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

Curso académico 2021/2022

Trabajo de Fin de Máster

# Optimización a través de la Observabilidad con OpenTelemetry y la Implementación de CQRS

Autor: Víctor H. Vallejos
Tutor: Micael Gallego

# Índice

# Resumen

Este trabajo explora la optimización de aplicaciones mediante la implementación de Observabilidad con OpenTelemetry y la adopción del patrón Command Query Responsibility Segregation (CQRS). Se parte de una aplicación bien diseñada y arquitectónicamente sólida, cuyo éxito inicial y buen funcionamiento han sentado las bases para su evolución y mejora continua.

La aplicación original, necesita evolucionar para poder enfrenta nuevos desafíos de rendimiento y escalabilidad a medida que crecen sus demandas. Este estudio aborda estos desafíos a través de la introducción la Observabilidad con OpenTelemetry. Esta herramienta permite un análisis mediante la recopilación de métricas, trazas y logs, revelando así áreas de mejora.

La adopción del patrón CQRS emerge como una solución estratégica para estos retos, separando las operaciones de lectura y escritura, permitiendo mejorar tanto el rendimiento como la escalabilidad. La sinergia entre el patrón CQRS, la Observabilidad con OpenTelemetry y una arquitectura Microservicios nos permite rastrear una solicitud a lo largo de todo su recorrido, proporcionando una visión clara de cómo interactúan los diferentes servicios y dónde pueden surgir los cuellos de botella o errores.

Los resultados obtenidos destacan no solo la mejora en el rendimiento y la escalabilidad, sino también la importancia de un diseño y una arquitectura iniciales robustos que permiten adaptaciones y mejoras continuas. La aplicación evoluciona para manejar de manera más eficiente las consultas y operaciones de base de datos, mejorando la experiencia del usuario y su capacidad de adaptación a nuevas demandas.

Este trabajo no solo subraya la importancia de la Observabilidad en arquitecturas de Microservicios y la efectividad de CQRS en la optimización de aplicaciones existentes, sino que también establece un marco para la evaluación y mejora continua de otras aplicaciones con desafíos similares.

# Introducción y objetivos

## Contexto del proyecto y su relevancia

En la era actual de la tecnología, la capacidad de una aplicación para adaptarse y escalar eficientemente es de suma importancia. Este proyecto pretende empoderar y dotar de de herramientas que nos permitan obtener información  para la correcta toma de decisiones que permitan la evolución de nuestras aplicaciones, un campo que cobra cada vez más relevancia en el ámbito del desarrollo de software. Con la creciente complejidad de los sistemas y las demandas de rendimiento y escalabilidad, la implementación de soluciones adecuadas y eficientes se convierte en una necesidad imperante.

## Problemática Inicial y necesidad de evolución de la aplicación

La aplicación inicial, diseñada para pocos usuarios como Producto Mínimo Viable (MVP), funcionaba eficientemente bajo un conjunto específico de condiciones. Sin embargo, a medida que las demandas del sistema aumentaban, se hacía evidente la necesidad de evolucionar la aplicación para mejorar su rendimiento y capacidad de escalabilidad. La principales problemáticas radicaba en alta demanda de consultas complejas a bases de datos que implicaba el acceso a muchas tablas relacionadas, por otro lado, existían cuellos de botella en operaciones de lectura y escritura donde las consultas tenían mucho más peso.

## Objetivo

El objetivo principal de este trabajo es demostrar cómo la implementación de la Observabilidad en arquitecturas monolíticas y Microservicios con OpenTelemetry, en combinación con el patrón CQRS, puede ayudarnos a optimizar y escalar una aplicación de software. Los objetivos específicos incluyen:

- Realizar un análisis de la aplicación existente para identificar áreas clave de mejora.
- Implementar OpenTelemetry para mejorar la Observabilidad, permitiendo un análisis exhaustivo a través de métricas, trazas y logs.

- Aplicar el patrón CQRS para optimizar las operaciones de lectura y escritura, mejorando así el rendimiento y la escalabilidad.
- Implementar una Arquitectura de Microservicios añadiendo Observabilidad para poder realizar un rastreo de las solicitudes a lo largo de todo su recorrido por los diferentes Microservicios.
- Evaluar y comparar el rendimiento y la escalabilidad antes y después de implementar estas mejoras.
- Establecer una guía para la aplicación de estas técnicas en otros proyectos con desafíos similares.

# Metodología Empleada para el Desarrollo del Proyecto

**Revisión de la Aplicación Existente:** Inicialmente, se realizó una revisión exhaustiva de la aplicación original housekeeping-service, incluyendo su arquitectura, tecnologías utilizadas, y estructura de código. Esta revisión tenía como objetivo entender completamente la base sobre la cual se construirían las mejoras.

**Integración de OpenTelemetry para Observabilidad:** Se implementó OpenTelemetry para mejorar la Observabilidad de la aplicación. Esta fase incluyó la configuración de colectores y la integración con el APM Signoz para un monitoreo efectivo.

**Identificación de Problemas y Áreas de Mejora:** Utilizando los datos recopilados a través de OpenTelemetry, se identificaron los cuellos de botella y las áreas donde la aplicación podría optimizarse, centrándose en problemas de rendimiento y escalabilidad.

**Diseño e Implementación de Mejoras:** Basándose en el análisis anterior, se diseñaron e implementaron mejoras. Esto incluyó la adopción del patrón CQRS, el uso de Debezium y Kafka para la gestión de eventos, y la creación de nuevos Microservicios, como el frontend-service en Node.js.

**Evaluación del Rendimiento y Escalabilidad:** Se realizaron pruebas de carga y se evaluó el rendimiento y la escalabilidad de la aplicación mejorada en comparación con la versión original. Esta evaluación utilizó tanto herramientas de análisis de rendimiento como métricas recopiladas a través de OpenTelemetry.

**Documentación y Análisis de Resultados:** Finalmente, se documentaron todas las fases del proyecto, analizando los resultados obtenidos y elaborando conclusiones. Este análisis también incluyó la identificación de limitaciones y la propuesta de futuras líneas de investigación y desarrollo.

# Arquitectura Inicial, Tecnologías y Limitaciones

## Arquitectura y Tecnología

La primera versión del software, housekeeping-service, está diseñada con Java Spring Boot 3.15, estableciendo una base sólida para futuros Microservicios. La Arquitectura se basa en una estructura por dominio con subdivisiones técnicas, utilizando DTOs y entidades mapeadas a la base de datos con JPA. La inclusión de varias dependencias en el fichero pom.xml refleja la complejidad y la modularidad de la aplicación.

## Modelo de gestión de código fuente

El control de versiones se realiza con un enfoque de monorepo para facilitar la gestión de múltiples servicios y librerías, proporcionando un entorno de desarrollo cohesivo y centralizado. Esta elección ha sido fundamental para mantener la consistencia y la eficiencia en todo el proceso gracias a las siguientes ventajas:

- Centralización: Todos los proyectos o componentes del software están en un solo lugar, facilitando la gestión y el seguimiento de cambios.
- Integración Continua: Los cambios en cualquier parte del repositorio pueden ser probados de manera integral, asegurando que los cambios en un proyecto no afecten negativamente a otros.

- Facilidad de Compartir Código: Debido a que todo el código está en un solo lugar, es más fácil compartir bibliotecas o dependencias comunes entre diferentes proyectos.
- Historial Unificado: El historial de cambios de todos los proyectos se mantiene en un solo lugar, lo que facilita la búsqueda y comprensión de las evoluciones y modificaciones a lo largo del tiempo.

## Identificadores de Modelos

Los identificadores de los modelos han sido cuidadosamente seleccionados para optimizar el rendimiento en diferentes contextos. Se ha utilizado TSID para modelos que requieren acceso cronológico, aprovechando su ordenación por tiempo. Por otro lado, se han empleado UUIDs para aquellos modelos donde la unicidad global y la independencia temporal son primordiales.

## Limitaciones encontradas

Se identifican las limitaciones de rendimiento y escalabilidad de la aplicación original. Estas limitaciones se descubren a través de la Observabilidad limitada y la estructura monolítica que, aunque eficiente inicialmente, no escala adecuadamente con el aumento de la demanda.

# Tecnologías utilizadas.

## Observabilidad y Microservicios

La Observabilidad es un aspecto relevante en la gestión de sistemas distribuidos como son la arquitecturas de Microservicios. Se refiere a la capacidad de medir y monitorear internamente un sistema, permitiendo comprender su estado, rendimiento y comportamiento en tiempo real. Esto es esencial en arquitecturas de Microservicios debido a su naturaleza compleja y distribuida.

## Ventajas de la Observabilidad en Microservicios

**Monitoreo en Tiempo Real:** Permite rastrear el comportamiento de los Microservicios en tiempo real, facilitando la identificación rápida de problemas y cuellos de botella.

**Detección y Solución de Problemas:** Ayuda a detectar fallos, fluctuaciones de rendimiento y vulnerabilidades de seguridad de manera eficiente, minimizando el tiempo de inactividad.

**Rastreo Distribuido:** En sistemas de Microservicios, donde diferentes componentes pueden estar distribuidos en varios servidores o incluso geografías, la Observabilidad permite rastrear y analizar llamadas entre servicios para identificar problemas de latencia o fallos en la comunicación.

**Escalabilidad y Flexibilidad:** La Observabilidad apoya la escalabilidad al proporcionar datos para tomar decisiones informadas sobre dónde y cuándo escalar.

**Mejora Continua:** Los datos recogidos facilitan la optimización continua del sistema, mejorando el rendimiento y la eficiencia.

## OpenTelemetry frente a Otras Soluciones

OpenTelemetry es un conjunto de APIs, bibliotecas y herramientas diseñadas para capturar, procesar y exportar datos de telemetría (métricas, trazas y logs). Es un proyecto de la Cloud Native Computing Foundation (CNCF) que proporciona una implementación unificada para una amplia variedad de lenguajes y frameworks.

## Ventajas de OpenTelemetry

**Interoperabilidad y Estándares Abiertos:** OpenTelemetry promueve un enfoque estándar para la recopilación de datos de telemetría, lo que facilita la integración con diversas herramientas y plataformas.

**Soporte Multi-Lenguaje:** OpenTelemetry ofrece soporte para una amplia gama de lenguajes de programación, lo que es ideal para entornos de Microservicios donde diferentes servicios pueden estar escritos en diferentes lenguajes.

**Flexibilidad en la Exportación de Datos:** Permite exportar datos a una variedad de backends y herramientas de análisis, dando a los equipos la libertad de elegir las mejores herramientas para sus necesidades.

**Facilidad de Uso y Configuración:** Tiende a ser más fácil de configurar y usar en comparación con soluciones específicas de lenguajes o frameworks, en muchas ocasiones, permite la auto-instrumentación sin necesidad de modificar el código de la aplicación a observar.

### Comparación con Alternativas

**Java - Micrometer:** Micrometer es una biblioteca de métricas para aplicaciones JVM que ofrece integración con Spring Boot. Aunque es potente y bien integrada con el ecosistema de Spring, su uso se limita principalmente a aplicaciones Java. OpenTelemetry, por otro lado, ofrece una solución más universal y adaptable a múltiples lenguajes.

**Node.js:** En el ecosistema de Node.js, existen varias bibliotecas para Observabilidad, como node-prometheus, que son específicas para Node.js. OpenTelemetry, siendo una solución transversal, brinda una ventaja al permitir una consistencia en la recolección y el análisis de telemetría a través de diferentes lenguajes y plataformas.

Como conclusión, OpenTelemetry se destaca por su enfoque en estándares abiertos, interoperatividad y soporte multi-lenguaje, lo que lo hace una opción robusta y flexible para la Observabilidad en arquitecturas de Microservicios, especialmente en entornos heterogéneos donde se utilizan múltiples tecnologías y lenguajes.

## Herramienta de Monitoreo de Rendimiento de Aplicaciones (APM) - Signoz

Para mejorar la monitorización y el análisis del rendimiento de nuestras aplicaciones, hemos optado por implementar el APM Signoz. Esta elección se basó en la facilidad de integración que ofrece Signoz.

Signoz se integra de manera nativa con los colectores de OpenTelemetry, facilitando la recopilación y el procesamiento de datos de telemetría de manera centralizada. Esta compatibilidad con OpenTelemetry es fundamental, ya que permite recoger métricas, trazas y logs de diversas fuentes de forma estandarizada y eficiente.

Además, Signoz viene equipado con un frontend intuitivo, que permite visualizar y analizar los datos recopilados de manera clara y accesible. Esta interfaz facilita la identificación rápida de problemas de rendimiento, cuellos de botella, permitiendo a los equipos de desarrollo y operaciones tomar decisiones informadas basadas en datos reales.

## Pruebas de Carga

Para las pruebas de carga en nuestra aplicación, hemos elegido utilizar Locust, una herramienta versátil desarrollada en Python. Locust permite simular comportamientos de usuarios reales, proporcionando una manera eficaz de evaluar el rendimiento de la aplicación bajo diversas condiciones de uso. Su capacidad para crear escenarios de prueba personalizados y su interfaz intuitiva lo hacen ideal para identificar y resolver posibles problemas de rendimiento antes de que afecten la experiencia del usuario

## CQRS y Microservicios - Debezium, PostgreSQL y Kafka

La combinación de CQRS (Command Query Responsibility Segregation) con un sistema de Captura de Cambios de Datos (CDC) mejora significativamente el rendimiento y la escalabilidad de las aplicaciones. CQRS permite optimizar separadamente las operaciones de lectura y escritura, mientras que el CDC asegura una sincronización en tiempo real y eficiente entre estas operaciones, manteniendo la consistencia y actualización de los datos en todo el sistema

### Ventajas de utilizar un CDC con Sistema de Mensajería.

Permite capturar y transmitir cambios de la base de datos en tiempo real. Esto es crucial para mantener la base de datos de consulta (Query) sincronizada con la base de datos de

comando (Command) en una arquitectura CQRS, asegurando una vista consistente y actualizada de los datos.

**Desacoplamiento y Escalabilidad:** Al utilizar un sistema de mensajería, CDC facilita un desacoplamiento efectivo entre los sistemas de producción de datos (bases de datos de comando) y los sistemas de consumo (bases de datos de consulta y otras aplicaciones). Esto mejora significativamente la escalabilidad y la capacidad de manejar grandes volúmenes de tráfico y datos.

**Resiliencia y Tolerancia a Fallos:** CDC garantiza la entrega de mensajes incluso en caso de fallos de red o del sistema. Un sistema de mensajería proporciona durabilidad y tolerancia a fallos a través de su sistema de replicación y almacenamiento distribuido.

**Consistencia Eventual y Reducción de Carga en el Sistema Principal:** Al mantener separadas las operaciones de lectura y escritura, se reduce la carga en el sistema principal y se asegura una consistencia eventual, crucial en sistemas distribuidos de gran escala.

## Debezium y Kafka en Comparación con Otras Soluciones

Para la Captura de Datos en Tiempo Real (CDC), hemos elegido Debezium debido al desacoplamiento que ofrece en comparación con otras opciones, como Apache Kafka Connect. A diferencia de Kafka Connect, que está intrínsecamente vinculado a Kafka, Debezium proporciona mayor flexibilidad y puede integrarse con varios sistemas de mensajería y almacenamiento de datos.

Por otro lado, hemos seleccionado Apache Kafka como nuestro sistema de mensajería principal. Kafka se destaca por su amplio ecosistema de herramientas y una robusta comunidad de soporte. Ofrece una gran variedad de soluciones y extensiones, lo que lo convierte en una opción preferida para gestionar flujos de datos en tiempo real y a gran escala. Su arquitectura escalable y tolerante a fallos asegura un procesamiento de

mensajes eficiente y confiable, lo que lo hace ideal para aplicaciones críticas y de alto rendimiento

## Elección de una Base de Datos en Memoria: Redis

Para nuestra base de datos en memoria, hemos seleccionado Redis debido a su naturaleza de código abierto y su rendimiento excepcionalmente alto. Redis no solo maneja las típicas estructuras de datos clave-valor, sino que también soporta una variedad de estructuras más complejas. Este aspecto lo convierte en una herramienta versátil y adaptable a diferentes necesidades.

Un aspecto clave de Redis es su capacidad de ofrecer persistencia de datos. Esta característica garantiza que la información se mantenga segura y accesible incluso después de reinicios del sistema, lo que es crucial para la integridad y la confiabilidad de los datos. Además, Redis proporciona mecanismos avanzados de replicación y escalabilidad, lo que asegura una alta disponibilidad del servicio, un factor importante para sistemas que requieren acceso constante y rápido a los datos.

Otro punto a destacar de Redis son sus características transaccionales y de publicación-suscripción. Estas funcionalidades son especialmente valiosas para el desarrollo de futuras características y funcionalidades en aplicaciones, permitiendo un manejo eficiente de los datos y una comunicación en tiempo real entre diferentes componentes del sistema. En resumen, Redis se presenta como una solución robusta y eficiente para manejar datos en memoria, adaptándose a las crecientes demandas de aplicaciones modernas

# Implementación de Observabilidad con OpenTelemetry: Estrategias para la Identificación de Problemas

Para instrumentar una aplicación utilizando OpenTelemetry, se presentan dos enfoques principales:

**Instrumentación Automática:** Este método implica adjuntar un agente de OpenTelemetry al iniciar la aplicación. La ventaja principal de este enfoque es la facilidad con la que se

puede capturar telemetría, ya que no requiere cambios en el código fuente de la aplicación. Esta opción es ideal para aquellos casos donde se desea una implementación rápida y eficiente, minimizando la intervención manual y maximizando la cobertura de telemetría con el menor esfuerzo posible.

**Instrumentación Manual:** Este enfoque implica la integración manual del código de OpenTelemetry en la aplicación. Aunque es más laborioso, ofrece un control más detallado y granular sobre qué datos se recopilan y cómo se procesan. La instrumentación manual es particularmente útil en situaciones donde se requiere una personalización específica o cuando se necesita recopilar métricas y trazas muy específicas que no son capturadas por la instrumentación automática.

Ambas opciones tienen sus propias ventajas y pueden ser seleccionadas en función de las necesidades específicas del proyecto y los recursos disponibles. La elección entre instrumentación automática y manual dependerá de factores como la complejidad de la aplicación, los requisitos de Observabilidad específicos y el nivel de personalización deseado en la recopilación de datos.

## Instrumentación automática con OpenTelemetry en Spring Boot

A continuación se describe los pasos mínimo para la instrumentación automática:

- Descargar el agente Java de OpenTelemetry. OpenTelemetry GitHub.
- Agregar el Agente de OpenTelemetry a la Ejecución de tu Aplicación
- Una vez descargado el agente, hay que agregarlo a la ejecución de la aplicación Spring Boot. Esto se consigue pasando el agente como un argumento de JVM al iniciar la aplicación. Por ejemplo: `java -javaagent:/ruta/a/OpenTelemetry-javaagent-all.jar -jar AppSpringBoot.jar -Dotel.exporter.otlp.endpoint=${OpenTelemetry.endpoint}-Dotel.resource.attributes=service.name=${project-name}`

En nuestro proyecto, para el entorno de desarrollo, no es necesario porque se ha automatizado gracias a Maven.

# Instrumentación en Node.js: Uso de la Pre-carga de Módulos

Para instrumentar nuestra aplicación en Node.js, vamos a utilizar la técnica de Pre-carga de Módulos. Esta estrategia asegura que ciertas configuraciones o instrumentaciones se apliquen antes de que el resto de la aplicación en Node.js se ejecute. El módulo que pre-cargaremos será el responsable de configurar la telemetría.

Un ejemplo práctico de esto sería iniciar la aplicación con un comando específico que incluya la pre-carga del módulo de instrumentación. Por ejemplo, podríamos usar el comando `node --require ./src/instrumentation.cjs ./src/app.js`. En este caso, `./src/instrumentation.cjs` sería el módulo que contiene toda la configuración necesaria para la telemetría.

Esta metodología ofrece varias ventajas:

**Centralización de la Configuración:** Al pre-cargar el módulo de instrumentación, se centraliza la configuración de la telemetría, facilitando su gestión y actualización.

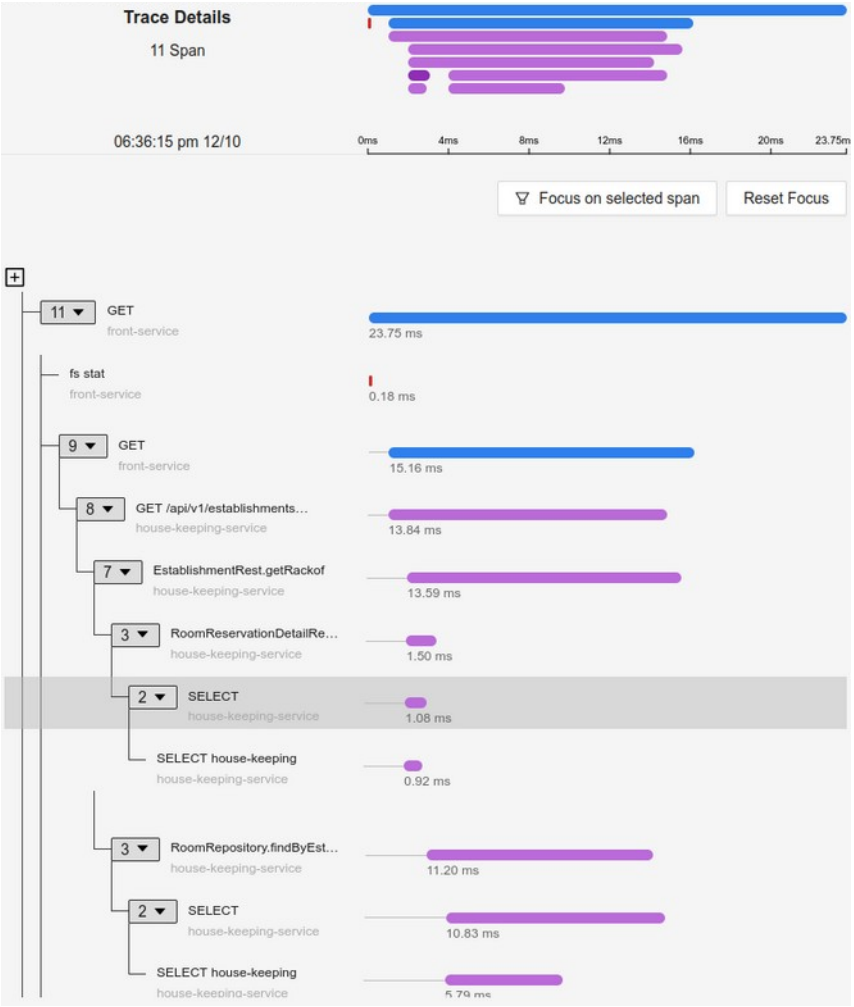
**Consistencia en la Instrumentación:** Se garantiza que la instrumentación se aplique de manera uniforme en toda la aplicación, lo cual es crucial para obtener datos coherentes y fiables.

**Eficiencia en el Rendimiento:** Al cargar la instrumentación antes de ejecutar la aplicación, se reduce la probabilidad de que se omitan datos importantes durante el proceso de arranque, optimizando así el rendimiento general de la telemetría.

## Diagnóstico de Problemas: Análisis Utilizando Observabilidad

Al utilizar las métricas, trazas y logs recopilados, se lleva a cabo un análisis detallado para identificar posibles cuellos de botella y áreas críticas que requieren mejoras en la aplicación. Este proceso implica una evaluación minuciosa de cada componente y su rendimiento en el entorno operativo.

Un ejemplo concreto de este diagnóstico es el análisis de la petición para obtener todas las habitaciones de un establecimiento. Se detecta que, de un tiempo total de respuesta de 23,75 milisegundos, el acceso a datos por sí solo consume 13,84 milisegundos. Este dato es significativo porque indica que más de la mitad del tiempo de respuesta se dedica a la interacción con la base de datos.




Este tipo de hallazgos son cruciales para optimizar el rendimiento de la aplicación. Al identificar que el acceso a datos es un cuello de botella importante en esta petición específica, se pueden tomar medidas para mejorar esta parte del proceso. Estas medidas pueden incluir la optimización de consultas, el uso de técnicas de caché, la revisión de índices en la base de datos, o incluso considerar una reestructuración de cómo se almacenan y recuperan los datos.

# Adopción de las soluciones

## Adopción del Patrón CQRS en el Desarrollo de la Aplicación

La aplicación inicial ya contaba con un diseño sólido, lo que ha permitido implementar mejoras de manera paralela, sin necesidad de interferir en su funcionamiento actual. A pesar de esta ventaja inicial, hay algunas consideraciones importantes que podríamos tener en cuenta en futuros desarrollos:

**Nombramiento de Entidades de Persistencia:** Nos encontramos con un desafío relacionado con el nombramiento de las entidades. Algunas entidades tenían atributos nombrados en formato 'camel case', pero JPA/Hibernate, al crear las tablas en la base de datos, convertía estos nombres al formato 'snake case'. Por ejemplo, un atributo llamado “isClean” en la aplicación se transformaba en “is_clean” en la base de datos. Esto ha requerido un proceso de conversión de estos nombres al momento de persistir los eventos en Redis, con el fin de cumplir con las especificaciones de la API.

**Integración con Debezium y Kafka:** La integración con Debezium y Kafka ha sido relativamente sencilla. Los ficheros de docker-compose proporcionan una guía clara sobre cómo se realiza esta integración. Debezium envía un evento a Kafka por cada operación de alta, baja o modificación en la base de datos, y crea un 'topic' específico para cada tabla. Esta funcionalidad es fundamental para el seguimiento eficiente de los cambios en los datos y facilita la implementación del patrón CQRS, al separar claramente las operaciones de lectura y escritura.

## Desarrollo del Servicio 'frontend-service'

El servicio denominado 'frontend-service' fue desarrollado utilizando Node.js, con el objetivo de demostrar la viabilidad de la instrumentación mediante OpenTelemetry en este entorno. Este servicio juega un papel crucial al centralizar todas las peticiones HTTP, redirigiendo los comandos al Backend 'housekeeping-service' y las consultas de lectura a la base de datos Redis.

Además, se ha integrado una interfaz web desarrollada con Vue.js, proporcionando una vista preliminar del aspecto visual de la aplicación.

Un aspecto destacable en la integración con Redis fue el descubrimiento, gracias a la telemetría, de que el paquete 'Redis' original no era eficiente al realizar consultas de varios registros. Este hallazgo llevó a un cambio en la librería de manejo de Redis, optando por 'ioredis' y aprovechando su funcionalidad de 'pipelines'. La implementación de 'pipelines' ha optimizado significativamente las consultas de datos, mejorando la eficiencia y el rendimiento de las operaciones de base de datos en el 'frontend-service'.

# Comparación entre la Arquitectura Original y la Nueva

## Metodología de Evaluación

Para evaluar y comparar el rendimiento y la escalabilidad de la versión original con la nueva implementación con CQRS, se diseñó un escenario de prueba de carga utilizando Locust. Este escenario buscaba simular el uso real de la aplicación en un entorno multiestablecimiento con los siguientes parámetros:

**Entorno de Prueba:** Simulación de 200 establecimientos, cada uno con 200 habitaciones.

**Perfil de Usuario:** En cada establecimiento, 18 usuarios realizando consultas y 5 usuarios ejecutando modificaciones (estado de las habitaciones como limpia, sucia, revisada).

**Carga de Trabajo:** Se realiza la creación de 20 usuarios en intervalos de 10 segundos, escalando hasta 500 usuarios durante un período de 10 minutos. Durante este tiempo, cada usuario realiza las peticiones con pausas entre 1 y 5 segundos.

Este enfoque permitió evaluar cómo cada arquitectura maneja una carga de trabajo intensiva y diversa, reflejando un entorno de uso realista.

## Resultados de la Evaluación

Los resultados de las pruebas de carga revelaron diferencias significativas en el rendimiento entre las dos arquitecturas:

**Arquitectura Original:** En la versión monolítica inicial, se observó un tiempo de respuesta promedio de 30ms. Este rendimiento, aunque adecuado, en detalles se aprecia que el tiempo de respuesta fluctuá entre 5ms y 485ms.

**Arquitectura Mejorada con CQRS y Opentelemetry:** La nueva arquitectura mostró una mejora notable, con un tiempo de respuesta promedio de solo 3ms. Este aumento en la eficiencia es atribuible a la separación de las operaciones de lectura y escritura a través de CQRS y Redis.

Los resultados detallados, incluyendo gráficos y análisis de datos, son presentados en los Anexos. Estos gráficos ilustran no solo los tiempos de respuesta, sino también cómo la nueva arquitectura maneja mejor las cargas de trabajo intensivas, manteniendo un rendimiento consistente y eficiente incluso bajo condiciones de alta demanda.

# Conclusiones y Proyecciones para Trabajos Futuros

La comparación entre las dos arquitecturas resalta claramente las ventajas de implementar CQRS y la integración de OpenTelemetry, especialmente en términos de rendimiento y escalabilidad. La mejora significativa en el tiempo de respuesta, de un orden de magnitud, enfatiza la eficacia de la nueva arquitectura para manejar un alto volumen de operaciones simultáneas, un aspecto crítico para aplicaciones en entornos con varios establecimientos, como el caso que nos ocupa.

Este estudio demuestra que, partiendo de una aplicación bien diseñada, con una arquitectura clara y siguiendo buenas prácticas de desarrollo, es posible lograr una evolución y mejora continua a medida que emergen nuevos requisitos.

Además, concluimos que incluso simples optimizaciones en las consultas a la base de datos pueden mejorar sustancialmente el rendimiento. La implementación de la arquitectura CQRS ha sido un paso para destacar estas ventajas.

## En que mejorar

Para trabajos futuros, se propone:

Enfocarse en optimizar las consultas de la base de datos y comparar los resultados obtenidos con los actuales.

Migrar la infraestructura y aplicación a Kubernetes.

Añadir funcionalidad para actualizaciones en tiempo real, se plantea implementar el sistema de publicación/suscripción de mensajes que ofrece Redis, complementado con una conexión WebSocket del lado del cliente.

# Biografia

Clean Architecture – Robert C. Marin

Clean Code – Robert C. Marin

The Pragmatic Programmer – David Thomas/Andrew Hunt

# Anexos

## Direcciones de referencia:

Documentación de Redis: https://redis.io/docs/interact/

Documentacion de Apache Kafka: https://kafka.apache.org/documentation/

Integración con OpenTelemetry: https://opentelemetry.io/docs/

Instrumentación a Signoz: https://signoz.io/docs/userguide/send-metrics/

Información general de CQRS: https://martinfowler.com/bliki/CQRS.html

CQRS con Redis: https://redis.com/solutions/use-cases/microservices/cqrs/

# Resultados de pruebas de carga

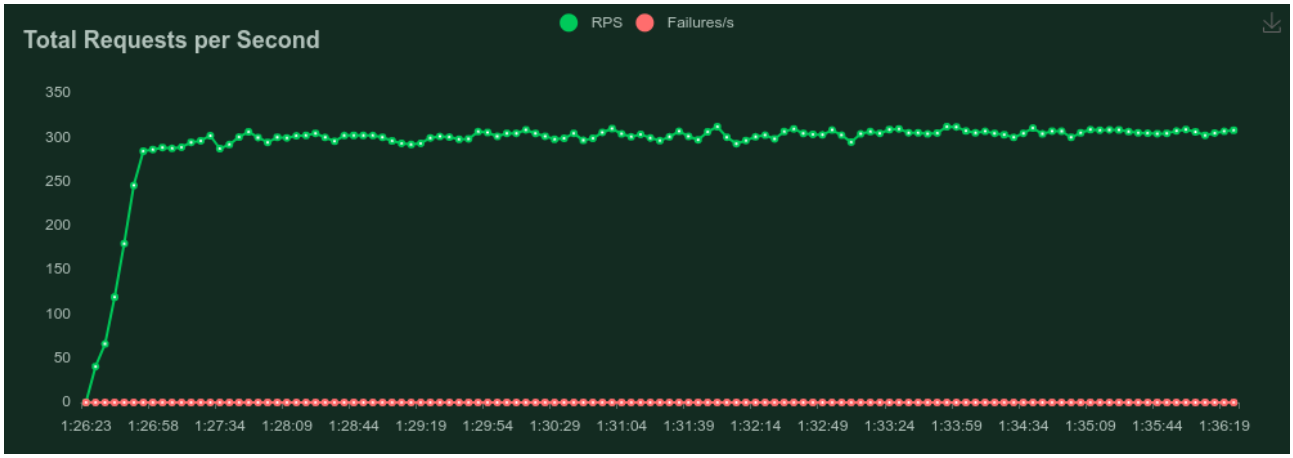
## Aplicación inicial

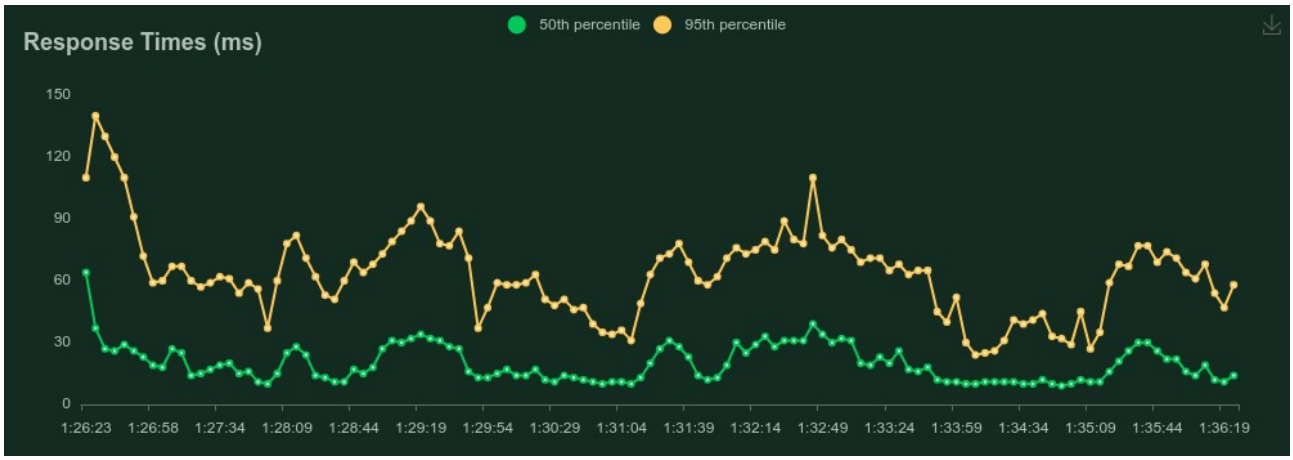
| Method | Name | # Requests | # Fails | Average (ms) | Min (ms) | Max (ms) | Average size (bytes) | RPS | Failures/s |
|---|---|---|---|---|---|---|---|---|---|
| PUT | /api/v1/establishments/[id]/room/[id]/clean | 32473 | 0 | 18 | 2 | 475 | 273 | 53.8 | 0.0 |
| PUT | /api/v1/establishments/[id]/room/[id]/dirty | 36508 | 0 | 18 | 3 | 457 | 274 | 60.5 | 0.0 |
| PUT | /api/v1/establishments/[id]/room/[id]/supervised | 12329 | 0 | 18 | 4 | 432 | 272 | 20.4 | 0.0 |
| GET | /api/v1/establishments/[id]/rooms | 97433 | 0 | 30 | 5 | 485 | 54839 | 161.5 | 0.0 |

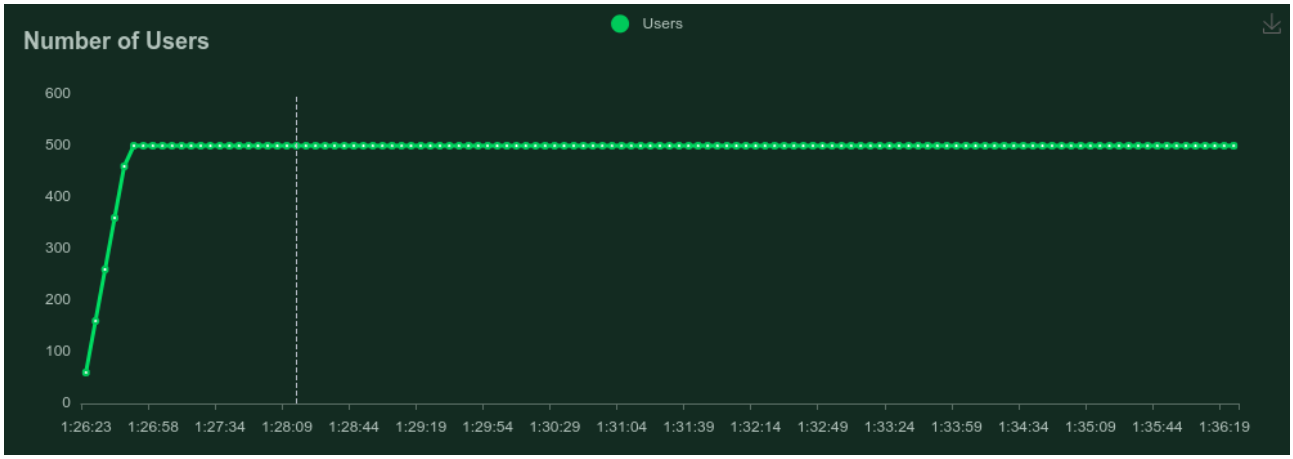
*Figura 1: Request Statistics v1*

| Method | Name | 50%ile (ms) | 60%ile (ms) | 70%ile (ms) | 80%ile (ms) | 90%ile (ms) | 95%ile (ms) | 99%ile (ms) | 100%ile (ms) |
|---|---|---|---|---|---|---|---|---|---|
| PUT | /api/v1/establishments/[id]/room/[id]/clean | 13 | 16 | 20 | 26 | 36 | 46 | 75 | 480 |
| PUT | /api/v1/establishments/[id]/room/[id]/dirty | 13 | 16 | 20 | 26 | 37 | 46 | 75 | 460 |
| PUT | /api/v1/establishments/[id]/room/[id]/supervised | 13 | 16 | 20 | 26 | 37 | 46 | 78 | 430 |
| GET | /api/v1/establishments/[id]/rooms | 23 | 30 | 38 | 48 | 62 | 74 | 110 | 490 |
| | Aggregated | 17 | 22 | 29 | 39 | 53 | 66 | 98 | 490 |

*Figura 2: Response Time Statistics v1*





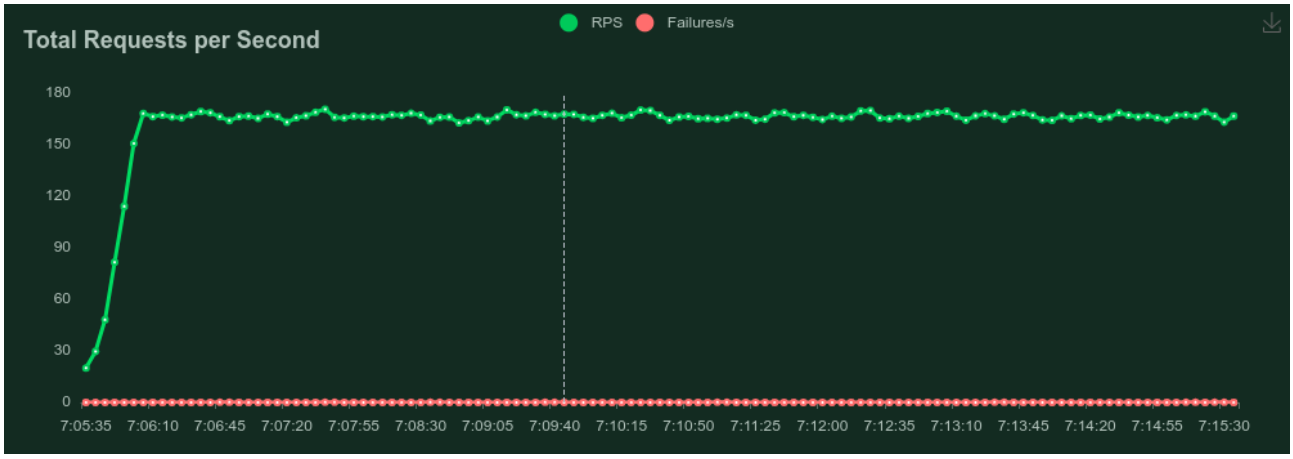



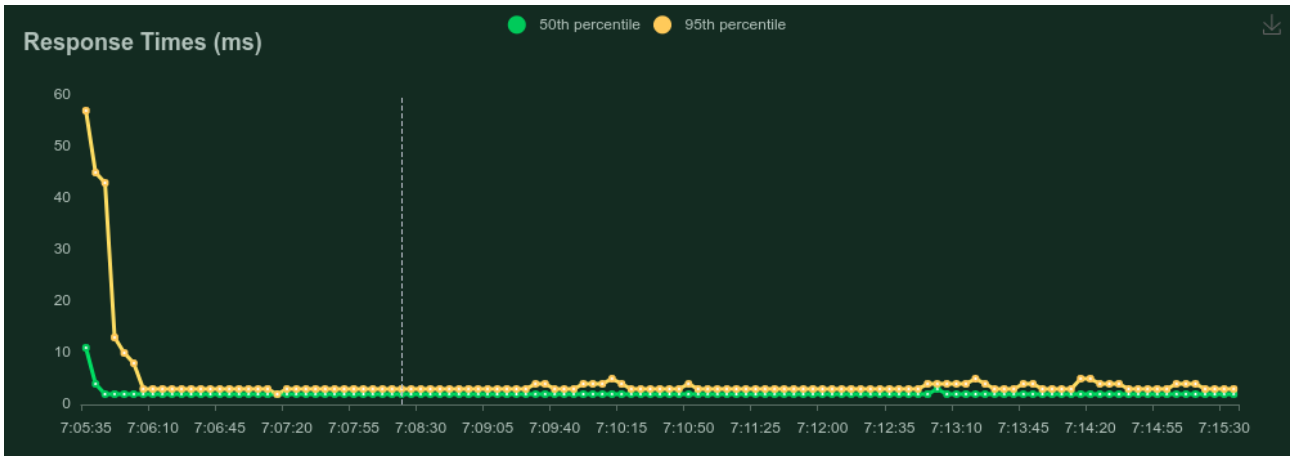

## Aplicación CQRS

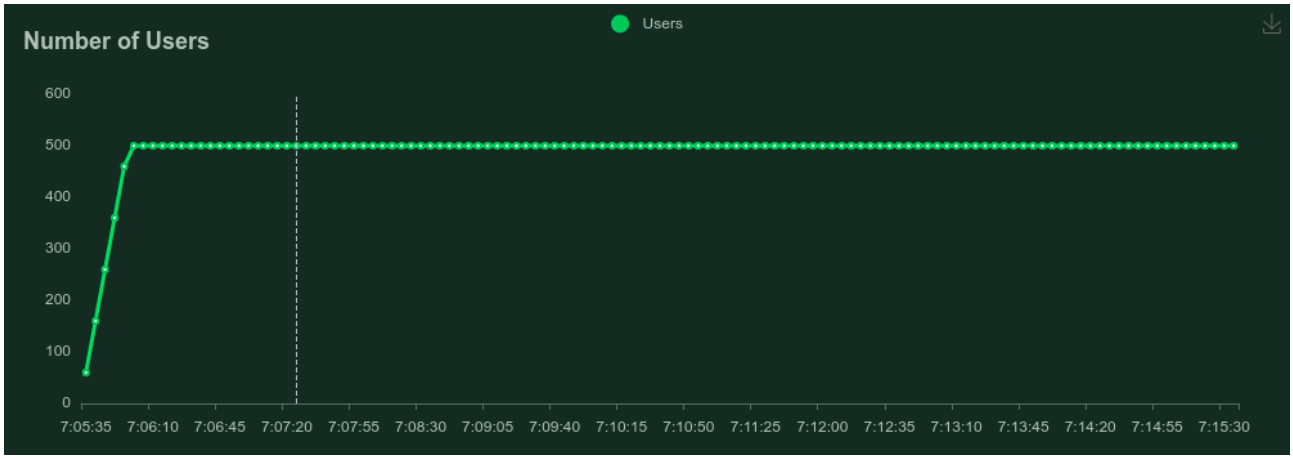
| Method | Name | # Requests | # Fails | Average (ms) | Min (ms) | Max (ms) | Average size (bytes) | RPS | Failures/s |
|---|---|---|---|---|---|---|---|---|---|
| PUT | establishments/[id]/room/[id]/dirty | 203 | 0 | 12 | 7 | 268 | 275 | 0.3 | 0.0 |
| PUT | establishments/[id]/room/[id]/supervised | 73 | 0 | 11 | 8 | 21 | 272 | 0.1 | 0.0 |
| PUT | establishments/[id]/room/[id]/clean | 173 | 0 | 11 | 7 | 60 | 274 | 0.3 | 0.0 |
| GET | /api/v1/establishments/[id]/rooms | 98540 | 8 | 2 | 0 | 71 | 369 | 163.1 | 0.0 |

*Figura 3: Request Statistics v2*

| Method | Name | 50%ile (ms) | 60%ile (ms) | 70%ile (ms) | 80%ile (ms) | 90%ile (ms) | 95%ile (ms) | 99%ile (ms) | 100%ile (ms) |
|---|---|---|---|---|---|---|---|---|---|
| PUT | /api/v1/establishments/[id]/room/[id]/clean | 10 | 10 | 11 | 12 | 14 | 20 | 35 | 60 |
| PUT | /api/v1/establishments/[id]/room/[id]/dirty | 10 | 11 | 11 | 12 | 15 | 17 | 30 | 270 |
| PUT | /api/v1/establishments/[id]/room/[id]/supervised | 10 | 11 | 12 | 12 | 15 | 16 | 21 | 21 |
| GET | /api/v1/establishments/[id]/rooms | 2 | 2 | 2 | 2 | 3 | 3 | 6 | 72 |
| | Aggregated | 2 | 2 | 2 | 2 | 3 | 3 | 8 | 270 |



# Flujo de telemetría con OpenTelemetry y Sinoz