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

Curso académico 2020/2021

Trabajo de Fin de Máster

Ejemplos de migración de Monolito a Microservicios: Base de Datos.

Autores: David Rey González / Juan Escribano Bonilla
Tutor: Micael Gallego

Table of Contents

1. Resumen

Hoy en día en un mundo donde las aplicaciones son cada vez más complejas y están ubicadas en entornos cloud, las arquitecturas de microservicios cobran especial protagonismo debido a las ventajas que ofrecen respecto a las grandes aplicaciones que se desarrollaban como un monolito.

En este trabajo se analizará distintos patrones de descomposición de base de datos, trasladando los ejemplos de alto nivel expuestos en el libro “Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith” (Sam Newman) a implementaciones de bajo nivel.

En el capitulo 2 Introducción y objetivos, se da una definición general de lo que es un microservicio y de las ventajas que ofrece, además se exponen los objetivos generales del proyecto.

En el siguiente capitulo, se desarrollan los cuatro patrones de descomposición de base de datos que hemos analizado. Para cada uno de los patrones, se explica un contexto del mismo, se habla de limitaciones y de usos. A continuación, se desarrolla el ejemplo teórico que expone el libro, en dos versiones diferenciadas, un escenario inicial donde exponemos el problema, y un escenario final donde se muestra la resolución del problema aplicando el patrón en cuestión.

En el capítulo de conclusión y trabajos futuros, se exponen los resultados obtenidos, las dificultades encontradas en el proyecto como una propuesta de evolución de trabajo a futuro.

En el capitulo de bibliografía, enumeramos las fuentes consultadas para la realización del trabajo realizado.

Finalmente, en el capítulo de Anexos, se ha insertado el detalle de los ficheros de despliegue necesarios para ejecutar cada uno de los ejemplos.

2. Introducción y objetivos

Los microservicios son servicios de implementación independiente modelados en torno a un dominio empresarial. Se comunican entre sí a través de redes y, como opción de arquitectura, ofrecen muchas ventajas. Podemos decir que una arquitectura de microservicios se basa en varios microservicios colaborando entre sí.

Las ventajas de los microservicios son muchas y variadas. La naturaleza independiente de las implementaciones abre nuevos modelos para mejorar la escala y la solidez de los sistemas, y le permite mezclar y combinar tecnología. Como los servicios se pueden trabajar en paralelo, puede hacer que más desarrolladores se ocupen de un problema sin que se interpongan en el camino de los demás. También puede ser más fácil para los desarrolladores comprender su parte del sistema, ya que pueden centrar su atención en solo una parte del mismo. El aislamiento de procesos también nos permite variar las opciones de tecnología que hacemos, quizás mezclando diferentes lenguajes de programación, estilos de programación, plataformas de implementación o bases de datos para encontrar la combinación correcta.

Quizás, sobre todo, las arquitecturas de microservicios brindan flexibilidad. Abren muchas más opciones con respecto a cómo puede resolver problemas en el futuro.

Sin embargo, es importante tener en cuenta que ninguna de estas ventajas es gratuita. Hay muchas formas de abordar la descomposición del sistema y, fundamentalmente, lo que está tratando de lograr impulsará esta descomposición en diferentes direcciones. Por lo tanto, es importante comprender lo que está tratando de obtener de su arquitectura de microservicio.

Los objetivos de este proyecto es trasladar los ejemplos expuestos en el libro a ejemplo concretos que puedan servir para la mayor comprensión de los patrones de descomposición de base de datos al transformar una arquitectura de monolito a microservicios.

La estructura de los ejemplos desarrollados, ha sido implementada de la misma forma, es decir, partimos de una versión inicial “v1” donde se muestra el problema y una versión final “v2” donde se aplica el patrón que se está exponiendo en cada apartado.

Para el despliegue de todos los casos de uso, se han dockerizado cada uno de los servicios necesarios para cada versión de cada patrón explicado y adicionalmente se han utilizado las imágenes publicadas en dockerhub para dar una coherencia funcional al ejemplo (ej. MySql, Debezium, Kafka). El despliegue de los mismos se realizará mediante docker compose.

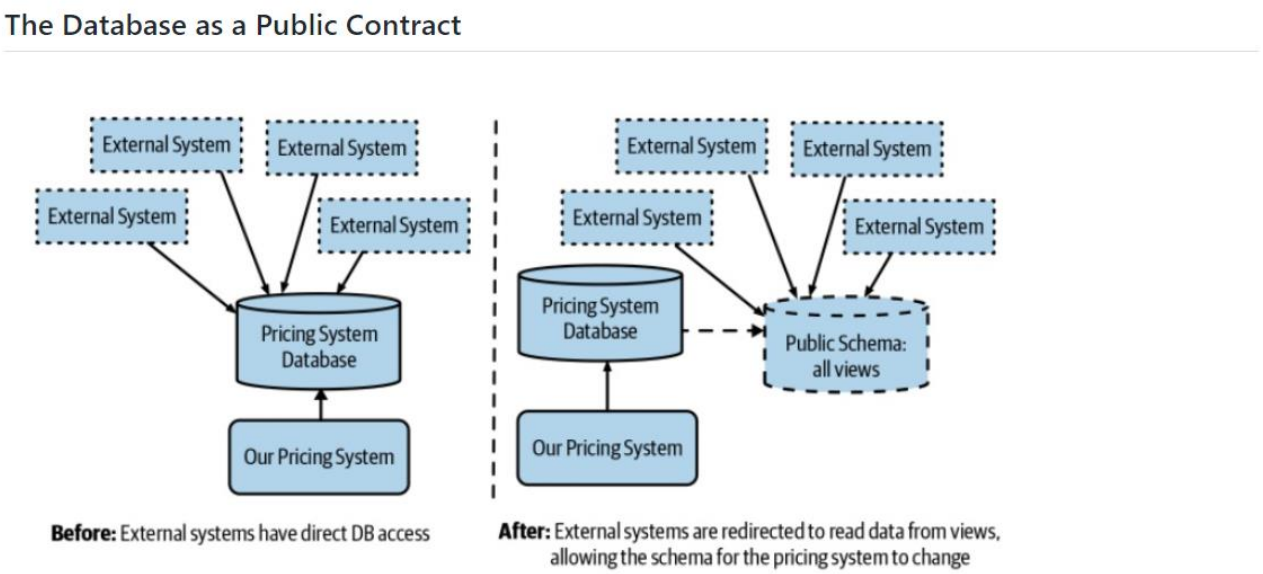
Por último, se ha utilizado MySql como motor de persistencia, Java como lenguaje de programación, JPA como ORM y Debezium como CDC (Change Data Capture). El framework utilizado ha sido SpringBoot, ya que facilitaba la implementación de los ejemplos y era con el que nos sentíamos más familiarizados.

3. Patrones de descomposición

A continuación, se desarrollan los cuatro patrones de descomposición de base de datos que hemos analizado. Para cada uno de los patrones, se explica un contexto del mismo, se habla de limitaciones y de usos. Se desarrolla el ejemplo teórico que expone el libro, en dos versiones diferenciadas, un escenario inicial donde exponemos el problema, y un escenario final donde se muestra la resolución del problema aplicando el patrón en cuestión.

3.1. Database View

En una situación en la que queremos una única fuente de datos para varios servicios, se puede utilizar una vista para mitigar los problemas relacionados con el acoplamiento. Con una vista, un servicio se puede presentar con un esquema que es una proyección limitada de un esquema subyacente. Esta proyección puede limitar los datos que son visibles para el servicio, ocultando información a la que no debería tener acceso. [1]



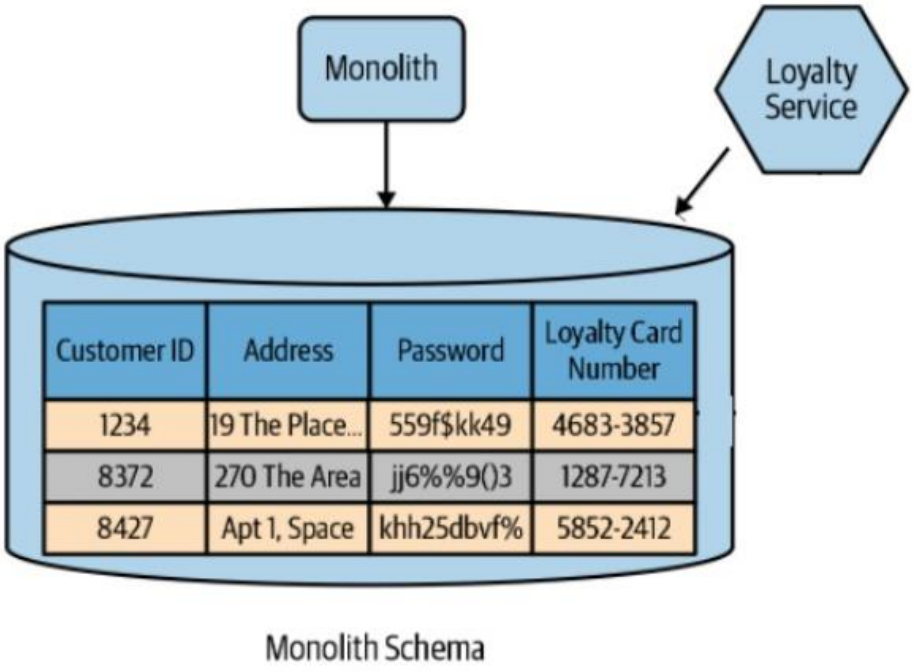
- Limitaciones
  - Es importante ver el impacto en el rendimiento de las vistas (caché frente a consulta en línea)
  - Hay motores de base de datos que solo admiten vistas de solo lectura

- Donde usarlo
  - Debe evitarse tanto como sea posible
  - Hay ocasiones en las que otros servicios no se pueden actualizar para utilizar un servicio y necesitan acceder directamente a la base de datos.

En nuestro ejemplo, es un proyecto para administrar clientes. Podemos buscar, agregar, actualizar o eliminar clientes. Usamos JPA como ORM para almacenar, acceder y administrar objetos Java en una base de datos Mysql.

En la v1, tenemos dos servicios independientes (Monolith & Loyalty Service) y ambos acceden al mismo repositorio (esquema Monolith). Desde el "Servicio Monolith" puedo ejecutar operaciones

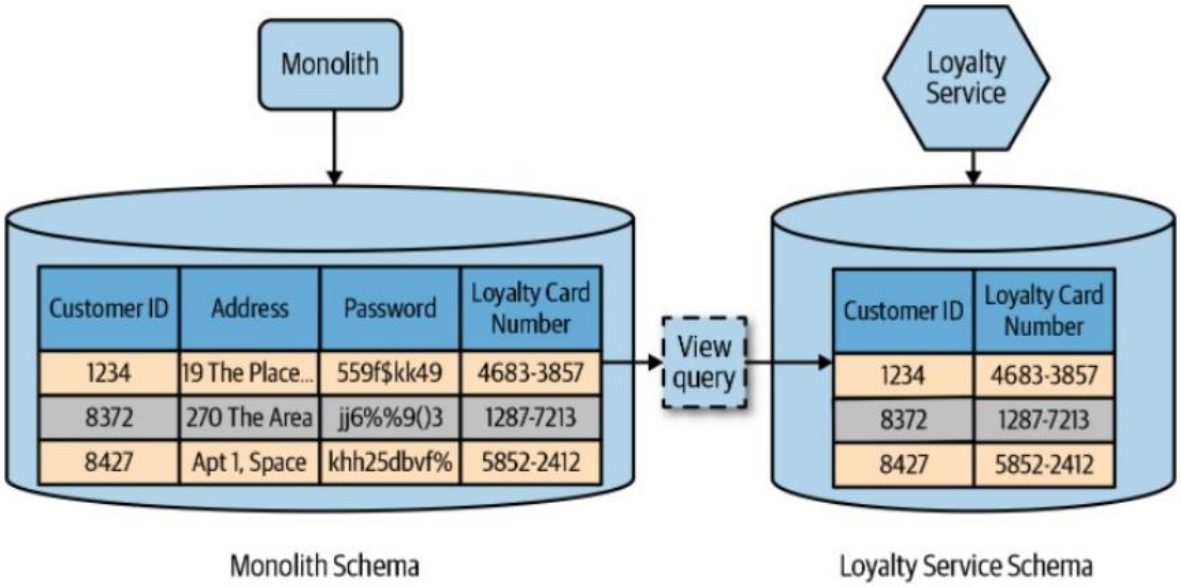
CRUD mientras que desde el "Servicio de Fidelización" solo puedo ejecutar operaciones de lectura. El "Monolith Schema" tiene una tabla de USER con cuatro campos: ID, ADDRESS, PASSWORD, LOYALTY_CARD_NUMBER. Cuando se inicie la aplicación, vamos a insertar algunos datos en la tabla de usuarios.



Monolith Schema

En la v2, tenemos dos microservicios (monolith y loyaltyService) donde ambos obtienen datos de diferentes esquemas de bases de datos (Monolith Schema y Loyalty Service Schema). La información de usuario requerida por "Servicio de fidelización" no se encuentra en "Esquema de servicio de fidelización", pero tenemos una vista en la tabla T_USER de "Esquema de monolito" de "Servicio de fidelización". La vista creada es la siguiente:

```
CREATE or replace VIEW loyalty.t_loyalty_card_number_view AS
  SELECT id AS id, loyalty_card_number AS loyalty_card_number
  FROM monolith.t_user;
```



Podemos ver el despliegue del servicio implementado en la versión de anexos y el enlace al repositorio es el siguiente:

(https://github.com/MasterCloudApps-Projects/Monolith-to-Microservices-DB-Examples/tree/main/01_DatabaseView)

3.2. Database as a Service

A veces, los clientes solo necesitan una base de datos para realizar consultas. En estas situaciones, puede tener sentido permitir que los clientes vean los datos que su servicio administra en una base de datos, pero debemos tener cuidado de separar la base de datos que exponemos de la base de datos que usamos dentro de los límites de nuestro servicio. Un enfoque es crear una base de datos dedicada diseñada para exponerse como un punto final de solo lectura y tener esta base de datos poblada cuando cambien los datos en la base de datos subyacente. En efecto, de la misma manera que un servicio podría exponer un flujo de eventos como un punto final y una API síncrona como otro punto final, también podría exponer una base de datos a consumidores externos. [1]

- La base de datos de informes está sincronizada
- Denominada "Base de datos de informes" por Fowler
- Una forma de implementar CQRS

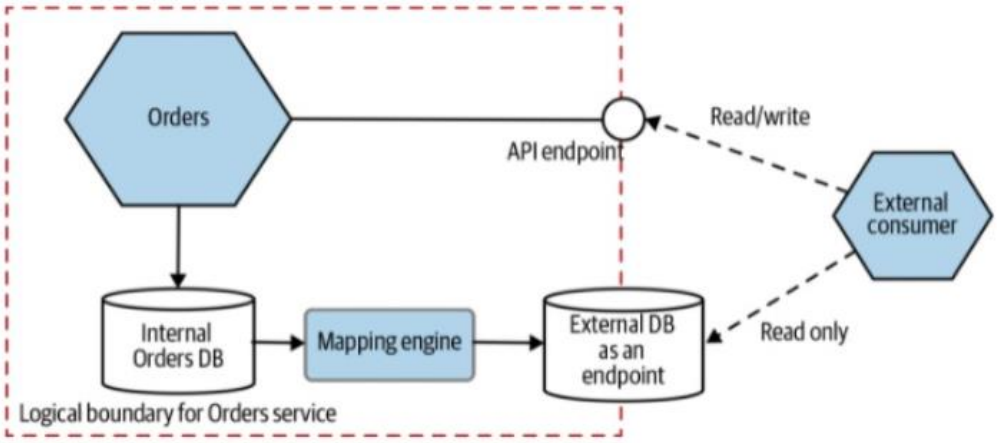


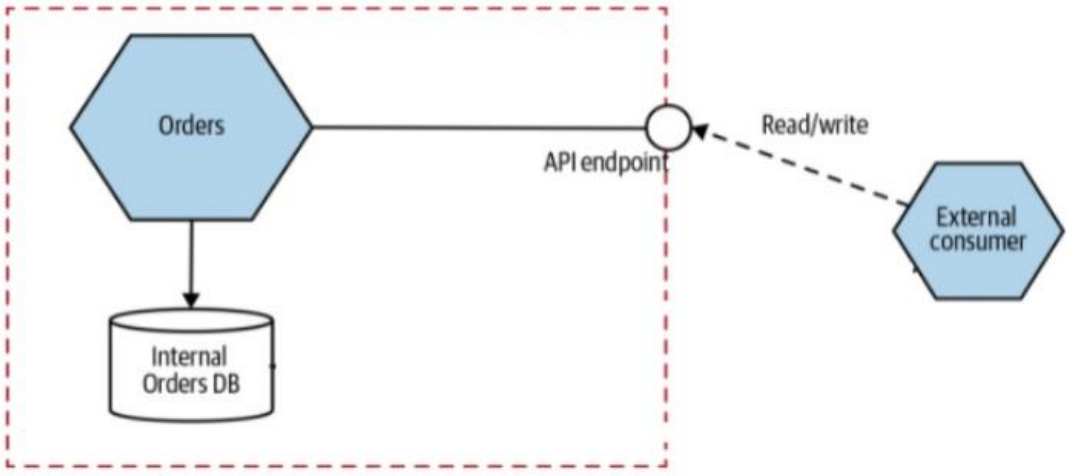
Figure 4-7. Exposing a dedicated database as an endpoint, allowing the internal database to remain hidden

- Impementación
  - CDC (Change data capture) servicios.
  - Batch actualización de datos.

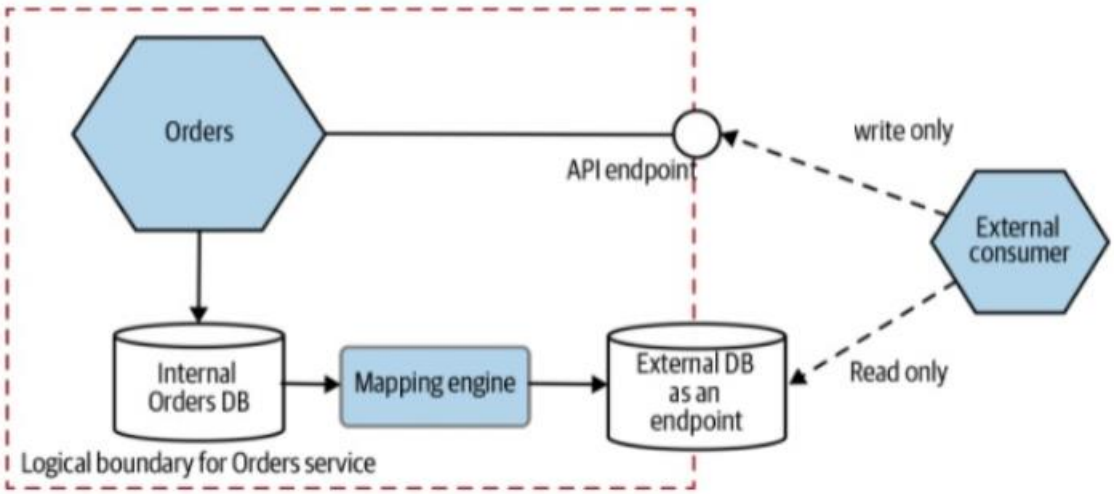
- Comparación con vistas:
  - Mas flexible
  - Permite diferentes motores de base de datos
  - Transformaciones más complejas
  - Pero a un coste ...

En nuestro ejemplo, es un proyecto para administrar pedidos. Podemos buscar, agregar, actualizar o eliminar pedidos. Usamos JPA como ORM para almacenar, acceder y administrar objetos Java en una base de datos Mysql.

En la v1, tenemos dos servicios, (Servicio de pedido y un servicio de consumidor externo) El "Servicio de pedido" puede ejecutar operaciones CRUD mientras que desde el "Servicio de consumidor externo" puede leer y escribir datos desde el punto final API desde el "Servicio de pedido". El "Esquema de pedidos" tiene una tabla de ORDERS con cuatro campos: DATE_ORDER, PURCHASER, ADDRESS, TOTAL_PRICE para registrar los pedidos.



En la v2, tenemos dos servicios, (Servicio de pedidos y un servicio al consumidor externo). Tenemos un consumidor externo que escribe datos desde el punto final de la API del servicio de pedidos y lee datos del repositorio de base de datos externo. La base de datos de orden interna y la base de datos externa están sincronizadas por un CDC (Debezium). Para este caso, vamos a sincronizar dos bases de datos mysql y la tabla Pedidos.



El enlace al repositorio es el siguiente:

https://github.com/MasterCloudApps-Projects/Monolith-to-Microservices-DB-Examples/tree/main/02_DatabaseAsAServiceInterface

3.3. Aggregate Exposing Monolith

En la imagen siguiente, se observa que nuestro nuevo servicio de facturación necesita acceder a una variedad de información que no está directamente relacionada con la gestión de la facturación. Como mínimo, necesita información sobre nuestros empleados actuales para gestionar los flujos de trabajo de aprobación. Actualmente, todos estos datos se encuentran dentro de la base de datos monolith. Al exponer información sobre nuestros Empleados a través de un punto final de servicio (podría ser una API o un flujo de eventos) en el monolito mismo, hacemos explícita la información que necesita el servicio de Factura. [2] [3]

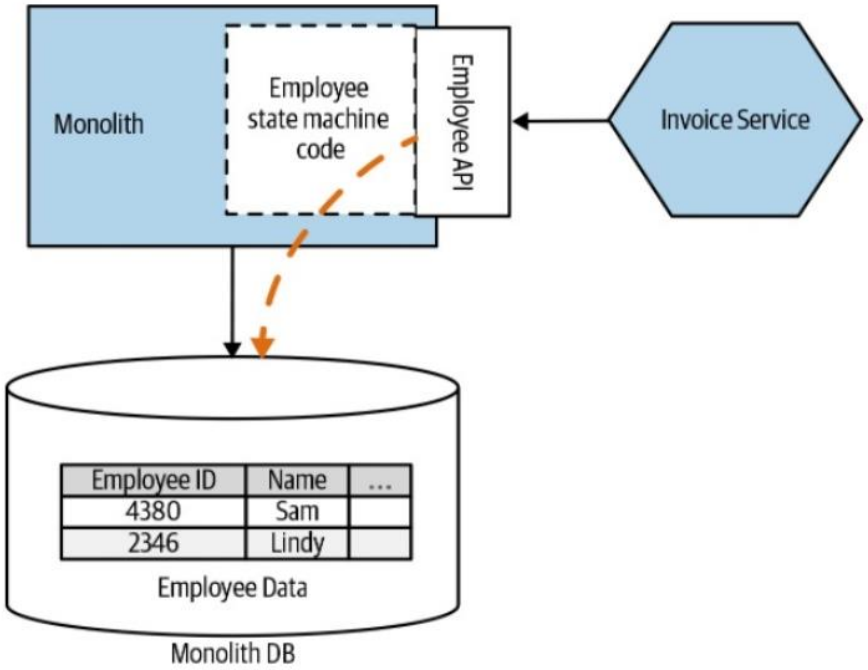
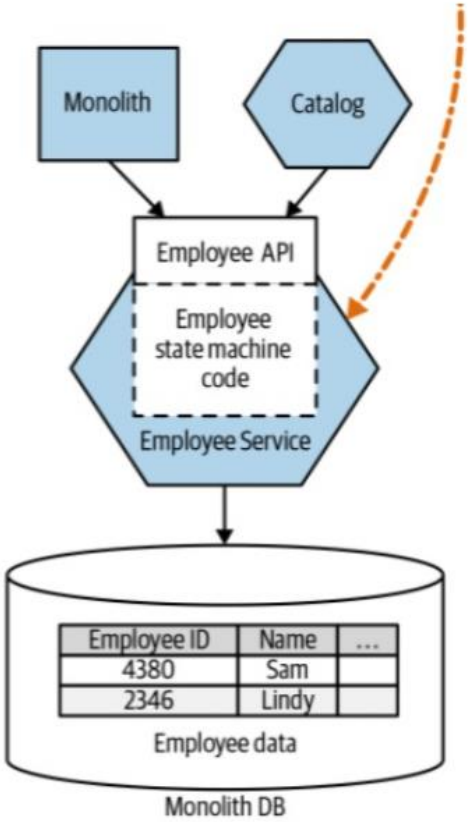


Figure 4-8. Exposing information from the monolith via a proper service interface, allowing our new microservice to access it

- Exponer un agregado como una API

  - No se trata simplemente de exportar la base de datos
  - Se exporta el servicio de gestión agregada
  - Se empieza a controlar el ciclo de vida de estos datos
  - Es un paso previo a la extracción de esa API en su propio microservicio en el futuro.



- Donde usarlo
  - Cuando necesita acceder a los datos del monolito porque aún no se han extraído
  - Es un paso preliminar para el nuevo microservicio razonable.
  - En general, es mucho mejor que usar vistas.
  - No se puede usar si el monolito no puede / no cambiará

En nuestro ejemplo, es un proyecto para administrar los datos de los empleados. Podemos buscar, agregar, actualizar o eliminar datos de empleados. Usamos JPA como ORM para almacenar, acceder y administrar objetos Java en una base de datos Mysql.

En la v1, tenemos un servicio de facturación que obtiene datos de empleados de Monolith DB a través de un punto final de empleados. Desde el punto final, puedo ejecutar operaciones CRUD. El "Esquema de Monolito" tiene una tabla EMPLOYEE con cuatro campos: DATE_BIRTH, DOCUMENT, LAST_NAME, NAME. Cuando se inicie la aplicación, vamos a insertar algunos datos en la tabla de empleados.

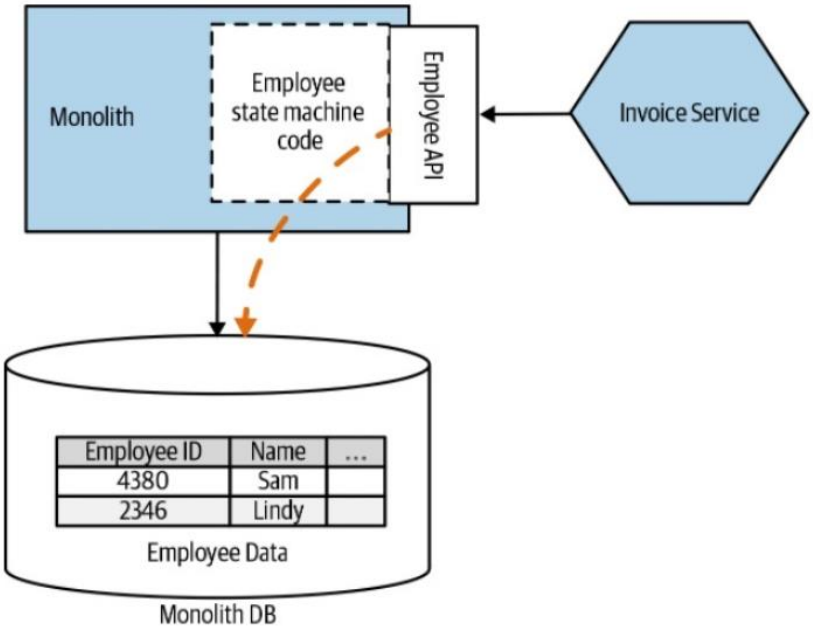
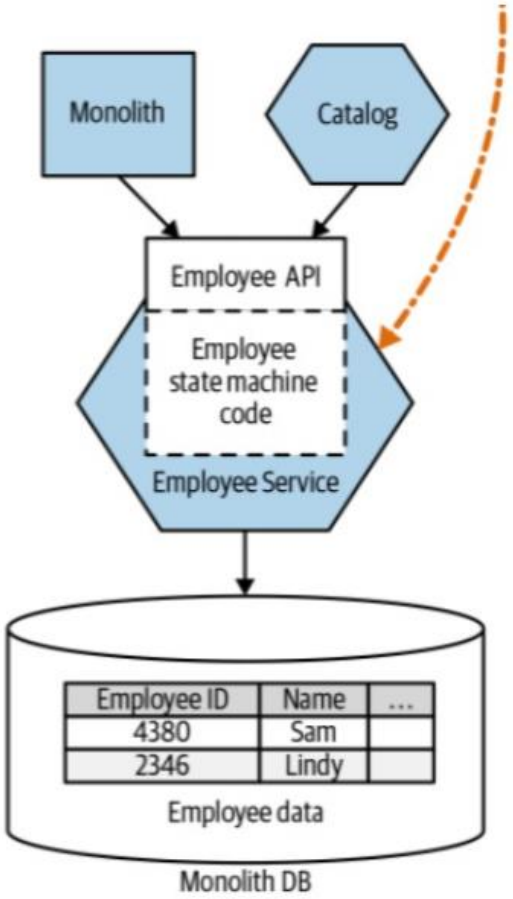


Figure 4-8. Exposing information from the monolith via a proper service interface, allowing our new microservice to access it

En la v2, los datos de los empleados están expuestos por el Servicio de empleados. Todos los servicios que necesitan datos de empleados obtienen los datos a través de la API de empleados



El enlace al repositorio es el siguiente:

https://github.com/MasterCloudApps-Projects/Monolith-to-Microservices-DB-Examples/tree/main/03_AggregateExposingMonolith

3.4. Split Table

A veces, encontrará datos en una sola tabla que deben dividirse en dos o más límites de servicio, y eso puede ser interesante. En la imagen siguiente vemos una sola tabla compartida, Artículo, donde almacenamos información no solo sobre el artículo que se vende, sino también sobre los niveles de existencias. [1]

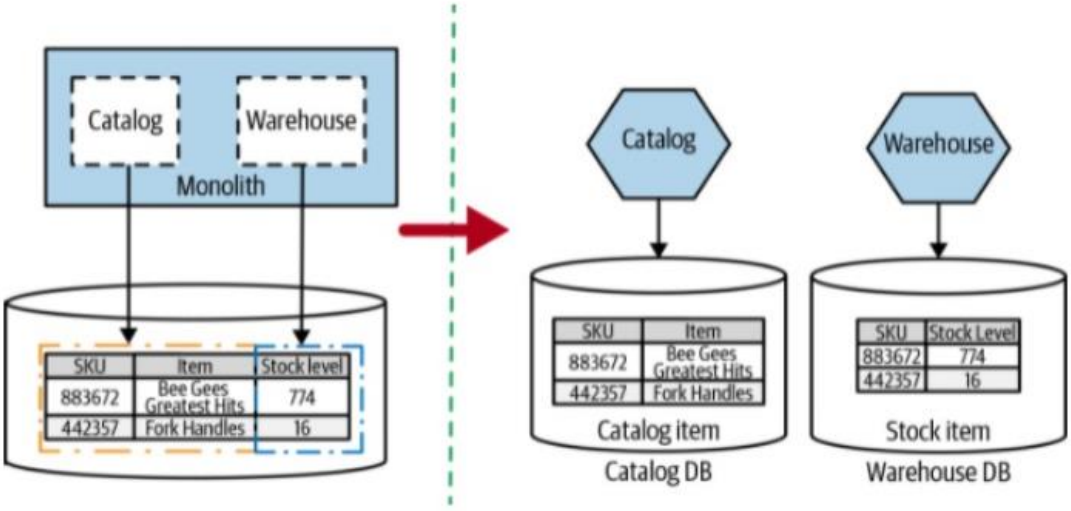


Figure 4-34. A single table that bridges two bounded contexts being split

¿Qué sucede cuando varios fragmentos de código actualizan la misma columna? En la siguiente imagine tenemos una tabla de Customer, que contiene una columna de Status.

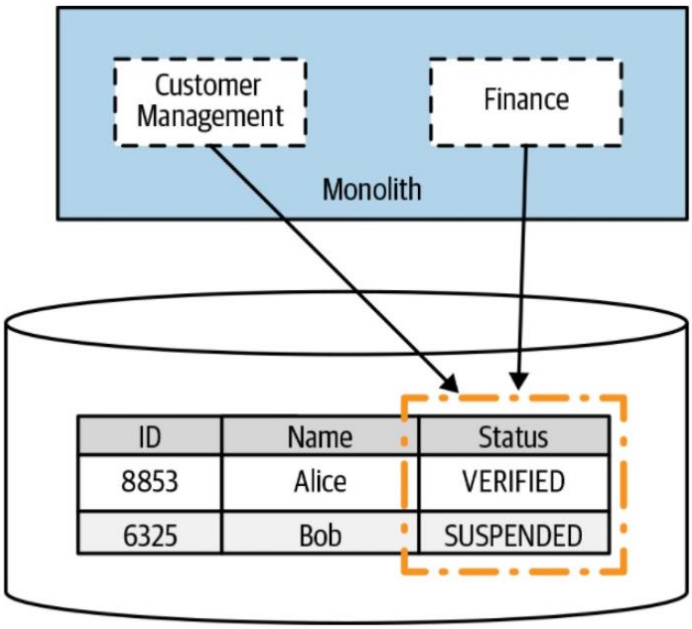


Figure 4-35. Both customer management and finance code can change the status in the Customer table

Esta columna se actualiza durante el proceso de registro del cliente para indicar que una persona determinada ha verificado (o no) su correo electrónico, con un valor que va de NOT_VERIFIED → VERIFIED. Una vez que un cliente está VERIFIED, puede comprar. Nuestro código financiero maneja la suspensión de clientes si sus facturas están impagadas, por lo que en ocasiones cambiarán el estado de un cliente a SUSPENDED. En este caso, el estado de un cliente todavía debería ser parte del modelo de dominio del cliente y, como tal, debería ser administrado por el servicio de atención al cliente que se creará próximamente. Se pretende mantener las máquinas de estado para nuestras entidades de dominio dentro de un límite de servicio único, y la actualización de un estado forma parte de la máquina de estado para un cliente. Esto significa que cuando se haya realizado la división del servicio,

nuestro nuevo servicio de Finanzas deberá realizar una llamada de servicio para actualizar este estado, como vemos en la imagen siguiente:

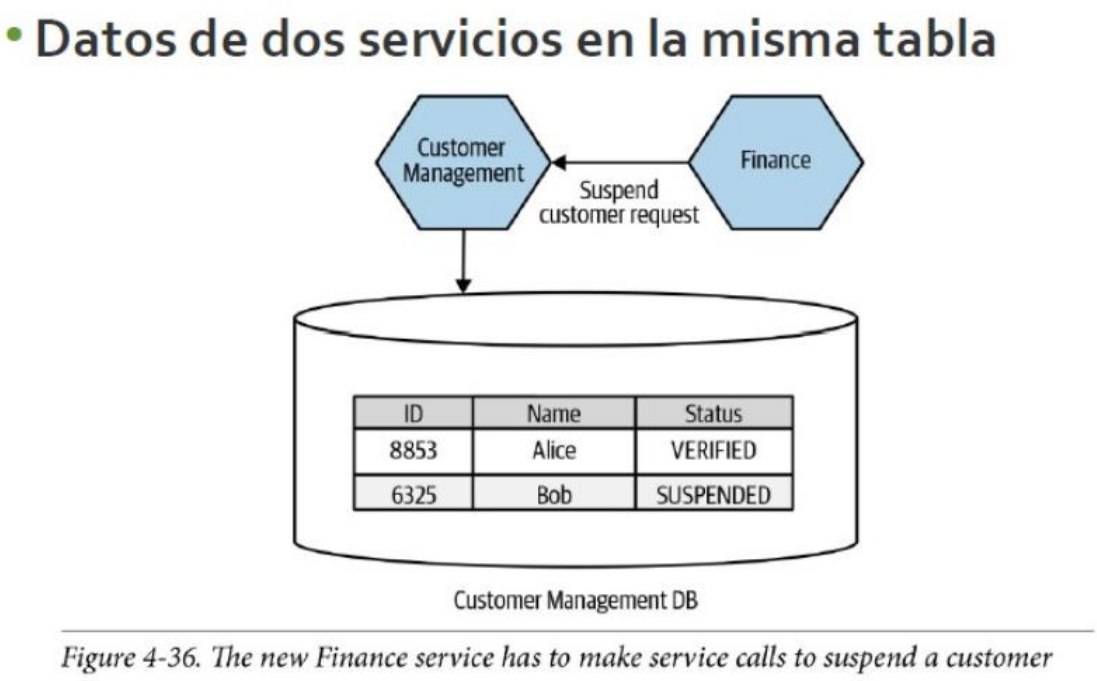


Figure 4-36. The new Finance service has to make service calls to suspend a customer

- Donde usarlo

A primera vista, estos parecen bastante sencillos. Cuando la tabla pertenece a dos o más contextos delimitados en su monolito actual, debe dividir la tabla a lo largo de esas líneas. Si encuentra columnas específicas en esa tabla que parecen estar actualizadas por múltiples partes de su base de código, debe tomar una decisión sobre quién debería "poseer" esos datos. ¿Es un concepto de dominio existente que tiene dentro del alcance? Eso ayudará a determinar dónde deben ir estos datos.

En nuestro ejemplo, es un proyecto para administrar el estado del cliente. Usamos JPA como ORM para almacenar, acceder y administrar objetos Java en una base de datos Mysql.

En la v1, tenemos un monolito con dos controladores (finanzas y cutomer) para cambiar el estado del cliente. El "Esquema de gestión de clientes" tiene una tabla CUSTOMER con 3 campos: ID, NAME, STATE. Cuando se inicie la aplicación, vamos a insertar algunos datos en la tabla de clientes.

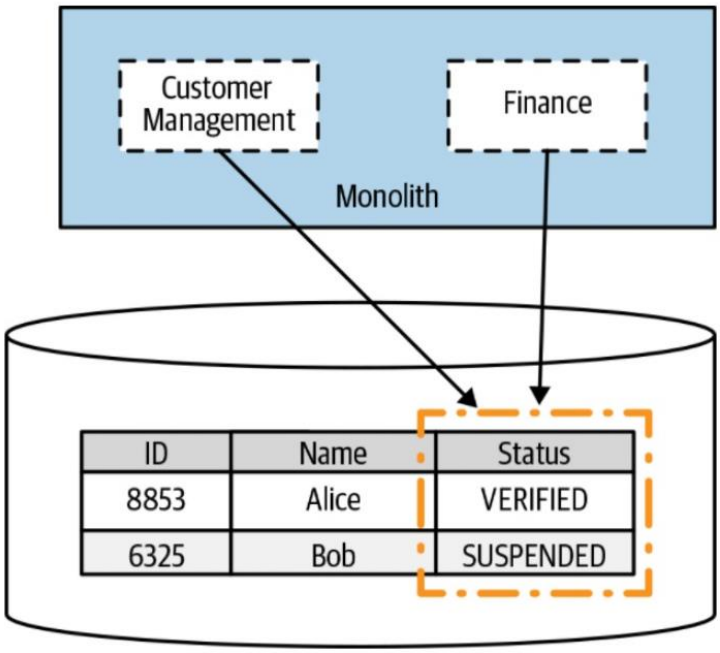


Figure 4-35. Both customer management and finance code can change the status in the Customer table

En la v2, tenemos dos servicios (cliente y finanzas). El único servicio que puede cambiar el estado del cliente directamente es el servicio de gestión de clientes. El servicio de finanzas debe pasar por el servicio de Gestión de clientes.

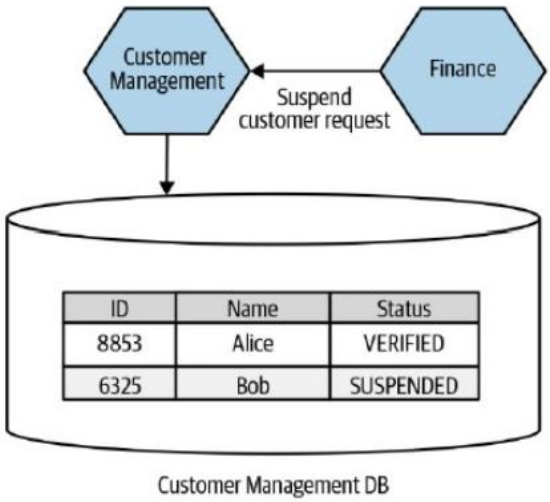


Figure 4-36. The new Finance service has to make service calls to suspend a customer

El enlace al repositorio es el siguiente:

https://github.com/MasterCloudApps-Projects/Monolith-to-Microservices-DB-Examples/tree/main/04_SplitTable

4. Conclusiones y trabajos futuros

En este TFM se puede encontrar ejemplos de diferentes patrones de descomposición de bases de datos, que ayudaran a la comprensión de los ejemplos de alto nivel que contiene el libro de referencia.

Este proyecto tiene una finalidad didáctica, donde los lectores encontrarán ejemplos completos implementados en código

A la hora de desarrollar estos ejemplos, el patrón que más dificultad presentaba, era el "Database as a Service", debido a que necesitábamos implementar un CDC (Change Data Capture) y para ello, nos apoyamos en Debezium. En concreto, la configuración de los diferentes conectores necesarios para la sincronización entre las bases de datos, supuso la mayor dificultad.

Durante el master, se ha hecho más hincapié en la descomposición de monolito a microservicios en la parte de código estudiando y realizando prácticas en este aspecto. Gracias a esta investigación en el TFM, hemos podido profundizar en diferentes técnicas para la descomposición de monolito a microservicios, pero poniendo el foco en la problemática derivada de la base de datos.

Una vez finalizada esta investigación, como trabajo futuro se propone la implementación del resto de patrones de base de datos, como son:

- Shared database
- Database wrapping service
- Change Data Ownership
- Synchronize Data in Application
- Tracer Write
- Move Foreign-Key Relationships to Code

5. Bibliografía

[1] Sam Newman (2019) “Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith”. < https://learning.oreilly.com/library/view/monolith-to-microservices/9781492047834/ch04.html > [Consulta: 20 de noviembre de 2021]

[2] Debezium – Oficial- <https://debezium.io/documentation/reference/tutorial.html> [Consulta: 01 de noviembre de 2021]

[3] Paradigma Digital: Primeros pasos con Debezium <https://www.paradigmadigital.com/dev/primeros-pasos-con-debezium/ > [Consulta: 01 de noviembre de 2021]

6. Anexos

En este apartado se detalle los ficheros de despliegue necesarios para ejecutar cada uno de los ejemplos:

6.1. Database View

En la versión v1 vamos a desplegar la base de datos, y la v1 del monolith y el servicio de fidelización. Ambos servicios se alojarán en dockerhub y se implementarán con un archivo docker-compose:

```
services:
  monolith:
  image: juaneb/database_view_monolith_v1
  ports:
  - 8080:8080
  environment:
  # Enviroment variables for connect to MySQL
  - MYSQL_HOST=mysql
  depends_on:
  - mysql
  restart: on-failure

  loyaltyservice:
  image: juaneb/database_view_loyalty_v1
  ports:
  - 8090:8090
  environment:
  - MYSQL_HOST=mysql
  restart: on-failure

  mysql:
  image: mysql:8.0.25
  ports:
  - 3306:3306
  environment:
  # Enviroment variables for securize MySQL and create default Da-
tabase
  - MYSQL_DATABASE=monolith
  - MYSQL_ROOT_PASSWORD=pass
  volumes:
  - ./mysql_db:/var/lib/mysql
  restart: always
```

En la versión v2 vamos a desplegar la base de datos, y la v2 del monolito y el servicio de fidelización. Ambos servicios se alojarán en dockerhub y se implementarán con un archivo docker-compose:

```
version: '3.9'
    services:
    monolith:
    image: juaneb/database_view_monolith_v2
```

```
    ports:
    - 8080:8080
    environment:
    # Enviroment variables for connect to MySQL
    - MYSQL_HOST=mysql
    depends_on:
    - mysql
    restart: on-failure

    loyaltyservice:
    image: juaneb/database_view_loyalty_v2
    ports:
    - 8090:8090
    environment:
    - MYSQL_HOST=mysql
    depends_on:
    - mysql
    - monolith
    restart: on-failure

    mysql:
    image: mysql:8.0.25
    ports:
    - 3306:3306
    environment:
    # Enviroment variables for securize MySQL and create default Database
    - MYSQL_DATABASE=monolith
    - MYSQL_ROOT_PASSWORD=pass
    volumes:
    - ./mysql_db:/var/lib/mysql
    restart: always
```

Ambos servicios utilizan mysql como base de datos, cuyas tablas son creadas por hibernación en el inicio. Vamos a utilizar la ruta migratoria para desplegar la vista en la V2 del servicio de fidelización.

6.2. Database as a service

En la versión v1 vamos a desplegar la base de datos, y la v1 del pedido y el servicio al consumidor externo. Ambos servicios se alojarán en dockerhub y se implementarán con un archivo docker-compose:

```
version: '3.9'
services:
       order:
              image: dreyg/database_as_a_service_order_v1
              ports:
              - 8080:8080
              environment:
              # Enviroment variables for connect to MySQL
              - MYSQL_HOST=mysql
              depends_on:
```

```
            - mysql
            restart: on-failure

      externalconsumer:
            image: dreyg/database_as_a_service_external_consumer_v1
            ports:
            - 8090:8090
            environment:
            - HOST=order
            restart: on-failure

      mysql:
            image: mysql:8.0.25
            ports:
            - 3306:3306
            environment:
            # Enviroment variables for securize MySQL and create de-
fault Database
            - MYSQL_DATABASE=order
            - MYSQL_ROOT_PASSWORD=pass
            volumes:
            - ./mysql_db:/var/lib/mysql
            restart: always
```

En la versión v2 vamos a desplegar Debezium (servicio zookeper, kafka y conector), dos bases de datos (mysqlproducer, mysqlsuscriber) y la versión v2 del pedido y el servicio de consumidor externo. Todos los servicios se alojarán en dockerhub y se implementarán con un archivo docker-compose:

```
version: '3.9'
services:
mysqlproducer:
      image: mysql:8.0.25
      ports:
      - 3306:3306
      environment:
      # Enviroment variables for securize MySQL and create default Data-
base
      - MYSQL_DATABASE=order
      - MYSQL_ROOT_PASSWORD=pass
      volumes:
      - ./mysql_db_producer:/var/lib/mysqlproducer
      restart: always

zookeeper:
      image: debezium/zookeeper:1.7
      ports:
      - 2181:2181
      - 2888:2888
      - 3888:3888
```

```
        restart: on-failure

kafka:
        image: debezium/kafka:1.7
        ports:
        - 9092:9092
        environment:
        - ZOOKEEPER_CONNECT=zookeeper:2181
        - HOST_NAME=kafka
        depends_on:
        - zookeeper
        restart: on-failure

mysqlsuscriber:
        image: mysql:8.0.25
        ports:
        - 3307:3306
        environment:
        # Enviroment variables for securize MySQL and create default Data-
base
        - MYSQL_DATABASE=order
        - MYSQL_ROOT_PASSWORD=pass
        volumes:
        - ./mysql_db_suscriber:/var/lib/mysqlsuscriber
        restart: always

connector:
        image: debezium/connect:1.7
        ports:
        - 8083:8083
        environment:
        - BOOTSTRAP_SERVERS=kafka:9092
        - GROUP_ID=2
        - CONFIG_STORAGE_TOPIC=my_connect_configs
        - OFFSET_STORAGE_TOPIC=my_connect_offsets
        - STATUS_STORAGE_TOPIC=my_connect_statuses
        depends_on:
        - mysqlproducer
        - mysqlsuscriber
        - zookeeper
        - kafka
        restart: on-failure

order:
        image: juaneb/database_as_a_service_order_v2
        ports:
      - 8080:8080
      environment:
      # Enviroment variables for connect to MySQL
      - MYSQL_HOST=mysqlproducer
      depends_on:
```

```
      - mysqlproducer
      - mysqlsuscriber
      - zookeeper
      - kafka
      restart: on-failure

externalconsumer:
      image: juaneb/database_as_a_service_external_consumer_v2
      ports:
      - 8090:8090
      environment:
      - MYSQL_HOST=mysqlsuscriber
      - HOST=order
      - BROKER_HOST=kafka
      depends_on:
      - mysqlproducer
      - mysqlsuscriber
      - zookeeper
      - kafka
      restart: on-failure
```

Necesitamos configurar el conector para sincronizar ambas bases de datos (mysqlproducer y mysqlsuscriber). Consumimos la API del conector:

curl-i-X POST-H "Accept:application/json"-H "Content-Type:application/json" localhost:8083/connectors/-d @conector-mysql.json

El conector-mysql.json tiene la configuración para el conector: [3]

' { "name": "database-as-a-service-connector", "config": { "connector.class": "io.debezium.connector.mysql.MySqlConnector", "tasks.max": "1", "database.hostname": "mysqlproducer", "database.port": "3306", "database.user": "root", "database.password": "pass", "database.server.id": "184054", "database.server.name": "dbserver1", "database.include.list": "order", "database.history.kafka.bootstrap.servers": "kafka:9092", "database.history.kafka.topic": "schema-changes.order" } }

6.3. Aggregate Exposing Monolith

En la versión v1 vamos a desplegar la base de datos, y la v1 del monolito y el servicio de facturación. Ambos servicios se alojarán en dockerhub y se implementarán con un archivo docker-compose:

```
version: '3.9'
services:
    mysql:
        image: mysql:8.0.25
        ports:
        - 3306:3306
        environment:
```

```
        # Enviroment variables for securize MySQL and create default Da-
tabase
        - MYSQL_DATABASE=monolith
        - MYSQL_ROOT_PASSWORD=pass
        volumes:
        - ./mysql_db:/var/lib/mysql
        restart: always
    monolith:
        image: juaneb/aggregate_exposing_monolith_monolith_v1
        ports:
        - 8080:8080
        environment:
        # Enviroment variables for connect to MySQL
        - MYSQL_HOST=mysql
        depends_on:
        - mysql
        restart: on-failure
    invoiceservice:
        image: juaneb/aggregate_exposing_monolith_invoice_v1
        ports:
        - 8090:8090
        environment:
        - MYSQL_HOST=mysql
        - HOST=monolith
        restart: on-failure
```

En la versión v2 vamos a desplegar la base de datos, y la v2 del servicio monolith, employee y catálogo. Todos los servicios se alojarán en dockerhub y se implementarán con un archivo docker-compose:

```
version: '3.9'
services:
    mysql:
        image: mysql:8.0.25
        ports:
        - 3306:3306
        environment:
        # Enviroment variables for securize MySQL and create default Da-
tabase
        - MYSQL_DATABASE=employee
        - MYSQL_ROOT_PASSWORD=pass
        volumes:
        - ./mysql_db:/var/lib/mysql
        restart: always

    employee:
        image: juaneb/aggregate_exposing_monolith_employee_v2
        ports:
        - 8080:8080
        environment:
        - MYSQL_HOST=mysql
        restart: on-failure
```

```
    monolith:
        image: juaneb/aggregate_exposing_monolith_monolith_v2
        ports:
        - 8090:8090
        environment:
        # Enviroment variables for connect to MySQL
        - MYSQL_HOST=mysql
        - HOST=employee
        depends_on:
        - mysql
        restart: on-failure

    catalog:
        image: juaneb/aggregate_exposing_monolith_catalog_v2
        ports:
        - 8100:8100
        environment:
        # Enviroment variables for connect to MySQL
        - MYSQL_HOST=mysql
        - HOST=employee
        depends_on:
        - mysql
        restart: on-failure
```

6.4. Split Table

En la versión v1 vamos a implementar la base de datos y la v1 del monolito. El monolito se alojará en dockerhub y se implementará con un archivo docker-compose:

```
version: '3.9'
services:
mysql:
    image: mysql:8.0.25
    ports:
    - 3306:3306
    environment:
    # Enviroment variables for securize MySQL and create default Database
    - MYSQL_DATABASE=order
    - MYSQL_ROOT_PASSWORD=pass
    volumes:
    - ./mysql_db:/var/lib/mysql
    restart: always

monolith:
    image: juaneb/split_table_monolith_v1
    ports:
    - 8080:8080
    environment:
    # Enviroment variables for connect to MySQL
    - MYSQL_HOST=mysql
```

```
    depends_on:
    - mysql
    restart: on-failure
```

En la versión v2 vamos a desplegar la base de datos y la v2 del cliente un servicio financiero. Todos los servicios se alojarán en dockerhub y se implementarán con un archivo docker-compose:

```
version: '3.9'
services:
mysql:
    image: mysql:8.0.25
    ports:
    - 3306:3306
    environment:
    # Enviroment variables for securize MySQL and create default Database
    - MYSQL_DATABASE=customer
    - MYSQL_ROOT_PASSWORD=pass
    volumes:
    - ./mysql_db:/var/lib/mysql
    restart: always

customer:
    image: juaneb/split_table_customer_v2
    ports:
    - 8080:8080
    environment:
    # Enviroment variables for connect to MySQL
    - MYSQL_HOST=mysql
    depends_on:
    - mysql
    restart: on-failure

finance:
    image: juaneb/split_table_finance_v2
    ports:
    - 8090:8090
    environment:
    - MYSQL_HOST=mysql
    - HOST=customer
    restart: on-failure
```