será el encargado de administrar el uso de la CPU y otros recursos. Por tanto, se podría decir que este tipo se encarga de traducir órdenes de un sistema operativo a otro.

Además de la falta de aislamiento, su principal desventaja es que al añadir un sistema operativo anfitrión incrementa el uso de recursos y la latencia en la ejecución del software en las máquinas virtuales.

En algún momento de nuestra vida hemos utilizado este tipo, por ejemplo: Virtual Box o VMWare.

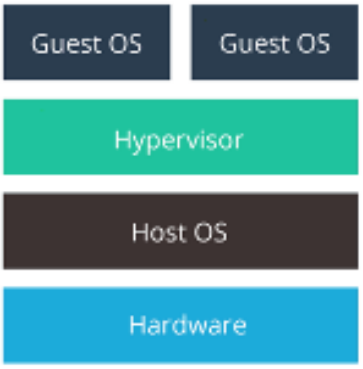

Figura 9. Virtualización con Hypervirsor Type 2.

3.2. Contenedores

Un contenedor es un entorno aislado de ejecución en el que uno o varios procesos pueden ejecutarse de manera aislada como un único proceso. La acción de crear contenedores y ejecutar una aplicación como un proceso es conocido como *containerization*. El Kernel de Linux tiene un mecanismo llamado Linux Based Containers (LXC), para hacer esto posible hace uso de Namespaces y Control Groups.

Los Namespaces permiten particionar recursos del Kernel como el interfaz de red para obtener una IP, volúmenes montados para tener un sistema de ficheros e identificadores de procesos (PID).

Los Control Group se encargan de controlar cuántos recursos del sistema como la CPU y la memoria son destinados a esos procesos.

Cada contenedor tiene un namespace y todos los procesos corriendo dentro de él tendrán acceso a sus recursos compartidos, asignados por el control group del contenedor. Un proceso dentro de este contenedor no podrá acceder a los recursos asignados a otro contenedor. Todos los contenedores comparten el mismo Kernel, si algún contenedor necesita un Kernel distinto entonces hay que hacer uso de la virtualización mencionada previamente.

En general, estos mecanismos que crean containers se les conoce como Container Engine.

Contenedores vs Máquinas Virtuales (VM)

Ambos son capaces de hacer, en distintos niveles, una aislación de la aplicación ejecutada, pero funcionan de distinta manera.
Una máquina virtual arranca un sistema operativo completo, ya sea en un Hypervisor Type 1 o 2. Esto bloquea los recursos del sistema en el momento de su creación. Además estas máquinas virtuales necesitan tener un sistema operativo, con todos sus binarios, librerías y dependencias para el hardware sobre el que se ejecuta. Por lo tanto, estas máquinas virtuales son muy pesadas.

Al contrario, un contenedor es un paquete que contiene los binarios y librerías de la aplicación, así como sus variables de entorno. Cómo utiliza el Kernel del sistema operativo anfitrión, los binarios y drivers del sistema operativo no son necesarios. Además, un contenedor no bloquea los recursos del sistema, y los irá solicitando según los vaya necesitando.

Por lo tanto, un contenedor es más ligero en comparación a una máquina virtual tanto en espacio, como en los recursos que consume (al no bloquearlos) y pueden inicializarse más rápido que una VM que tiene que arrancar también el sistema operativo.

3.3. ¿Qué es Docker?

Ahora que entendemos el concepto de virtualización y *containerization* podemos hablar sobre Docker, el cual es un Container Engine que ha creado un standard y ha facilitado la portabilidad de contenedores.

La pieza principal es el Docker Daemon, un proceso ejecutándose en segundo plano que recibe comandos de un cliente de Docker local o remoto usando el protocolo HTTP REST y es el encargado de iniciar, parar y administrar los contenedores.

Contenedores de Docker

Los contenedores de Docker contienen el código de la aplicación y otras dependencias específicas de la aplicación. Por ejemplo, una aplicación de NodeJS necesitaría:

- Binarios de NodeJS (node & npm)
- Dependencias de la aplicación, que se encuentran en node_modules
- Código de la aplicación, que serían nuestros ficheros javascript.

Imágenes de Docker

Para crear un contenedor primero hay que crear una imagen con la aplicación, librerías y dependencias dentro. En el momento que una imagen se pone en ejecución recibe el nombre de contenedor.

Las imágenes se definen mediante un fichero en texto plano comúnmente llamado Dockerfile, que contiene la configuración de la máquina virtual que queremos utilizar.

Esta configuración se realiza mediante una serie de comandos. Los comandos fundamentales son:

Comando	
FROM	Este comando es obligatorio, y se sitúa siempre en primera posición, a excepción de ARG que es opcional. Se encarga de definir la imagen base sobre la que vamos a trabajar. Esta imagen puede ser la imagen *base* o una imagen publicada en un registro como podría ser node:13.10.1-alpine, que sería una imagen preparada para correr una aplicación en node v13.10.1 sobre el sistema operativo de Linux alpine. Es la forma de poder ir extendiendo las imágenes.

RUN	Con este comando podemos ejecutar instrucciones unix como crear directorios o dar permisos a un usuario.
COPY	Este comando con el siguiente formato permite copiar código de nuestra máquina a la imagen en el momento de construirse.
CMD	Solo puede haber una instrucción CMD por fichero Dockerfile, que es el que indica que será ejecutado en la imagen cuando esté corriendo como un contenedor, por ejemplo una aplicación de NodeJS.

Fuente https://docs.docker.com/engine/reference/builder/

El identificador de una imagen tiene dos partes:

- Repositorio: Un texto descriptivo para saber que tiene nuestra imagen, por ejemplo: app/backend, app/frontend, app/database, etc
- Etiqueta: Una imagen puede tener varias etiquetas, normalmente se usan para indicar la versión de imagen, por ejemplo: app-backend:1.4.3. Se pueden aplicar varias etiquetas a la misma imagen, y a veces se etiquetan de una forma semántica como puede ser *latest*.

Docker Layers

Docker trabaja con un sistema de ficheros por capas llamado Union File System. Esto es importante para entender cómo se construyen las imágenes de Docker, ya que cada instrucción que se ejecuta va a generar un cambio sobre el sistema de ficheros, en el que cada capa tiene la diferencia con su estado anterior. Estas capas suelen ser almacenadas en una caché, lo cual aumenta la velocidad de la reconstrucción de una imagen. Cada vez que una capa cambie, ésta será invalidada y por tanto las posteriores.

Este concepto es importante ya que un buen uso y ordenamiento de los comandos puede optimizar el tiempo de creación de las imágenes. El tiempo que tarda en construirse las imágenes en nuestros procesos CI/CD se traduce en menos agilidad para desplegar cambios, y a su vez en dinero por el consumo de las máquinas que se encargan de construir las imágenes.

Construcción de imágenes

Una vez hemos definido la imagen de Docker llega el momento de construirla. Para ello se hace uso de la interfaz de comandos y su comando *build* que tiene la siguiente definición:

```
docker build [OPTIONS] PATH | URL | -
```

Para construir una imagen necesitamos un Dockerfile, previamente explicado, y un contexto. El contexto pueden ser de dos tipos:

- PATH: delimitará los ficheros a los que podrán acceder Docker durante su construcción como, por ejemplo, el comando COPY, permitiendo acceder al directorio del contexto y sus subdirectorios únicamente.
  P.E. /home/developer1/work/app/backend
- URL: apunta a un repositorio de git, docker se encargará de clonar el repositorio e iniciar la construcción de nuestra imagen sobre la raíz de ese repositorio. Además podemos hacer referencias a distintas ramas, commit o carpetas de ese repositorio. Algunos ejemplos:
  - github.com/org/app-backend.git
  - github.com/org/app-backend.git#feature-branch

Entre las opciones también podemos definir cual es el Dockerfile que queremos utilizar, por defecto se buscará un fichero llamado *Dockerfile.* Pero podemos definirlo con el parámetro -f, podría darse el caso que queramos guardar en nuestro repositorio varios ficheros de Docker, uno para entorno de desarrollo y otro productivo, en el que quitaremos dependencias para hacer depuraciones.

Además del fichero Dockerfile hay otro importante de mencionar, el fichero .dockerignore, que al igual que en un .gitignore se definen los ficheros que no queremos dentro de nuestras imágenes, por ejemplo un fichero con las variables de entorno.

Puedes consultar el resto de la documentación aquí: https://docs.docker.com/engine/reference/commandline/build/

Docker Registry

Una vez se ha construido una imagen de Docker se tiene que almacenar para que el Docker Daemon sea capaz de obtener la imagen y poner los contenedores en ejecución. Para ello existen los registros de imágenes donde se pueden publicar.
En Docker tenemos un registro local en el cual se publican automáticamente tras construirse y nos permitirá utilizar esas imágenes en nuestro entorno local si tenemos Docker en ejecución.

Después, mediante el comando *push* podremos publicar estas imágenes en un repositorio que puede ser público o privado, del cual el daemon hará posteriormente un *pull* para descargar la imagen.

Publicación de imágenes

Como se ha mencionado, tras haber construido una imagen de docker tenemos que publicarla para poder ser compartida mediante el comando push.

```
docker push [OPTIONS] NAME[:TAG]
```

Para ello, además de el nombre y las etiquetas hay que definir la url del registry de docker, por ejemplo: docker push my-registry.com:8081/app/backend:1.4.3

3.4. Docker Compose

3.4.1. Descripción

Es una herramienta para definir y ejecutar aplicaciones multicontenedor de Docker con un solo comando. Te permite definir las redes, los volúmenes y crear servicios para comunicarse internamente entre ellos.

Resulta de gran utilidad ya que mediante un único fichero llamado *docker-compose.yml* nos permite configurar cómo se tienen que construir y qué imágenes se tienen que ejecutar definiendo los mismos parámetros que se pasan a *docker build* y *docker run.*

No es muy utilizado en entornos productivos, pero es una herramienta de gran utilidad en entornos de desarrollo ya que te permite simular, más o menos y de manera ligera, un entorno productivo ejecutado sobre cualquier plataforma.

Para levantar la aplicación solo hace falta ejecutar el comando up de su binario: `docker-compose up.`

3.4.2. El fichero docker-compose.yml

El fichero es en formato YAML y define en su raíz los siguientes parámetros:

- **version**: Es informativo pero sirve para definir la versión de docker-compose necesaria.
- **services**: Aquí definimos cada uno de los servicios que queremos construir y arrancar.
- **networks**: Definimos las redes que necesitamos, para poder comunicar los servicios entre si, o incluso con servicios en otro docker-compose.
- **volumes**: Permite definir volúmenes de persistencia.
- **configs**: Permite modificar la configuración de una imagen de docker sin tener que reconsturirla.
- **secrets:** Como la configuración, pero con datos sensibles, que se pueden manejar desde el gestor de secretos de Docker.

Un fichero de configuración, por tanto luce como a continuación:

```
version: '3.7'
services:
 service-a:
    ... other parameters
 service-b:
    ... other parameters
networks:
 main_network:
   driver: bridge
 alt_network:
   driver: bridge
volume:
   driver: {}
```

Servicios

El apartado *services* es la pieza más relevante, en ella podemos definir los siguientes parámetros:

- **container_name**: El nombre que queremos darle al contenedor en ejecución.
- **ports**: Mapping de puertos internos y externos
- **links**: Lista de servicios con los que queremos que se conecte.
- **networks**: Lista de redes donde se requiere que esté presente.
- **environment**: Mapping de variables de entorno.
- **build**: Parámetros para la construcción de la imagen de Docker, como el path al fichero Dockerfile o el contexto.
- **volumes**: Mapping de volúmenes indicando el volumen externo y la ruta dentro del contenedor.
- **depends_on**: Servicios que tiene que esperar a estar en pie para arrancar después. Pero es muy básico y espera a que el contenedor esté listo, pero no la aplicación de dentro.

A continuación se muestra un ejemplo con dos servicios:

```
service-a:
  depends_on:
    - service-b
  container_name: service-a
  volumes:
    - hostPath:containerPath
  ports:
    - hostPort:containerPort
  links:
    - service-c
  networks:
    - main_network
  environment:
    - SOME_ENV=some_value
  build:
    context: .
    dockerfile: path/to/Dockerfile
service-b:
  image: docker-image:tag
  container_name: service-b
  networks:
    - main_network
    - alt_network
```

3.5. Conclusión

Docker es una herramienta que facilita mucho la portabilidad y la definición de los requisitos de nuestra aplicación. Todo esto en un fichero de texto, lo cual permite que se puede mantener con mayor facilidad incluyendo ese fichero en nuestros repositorios de código (Git, Svn, etc…), y facilita la detección de bugs al poder ver con exactitud cuales son los cambios realizados.

Además, tiene una baja curva de aprendizaje inicial, y una gran aceptación por parte de la comunidad, haciendo que sea muy popular, y esto se traduce en un buen soporte. Finalmente, cabe mencionar que es el combo por defecto a utilizar junto a Kubernetes ya que la mayoría de documentación está escrita para ambos.

Todo esto lo convierte en la elección de la mayoría de proyectos a día de hoy,

4. Kubernetes

4.1. Antecedentes

Para entender los beneficios que ofrece Kubernetes hay que entender cómo ha evolucionado el despliegue de aplicaciones, esto se puede categorizar en tres épocas.

Despliegue tradicional

Inicialmente se hacían despliegues en máquinas físicas, se podía almacenar varias aplicaciones en el mismo servidor, que podía ocasionar que una aplicación consumiese todos los recursos, dejando al resto sin ellos. Otra alternativa era utilizar una máquina por aplicación, pero eso podía suponer desaprovechar demasiados recursos, a pesar de ganar aislamiento, y mantener ese sistema es costoso.

Despliegue de máquinas virtuales

Como se ha mencionado anteriormente, las máquinas virtuales ofrecen muchas ventajas. Con este modelo podemos tener varias aplicaciones en un mismo servidor, con aislamiento y limitación de los recursos que va a utilizar cada máquina virtual. Además facilita el escalamiento de los servicios.

Despliegue de contenedores

La última tendencia hace uso de tecnologías como los contenedores, que herramientas como Docker te ofrecen una portabilidad muy alta y un consumo de recursos inferior, al compartir el sistema operativo y poder pedir recursos bajo demanda.

4.2. ¿Qué es Kubernetes?

Con los contenedores tenemos resuelto el soporte de las aplicaciones, y ahora nos enfrentamos a otro problema, que es conseguir que nuestra aplicación esté disponible siempre, comprobar el estado de cada contenedor y administrarlos para que se vayan arrancando (rollout) o parando (rollback) según la demanda.
Esas capacidades son las que ofrece Kubernetes, un framework para correr sistemas distribuidos de manera resiliente. Sus prestaciones principales son:

Descubrimiento de servicios y balanceo de carga

Cuando tienes varios contenedores en funcionamiento tienes que asignarles una dirección IP y repartir la carga entre todos ellos. Kubernetes se encarga de esta labor, además no necesitas saber la IP ya que puedes asignarle un nombre a cada servicio y será resuelto internamente.

Orquestación de almacenamiento

Con Kubernetes puedes tener varios sistemas de almacenamiento tanto local como por red y compartirlo entre distintos servicios.

Bin Packing automático

El problema de Bin Packing consiste en optimizar el número de máquinas físicas (nodos) necesarias para albergar los contenedores con los recursos necesarios, sin dejar recursos libres sin poder ser utilizados de manera eficiente. Kubernetes se encarga de esto.

Automatización de rollouts y rollbacks

Cuando quieres aplicar cambios a los contenedores, por ejemplo para hacer el despliegue de una nueva versión, Kubernetes se encarga de retirar los contenedores antiguos mientras despliega los nuevos de manera progresiva, y si falla algo, deshará los cambios, lo cual es bastante cómodo.

Recuperación automática

Kubernetes se encarga de restaurar o reemplazar los contenedores que están fallando o no responden comprobando periódicamente los health check definidos por el usuario.

Gestión de secretos

Una buena práctica a la hora de crear contenedores es hacer uso de variables de entorno y que la imagen no almacene esta información por razones de seguridad. Además, esto permite que podamos crear un contenedor con la misma imagen para distintos entornos. La gestión de estas variables de entorno puede hacerse desde Kubernetes, y cuando arranque los contenedores se encargará de asignar las variables de entorno.

4.3. Conceptos

4.3.1. Componentes

Un clúster de Kubernetes consiste en una serie de máquinas, los nodos, que ejecutan aplicaciones en contenedores. Cada clúster debe tener al menos un nodo.

Los nodos alojan los pods, que son los componentes de una carga de trabajo. El *control plane* administra los nodos y los pods en un cluster. Se encarga de tomar decisiones globales sobre el cluster y escuchar a sus eventos. Puede ejecutarse en cualquier máquina dentro del cluster, pero normalmente se ejecuta sobre una única máquina en la que no se ejecutan los contenedores de los usuarios. Está formado por los siguientes componentes:

- **kube-apiserver**. Es el componente que expone la API de kubernetes, actúa de frontend para el control plane. Puede escalar horizontalmente.
- **etcd:** Almacenamiento llave valor (kv) para almacenar toda la info del cluster.
- **kube-scheduler**: Se encarga de vigilar los Pods que no han sido asignados a un nodo y elige en cual será ejecutado utilizando distintos factores como el hardware, policies y otras especificaciones.
- **kube-controller-manager:** Se encarga de gestionar los nodos, los jobs, servicios pods y otros recursos.

A nivel de nodo se encuentra los siguientes componentes:

- **kubelet**: Es un agente que se ejecuta en cada nodo y se asegura que los contenedores se están ejecutando en algún pod y que lo hacen de manera sana.
- **kube-proxy**: Es un proxy de red que implementa el concepto de servicio en kubernetes.

4.3.2. Nodo

Definición

Un nodo es una máquina de trabajo en Kubernetes, puede ser física o virtual dependiendo de la configuración del clúster y es sobre donde se van a desplegar los Pod.

Estado

El estado de un nodo viene definido por los siguientes aspectos:

- Direcciones:
    - **HostName**
    - **ExternalIP**: Accesible fuera del clúster
    - **InternalIP**: Accesible sólo dentro del mismo clúster
- Condiciones: Cuando un nodo está en ejecución puede tener presentar las siguientes condiciones:
    - **OutOfDisk**: no hay espacio en disco suficiente para ejecutar un pod.
    - **Ready:** está en buen estado y listo para ejecutar nuevos pod.
    - **MemoryPressure**: el consumo de memoria es elevado.

    - **PIDPressure**: demasiados procesos en ejecución.
    - **DiskPressure**: la capacidad del disco es baja.
    - **NetworkUnavailable**: hay un error en la configuración de red.
- Capacidad: Describe los recursos disponibles en el nodo como la CPU, memoria y la cantidad máxima de pods que puede ejecutar.
- Info: contiene información sobre el sistema operativo del nodo, la versión de Kubernetes, datos de los contenedores y otra información general.

4.3.3 Pod

Definición

Los Pod son la unidad de computación mínima que se puede desplegar en Kubernetes. Éste puede contener y ejecutar uno o varios contenedores acoplados y relacionados entre sí con un almacenamiento y red compartido. Tiene una única IP, por lo tanto cada uno de los contenedores deberán tener un puerto expuesto distinto y se pueden comunicar con otros contenedores mediante el dominio *localhost.*
Aunque pueden ejecutarse varios contenedores en un mismo pod, la técnica más común es desplegar un único contenedor por pod.

Ciclo de vida

A cada Pod, en el momento de creación se le asigna un UID. Éste pod se ejecuta hasta que termina su ejecución o es detenido. Los pods no tienen capacidad para curarse a sí mismos. Si el nodo donde se encuentran falla son eliminados, y si vuelven a ser lanzados serán siempre con un UID distinto, aunque se puede mantener el mismo nombre. Si un recurso tiene el mismo tiempo de vida, como un volumen, esto significa que existirá tanto como lo haga el Pod con el que está enlazado.

Fases de un Pod

El estado de un pod viene definido por un objeto de tipo *PodStatus*, que tiene un campo llamado *phase.*
Esta fase indica en qué parte del ciclo de vida se encuentra el Pod. Sus posibles valores son:

- **Pending**: aceptado por el cluster de kubernetes para ser desplegado, pero todavía no está en ejecución
- **Running**: Está en ejecución.
- **Succeeded**: Su ejecución ha terminado satisfactoriamente.
- **Failed**: Todos los container han terminado y al menos uno de ellos ha fallado, ya sea por un error del contenedor o porque haya sido terminado por el sistema.
- **Unknown**: Cuando hay un error desconocido, normalmente por error de comunicación con el nodo donde se está ejecutando el pod.

Algunos comandos de kubectl pueden mostrar como fase *Terminating* si está en proceso de ser eliminado.

Estado de un contenedor

Al igual que con los Pod, Kubernetes hace un seguimiento del estado de cada contenedor en un Pod. Una vez el scheduler asigna un Pod a un Nodo, el kubelet empieza a crear los contenedores para ese Pod. Pueden tener tres estados:

- **Waiting**: Cuando el contenedor está ejecutando las operaciones necesarias para empezar, como, entre otras, descargar la imagen del registro o aplicar un Secret. Cuando se consulta el estado del Pod con un contenedor en este estado también da una razón.
- **Running**: El contenedor se está ejecutando sin problemas. Si había un hook *postStart* ya habrá sido ejecutado y con éxito.
- **Terminated**: Cuando un contenedor termina de ejecutarse por error o cualquier otra razón. Si se consulta el estado del Pod podremos ver la razón por la que ha ocurrido. Si tiene un hook *preStop* configurado, éste será ejecutado antes de pasar a este estado.

Políticas de reinicio de un container

Al definirse un Pod se puede indicar en el campo *restartPolicy* si queremos que se reinicie siempre (Always, por defecto), cuando haya un fallo (OnFailure) o nunca (Never). Esta política se aplica a todos los contenedores del Pod. Cuándo uno de los contenedores termina, kubelet intentará reiniciarlos con un back-off delay hasta un máximo de cinco minutos. Si el contenedor funciona durante 10 minutos sin problema, el retraso del backoff es restaurado.

Diagnósticos de un Container (Probes)

Un *probe* es un diagnóstico ejecutado periódicamente por kubelet en los contenedores y permiten saber el estado en el que se encuentran. Existen varios mecanismos:

- **exec**: Ejecuta un comando indicado por el desarrollador en el container. Si devuelve un código de estado igual a cero se considera exitoso.
- **grpc**: Ejecuta una llamada RPC, esté método debe devolver *SERVING.*
- ***httpGet****:* Realiza una llamada HTTP GET contra la IP del Pod en el puerto especificado. Cualquier respuesta con código HTTP mayor o igual que 200 y menor que 400 será considerada exitosa.
- **tcpSocket**: Realiza una conexión TCP contra un puerto, si éste está abierto se considera exitoso.

Un diagnóstico puede tener tres resultados:

- **Success**: El contenedor pasó el diagnóstico.
- **Failure**: El contenedor falló diagnóstico.
- **Unknown**: El diagnóstico falló durante la ejecución, no se tomará ninguna acción y kubelet intentará hacer más comprobaciones.

Hay tres tipos de diagnósticos:

- **livenessProbe**: Indica si el pod está en ejecución, si falla, kubelet destruirá el contenedor y se aplicará la politica de reinicio definida.
- **readinessProbe:** Indica cuando el contenedor está preparado para responder a peticiones. Mientras no sea exitoso los servicios no transmitirán el tráfico a este contenedor.
- **startupProbe**: Indica cuando la aplicación del contenedor ha empezado, el resto de diagnósticos son ignorados si se define este.

4.4. Workloads (Cargas de trabajo)

Una aplicación en ejecución sobre Kubernetes es llamada Workload. Ésta puede estar compuesta de un único componente o varios. Cada uno de estos componentes se ejecutan en pods**.** Si despliegas un Pod y éste falla es responsabilidad del desarrollador volverla a poner ejecución, lo mismo pasa si falla un Nodo, pero a mayor escala, ya que serán todos los Pod dentro de éste los que no se van a levantar. Por ésta razón no es habitual que se gestionen manualmente y se utilizan los recursos llamados Workload, que son controladores que se encargan de administrar estos Pod y asegurar la disponibilidad de los recursos enlazados de distinta manera según el tipo de Workload que se use y cómo se configure.

Kubernetes viene con una serie de workloads para ayudar a gestionar conjuntos de pod, tiene varios tipos predefinidos:

- **Deployment y ReplicaSet:** Permite desplegar aplicaciones stateless.
- **StatefulSet**: Como los deployment pero permite crear aplicaciones que almacenen estado
- **DaemonSet**: Permite gestionar conjuntos de pods y asegurar que se ejecute en todos los nodos, un uso habitual es recolección de logs.
- **Job y CronJob**: Permite definir, respectivamente, tareas a ejecutar una única vez o de manera periódica.

Vamos a profundizar un poco más en los Deployment y los StatefulSet ya que van a ser los utilizados en el caso de uso real de este trabajo.

4.4.1. Deployment

En este tipo de controlador de cargas de trabajo los pods son completamente intercambiables y no almacenan estado.

4.4.2. StatefulSet

A diferencia de los Deployment, éste controlador controla los identificadores de los pods, y mantendrá el identificador cuando los está reemplazando o reiniciando.

4.5. Almacenamiento

Los ficheros que se almacenan en el disco de un contenedor son efímeros. En una arquitectura de microservicios no necesitamos tener espacio en disco en todos nuestros microservicios, como por ejemplo en una caché en memoria o una API REST.

No obstante, en casi cualquier aplicación sí que habrá piezas que necesiten espacio en disco adicional, y se consigue montando volúmenes dentro de los contenedores. Se puede dividir los volúmenes en dos categorías:

- **Persistente**: El contenido se va a mantener tras un reinicio o crash de la aplicación. Útil para una base de datos.

- **Efímero**: El contenido se borra cuando el Pod termina su ejecución. Útil para escribir archivos temporales o ficheros de configuración.

Por ello necesitamos hablar de los sistemas que nos ofrece Kubernetes para dotar de almacenamiento.

4.5.1. Almacenamiento en Kubernetes

Para gestionar el almacenamiento existen tres tipos de objetos en Kubernetes:

PersistentVolume (PV)

Almacenamiento previamente aprovisionado por un administrador o de manera dinámica por un StorageClass. Es un recurso más en el cluster al igual que los nodos. Es la abstracción del almacenamiento físico, y por tanto, la representación de bajo nivel de un volumen de almacenamiento.

Su ciclo de vida es independiente de los pods, por tanto, sus datos siguen existiendo aunque el cluster tenga cambios sobre los pods en ejecución. Además, no tienen espacio de nombres, por lo tanto, son accesibles para todo el cluster.

Otra cualidad importante son los modos de acceso, existen los siguientes:

- **ReadWriteOnce(RWO):** Accesible en modo escritura-lectura por un único nodo.
- **ReadOnlyMany(ROX)**: Montar en modo de solo lectura por varios nodos.
- **ReadWriteMany(RWX)**: Varios nodos pueden escribir y leer al mismo tiempo.
- **ReadWriteOncePod(RWOP)**: Modo escritura-lectura para un único Pod.

PersistentVolumeClaim (PVC)

Es una petición de almacenamiento por parte de un desarrollador. Al crear un Pod, al igual que solicitamos CPU y memoria, con un PVC se puede solicitar espacio en disco, indicando la cantidad, el modo de escritura y lectura, y otros aspectos. Básicamente, es la manera de enlazar un PV a un Pod.

Un aspecto muy importante es que la relación entre PV y PVC es de uno a uno, es decir, una vez un PV sea asignado a un PVC, no podrá ser asociado a otro, por lo tanto hay que definir con cuidado los tamaños de los PV y las peticiones de los PVC porque si se tiene un PV con 10Gb y dos PVC de 5Gb, uno de ellos se quedará sin recursos.

StorageClass (SC)

Permite el aprovisionamiento dinámico de PVs.

Ciclo de vida

Los PVs son recursos en el cluster y los PVCs son peticiones de estos recursos. La interacción entre ellos sigue el siguiente ciclo de vida:

1. **Aprovisionamiento (Provisioning):** Puede ser de dos tipos:

a. Estático: El administrador crea una serie de PVs definiendo los detalles reales del almacenamiento.
b. Dinámico: Cuando la petición no coincide con ninguno de los PVs creados, Kubernetes intenta ofrecer de manera dinámica este espacio, para ello es necesario este tipo de peticiones creando previamente un StorageClass.

2. **Enlace (Binding)**: Es el proceso mediante el cual se asigna un PV al PVC.
3. **En uso (Using):** Cuando el Pod se ha unido a un PVC con un PV enlazado .
4. **Reclamo (Reclaiming)**: Cuando un PVC es eliminado, el PV sigue existiendo y pueden pasar dos cosas, que el contenido en disco se mantenga o elimine dependiendo de si se elige una política de reclamo de *retain* o *delete* respectivamente.

4.6. Servicios

Cada Pod en un cluster tiene su propia IP única a nivel de cluster.

4.6.1. ClusterIP

Este tipo de servicio es el por defecto, y permite asignar una dirección IP interna a tu servicio dentro del cluster, permitiendo así, la comunicación entre los distintos servicios. Tenemos la opción de fijar la IP en el fichero de configuración, pero es opcional, si no lo haces se asignará automáticamente y será considerado un servicio headless. Éste tipo de servicio no será visible fuera del cluster.

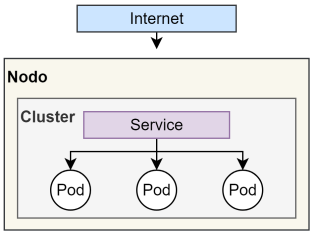

Figura 10. K8s Cluster IP Service.

4.6.2. NodePort

Para exponer un servicio a la red pública WAN podemos usar el tipo NodePort que abre un puerto en el nodo redirigiendo todo el tráfico que le llegue a nuestro servicio. Solo se puede elegir un puerto entre 30000 y 32767.

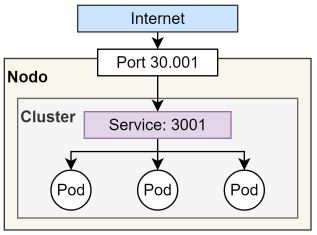

Figura 11. K8s NodePort Service.

4.6.3. LoadBalancer

Un LoadBalancer en Kubernetes expone el servicio a la red pública utilizando un balanceador de carga del proveedor cloud en el que esté alojado. Esta solución puede elevar el coste ya que cada servicio expuesto tendrá su propia IP.

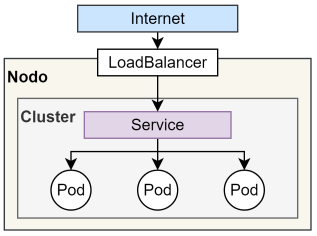

Figura 12. K8s LoadBalancer Service.

4.6.4 Ingress

Un Ingress no es un tipo de servicio, es una pieza que se sitúa entre los servicios y la red pública. Se utiliza con varios fines como añadir certificados SSL, Authentication o exponer varios servicios bajo la misma IP, permitiendo definir subdominios.

5. Buenas prácticas y metodologías ágiles

5.1. Arquitectura de software

Antes de continuar necesitamos hablar sobre la arquitectura de software. Podemos definir la arquitectura de software como el diseño de más alto nivel de un sistema, definiendo sus componentes, relaciones y comunicaciones entre ellos.

Los microservicios son una arquitectura, pero de muy alto nivel, en el que cada servicio es un componente que se intercomunican entre sí formando un sistema. Pero, ¿qué arquitectura usamos dentro de cada uno de esos microservicios?

Arquitectura hexagonal

También conocida como arquitectura de adaptadores y puertos, tiene como propósito separar y desacoplar la aplicación en distintas capas con su propia responsabilidad, permitiendo así que evolucionen de forma aislada. Todos los puntos de entrada y salida (HTTP, RPC, Broker, BBDD…) son los puertos de la aplicación, definidos con interfaces, y los adaptadores son implementaciones concretas de esos puertos.

Siguiendo esta arquitectura aplicamos el principio de inversión de dependencias (DIP) y nos permite el poder cambiar elementos ajenos a nuestra lógica de negocio sin afectar a esta.

Al tener todas las capas desacopladas podemos mantener y reutilizar mejor estas piezas. Otra ventaja muy importante, efecto colateral del DIP es que mejora la testabilidad ya que podemos reemplazar componentes reales como una base de datos por un mock.

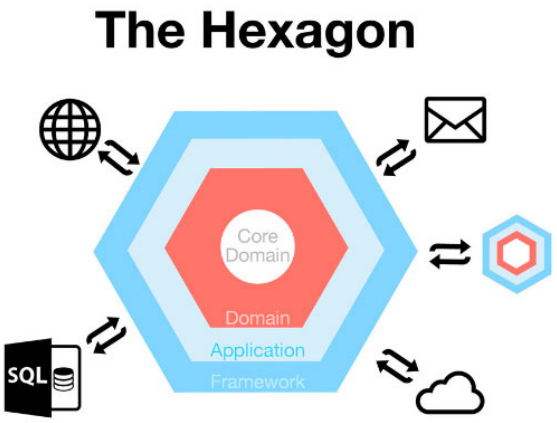

Figura 13. Arquitectura Hexagonal.

Esta arquitectura fomenta que nuestra lógica de negocio o dominio sea el núcleo de la aplicación encaja a la perfección con la idea de Domain Driven Design (DDD).

5.2. Test Drive Development (TDD)

Es la metodología de desarrollo que consiste en escribir los test antes de escribir el código y que los desarrolladores escriban código haciendo pasar las pruebas diseñadas. Es una práctica muy extendida, aunque por el esfuerzo que requiere de planificación, la mayoría de las empresas tienden a escribir los test sobre la marcha por parte de los desarrolladores, teniendo una etapa de QA en la que cuando la nueva funcionalidad está lista se encargará de escribir tests de integración que comprueben esta funcionalidad, y que a posteriori sirvan como tests de regresión, asegurando que nuevos cambios no afecten a implementaciones pasadas.

Ésta metodología es de vital importancia desarrollando en entornos de CI/CD debido a que al tener los procesos de integración y despliegue automatizados son más propensos a que se introduzcan errores.

5.2. CI / CD

5.2.1. Qué es CI/CD

Integración contínua (CI)

Es una estrategia de desarrollo de software que incrementa la velocidad de desarrollo manteniendo la calidad del código que los equipos despliegan. Los desarrolladores harán commits de pequeñas piezas de código diariamente y será automáticamente construido y testeado antes de ser llevado al repositorio. Si una de esas etapas falla, el desarrollador no podrá publicar sus cambios.

Despliegue contínua (CI)

Es una disciplina de desarrollo de software en la que el software puede ser desplegado a producción en cualquier momento. Los beneficios principales de esta disciplina son:

- **Riesgo reducido en los despliegues**: Cómo se están haciendo despliegues constantemente con pequeños cambios, es más difícil que algo vaya mal y es más fácil de solucionar.
- **Progreso fiable:** Muchas veces, al hacer el seguimiento de un proyecto buscamos que ciertas features estén hechas, y es más fácil si esos cambios realizados se ven cuanto antes.
- **Feedback de usuario**: Cuánto antes lleguen las funcionalidades nuevas a producción, antes se podrá recibir feedback de los usuarios.

6. Caso de uso real: E Commerce

6.1. Visión general

Los microservicios estarán escritos en NodeJS, con una arquitectura hexagonal, con base de datos individual y un broker RabbitMQ en común para comunicarse entre ellos. Además, tendrán sus propias API Rest para poder consultar información entre ellas si fuese necesario.

A continuación se describen brevemente los microservicios necesarios en esta aplicación y sus características:

6.1.1. Requisitos

Para el desarrollo de este proyecto de prueba se va a tener en cuenta los siguientes requisitos funcionales y no funcionales para definir las necesidades y el alcance del proyecto. Habrán dos tipos de usuarios en la aplicación:

- Cliente: usuario que va a realizar pedidos sobre nuestra aplicación.
- Administrador: operador del almacén que se encarga de la preparación del pedido y podrá hacer cambios sobre el estado del pedido.

Requisitos funcionales (RF)

Los requisitos funcionales son las declaraciones de los servicios que va a ofrecer nuestra aplicación, no solo a usuarios que interaccionan con ella sino también a los diferentes microservicios que la componen. Se podría resumir con la pregunta: ¿Qué va a hacer la aplicación?.

#	Descripción
RF1	El cliente debe poder crear pedidos.
RF2	El cliente tiene que estar al tanto de las actualizaciones sobre su pedido.
RF3	El cliente no será cobrado hasta que esté confirmado el pedido, entendiéndose como confirmado que el stock de su producto está disponible.
RF4	El cliente puede cancelar su pedido mientras no haya sido recogido por el transportista.
RF5	No se pedirá la recogida de un paquete hasta que no se haya completado satisfactoriamente el pago.

Requisitos no funcionales (RNF)

Al igual que los requisitos funcionales definen **el qué** va a hacer un sistema Los requisitos no funcionales hablan de **cómo** va a hacer el sistema.

#	Descripción
RNF1	El backend estará hecho en NodeJS.
RNF2	Los desarrollos deben ser dotados de portabilidad.
RNF3	Los servicios deben escalar horizontalmente bajo demanda.
RNF4	La aplicación va a ser dividida en diferentes microservicios para llevar a cabo sus mantenimientos por distintos equipos y facilitar su escalado.
RNF5	Solo quedarán expuestos los servicios mínima y exclusivamente necesarios para la operatividad de los usuarios finales con la aplicación.
RNF6	El cliente tiene que verse afectado lo menos posible por la caída o sobrecarga de servicios internos y externos.
RNF7	Cada microservicio tendrá su propia base de datos.

6.1.2. Organización del proyecto

El proyecto se organiza en una estructura monorepo, que consiste en un único repositorio con distintos paquetes, uno por servicio más uno llamado shared que tiene utilidades varías usadas para los adaptadores HTTP y RabbitMQ, y algunas clases base utilizadas para las entidades. Se utiliza la librería Lerna, que tiene muchas utilidades para gestionar monorepos. En este caso se va a utilizar solo para enlazar la dependencia del paquete *shared* con el resto, y ejecutar el comando de compilación en todos a la vez.

6.1.3. Diseño de alto nivel

La aplicación va a estar dividida en cuatro microservicios que van a permitir la especialización de los equipos que la mantienen y escalamiento (RNF4). Son los siguientes:

- **GatewayService:** Una API con el patrón CQRS que se utilizará para construir las snapshots de los pedidos en base a los eventos detectados. Este gateway también será el encargado de recibir llamadas HTTP desde fuera para enviar los comandos a los microservicios.
- **OrderService**: Almacena todos los datos de un pedido.
- **PaymentService**: Procesa los pagos y notifica al resto de sus actualizaciones.
- **ShippingService**: Gestiona los envíos de los pedidos y notifica de las notificaciones de su estado.

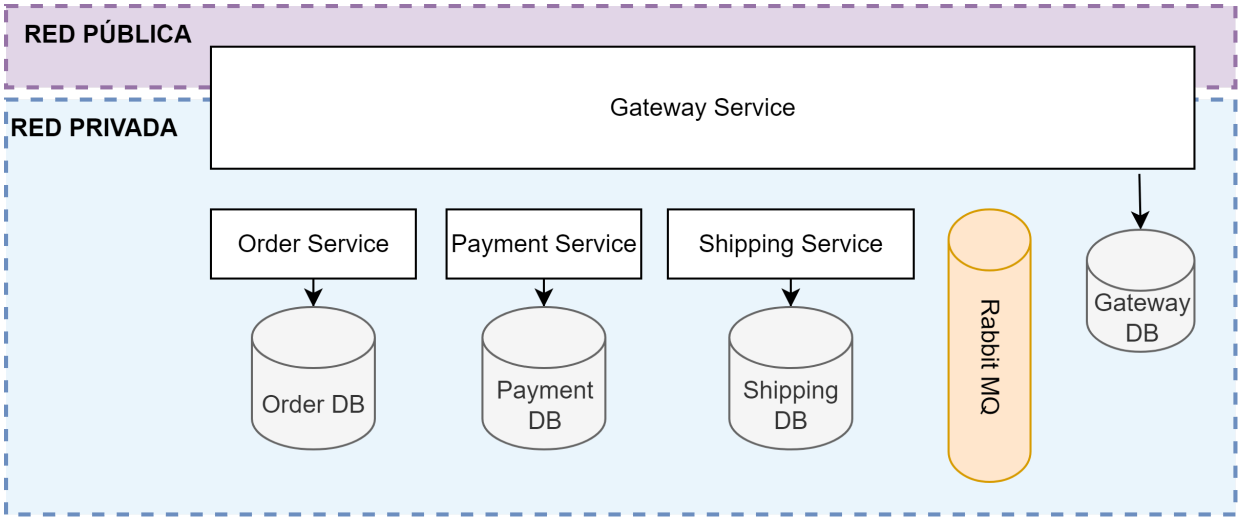

Figura 14. Diagrama de alto nivel de los microservicios, base de datos, broker y redes.

La intercomunicación se hará de forma síncrona mediante protocolo HTTP y de manera asíncrona mediante protocolo AMQP con el broker de mensajería RabbitMQ. La comunicación asíncrona es especialmente necesaria principalmente por las siguientes razones:

- No perder información de las peticiones si hay alguna caída del sistema y poder hacer encolamiento de estas.
- No bloquear a los usuarios ni operadores mientras ocurre toda la orquestación de las llamadas o tener que repetirlas en caso de que éstas fallen.
- Desacoplamiento de otros microservicios ya que al usar una cola, cada microservicio se encargará de hacer eventos sobre la actualización de los pedidos, permitiendo que los microservicios interesados se suscriban a estos cambios.

La infraestructura se va a montar sobre Kubernetes para dotar a la aplicación de escalabilidad horizontal según demanda (RNF3) y todos los microservicios quedarán ocultos en una red privada a la que sólo tendrá acceso el Gateway que estará en la red pública, exponiendo así nuestra aplicación lo mínimo al exterior (RNF5).

Las aplicaciones van a ser desarrolladas y desplegadas con Docker para facilitar su portabilidad y sus despliegues (RNF2).

6.2. Orquestación mediante Saga

En la aplicación hay tres microservicios que albergan lógica de negocio con sus respectivas transacciones a nivel local. Para poder realizar esta segmentación hay que aplicar la metodología de mensajería transaccional conocida como saga que se ha explicado previamente (Apartado 2.5).

Aunque la aplicación es relativamente simple, se decide usar la coordinación por orquestación ya que trae ventajas de cara a tener todo el flujo definido en un único sitio, facilitando la extensión. Por simplicidad, se ha elegido el OrderService, como orquestador además de actuar como participante ya que es el núcleo de la aplicación. Hay dos casos de uso: Crear pedido y cancelar pedido.

6.2.1 Crear Pedido

Tipo	#	Servicio	Transacción	Tx de Compensación
Compensable	1	OrderService	createOrder()	cancelOrder()
	2	PaymentService	chargePayment()	refundPayment()
Pivot	3	ShippingService	sendShipment()	
Retriable	4	ShippingService	markAsDelivered()	
	5	OrderService	markAsCompleted()	

La transacción `sendShipment()` es seleccionada como transacción pivote ya que a partir de ésta se comunica con un proveedor de logística para que se encargue de recoger el pedido en los almacenes y lo envíen, desde mi experiencia profesional, si es posible cancelar un pedido en este punto, pero en muchas ocasiones requiere una gestión por parte del personal y es preferible mantenerlo así, por tanto, si el usuario intenta cancelar el pedido pasado este punto se rechaza. En cambio, el realizar un reembolso del dinero no es tan problemático porque no implica una gestión por una persona física. El flujo es el siguiente:

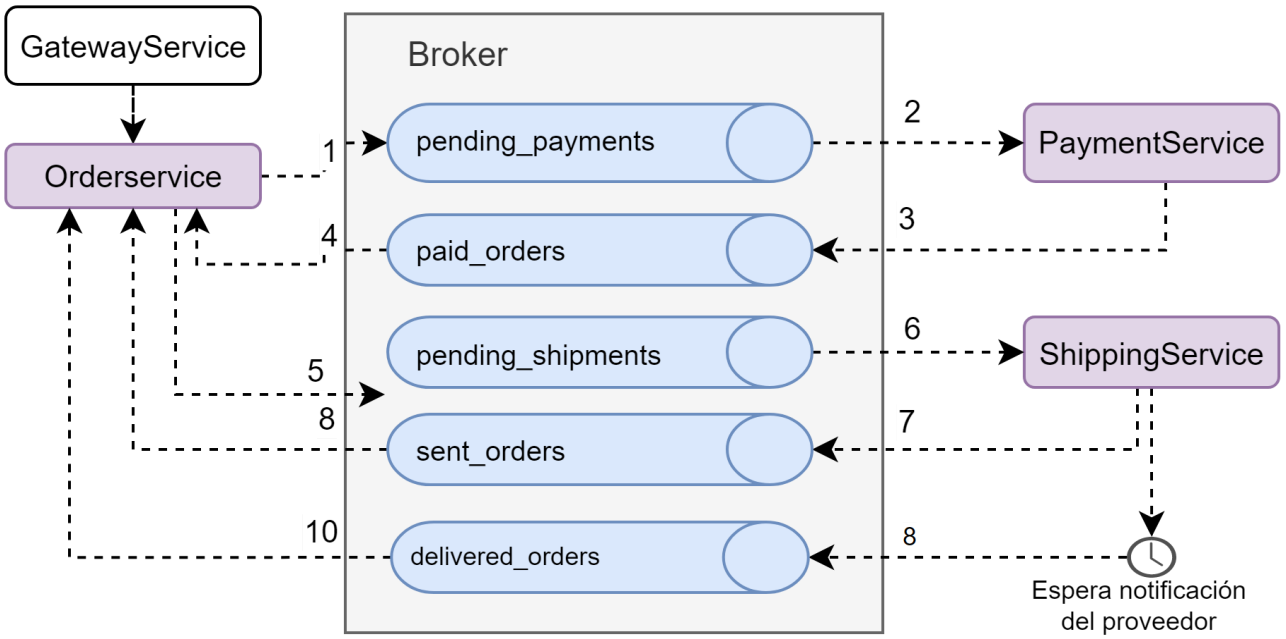

Figura 15. Saga crear pedido.

Si

6.2.1 Devolver Pedido

A la hora de devolver un pedido el saga es más sencillo, según el usuario ordene la devolución la primera transacción es pivotable ya que en el instante que se ordene se va a procesar a la recogida del paquete por parte de un transportista y no se va a poder revertir.

Tipo	#	Servicio	Transacción	Tx de Compensación
Pivot	1	OrderService	returnOrder()	
Retriable	2	ShippingService	sendPickUp()	
	3	ShippingService	markAsReceived()	
	4	OrderService	markAsReturned()	

6.3. API Gateway y CQRS

Al tener varios microservicios, la consulta del estado de las distintas entidades de negocio, se complica, y también se añaden más vectores de ataque si queremos exponer estas apis de manera pública, por lo tanto se ha decidido hacer un API Gateway, que es la única pieza expuesta al público.

Éste microservicio tiene su propia base de datos con el propósito de optimizar las operaciones de lectura. Se encarga de suscribirse a todos los eventos que emite el orquestador, OrderService, a través de su exchange Order, de esa manera, el Gateway se entera de todos los cambios de estado y hace una consulta a cada microservicio para guardar una snapshot de todos los datos de manera conjunta.

Para las operaciones de escritura, que son crear pedido y devolver, utiliza comunicación síncrona por HTTP enviando una petición al OrderService.

6.4. Mensajería

Para comunicar los microservicios se va a utilizar un broker llamado RabbitMQ, un broker de mensajería open source que implementa el protocolo AMQP (Advanced Messaging Queue Protocol). Éste broker tiene cuatro entidades fundamentales:

6.4.1. RabbitMQ

Exchange

Utilizado por los productores para publicar mensajes. En RabbitMQ no se publica directamente en colas, hace falta configurar a que colas tienen que dirigirse los mensajes, si no se configura nada un productor publicará mensajes que, simplemente, no serán consumidos. Hay varios tipos de exchange:

- **Fanout**: Envía en paralelo una copia de cada mensaje a todas las colas enlazadas.
- **Direct**: Envía en paralelo una copia de cada mensaje a las colas enlazadas que tengan el mismo routing key.
- **Topic**: Funciona igual que Direct, pero permite hacer wildcards en las routing keys.
- **Header**: Permite enviar cabeceras, al igual que en el protocolo HTTP, para hacer reglas más complejas de enrutamiento.

Colas

Utilizado por los consumidores para suscribirse a los mensajes. Hay dos tipos según su durabilidad:

- **Persistente**: Se guarda el mensaje en disco, permite recuperar esta información en un reinicio.
- **Transient**: Se almacenan los mensajes en memoria.

Routing Keys

Sirve para categorizar los mensajes y que los exchanges sepan a donde tienen que transmitirlos. Se definen con hasta 255 caracteres de palabras separadas por puntos.

Bindings

Existen los exchanges y las colas. Para que las colas reciban mensajes se deben enlazar.

6.4.2. PubSub

PubSub es un patrón de mensajería que abstrae al productor de mensajes de los consumidores. El consumidor elige los mensajes que resultan de su interés y así el productor no necesita saber de ellos y escalar con mayor facilidad.

Cuando se realiza una orquestación de microservicios existe cierto acoplamiento aunque el productor se abstraiga, ya que puede que se quede a la espera de que un consumidor lea un mensaje y responda con otro, como puede ser en el caso del OrderService, esperando a que el PaymentService emita un mensaje de pago correcto.

En cambio, en otros muchos casos nos puede permitir ampliar las funcionalidades en servicios independientes, como puede ser un recolector de métricas o el propio GatewayService, que en base a estos eventos irá actualizando la snapshot de su base de datos.

6.4.3. Topología

Por cada microservicio habrá un exchange en el que cada uno de ellos publicarán sus propios mensajes. Cómo se ha mencionado, para conectar exchanges y colas hay que definir las routing keys. Cómo se publica en base a eventos, los routing keys tendrán la siguiente estructura: `{entity_name}.evt.{event_name}`

Exchange	Routing Key	Queue	Consumer
Order	order.evt.created	pending_payments	PaymentService
	order.evt.paid	pending_shipments	ShippingService
	order.evt.*	all_order_events	GatewayService
Payment	payment.evt.charged	paid_orders	OrderService
Shipping	shipment.evt.sent	sent_orders	OrderService
	shipment.evt.delivered	delivered_orders	OrderService

En la siguiente página se muestra el diagrama mostrando esta configuración completa.

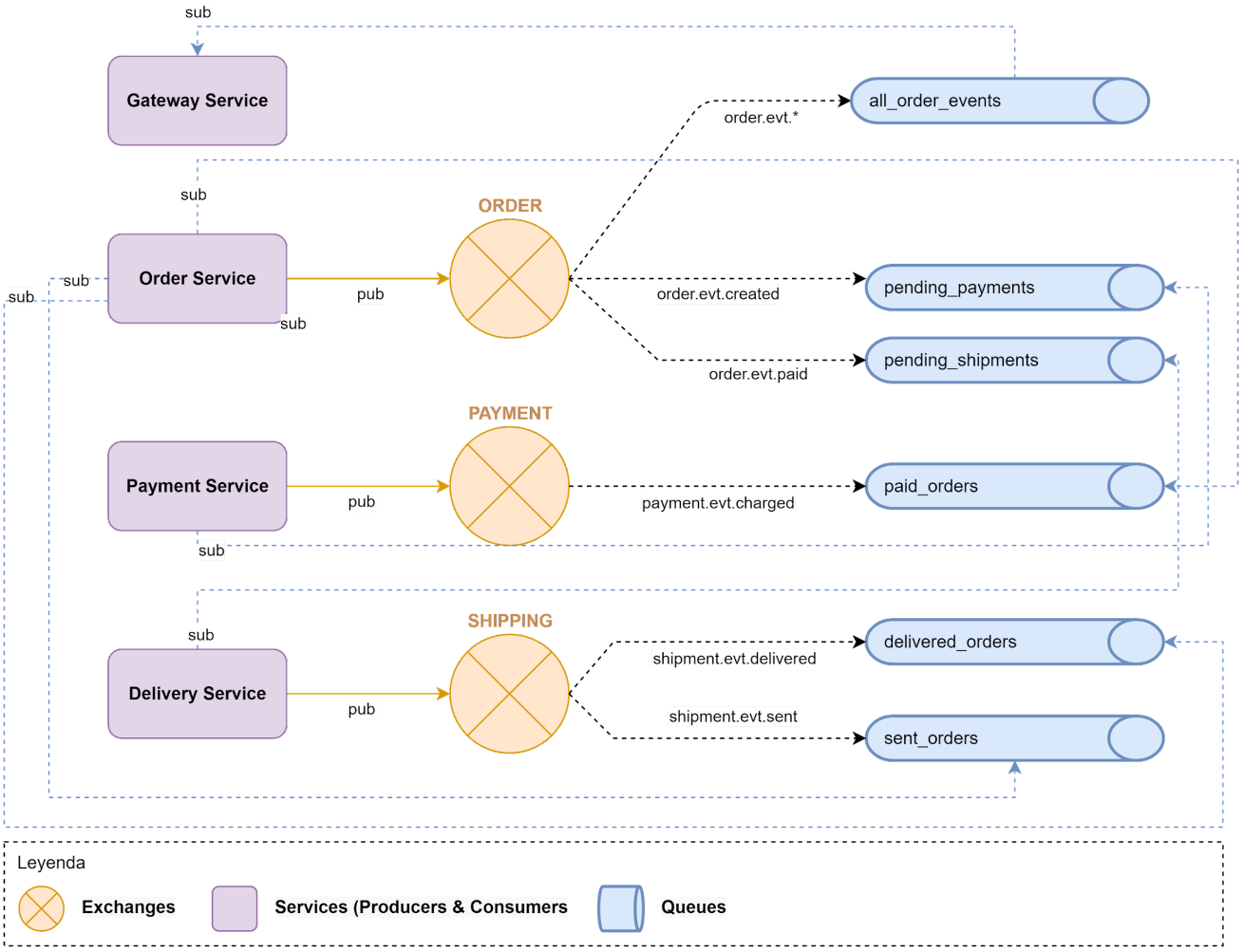

Figura 16. Diagrama que muestra los Productores/Consumidores y los flujos de la mensajería.

6.5. Creando contenedores de Docker

En el desarrollo de la aplicación hay dos usos principales a la hora de crear los contenedores de docker para nuestras aplicaciones. Cómo ya se ha mencionado, la principal funcionalidad es la portabilidad, pero las necesidades que se tiene al estar en local desarrollando la aplicación y cuando está en producción son distintas, por lo tanto, se va a hablar de los contenedores en dos entornos: Local y Producción

6.5.1. Dockerfile Local

Al igual que se quiere portar la aplicación en producción, es recomendable que cuando un desarrollador se una al equipo pierda el menor tiempo posible configurando su máquina y que el entorno sea lo más parecido a producción.

Para facilitar el desarrollo en local se puede usar *nodemon,* una herramienta que detecta los cambios de ficheros en disco y relanza la aplicación automáticamente. Esto se puede conseguir montando en el contenedor la ubicación de los ficheros como volumen, así, cuando se cambie un fichero en la máquina local del desarrollador, se refrescarán en el contenedor de docker.

Dockerfile.local
```
FROM node:16-alpine3.14 AS appbuild
WORKDIR /app
COPY . .

RUN yarn
RUN yarn bootstrap
RUN yarn build:all

WORKDIR /app/packages/order-service
CMD yarn start:dev:watch
```

En el fichero copiamos todo el proyecto, ya que al utilizar un monorepo necesitamos acceso desde todos los paquetes al shared.

Posteriormente se ejecuta el comando de yarn para instalar dependencias y yarn bootstrap para que, lerna, enlace la dependencia *shared* a cada microservicio.

A continuación se ejecuta yarn build:all para construir todo el proyecto.

Finalmente ejecutamos yarn start:dev:watch que ejecuta el siguiente comando.

package.json > scripts > start:dev:watch
```
nodemon --exec yarn node -r ts-node/register src/index.ts
```

Cada vez que cambie un fichero con extensión typescript dentro del directorio *src* del servicio en cuestión o el paquete *shared*, volverá a lanzar la aplicación.

La configuración de nodemon viene dada en el fichero nodemon.json.

nodemon.json
```
{
 "watch": ["src/**", "../shared/**"],
 "ext": "ts,json"
}
```

6.5.3. Dockerfile Producción

Al igual que en la etapa de desarrollo, en la imagen de Docker de producción se hace una compilación de todo el proyecto, pero añadimos la construcción multi etapa de Docker para en una segunda etapa copiar solo los ficheros necesarios, que en este caso son:

- Dependencias globales del proyecto: ./node_modules
- Paquete shared: ./packages/shared
- Paquete del servicio en cuestión: ./packages/{order|payment|shipping}-service

```
FROM node:16-alpine3.14 AS appbuild

WORKDIR /usr/src/app

COPY . .

RUN yarn
RUN yarn bootstrap
RUN yarn build:all

FROM node:16-alpine3.14

WORKDIR /usr/src/app
COPY --from=appbuild /usr/src/app/packages/order-service
./packages/order-service
COPY --from=appbuild /usr/src/app/packages/shared ./packages/shared
COPY --from=appbuild /usr/src/app/node_modules ./node_modules

WORKDIR /usr/src/app/packages/order-service

CMD yarn start
```

6.5.4. Publicación de imágenes en el registro

Para publicar las imágenes se puede hacer uso del registro público gratuito de Docker Hub (hub.docker.com).

6.6. Despliegue en Kubernetes

Para el despliegue de la aplicación vamos a usar Kubernetes con Google Kubernetes Engine (GKE) disponible en la plataforma de servicios en la nube Google Cloud Platform (GCP).

Al definir a continuación los Deployment se obvia por brevedad los recursos asignados a cada servicio.

6.6.1. APIs

Cómo los servicios OrderService, PaymentService y el ShippingService son similares en infraestructura podemos agruparlos en un único apartado.

Carga de trabajo

Para estas aplicaciones que son API REST, no necesitamos estado, así que vamos a usar una carga de trabajo de tipo Deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: order-service
 labels:
   app: order-service
spec:
 replicas: 1
 selector:
   matchLabels:
     app: order-service
 template:
   metadata:
     labels:
       app: order-service
   spec:
     containers:
     - name: order-service
       image: $IMAGE
       env:
       - name: APP_NAME
         valueFrom:
           configMapKeyRef:
             name: order-service
             key: app_name
       - name: BROKER_URI
         valueFrom:
```

```
          secretKeyRef:
            name: rabbitmq
            key: broker_uri
      ports:
      - containerPort: 3000
```

Servicio

Se necesita un servicio ya que debe ser accesible para el resto de cargas de trabajo para realizar sus conexiones asíncronas. Además, tiene que ser accesible al público para poder acceder al panel de administración, por lo tanto, será de tipo *LoadBalancer,* en lugar de ClusterIP cómo es en el caso de las APIs.

```
apiVersion: v1
kind: Service
metadata:
 name: order-service
 labels:
   app: order-service
spec:
 type: ClusterIP
 ports:
 - port: 3001
   targetPort: 3000
   protocol: TCP
 selector:
   app: order-service
```

6.6.2. Gateway

Carga de trabajo

Es una API REST, no necesitamos estado, así que vamos a usar una carga de trabajo de tipo Deployment. La definición es igual que la de las APIs en el apartado anterior.

ConfigMaps

El config map de este servicio trae las direcciones del resto de servicios ya que lo necesita para coger los datos y realizar las snapshots.

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: gateway-service
data:
 app_name: gateway-service
 order_svc_url: "http://order-service:3001"
 payment_svc_url: "http://payment-service:3002"
 shipping_svc_url: "http://shipping-service:3003"
```

Servicio

Necesitamos un servicio de tipo LoadBalancer ya que esta va a ser la capa accesible al público.

```
apiVersion: v1
kind: Service
metadata:
 name: gateway-service
 labels:
   app: gateway-service
spec:
 type: LoadBalancer
 ports:
 - port: 3000
   targetPort: 3000
   protocol: TCP
 selector:
   app: gateway-service
```

6.6.3. Broker RabbitMQ

Secretos

Esta aplicación necesita el uso de secretos para almacenar de manera segura el usuario y contraseña para acceder al panel de administración. Este secreto se va a crear para la configuración de credenciales en el arranque y para compartir las credenciales con los distintos microservicios que se van a conectar. Deben ser aplicados en el cluster antes de el resto de despliegues ya que se van a usar en los Deployment de cada servicio para recoger estas credenciales.

En el fichero se definen con variables que son reemplazadas durante el Pipeline, ya que no es seguro almacenar secretos en GitHub y que cualquier usuario pueda verlos. Por esa razón se usa la herramienta de Secretos que tiene las GitHub Actions, que permiten inyectar como variables de entorno durante el pipeline para alimentar este fichero.

```
apiVersion: v1
kind: Secret
metadata:
 name: rabbitmq
type: Opaque
data:
 broker_uri: $BROKER_URI
 admin_user: $BROKER_USER
 admin_pass: $BROKER_PASS
```

Servicio

Se necesita un servicio ya que debe ser accesible para el resto de cargas de trabajo para realizar sus conexiones asíncronas. Además, tiene que ser accesible al público para poder acceder al panel de administración, por lo tanto, será de tipo *LoadBalancer,* en lugar de ClusterIP cómo es en el caso de las APIs, aunque lo ideal sería crear una VPN y no dejarlo expuesto al público.

```
apiVersion: v1
kind: Service
metadata:
 name: rabbitmq
 labels:
   app: rabbitmq
spec:
 type: LoadBalancer
 ports:
 - name: amqp
   port: 5672
   protocol: TCP
```

```
  - name: http
    port: 15672
    protocol: TCP
  selector:
    app: rabbitmq
```

Carga de trabajo

Para desplegar RabbitMQ se va a utilizar un Workload de tipo StatefulSet, que están preparados para aplicaciones que necesitan guardar el estado de la aplicación, de esta manera, podemos añadir persistencia para que cuando se desplieguen cambios no se pierda la configuración ni los mensajes de la cola. Por tanto, necesitamos montar un volumen, de lo contrario, todo quedará almacenado en el Pod y no habría persistencia.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rabbitmq
  labels:
    app: rabbitmq
spec:
  serviceName: rabbitmq
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      volumes:
      - name: rabbitmq-data
        persistentVolumeClaim:
          claimName: rabbitmq-data
      containers:
      - name: rabbitmq
        image: rabbitmq:3-management-alpine
        env:
          - name: RABBITMQ_USER
            valueFrom:
              secretKeyRef:
                name: rabbitmq
                key: admin_user
          - name: RABBITMQ_PASSWORD
            valueFrom:
              secretKeyRef:
```

```
            name: rabbitmq
            key: admin_pass
      ports:
      - containerPort: 5672
      - containerPort: 15672
      volumeMounts:
      - name: rabbitmq-data
        mountPath: "/var/lib/rabbitmq"
```

6.7. CI/CD

Para hacer todos los despliegues arriba mencionados entra en juego las pipelines de Integración Continua (CI) y Despliegue Continuo (CD).

Cómo estamos almacenando el código en GitHub, vamos a aprovechar sus Github Actions, que nos permiten definir en ficheros YAML una serie de trabajos que nos permiten automatizar las pipeline en base a los commits que realizamos.

6.7.1. Visión general del Pipeline

En el pipeline se realizan principalmente dos tareas: Construir y Desplegar.

Construir y publicar

En esta etapa se construyen las imágenes de Docker, usando el fichero *Dockerfile.prod* teniendo tres etapas:

1. **Construcción:** Se ejecuta el comando que instala las dependencias y hace el build del proyecto
2. **Test:** Se ejecutan los test de la aplicación, fallando la construcción en caso de que no pasen todos.
3. **Empaquetado:** Se recogen los ficheros necesarios para ejecutar la aplicación para eliminar ficheros innecesarios y ocupar menos espacio en el registro de Docker.

Una vez se han construido las imágenes se pueden publicar en **DockerHub**, utilizando como versión en el tag de la imagen el hash del commit que hizo el trigger de la pipeline en GitHub.

Desplegar

Una vez están las imágenes publicadas en **DockerHub** se procede a desplegar en Kubernetes. Todos los pasos de despliegue dependen de la construcción de sus respectivas imágenes, y del despliegue de RabbitMQ, ya que todos dependen de éste, no sólo a nivel de aplicación, sino de infraestructura ya que usan sus secretos. En la figura 6.1 se muestra el diagrama del pipeline.

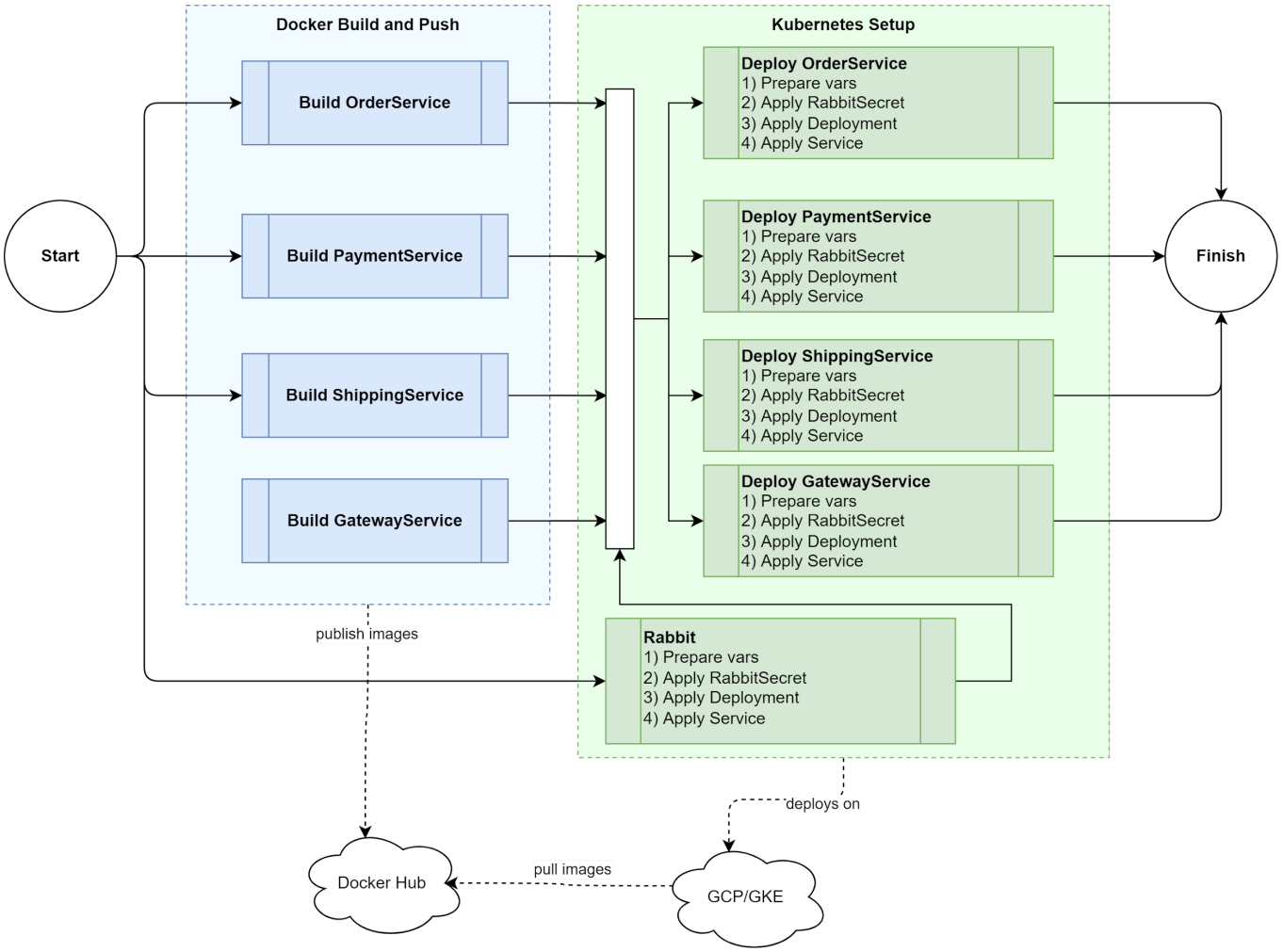

Figura 17. Pipeline de Github.

6.7.2. GitHub Actions.

Las GitHub Actions, permiten definir unas pipeline que se ejecutan en base a eventos de git como un commit o una pull request en unos ficheros YAML dentro de la carpeta *.github/workflows* en el raíz del repositorio.

Cada fichero que se encuentre en esa carpeta será procesado por GitHub y tratará de lanzar la pipeline definida dentro en caso de ser válida. La estructura de estos ficheros es:

- **on:**
- **env:**
- **jobs:**

En los pipelines que se han creado para este proyecto, cualquier cambio en el repositorio hace disparar la construcción y el despliegue de todas las apps en conjunto. Por supuesto, esto se puede optimizar decidiendo que pipelines se pueden saltar en base a los ficheros que se han ejecutado, aunque si se tienen recursos se puede mantener así para tener más confianza.

Para reutilizar pasos recurrentes se han creado plantillas para la etapa de Build & Push de Docker y el despliegue en Kubernetes.

Tareas Reutilizables

Pipeline Docker

Al definir en el trigger *workflow_call*, indica a GitHub que este snippet no se ejecuta por si solo. Las variables de entorno son alimentadas a través del input de la plantilla, indicando el nombre y el tag de la imagen, así como la ruta al fichero de Docker.

En el Primer paso se preparan las credenciales de Docker y posteriormente se hace el Build y el Push.

```
on:
 workflow_call:
   inputs:
     app_name:
       required: true
       type: string
env:
 APP_NAME: ${{ inputs.app_name }}
 # global
 IMAGE: sergiazow/${{ inputs.app_name }}
 IMAGE_TAG: sergiazow/${{ inputs.app_name }}:${{ github.sha }}
 DOCKERFILE_PATH: ./packages/${{ inputs.app_name }}/Dockerfile.prod
 jobs:
 docker:
   name: Docker
   runs-on: ubuntu-latest
   steps:
     - name: Set up QEMU
       uses: docker/setup-qemu-action@v2
     - name: Set up Docker Buildx
       uses: docker/setup-buildx-action@v2
     - name: Login to DockerHub
       uses: docker/login-action@v2
       with:
         username: ${{ secrets.DOCKER_USER }}
         password: ${{ secrets.DOCKER_TOKEN }}
     - name: Build and push
       uses: docker/build-push-action@v3
       with:
         push: true
         tags: ${{env.IMAGE}}:latest,${{env.IMAGE_TAG}}
         file: ${{env.DOCKERFILE_PATH}}
```

Despliegue en Kubernetes

En esta etapa se configuran los credenciales de Kubernetes al igual que se hace en la tarea de Docker. Cómo los job de github funcionan en paralelo y la única dependencia que hay entre todos son los secretos de RabbitMQ, se decide meter aquí para que se asegure antes de desplegar cualquier aplicación que esos secretos están actualizados. Las etapas son las siguientes:

1. Setup de credenciales de K8S
2. Aplicar secretos de RabbitMQ
3. Aplicar el ConfigMap
4. Aplicar el Deployment
5. Aplicar el Service

Cómo se puede ver, al igual que en la tarea de Docker, esta tarea está parametrizada para poder configurar por los input distintos valores para encontrar los ficheros de Kubernetes que se desean aplicar.

```
on:
 workflow_call:
   inputs:
     app_name:
       required: true
       type: string
env:
 APP_NAME: ${{ inputs.app_name }}
 IMAGE_TAG: sergiazow/${{ inputs.app_name }}:${{ github.sha }}
 # k8s:config
 PROJECT_ID: ${{ secrets.GKE_PROJECT }}
 GKE_CLUSTER: tfg-cluster        # Add your cluster name here.
 GKE_ZONE: europe-north1-a      # Add your cluster zone here.
 # k8s:secrets
 BROKER_USER: ${{ secrets.BROKER_USER }}
 BROKER_PASS: ${{ secrets.BROKER_PASS }}
 BROKER_URI: ${{ secrets.BROKER_URI }}

 jobs:
 k8s_deploy:
   name: Deploy to K8S
   runs-on: ubuntu-latest
   environment: production
   permissions:
     contents: 'read'
     id-token: 'write'

   steps:
   - name: Checkout
     uses: actions/checkout@v3

   - uses: google-github-actions/setup-gcloud@94337306dda8180d967a56932ceb4ddcf01edae7
     with:
       service_account_key: ${{ secrets.GKE_SA_KEY }}
```

```
      project_id: ${{ secrets.GKE_PROJECT }}
   # Get  the GKE credentials so we can deploy to the cluster
   - uses:
google-github-actions/get-gke-credentials@fb08709ba27618c31c09e014e1d8364b02e5042e
     with:
       cluster_name: ${{ env.GKE_CLUSTER }}
       location: ${{ env.GKE_ZONE }}
       credentials: ${{ secrets.GKE_SA_KEY }}
   # Global kubectl applies
   - name: Apply Secrets
     run: |-
       FILE=infra/k8s/secrets/rabbitmq.secret.yaml
       bash ./infra/k8s/scripts/set-var.sh $FILE BROKER_USER=$BROKER_USER
BROKER_PASS=$BROKER_PASS BROKER_URI=$BROKER_URI
       kubectl apply -f $FILE
   # Deploy the Docker image to the GKE cluster
   - name: Apply ConfigMap
     run: kubectl apply -f infra/k8s/apps/$APP_NAME/$APP_NAME.config-map.yaml
   - name: Apply Deployment
     run: |-
       FILE=infra/k8s/apps/$APP_NAME/$APP_NAME.deployment.yaml
       bash ./infra/k8s/scripts/set-var.sh $FILE IMAGE=$IMAGE_TAG
       kubectl apply -f $FILE
   - name: Apply Service
     run: |-
       FILE=infra/k8s/apps/$APP_NAME/$APP_NAME.service.yaml
       if test -f "$FILE"; then
         kubectl apply -f $FILE
       fi
   - name: Get Deployed services
     run: kubectl get services -o wide
```

Pipeline final

Por brevedad se muestra a continuación como queda el pipeline para el OrderService, ya que es el mismo para todos, por supuesto, cambiando sus parámetros. A excepción de RabbitMQ, que es una pipeline distinta en la que se obvia la construcción de la imagen de Docker al utilizar la imagen por defecto oficial.

```
name: "[Order] Build & Deploy"

on:
 push:
   branches:
     - "master"
 jobs:
 build:
   uses: ./.github/workflows/docker_build_n_push.yml
```

```
  with:
    app_name: order-service
  secrets: inherit


deploy:
  needs: [build]
  uses: ./.github/workflows/k8s_deploy.yml
  with:
    app_name: order-service
  secrets: inherit
```

6.7.3 Consideraciones y mejoras

Múltiples entornos

Se ha definido las pipeline para ejecutarse cada vez que hayan cambios en la rama Master, esto es perfectamente aceptable según la metodología Trunk Base Development, que consiste en realizar constantes y pequeñas actualizaciones en lugar de abrir Pull Requests a tutiplén. Pero se podrían configurar dos clústeres, para mantener el entorno de producción y desarrollo separado, y desplegar en el último todo lo que no sea de producción.

Ejecución selectiva

Sería interesante poder definir reglas para indicar que ficheros se tienen que tocar para que se lance el pipeline de esa aplicación, por ejemplo, que si solo se cambia la ruta *packages/order-service* solo se despliegue la aplicación OrderService. Pero hay que hacerlo minuciosamente para evitar saltarse dependencias. Teniendo recursos suficientes es más fácil desplegar toda la aplicación de una vez.

BlueGreen Deployment

Sería importante aplicar la técnica de Blue-Green deployment que consistiría en hacer una copia del entorno de producción, y una vez se han ejecutado correctamente todas las etapas, actualizar los servicios para que apunten a los nuevos despliegues, evitando así los siguientes problemas:

- **Inestabilidad**: Al ir desplegando por partes la aplicación mostraría un comportamiento errático al tener actualizadas partes de la aplicación dependientes de otras debido a haber aplicado breaking changes.
- **Recuperación:** Si falla un despliegue de las aplicaciones hay que hacer un rollback de todas las que sí han sido desplegadas correctamente.

7. Conclusiones

Durante la realización de este trabajo he ido informando sobre distintos beneficios y desventajas de las distintas técnicas que hay para desarrollar microservicios en Kubernetes. Al elaborar la parte práctica del caso de uso con un E-Commerce en microservicios he podido experimentar algunas de ellas.

Desarrollar una aplicación desde cero en microservicios conlleva una gran carga de trabajo, ya que un cambio sobre los DTO implica realizar cambios en más ubicaciones, por no hablar de cambios en la lógica de negocio, que pueden no romper las integraciones, pero si cambia el comportamiento puede ser nefasto si no se notifica a los equipos correspondientes de llevar el resto de microservicios.

Las transacciones SAGA también añaden una complejidad alta, que solo se va a poder aprovechar si los sistemas y dificultan la trazabilidad, dependiendo en muchas ocasiones de los logs, ya que si estás en un conjunto de microservicios bajo tu control, puedes hacer depuraciones del código, pero si los microservicios están en manos de terceros equipos, la cosa se complicaría.

También se complica el despliegue de la aplicación ya que toca replicar varias veces las mismas pipelines, parametrizarse para que sean reutilizables y buscar que se ejecuten en un orden adecuado, o incluso simultáneo para no perder estabilidad durante los despliegues y realizarlos de manera casi atómica, como en las transacciones de una base de datos. Además, revertir el proceso es complicado y conlleva, preferiblemente, aplicar técnicas complejas como blue-green deployment. Y durante este trabajo no se ha hablado mucho sobre migraciones de base de datos, pero el sincronizar despliegue de aplicaciones y estas migraciones, más, hacer rollback de la base de datos en sintonía es una árdua tarea.

Mi conclusión en resumidas cuentas es que los microservicios, como ya se menciona en el libro de Richardson, es que no es una metodología de desarrollo para comenzar desde el principio y en cambio, Docker es la única pieza realmente indispensable para cualquier equipo de desarrollo.

Bibliografía

[1] Richardson, C. (2019, January 25)**.** *Microservices Patterns.*
[2] Documentación oficial de Kubernetes. https://kubernetes.io/docs/home
[3] Documentación oficial de Docker. https://docs.docker.com
[4] Wilson, B. (2021, June 17). *How To Add Persistent Volume In Google Kubernetes Engine*. DevopsCube. https://devopscube.com/persistent-volume-google-kubernetes-engine
[5] Salguero, E. (2020, August 31). *Arquitectura Hexagonal - Edu Salguero*. Medium. https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f
[6] Lerna, a Javascript Tool for monorepos. - https://lerna.js.org

Índice de figuras

Universidad
de Alcalá