TALLER 2

Santiago Yañez Barajas - 20477648

Alejandro Enrique Suárez - 20472591

Hermann David Hernandez - 20469294

PRESENTADO A:

Andrés Armando Sánchez Martín

ARQUITECTURA DE SOFTWARE

PONTIFICIA UNIVERSIDAD JAVERIANA

BOGOTÁ D.C

2024

CONTENIDO

URL TAG repositorios git públicos del Taller

https://github.com/suaracost/FrontCQRS/releases/tag/Entrega_Taller_2

https://github.com/suaracost/RailsAPI/releases/tag/Entrega_Taller_2

https://github.com/SantiagoYB/RailsAPIWrite/releases/tag/Entrega_Taller_2

Introducción

Para el desarrollo de este taller se hará mediante componentes dados por el profesor donde el objetivo es implementar integrar diversas tecnologías utilizadas en la industria actual.

En este taller, vamos a trabajar con un ecosistema tecnológico que abarca desde bases de datos NoSQL hasta frameworks de desarrollo web y tecnologías móviles, todo orquestado mediante contenedores para facilitar la gestión y el despliegue de la aplicación. Además, adoptaremos la arquitectura **CQRS** (Command Query Responsibility Segregation), la cual nos permitirá manejar de manera más eficiente la separación de comandos y consultas en nuestra aplicación, optimizando el manejo de los datos y mejorando el rendimiento.

Componentes de la arquitectura:

1. **Base de Datos en Cassandra**:
   Utilizaremos Apache Cassandra, una base de datos NoSQL altamente escalable que permite la replicación de datos y garantiza una alta disponibilidad.
2. **Backend en Ruby on Rails**:
   El backend será implementado en Ruby on Rails, un framework ágil que facilita el desarrollo de aplicaciones web y APIs. En el contexto de CQRS, Ruby on Rails gestionará los comandos (operaciones de escritura y actualización) y se integrará con otras capas del sistema para asegurar una correcta segregación de responsabilidades entre el procesamiento de comandos y las consultas.
3. **Conexión Back-Front con GraphQL**:
   Para la comunicación entre el backend y el frontend, utilizaremos GraphQL, que permitirá gestionar las consultas de manera eficiente, optimizando la forma en que los datos se solicitan y se envían entre el frontend y el backend. Con GraphQL, solo se obtendrán los datos necesarios, lo que minimiza el tráfico entre capas y mejora el rendimiento del sistema, sobre todo cuando se integra con CQRS para manejar las peticiones de lectura.

4. **Frontend en Android**:
   El frontend estará desarrollado en Android, permitiendo que los usuarios interactúen con la aplicación desde sus dispositivos móviles. La aplicación móvil enviará tanto comandos (por ejemplo, para actualizaciones o inserciones de datos) como consultas al backend a través de la arquitectura CQRS, diferenciando claramente las responsabilidades entre lectura y escritura para optimizar la experiencia del usuario.
5. **Uso de Docker**:
   Para garantizar un entorno de desarrollo consistente y replicable, emplearemos Docker. Cada componente del sistema será desplegado en contenedores separados, facilitando no solo el desarrollo colaborativo, sino también la escalabilidad y la gestión del entorno en producción. Docker es especialmente útil en arquitecturas distribuidas, como la que implementamos con CQRS, permitiendo una fácil administración y orquestación de los distintos servicios involucrados.
6. **Arquitectura CQRS (Command Query Responsibility Segregation)**:
   Uno de los enfoques más importantes de este taller es la implementación de la arquitectura CQRS. En esta arquitectura, separamos las responsabilidades de comandos (acciones que modifican el estado de los datos) y consultas (solicitudes que leen los datos). Esto nos permite optimizar ambos tipos de operaciones, enfocándonos en la eficiencia de las consultas y la consistencia de los comandos, utilizando tecnologías específicas para cada caso. Por ejemplo:
   - **Comandos**: Se manejarán a través de servicios backend en Rails que procesan y validan la lógica de negocio antes de modificar los datos.
   - **Consultas**: Utilizaremos Cassandra y GraphQL para entregar resultados rápidos y escalables a las solicitudes de lectura de datos, optimizando el rendimiento y reduciendo la complejidad en las consultas.

Patrón CQRS

- ## Definición

El patrón CQRS (Command Query Responsibility Segregation) es una estrategia arquitectónica que separa las operaciones de lectura (consultas) de las de escritura (comandos), permitiendo manejar cada una de forma independiente. Esto significa que los modelos de datos y la lógica de negocio utilizados para consultar información pueden ser completamente diferentes de aquellos que se usan para modificarla. Este enfoque resulta útil en sistemas complejos o distribuidos, donde las necesidades de lectura y escritura pueden ser dispares y deben ser

optimizadas por separado. A través de CQRS, se mejora el rendimiento y la escalabilidad de los sistemas.

- ## Características

Una de las características clave de CQRS es la separación de las responsabilidades entre las operaciones de comando y consulta. Las consultas son operaciones que no modifican el estado del sistema y se centran en devolver datos, mientras que los comandos son responsables de cambiar el estado. CQRS permite que cada operación use diferentes modelos de datos, optimizando los procesos de lectura y escritura por separado. Además, CQRS es frecuentemente combinado con Event Sourcing, lo que permite un registro detallado de todos los cambios en el estado de la aplicación, lo que a su vez mejora la trazabilidad. Esto facilita también la escalabilidad individual de las consultas y comandos, ya que cada lado puede ser adaptado a sus propios requerimientos tecnológicos sin interferir con el otro.

- ## Historia y evolución

El concepto de CQRS tiene sus raíces en el patrón Command Query Separation (CQS), propuesto por Bertrand Meyer, que promovía la idea de que los métodos en un sistema deberían ser o comandos que modifican el estado, o consultas que devuelven datos, pero no ambos. CQRS tomó esta idea y la extendió al nivel arquitectónico en sistemas distribuidos. Aunque fue popularizado por Martin Fowler y Greg Young en la década de 2010, la adopción de CQRS ha crecido principalmente en entornos donde el rendimiento y la gestión eficiente de grandes volúmenes de datos son esenciales. La evolución del patrón ha sido influenciada por la necesidad de manejar las complejidades del mundo digital moderno, donde la lectura y escritura de datos tienen diferentes necesidades de optimización.

- ## Ventajas y desventajas

Ventajas:

- Escalabilidad: Permite escalar de manera independiente las operaciones de lectura y escritura, lo que optimiza el uso de recursos según las necesidades del sistema.
- Separación de responsabilidades: La división entre comandos y consultas simplifica el desarrollo y mantenimiento, ya que cada uno puede evolucionar de forma independiente.
- Optimización específica: Facilita el uso de diferentes tecnologías o modelos de datos para optimizar el rendimiento de lectura y escritura.

- Compatibilidad con Event Sourcing: CQRS se complementa bien con Event Sourcing, lo que mejora la trazabilidad de los cambios en el sistema.

Desventajas:

- Complejidad adicional: Introduce una mayor complejidad en la arquitectura, lo que puede ser innecesario para sistemas más simples o de menor tamaño.
- Dificultad de sincronización: La sincronización entre los modelos de lectura y escritura puede ser complicada, especialmente si se utilizan tecnologías diferentes.
- Coste de implementación: Requiere más esfuerzo y recursos para su implementación en comparación con arquitecturas más tradicionales como CRUD.
- No siempre necesario: En sistemas simples o de baja carga, la adopción de CQRS puede ser excesiva, añadiendo complejidad sin proporcionar beneficios claros.

● Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

El patrón CQRS es ideal en situaciones donde las operaciones de lectura y escritura tienen características significativamente diferentes. Por ejemplo, es común en aplicaciones con más consultas que comandos, como en plataformas de comercio electrónico donde los usuarios consultan constantemente productos, pero el número de transacciones o modificaciones es menor. También es adecuado para aplicaciones que requieren alta disponibilidad y velocidad de respuesta en las consultas, como en los sistemas de seguimiento en tiempo real o plataformas de análisis de datos. Además, CQRS es útil cuando se requiere un control estricto sobre el acceso a datos para la lectura y la escritura, garantizando que los procesos se mantengan separados y optimizados.

● Casos de aplicación (Ejemplos y casos de éxito en la industria)

En aplicaciones como plataformas de comercio electrónico, CQRS se usa para manejar grandes volúmenes de consultas, como búsquedas de productos, y mantener separadas las operaciones de escritura, como compras o actualizaciones de perfil. Esto permite escalar eficientemente y optimizar el rendimiento, asegurando una experiencia de usuario fluida en entornos de alto tráfico donde las lecturas son mucho más frecuentes que las escrituras.

- Amazon es una de las mayores plataformas de comercio electrónico del mundo, conocida por manejar millones de transacciones diarias y ofrecer una amplia gama de productos. Amazon utiliza CQRS para manejar eficientemente las numerosas consultas

que los usuarios realizan cuando buscan productos y revisiones. Al separar las lecturas de las escrituras, como en el caso de una compra o actualización de perfil, Amazon optimiza el rendimiento y asegura que las operaciones de alta demanda no afecten la velocidad y disponibilidad de las escrituras.

- eBay es una plataforma de subastas y comercio en línea donde los usuarios pueden comprar y vender productos de segunda mano o nuevos. Dada la gran cantidad de consultas que se generan al buscar artículos, junto con las transacciones (pujas, ventas), eBay implementa CQRS para escalar de forma eficiente. Las operaciones de lectura, como la búsqueda de productos, están optimizadas para manejar el alto tráfico, mientras que las operaciones de escritura, como las pujas o compras, son gestionadas de manera separada para evitar interferencias.

Android

- ### Definición

Android es un sistema operativo diseñado principalmente para teléfonos móviles, aunque también se utiliza en otros dispositivos como tabletas, relojes inteligentes, televisores y automóviles. Lo que lo diferencia de otros sistemas operativos es que está basado en el núcleo Linux, un sistema operativo libre, gratuito y multiplataforma.

Una de las características más destacadas de Android es que permite desarrollar aplicaciones de manera sencilla mediante el uso de una variante de Java llamada Dalvik. El sistema operativo proporciona todas las interfaces necesarias para que los desarrolladores puedan acceder de manera eficiente a las funciones del dispositivo. Además, al ser un sistema operativo completamente libre, ni los fabricantes de dispositivos ni los desarrolladores de aplicaciones deben pagar licencias para usarlo o incluirlo en sus productos. Esto contribuye a que los costos para lanzar un teléfono o desarrollar aplicaciones sean relativamente bajos.

Android está revolucionando el uso de los teléfonos inteligentes y ha ampliado su presencia en una variedad de dispositivos. Actualmente, se utiliza en más de 2,500 millones de dispositivos activos en todo el mundo, consolidándose como una de las plataformas tecnológicas más influyentes.

- ### Características
    - Android está disponible para desarrolladores, diseñadores y fabricantes de dispositivos, permitiendo que más personas puedan experimentar, imaginar y crear soluciones innovadoras.

- Núcleo basado en Kernel de Linux.
- Utiliza SQLite para el almacenamiento de datos.
- Android ofrece protección desde el primer uso del dispositivo. Google Play Protect analiza todas las aplicaciones, el software recibe actualizaciones de seguridad regulares, y la plataforma se mejora de manera constante. El teléfono cuenta con herramientas de seguridad integradas.
- Incluye un emulador de dispositivos, herramientas para depuración de memoria y análisis del rendimiento de software.
- Los usuarios de Android tienen control sobre sus datos. Pueden decidir si comparten información como la actividad en la web, en aplicaciones, o el historial de ubicaciones. Además, si una aplicación accede a la ubicación cuando no está en uso, el usuario recibirá una notificación. Las opciones de privacidad están fácilmente accesibles para modificar los permisos.
- Navegador web basado en WebKit incluido.
- Android ofrece herramientas de bienestar digital, permitiendo al usuario monitorear el uso del teléfono y decidir qué funciones se adaptan mejor a sus necesidades, como desconectar, eliminar distracciones o activar el modo descanso, todo mediante un panel de control dedicado.

Historia y evolución

Android, cuyo nombre proviene del apodo que su creador, Andy Rubin, recibió durante su tiempo en Apple debido a su afición por los robots, existe oficialmente desde 2008 y es desarrollado por Google. El primer momento en que se comenzó a hablar de Android fue a través de un tweet realizado por Rubin, lo que despertó el interés en la comunidad tecnológica sobre este nuevo sistema operativo.

Android fue fundada en octubre de 2003 por Andy Rubin, Rich Miner, Nick Sears y Chris White. Inicialmente, fue concebido como un sistema operativo avanzado para cámaras digitales, pero rápidamente cambió su enfoque hacia los teléfonos inteligentes, al reconocer el potencial de

este mercado en expansión. En 2005, Google adquirió Android Inc. por 50 millones de dólares, lo que marcó el comienzo del desarrollo de un sistema operativo móvil de código abierto basado en Linux, con el objetivo de competir directamente con otros sistemas operativos móviles como Symbian y Windows Mobile.

El primer dispositivo Android fue el HTC Dream, lanzado en octubre de 2008. Este teléfono presentaba una pantalla táctil y un teclado físico deslizable, siendo el primero en introducir al mercado la interfaz de usuario de Android y su ecosistema de aplicaciones.

Las primeras versiones de Android (1.0 y 1.1) sentaron las bases del sistema operativo, pero fue con Android 1.5 Cupcake, lanzado en abril de 2009, cuando se empezaron a introducir funciones clave como el teclado en pantalla y la posibilidad de subir videos directamente a YouTube. Android 2.0 Eclair, lanzado en octubre de 2009, incorporó la navegación GPS y la sincronización de cuentas, mientras que Android 2.2 Froyo, en mayo de 2010, mejoró la velocidad del sistema y añadió soporte para Adobe Flash.

En 2011, Android 3.0 Honeycomb fue diseñado exclusivamente para tablets, con una interfaz adaptada a pantallas más grandes. Más tarde, Android 4.0 Ice Cream Sandwich unificó las versiones de tabletas y teléfonos, presentando un diseño más moderno y coherente. En 2013, Android 4.4 KitKat optimizó el sistema para dispositivos de gama baja, mejorando la experiencia de usuario y añadiendo funcionalidades como Google Now y búsqueda por voz.

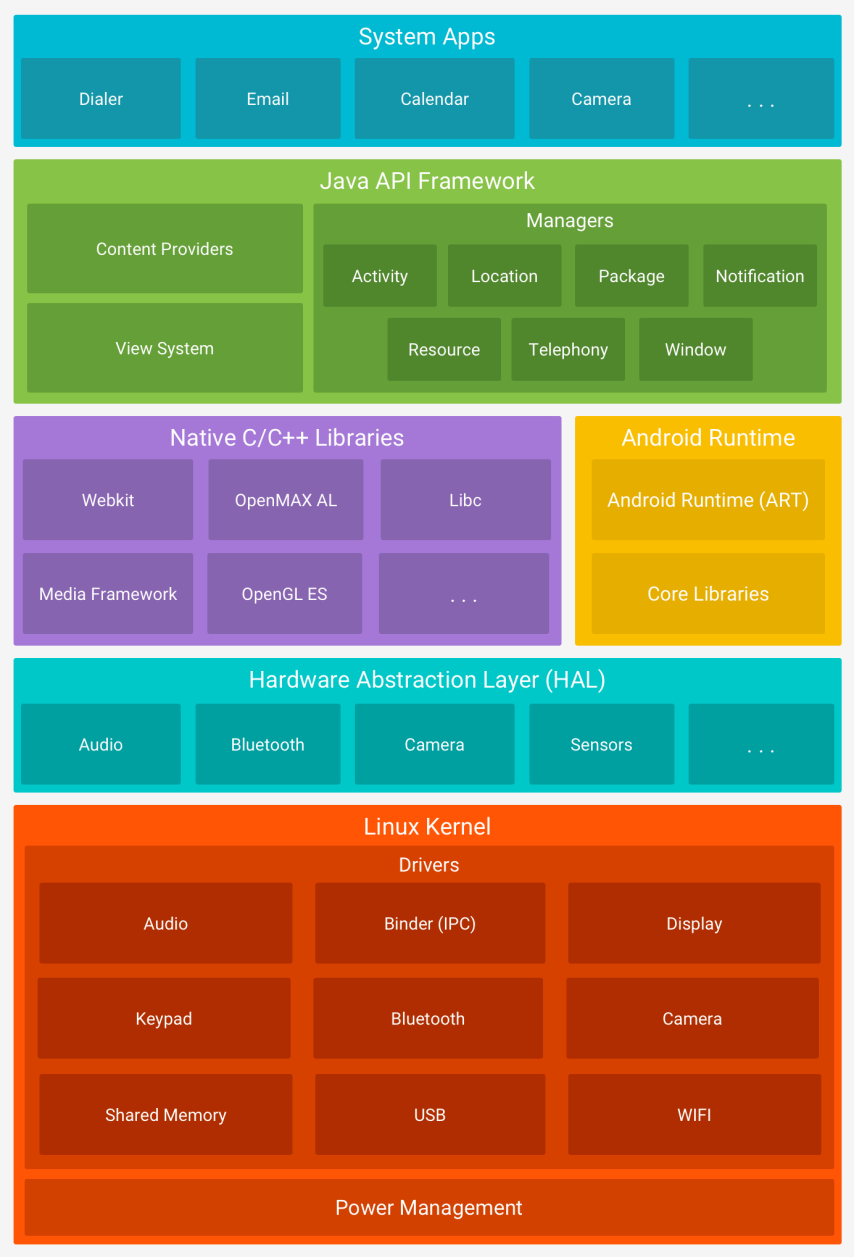
Android 5.0 Lollipop, lanzado en 2014, trajo el rediseño de Material Design, una interfaz más visual y colorida. Android 6.0 Marshmallow, en 2015, mejoró la gestión de permisos y la duración de la batería. En 2016, Android 7.0 Nougat introdujo la pantalla dividida y mejoras en las notificaciones, permitiendo respuestas rápidas desde el panel.

Android 8.0 Oreo, lanzado en 2017, mejoró el rendimiento y la seguridad, además de añadir el modo Picture-in-Picture. En 2018, Android 9.0 Pie se enfocó en la inteligencia artificial para

optimizar la batería y las recomendaciones de aplicaciones. Android 10, en 2019, trajo el modo oscuro y nuevas opciones de privacidad, mientras que Android 11, en 2020, mejoró la gestión de conversaciones y dispositivos conectados. Android 12, lanzado en 2021, introdujo el rediseño "Material You" para una personalización más profunda, y Android 13, en 2022, añadió configuraciones de idioma por aplicación y mejoras en la privacidad.

En 2023, Android 14 llegó con mejoras en la duración de la batería, mayor seguridad, mejor multitarea y una integración más fluida con dispositivos portátiles y sistemas de hogar inteligente. Con vistas al futuro, se espera que Android siga enfocándose en la inteligencia artificial, la personalización y las capacidades que ofrece la conectividad 5G.

Ahora, se presenta la arquitectura que utiliza Android. Esta estructura se compone de varias capas que trabajan juntas para gestionar el hardware, las aplicaciones y los servicios, proporcionando una experiencia de usuario eficiente y adaptable a diferentes tipos de dispositivos y configuraciones.



- **System Apps (Aplicaciones del Sistema):** Estas son las aplicaciones incluidas por defecto en los dispositivos Android, como el marcador de llamadas, el correo electrónico, el calendario y la cámara. Proporcionan funciones esenciales del sistema y sirven como aplicaciones clave para el usuario final.
- **Java API Framework (Marco de Trabajo API de Java):** Este nivel proporciona un conjunto de APIs que los desarrolladores pueden usar para crear aplicaciones. Incluye

servicios clave como el sistema de vistas para manejar la interfaz de usuario, y los "managers" para gestionar actividades, ubicaciones, paquetes, notificaciones, recursos, y servicios de telefonía y ventanas.

- **Native C/C++ Libraries (Bibliotecas Nativas en C/C++):** Estas bibliotecas proporcionan soporte para funciones de bajo nivel, como Webkit para la visualización web, OpenGL ES para gráficos 3D, y Media Framework para gestionar formatos multimedia. Permiten un rendimiento óptimo y acceso a hardware para tareas específicas.
- **Android Runtime (Entorno de Ejecución de Android):** Cada aplicación en Android se ejecuta en su propia instancia de Android Runtime (ART), que actúa de manera similar a la Máquina Virtual de Java (JVM). Este entorno proporciona las bibliotecas necesarias para que las aplicaciones funcionen correctamente y de manera aislada.
- **Hardware Abstraction Layer (HAL):** La capa de abstracción de hardware permite que Android acceda a las capacidades específicas del hardware del dispositivo, como el audio, Bluetooth, cámaras y sensores, a través de una interfaz común. Esto facilita que el sistema operativo se ejecute en una variedad de dispositivos con diferentes configuraciones de hardware.
- **Linux Kernel:** Android se basa en el núcleo de Linux, que maneja las interacciones básicas con el hardware. Proporciona controladores (drivers) para dispositivos como el audio, Bluetooth, teclado, cámara, y la administración de energía. Este núcleo se encarga de la gestión de memoria, la comunicación entre procesos (IPC) y el manejo de los recursos de hardware.

● Ventajas y desventajas

Ventajas:

- Las aplicaciones nativas en Android tienen acceso completo a todas las capacidades del dispositivo.
- Ofrece un rendimiento robusto tanto en modo online como offline.
- Las marcas se benefician de tener su aplicación disponible en la tienda, mejorando el reconocimiento de la marca y su visibilidad.

- Android proporciona una interfaz rica en funcionalidad con un excelente rendimiento, incluyendo tiempos de carga más rápidos.
- Android está presente en una amplia gama de dispositivos, desde modelos económicos hasta los más premium. Diferentes fabricantes ofrecen diversas opciones en términos de diseño, características y precios, lo que permite a los usuarios elegir un dispositivo que se ajuste a sus necesidades y presupuesto.
- Una de las características distintivas de Android es su profunda integración con los servicios de Google. Los usuarios pueden sincronizar fácilmente sus cuentas de Gmail, Google Calendar, Google Drive y otros servicios, lo que facilita el acceso a correos electrónicos, calendarios, documentos y fotos en cualquier momento y desde cualquier lugar.
- Google se enfoca en el desarrollo continuo de Android, implementando actualizaciones regulares que mejoran la seguridad, estabilidad y funcionalidad del sistema operativo, garantizando una experiencia optimizada para los usuarios.

Desventajas:

- La opción más costosa implica la necesidad de aprender a usar nuevas herramientas y lenguajes.
- Se requiere una implementación diferente para cada plataforma, con muy poca reutilización de código.
- Los desarrolladores especializados en plataformas nativas suelen ser costosos.
- La fragmentación del sistema operativo sigue siendo uno de los principales desafíos de Android. La diversidad de dispositivos y fabricantes, con diferentes personalizaciones de software y ciclos de actualización, puede generar una experiencia desigual. Mientras algunos dispositivos reciben rápidamente las últimas actualizaciones, otros pueden tardar más tiempo o incluso no recibirlas, lo que puede causar problemas de seguridad y una experiencia de usuario desactualizada en algunos casos.

- **Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)**

Aplicaciones como las redes sociales, entre ellas X, Facebook, TikTok e Instagram, son algunas de las más utilizadas por los usuarios y las que mayor tiempo de retención generan. Estas plataformas permiten la creación de grandes comunidades y tienen un impacto significativo en el estilo de vida de la sociedad, influyendo en la manera en que las personas interactúan y se comunican.

El desarrollo de aplicaciones Fintech también ha crecido notablemente, ofreciendo soluciones tecnológicas para productos y servicios financieros. Estas aplicaciones emplean innovaciones como la tecnología blockchain, utilizada para desarrollar billeteras digitales, mercados o sistemas de custodia de criptodivisas. Ejemplos destacados incluyen Wise, PayPal, Revolut, Robinhood y Nubank, entre otros.

Las aplicaciones de estilo de vida, como las destinadas a encontrar restaurantes, pedir comida a domicilio, organizar música o planificar trayectos, tienen un gran potencial en el mercado. Ejemplos de éxito incluyen Uber, Spotify, Tripadvisor y Booking, que facilitan la vida diaria de los usuarios.

Aplicaciones de utilidad, como despertadores, alarmas y calculadoras, forman parte del conjunto de herramientas que se utilizan con frecuencia en el día a día, brindando soluciones sencillas pero necesarias.

Por otro lado, las aplicaciones de productividad se destacan en entornos de trabajo, ya que, a diferencia de otras que pueden distraer, estas ayudan a realizar tareas de manera más eficiente. Ejemplos como Slack, Todoist, Trello y Evernote permiten a los usuarios gestionar sus tareas y colaborar de forma más efectiva.

Finalmente, las aplicaciones de juegos son uno de los sectores más lucrativos, generando altos ingresos en diversas categorías. Existen desde juegos con publicidad hasta modelos Freemium, en los que los usuarios pueden mejorar su experiencia comprando recursos a medida que avanzan en el juego. Entre los más populares se encuentran Clash of Clans, Candy Crush, Pokémon Go y Clash Royale.

- **Casos de aplicación (Ejemplos y casos de éxito en la industria)**

Además de las populares redes sociales y juegos, Android ofrece herramientas avanzadas enfocadas en la gestión y seguridad de dispositivos para empresas. Android Enterprise facilita

la administración eficiente y segura de dispositivos móviles, brindando soluciones adaptadas a las necesidades corporativas. A continuación, se explicara cómo estas características contribuyen a mejorar la productividad y seguridad en el entorno empresarial.

- **Vaxcare:** VaxCare facilita la vacunación para los profesionales sanitarios mediante la activación automática y herramientas de gestión flexibles. Gracias a Android Enterprise, han podido desarrollar soluciones adaptadas a las necesidades de sus clientes, con herramientas de seguridad y gestión simples y flexibles, mejorando así la eficiencia en la administración de los dispositivos.
- **Schneider Electric:** Schneider Electric emplea perfiles de trabajo y activación automática para mejorar la productividad de sus equipos internacionales. La creación de perfiles de trabajo separados permite a los empleados gestionar de manera eficiente su vida laboral y personal en un mismo dispositivo, optimizando así el uso de los dispositivos móviles en el ámbito laboral.
- **National Australia Bank (NAB):** National Australia Bank ha simplificado la gestión y seguridad de sus dispositivos con la implementación de dispositivos Pixel y Android Enterprise. Esta plataforma es un componente clave en su estrategia de movilidad, ya que ofrece la flexibilidad y seguridad que los equipos de trabajo requieren para operar de manera eficiente.
- **McLaren Racing** utiliza Android Enterprise para desarrollar y gestionar soluciones móviles personalizadas que mejoran la eficiencia operativa de sus equipos. A través de la plataforma, pueden administrar sus dispositivos de manera segura, optimizando el acceso en tiempo real a datos y herramientas críticas durante las carreras. Esto les permite gestionar información vital de forma rápida y precisa, lo que impacta directamente en el rendimiento competitivo de la escudería

GraphQL

- ### Definición

  GraphQL es un lenguaje de consulta para APIs que permite a los clientes solicitar únicamente los datos que necesitan. Fue diseñado para proporcionar mayor flexibilidad y eficiencia en comparación con REST, donde las respuestas están predeterminadas y, a menudo, contienen más información de la necesaria. GraphQL se basa en un sistema de tipos que define los datos disponibles, lo que permite a los desarrolladores crear consultas precisas y adaptables. Su implementación consiste en crear un esquema, que define tipos y campos, y proporciona funciones que resuelven esos tipos.

Adicional a lo anterior, otro aspecto en el que se diferencia GraphQL de REST, es que en vez de organizarse en varios endpoints, GraphQL opera a través de un único endpoint para todas las operaciones de lectura (consultas) y escritura (mutaciones).

- ## Características
    - Tipado estático: utiliza un esquema estrictamente tipado para describir los datos disponibles en una API. Esto asegura que los clientes sepan exactamente qué datos pueden solicitar, lo que facilita la validación y generación de errores antes de ejecutar las consultas.
    - Consultas personalizadas: permite a los clientes definir las estructuras exactas de las respuestas que necesitan, lo que evita el "over-fetching" (cuando se reciben más datos de los necesarios) y el "under-fetching" (cuando se requieren múltiples llamadas para obtener todos los datos).
    - Mutaciones: además de las consultas para obtener datos, también soporta mutaciones, que son operaciones para modificar o escribir datos en el servidor.
    - Evolución sin versiones: los campos obsoletos pueden ser eliminados gradualmente sin interrumpir los clientes existentes. Esto se gestiona mediante la deprecación, permitiendo que las API evolucionen sin necesidad de múltiples versiones.
    - Agregación de datos: permite obtener datos de múltiples fuentes (bases de datos, microservicios, etc.) en una sola consulta, lo que es útil en arquitecturas de microservicios donde.
- ## Historia y evolución

    GraphQL fue desarrollado por Facebook en 2012 para resolver problemas de su app móvil, problemas como: la necesidad de optimizar el tráfico de red y reducir el "over-fetching" con su API REST. Al hacerse público en 2015, rápidamente atrajo la atención de grandes empresas tecnológicas, entre estas GitHub, que implementó su API GraphQL en 2016, lo cual demostró su flexibilidad y facilidad para manejar consultas complejas.

    Desde entonces, ha evolucionado como un estándar global para APIs, con implementaciones en diversos lenguajes y frameworks, tanto así que empresas como Twitter, Shopify y Airbnb también han adoptado GraphQL para optimizar la entrega de datos y mejorar la experiencia del usuario.

- ## Ventajas y desventajas

  Ventajas:

  - Optimización de datos: al permitir que los clientes soliciten solo los datos que necesitan, se reduce significativamente el tráfico de red, lo que mejora la eficiencia en aplicaciones móviles y de baja conectividad.
  - API flexible y evolutiva: permite agregar nuevos campos al esquema sin afectar a los clientes existentes, lo que evita la necesidad de mantener múltiples versiones de la API.
  - Mayor coherencia entre cliente y servidor: al ser fuertemente tipada de GraphQL facilita la generación de documentación automática y la validación de consultas.

  Desventajas:

  - *Consultas complejas pueden sobrecargar el servidor:* Las consultas anidadas o muy complejas pueden resultar en una mayor carga del servidor si no se gestionan correctamente. Esto ocurre cuando los clientes solicitan grandes cantidades de datos o realizan múltiples operaciones simultáneas.
  - Mayor curva de aprendizaje: los equipos acostumbrados a trabajar con REST pueden necesitar tiempo para aprender a modelar los datos y las consultas de manera eficiente en GraphQL.
  - Falta de control granular en caché: Aunque permite obtener datos específicos, carece de un mecanismo de caché nativo como REST, lo que puede aumentar la carga del servidor en algunas situaciones.

  .

- ## Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)
  - **Aplicaciones móviles**: a pesar de ser útil en aplicaciones móviles donde la conectividad es limitada y es importante optimizar el tráfico de red. Permite que los clientes móviles reciban solo los datos necesarios, lo que reduce el uso de ancho de banda.
  - **Microservicios:** Al operar sobre un único endpoint, GraphQL puede unificar datos de varios microservicios en una sola respuesta. Esto simplifica las interacciones entre sistemas distribuidos y mejora el rendimiento general.

- **Aplicaciones con interfaces de usuario ricas:** En aplicaciones donde múltiples componentes necesitan diferentes tipos de datos, GraphQL permite satisfacer todas estas necesidades en una única consulta, mejorando el rendimiento y reduciendo la complejidad del código del cliente.

● Casos de aplicación (Ejemplos y casos de éxito en la industria)

- **GitHub:** La API GraphQL de GitHub permite a los desarrolladores consultar datos específicos sobre usuarios, repositorios y contribuciones en una única petición, en lugar de múltiples llamadas REST. Esto mejora la eficiencia y simplifica las interacciones con la plataforma.
- **Shopify**: Shopify utiliza GraphQL para manejar grandes volúmenes de datos en sus sistemas de comercio electrónico, permitiendo a los desarrolladores crear consultas personalizadas que optimizan el rendimiento en las tiendas en línea.
- **Twitter**: Twitter ha integrado GraphQL en su infraestructura para mejorar la entrega de datos en tiempo real, optimizando el rendimiento en sus aplicaciones móviles y web.

Ruby on rails

● Definición

Ruby on Rails (RoR) es un framework de desarrollo web de código abierto escrito en Ruby. Está diseñado para simplificar la creación de aplicaciones web al proporcionar una estructura predefinida basada en el patrón Modelo-Vista-Controlador (MVC). Rails automatiza tareas comunes en el desarrollo de software, como la gestión de bases de datos, lo que permite a los programadores centrarse en la lógica empresarial. Esta herramienta es ideal para aplicaciones rápidas y eficientes, proporcionando una plataforma sólida para desarrollar tanto proyectos simples como complejas aplicaciones empresariales.

● Características

Ruby on Rails se destaca por su filosofía "Convención sobre configuración", lo que significa que los desarrolladores pueden adherirse a convenciones estándar sin necesidad de configurar extensivamente el entorno de trabajo. Además, implementa el principio DRY (Don’t Repeat Yourself), que fomenta la escritura de código limpio y reutilizable, minimizando la redundancia.

Rails incluye generadores de código que crean rápidamente estructuras comunes, lo que reduce el tiempo necesario para tareas repetitivas. A su vez, la modularidad del framework permite la integración de numerosas bibliotecas, optimizando el desarrollo en diversos escenarios.

● Historia y evolución

Ruby on Rails fue creado por David Heinemeier Hansson en 2003, y su primera versión fue lanzada en 2004. Nació como parte del proyecto Basecamp, con el objetivo de facilitar el desarrollo de aplicaciones web mediante la eliminación de complejidades innecesarias. Rails ha evolucionado desde entonces, incorporando mejoras en el rendimiento, la seguridad y la capacidad de integración con tecnologías emergentes como Hotwire para interactividad en tiempo real. Durante los primeros años de su existencia, Rails ganó gran popularidad gracias a su capacidad para acortar tiempos de desarrollo y su idoneidad para proyectos ágiles, especialmente en startups.

● Ventajas y desventajas

Ventajas:

- **Rapidez en el desarrollo**: RoR permite crear aplicaciones de forma rápida gracias a herramientas como generadores automáticos de código, lo que es ideal para startups y prototipos.
- **Curva de aprendizaje suave**: Su estructura organizada y el uso del patrón MVC facilitan el aprendizaje, incluso para desarrolladores nuevos.
- **Comunidad activa**: Una gran cantidad de recursos, bibliotecas (gems), y foros de discusión están disponibles, lo que mejora el soporte y la productividad.
- **Convención sobre configuración**: Simplifica el desarrollo siguiendo convenciones predefinidas que evitan la necesidad de configuraciones complejas.
- **Código limpio y mantenible**: Gracias al principio DRY (Don't Repeat Yourself), el código es menos repetitivo, lo que facilita su mantenimiento y escalabilidad.

Desventajas

- **Rendimiento inferior en aplicaciones de alta demanda**: Comparado con otros lenguajes como Java o Node.js, Rails puede ser más lento en aplicaciones que manejan grandes volúmenes de tráfico o en tiempo real.

- **Problemas de escalabilidad**: Aunque es eficiente para proyectos pequeños y medianos, cuando las aplicaciones crecen en complejidad o tamaño, Rails puede requerir optimizaciones adicionales.
- **Consumo de recursos**: En servidores, Rails tiende a consumir más memoria y CPU en comparación con otros frameworks más ligeros.
- **Estructura prescriptiva**: Si bien las convenciones facilitan el desarrollo, en proyectos grandes, esta rigidez puede volverse un obstáculo al intentar implementar soluciones personalizadas o fuera de las convenciones estándar de Rails.

● Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

Ruby on Rails es ideal para el desarrollo de aplicaciones que requieren una base de datos sólida y una gestión eficiente del flujo de datos. Esto incluye plataformas de comercio electrónico, donde la gestión de usuarios, productos y transacciones necesita una solución rápida y confiable. También es adecuado para sistemas de gestión de contenido (CMS), aplicaciones web interactivas o redes sociales, donde las actualizaciones en tiempo real y la capacidad de manejar múltiples usuarios son clave. Además, se utiliza ampliamente en proyectos de software colaborativo y plataformas que necesitan integraciones de servicios de terceros.

● Casos de aplicación (Ejemplos y casos de éxito en la industria)

GitHub: Es la plataforma de desarrollo colaborativo más grande del mundo, que utiliza Git para facilitar el control de versiones y la colaboración entre desarrolladores. Permite gestionar proyectos a gran escala, automatizar pruebas, y desplegar aplicaciones. Empresas y comunidades open source lo usan ampliamente, y aloja millones de repositorios.

Shopify: Plataforma líder en comercio electrónico que permite a emprendedores y grandes empresas crear y gestionar tiendas en línea. Es robusto y escalable, manejando millones de transacciones diarias, y es valorado por su facilidad de uso para usuarios no técnicos.

Airbnb: Plataforma que conecta a anfitriones con huéspedes para ofrecer alojamiento en cualquier parte del mundo. Utiliza Rails para gestionar perfiles de usuarios, búsquedas avanzadas, y sistemas de pago. Su arquitectura escalable ha permitido a Airbnb crecer rápidamente y gestionar millones de usuarios y transacciones.

SoundCloud: Plataforma de streaming de música donde artistas independientes pueden compartir y distribuir su música. Rails ayudó a SoundCloud a integrar funcionalidades como comentarios en tiempo real y listas de reproducción, facilitando su crecimiento hasta convertirse en una plataforma con millones de pistas y usuarios.

Cassandra

- ## Definición

Inspirada en Amazon Dynamo y Google BigTable, Apache Cassandra es un sistema de gestión de bases de datos de código abierto diseñado para manejar grandes volúmenes de datos estructurados. Su excelente escalabilidad permite la distribución de la base de datos en múltiples clusters, facilitando su gestión en entornos de gran tamaño. Además, Cassandra adopta un enfoque redundante que minimiza el riesgo de fallos, aunque puede enfrentar dificultades en la replicación de datos. Esta solución distribuida y de código abierto es apta para su despliegue en la nube, en instalaciones locales (on-premise) o en entornos híbridos.

- ## Características

- **Distribución y escalabilidad:** Apache Cassandra es un sistema distribuido, donde los servidores están distribuidos entre múltiples nodos. Ofrece escalabilidad horizontal y lineal, lo que significa que al duplicar el número de nodos, también se duplica el número de operaciones por segundo (por ejemplo, con dos nodos se pueden realizar 100,000 operaciones por segundo).
- **Arquitectura peer-to-peer (P2P):** A diferencia de otros sistemas que utilizan una arquitectura maestro-servidor, Cassandra adopta un modelo P2P, lo que garantiza que, si un nodo falla, el sistema continúe funcionando sin interrupciones.
- **Tolerancia a fallos y replicación:** Cassandra es tolerante a fallos gracias a su sistema de replicación de datos. Si un nodo falla, los datos ya estarán replicados en otros nodos, garantizando su disponibilidad.
- **Replicación entre centros de datos:** Permite la replicación de datos en múltiples centros de datos, creando "anillos" de máquinas Cassandra, donde los datos del anillo 1 pueden replicarse en el anillo 2.
- **Consistencia ajustable:** Basado en el teorema CAP, Cassandra permite ajustar la disponibilidad y consistencia de los datos mediante la configuración de las propiedades de "replication factor" y "consistency level", lo que define cuándo los resultados se entregan a los clientes y cómo se distribuyen las escrituras a las réplicas.

- ## Historia y evolución

Apache Cassandra fue lanzada en 2008 y originalmente desarrollada por Facebook para mejorar la funcionalidad de búsqueda en su sistema de mensajes. Más tarde, fue transferida a la Fundación Apache, convirtiéndose en una herramienta de código abierto que sigue siendo mantenida hasta la fecha. En febrero de 2010, se convirtió en un proyecto de nivel superior (top-level) dentro de la Fundación Apache.

Cassandra está inspirada en los trabajos de Amazon Dynamo (2007) y Google BigTable (2006), dos sistemas pioneros en el manejo de grandes volúmenes de datos distribuidos. Su nombre proviene de Casandra, la sacerdotisa de la mitología griega que tenía el don de la profecía y predijo el engaño del caballo de Troya. Actualmente, su desarrollo está liderado por la compañía DataStax.

- ## Ventajas y desventajas

Ventajas:

- **Alta disponibilidad:** Cassandra es ideal para sistemas críticos donde las caídas no son tolerables, ya que está diseñada para funcionar con mínima interrupción.
- **Tolerancia a particiones y escalabilidad:** Cassandra soporta la tolerancia a particiones, lo que le permite seguir operando incluso cuando hay problemas de red o fallos de nodos. Además, es altamente escalable, pudiendo crecer sin afectar su rendimiento.
- **Escalabilidad y baja probabilidad de fallo:** Cassandra ofrece una excelente escalabilidad, permitiendo agregar nodos sin comprometer la estabilidad o el rendimiento del sistema, con un riesgo mínimo de fallos.
- **Almacenamiento de datos flexible:** Cassandra puede gestionar datos estructurados, semiestructurados y no estructurados, brindando a los usuarios gran flexibilidad para almacenar y manejar distintos tipos de información.
- **Distribución de datos flexible:** Cassandra permite distribuir datos fácilmente a través de múltiples centros de datos, lo que facilita su uso en diferentes ubicaciones y asegura la redundancia geográfica.
- **Soporte para propiedades ACID:** Aunque no es completamente ACID en su totalidad, Cassandra ofrece soporte para estas propiedades, lo que garantiza transacciones seguras y consistentes bajo ciertas configuraciones.
- **Recursos ajustables:** Cassandra puede gestionar eficientemente los recursos disponibles, ajustando su configuración según la capacidad y necesidades del sistema.

Desventajas

- **Complejidad en la adición de nodos:** Añadir nuevos nodos no es una tarea sencilla, ya que el sistema debe sincronizarse con los nodos existentes, lo cual puede llevar tiempo y afectar el rendimiento temporalmente.
- **Optimización de consultas:** Cassandra requiere que las consultas (queries) estén bien definidas con antelación, ya que sufre en rendimiento al realizar consultas tipo "SELECT" debido a la forma en que los datos están almacenados.
- **Consistencia eventual:** Aunque ofrece tolerancia a fallos, Cassandra puede sacrificar la consistencia en algunos casos, ya que sigue un modelo de consistencia eventual, lo que puede generar retrasos en la actualización de datos en entornos distribuidos.
- **Mayor complejidad operativa:** La administración y optimización de Cassandra puede ser más complicada en comparación con otros sistemas de bases de datos, requiriendo un conocimiento profundo para su correcta configuración y mantenimiento.

https://www.paradigmadigital.com/dev/cassandra-la-dama-de-las-bases-de-datos-nosql/

● Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

A continuación, algunos casos de uso clave donde Cassandra es particularmente adecuada:

- **Aplicaciones de redes sociales**:
    - Las plataformas de redes sociales necesitan gestionar grandes volúmenes de datos generados por los usuarios, como publicaciones, comentarios, me gusta y conexiones entre usuarios. Cassandra se utiliza en estos escenarios debido a su capacidad para manejar grandes cantidades de datos distribuidos en diferentes nodos con baja latencia y sin un solo punto de falla. Un ejemplo es Facebook, que utiliza Cassandra para gestionar su servicio de mensajería.
- **Sistemas de recomendación**:
    - Las aplicaciones que requieren generar recomendaciones personalizadas (por ejemplo, Netflix o Amazon) pueden beneficiarse de Cassandra. Este tipo de sistemas necesitan almacenar grandes volúmenes de datos de usuarios y comportamientos, y acceder a ellos de manera rápida para procesar recomendaciones en tiempo real.
- **Registros de eventos y auditoría**:
    - Las empresas que necesitan almacenar registros detallados de eventos o actividades, como logs de servidores, transacciones bancarias o actividades de usuarios en aplicaciones web, utilizan Cassandra debido a su capacidad para escribir grandes volúmenes de datos de manera rápida y eficiente. Su

arquitectura distribuida permite manejar escrituras a gran escala sin comprometer el rendimiento.

- **Gestión de datos geoespaciales**:
    - Aplicaciones que requieren manejar grandes cantidades de datos geoespaciales, como las de transporte y logística, se benefician de Cassandra. Por ejemplo, empresas como Uber utilizan bases de datos como Cassandra para almacenar datos relacionados con la ubicación de los usuarios, conductores y rutas, aprovechando su capacidad para escalar horizontalmente y su baja latencia.
- **Monitoreo de IoT**:
    - En soluciones de Internet de las Cosas (IoT), donde los dispositivos generan grandes cantidades de datos en tiempo real, Cassandra se utiliza para almacenar y procesar estos flujos de datos. Empresas que gestionan redes de sensores o dispositivos conectados la eligen por su capacidad para manejar altas tasas de escritura y proporcionar consistencia eventual a gran escala.
- **E-commerce**:
    - Los sitios web de comercio electrónico requieren manejar grandes cantidades de datos sobre productos, clientes, y transacciones. Cassandra permite gestionar catálogos de productos en diferentes regiones geográficas, optimizando el rendimiento mediante su distribución en múltiples nodos y garantizando la alta disponibilidad incluso en caso de fallos en algunos de ellos.
- **Servicios financieros y análisis en tiempo real**:
    - Las instituciones financieras que manejan datos de transacciones, procesamiento de pagos y análisis en tiempo real también encuentran en Cassandra una solución viable. Empresas como ING y Walmart utilizan Cassandra para sus operaciones de análisis en tiempo real, aprovechando su modelo de replicación para garantizar la integridad de los datos y minimizar la latencia.

- ## Casos de aplicación (Ejemplos y casos de éxito en la industria)

Netflix: Utiliza Cassandra para gestionar su enorme cantidad de datos, particularmente para el registro de auditorías y garantizar la disponibilidad continua de su servicio de streaming a nivel global..

Apple: Maneja más de 160,000 instancias de Cassandra para almacenar más de 100 petabytes de datos, repartidos en más de 1,000 clusters. Esto les permite manejar eficientemente grandes volúmenes de datos distribuidos.

Uber: Emplea Cassandra para asegurar la baja latencia en sus servicios de transporte, lo que es crucial para gestionar datos en tiempo real como las posiciones de conductores y usuarios.

Spotify: Gestiona listas de reproducción y la actividad de usuarios utilizando Cassandra, aprovechando su capacidad para manejar grandes volúmenes de datos y realizar actualizaciones frecuentes.

Instagram: Usa Cassandra para soportar la infraestructura de su aplicación, gestionando fotos, comentarios y otros datos de usuario en tiempo real.

Reddit: También se beneficia de Cassandra para manejar la gran cantidad de interacciones entre usuarios, como votos y comentarios en sus publicaciones.

Relación entre los temas asignados:

- ## Qué tan común es el stack designado

Tecnologías:

1. Frontend - Framework de Android con Kotlin: Kotlin es el lenguaje oficial para el desarrollo de aplicaciones Android, adoptado en 2017 por Google debido a su sintaxis más moderna y concisa en comparación con Java. Es ampliamente utilizado en la industria, especialmente por equipos que buscan crear aplicaciones móviles eficientes y escalables. Kotlin se ha convertido en una elección estándar para los desarrolladores de Android, con una adopción creciente en la mayoría de las empresas que crean aplicaciones nativas.
2. Backend - Ruby on Rails: Ruby on Rails es un framework de desarrollo backend conocido por su capacidad de facilitar el desarrollo rápido de aplicaciones web. Aunque fue extremadamente popular entre 2005 y 2015, ha perdido algo de protagonismo en favor de otros frameworks como Node.js y Django. Sin embargo, sigue siendo una opción sólida para startups y pequeños equipos que valoran la velocidad de desarrollo y la simplicidad, aunque puede no ser tan escalable como otras opciones modernas para aplicaciones más grandes.
3. Integración - GraphQL: GraphQL ha crecido rápidamente en popularidad debido a su capacidad para optimizar las consultas entre el frontend y el backend, permitiendo a los clientes solicitar sólo los datos que necesitan. Esta flexibilidad lo ha convertido en una opción atractiva frente a las APIs tradicionales basadas en REST. Aunque REST sigue siendo dominante, GraphQL es especialmente apreciado en proyectos modernos que

requieren eficiencia en el manejo de datos, particularmente en aplicaciones móviles y web.

4. Base de datos - Cassandra: Apache Cassandra es una base de datos NoSQL altamente escalable y distribuida, diseñada para manejar grandes cantidades de datos en múltiples servidores. Es común en aplicaciones que requieren alta disponibilidad y rendimiento en tiempo real, como plataformas de redes sociales o servicios financieros. Aunque no es la opción más común para proyectos pequeños o medianos, es extremadamente efectiva en aplicaciones que necesitan gestionar grandes volúmenes de datos y garantizar la disponibilidad en entornos distribuidos.

Stack:

El stack designado no es muy común en su conjunto, principalmente porque las tecnologías que lo componen no suelen utilizarse juntas. Ruby on Rails (RoR) y Cassandra no se integran frecuentemente en proyectos, ya que suelen encontrarse en contextos diferentes. Rails tiende a ser utilizado con bases de datos relacionales, mientras que Cassandra es una base de datos NoSQL más específica para ciertos escenarios.

Por otro lado, Ruby on Rails ha visto una disminución en su popularidad en los últimos años. Si bien sigue siendo una herramienta muy eficaz para desarrollar rápidamente aplicaciones web, muchos proyectos modernos están optando por frameworks más escalables, como Node.js o Django. Esto se debe a la necesidad de soportar aplicaciones más complejas y a la mayor flexibilidad de estas alternativas.

Cassandra, por su parte, es una base de datos NoSQL diseñada para manejar grandes volúmenes de datos distribuidos, siendo ideal para proyectos que requieren alta disponibilidad y escalabilidad a gran escala. No es una opción común en proyectos más pequeños o medianos, donde las bases de datos relacionales como PostgreSQL o MySQL suelen ser la norma. Sin embargo, Cassandra es perfecta para aplicaciones que necesitan gestionar grandes cantidades de datos en tiempo real.

Matriz de análisis de Principios SOLID vs Temas

Ahora, se analizarán los principios SOLID en el contexto de tecnologías avanzadas como Cassandra, Ruby on Rails, GraphQL y Android, así como la implementación del patrón de diseño CQRS (Command Query Responsibility Segregation). Los principios SOLID proporcionan una base sólida para el diseño de software orientado a objetos, garantizando que los sistemas sean mantenibles, escalables y fáciles de modificar. A lo largo del taller, se detallará cómo estos principios se aplican específicamente en sistemas distribuidos, donde

tecnologías como Cassandra permiten gestionar grandes volúmenes de datos, y cómo Ruby on Rails facilita la segregación de responsabilidades en la capa de backend. Además, veremos cómo el patrón CQRS ayuda a dividir las responsabilidades de lectura y escritura, mejorando el rendimiento en sistemas con alta concurrencia y grandes volúmenes de datos. De esta forma, se establecerán las bases para entender cómo construir software robusto y eficiente utilizando las mejores prácticas de diseño.

	Android native	GraphQL	Ruby on rails	Cassandra	CQRS
Single Responsibility	Medio	Alto	Medio	Bajo	Alto
Open/Closed	Medio	Alto	Alto	Medio	Medio
Liskov Substitution	Bajo	Medio	Alto	Bajo	Medio
Interface Segregation	Alto	Medio	Medio	Bajo	Alto
Dependency Inversion	Medio	Medio	Alto	Bajo	Medio

Android Native: Android Native se refiere al desarrollo de aplicaciones móviles directamente en el sistema operativo Android utilizando lenguajes como Java o Kotlin. Ofrece un alto rendimiento y acceso directo a las funcionalidades nativas del dispositivo, como la cámara y el GPS. Sin embargo, puede requerir una mayor gestión de dependencias y modularidad, lo que a veces puede dificultar la implementación de principios de diseño como SOLID.

GraphQL: GraphQL es un lenguaje de consulta de datos que permite a los clientes solicitar exactamente los datos que necesitan, sin tener que depender de endpoints rígidos como en REST. Esto hace que sea muy flexible y extensible, facilitando la implementación de principios como la responsabilidad única y abierto/cerrado. GraphQL es especialmente útil en aplicaciones que requieren optimización del manejo de datos y respuesta rápida en las consultas.

Ruby on Rails: Ruby on Rails es un framework de desarrollo web basado en el lenguaje Ruby, que sigue el patrón de arquitectura MVC (Modelo-Vista-Controlador). Rails promueve la "Convención sobre Configuración", lo que facilita a los desarrolladores crear aplicaciones

rápidamente con menos configuración manual. Su enfoque estructurado y modular lo hace adecuado para aplicar principios SOLID, especialmente abierto/cerrado y sustitución de Liskov.

Cassandra: Apache Cassandra es una base de datos NoSQL distribuida diseñada para manejar grandes cantidades de datos en múltiples nodos sin un único punto de falla. Es altamente escalable y confiable, especialmente para aplicaciones que requieren disponibilidad continua y replicación geográfica. Aunque su arquitectura orientada a datos la hace poco adecuada para algunos principios SOLID, como responsabilidad única o inversión de dependencias, es excepcional para el manejo de datos masivos de manera distribuida.

CQRS: El patrón CQRS aplica alto en el principio de responsabilidad única (S), ya que separa claramente las responsabilidades de lectura y escritura en distintas capas o modelos, simplificando el código y enfocando cada componente en una sola tarea. En cuanto al principio de segregación de interfaces (I), también tiene una alta aplicación, dado que define interfaces específicas para las consultas y comandos, evitando interfaces monolíticas. Por otro lado, en los principios como abierto/cerrado (O), sustitución de Liskov (L) y inversión de dependencias (D), CQRS tiene una aplicación media. Aunque favorece la extensión del sistema sin modificar componentes existentes (O), su implementación puede requerir ajustes más complejos al separar el modelo de comandos y consultas. La sustitución de Liskov (L) no siempre es directa debido a la separación estricta entre los dos modelos, y la inversión de dependencias (D) es posible, pero depende de una correcta configuración de los repositorios y servicios para manejar ambas capas. Este patrón fomenta una arquitectura flexible, pero su implementación puede ser más compleja en algunos principios si no se gestiona adecuadamente.

- ## Matriz de análisis de Atributos de Calidad vs Temas

A continuación, se presenta una tabla que tiene como objetivo comparar los atributos de calidad de la ISO 25000 en las tecnologías mencionadas previamente. Estos atributos son fundamentales para evaluar el rendimiento general y la capacidad de cada tecnología en función de su capacidad para cumplir con dichos atributos. La tabla proporciona una visión integral sobre su adecuación en términos de calidad de software, lo que facilita la selección de las herramientas más apropiadas para cumplir con los requisitos del proyecto, tanto a nivel técnico como en lo referente a la experiencia de usuario.

	Android native	GraphQL	Ruby on rails	Cassandra	CQRS
Idoneidad funcional	Alto	Alto	Alto	Medio	Alto
Eficiencia	Medio	Alto	Medio	Alto	Alto
Compatibilidad	Bajo	Medio	Medio	Bajo	Alto
Usabilidad	Alto	Alto	Alto	Medio	Bajo
Confiabilidad	Alto	Medio	Medio	Alto	Alto
Seguridad	Alto	Medio	Alto	Alto	Alto
Mantenibilidad	Medio	Alto	Alto	Medio	Medio
Portabilidad	Bajo	Alto	Medio	Medio	Alto

Android Native:

- La idoneidad funcional es alta debido a la capacidad de la plataforma para crear aplicaciones móviles específicas y ricas en funciones.
- La eficiencia es media debido a la optimización nativa, aunque podría mejorar en algunos escenarios de rendimiento frente a plataformas de mayor escalabilidad.
- Compatibilidad y Portabilidad son bajas, ya que las aplicaciones Android nativas están diseñadas principalmente para dispositivos Android.
- Usabilidad y confiabilidad son altas por su soporte robusto y madurez en el desarrollo móvil.
- La seguridad es alta, con muchas opciones para seguridad a nivel del sistema operativo.
- Mantenibilidad es media, con mayor complejidad en comparación con frameworks multiplataforma.

GraphQL:

- La idoneidad funcional es alta, especialmente en la capacidad de recuperar datos de manera precisa y eficiente.
- La eficiencia es alta debido a la capacidad de optimizar las consultas, trayendo solo los datos necesarios.

- Compatibilidad y Portabilidad son altas, ya que GraphQL se puede usar con diversos lenguajes y plataformas.
- La usabilidad es alta porque es fácil de usar y flexible, lo que permite a los desarrolladores definir exactamente lo que necesitan.
- Confiabilidad y Seguridad son medias, dependiendo de cómo se implemente el esquema y las validaciones.
- Mantenibilidad es alta debido a su flexibilidad y diseño enfocado en la reutilización de consultas.

Ruby on Rails:

- La idoneidad funcional es alta, con muchas bibliotecas y funciones listas para crear aplicaciones web rápidamente.
- La eficiencia es media, ya que Ruby no es tan rápido como otros lenguajes, aunque Rails es eficiente en términos de desarrollo ágil.
- La compatibilidad es media; Ruby on Rails puede funcionar en diversas plataformas, pero es más común en entornos web específicos.
- La usabilidad es alta gracias a su enfoque en la simplicidad y facilidad de uso.
- La confiabilidad es media; aunque el framework es maduro, su escalabilidad puede verse limitada en grandes aplicaciones.
- La seguridad es alta, ya que Rails ofrece muchas funciones de seguridad por defecto.
- Mantenibilidad es alta, con un enfoque en la convención sobre la configuración.
- La portabilidad es media, limitado principalmente a entornos web.

Cassandra:

- La idoneidad funcional es media, excelente para grandes volúmenes de datos distribuidos, pero no es ideal para todas las aplicaciones.
- La eficiencia es alta, con alta capacidad de procesamiento de datos y baja latencia.
- Compatibilidad y Portabilidad son medias, ya que está diseñado para entornos distribuidos específicos.
- La usabilidad es media debido a su complejidad para los usuarios nuevos.
- Confiabilidad y Seguridad son altas, ya que Cassandra está diseñado para alta disponibilidad y redundancia.
- Mantenibilidad es media, con costos de mantenimiento altos en entornos distribuidos.

CQRS:

- La idoneidad funcional es alta, ideal para sistemas donde se requieren operaciones complejas de lectura y escritura por separado.

- La eficiencia es alta porque separa las lecturas y escrituras, optimizando cada operación.
- Compatibilidad y Portabilidad son altas, ya que es un patrón arquitectónico aplicable en diversas tecnologías.
- La usabilidad es baja, ya que su implementación puede ser compleja y requiere una arquitectura bien pensada.
- La confiabilidad es alta debido a su capacidad de manejar grandes cargas de datos y procesos distribuidos.
- La seguridad es alta, dependiendo de la implementación del sistema subyacente.
- Mantenibilidad es media, ya que puede ser complejo mantener la sincronización entre los comandos y las consultas.
- La portabilidad es alta, dado que se puede aplicar en diferentes entornos tecnológicos y arquitecturas.

● Matriz de análisis de Tácticas vs Temas

La siguiente tabla permite evaluar cómo las distintas tecnologías utilizadas en tu proyecto (Cassandra, Ruby on Rails, GraphQL, Android) responden a estrategias específicas, como la optimización de consultas o el manejo de concurrencia. La tabla clasifica la efectividad de cada tecnología en diferentes áreas clave, facilitando la toma de decisiones informadas para mejorar el rendimiento, escalabilidad y seguridad del sistema.

	Cassandra	RubyOnRails	GraphQL	Android
Optimización de consultas	Alto	Medio	Alto	Bajo
Manejo de concurrencia	Medio	Bajo	Alto	Medio
Manejo de errores y fallos	Medio	Alto	Medio	Medio
Escalabilidad	Alto	Medio	Alto	Bajo
Facilidad de integración	Bajo	Alto	Alto	Medio
Seguridad	Medio	Alto	Alto	Bajo

Replicación	Alto	Bajo	N/A	N/A
Monitoreo	Medio	Alto	Medio	Bajo

Optimización de consultas:

- **Cassandra**: Es una base de datos NoSQL que está diseñada para manejar grandes cantidades de datos de forma rápida, por lo que su optimización de consultas podría considerarse Alta.
- **Ruby on Rails**: Tiene capacidades moderadas para optimizar consultas con ActiveRecord, pero depende del diseño de las consultas, por lo que lo clasificamos como Medio.
- **GraphQL**: Su diseño permite pedir solo los datos necesarios, lo que lo hace muy eficiente, por eso se califica como Alto.
- **Android**: La optimización de consultas no está directamente relacionada con el frontend, por lo que aquí sería Bajo.

Manejo de concurrencia:

- **Cassandra**: Puede manejar muchas solicitudes simultáneamente, por lo que lo calificamos como Medio.
- **Ruby on Rails**: No está optimizado para manejar concurrencia de forma nativa sin configuración adicional, por lo que sería Bajo.
- **GraphQL**: Como intermediario entre el backend y frontend, puede manejar muchas solicitudes a la vez, por eso se clasifica como Alto.
- **Android**: El manejo de concurrencia depende de la implementación en la app móvil, por eso lo calificamos como Medio.

Manejo de errores y fallos:

- **Cassandra**: Tiene capacidades moderadas para la gestión de errores y caídas, ya que depende de la configuración, por lo que sería Medio.
- **Ruby on Rails**: Tiene excelentes herramientas para manejar errores, como excepciones y logs, lo que lo hace Alto.
- **GraphQL**: Permite manejo de errores, pero su robustez depende mucho de la implementación, por lo que sería Medio.
- **Android**: Depende de la implementación en el frontend, por eso sería Medio.

Escalabilidad:

- **Cassandra**: Altamente escalable, diseñada para crecer horizontalmente, por lo que se clasifica como Alto.
- **Ruby on Rails**: Puede escalar, pero no es su punto fuerte sin configuraciones adicionales, por lo que lo calificamos como Medio.
- **GraphQL**: Su estructura es ligera y puede escalar fácilmente, por lo que sería Alto.
- **Android**: La escalabilidad depende principalmente del backend, por lo que sería Bajo aquí.

Facilidad de integración:

- **Cassandra**: Es más complejo de integrar con otros servicios, como Ruby on Rails, por lo que sería Bajo.
- **Ruby on Rails**: Altamente flexible e integrable con bases de datos y APIs, por eso es Alto.
- **GraphQL**: Diseñado para integrarse con múltiples fuentes de datos, lo que lo hace Alto.
- **Android**: Puede integrarse bien con APIs como GraphQL, por lo que sería Medio.

Seguridad:

- **Cassandra**: Ofrece mecanismos de seguridad, pero requiere configuración avanzada, por lo que sería Medio.
- **Ruby on Rails**: Tiene herramientas sólidas de seguridad por defecto, lo que lo hace Alto.
- **GraphQL**: Puede ser seguro, pero requiere una buena implementación, por lo que lo calificamos como Alto.
- **Android**: La seguridad en Android es más compleja, y depende mucho de cómo se implemente, por lo que sería Bajo.

Replicación:

- **Cassandra**: Diseñada para alta disponibilidad mediante replicación automática de datos en múltiples nodos. Esto hace que la replicación sea un punto fuerte, por lo que se clasifica como Alto.
- **Ruby on Rails**: No tiene mecanismos nativos de replicación, pero puede implementarse a través de bases de datos o herramientas externas, por lo que sería Bajo.
- **GraphQL**: No maneja replicación de datos directamente, ya que es un lenguaje de consulta, por lo que se clasifica como N/A.

- **Android**: La replicación de datos no es relevante en el frontend, por lo que se clasifica como N/A.

Monitoreo:

- **Cassandra**: Ofrece capacidades moderadas para monitorear la salud del sistema y el rendimiento, aunque requiere herramientas adicionales como Prometheus o Grafana para monitoreo en tiempo real, por lo que sería Medio.
- **Ruby on Rails**: Existen muchas herramientas nativas y externas para monitorear el rendimiento, como New Relic o Skylight, lo que lo hace Alto en monitoreo.
- **GraphQL**: No tiene capacidades de monitoreo nativas, pero el monitoreo se puede implementar a través del backend, por lo que sería Medio.
- **Android**: El monitoreo es más limitado en la app, y depende del uso de herramientas externas para rastrear el rendimiento en producción, por lo que sería Bajo.

● Matriz de análisis de Mercado laboral vs Temas

En el contexto tecnológico actual, el mercado laboral se caracteriza por la rápida evolución de herramientas y plataformas que transforman la forma en que las empresas y los desarrolladores abordan el desarrollo de software. Para entender mejor la relevancia de diversas tecnologías en función de su demanda, nuevas ofertas, competencia y necesidad, es fundamental hacer un análisis de cada una en relación con estos factores de mercado. A continuación, se presenta una descripción detallada de cinco tecnologías (Android Native, GraphQL, Ruby on Rails, Cassandra y CQRS) para evaluar cómo encajan en el panorama del mercado laboral actual y qué tan alineadas están con las tendencias emergentes en el sector tecnológico.

	Android native	GraphQL	Ruby on rails	Cassandra	CQRS
Demanda	Alta	Alta	Media	Media	Alta
Nuevas ofertas	Media	Alta	Baja	Media	Media
Competencia	Alta	Alta	Alta	Media	Media
Necesidad	Alta	Alta	Media	Alta	Alta

Android Native: sigue siendo una de las tecnologías más demandadas, impulsada por la popularidad global del sistema operativo Android. Las aplicaciones móviles nativas permiten a los desarrolladores explotar al máximo las capacidades de los dispositivos Android, pero el surgimiento de soluciones multiplataforma como Flutter ha generado una mayor competencia. Aunque las nuevas ofertas en el desarrollo nativo no son tan frecuentes, sigue siendo una tecnología de alta necesidad para empresas que buscan aprovechar el ecosistema de Android de manera eficiente y directa.

GraphQL: ha experimentado un auge en demanda debido a su capacidad para optimizar la comunicación entre el frontend y el backend, permitiendo a los desarrolladores solicitar solo los datos necesarios. En comparación con REST, GraphQL es más eficiente y flexible, lo que ha generado una ola de nuevas ofertas y adopciones. La competencia es fuerte, ya que REST aún tiene una amplia adopción, pero la necesidad de herramientas que manejen grandes volúmenes de datos de manera eficiente ha impulsado a GraphQL como una opción destacada en proyectos modernos.

Ruby on Rails: aunque sigue siendo una opción viable para el desarrollo ágil de aplicaciones web, ha visto una disminución en su demanda debido a la aparición de otros frameworks más modernos y eficientes como Node.js y Django. Las nuevas ofertas en el ecosistema Rails son limitadas, lo que ha reducido su innovación reciente. Sin embargo, sigue siendo competitivo en proyectos que requieren simplicidad y velocidad en el desarrollo, aunque la necesidad del framework ha disminuido en aplicaciones a gran escala donde se requiere mayor rendimiento y escalabilidad.

Cassandra: es ampliamente reconocida por su capacidad de manejar grandes volúmenes de datos distribuidos, siendo una solución preferida para empresas que necesitan alta disponibilidad y escalabilidad. La demanda de Cassandra es específica para proyectos de gran envergadura, como redes sociales y sistemas de análisis de datos en tiempo real. Aunque la competencia en el espacio de bases de datos distribuidas es moderada, Cassandra sigue siendo relevante en entornos donde la alta confiabilidad y la resistencia a fallos son imprescindibles.

CQRS: El patrón arquitectónico CQRS (Command Query Responsibility Segregation) está en alta demanda para proyectos que requieren separar las operaciones de lectura y escritura, lo que permite una optimización del rendimiento en sistemas distribuidos. La adopción de CQRS ha crecido junto con la tendencia de microservicios y arquitecturas distribuidas. Aunque no es una tecnología por sí misma, sino un patrón de diseño, ha visto nuevas implementaciones y

herramientas que facilitan su uso. La necesidad de CQRS es alta en sistemas que requieren consistencia eventual y escalabilidad, especialmente en sectores como el comercio electrónico y las finanzas.

- **Matriz de análisis de Patrones laboral vs Temas**

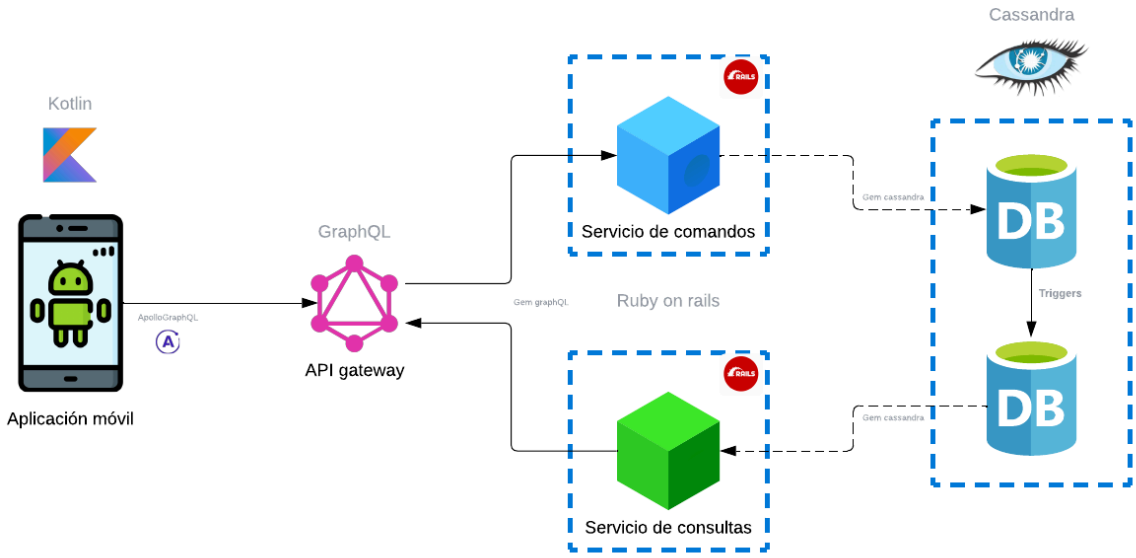
En la presente tabla se establecen conexiones entre distintos patrones laborales y tecnologías clave utilizadas en el desarrollo de software, destacando cómo cada patrón impacta el uso de bases de datos en Cassandra, el backend en Ruby on Rails, la integración entre backend y frontend mediante GraphQL, y el desarrollo frontend en Android. Se incluyen patrones como el desarrollo ágil, trabajo remoto, colaboración en equipo, mejora continua y la aplicación de metodologías ágiles, con el objetivo de ilustrar cómo estas prácticas laborales facilitan la eficiencia, la adaptabilidad y la evolución constante en cada uno de estos temas tecnológicos.

	Cassandra	RubyOnRails	GraphQL	Android
Desarrollo ágil	Permite iteraciones rápidas en ajustes de nodos y consultas	Facilita la implementación y cambios rápidos en las APIs	Soporta cambios rápidos en resolvers y consultas	Facilita la iteración continua en el diseño y funcionalidad
Trabajo remoto	Uso de clústeres distribuidos que facilitan el acceso remoto a datos	Permite un despliegue flexible y remoto de aplicaciones	Permite mantener la eficiencia en la comunicación entre equipos distribuidos	Herramientas de Android Studio para desarrollo remoto
Colaboración en equipo	Replicación y sincronización de datos entre equipos de desarrollo	Ruby on Rails permite la integración con equipos de frontend a través de APIs estandarizadas	Se centraliza la comunicación entre equipos, facilitando el control de versiones de APIs	Uso de herramientas como Figma o Zeplin para sincronizar el diseño entre equipos
Mejora continua	Optimización constante de índices y queries para mejorar el	Facilita la refactorización y mejora continua del código	Permite la actualización incremental de esquemas sin	Adaptación continua de interfaces de usuario a

	rendimiento		afectar al cliente	nuevos estándares y dispositivos
Metologías ágiles	Gestión iterativa de nodos y consultas, permitiendo cambios rápidos y pruebas de concepto	Se adapta bien a Scrum y otras metodologías ágiles por su facilidad de desarrollo rápido	Es flexible para soportar cambios rápidos y ajustables en los requerimientos del cliente	Soporta iteraciones ágiles para probar rápidamente nuevas funcionalidades

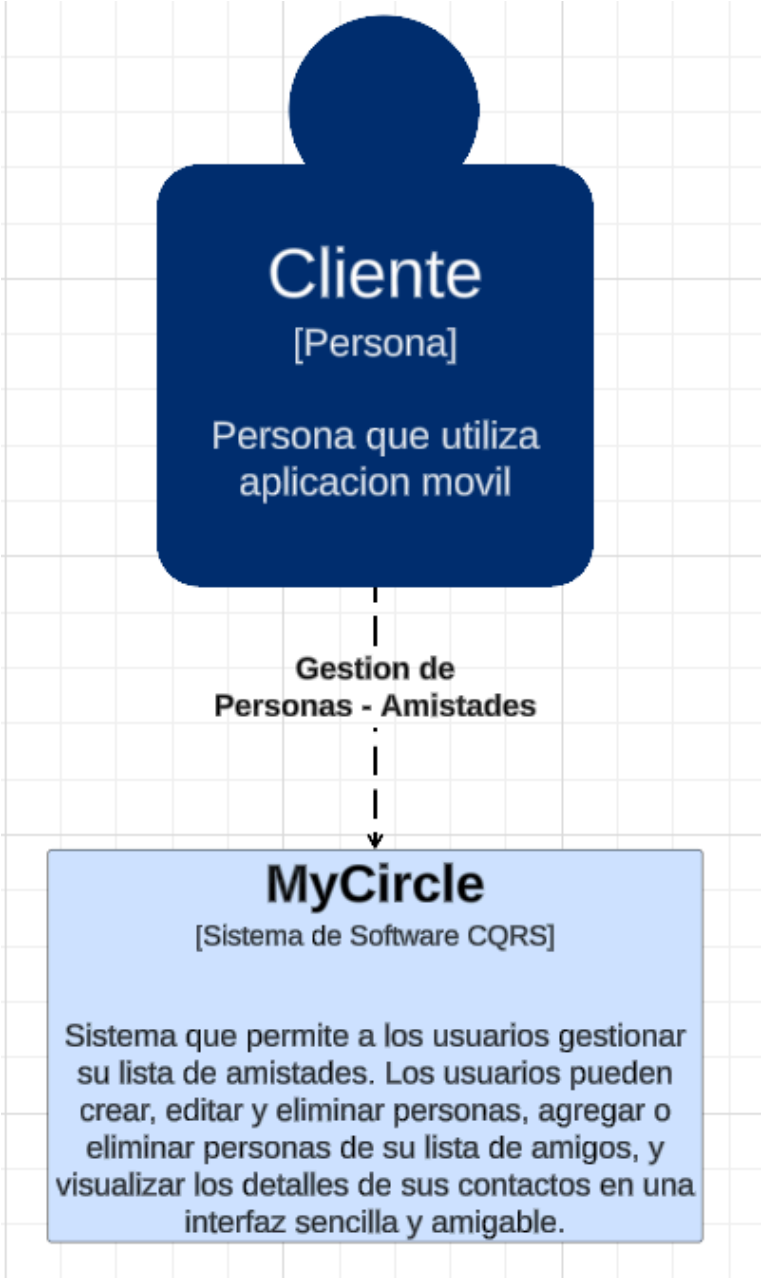
Ejemplo práctico y funcional relacionando los temas:

- **Alto nivel**

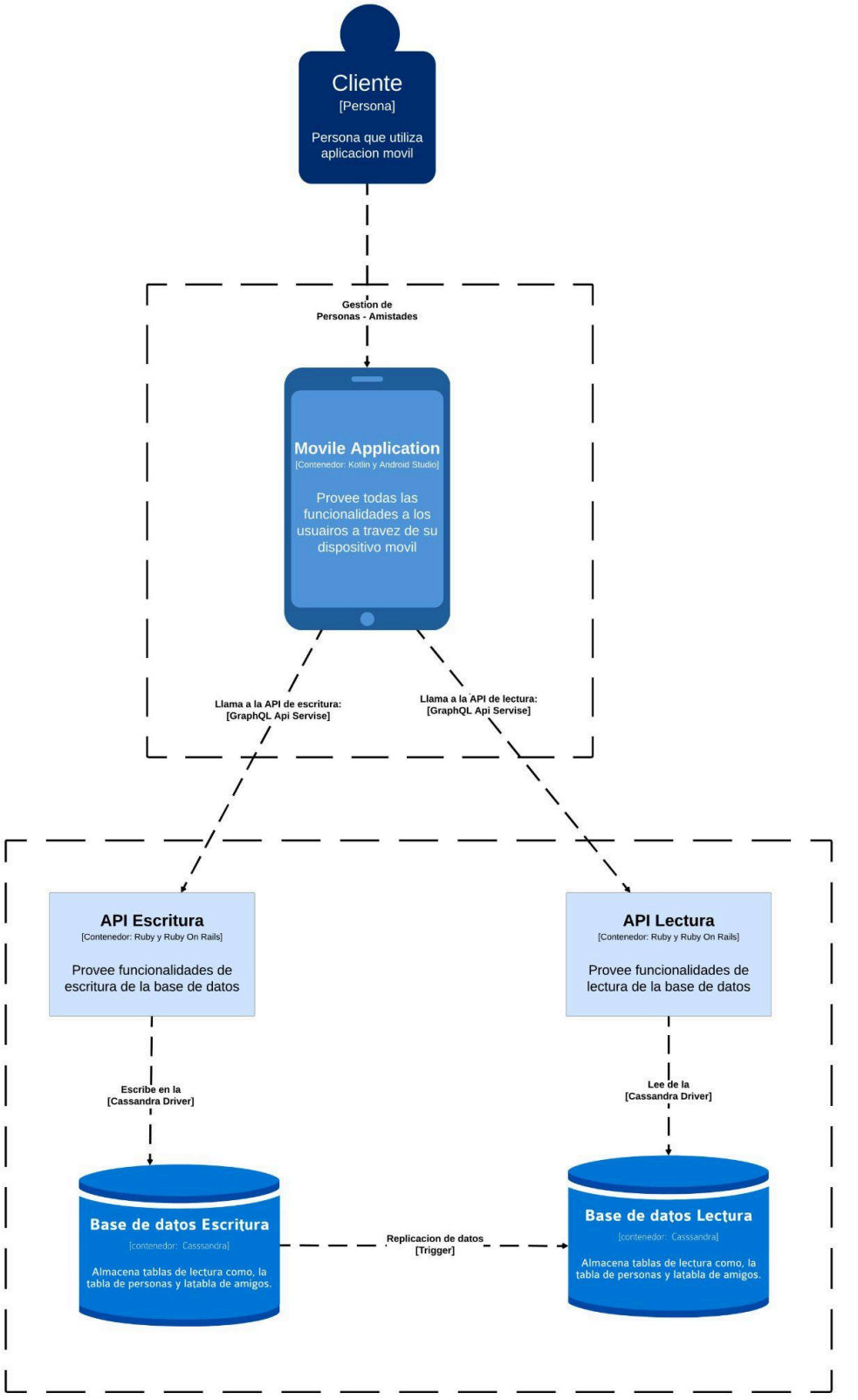


El diagrama muestra una arquitectura basada en el patrón CQRS, donde una aplicación móvil en Kotlin se comunica con un API Gateway mediante GraphQL. El backend separa las responsabilidades en un Servicio de Comandos (para escritura) y un Servicio de Consultas (para lectura), ambos implementados en Ruby on Rails. Los datos se almacenan en una base de datos Cassandra, con triggers para mantener la coherencia entre los servicios de escritura y lectura, optimizando el rendimiento y escalabilidad de cada operación.

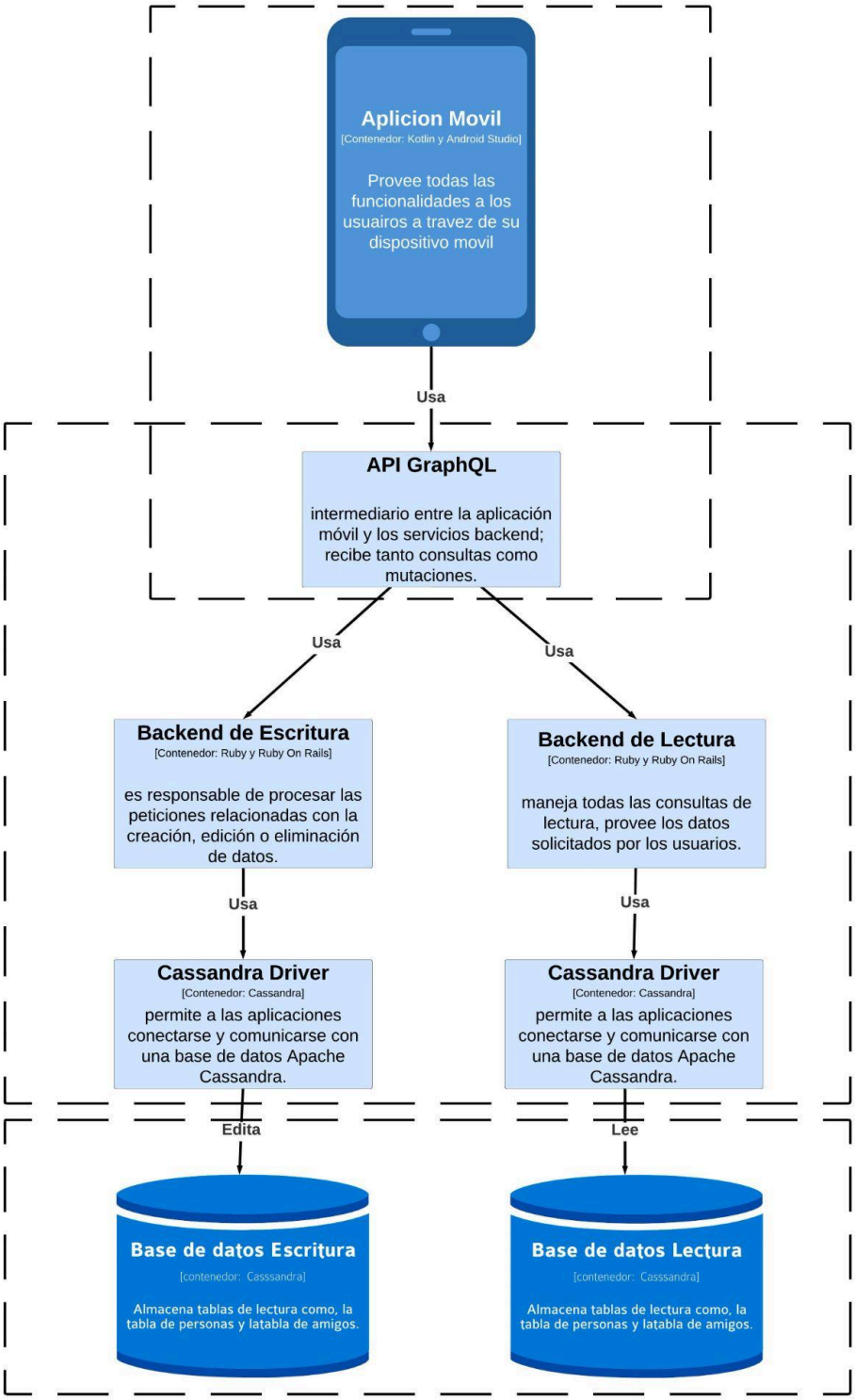
- **C4Model**
    - Contexto C4



- Contenedores C4

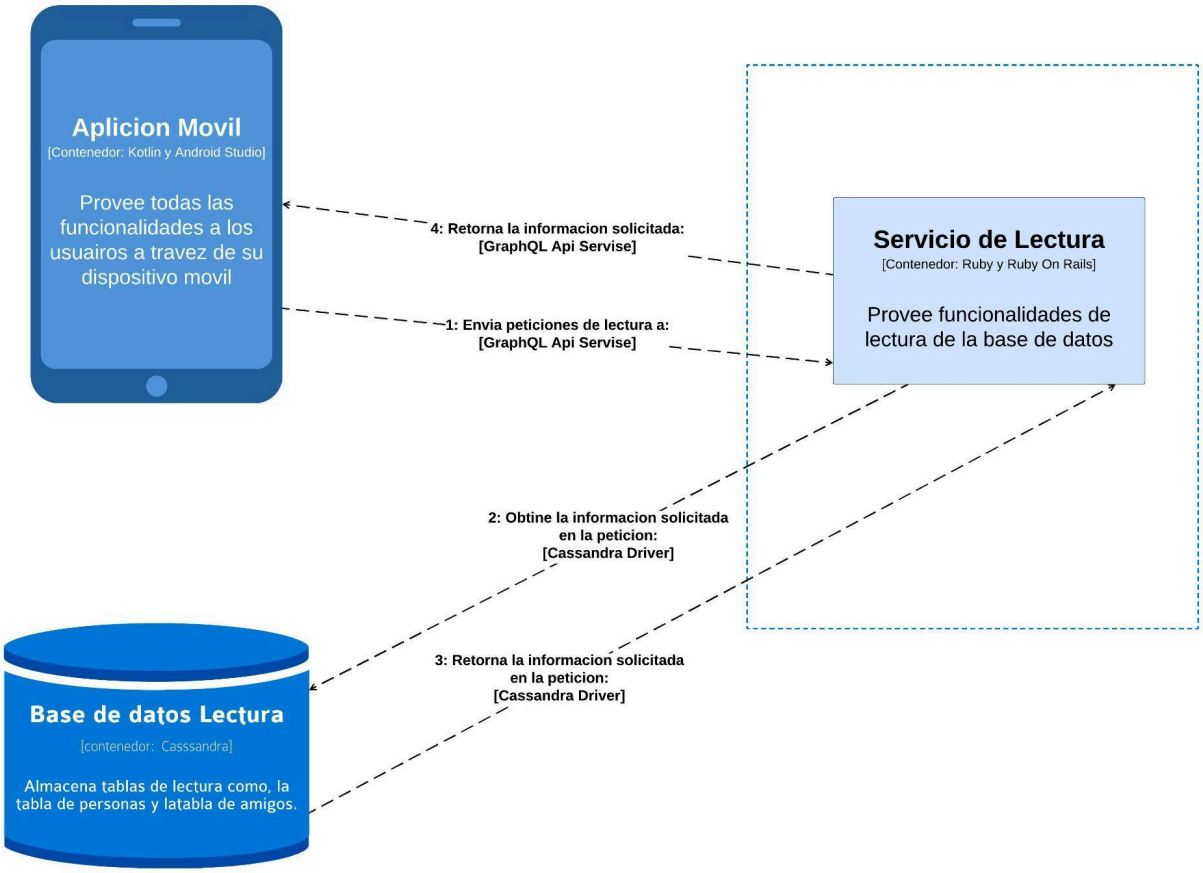


- Componentes C4

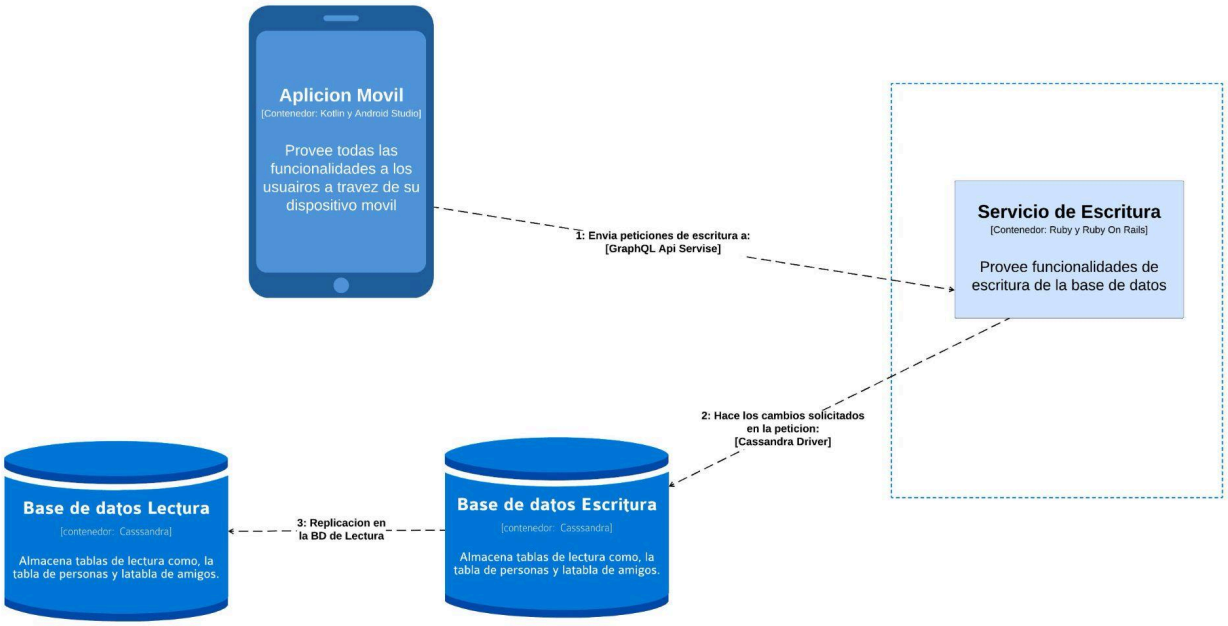


- **Diagrama Dynamic C4**

A continuación se presenta el diagrama dinámico C4 para las acciones de lectura:



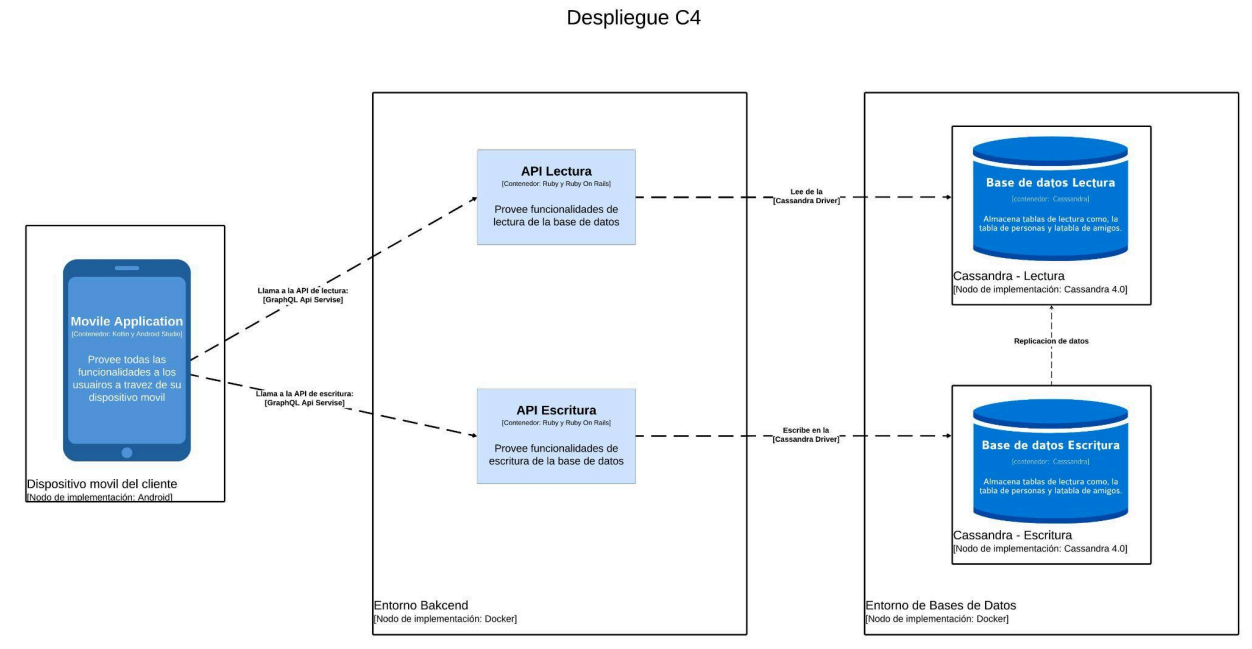
Ahora se presenta el diagrama dinámico C4 para las acciones de escritura o como tal mutaciones de la base de datos:



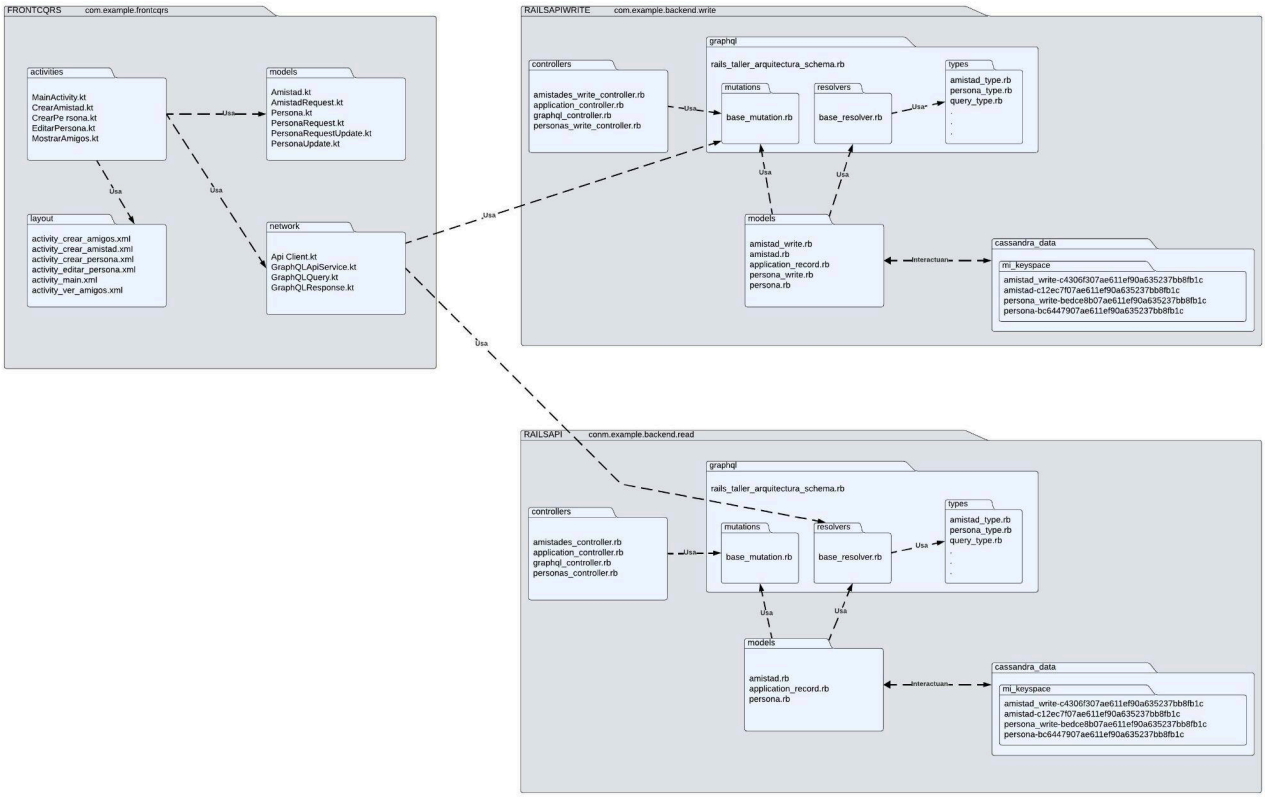
Como se puede observar a diferencia del diagrama de solo lectura, en este se realiza un proceso de actualización a la base de datos de lectura ya que la de escritura sufrió algún cambio.

- **Diagrama Despliegue C4**

A continuación se muestra el diagrama de despliegue C4:



- **Diagrama de paquetes UML de cada componente**



- **Código Fuente en repositorio/s públicos git**

Link repositorio escritura: https://github.com/SantiagoYB/RailsAPIWrite

Link repositorio lectura: https://github.com/suaracost/RailsAPI

Link repositorio front: https://github.com/suaracost/FrontCQRS

- **Muestra de funcionalidad, en vivo o video**

Para la muestra de funcionalidad hemos decidido grabar el siguiente video donde se explica la estructura del taller y se muestran todos sus componentes al igual que el producto final en funcionamiento

Link del video: https://www.youtube.com/watch?v=lbzI6d0gpGY&feature=youtu.be

Referencias

- Android. ¿Qué es Android? https://www.android.com/intl/es_es/what-is-android/

- Qué es Android. (2022, febrero 17). Xataka Android. https://www.xatakandroid.com/sistema-operativo/que-es-android
- Android. Características del sistema operativo Android. https://androidos.readthedocs.io/en/latest/data/caracteristicas/
- Doonamis.. Historia y evolución de Android. https://www.doonamis.com/historia-y-evolucion-android/#:~:text=La%20Historia%20de%20Android&text=fue%20fundada%20en%20octubre%20de,de%20un%20mercado%20en%20expansi%C3%B3n.
- Daza, Julián Camilo. *Computación móvil* [Diapositivas]. Profesor, Pontificia Universidad Javeriana.
- WabiMovil. El sistema operativo Android: Ventajas e inconvenientes. https://wabimovil.es/el-sistema-operativo-android-ventajas-e-inconvenientes/
- App Design. *Principales tipos de aplicaciones y ejemplos por sectores*. https://appdesign.dev/principales-tipos-de-aplicaciones-y-ejemplos-por-sectores/
- Android. *Casos de éxito de Android Enterprise*. https://www.android.com/intl/es_es/enterprise/customers/
- Android Enterprise Community. (2024). *Customer story: Driving the future – How Android Enterprise helps McLaren Racing "brake" boundaries*. https://www.androidenterprise.community/t5/tips-guides/customer-story-driving-the-future-how-android-enterprise-helps/ba-p/2500
- Fowler, M. (2011, Julio 14). CQRS. Martin Fowler. https://martinfowler.com/bliki/CQRS.html
- NetMentor. (2022, Marzo 5). Patrón CQRS explicado en 10 minutos. NetMentor. https://www.netmentor.es/entrada/patron-cqrs-explicado-10-minutos
- CosasDeDevs. (2021, Mayo 26). ¿Qué es CQRS? (Command Query Responsibility Segregation). CosasDeDevs. https://cosasdedevs.com/posts/que-es-cqrs/
- Ruby on Rails. (n.d.). Ruby on Rails. https://rubyonrails.org/
- Kódigo. (2022, marzo 1). Hablemos de Ruby on Rails: Qué es y para qué sirve. https://kodigo.org/hablemos-de-ruby-on-rails-ruby-on-rails-que-es-y-para-que-sirve/
- Platzi. (2020, septiembre 23). El origen de Ruby on Rails, el framework orientado a la felicidad. https://platzi.com/blog/historia-ruby-on-rails/
- OpenWebinars. (s.f.). ¿Qué es Apache Cassandra? Recuperado de https://openwebinars.net/blog/que-es-apache-cassandra/
- IONOS. (s.f.). Apache Cassandra: ¿qué es y para qué sirve? Recuperado de https://www.ionos.com/es-us/digitalguide/hosting/cuestiones-tecnicas/apache-cassandra/

- Aprender Big Data. (s.f.). Introducción a Apache Cassandra. Recuperado de https://aprenderbigdata.com/introduccion-apache-cassandra/
- Paradigma Digital. (2019, octubre 3). Cassandra, la dama de las bases de datos NoSQL. Recuperado de https://www.paradigmadigital.com/dev/cassandra-la-dama-de-las-bases-de-datos-nosql/
- Studocu. (s.f.). Historia de Cassandra. Recuperado de https://www.studocu.com/latam/document/universidad-interamericana-de-panama/legislacion-laboral/historia-de-cassandra/3930106
- Lakshman, A., & Malik, P. (2010). "Cassandra: a decentralized structured storage system". ACM SIGOPS Operating Systems Review:
  Puedes acceder a esta publicación a través del ACM Digital Library. Aquí está el enlace: https://dl.acm.org/doi/10.1145/1773912.1773922
- Hewitt, E. (2010). "Cassandra: The Definitive Guide". O'Reilly Media:
  Este libro está disponible en la plataforma de O'Reilly. Aquí tienes el enlace para verlo: https://www.oreilly.com/library/view/cassandra-the-definitive/9781449390412/
- Vaughn, J. (2021). "Distributed Data Show: How Uber leverages Cassandra to scale its data infrastructure":
  Este episodio está disponible en el sitio oficial del *Distributed Data Show*. Aquí está el enlace para acceder:
  https://www.datastax.com/resources/podcast/how-uber-leverages-cassandra
- Datastax. (n.d.). *What are People Using Cassandra for Anyway?*. Datastax. Recuperado de https://www.datastax.com
- HackerNoon. (2023). *A Guide to Using Apache Cassandra as a Real-time Feature Store*. HackerNoon. Recuperado de https://hackernoon.com
- GlobeNewswire. (2023). *Apache Cassandra® Releases Major Update, Enabling*. GlobeNewswire. Recuperado de https://www.globenewswire.com
- GraphQL. (n.d.). *Learn GraphQL*. Recuperado de https://graphql.org/learn/
- Red Hat. (n.d.). *¿Qué es GraphQL?*. Red Hat. Recuperado de https://www.redhat.com/es/topics/api/what-is-graphql#ventajas-y-desventajas
- AWS. (n.d.). *The difference between GraphQL and REST*. Amazon Web Services. Recuperado de https://aws.amazon.com/es/compare/the-difference-between-graphql-and-rest/

- Instaclustr. (n.d.). *Cassandra monitoring: A best practice guide*. Instaclustr. Recuperado de https://www.instaclustr.com/cassandra-monitoring-best-practices:contentReference[oaicite:0]{index=0}
- DNSstuff. (n.d.). *Apache Cassandra Monitoring How-To Guide*. DNSstuff. Recuperado de https://www.dnsstuff.com/apache-cassandra-monitoring
- Talent500. (2024). *GraphQL: Make it Run Like a Rocket! - Performance Optimization Techniques*. Recuperado de https://www.talent500.co/blog/graphql-performance-optimization-techniques
- Poespas Blog. (2024). *Optimizing Your Ruby on Rails Application for Lightning-Fast Performance*. Recuperado de https://blog.poespas.me/posts/2024/05/23/optimizing-ruby-on-rails-application-performance/
- Squash. (n.d.). *Ruby on Rails Performance Tuning and Optimization*. Recuperado de https://www.squash.io/blog/ruby-on-rails-performance-tuning-and-optimization
- Android Developers. (n.d.). *Best Practices for Performance*. Recuperado de https://developer.android.com/topic/performance