6. Un mejor amigo para el frontend

En este capítulo, nos centraremos en dar soporte a los usuarios finales. Aprenderás cómo crear servicios backend for frontend que atiendan los tipos de cambios impulsados por los actores humanos que usan el sistema.

Enfoque en las actividades del usuario

Habilitamos el cambio continuo definiendo límites fortificados alrededor de las cosas que cambian juntas para poder controlar el alcance y el impacto de cualquier cambio dado. **La clave para definir estos límites es comprender la fuerza impulsora detrás del cambio.**

En este capítulo, profundizamos en los detalles del patrón de servicio Backend for Frontend (BFF). El patrón de servicio BFF trabaja en el límite del sistema para dar soporte a los usuarios finales y las actividades que realizan.

El propósito del patrón de servicio BFF es centrarse en los requisitos de las actividades del usuario final y dejar que todas las demás preocupaciones pasen a segundo plano. Las preocupaciones sobre integraciones no deben sobrecargar un servicio BFF. Cada actividad de usuario es un paso en un proceso de negocio más amplio, pero solo las entradas y salidas del paso son relevantes para un servicio BFF.

Un servicio BFF es responsable de una única actividad de usuario

Siguiendo el Principio de Responsabilidad Única (SRP), un servicio BFF es responsable ante uno y solo un actor humano que interactúa con la interfaz de usuario del sistema. Sin embargo, un solo actor humano puede interactuar con un sistema de múltiples formas para realizar diferentes actividades. En este caso, tener un servicio BFF que soporte múltiples actividades de usuario podría generar demandas en competencia, a menos que las actividades estén estrechamente relacionadas.

Considerando todos estos factores, lo mejor es crear un servicio BFF separado para cada actividad de usuario discreta.

Un servicio BFF es propiedad del equipo

El patrón de servicio BFF facilita el cambio porque el mismo equipo es dueño e implementa tanto el frontend como el backend.

Esto permite que un equipo siga un enfoque UI-first. El equipo prototipa rápidamente las interfaces de usuario sin backend, para obtener retroalimentación rápida del usuario final. A su vez, la UI impulsa los requisitos del servicio BFF. Este enfoque identifica fácilmente las consultas y comandos necesarios y elimina el desperdicio que surge cuando un equipo de backend separado implementa funcionalidades que asume que un frontend necesitará.

Cuando un equipo está listo para implementar un servicio BFF, normalmente aprovechará el mismo lenguaje usado para implementar el frontend, como JavaScript. Esto permite que los mismos desarrolladores usen las mismas herramientas para trabajar en el frontend y el backend.

Esto aumenta la productividad porque reduce el cambio de contexto a medida que los desarrolladores se mueven entre tareas relacionadas de frontend y backend.

Un servicio BFF es desacoplado, autónomo y resiliente

Un enfoque común es implementar un BFF como un servicio facade que envuelve y agrega las llamadas a otros servicios. **Pero este enfoque es demasiado frágil.** Elimina el acoplamiento en tiempo de diseño entre el frontend y los diversos servicios backend, pero no elimina el acoplamiento en tiempo de ejecución. Si cualquiera de estos otros servicios sufre una interrupción, el usuario del frontend también experimentará una funcionalidad degradada.

En cambio, implementamos nuestros BFF como servicios autónomos que eliminan las dependencias síncronas entre servicios. Estos servicios no envuelven otros servicios. Son dueños de todos los recursos que necesitan para continuar operando cuando los servicios relacionados están caídos.

Disección del patrón backend for frontend

A nivel macro, un subsistema autónomo se compone de muchos de estos servicios autónomos, por lo que es importante que todos tengan una estructura y apariencia similar. Esto permitirá que los desarrolladores trabajen fácilmente en diferentes servicios. Para ayudar a lograrlo, es buena práctica iniciar un nuevo servicio con un esqueleto a partir de una plantilla estándar, levantarlo y luego añadir la funcionalidad específica.

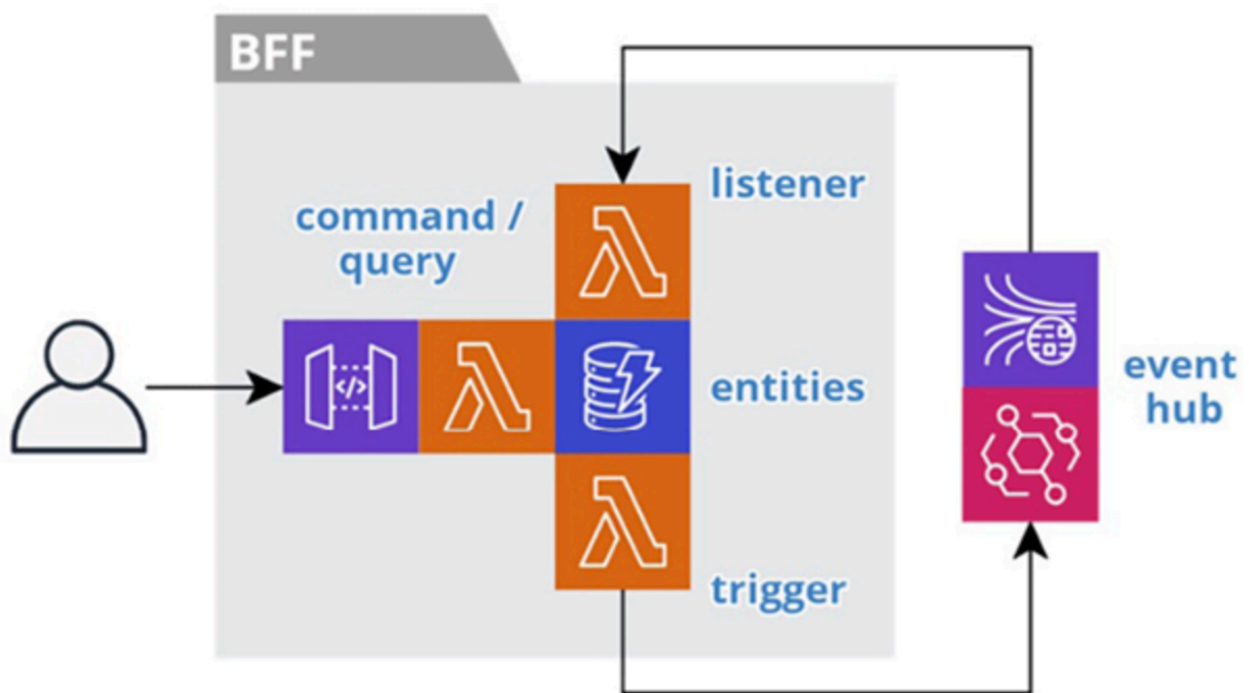

Figure 6.1: BFF pattern

Lo primero que hay que notar es que el servicio BFF típico tiene una API trilateral:

1. Tiene una interfaz síncrona de `command` y `query` que da soporte a su frontend.
2. Tiene una interfaz asíncrona de `listener` que consume eventos de dominio de servicios ascendentes a través del `event hub`.
3. Tiene una interfaz asíncrona de `trigger` que produce eventos de dominio hacia el `event hub` para consumo de servicios descendentes.

Almacén de datos

Cada servicio BFF mantiene su propio almacén de datos altamente disponible y completamente gestionado, como AWS DynamoDB o S3. Así es como logran su autonomía. **El almacén de datos almacena en caché entidades de eventos de dominio ascendentes, de modo que los datos están disponibles de inmediato para los usuarios finales.**

Es importante entender que un servicio BFF almacena los datos en un formato que es más adecuado para el consumo por parte del frontend. Esto producirá una experiencia de usuario más responsiva y simplificará el código del frontend al eliminar la necesidad de transformar continuamente los datos.

```
Resources:
    EntitiesTable:
      Type: AWS::DynamoDB::GlobalTable
      Properties:
        TableName: ${self:service}-${opt:stage}-entities
        AttributeDefinitions:
          ...
        KeySchema:
          - AttributeName: pk
            KeyType: HASH
          - AttributeName: sk
            KeyType: RANGE
        GlobalSecondaryIndexes:
          - IndexName: gsi1
            ...
        BillingMode: PAY_PER_REQUEST
        StreamSpecification:
          StreamViewType: NEW_AND_OLD_IMAGES
        TimeToLiveSpecification:
          AttributeName: ttl
          Enabled: true
```

Las características de Change Data Capture (CDC) streaming y Time To Live (TTL) están habilitadas, junto con la facturación bajo demanda.

API gateway

El frontend debe tener acceso al servicio BFF para realizar consultas y ejecutar comandos. El API gateway proporciona esta capacidad. Crea un perímetro seguro y altamente disponible alrededor de la funcionalidad para dar soporte al frontend.

Es importante reconocer que esta interfaz da soporte a un frontend específico y solo a ese frontend. **Otros sistemas no deberían llamar a esta interfaz, ya que son actores diferentes.**

Funciones de consulta y comando

El trabajo más importante de un servicio BFF es proporcionar a un actor humano la capacidad de ejecutar un conjunto específico de consultas y/o comandos. Recuperan los datos solicitados del almacén de datos y registran los resultados de las actividades del usuario en el almacén de datos. **Implementamos esta funcionalidad en una o más funciones, usando la característica Function-as-a-Service (FaaS) del proveedor de nube, como AWS Lambda.**

Una pregunta común respecto a FaaS es si usar muchas funciones de grano fino o menos funciones de grano grueso. Nuestros servicios BFF enfocados en actividades de usuario ya son bastante granulares, así que **prefiero comenzar con una sola función que ejecute todas las consultas y comandos del BFF**.

Para un servicio basado en GraphQL, esto encaja naturalmente con el popular framework Apollo. **Para un servicio basado en REST, eventualmente podría separar una función para consultas y otra para comandos si los volúmenes de tráfico son suficientemente diferentes.**

Funciones listener

La función `listener` desempeña un rol crucial en un servicio autónomo al implementar el patrón CQRS. Es un stream processor que consume eventos de dominio del event hub y materializa entidades en el almacén de datos para dar soporte a las consultas.

Es una sola función con el único propósito de materializar entidades, pero puede haber muchos tipos diferentes de eventos de dominio que procesar. La función aprovecha la característica de pipeline del framework de procesamiento de streams para preservar la mantenibilidad de la lógica para los tipos de eventos individuales.

Función trigger

La función trigger cumple el rol fundamental en un servicio autónomo de implementar la variante database-first del patrón Event Sourcing. Es un stream processor que consume eventos de cambio del stream CDC del almacén de datos y publica eventos de dominio al event hub para consumo de servicios descendentes.

Es una sola función con el único propósito de publicar eventos de dominio después de que un usuario ejecuta un comando, pero puede haber muchos tipos diferentes de eventos de cambio que procesar.

Las funciones trigger son opcionales. Si un servicio BFF es de solo lectura (es decir, no tiene comandos), entonces no necesita una función trigger.

Disección de la nano arquitectura a nivel de función

Este nivel de arquitectura describe cómo creamos una estructura limpia dentro de una función dada para que todas las dependencias internas sean acíclicas y desacoplemos la lógica de negocio de las dependencias externas.

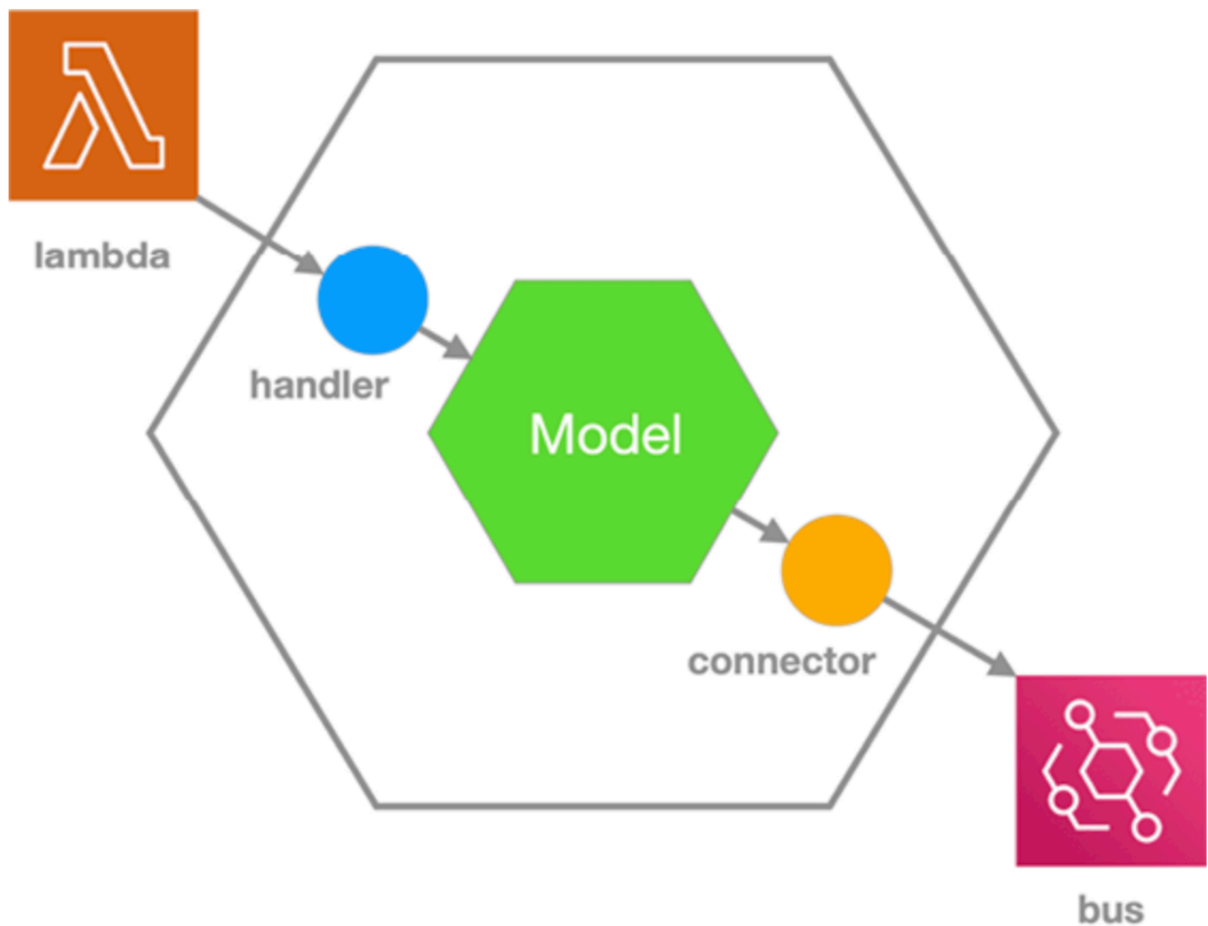

Figure 2.11: Function-level – nano hexagonal architecture

Por lo tanto, implementamos la lógica de negocio en clases Model, envolvemos todas las llamadas a recursos en clases Connector y ocultamos los detalles del entorno de ejecución en los handlers. Esto hace más que hacer el código más portable; **lo más importante, lo hace más testeable.**

Un servicio BFF trabaja con una pequeña porción del modelo de dominio lógico global del sistema. Cada clase Model representa un agregado de dominio separado utilizado en el servicio BFF.

Un servicio BFF almacena las entidades de dominio en un formato más adecuado para el frontend. Los modelos proporcionan funciones mapper, que transforman los datos entre los formatos de datos internos y externos.

Connectors

Una clase Model tiene un constructor para inyectar connectors. La clase Model dicta las firmas de los métodos del Connector, y el connector encapsula los detalles de la interacción con el recurso.

Los connectors también facilitan la escritura de pruebas unitarias que se ejecutan de forma aislada, porque es más fácil escribir mocks para connectors que escribir mocks para recursos.

Cada connector define un constructor con opciones de configuración. Esto proporciona una separación limpia entre el connector y el entorno de ejecución. Por ejemplo, en tiempo de ejecución, una función recibirá información de conexión en variables de entorno, mientras que las pruebas unitarias simplemente asignarán valores de prueba.

Handlers

Los modelos son agnósticos al entorno de ejecución. En cambio, usamos un handler para envolver y adaptar un modelo al entorno dado.

Realizamos la inyección de dependencias en los handlers. Usamos inyección de dependencias basada en constructor porque es simple y ligera y, por lo tanto, no tiene un impacto negativo en los tiempos de cold-start.

Deberíamos poder implementar un servicio BFF usando REST o GraphQL sin tener que cambiar las clases Model y Connector.

Elección entre REST y GraphQL

La respuesta corta es que depende de la actividad del usuario. **¿La actividad de usuario involucra muchos usuarios interactuando con los mismos datos o pocos usuarios interactuando con datos diferentes?**

Representational State Transfer (REST) fue diseñado para aprovechar al máximo la infraestructura que compone internet, como routers y redes de distribución de contenido. Así que tiene sentido usar REST para un servicio de menú orientado al cliente. (Muchos usuarios interactuando con los mismos datos)

GraphQL, por otro lado, típicamente usa la operación POST, así que no es muy adecuado para escenarios de solo lectura. Sin embargo, es muy adecuado para actividades más ad hoc donde el usuario navega y edita datos.

La compatibilidad hacia atrás es un área donde GraphQL sobresale porque los clientes declaran los campos exactos que necesitan. Esto facilita rastrear cómo los clientes usan la interfaz y elaborar cambios compatibles hacia atrás.

Una ventaja importante de GraphQL es que su enfoque de consultas dirigidas por el cliente tiende a ser menos verboso que REST. Una interfaz RESTful típica requerirá que una aplicación haga varias llamadas para recuperar un agregado de dominio completo.

Mi directriz general es usar REST para servicios BFF y GraphQL para Open APIs. A veces, puede ser necesario soportar ambos.

REST

REST es la elección tradicional para implementar servicios. Como hemos discutido, es mejor para servicios de solo lectura, webhooks y servicios BFF.

La función route tiene acceso a los modelos a través del objeto request. El modelo maneja la lógica de negocio, y el connector interactúa con el almacén de datos. Esto significa que la función route solo necesita implementar el código de enlace para mapear las entradas del objeto request al modelo y mapear la salida del modelo al objeto response.

GraphQL

GraphQL es una opción popular para implementar servicios. Como hemos discutido, es mejor para implementar interfaces abiertas, porque las consultas dirigidas por el cliente son naturalmente compatibles hacia atrás y se autodocumentan.

Hay una ruta que dirige las solicitudes POST del path graphql a la función que implementa un esquema GraphQL. Este esquema corresponde a la clase Model que definimos en la sección Modelos. Típicamente habrá una relación uno a uno entre fragmentos de esquema y clases Model.

El modelo maneja la lógica de negocio, y el connector interactúa con el almacén de datos. Esto significa que el resolver solo necesita implementar el código de enlace para mapear las entradas de la consulta al modelo y devolver la salida del modelo.

Implementación de diferentes tipos de servicios BFF

Veamos algunos tipos diferentes de servicios BFF que atienden las distintas fases del ciclo de vida de los datos.

Servicios BFF CRUD

Estos servicios BFF nos permiten Crear, Leer, Actualizar y Eliminar (CRUD) datos. Soportan actividades de usuario en la fase de Creación del ciclo de vida de los datos.

A menudo nos referimos a los datos en estos servicios como datos de referencia o datos maestros que soportan todos los procesos de negocio descendentes. También nos referimos a estos servicios como el sistema de registro para estos datos. También podemos pensar en esto como datos lentos porque cambian con poca frecuencia.

Servicios BFF de lista de valores (Lov)

Un servicio BFF Lov es una variante del patrón de servicio BFF que proporciona acceso eficiente y de solo lectura a datos que necesitamos para soportar actividades de usuario de entrada de datos.

Por ejemplo, un formulario de entrada de datos usualmente tiene listas desplegables que dan al usuario los valores válidos para un campo dado en el formulario. La mayoría de estas listas son estáticas, pero muchas listas necesitan acceso a datos dinámicos.

Los datos dinámicos fueron creados en un servicio BFF ascendente en una actividad de usuario separada que optimizamos para la escritura de datos. Ahora estamos usando los datos en una actividad de usuario descendente diferente, y queremos acceder a los datos desde un servicio BFF que optimizamos para este tipo de lecturas.

Un servicio Lov es un BFF de solo lectura, así que no tiene funciones de comando ni trigger. Solo tiene una función listener y una función de consulta. La función listener consume tipos de eventos created, updated y deleted para varios tipos de entidades de dominio y materializa los datos en la tabla de entidades.

Solo almacenamos los datos que necesitamos para poblar las listas desplegables, como un código/identificador y una descripción legible por humanos, y quizás algunos indicadores que podemos usar para filtrar las listas.

Siguiendo el patrón de Single-Table design, usaremos el código como pk, y estableceremos el campo discriminador en cada fila para identificar el tipo de entidad de dominio. La función de consulta usará un índice sobre el discriminador para que podamos acceder fácil y eficientemente a la lista de valores para un tipo de entidad de dominio específico.

Cada subsistema autónomo típicamente tendrá un único servicio Lov. Estos servicios simplifican el frontend porque podemos recuperar todos los datos de referencia necesarios desde un único servicio al inicio y almacenarlos en caché para que estén disponibles offline.

Servicios BFF de tarea

Un servicio BFF de tarea es una variante del patrón de servicio BFF que soporta un paso (es decir, una tarea) en un proceso de negocio más amplio.

Las tareas son los caballos de batalla de los servicios BFF, ya que constituyen la mayoría de los servicios BFF. Un BFF de tarea proporciona una consulta bien definida para acceder a los datos que el usuario necesita para realizar la tarea específica. También proporciona el conjunto de comandos que el usuario necesita para completar la tarea, incluyendo un comando para señalar cuándo la tarea está completa.

Un BFF de tarea implementa un paso en un proceso de negocio más amplio. La función listener consume eventos de dominio que inician las tareas. Materializa los datos para que los usuarios puedan consultar su trabajo. Estos eventos de dominio pueden provenir de servicios de control que orquestan los procesos de negocio o los servicios individuales pueden coreografiar los procesos de negocio por sí mismos.

La función trigger produce eventos de dominio para registrar el resultado de cualquier comando intermedio, y en última instancia produce el evento de dominio que señala la finalización de una tarea y hace avanzar el proceso de negocio más amplio.

Servicios BFF de búsqueda

Un servicio BFF de búsqueda es una variante del patrón de servicio BFF que proporciona acceso de solo lectura a los datos producidos por uno o más servicios ascendentes.

Mientras que un servicio BFF de tarea proporciona sus propias consultas bien definidas, un servicio BFF de búsqueda ayuda a los usuarios a acotar lo que están buscando. Los usuarios pueden no saber exactamente qué están buscando o incluso si la información existe. Pueden tener solo una idea general de lo que quieren y están buscando ver qué hay disponible.

En el corazón de un servicio de búsqueda está un índice, como Elasticsearch. La interfaz de búsqueda proporciona acceso al índice a través de un API gateway y una función de búsqueda. Cada resultado de búsqueda contiene información resumida y la URL para acceder a la información detallada de la entidad.

Los datos de las entidades de dominio son el candidato perfecto para almacenamiento de objetos, como un bucket de AWS S3, porque el volumen de datos es ilimitado y es de solo lectura, razonablemente estático, y solo se accede por el identificador.

Almacenamos los datos en formato JSON, que devolveremos al cliente, junto con cualquier imagen. Las claves de objeto deben seguir una buena estructura RESTful, como `/restaurants/123/menuitems/987`.

Para información pública, servimos las entidades de dominio desde el borde de la nube usando una Red de Distribución de Contenido (CDN), como AWS CloudFront. Logramos esto estableciendo la propiedad cache-control en los objetos en almacenamiento para que podamos cachear las solicitudes en el navegador y la CDN. **Esto proporciona la baja latencia y la escala masiva necesarias para soportar un número ilimitado de clientes.**

La función listener es responsable de materializar las entidades e imágenes en el bucket S3. Luego, la función trigger indexa los datos, solo después de que estén disponibles en el bucket. Encadenar estos pasos es importante porque cada paso es atómico. **Separar estos pasos asegura que el índice no devolverá un resultado con un enlace roto.**

Servicio BFF de acción

Un servicio BFF de acción es una variante del patrón de servicio BFF que realiza una acción específica en el contexto de otro servicio BFF.

Un servicio de acción es responsable de realizar un conjunto de comandos. Podemos pensar en estos servicios como headless porque necesitan que otros servicios proporcionen el contexto para las entradas de un comando. En otras palabras, **pasamos toda la información necesaria a un comando.** No tienen un listener para crear vistas materializadas que el servicio pueda consultar para proporcionar su propio contexto.

Un servicio de acción puede tener una consulta para devolver los resultados de sus propias acciones, pero el servicio no retiene esta información por más tiempo del necesario. Finalmente, una función trigger produce eventos de dominio para registrar el resultado de los comandos.

Servicios BFF de dashboard

Un servicio BFF de dashboard es una variante del patrón de servicio BFF que proporciona a los usuarios un conteo actualizado y de solo lectura de valores de dominio importantes.

Estas actividades de usuario son típicamente orientadas al exterior y, por lo tanto, requieren tiempos de respuesta rápidos y alta disponibilidad. Estos requisitos, combinados con el hecho de que la experiencia de usuario no es ad hoc, significan que **estos servicios no necesitan y no deberían depender de infraestructura compleja de analítica**.

En cambio, un servicio BFF de dashboard trabaja con un servicio de control que aprovecha las transacciones ACID 2.0 para realizar los cálculos necesarios. El servicio de control consume eventos de dominio de orden inferior, calcula el valor actual y produce un evento de dominio de orden superior, como `balance-updated`.

Esto deja al servicio BFF con la simple responsabilidad de entregar la información de solo lectura al usuario. Las actividades de usuario de dashboard tienen un patrón de acceso bien definido.

La función listener consume los eventos de orden superior y materializa los valores. La consulta simplemente devuelve el conjunto de valores precalculados para el usuario actual. Como el servicio de dashboard es autónomo, siempre puede devolver

los valores precalculados más recientemente recibidos.

Servicios BFF de reportes

Un servicio BFF de reportes es una variante del patrón de servicio BFF que proporciona a los usuarios una vista histórica de los datos en forma de reportes periódicos.

Esta es una actividad de usuario diferente, para un actor diferente, en una fase posterior del ciclo de vida de los datos. Así que creamos servicios de reportes descendentes para estas actividades.

De forma periódica, un actor espera tener acceso a reportes, como un reporte mensual sobre su área de responsabilidad. Podemos preparar continuamente los reportes a medida que fluyen nuevos datos para que estén listos en la frecuencia deseada.

Organizamos, versionamos y almacenamos los reportes generados en un bucket S3 con una estructura de claves apropiada, como /division/yyyy/mm/dd/report-name.pdf. La interfaz de consulta permite al usuario explorar los reportes.

1. Primero, consumimos eventos de dominio a través de un canal, como AWS Firehose, que almacena los eventos en S3 en un formato adecuado.
2. Luego, la función trigger invoca un servicio, como AWS Athena, para consultar los eventos y escribir los resultados de vuelta al bucket.
3. Esto a su vez hace que la función trigger genere el reporte a partir de los resultados de la consulta y lo almacene en el bucket.
4. Finalmente, en cualquier momento, el usuario puede ver la última versión de un reporte con los datos más recientes.

Los eventos en el bucket tienen un TTL que es lo suficientemente corto para controlar el tamaño del bucket pero lo suficientemente largo para que los datos que llegan tarde generen una nueva versión del reporte. Podemos retener los reportes indefinidamente y envejecerlos hacia almacenamiento en frío.

Servicios BFF de archivo

Un servicio BFF de archivo es una variante del patrón de servicio BFF que sirve como el recolector de datos definitivo al final del ciclo de vida de los datos. **Estos servicios están gobernados por requisitos de gestión de registros que especifican cuánto tiempo una empresa debe retener diversas piezas de datos.**

Los servicios establecen un TTL en cada registro para eliminar los datos cuando ya no los necesitan. Los servicios BFF de archivo recopilan todos los eventos, lo que permite que otros servicios se mantengan ligeros.

Los servicios BFF de archivo son proactivos porque no esperan a recopilar datos al final del ciclo de vida. Recopilan datos continuamente desde el momento en que los usuarios los crean para que los servicios ascendentes puedan eliminar datos libremente.

La función listener consume eventos de dominio y almacena los datos en un bucket S3. Esto es similar a un event lake, pero organizamos los datos alrededor de un identificador de entidad de dominio específico, como cliente, restaurante o conductor. Se acumula un historial detallado que podemos usar en el futuro para auditoría y restauración. La interfaz de consulta permite a un usuario explorar el archivo. Los datos envejecen hacia almacenamiento en frío y los purgamos después de un período de tiempo regulado.

Para poner el servicio de archivo en contexto, podemos usar un sistema de gestión de casos como ejemplo. Un servicio BFF de tarea es responsable de los casos activos. Mientras un caso está activo, el índice en el servicio BFF de búsqueda apunta a los datos en el servicio BFF de tarea. Cuando un oficial de casos cierra un caso, el servicio BFF de tarea elimina su copia de los datos del caso, y actualizamos el índice de búsqueda para que apunte a los datos del caso que se han acumulado en el servicio BFF de archivo. Si necesitamos reabrir un caso, entonces restauramos los datos del caso desde el archivo al servicio BFF de tarea y actualizamos el índice de búsqueda.

Cada subsistema autónomo debería tener un servicio de archivo para sus agregados de dominio para que sus servicios se mantengan ligeros. Si un subsistema tiene muchos agregados de dominio, entonces puede justificar múltiples archivos. En su conjunto, un sistema debería tener un subsistema de archivo al final del ciclo de vida de los datos que sea responsable ante el actor regulatorio. Esto permite que los demás subsistemas autónomos se mantengan ligeros.