UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

REFACTORIZACIÓN E IMPLEMENTACIÓN DE PRUEBAS UNITARIAS EN EL MOTOR DE TRANSACCIONES DIGITALES DE INSTANCE

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

SANTIAGO FRANCISCO JOSÉ ROJAS ILLANES

PROFESORA GUÍA:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
DIONISIO GONZÁLEZ GONZÁLEZ

SANTIAGO DE CHILE
2023

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL
EN COMPUTACIÓN
POR: SANTIAGO FRANCISCO JOSÉ
ROJAS ILLANES
FECHA: 2023
PROF. GUÍA: JOCELYN SIMMONDS

REFACTORIZACIÓN E IMPLEMENTACIÓN DE PRUEBAS UNITARIAS EN EL MOTOR DE TRANSACCIONES DIGITALES DE INSTANCE

El Motor de Transacciones Digitales de Instance se creó respondiendo a la necesidad de Instance de obtener y procesar órdenes de todos los canales de venta que sus clientes estuvieran integrados. Esta necesidad de automatizar el proceso de venta digital fue acelerada inesperadamente por la pandemia.

Es por esto que el Motor de Transacciones Digitales (MTD), fue construido de manera muy rápida y respondiendo a requerimientos sin tomar en cuenta mantenibilidad, escalabilidad y eficiencia para poder obtener lo antes posible una plataforma funcional. Y si bien la plataforma funciona y soluciona el problema de Instance, es ahora cuando la empresa se está abriendo a mercados internacionales e integrando cada vez más tiendas, sistemas de ecommerce, marketplaces y facturadores entre otros, donde esta deuda técnica está cobrando especial relevancia.

Estos problemas fueron abordados colectivamente mediante una profunda refactorización del código del MTD y de la base de datos. Logrando mantener su funcionalidad, pero estableciendo mejoras de rendimiento, buenas prácticas y correcciones, que permitan a futuro mantener y escalar el software de forma simple y a pruebas de errores mediante las pruebas unitarias.

Con los cambios realizados, ahora se puede ingresar un nuevo cliente sin intervención del equipo de desarrollo y las integraciones de nuevas plataformas al MTD solamente se deben realizar una sola vez y sobre campos ya conocidos y definidos. Además, ahora el sistema tiene un mejor rendimiento, reduciendo los tiempos y costos de ejecución.

Teniendo en cuenta el objetivo y los problemas de la plataforma al iniciar el proyecto y el estado actual en el cual se encuentra la plataforma, se cumplieron todos los objetivos propuestos. Las validaciones muestran una mejor mantenibilidad, escalabilidad y rendimientos a través de toda la plataforma.

Any fool can write code that a computer can understand.

Good programmers write code that humans can understand.

Martin Fowler

Agradecimientos

Agradezco en primer lugar a mis padres y mi familia, que me apoyaron y acompañaron a lo largo de toda la carrera. A mis hermanos, especialmente a María Teresa por renovar mi motivación por la ingeniería en computación. Gracias a todos por su comprensión, consejos y paciencia.

Le doy las gracias a mis amigos, Armando, Andrés, Agustín, Domingo, Max, Felipe, Fabián, Maximiliano, Ignacio, Pablo, Sergio y Alejandro. Además, quisiera agradecer especialmente a Juan Pablo, Raimundo y Sebastián, por compartir el departamento conmigo y acompañarme en esta etapa. Gracias a todos por las risas y los momentos compartidos.

Agradezco a María Josefa Sánchez por acompañarme en la recta final de la memoria, por su paciencia, por las risas y los consejos. Realmente fue el apoyo que necesitaba para terminar esta etapa.

Mis agradecimientos a Instance y el equipo de desarrollo, Julián, Mauricio, Matías, Rodrigo y Ricardo. Sin ustedes no habría sido posible esta memoria.

Finalmente, agradezco a mi profesora guía, Jocelyn Simmonds, por toda su ayuda y disposición.

Tabla de Contenido

Capítulo 1 Introducción **1**
1.1. Contexto 1
1.2. Problema 2
1.3. Objetivos 3
1.3.1. Objetivo General 3
1.3.2. Objetivos Específicos 3
1.4. Solución 4
1.5. Metodología 4
1.6. Estructura del Documento 5
Capítulo 2 Estado del Arte **6**
2.1. Experiencias con refactorizaciones industriales 6
2.2. Lenguajes, Frameworks y Tecnologías 8
2.2.1. Javascript 8
2.2.2. Node.js 9
2.2.3. MongoDB 9
2.2.4. Mocha.js y Chai.js 11
2.2.5. Herramientas 11
2.2.5.1. JetBrains Webstorm 11
2.2.5.2. SonarLint 12
2.2.5.3. perf_hooks.js 12
Capítulo 3 Análisis y Diseño **14**
3.1. Flujo de Información en el MTD 14
3.2. Diseño 15
3.3. Análisis 16
3.3.1. Código 17
3.3.2. Base de Datos 18
Capítulo 4 Implementación **19**
4.1. Alternativa de Ejecución 19
4.1.1. Github Actions 20
4.1.2. AWS: ECS Schedule Tasks 20
4.1.3. AWS: CloudWatch Events y Lambda 21

4.2. Base de Datos 21
4.2.1. Reestructuración de la base de datos 22
4.2.2. Conexiones a la base de datos 24
4.3. Código 24
4.3.1. Documentación y buenas prácticas 25
4.3.2. Reestructuración del proyecto 25
4.3.3. Parametrización del código 27
4.3.4. Ejecución asíncrona 27
4.3.4.1. Definición de las promesas 27
4.3.4.1.1. Antipatrón de promesas 28
4.3.4.2. Solución totalmente asíncrona 28
4.3.4.3. Solución parcialmente asíncrona 29
4.3.5. Refactorización del flujo de órdenes 31
4.3.5.1. Modelo de orden unificada 32
4.3.5.2. Estructura del nuevo flujo de órdenes 32
4.3.6. Preparación para la integración de pruebas unitarias 33
4.4. Pruebas Unitarias 34
4.5. Despliegue 34
Capítulo 5 Validación 36
5.1. Estado después de los cambios 36
5.1.1. Escalabilidad 36
5.1.2. Mantenibilidad 37
5.1.3. Rendimiento 37
5.2. Resultados Cuantitativos 38
5.2.1. Tiempos de ejecución 38
5.3. Resultados Cualitativos 42
5.3.1. Reportes de Sonarlint 42
5.3.2. Comentarios equipo de desarrollo 42
5.3.2.1. Pérdida de familiaridad con el código 42
5.3.2.2. Conocimiento del código 43
5.3.2.3. Calidad técnica de la solución 44
5.3.3. Mejores prácticas 45
Capítulo 6 Conclusiones 46
6.1. Trabajo Futuro 47
Bibliografía 48
Anexo A 49
Diagramas de flujo de órdenes de Instance 49

Capítulo 1

Introducción

En este capítulo se introducirán a los principales actores de la memoria. Se presenta el contexto y el problema, se definirán los objetivos y se expone la solución propuesta y metodología a seguir. Finalmente se presenta la estructura general del documento.

En la sección 1.1 se presenta el contexto de MTD de Instance, el cual gestiona ventas digitales en diversos canales, coordinando las órdenes y envíos a través de API. Luego, en la sección 1.2, se presenta el problema identificado, la deuda técnica acumulada del MTD afecta la estabilidad y escalabilidad. Después, en la sección 1.3, se definen los objetivos con respecto a la deuda técnica. Para luego, en la sección 1.4, presentar la solución propuesta al problema, la refactorización e implementación de pruebas unitarias en el MTD, y en la sección 1.5, presentar la metodología a seguir para la implementar la solución. Finalmente, en la sección 1.6, detalla la estructura que seguirá el documento.

1.1. Contexto

Instance [1] es una empresa cuyo rubro es lo que se denomina el fullcommerce [2], lo cual se traduce en la gestión completa de ventas digitales. Esto significa que Instance se hace cargo de la publicación de productos, del mantenimiento del stock, de la facturación, del bodegaje y del envío al cliente final. Esto para cada una de las diferentes tiendas en todos los canales de venta acordados.

El proceso descrito es realizado conectándose a las API de todas las tiendas digitales y marketplaces de los clientes de Instance, tales como: MercadoLibre, Falabella o Shopify, entre otros. Todos estos lugares donde se publican los productos de los clientes serán llamados Canales de Venta.

Los clientes de Instance son empresas que desean publicar sus productos en múltiples plataformas digitales y esto se realiza con la ayuda del área comercial de Instance que se encargan de subir el catálogo de productos al PIM (Product Information Management) y conectan esta información con los canales de venta escogidos por el cliente, los cuales publican los productos y gestionan las compras por el cliente final.

Es el denominado Motor de Transacciones Digitales de Instance, por sus siglas MTD, el que une y coordina todas las órdenes desde los canales de venta a la bodega y el courier. El MTD se conecta a las API de todos los canales de venta para obtener las órdenes de compra de los clientes de Instance. De las órdenes se extrae la información relevante para la facturación y para la bodega, y se envían los datos a cada uno respectivamente.

Este flujo es recorrido por, aproximadamente, 20000 órdenes de compra con un valor aproximado de 45 millones de pesos solamente en Chile. Hoy en día hay 34 clientes operando con Instance en Chile, entre los que destacan Softys, Unilever y Carozzi. Estos clientes tienen sus productos publicados en 17 canales de venta.

Actualmente, Instance tiene presencia en Chile, Colombia y Perú, está empezando a integrarse a México y hay planes para ingresar al mercado estadounidense, entonces es una empresa en crecimiento la cual tendrá que integrar múltiples nuevos clientes, canales de venta, plataformas de facturación y bodegaje.

1.2. Problema

El Motor de Transacciones Digitales de Instance tiene una deuda técnica que se ha ido acumulando por más de un año, esto significa que la plataforma lleva todo este tiempo recibiendo soluciones parche y mucho código en duro, lo que dificulta la mantención a futuro y la adición de nuevas funcionalidades o integraciones. Por lo tanto, solucionar esta deuda técnica es necesario para hacer una plataforma estable, escalable y eficiente.

Es por esto que la plataforma tiene varios problemas que abordar, partiendo por el método de ejecución de la plataforma, el cual posee múltiples desventajas, ya que requiere muchas manualidades.

Por otro lado, la base de datos tiene problemas estructurales que se deben solucionar para mejorar profundamente la escalabilidad y mantenibilidad de la plataforma. Uno de los problemas es el almacenamiento de las configuraciones de los clientes en cada uno de los canales de venta, lo cual a su vez hace que sea muy complejo parametrizar el código y reemplazar el código en duro. También hay un muy alto número de conexiones concurrentes a la base de datos que no son necesarias, estas conexiones causan inestabilidad en la plataforma y aumentan el costo de la administración.

Además, el código mismo debe ser sujeto a cambios, ya que no está aprovechando las mejoras en rendimiento y en eficiencia que puede otorgar el código

asíncrono [3]. Así como también se pueden aplicar varias buenas prácticas para que el código tenga mejor mantenibilidad y legibilidad.

Durante el análisis del flujo que recorren las órdenes de compra dentro de la plataforma se identificó que mucho de este código está replicado para todos los canales de venta. Abriendo una oportunidad para optimizar el código, refactorizando el flujo entero de las órdenes de compra.

Finalmente, no hay ningún método de pruebas implementado y, por lo tanto, errores que serían fácilmente detectables son publicados en producción.

1.3. Objetivos

1.3.1. Objetivo General

El objetivo es refactorizar e integrar pruebas unitarias al Motor de Transacciones Digitales de Instance, ya que si bien esta es utilizable y funcional, el código tiene varias falencias técnicas que se deben solucionar previo al crecimiento explosivo que se prevé. La refactorización permitirá que nuevos clientes sean añadidos a la plataforma sin intervención de los desarrolladores y simplemente modificando la base de datos. Además, al introducir pruebas unitarias se podrán detectar errores rápidamente e integrar nuevos canales de venta, facturadores o WMS teniendo seguridad de que el resto de la plataforma se mantendrá operacional.

1.3.2. Objetivos Específicos

El objetivo general se descompone en los siguientes objetivos específicos:

1. Evaluar una mejor solución para ejecutar los scripts, dado que el actual método posee varias desventajas.
2. Mejorar el rendimiento y eficiencia en código asíncrono. Analizando el código se detectó que hay múltiples optimizaciones que se podrían realizar.
3. Parametrizar el código para recibir configuraciones desde la base de datos. Esto permite que el proceso de agregar un nuevo cliente sea simplificado, solamente requiriendo que se modifique la base de datos.
4. Definir mejores prácticas a seguir por el equipo de desarrollo. Esto busca implementar las mejores prácticas del lenguaje usado, así como mejorar la legibilidad y la documentación del código, definiendo reglas para nombramiento de variables, funciones, etc.

5. Reducir el número de conexiones concurrentes a la base de datos. Con una refactorización bien realizada se podrá disminuir en al menos un 50% el número de conexiones abiertas reduciendo la carga para el servidor.
6. Preparar la plataforma para integrar pruebas unitarias. Debido a la estructura actual de la plataforma no se pueden integrar pruebas unitarias y, por lo tanto, una refactorización del código orientado a permitir la creación de pruebas unitarias es necesario antes de escribir las mismas pruebas.
7. Escribir pruebas unitarias. Esto implicaría escribir pruebas unitarias para cada uno de los procesos críticos del negocio, como lo son las integraciones con los canales de venta, facturadores y WMSs.

Donde los primeros seis objetivos están orientados a una refactorización efectiva de la plataforma, mientras que el último objetivo tiene relación con la implementación de las pruebas unitarias.

1.4. Solución

La solución propuesta consiste en refactorizar el código actual del Motor de Transacciones Digitales de Instance manteniendo su funcionalidad, pero estableciendo mejoras de rendimiento, buenas prácticas y correcciones, que permitan a futuro mantener y escalar el software de forma simple y a pruebas de errores mediante las pruebas unitarias.

La solución contaría con tres aspectos a desarrollar: primero, modificaciones a la base de datos para que la estructura permita un mejor almacenamiento de las configuraciones de los clientes y reducir el número de conexiones a esta. Segundo, una profunda refactorización del código de la plataforma para mejorar el rendimiento, usando ejecución asíncrona y aplicando buenas prácticas. Además, se propone modificar el flujo completo de las órdenes de compra unificando para que un solo módulo sea el que contenga las integraciones a las otras plataformas. Y tercero, la creación e implementación de pruebas unitarias.

1.5. Metodología

La metodología utilizada para realizar la solución fue seguir un orden secuencial, de modo que cada etapa prepara y es necesaria para la siguiente. Lo primero fue la reestructuración de la base de datos, sin estos cambios, el código no se puede refactorizar correctamente. Para luego continuar con la refactorización del código de manera modular, de esta manera cada módulo ya refactorizado podría integrarse a la plataforma y ser subido a producción. Utilizando esta metodología de trabajo, cualquier

eventual error en los módulos podrían ser más rápidamente identificados, ya que no se reemplaza la plataforma actual de una sola vez.

Finalmente, luego de cada módulo refactorizado, se escribirán pruebas unitarias usando el framework Mocha.js que permite pruebas sobre Node.js para código asíncrono. Estas pruebas permitirán detectar errores en el código tempranamente y confirmar que el flujo completo de las órdenes de compra funciona correctamente.

1.6. Estructura del Documento

A continuación se detalla la estructura de este informe, listando los capítulos que lo componen, junto con un breve resumen de su enfoque:

- Capítulo 1: Introducción. Provee el contexto, los objetivos y el problema a solucionar.
- Capítulo 2: Estado del Arte. Se presentan las tecnologías y frameworks que serán utilizadas durante este trabajo de título.
- Capítulo 3: Análisis y Diseño. Se expone el estado del sistema, su arquitectura y las soluciones propuestas.
- Capítulo 4: Implementación. Se describe la implementación de la solución, las principales tareas desarrolladas y los mayores desafíos enfrentados.
- Capítulo 5: Validación. Se exponen los resultados obtenidos de la aplicación de la solución.
- Capítulo 6: Conclusiones. Se analizan los resultados obtenidos con respecto de los objetivos planteados, se exponen próximos trabajos a realizar.

Capítulo 2

Estado del Arte

En este capítulo, se profundizará en el estado del arte con respecto a las experiencias de la industria con refactorizaciones y, se identificarán y describirán los principales lenguajes, frameworks y herramientas.

En la sección 2.1 se analizarán las experiencias en la industria con refactorizaciones a gran escala, las prácticas, los aprendizajes y las herramientas que se han utilizado. Además, en la sección 2.2, se presentarán las principales herramientas que se utilizaran y los lenguajes de programación y frameworks que se utilizan en la plataforma, y por lo tanto, en la memoria.

2.1. Experiencias con refactorizaciones industriales

En la actualidad las plataformas no son estáticas y deben estar en constante mantención y evaluación, lo cual se puede ver reflejado en refactorizaciones a gran escala de plataformas para solucionar estos problema, es por esto que la refactorización de software es una práctica esencial en la industria. Y a medida que las empresas buscan ofrecer productos y servicios más avanzados, la necesidad de optimizar y mejorar el código existente se vuelve cada vez más importante. La refactorización abarca una amplia gama de actividades, desde mejorar la calidad del código hasta adaptarse a cambios en los requisitos y la arquitectura.

En el estudio "Industry Experiences with Large-Scale Refactoring"[4], los autores analizaron cómo los desarrolladores llevan a cabo refactorizaciones a gran escala y las herramientas que utilizan para apoyar este proceso. Los resultados mostraron que si bien las refactorizaciones a pequeña escala son más comunes, las refactorizaciones a gran escala no son infrecuentes en la industria del software.

Se encontró que estas refactorizaciones a gran escala están estrechamente relacionadas con objetivos comerciales y pueden tener un alcance mucho más amplio que las mejoras locales de código. Uno de los desarrolladores encuestados por la investigación entrega una respuesta que se alinea con esta idea: “... la refactorización a gran escala es diferente en el sentido de que hay un costo oportunidad por hacer o no hacer el trabajo. Se convierte en una decisión empresarial sobre dónde invertir.”

“Large-scale refactoring is a distinct activity for which significant resources need to be allocated, rather than being something that developers can easily weave into their day-to-day work (...) it includes activities like persuading stakeholders of benefits and managing expectations”

Por otro lado, el estudio profundiza sobre el uso y la adopción de herramientas automatizadas de refactorización. Este concluye que la principal razón por los bajos niveles de uso de estas es la falta de confianza de los desarrolladores en lo que estas herramientas pueden lograr, por esto, los desarrolladores a menudo optan por realizar la refactorización manualmente debido a esta falta de confianza, especialmente en las etapas o procesos de planificación de refactorización, como demuestra la Tabla 4 del estudio. Esto destaca la necesidad de mejorar la confianza y eficacia de las herramientas de refactorización automatizadas para promover su adopción en la industria.

Este estudio pone en evidencia la importancia de la refactorización de software en la industria y la necesidad de abordar los desafíos y limitaciones existentes en el proceso. La refactorización a gran escala puede ser una tarea compleja y costosa, pero es fundamental para garantizar que el software se mantenga ágil, escalable y de alta calidad. La automatización de herramientas de refactorización y el desarrollo de técnicas específicas para diferentes frameworks y lenguajes de programación pueden ser clave para facilitar y promover la refactorización efectiva en la industria.

Una refactorización efectiva puede mejorar significativamente la calidad del código y reducir la deuda técnica, lo que a su vez conduce a un desarrollo de software más eficiente y sostenible. Como explicitan los autores del estudio en la conclusión de este: "El costo anticipado de dicha refactorización, junto con las prioridades empresariales que favorecen las nuevas características, suelen hacer que las organizaciones renuncien a la refactorización a gran escala, lo que a su vez suele tener consecuencias problemáticas."

La refactorización no solo mejora la calidad del código, sino que también facilita la adaptación a cambios en los requisitos y la arquitectura del software. Como parte del proceso de refactorización, los desarrolladores pueden optimizar el diseño y la estructura del código para garantizar que sea más flexible y fácil de mantener. Al respecto, otro desarrollador en el estudio menciona: "Creo que este tipo de refactorizaciones es una tarea muy creativa, que a menudo requiere aportar ideas arquitectónicas totalmente nuevas, o incluso sugerir cambios reales en el producto para que la arquitectura sea más limpia."

Es por la importancia de las refactorizaciones que la automatización de herramientas de refactorización toman especial relevancia, ya que pueden tener un impacto significativo en la productividad y eficiencia de los equipos de desarrollo. Al

reducir la cantidad de trabajo manual necesario para refactorizar el código, las herramientas automatizadas pueden permitir que los desarrolladores se centren en tareas más creativas y estratégicas. Esto puede resultar en una mayor eficiencia y calidad del desarrollo de software en general.

La refactorización en la industria del software también puede ser especialmente relevante en el contexto de aplicaciones web y bases de datos. Con el aumento de la demanda de aplicaciones web y el crecimiento de bases de datos en el almacenamiento y procesamiento de datos, la optimización del rendimiento es esencial para brindar una experiencia de usuario fluida y eficiente. El estudio "An Industrial Experience Report on Performance-Aware Refactoring on a Database-centric Web Application"[5] trata sobre la refactorización consciente del rendimiento en aplicaciones web basadas en Laravel destaca cómo los anti-patrones de rendimiento en la capa ORM pueden afectar significativamente el rendimiento de las aplicaciones web. Algunos de estos anti-patrones son específicos del framework o lenguaje de programación, mientras que otros son más generales. Al identificar y abordar estos anti-patrones, los desarrolladores pueden lograr mejoras significativas en el tiempo de respuesta y la eficiencia de las aplicaciones, como lo demuestran los resultados del estudio, logrando reducir el tiempo de respuesta hasta en un 93.4%.

2.2. Lenguajes, Frameworks y Tecnologías

2.2.1. Javascript

JavaScript es un lenguaje de programación flexible y ampliamente utilizado en el desarrollo web. Fue creado inicialmente como un lenguaje de scripting para mejorar la interactividad de las páginas web, pero con el tiempo ha evolucionado para convertirse en un lenguaje de programación completo que puede utilizarse tanto en el lado del cliente como en el lado del servidor utilizando Node.js.

Una de las principales ventajas de JavaScript es su amplia adopción y soporte en todos los navegadores modernos. Esto significa que los desarrolladores pueden escribir código JavaScript y estar seguros de que funcionará en la mayoría de los dispositivos y plataformas. Además, JavaScript es un lenguaje interpretado, lo que significa que los cambios en el código se pueden ver de inmediato sin la necesidad de recompilar.

Otra ventaja de JavaScript es su simplicidad y facilidad de uso. Su sintaxis es similar a otros lenguajes de programación como C y Java, lo que facilita su aprendizaje para aquellos que ya están familiarizados con esos lenguajes. Además, JavaScript tiene

una gran cantidad de bibliotecas y frameworks disponibles, lo que permite a los desarrolladores crear aplicaciones de manera más rápida y eficiente.

JavaScript también es un lenguaje dinámico, lo que significa que no es necesario declarar tipos de variables antes de usarlas. Esto puede ser una ventaja para algunos desarrolladores, ya que permite una mayor agilidad en el desarrollo. Sin embargo, también puede llevar a errores y problemas de rendimiento si no se tiene cuidado con el manejo de tipos de datos.

Por otro lado, una de las principales desventajas de JavaScript es su falta de capacidad para manejar operaciones en segundo plano y la falta de soporte para programación concurrente. Esto puede llevar a problemas de rendimiento en aplicaciones complejas que requieren procesamiento intensivo.

Además, JavaScript puede ser propenso a errores debido a su debilidad tipográfica y su sintaxis flexible. Los errores pueden ser difíciles de depurar, especialmente en aplicaciones más grandes y complejas.

2.2.2. Node.js

Node.js no es ni un lenguaje de programación ni un framework, es solamente una forma de ejecutar código JavaScript fuera de un navegador, pudiendo usar de esta forma el lenguaje JavaScript en el servidor.

JavaScript, como se mencionó anteriormente, es un lenguaje single thread pero permite ejecución asíncrona ocupando de forma más eficiente los recursos y el tiempo de ejecución. Por esto, JavaScript y, por extensión, Node.js son muy rápidos y con un alto rendimiento donde necesita ejecución en tiempo real. Además, ya que la compilación de Node.js se realiza en tiempo de ejecución, esto genera mayores optimizaciones a las funciones que más veces sean llamadas.

Otro de los grandes beneficios de JavaScript es su gran flexibilidad y comodidad de usar el mismo lenguaje de programación para el front-end como el back-end.

2.2.3. MongoDB

MongoDB es una base de datos NoSQL que ha ganado popularidad en los últimos años debido a su enfoque en la escalabilidad, flexibilidad y rendimiento. Se trata de una base de datos orientada a documentos, a diferencia de filas y columnas como en las bases de datos relacionales tradicionales. Esto significa que almacena los datos en

documentos usando estructuras BSON[13] (Binary JSON) el cual es muy similar a un JSON [14], lo cual lo hace un complemento perfecto para JavaScript.

Como se describió anteriormente, los datos en este tipo de bases de datos se almacenan como documentos y estos son estructurados como un JSON, los documentos serían el equivalente a las filas en una base de datos relacional, y una agrupación de documentos es una colección y por lo tanto su equivalencia relacional es una tabla.

Una de las principales ventajas de MongoDB es su flexibilidad en el esquema de datos. A diferencia de las bases de datos relacionales que requieren una estructura de tabla fija, MongoDB permite que los documentos dentro de una colección tengan esquemas diferentes. Esto es especialmente útil en aplicaciones que evolucionan con el tiempo y requieren cambios frecuentes en la estructura de datos.

Otra ventaja de MongoDB es su capacidad para manejar grandes volúmenes de datos y cargas de trabajo intensivas. Su arquitectura distribuida permite que los datos sean distribuidos en varios servidores, lo que facilita la escalabilidad horizontal. Esto significa que se pueden agregar más servidores para manejar un mayor volumen de datos y usuarios sin afectar negativamente el rendimiento.

MongoDB también es rápido en operaciones de lectura y escritura debido a su diseño de almacenamiento basado en documentos, su capacidad para indexar datos de manera eficiente y debido a que está escrito en C++, lo que significa que las consultas, ediciones y otras modificaciones en la base de datos se producen de forma muy rápida.

Sin embargo, MongoDB también tiene algunas desventajas. Una de ellas es su consumo de memoria, ya que necesita mantener un índice en memoria para acelerar las consultas. Esto puede ser un problema en sistemas con recursos limitados o cuando se manejan grandes volúmenes de datos.

Otra desventaja es la falta de soporte transaccional completo. Aunque MongoDB admite operaciones atómicas en un solo documento, no tiene soporte para transacciones distribuidas que abarquen múltiples documentos o colecciones. Esto puede ser un problema en aplicaciones que requieren operaciones transaccionales complejas.

Además, MongoDB puede tener un rendimiento ligeramente inferior en comparación con algunas bases de datos relacionales en ciertas operaciones complejas que requieren unir datos de múltiples colecciones. En tales casos, las bases de datos relacionales con capacidades de optimización de consulta más maduras pueden ser más adecuadas.

2.2.4. Mocha.js y Chai.js

Mocha.js[11] es un framework de pruebas de JavaScript que se puede ejecutar en Node.js, las pruebas en Mocha se ejecutan secuencialmente y se entregan informes precisos.

Hay cuatro funciones básicas que se deben conocer para entender las pruebas en Mocha: `describe(), before(), after() e it():`

- `describe()`: agrupa un conjunto de pruebas.
- `before()`: las pruebas ejecutarán lo que se agregue en esta función antes de las pruebas, muy útil para inicializar variables que serán utilizadas por toda las pruebas, como una conexión a la base de datos.
- `after()`: análoga a la función anterior, esta función se ejecuta al finalizar todas las pruebas, necesaria para, por ejemplo, cerrar la conexión a base de datos.
- `it()`: se utiliza para definir una prueba individual, tomando solo dos parámetros: un texto que describe la prueba y una función que contiene la prueba misma.

Además, este framework permite realizar pruebas sobre código asíncrono sin problemas.

Mientras que Chai.js[12] es una librería de aserciones, por lo cual la hace una pareja perfecta para Mocha. Tiene múltiples funciones que permiten la comparación entre el resultado esperado y el resultado obtenido, añadiendo mucha flexibilidad al momento de escribir las pruebas.

2.2.5. Herramientas

Dado que esta memoria requirió mucha reestructuración y reescritura de código, se utilizaron herramientas que facilitan este proceso. Las más notables fueron: el IDE Jetbrains Webstorm, la extensión Sonarlint y la API de medición de rendimiento perf_hooks.js.

2.2.5.1. JetBrains Webstorm

JetBrains WebStorm es un IDE (Integrated Development Environment [7]) diseñado específicamente para el desarrollo de aplicaciones web y JavaScript. Incluye un conjunto completo de herramientas y características para facilitar la programación, depuración y pruebas de aplicaciones web modernas.

Una de las principales ventajas de WebStorm es su potente soporte para JavaScript y sus tecnologías relacionadas, como HTML, CSS, Node.js y frameworks

populares. El IDE tiene un editor de código que ayuda a los desarrolladores a escribir código más rápido y evitando errores, con funciones como autocompletado, resaltado de sintaxis y navegación rápida por el código. Estas funciones entregan varias comodidades al momento de refactorizar grandes proyectos, como el MTD.

El IDE tiene particularmente una función de ayuda para refactorizaciones que permite tener mayor seguridad al momento de hacer cambios y correcciones. Además, facilita la visualización de documentación de las librerías.

WebStorm también incluye un depurador integrado que permite a los desarrolladores inspeccionar el estado de sus aplicaciones en tiempo de ejecución y resolver problemas de manera eficiente. Además, ofrece integración con herramientas de prueba, lo que facilita la gestión y ejecución de pruebas unitarias.

2.2.5.2. SonarLint

SonarLint es una herramienta de análisis de código diseñada para mejorar la calidad y la seguridad del código en proyectos de desarrollo de software. Es una extensión que se integra con entornos de desarrollo, como Jetbrains Webstorm, para proporcionar alertas en tiempo real sobre posibles problemas en el código.

SonarLint destaca por su capacidad para identificar problemas en el código, como vulnerabilidades de seguridad, librerías deprecadas, errores de programación, malas prácticas y duplicación de código. Utiliza reglas definidas basadas en estándares de programación y buenas prácticas. Además, la integración en tiempo real permite a los desarrolladores recibir sugerencias y advertencias mientras escriben código, lo que ayuda a corregir problemas en las primeras etapas del desarrollo y a mantener un alto nivel de calidad en el código, pudiendo así solucionarlos incluso antes de subir el commit con el código.

2.2.5.3. perf_hooks.js

perf_hooks.js[15] es una API de medición de rendimiento para JavaScript. Es una herramienta esencial para evaluar y analizar el rendimiento de funciones, clases, módulos, etc. Fue diseñada para brindar una visión detallada del tiempo de ejecución y la eficiencia de código, esta API ofrece capacidades avanzadas de medición que son muy facilitan el trabajo de optimización en plataformas, tanto aplicaciones web como de Node.js.

Con esta herramienta, se pueden crear puntos de marca en su código para medir el tiempo transcurrido entre ellos. Esto permite identificar con precisión cuánto tiempo lleva ejecutar secciones específicas del código. Además, la API proporciona

información sobre la duración de eventos, lo que resulta valioso para entender el comportamiento temporal de las funciones y operaciones. Finalmente, como esta API permite realizar mediciones en microsegundos, esto lo que garantiza una gran precisión en las mediciones obtenidas.

Capítulo 3

Análisis y Diseño

En este capítulo se describe el flujo de la información en del MTD, desde los canales de venta hasta que la orden de compra llega a bodega para ser elaborada y despachada.

En la sección 3.1 se describe el flujo de información de las órdenes en el MTD. Luego, en la sección 3.2 se presenta el diseño de la estructura de la plataforma que ejecuta el flujo anterior. Finalmente, en la sección 3.3 se hace un análisis de la plataforma, tanto a nivel de código como de base de datos.

3.1. Flujo de Información en el MTD

Como se presentó en la Introducción, las órdenes de compra se obtienen desde los canales de venta utilizando las API dispuestas por ellos. Son estas órdenes de compra las que recorren el flujo completo del MTD para su procesamiento, lo cual incluye notificaciones al cliente del estado de la orden, creación de etiquetas, envío a facturación y, finalmente, envío al WMS y Courier.

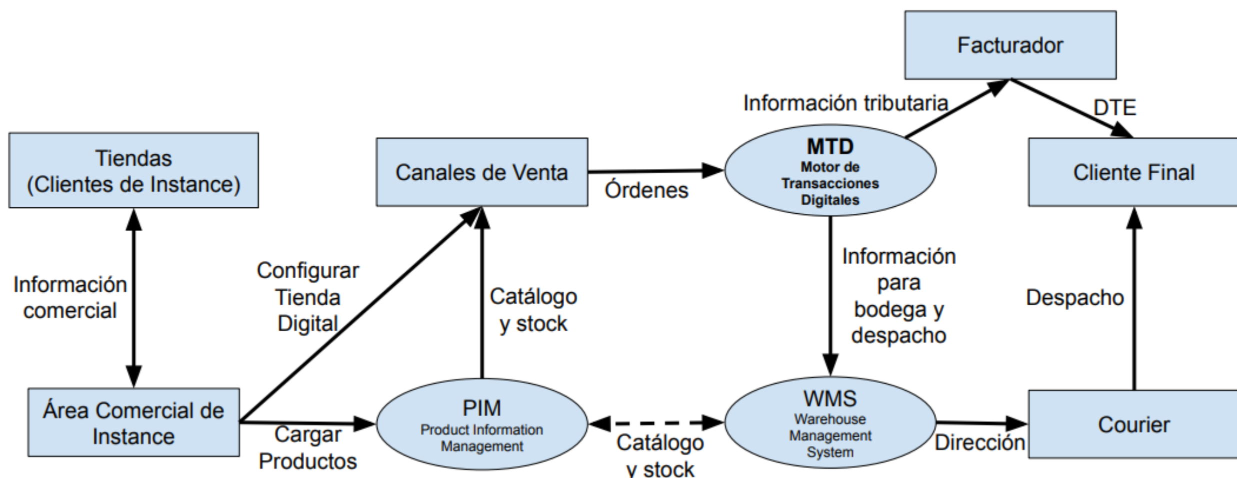

Figura 3.1: Flujo de Información en las plataformas de Instance y plataformas asociadas. Notar cómo se relacionan los diferentes procesos del flujo.

Como se puede observar en la Figura 3.1, el MTD opera integrándose a varias plataformas asociadas, las API de los canales de venta para obtener las órdenes de compra y a facturadores, WMS y Couriers para completar el procesamiento y despacho de la orden con sus respectivos productos. El facturador es una empresa externa que puede generar el Documento Tributario Electrónico (DTE) y retornarlo al MTD para su entrega al cliente final. Mientras que un WMS (Warehouse Management System) es una plataforma que facilita la administración del proceso de bodegaje. Mientras que el Courier es una empresa que realiza el despacho mismo del producto.

Por lo tanto, el proceso de una orden de compra en el MTD comienza con la integración al canal de venta, para luego generar el DTE mediante un facturador y finaliza cuando las órdenes llegan al WMS y se entrega el pedido al courier. Todo el flujo descrito es recorrido por un promedio de 20000 órdenes de compra por semana.

3.2. Diseño

El flujo descrito en la sección anterior se realiza mediante múltiples scripts de Node.js que se ejecutan sobre un servidor EC2 de AWS usando crons, teniendo más de 50 mil líneas de código. Estos scripts están encargados de conectarse a las API de todos los canales de venta para obtener todas las órdenes de los clientes de Instance. La información de cada una de estas órdenes es enviada a facturación, al WMS que hace la gestión de bodega y, finalmente, a nuestra base de datos. Como se puede observar en la Figura 3.2, el MTD es crítico y central en el control del flujo de información de una orden de compra.

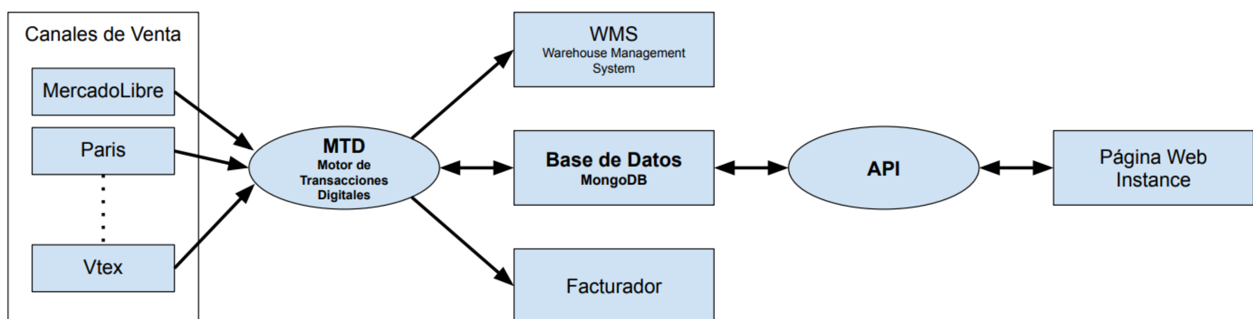

Figura 3.2: Flujo de Información de una orden de compra.

Este proceso realizado es específico para cada canal de venta debido a que hay que conectarse a las API de estos y luego transformar la información recibida para unificar y normalizar los datos para continuar con el flujo anteriormente descrito.

Al finalizar el proceso descrito anteriormente, la información es almacenada en una base de datos no relacional, específicamente usando MongoDB lo cual permite adaptarse a la información, ya que estas son más flexibles que las bases de datos relacionales y, además, son más rápidas y eficientes para responder consultas. La información en la base de datos es utilizada para coordinar todo el proceso y mantener actualizado el estado de cada orden.

Es la API de Instance la que hace uso de la información en la base de datos para realizar la lógica de negocio para la página web y también proveer una conexión con los datos Instance para empresas externas.

Finalmente, la página web de Instance construida usando React JS se usa para presentar a los clientes y el área comercial de Instance con reportes e informes de sus ventas en los diferentes canales de venta.

3.3. Análisis

La plataforma está creada usando JavaScript sobre Node.js en un servidor EC2 de AWS ejecutándose mediante crons, estos crons están encargados de ejecutar los scripts que recuperan la información desde las API de los canales de venta, procesan los datos y envían la información al facturador y el WMS. Este método para ejecutar los scripts, los cuales son una parte crítica del negocio, posee varias desventajas, ya que se requiere estar constantemente conectándose al servidor a revisar que los procesos estén activos correctamente, además no posee ningún método de CI (Continuous Integration) lo cual significa que manualmente se debe hacer el git pull hacia el código en producción y volver a subir el cron de todos los archivos modificados, estos pasos se deben hacer cada vez que se desea pasar una nueva funcionalidad, integración o corrección a producción, y ya que esto es manual, donde hay manualidades pueden surgir errores evitables con una automatización bien realizada.

La deuda técnica descrita en secciones anteriores se ve reflejada en llamadas ineficientes a código asíncrono y en conexiones abiertas innecesariamente a la base de datos. Estos dos problemas pueden ser solucionados con una revisión y refactorización profunda del código.

Una refactorización efectiva también sería capaz de parametrizar la información de los clientes en vez de tenerlos “en duro” produciendo una solución general para cada integración en vez de una particular por cada cliente. Esto implicaría que la estructura de la base de datos para estos parámetros sea definida y mantenida para nuevos clientes, para lo cual es de suma importancia definir reglas y buenas prácticas para el equipo de desarrollo con la base de datos.

Pero no solamente se deben definir estas prácticas con la base de datos, sino que también con el manejo del código. Actualmente, el código no posee ninguna regla definida para el nombramiento de variables, clases o funciones, tampoco hace uso de comentarios para describir el flujo de una función, ni descripciones de estas. Se deben definir estas reglas e implementar y fomentar las mejores prácticas de JavaScript ES6+ que entrega múltiples beneficios del tipo azúcar sintáctico que favorecen y facilitan la legibilidad del código, pero también beneficios que directamente afectan el rendimiento y estabilidad de la plataforma. Para esto, se usará el apoyo de SonarLint, la cual es una extensión del IDE de código abierto que facilita la detección de errores de este tipo en el código.

La estructura de la plataforma no permite integrar pruebas unitarias debido a que el código actualmente produce las llamadas a una función y es esta misma función la cual en vez de retornar un valor, llama a la siguiente función en el flujo. Entonces, se debe modificar el paradigma actual de la plataforma, refactorizando de una forma que admita pruebas unitarias.

3.3.1. Código

Una refactorización del MTD apunta a hacer que esta sea una plataforma estable, escalable y eficiente. Para hacerlo se deben solucionar los problemas técnicos que posee la plataforma actualmente.

Realizando un estudio del MTD se encontró que se pueden hacer optimizaciones y mejoras en código, especialmente en la ejecución de código asíncrono en JavaScript. Corregir esto aumentará considerablemente la eficiencia y velocidad de respuesta de la plataforma. Además, se encontraron secciones de código que tienen configuraciones que deberían ser definidas en la base de datos para tener mejor control y escalabilidad de la plataforma.

El código actualmente no está diseñado para permitir la integración de pruebas unitarias debido a su estructura. Hoy en día, las funciones de los scripts se llaman en un flujo para realizar las tareas necesarias, pero nunca retornan. Entonces, se debe hacer una refactorización profunda para permitir las pruebas unitarias.

Por último, el código del MTD actualmente tiene varios problemas ajenos a la estabilidad y funcionalidad del código, ya que no hay documentación alguna y hay muchas malas prácticas, por ejemplo múltiples variables en una función llamadas aux, aux1 o aux2 y no hay comentarios que ayuden a seguir el flujo de funciones de más de 300 líneas de código. Además, el código no está utilizando las reglas y beneficios añadidos en JavaScript ES6+, tales como const, let o for .. of .., lo cual facilita la lectura

del código y no es simplemente azúcar sintáctico, sino que realmente tiene efectos en el rendimiento de la plataforma.

3.3.2. Base de Datos

El principal problema que tiene la base de datos es la estructura para almacenar las configuraciones de las diferentes tiendas en los canales de venta. Ya que toda esta información está actualmente almacenada en un único arreglo que contiene objetos JSON. Esto dificulta de gran manera el acceso a estas configuraciones y, por lo tanto, la mantención de estas.

Por otro lado, la plataforma tiene una cantidad muy alta de conexiones a la base de datos, es decir, hay aproximadamente 2000 conexiones por hora, cuando debido al tamaño de la operación no debería haber más de 600. Es por eso que se utiliza un servidor elástico, que dependiendo de la cantidad de conexiones, habilita más o menos espacio y poder de cómputo en el servidor. Este tipo de servidores de igual forma tienen un tope máximo de 3000 conexiones concurrentes y al superar este número el servidor se cae, esto pasa aproximadamente una vez al mes. Esto se debe a que en el código no se cierran muchas conexiones creadas y, por lo tanto, se terminan acumulando.

Capítulo 4

Implementación

En esta sección se describe la implementación de la refactorización del código del Motor de Transacciones Digitales de Instance y la integración de pruebas unitarias a este.

En la sección 4.1 se presenta la investigación sobre mejores alternativas de ejecución a los crons. Luego, en la sección 4.2, se presentan los cambios en en la base de datos, específicamente la reestructuración de esta para una mayor escalabilidad de la plataforma y la disminución de las conexiones concurrentes a esta. Después, en la sección 4.3, se presenta la refactorización al código mejorando la eficiencia, mantenibilidad y escalabilidad de este. Además, se prepara el código para la integración de las pruebas unitarias. Para luego, en la sección 4.4, se presentan las pruebas unitarias integradas al MTD. Finalmente, en la sección 4.5, se presentan las situaciones y problemas encontrados al momento de pasar a producción las modificaciones realizadas.

4.1. Alternativa de Ejecución

Como se presentó anteriormente, el MTD utiliza crons ejecutándose sobre una máquina EC2 de AWS para realizar todo el proceso descrito en la sección 3.2. Este método de ejecución es bastante incómodo y con muchas manualidades, las cuales pueden introducir errores y problemas en una parte crítica del negocio. Es por esto que se decidió investigar nuevas y mejores opciones para ejecutar estos scripts, ya que AWS y otras plataformas ofrecen varias alternativas que se pueden adaptar a las necesidades del MTD.

En este caso no se implementó la solución, sino que solamente se realizó la investigación y se presentaron los resultados al líder del equipo de desarrollo porque la decisión de implementar alguna de estas soluciones es determinada por él puesto que se deben tomar en cuenta los costos en servidores y tecnologías de estas soluciones.

4.1.1. Github Actions

Github Actions [6] es una plataforma de integración y despliegue continuo (CI/CD por sus siglas en inglés), que permite automatizar las pruebas y el despliegue en el servidor. En el caso del MTD, esta alternativa implica crear flujos de trabajo (workflows) que se ejecutarán al momento de que realice una acción sobre Github, como realizar un commit, push o crear un pull request.

Hay varios flujos de trabajo que serían un aporte al MTD, automatizando tareas y, también, comprobando que se cumplan buenas prácticas. Algunos de los flujos de trabajo que benefician de mayor manera al MTD serían:

- Lint Check:

  Este flujo de trabajo previene que código con errores sea publicado en la rama de producción, agregando de esta forma otra capa de seguridad y estabilidad a la plataforma. Además, permite establecer buenas prácticas en el código para el equipo de desarrollo.

- Commit Check:

  Este flujo de trabajo asegura que los commits contengan un formato definido, en el caso del MTD este está definido por Conventional Commits [8].

- CD:

  El Despliegue Continuo, por sus siglas en inglés (CD), sería lo que más beneficiaría al MTD, ya que esto eliminaría gran parte de las manualidades en el paso a producción. Esto debido a que se automatiza el proceso de pull en el servidor actualizando automáticamente el cron.

Esta opción es la más simple de implementar, porque solamente requiere crear nuevos flujos de trabajo e integrarlos al proyecto en GitHub. Además, no introduce ningún tipo de costo extra para la empresa, porque se usaría el mismo servidor actual.

Pero esta alternativa tiene la desventaja de que, si bien se automatiza la actualización del cron, en caso de necesitar crear un nuevo cron de igual forma hay que hacerlo manualmente conectándose al servidor usando SSH.

4.1.2. AWS: ECS Schedule Tasks

AWS provee una herramienta llamada ECS Schedule Tasks, con la cual se puede usar para configurar la ejecución de scripts de Node.js que se ejecuten en ciertos intervalos de tiempo, lo cual significa que es una alternativa viable para efectivamente

reemplazar el método de ejecución actual. Además, esta opción no tiene límite de tiempo de ejecución, a diferencia de la siguiente alternativa.

Pero esta alternativa requiere que el código este dockerizado, lo cual agrega un proceso en el cual nadie del equipo de desarrollo tiene experiencia, lo cual aumenta las posibilidades de que se produzcan errores.

4.1.3. AWS: CloudWatch Events y Lambda

Esta alternativa es similar a la anterior, ya que se utilizan servicios provistos por AWS, pero esta vez en vez de tener que dockerizar el código, este hay que hacer un paquete Lambda. Una vez creado el paquete Lambda este se ejecuta usando AWS CloudWatch. Además, nos provee herramientas de control de ejecución, facilidad de programar intervalos de tiempo de ejecución y se muestra el uso de recursos para la ejecución.

Esta opción tiene la gran ventaja de que no hay necesidad de dockerizar cada uno de los crons y, a diferencia de la alternativa anterior, sí hay experiencia en el equipo de desarrollo en el uso y manejo de Lambdas de AWS.

Pero también tiene una desventaja, tiene un límite de tiempo de 15 minutos, es decir, la ejecución del script de Node.js debe demorar menos de 15 minutos en realizar toda su ejecución. Y si bien esto debería ser tiempo suficiente, ya que se depende de la velocidad de respuesta de las plataformas de canales de venta, facturadores y WMS, no se puede asegurar que se cumpla siempre.

Es esta última alternativa la cual es la más favorable para implementar en el MTD, ya que el límite de tiempo no es un problema luego de la refactorización, dado que al utilizar ejecución asíncrona se reduce mucho el tiempo de ejecución debido a la paralelización de la espera de la respuesta del servidor de las plataformas asociadas.

Además, es compatible con la primera alternativa, lo que significa que se pueden crear flujos de trabajo para comprobar buenas prácticas (Lint check y Commit check) del equipo de desarrollo sin afectar la integración con CloudWatch Events y Lambda.

4.2. Base de Datos

Como se describe en la sección de Problema, tanto la base de datos como el modelo de datos tenía múltiples problemas, de los cuales los más importantes eran la estructura de la base de datos que limitaba la escalabilidad de la plataforma y el gran número de conexiones a la base de datos.

4.2.1. Reestructuración de la base de datos

Si bien la flexibilidad de MongoDB es una ventaja, es debido a esto que no hubo una buena planificación inicial sobre la escalabilidad y mantenibilidad de la base de datos. Por lo tanto, ahora se tuvo que volver a estudiar y reestructurar la base de datos, para que esta pueda admitir más datos y mantener control sobre estos datos.

El principal problema se encontraba en la configuración de las tiendas para cada canal de venta, ya que toda esta información estaba almacenada en un solo arreglo de objetos JSON. Esto perjudicaba enormemente el acceso a la configuración que se necesitase y disminuye fuertemente los el rendimiento de la base de datos porque no se está buscando entre documentos, que es lo que se espera, sino que siempre va a buscar el mismo documento y es en el código donde se debe hacer el filtro correspondiente para encontrar la información necesitada.

Por lo tanto, el modelo de datos anterior no tenía forma de almacenar más configuraciones de tiendas en canales de venta sin empeorar cada vez más el rendimiento de la plataforma y la mantenibilidad de la base de datos para el equipo de desarrollo.

Entonces se creó una colección de datos para cada canal de venta, facturador, WMS y Courier a los cuales estuviéramos integrados. Dentro de estas colecciones, cada uno de los documentos representa la configuración de esa tienda en la plataforma asociada. Ahora entonces, para obtener la configuración de una tienda para cierto canal de venta, simplemente se debe buscar el documento deseado en la colección del canal de venta. Como se puede observar en la Figura 4.1, ejemplificando para el caso de los canales de venta, se guardaron cada una de las configuraciones de todas las tiendas asociadas al canal en un documento.

Además de solucionar los problemas de las configuraciones de las tiendas en los diferentes canales de venta, también se decidió abordar, mirando hacia el futuro, hacer lo mismo para las configuraciones de las tiendas en los diferentes facturadores, WMS y Couriers. De esta forma, implementando una solución global y mejorando la escalabilidad y la mantenibilidad de la base de datos.

Estos cambios, enfocados en las configuraciones de tiendas en cada una de las diferentes plataformas a las cuales el MTD está integrado, serían de vital importancia para la parametrización del código que se verá en la próxima sección.

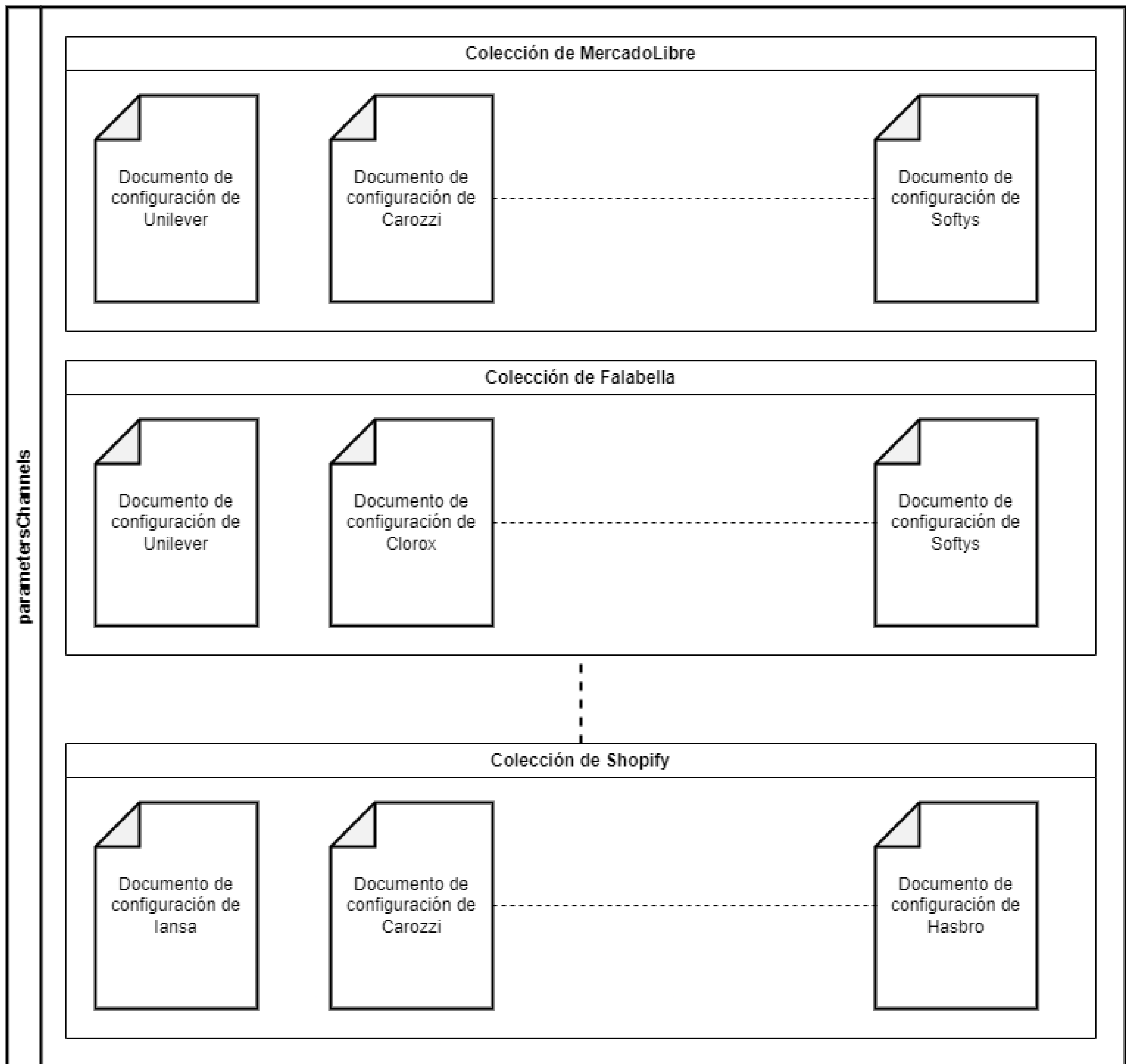

Figura 4.1: Nueva estructura para almacenar la configuraciones de los clientes en los canales de venta.

4.2.2. Conexiones a la base de datos

Como se menciona en la sección de Problema, la base de datos tiene una gran cantidad de conexiones concurrentes. El mínimo número de conexiones concurrentes posible es una conexión por cron en ejecución, utilizando esta misma conexión para toda la ejecución del cron, y que cada módulo ejecutado por este cron use esa misma conexión.

Por lo tanto, luego de refactorizar el código, ahora se abre solamente una conexión a la base de datos por cron en ejecución y cuando termina la ejecución esta

conexión se cierra. Pero se tuvo que tener particular cuidado con que todas las promesas asíncronas hayan retornado y que no quede una promesa pendiente en ejecución que requiera esa conexión, ya que en este caso se produciría un error bastante complejo de manejar.

4.3. Código

Como se mencionó anteriormente, el código contenía una gran deuda técnica, la cual se buscó solucionar aplicando buenas prácticas a todo el equipo de desarrollo y refactorizando el código para que este tenga una mayor eficiencia, mantenibilidad y escalabilidad. Esto se logró manteniendo su funcionamiento (en algunos casos se modificó de gran manera, como se verá en la sección 4.3.5), pero mejorando el rendimiento usando ejecución asíncrona eficientemente, parametrizando las configuraciones correspondientes y optimizando las conexiones activas a la base de datos.

4.3.1. Documentación y buenas prácticas

A diferencia del resto de los cambios realizados en esta sección, estas mejores prácticas fueron implementadas a nivel del equipo de desarrollo y aplicadas durante todo el proceso de refactorización.

Se definieron reglas y mejores prácticas a seguir, de esta forma se mejora la legibilidad del código y también el mantenimiento del repositorio en Github. Por políticas de la empresa, todos estos nombres serán en inglés. Estas prácticas incluyen:

- Nombramiento de variables, funciones y clases:
  - Variables[cualquier tipo]: camelCase.
  - Variables[Boolean]: camelCase con `<is/are/has>` como prefijo.
  - Funciones: camelCase con un verbo como prefijo.
  - Clases: PascalCase
  - Métodos: camelCase con un verbo como prefijo.
  - Métodos privados: camelCase con guión bajo como prefijo.
  - Constantes: UPPER_SNAKE_CASE.
- Descripción de funciones, métodos y clases antes de su definición.
- Definición de reglas para commits y ramas en git
  - Commits: ConventionalCommits[8]
  - Ramas:`<tipo de desarrollo>/<descripción en kebab-case>`

Además de fomentar el uso de buenas prácticas, se propuso el uso de dos nuevas acciones para github Actions: una que verifique la calidad del código(Lint check) y otra que verifique que los commits hayan sido escritos correctamente(Commit check) como se mencionó en la sección 4.1.

4.3.2. Reestructuración del proyecto

El proyecto estaba compuesto por múltiples módulos, sin orden en particular, que eran ejecutados por los crons. Pero esto significaba que estos módulos luego de varias modificaciones contenían varios miles de líneas de código, haciendo muy engorrosa la mantención de estos. Por lo tanto, se decidió modificar la estructura

Se modificó esta estructura para dar mayor escalabilidad y separación a los módulos de ejecución (los crons) que estaban empezando a aumentar en tamaño y cantidad de funciones sin control alguno.

La estructura que se decidió usar fue una arquitectura por capas, la cual separa el código con un enfoque técnico y tiene una clara separación de responsabilidades para cada una de las capas.

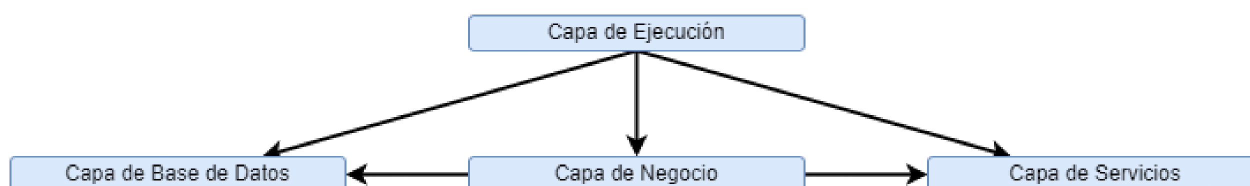

Figura 4.2: Diagrama de la arquitectura por capas implementada para el MTD

Como muestra la Figura 4.2, se pueden notar las relaciones entre las capas, siendo la capa de ejecución la que comienza la ejecución y hace llamadas a las otras capas y se abstrae de la lógica. Mientras que la capa de negocio contiene la lógica y los cálculos y, al igual forma que la capa de ejecución, se abstrae de las consultas, ya sea a base de datos como a API externas. Finalmente, son las capas de bases de datos y de servicios se encargan de realizar las consultas y conexiones a bases de datos y a las API de las diferentes plataformas asociadas, respectivamente.

La tradicional capa de presentación, la cual es el punto de entrada o de ejecución de la plataforma, fue reemplazada por una capa de ejecución compuesta por los propios módulos que se ejecutan con crons. Esto se debe a que la plataforma no tiene ninguna necesidad de renderización o de presentación en html. Si bien esta capa inicia la ejecución, esta capa no contiene la lógica de negocios y hace llamadas a las diferentes capas para completar con su ejecución.

La capa de negocio (BLL por sus siglas en inglés, Business Logic Layer) contiene todas las funciones y fórmulas particulares a un canal de venta, facturador, WMS o Courier.

La capa de base de datos (DAL por sus siglas en inglés, Data Access Layer) contiene las funciones específicas de un canal de venta, facturador, WMS o Courier para obtener y guardar la información en la base de datos.

La capa de servicios (SAL por sus siglas en inglés, Service Access Layer) contiene todas las conexiones a las diferentes API para obtener las órdenes o actualizarlas, de los canales de venta, facturadores, WMS y Couriers.

Nuestra estructura basada en arquitectura por capas, no fue una aplicación estricta de la arquitectura, sino que fue más bien inspirada por esta arquitectura pero con las modificaciones para adaptarla a las necesidades del proyecto.

4.3.3. Parametrización del código

Como se mencionó en la sección 4.2.1 la reestructuración de la base de datos permite que se parametrice la ejecución del código dependiendo de la configuración de la tienda en el canal de venta, facturador, WMS o Courier.

Para cumplir con esto, cada una de las conexiones a las múltiples API que la plataforma requiere conectarse para completar el flujo de una orden fue parametrizada en la base de datos para su control y configuración correspondiente. Además, se reemplazó todo el código en duro que existía en la plataforma por la configuración en base de datos de las tiendas. Este proceso fue lento y delicado para no cometer ningún error y parametrizar todas las variables en la ejecución del cron.

Esta modificación aumenta considerablemente la escalabilidad de la plataforma, ya que no se debe modificar el código para ingresar nuevas integraciones y de esta forma, disminuyen los posibles errores de estas manualidades.

También, esto nos permite integrar nuevas tiendas al MTD sin ninguna intervención del equipo de desarrollo, solamente requiriendo que el equipo comercial, o hasta el cliente mismo, defina las reglas de ejecución de la tienda en cada una de las plataformas deseadas.

4.3.4. Ejecución asíncrona

Se refactorizó el código con un enfoque que hiciera uso de la ejecución asíncrona de JavaScript ES6+, pero manteniendo el comportamiento y flujo de las órdenes de compra. Con los cambios se logró reducir en más de un 90% el tiempo de

ejecución, como veremos en el capítulo de Validación, lo que permite una integración con AWS CloudWatch y Lambda para una mejor alternativa de ejecución.

Esto se logró modificando la ejecución para que sea parcialmente asíncrona, es decir, será asíncrona pero por partes. A continuación se explorará por qué una solución completamente asíncrona no es correcta en algunos casos y, por lo tanto, se decidió no usarla.

La ejecución asíncrona se basa en dos partes, la creación de la promesa y en la espera de que la promesa retorne, esto es, por lo menos, para la implementación de la plataforma.

4.3.4.1. Definición de las promesas

Las promesas son objetos que representan la terminación o el fracaso de una operación asíncrona. Las promesas en JavaScript son lo que retornan las funciones que contienen el prefijo `async` en su firma. Esto fue agregado a JavaScript recientemente intentando hacer que la escritura de código asíncrono sea lo más similar posible al código síncrono.

Para la creación de promesas, solamente se requiere llamar a una función que sea asíncrona y no esperar el retorno con la palabra clave `await`. Solamente funciones asíncronas pueden contener la palabra clave `await` debido a que esta pausa la ejecución esperando a que esta promesa retorne. Otras formas de esperar, o mejor dicho en este caso, continuar la ejecución es mediante el método `then()`, este método ejecuta la función que se le pase como parámetro al momento en que la promesa retorne.

Para definir una función asíncrona la cual retorna una promesa, tal como es el caso de las funciones getOrders y processOrder de que se verán en la próxima sección, se debe definir esta de forma correcta, ya que no se definen de igual manera que funciones asíncronas.

4.3.4.1.1. Antipatrón de promesas

Uno de los errores que se pueden cometer al definir las funciones asíncronas, es el antipatrón de promesas [9]. Este antipatrón es una solución común para el uso y definición de promesas, pero es realmente ineficaz y puede causar errores muy difíciles de manejar y diagnosticar.

Este antipatrón es muy común cuando no se conoce correctamente la ejecución asíncrona y el uso de las promesas, esto lleva al uso del método `defer()` y del método `then()`, de forma ineficiente e innecesaria.

4.3.4.2. Solución totalmente asíncrona

La ejecución comienza con la creación de las promesas, una por cada tienda del canal de venta en ejecución, para conseguir las órdenes de compra. Con esto se logra que conseguir las órdenes de compra de todas las tiendas sea realizado de forma asíncrona.

Son estas mismas promesas las cuales en su ejecución crearán promesas de procesamiento de cada una de las órdenes obtenidas, las cuales se mezclaran con las promesas de las tiendas para obtener órdenes de compra, produciendo una ejecución completamente asíncrona.

Una vez que todas las promesas de obtención de órdenes hayan sido retornadas, se espera a que todas las promesas cargadas en los arreglos de promesas retornen.

```
1  // Este arreglo se debe crear afuera de la función main
2  // para que pueda ser accedido por la función getOrders,
3  // la cual cargará este arreglo de promesas de la función
4  // processOrder.
5  const promesasProcessOrders = [];
6  async function main() {
7      // Se abre una conexión a la base de datos,
8      // se usara esta conexión para toda la ejecución del modulo.
9      const db = MongoClient.connect(url);
10     const clientesCanalDeVenta = await getClientes(db);
11
12     // Ejecución totalmente asíncrona
13     const promesasGetOrders = [];
14     for (const cliente of clientesCanalDeVenta) {
15         // Se cargan las promesas de getOrders de cada cliente
16         // en el arreglo promesasGetOrders.
17         promesasGetOrders.push(sal_canal.getOrders(cliente, db));
18     }
19
20     // Se espera a que todas las promesas retornen, pero
21     // tomando el estado de los arreglos en el momento en que
22     // se ejecuta la línea 23
23     Promise.allSettled([...promesasGetOrders, ...promesasProcessOrders])
24     .then(async () => {
25         // Luego de finalizadas todas las promesas,
26         // se cierra la conexión a la base de datos
27         await db.close();
28     });
29 }
30
```

Código 4.1: Ejecución totalmente asíncrona

Esta solución, como pueden ver en el Código 4.1, carga de forma totalmente asíncrona tanto las promesas de getOrders como las promesas del processOrders y espera a que ambos arreglos de promesas retornen.

Pero como se menciona en un comentario en el código, las promesas que serán esperadas son las que están cargadas en el momento que se ejecuta la línea 23, pero ya que algunos getOrders siguen ejecutándose, no se esperaran todas las promesas de promesasProcessOrders que no hayan sido cargadas. Esto provoca que muchas órdenes, y prácticamente imposible saber cuáles, no sean procesadas.

4.3.4.3. Solución parcialmente asíncrona

Esta solución está compuesta por dos partes, cada una de las cuales es asíncrona, pero no entre ellas. Lo que significa que es parcialmente asíncrona.

La primera parte de la ejecución, la creación de las promesas, es la creación de una promesa por cada tienda del canal de venta en ejecución para la obtención de sus órdenes. Es decir, se buscan las órdenes de cada canal de venta de forma asíncrona, así aprovechando de continuar la ejecución mientras se espera que retorne la API del canal de venta.

Luego se debe esperar a que se retornen todas las órdenes concluyendo las promesas que se crearon en la parte anterior. Cada una de estas órdenes ahora deben procesarse para enviarse a facturadores, WMS y Couriers.

Ahora, en la segunda parte de la ejecución, se vuelven a crear promesas, pero esta vez para el procesamiento de cada una de las órdenes y sus envíos a plataformas asociadas respectivamente. Y finalmente se espera que estas promesas retornen, lo que implica que todas las órdenes fueron procesadas correctamente y se puede finalizar la ejecución cerrando las conexiones correspondientes.

```
async function main() {
    // Se abre una conexión a la base de datos,
    // se usara esta conexión para toda la ejecución del modulo.
    const db = MongoClient.connect(url);
    const clientesCanalDeVenta = await getClientes(db);

    // Ejecución parcialmente asíncrona
    const promesasGetOrders = [];
    for (const cliente of clientesCanalDeVenta) {
        // Se cargan las promesas de getOrders de cada cliente
        // en el arreglo promesasGetOrders.
        promesasGetOrders.push(sal_canal.getOrders(cliente, db));
    }

    // Se frena la ejecución para esperar las promesas creadas
    const promesasProcessOrders = [];
    const ordenesClientes = await Promise.allSettled(promesasGetOrders);
    for (const ordenesCliente of ordenesClientes) {
        for (const orden of ordenesCliente) {
            // Se cargan las promesas de processOrder de cada cliente
            // en el arreglo promesasProcessOrders.
            promesasProcessOrders.push(processOrder(order, db));
        }
    }

    // Se espera a que todas las promesas retornen
    return await Promise.allSettled(promesasProcessOrders)
    .then(async () => {
        // Luego de finalizadas todas las promesas,
        // se cierra la conexión a la base de datos
        await db.close();
    });
}

```

Código 4.2: Ejecución parcialmente asíncrona

Esta solución, a diferencia de la solución anterior y como pueden ver en el Código 4.2, carga todas las promesas de getOrders y espera a que estas retornen, para luego cargar las promesas de processOrder y espera a que estas retornen.

Como se mencionó anteriormente y pueden notar en la descripción, la ejecución es parcialmente asíncrona, ya que se crean dos grupos de promesas y se esperan secuencialmente a que terminen. Pero esta solución no tiene el problema descrito en la parte anterior, por lo tanto, esta solución en todos los casos procesa todas las órdenes que le corresponden. Además, el rendimiento de esta solución es marginalmente peor que la solución anterior. Por último, esta solución es bastante más fácil de interpretar y leer para otros desarrolladores, ya que su formato es más similar a funciones síncronas.

4.3.5. Refactorización del flujo de órdenes

La modificación presentada a continuación altera de forma profunda el funcionamiento y flujo de información en la plataforma. El cambio fue agregado al trabajo de título al final de este e implementa todos los cambios mencionados anteriormente.

La estructura de la plataforma está pensada en una ejecución totalmente independiente por canal de venta, siendo esta la que integra cada uno de los facturadores, WMSs y Couriers. Esta estructura es bastante ineficiente e inescalable, ya que cualquier plataforma que se desee integrar debe ser realizada en cada uno de los módulos de los canales de venta. Y a medida que la cantidad de canales de venta y plataformas aumentan, más compleja y difícil se vuelve la mantención de la plataforma.

Como se puede ver en la Figura A.1 presente en el Anexo, hay mucha duplicación de código y mucha parte del flujo puede ser unificada, simplificando la plataforma y manteniendo todas las funcionalidades actuales.

Por lo tanto, se propuso e implementó una refactorización profunda al funcionamiento del flujo de órdenes del MTD.

Esta modificación está compuesta de dos partes, primero, la definición e implementación de un modelo de orden unificada que contenga todos los campos requeridos y relevantes tanto para la operación, para el área de datos y para el área comercial. Segundo en la refactorización del código de los canales de venta para la obtención de órdenes y su unificación en un puro módulo de ejecución que contenga todas las integraciones y funcionalidades.

Como se puede ver en la Figura A.2 del Anexo, la modificación implementada utiliza los módulos de los canales de venta para obtener las órdenes y pasarlas al modelo unificado. Para que luego el módulo de mtdOrders obtenga cada una de estas órdenes y envíe la información a las plataformas requeridas. Pero ahora solamente está una vez la integración a estas plataformas asociadas, simplificado la ejecución y aumentando considerablemente la escalabilidad de la plataforma.

También, esta unificación provee algo muy valioso para la empresa, se facilita de sobremanera el estudio y análisis de los datos disponibles, lo cual le entregará información muy importante al área comercial y financiera de la empresa. Y se podrán tomar decisiones con más información y, por lo tanto, mejores.

4.3.5.1. Modelo de orden unificada

El modelo de datos de las órdenes unificadas intenta rescatar la mayor cantidad de datos de los canales de venta, pero que también ese campo esté en una

cantidad importante de canales de venta, si no este campo será inútil, tanto para la operación como para el análisis de estos por el área de datos y área comercial.

Los campos de una orden unificada se pueden dividir en siete tipos:

- campos de identificación de la orden, canal de venta y tienda,
- campos de facturación,
- campos de despacho,
- campos del comprador,
- campos de productos de la orden,
- campos de estados y de seguimiento de estados y
- campos de ejecución

Específicamente, los campos de facturación y de productos de la orden contienen la mayor parte de la información a enviar al facturador. Mientras que los campos del comprador, de despacho y de productos de la orden contienen los datos para enviar al WMS y Courier. Finalmente, los campos de estados de la orden y los campos de ejecución determinan si se deben o no enviar al facturador, WMS o Courier y a cuáles de estos.

Con los campos de ejecución se logra mantener la funcionalidad y flexibilidad para definir la lógica de cómo y a qué plataforma enviar las órdenes a nivel de cada canal de venta, pero manteniendo un nivel de abstracción por sobre estos, ya que la ejecución del módulo de mtdOrders es completamente agnóstico con respecto al canal de venta del cual que provenga la orden.

4.3.5.2. Estructura del nuevo flujo de órdenes

La refactorización por parte de los canales de venta los redujo a la parte fundamental de estos, es decir, las conexiones a los diferentes canales de venta y la interpretación de los campos de los canales de venta hacia el modelo unificado. También, se quitó la ejecución de las integraciones a diferentes plataformas, pero se mantuvieron las condiciones para su ejecución para que estas sean pasadas ya procesadas al módulo mtdOrders, mediante los campos de ejecución descritos en la parte anterior.

Además, se maneja la carga de las órdenes a ambas bases de datos, la del canal de venta y la de las órdenes unificadas, esto para mantener un respaldo y acceso a la información completa enviada por el canal de venta en caso de ser necesario.

Mientras que en el módulo mtdOrders procesa todas las órdenes que deban ser enviadas a alguna plataforma asociada, ya sea facturador, WMS o Courier. Este contiene todas las integraciones y tiene definido todos los campos a los que se deben

enviar. Además, contiene la lógica de notificación a los clientes sobre cambios de estado de sus órdenes.

Como se puede notar, el módulo mtdOrders es bastante simple, ya que tiene una responsabilidad única [10] y esta está bien definida. A diferencia de los módulos de los canales de venta antes de la reestructuración, ya que estos incluían muchas tareas en un mismo módulo, lo cual producía que estos módulos sean muy difíciles de interpretar y de modificar.

Con esta reestructuración, la plataforma es mucho más escalable y mantenible, por lo que es más fácil para el equipo de desarrollo integrar nuevos facturadores, WMS o Couriers y, si hay algún error, es más simple determinarlo y corregirlo.

4.3.6. Preparación para la integración de pruebas unitarias

Las funciones del código previo a la refactorización no retorna la mayoría de las veces, sino que, llaman a la siguiente función en el flujo, sin ninguna función que controle este flujo realmente. Para solucionar este problema, se modificaron todas las funciones para que, además de la ejecución asíncrona, retornen el resultado en vez de llamar a la siguiente función, de esta forma es la tarea del test el imitar el flujo de la orden llamando a las funciones correspondientes, pero podrá comparar los resultados obtenidos de la función y asegurar que esta está funcionando correctamente.

Estos tests son independientes de la ejecución asíncrona y permiten asegurar que la ejecución de las funciones de un módulo entero no tienen errores esperables. Ya que, como veremos en la siguiente sección, los errores se pueden producir por múltiples razones cuando la plataforma depende de tantas integraciones.

4.4. Pruebas Unitarias

Como se describe anteriormente, para escribir las pruebas unitarias se debió modificar el flujo de las órdenes de compra para imitar el flujo de un canal de venta y asegurar que los resultados de las funciones son los correctos.

Las pruebas mismas se escribieron usando Mocha.js que cubren todo el flujo de una orden de compra, desde el canal de venta hasta el facturador y el WMS. Además, se usó la librería Chai.js, la cual provee varios métodos de comparación de resultados (assert). Estas pruebas unitarias protegen la plataforma de resultados incorrectos al

modificar una función y de esta forma podemos reducir el riesgo de errores y alertas a Slack.

Aunque si bien podemos reducir este riesgo con las pruebas unitarias, no podemos eliminarlo, ya que dependemos de múltiples proyectos en desarrollo y plataformas que cambian. Por ejemplo, si un canal de venta hace cambios no anunciados a su API, podemos comenzar a tener errores sin que las pruebas lo detecten. En este caso, habría que reescribir las funciones y las pruebas unitarias.

Esta es una de las razones por las que escribir código bien documentado y con una buena legibilidad es muy importante, ya que quizás tendremos que reescribir las funciones y las pruebas unitarias, y es importante entender fácilmente la intención del código.

4.5. Despliegue

Ya que los cambios realizados en este trabajo de título son sobre una plataforma en uso de la cual una empresa depende de su correcto funcionamiento, se tomaron medidas para evitar, o por lo menos, disminuir la chance de que errores lleguen a código en producción, especialmente tomando en cuenta la profundidad de los cambios realizados.

Aprovechando que la plataforma tiene una estructura modular, se pasó a producción cada vez que un módulo había sido refactorizado. Esto hizo para evitar que grandes cantidades de modificaciones se acumularan y, por lo tanto, si alguna de estas produce algún error, este sea fácilmente identificable y solucionable.

Un problema surgió durante la refactorización del flujo de órdenes, ya que este cambio hace modificaciones profundas al funcionamiento fundamental de la plataforma. Durante el desarrollo y control de calidad, se hicieron múltiples pruebas para evitar que se produjera algún error en el nuevo flujo.

Pero no se tomó en cuenta que es un sistema en uso actualmente y todas las pruebas fueron realizadas en un ambiente de desarrollo limpio y que, por lo tanto, funcionaron sin problemas, pero al momento de pasar a producción, se duplicaron varias órdenes de compra existentes en el sistema viejo.

El problema se solucionó rápidamente, haciendo una verificación antes por cada orden en la ejecución del módulo del canal de venta, actualizando los campos de estados usando la orden ya existente del canal de venta. Esta solución a largo plazo puede ser eliminada, ya que todas las órdenes vendrán del nuevo sistema.

Capítulo 5

Validación

Con el objetivo de confirmar que la plataforma modificada resuelve los problemas de esta, mantiene la funcionalidad y cumple con el objetivo general, se validó el sistema de tres formas: primero haciendo un análisis del estado de la plataforma después de los cambios para contrarrestar con el estado anterior. La segunda y tercera manera es mediante resultados cuantitativos y cualitativos de la plataforma en comparación a antes de los cambios realizados.

En la sección 5.1 se analiza el estado de la plataforma después de los cambios para cada uno de los aspectos en los que se enfocó en la refactorización. Luego, en la sección 5.2 y 5.3, se presentan los resultados cuantitativos y cualitativos, respectivamente, luego de los cambios a la plataforma.

5.1. Estado después de los cambios

En el objetivo general se planteó que el código tiene múltiples falencias técnicas que se deben solucionar, en particular para mejorar la escalabilidad, la mantenibilidad y el rendimiento de la plataforma.

5.1.1. Escalabilidad

La escalabilidad de la plataforma mejoró considerablemente, esto es reflejado por la cantidad de módulos a los cuales hay que modificar para integrar nuevas plataformas.

En el caso de las integraciones de nuevos canales de venta, antes se debía crear un módulo que obtuviera las órdenes, interpretara los campos de las órdenes para integrarse a los facturadores, WMS y Couriers y también una integración particular a cada uno de estos. Mientras que actualmente, solamente se deben obtener las órdenes e interpretar los campos de estas para unirse al nuevo flujo de órdenes.

En el caso de las integraciones de nuevos facturadores, WMS y Couriers, las diferencias son todavía más pronunciadas, ya que antes se debía crear una integración por cada canal de venta existente, los cuales en este momento son 17. Mientras que

ahora, sin importar el número de canales de venta, solamente se deben integrar estas plataformas al módulo de mtdOrders el cual maneja todas las integraciones.

Por otro lado, la mejora en escalabilidad también se ve reflejada en la base de datos, ya que el cambio de estructura de esta mejora profundamente la capacidad de esta de almacenar los datos de forma eficiente y personalizable. Al punto que no se requiere intervención del equipo de desarrollo para agregar un nuevo cliente. Los cambios a la base de datos también reflejan una mejor mantenibilidad como se verá a continuación.

5.1.2. Mantenibilidad

La estructura de la base de datos no permitía una mantenibilidad efectiva y eficaz por parte del equipo de desarrollo, ya que las configuraciones de los clientes en cada uno de los canales de venta, facturadores, WMS y Couriers estaban bajo un mismo arreglo, dificultando de gran manera las actualizaciones y modificaciones a estos. Mientras que ahora todas las configuraciones están bajo una misma colección de separadas por canal de venta, facturador, etc, por lo tanto, son más fáciles de identificar, modificar y mantener actualizadas con las necesidades de los clientes.

Mientras que en el caso del código, la refactorización del flujo de órdenes produce que sea más fácil aprender y modificar el código, ya que sigue de forma más estricta el principio de responsabilidad única, simplificando la interpretación de las funciones y módulos. Esto a su vez significa que es más fácil identificar y solucionar errores que puedan surgir en la ejecución.

Además, la implementación de buenas prácticas en el código mismo de la plataforma le da una mayor calidad al código, haciendo que este sea más consistente y, por lo tanto, cómodo de leer. Y agregando a esto, las buenas prácticas asociadas a git, como el nombramiento de los commits y ramas producen que el simplemente revisar el historial de commits sea una documentación misma del historial de la plataforma.

Por último, las pruebas unitarias aportan directamente a la mantenibilidad de la plataforma al ayudar en la detección de errores y de cambios en las API de las diferentes plataformas asociadas, y al evitar que nuevos errores sean publicados.

5.1.3. Rendimiento

El rendimiento de la plataforma mejoró sustancialmente, ya que la ejecución asíncrona produjo mejoras al rendimiento de hasta un 97.3% en algunos canales de venta. En la siguiente sección se presentarán los resultados cuantitativos, los cuales incluyen las mejoras en tiempos de ejecución.

5.2. Resultados Cuantitativos

En esta sección se detallan los resultados de las modificaciones a la plataforma, en términos de tiempos de ejecución.

5.2.1. Tiempos de ejecución

Para obtener los resultados presentados a continuación, se utilizó la herramienta perf_hooks.js que permite medir y analizar el rendimiento de código, en particular, los tiempos de ejecución. Para entender los tiempos de ejecución, es necesario entender los estados de los módulos sobre los cuales se realizaron las mediciones.

- Antes: Este es el estado de la plataforma previo a cualquier modificación, y es el punto de partida para las comparaciones.
- Después/Ejecución asíncrona: Este es el estado de los módulos de la plataforma cuando ya se les realizó una primera aproximación a la refactorización. En específico, se les aplicó ejecución asíncrona a los módulos. Este estado se obtiene al cumplir el objetivo específico número dos.
- Refactorizado: Este es el estado final de los módulos de la plataforma una vez ya aplicada la refactorización a los módulos. Este estado se obtiene al cumplir los objetivos específicos 2 al 6.

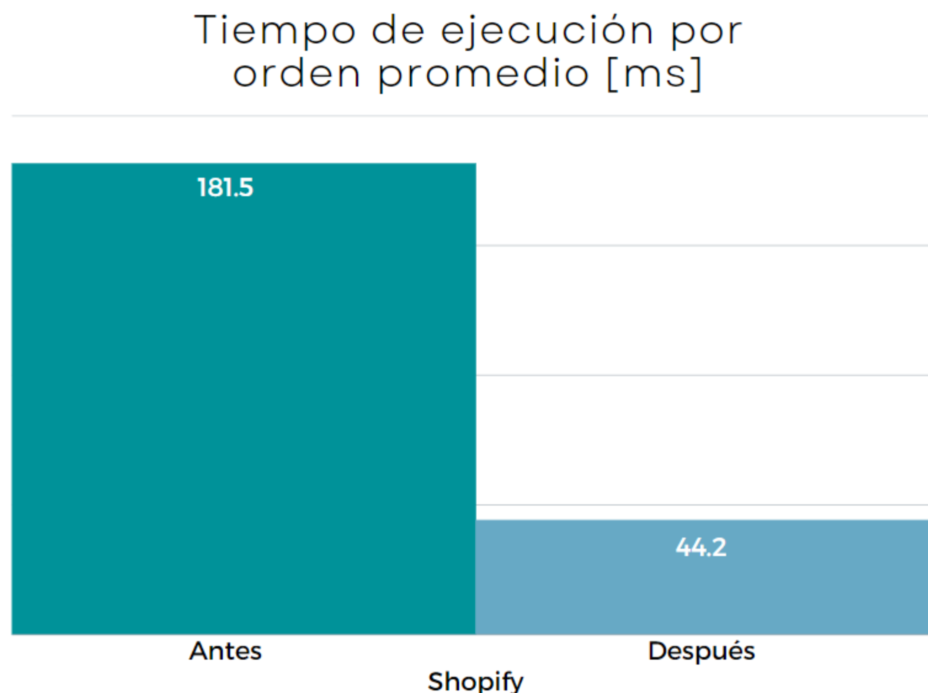

Figura 5.1: Mejora en tiempos de ejecución de Shopify

Los tiempos de ejecución mejoran considerablemente en comparación con el estado inicial del proyecto. Como se puede apreciar en los gráficos en esta subsección, hubo mejoras en los tiempos de ejecución de hasta un 97.3%. En particular, en la Figura 5.1, los tiempos de ejecución mejoraron sustancialmente, logrando reducirlos en un 75.7%.

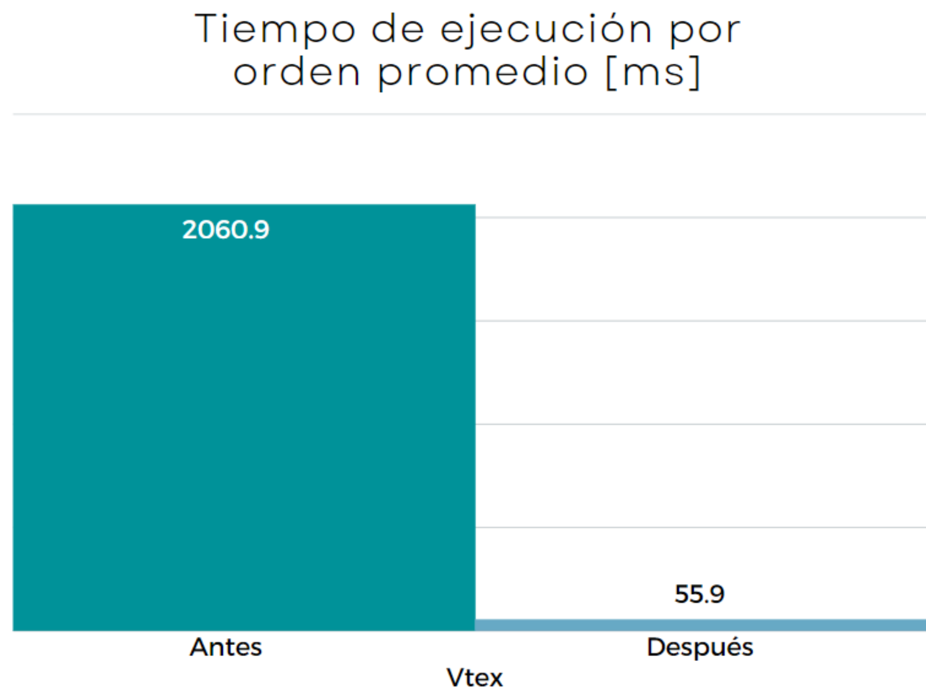

Figura 5.2: Mejora en tiempos de ejecución de Vtex

Como se puede observar en la Figura 5.2, la ejecución asíncrona entrega todavía mejores resultados para el canal de venta Vtex, reduciendo los tiempos de ejecución en un 97.3%. Esto probablemente se debe a la particularidad de que la ejecución de este canal incluye una consulta extra a la API del canal de venta para obtener los detalles de la orden, lo cual no es necesario para el resto de los canales de venta.

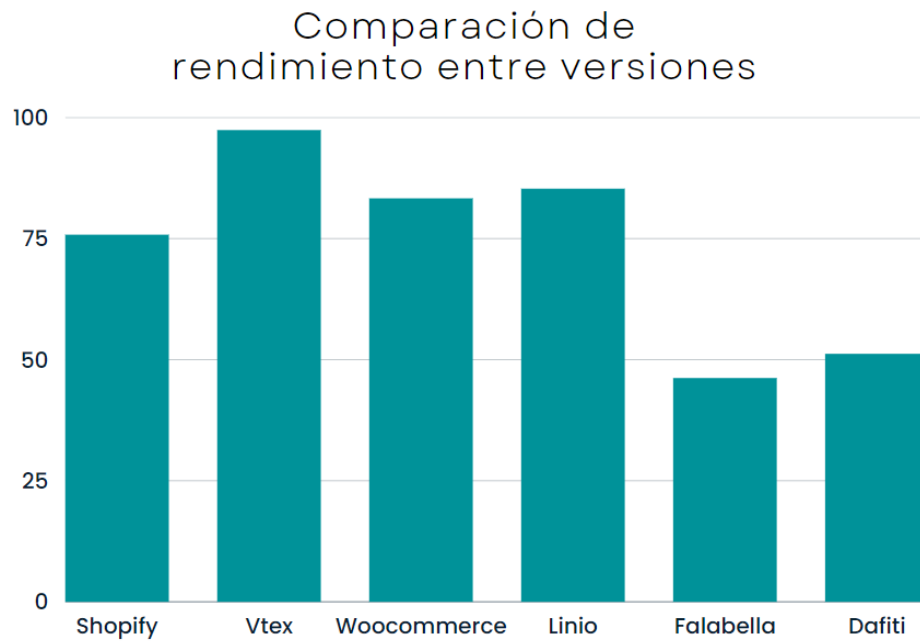

Figura 5.3: Porcentaje de mejora en el tiempo de ejecución.

Los gráficos anteriores muestran las diferencias en tiempos de ejecución antes y después de la aplicación de ejecución asíncrona en los diferentes módulos, ya que los cambios de la ejecución asíncrona mantuvieron el flujo y realizan los mismos procesos. En la Figura 5.3 se presentan los porcentajes de mejora en los tiempos de ejecución con luego de la aplicación de ejecución asíncrona con respecto al estado inicial. Los módulos mejoraron sustancialmente el rendimiento, logrando reducir hasta en un 97.3% el tiempo de ejecución en el caso de Vtex y en un 73% en promedio.

A diferencia de los cambios realizados en la refactorización del flujo de órdenes, ya que estos efectivamente dividen la ejecución en dos partes, reduciendo profundamente los procesos que se realizan en el canal de venta. Tomando lo anterior en consideración, a continuación se analizarán los tiempos de ejecución de los canales de venta entre los tres cambios.

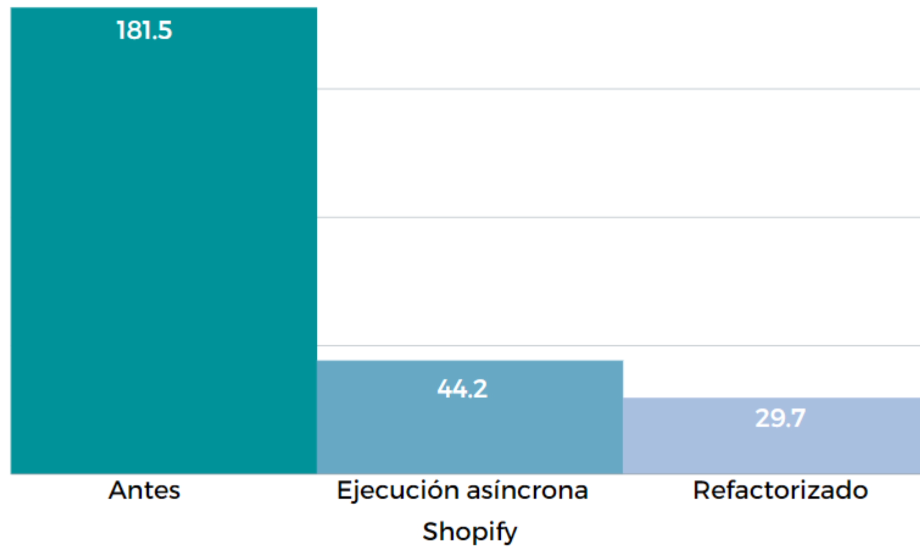

Figura 5.4: Mejora en tiempos de ejecución entre los tres estados para Shopify.

Como se puede apreciar en la Figura 5.4, la refactorización del flujo mejora todavía más la velocidad de ejecución en aproximadamente un 33% en el caso de Shopify, y se encontraron resultados similares para el resto de los canales de venta.

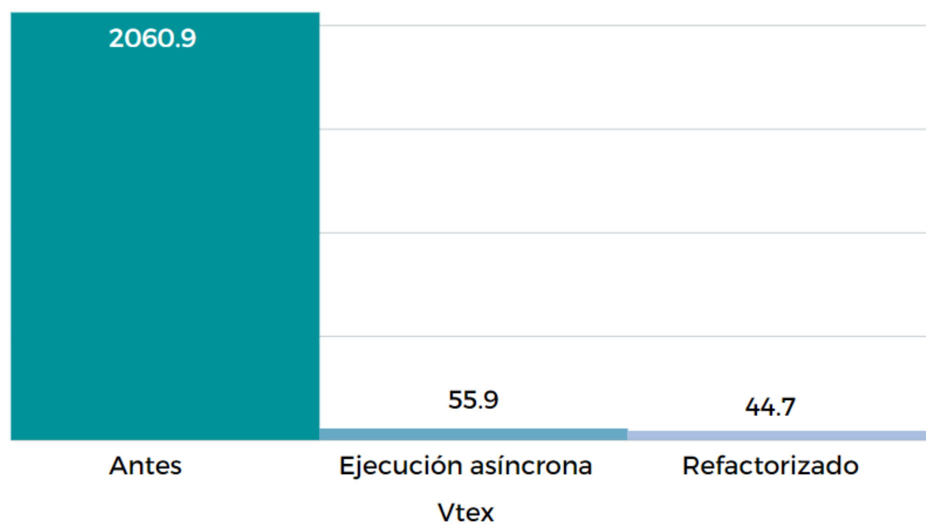

Figura 5.5: Mejora en tiempos de ejecución entre los tres estados para Vtex.

Como se puede observar en la Figura 5.5, al igual que para Shopify, se obtuvieron resultados exitosos, mejorando aún más el tiempo de ejecución. En este caso, el nuevo flujo mejora el tiempo de ejecución aproximadamente en un 20%.

5.3. Resultados Cualitativos

Esta sección presenta los resultados cualitativos obtenidos luego de los cambios realizados a la plataforma.

5.3.1. Reportes de Sonarlint

Los resultados obtenidos tras la refactorización de los módulos de canales de venta reflejan un gran progreso en términos de calidad y robustez del código. La disminución drástica, de hasta un 90%, en los errores reportados por SonarLint es un indicador claro de la mejora en la calidad del código, demostrando el éxito de la estrategia de refactorización implementada. Aunque, si bien, algunas alertas de bajo nivel persisten, su impacto es considerablemente menor.

5.3.2. Comentarios equipo de desarrollo

Se le entregó un cuestionario al equipo de desarrollo, compuesto por 6 desarrolladores, además del líder del equipo de desarrollo, todos son ingenieros civiles en computación o carreras afines, además en particular, el líder del equipo tiene amplia experiencia en el desarrollo de software. Para el cuestionario, se decidió realizar uno de respuesta abierta, en contraste a uno en escala de Likert, debido a la cantidad de participantes, ya que el equipo es pequeño, los resultados no habrían sido suficientemente precisos.

Este cuestionario tiene el propósito de evaluar los efectos de una refactorización sobre este mismo. Las respuestas proveen una idea sobre la percepción de los cambios realizados y su impacto en la calidad técnica y la comprensión del código. Estas observaciones dan otra perspectiva sobre los efectos reales de una refactorización exitosa.

5.3.2.1. Pérdida de familiaridad con el código

El proceso de refactorización en el desarrollo de software es esencial para mejorar la escalabilidad, mantenibilidad y rendimiento de la plataforma. Una de las preocupaciones comunes durante y después de una refactorización es la pérdida de familiaridad con el código. Para comprender mejor este problema, se le realizó la

siguiente pregunta al equipo de desarrollo: "¿Sabes cómo está organizado y puedes ubicarte en el código?" Las respuestas obtenidas proporcionan una perspectiva valiosa sobre cómo la refactorización afecta la comprensión y navegación en el código.

Una respuesta enfatiza la importancia de los nombres significativos de variables y funciones, lo que facilita la búsqueda y comprensión de diferentes partes del código. Además, la estructura del proyecto en capas, como la lógica de negocios, de servicios y de acceso a datos, es fundamental para encontrar fácilmente funciones específicas. Esto demuestra cómo la organización lógica y jerárquica del proyecto ayuda a los desarrolladores a ubicar rápidamente el código relevante.

Otra respuesta destaca la influencia positiva de los estándares de programación durante la refactorización. La aplicación de normas coherentes y consistentes en todo el código ayuda a los desarrolladores a encontrar y entender más fácilmente los elementos clave. Además, la parametrización adecuada de las variables en la base de datos y la estructuración correcta contribuyen a una mayor claridad en cuanto a los datos con los que están trabajando. Este enfoque también facilita la identificación de los parámetros y su uso en diferentes partes del código, lo que produce que el trabajo sea más eficiente y preciso.

Adicionalmente, se menciona cómo la refactorización puede impactar la ejecución asincrónica del código, lo que a su vez mejora la velocidad de prueba y desarrollo de nuevas funciones. Esta observación resalta cómo una refactorización exitosa no solo afecta la comprensión del código, sino que también puede tener beneficios tangibles en términos de rendimiento y eficiencia en la ejecución de tareas.

5.3.2.2. Conocimiento del código

Los resultados cualitativos obtenidos a través de la pregunta "¿Entiendes los cambios realizados y estos te facilitan el trabajo?" Brindan una información importante sobre los efectos de una refactorización exitosa en la comprensión y eficiencia del equipo de desarrollo. Estas respuestas reflejan cómo los cambios implementados durante la refactorización influyen en la forma en que los desarrolladores interactúan con el código y cómo estos cambios pueden afectar positivamente su trabajo diario.

Una de las respuestas destaca la importancia de la legibilidad y documentación del código. La refactorización exitosa se refleja en un código más claro y fácil de entender, con comentarios que describen tanto la entrada como la salida de las funciones. Esta observación resalta la necesidad de una comunicación efectiva en el código a través de la documentación adecuada, lo que a su vez mejora la comprensión y facilita el proceso de desarrollo.

Además de la legibilidad, las respuestas también enfatizan la escalabilidad del sistema como resultado de la refactorización. La integración de diferentes servicios,

como canales de venta y facturación, se vuelve más fluida y confiable. Esto muestra cómo la refactorización no solo aborda la comprensión del código actual, sino que también se anticipa a futuras expansiones y cambios en el sistema, lo que es crucial para el crecimiento sostenible de la plataforma.

Otra respuesta resalta cómo la refactorización no solo se traduce en una mejor comprensión del código existente, sino que también facilita la introducción de nuevas funciones. La modularización y segmentación del código permiten que los desarrolladores trabajen en nuevas funcionalidades sin la necesidad de revisar extensamente el código de otros módulos. Este enfoque modularizado agiliza el proceso de desarrollo y evita la dependencia excesiva entre diferentes partes del sistema.

La incorporación de pruebas unitarias también surge como un beneficio importante de la memoria. Las respuestas mencionan cómo las pruebas unitarias agilizan los pasos hacia la producción de nuevas funcionalidades. Esto evidencia un impacto no solo en el proceso de desarrollo, sino que también influye en la calidad del software y en la confianza en el rendimiento de las nuevas implementaciones.

En general, estas respuestas cualitativas ilustran cómo una refactorización exitosa no solo se centra en mejorar la comprensión y legibilidad del código, sino que también tiene un impacto positivo en otros aspectos del desarrollo. Desde la escalabilidad hasta la agilidad en el desarrollo y la confianza en las pruebas, estas respuestas indican cómo una refactorización bien planificada puede llevar a una mejora integral en el flujo de trabajo y en la calidad del software resultante.

5.3.2.3. Calidad técnica de la solución

Finalmente, se les pidió a los desarrolladores que entregarán comentarios libres de la calidad técnica de la solución y sus percepciones del estado final de la plataforma.

Los comentarios obtenidos del equipo de desarrollo sobre la calidad técnica de la refactorización entregan una nueva perspectiva sobre los impactos positivos que una mejora estructural en el código puede tener en el proyecto. Estas respuestas reflejan cómo una refactorización exitosa no solo se traduce en un código más legible y eficiente, sino que también soluciona problemas técnicos fundamentales y tiene beneficios tangibles para la plataforma.

Una de las respuestas resalta cómo la refactorización ha impactado en dos áreas clave: escalabilidad y rendimiento. Al reestructurar la parametrización de clientes y canales de venta, y al unificar el procesamiento de órdenes, se han aplicado conceptos que aprovechan la flexibilidad de MongoDB. Además, la ejecución asíncrona en el procesamiento de órdenes explora los beneficios para obtener el mejor rendimiento

posible de JavaScript. Esta respuesta resalta cómo la refactorización puede abordar desafíos técnicos específicos y optimizar la plataforma en áreas críticas.

Otra respuesta enfatiza la mejora en la calidad técnica como resultado de la refactorización. Se destaca la aplicación de principios básicos de programación, como una sintaxis correcta y comentarios adecuados, que previos a esta refactorización no se utilizaban de la manera adecuada. Esta observación subraya cómo una refactorización no solo se trata de reorganizar el código, sino de elevar el nivel de la solución en términos de buenas prácticas y estándares de programación. Además, se menciona una mayor eficiencia en los procesos gracias al uso de funciones asíncronas y una escalabilidad mejorada debido a la parametrización de los canales de venta. Estos cambios técnicos impactan directamente en la eficiencia y en la capacidad de la plataforma para manejar mayores demandas.

5.3.3. Mejores prácticas

El enfoque en las buenas prácticas durante la refactorización ha llevado a una transformación significativa en el código. Mediante la optimización de la estructura y la aplicación de estándares de programación, se ha mejorado su calidad y legibilidad. Esta legibilidad mejorada se traduce en un código más claro y comprensible, facilitando su mantenimiento y futuras modificaciones. La refactorización ha logrado no solo resolver deficiencias técnicas, sino también mejorar la eficiencia y la capacidad de respuesta del equipo de desarrollo. Es decir, la implementación coherente de las mejores prácticas ha dado como resultado un código más sólido y mantenible.

Capítulo 6

Conclusiones

El Motor de Transacciones Digitales de Instance tenía una deuda técnica que se había ido acumulando por más de un año, esto significaba que la plataforma llevaba todo ese tiempo recibiendo soluciones parche y mucho código en duro, lo que hacía muy difícil la mantención y la adición de nuevas funcionalidades o integraciones. Por lo tanto, fue necesario solucionar este problema para transformarla en una plataforma estable, escalable y eficiente.

Se tuvieron que realizar muchas modificaciones para solucionar este problema, primero que todo se investigó mejores alternativas para ejecutar el código, ya que el método que se utiliza no es el apropiado debido a la gran cantidad de desventajas que posee.

Para poder solucionar los problemas mencionados del código, primero se tuvo que resolver los problemas presentes en la base de datos, porque si esta no era capaz de almacenar la información efectivamente, la plataforma no podría crecer. Solo una vez solucionados estos problemas se implementó una parametrización de las configuraciones, quitando efectivamente todo el código en duro, además de implementar una ejecución asíncrona sobre el código que logró mejorar los tiempos de ejecución en hasta un 97%.

Fue durante el desarrollo del trabajo de título mismo que se identificó otra oportunidad de mejora, una unificación del flujo de órdenes que permite definir claramente las responsabilidades de los módulos, simplificando la plataforma en gran medida. Además, este cambio se enfoca directamente en la escalabilidad y mantenibilidad de la plataforma, reduciendo drásticamente las complejidades para integrar nuevas plataformas y mantenerlas.

Todos estos cambios permitieron que se integren pruebas unitarias a la plataforma, reduciendo enormemente las posibilidades de que errores lleguen a producción.

En cuanto a los objetivos propuestos, todos estos se cumplieron en su totalidad, ya que se logró mejorar de gran manera la mantenibilidad, la escalabilidad y el rendimiento del Motor de Transacciones Digitales de Instance. Solucionando las

falencias técnicas de este y permitiendo la integración de nuevas plataformas previstas por la integración de nuevos países a la plataforma.

6.1. Trabajo Futuro

Para continuar el trabajo sobre el sistema, después de la refactorización y de la integración de pruebas unitarias, todavía quedan mejoras adicionales que consolidarán aún más su eficiencia y funcionalidad.

Uno de los desarrollos clave que se pueden implementar es la incorporación de colas para la ejecución de procesos de facturación y WMS. La introducción de estas colas en el proceso simplificaría y modularizaría el envío de órdenes a estas plataformas, permitiendo una abstracción eficiente del proceso de ejecución. La adopción de tecnologías como RabbitMQ o Amazon SQS podría ser una estrategia efectiva para lograr esta optimización. Al descentralizar y modularizar el proceso, se aumenta la capacidad de respuesta y se reduce la posibilidad de interrupciones en el flujo de trabajo.

La unificación de las órdenes, como parte de la refactorización, abre nuevas oportunidades para el análisis de datos y la obtención de información valiosa sobre la empresa. Aplicar técnicas de ciencia de datos sobre esta nueva colección de datos podría proporcionar información estratégica que impulse la toma de decisiones informada en la empresa.

Finalmente, se identificó la necesidad de implementar un proceso automatizado para eliminar o anonimizar la información de identificación personal almacenada por la nueva colección de órdenes. La seguridad de los datos es fundamental hoy en día, y la adopción de este proceso mitigaría los riesgos asociados con la información sensible de los usuarios. La incorporación de medidas proactivas en términos de seguridad de datos minimizaría la posibilidad de brechas de datos que puedan afectar tanto a los individuos como a la integridad de la plataforma.

Bibliografía

[1] *Página web de Instance.*
`https://www.instancelatam.com/instance`,
01/2023

[2] *Fullcommerce.*
`https://www.df.cl/brandcorner/ecomsur/mario-miranda-ceo-de-ecomsur-el-e-commerce-no-arriesga-la-marca`,
01/2023

[3] *Asynchronous JavaScript.*
`https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing`,
01/2023

[4] *Ivers, J., Nord, R. L., Ozkaya, I., Seifried, C., Timperley, C. S., & Kessentini, M. (2022). Industry experiences with large-scale refactoring. Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* https://doi.org/10.1145/3540250.3558954

[5] *Chen, B., Jiang, Z. M., Matos, P., & Lacaria, M. (2019). An Industrial Experience Report on Performance-Aware Refactoring on a Database-Centric Web Application. 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). https://doi.org/10.1109/ase.2019.00066*

[6] Github Actions.
`https://docs.github.com/es/actions`,
01/2023

[7] IDE.
`https://es.wikipedia.org/wiki/Entorno_de_desarrollo_integrado`,
01/2023

[8] ConventionalCommits.
`https://www.conventionalcommits.org/en/v1.0.0/`,
01/2023

[9] Antipatrón de promesas.
`https://github.com/petkaantonov/bluebird/wiki/Promise-anti-patterns`,
01/2023

[10] Principio de Responsabilidad Única.
`https://es.wikipedia.org/wiki/Principio_de_responsabilidad_%`

C3%BAnica,
01/2023

[11] Mocha.js
https://mochajs.org/,
01/2023

[12] Chai.js
https://www.chaijs.com/,
01/2023

[13] *BSON.*
https://es.wikipedia.org/wiki/BSON,
01/2023

[14] *JSON.*
https://es.wikipedia.org/wiki/JSON,
01/2023

[15] *perf_hooks.*
https://nodejs.org/api/perf_hooks.html,
08/2023

Anexo

Diagramas de flujo de órdenes de Instance

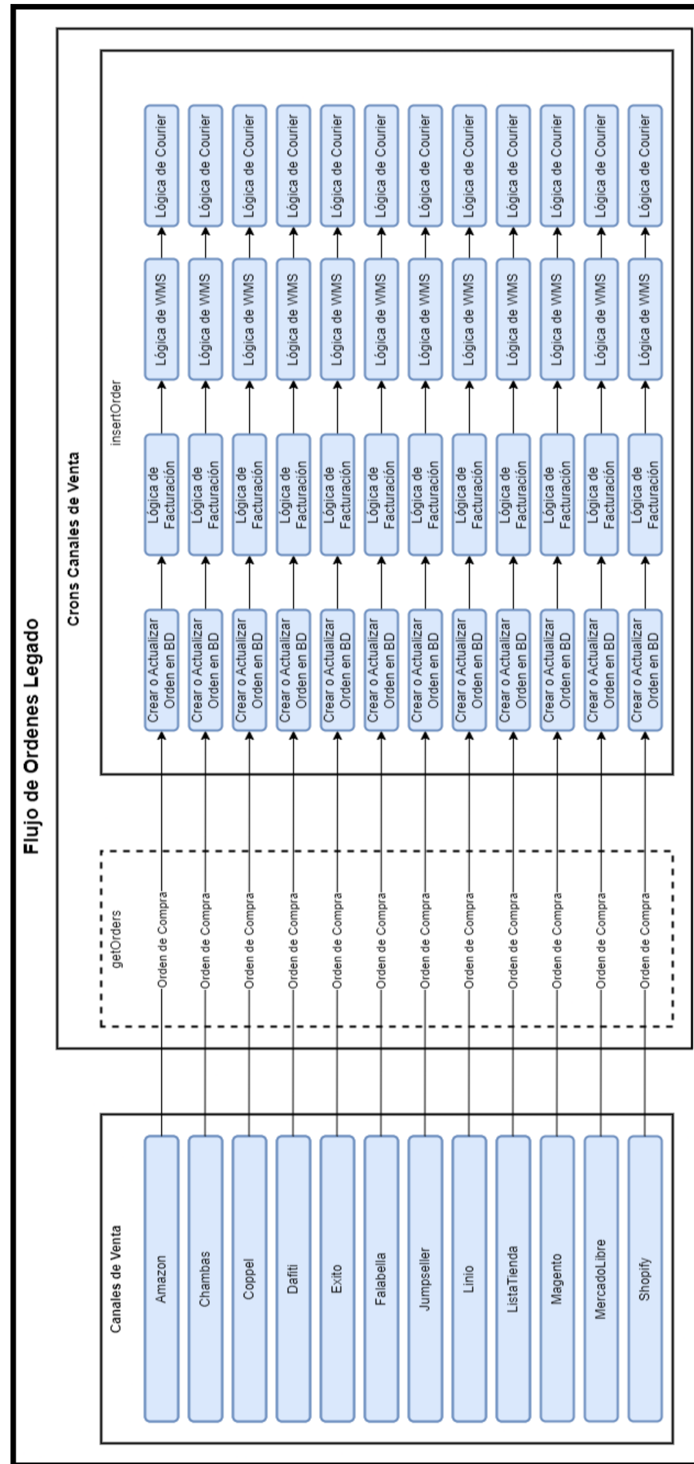

Figura A.1: Diagrama de flujo de órdenes en el MTD previo a la refactorización.

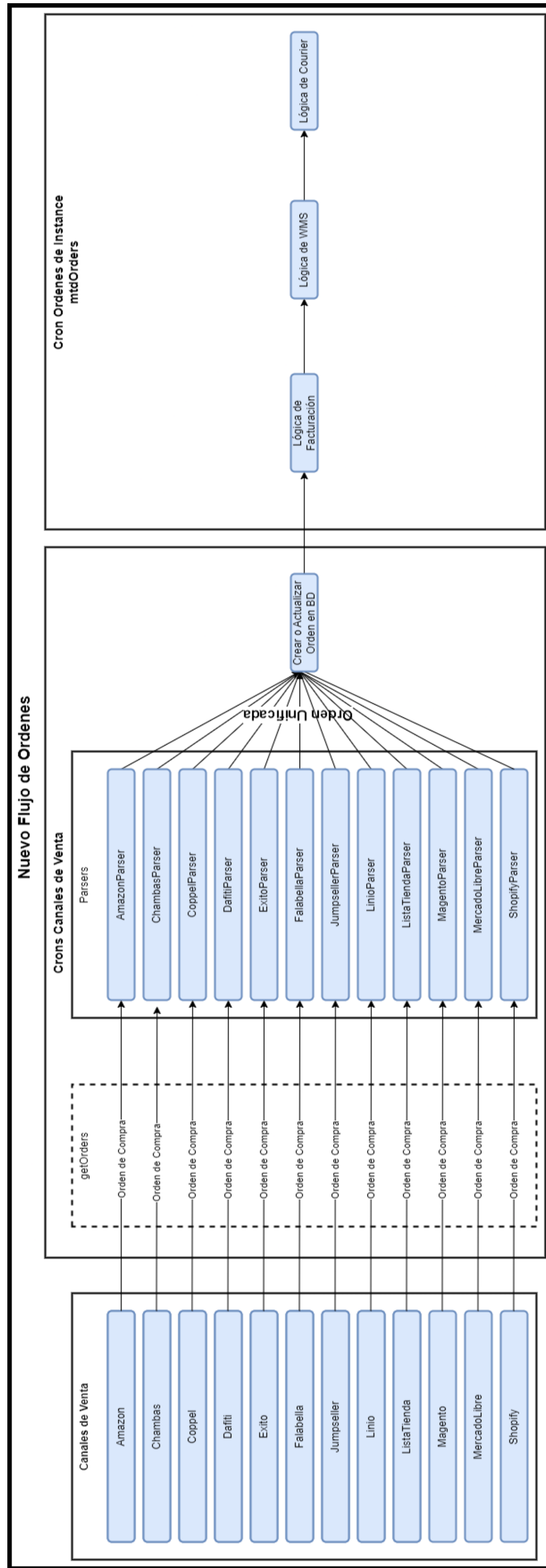

Figura A.2: Diagrama de flujo de órdenes en el MTD refactorizado.