UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Politécnica Superior de Gandia

Aplicación SaaS basada en microservicios para la gestión de pedidos en empresas de comercio electrónico

Trabajo Fin de Grado

Grado en Tecnologías Interactivas

AUTOR/A: Soler Navarro, Adrián

Tutor/a: Alberola Oltra, Juan Miguel

Cotutor/a: Sánchez Anguix, Víctor

CURSO ACADÉMICO: 2022/2023

Contenido

Ilustraciones

Acrónimos

- **SaaS:** Software as a Service
- **API:** Application Programming Interface
- **DDD**: Domain Driven Design
- **SOLID**: Software Design Principles
- **I/O**: Input/Output
- **ISO**: International Organization for Standardization
- **Endpoint**: Api procedure call
- **DTO**: Data Transfer Object
- **Topic**: Data stream about a particular topic, identified by a name
- **WebHook**: User-Defined HTTP callbacks
- **CRON JOB**: Job scheduler
- **AAA**: Test pattern (Arrange, Act, Assert)

Resumen

La pérdida de contacto orgánico entre las empresas de comercio electrónico y los clientes es un desafío común en la era digital. A medida que las transacciones se realizan en línea, se pierde la conexión personalizada que solía existir en las tiendas físicas. Para superar esta brecha y no perder, e incluso, reforzar el vínculo entre comprador y vendedor, el proyecto desarrollado pretende mejorar el sistema de comunicación basándose en un sistema de notificación en tiempo real, que, a la vez, consigue aliviar la carga que supone para las empresas tener que mantener una comunicación constante y personalizada con cada cliente.

Palabras clave

Software como servicio, comercio electrónico, seguimiento de pedidos, fidelización de clientes, tecnología

Abstract

The loss of organic contact between eCommerce businesses and customers is a common challenge in the digital age. As transactions take place online, the personalized connection that used to exist in physical stores is lost. In order to overcome this gap and not lose, and even strengthen the link between buyer and seller, the developed project aims to improve the communication system based on a real-time notification system, which, at the same time, manages to alleviate the burden that for companies to have to maintain constant and personalized communication with each client.

Keywords

Software as a Service, e-commerce, order tracking, customer loyalty, technology

1. Introducción

En los últimos años, el comercio electrónico ha experimentado un crecimiento significativo. Cada vez más personas prefieren realizar compras en línea debido a la comodidad, la variedad de opciones y la facilidad de comparar precios. El avance de la tecnología y el acceso generalizado a Internet han impulsado este crecimiento, con un aumento constante de las transacciones y ventas en plataformas de comercio electrónico. El comercio electrónico ha revolucionado la forma en que compramos y vendemos productos, convirtiéndose en un sector en constante expansión y transformación.

1.1 Contexto y problemática

En el ámbito de la venta electrónica [1], la comunicación fluida y el intercambio de información entre empresas y clientes desempeñan un papel crítico para establecer y mantener relaciones comerciales sólidas. Como se puede observar en la Ilustración 1, en un contexto nacional, este tipo de venta es muy habitual, sobre todo en edades comprendidas entre 25-54 años. En el contexto de envío de pedidos mediante empresas de transporte, es vital mantener una buena relación vendedor-comprador, ya que brinda a estos últimos la información sobre la última etapa de sus compras, permite conocer en tiempo real el estado de sus pedidos y planificar la recepción eficientemente. Para las empresas, garantizar que los clientes estén debidamente informados y satisfechos en cada etapa del proceso de envío se convierte en una prioridad absoluta.

Ilustración 1. Perfil de usuario de compras online en España

Dentro de estos seguimientos, se incluye información relevante como los datos del cliente, el transportista, el estado actualizado del envío y las fechas estimadas tanto de salida como de entrega, así como cualquier percance que pueda haber acontecido durante el proceso de envío. Al proporcionar esta información de manera precisa y oportuna, las empresas logran mejorar notablemente la experiencia del cliente y reducir la incertidumbre y la ansiedad asociadas con la entrega de los pedidos. Para alcanzar esta eficiencia en la comunicación, las empresas deben apostar por automatizar sus procesos de envío y centrar sus esfuerzos en sistemas de seguimiento en tiempo real capaces de ofrecer información confiable a los clientes. Además, les permite despreocuparse de parte de la carga de gestión y mantenimiento de la comunicación una vez el pedido ha salido de sus dominios.

En resumen, la transferencia de datos de envío se convierte en un pilar esencial para mantener relaciones comerciales efectivas y satisfactorias en el ámbito de la venta electrónica [2]. Al automatizar los procesos de envío y aprovechar sistemas de seguimiento en tiempo real, las empresas no solo mejoran la experiencia del cliente, sino que también aumentan su nivel de satisfacción. Este enfoque centrado en el cliente puede generar una mayor lealtad hacia la marca y, en última instancia, asegurar el éxito sostenible de las empresas en el competitivo mercado de la venta electrónica. Al poner en práctica estas estrategias, las empresas se posicionan como líderes innovadores en la industria, destacando por su capacidad para proporcionar una experiencia de compra confiable y satisfactoria que fortalece los vínculos con los clientes y establece una base sólida para el crecimiento y el éxito a largo plazo.

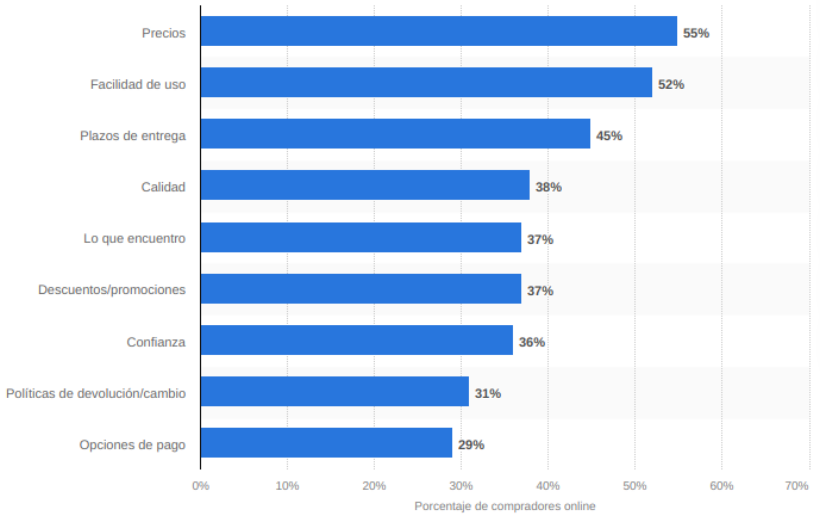

Ilustración 2. Motivos de satisfacción de los internautas sobre el e-commerce en España

Como se puede apreciar en la Ilustración 2, uno de los principales motivos de satisfacción tiene que ver con los plazos de entrega y, por consiguiente, con el conocimiento del estado del pedido durante todo el proceso de envío.

Analizando los puntos clave de esta problemática, nos sugiere la posibilidad de considerar una solución en forma de aplicación *SaaS (Software as a Service)* [3] para mejorar la comunicación entre vendedor y comprador.

Esta herramienta permite que el vínculo gane solidez y consiga afianzar la relación entre ambas partes. Con una plataforma centralizada y una comunicación en tiempo real, este tipo de SaaS se convierte en un elemento fundamental en el proceso de venta en línea.

2. Objetivos

A continuación, se define tanto el objetivo principal de este proyecto, como los distintos objetivos secundarios que cubren la mayoría de los casos de uso encontrados y para los cuales necesitamos implementar una solución.

2.1 Objetivo principal

Desarrollar una aplicación SaaS que ofrezca a las empresas de comercio electrónico la posibilidad de externalizar el seguimiento de sus pedidos hacia diversos transportistas y la posterior gestión y centralización de la comunicación con el cliente mediante una plataforma de mensajería instantánea.

2.2 Objetivos secundarios

Objetivos técnicos

1. Desarrollar una aplicación como servicio para conectarse a diferentes APIs (Application Programming Interface) y poder enviar y recibir información en tiempo real.
2. Implementar microservicios siguiendo el patrón arquitectónico hexagonal.
3. Aplicar buenas prácticas de desarrollo de software para garantizar un diseño de software modular, escalable y mantenible.

4. Utilizar diferentes bases de datos según las necesidades.
5. Aplicar una correcta autenticación de usuarios para evitar cualquier acceso no deseado.
6. Garantizar el correcto funcionamiento de la infraestructura interna mediante pruebas y cumplir con los requisitos y expectativas del cliente.

Objetivos de negocio

1. Brindar una solución efectiva y confiable para la gestión de envíos y seguimiento de pedidos.
2. Mejorar la experiencia del cliente al proporcionar notificaciones actualizadas sobre el estado de los envíos.
3. Incrementar la eficiencia operativa de las empresas clientes al automatizar los procesos de seguimiento de envíos.
4. Ofrecer una arquitectura escalable y flexible que permita adaptarse a los requisitos cambiantes del negocio.
5. Garantizar la seguridad de los datos de los clientes y la integridad de la información de seguimiento.
6. Obtener una ventaja competitiva en el mercado al ofrecer una solución avanzada y eficiente en la gestión y notificación de envíos.

Objetivos personales y académicos

1. Ampliar conocimientos y comprender mejor las herramientas del mercado para identificar sus fortalezas y debilidades y poder desarrollar una perspectiva crítica y analítica en la evaluación de soluciones tecnológicas.
2. Aplicar los conocimientos adquiridos de manera profesional, tanto en el ámbito laboral como en la vocación personal, mediante la elaboración y defensa de argumentos sólidos y la resolución efectiva de problemas relacionados con el área de estudio.
3. Fomentar el pensamiento crítico y reflexivo para poder abordar de manera profunda y significativa los temas relevantes de la disciplina.
4. Fomentar el pensamiento creativo e innovador en la resolución de problemas dentro del área de estudio, buscando enfoques novedosos y soluciones originales.
5. Perfeccionar la actitud de aprendizaje continuo.

3. Marco teórico

En el marco teórico, se explorarán dos áreas clave: el análisis de competencias y las tecnologías (infraestructura y software). El análisis de competencias es una sección esencial en cualquier estudio exhaustivo, ya que nos permite evaluar y comparar las fortalezas y debilidades de las herramientas disponibles en el mercado. Por otro lado, el área de tecnologías permite averiguar la infraestructura tecnológica requerida para respaldar los sistemas informáticos, así como el software utilizado en las aplicaciones y programas. Estos dos aspectos son esenciales para comprender el entorno empresarial actual y los recursos necesarios para operar eficientemente.

3.1 Análisis de soluciones similares

Nos enfocaremos en dos soluciones ampliamente utilizadas en la gestión de envíos: ParcelLab y AfterShip. Ambas herramientas en línea han servido de producto de investigación para obtener ideas de negocio e intentar encontrar necesidades no cubiertas para poder mejorar el producto ya existente en el mercado.

ParcelLab es una plataforma de seguimiento y gestión de envíos que proporciona a las empresas y a los clientes finales una experiencia de seguimiento transparente y personalizada. Su principal foco consiste en brindar visibilidad y control sobre los envíos en tiempo real. El hecho de operar en multitud de países y admitir decenas de empresas de transporte la hace interesante para situarse en el punto de mira de comercios de venta electrónica internacionales. Aun así, se han detectado algunos contras que se exponen a continuación:

- Elevado número de integraciones con APIs de transportistas: Esto puede poner en riesgo la disponibilidad de los datos de seguimiento al depender de diversos y diferentes sistemas de terceros para recibir información, incluso sobrecargar el sistema de atención al cliente en un hipotético fallo múltiple, con la consiguiente insatisfacción y pérdida de ingresos de las empresas contratantes.
- Personalización de notificaciones: Dado que permite la personalización de los mensajes de seguimiento enviados a los usuarios. A priori, es interesante que una empresa pueda acercar la identidad de su marca con notificaciones, pero si el número de empresas contratantes aumenta significativamente, puede generar desacuerdos en las preferencias específicas de algunas empresas y perder fidelización.

- Complejidad de implementación: La configuración del SaaS de ParcelLab puede requerir cierto nivel de conocimientos técnicos, lo que supone un problema para empresas con recursos limitados o sin experiencia técnica suficiente.

Por otro lado, "AfterShip" es otra herramienta popular en el mercado que se centra en el seguimiento de envíos y ofrece actualizaciones en tiempo real a los clientes. Aunque destaca por su capacidad para proporcionar información actualizada sobre el estado de los envíos, también presenta limitaciones en términos de personalización y automatización de procesos.

- Limitaciones en la integración de transportistas: Aunque AfterShip admite una amplia gama de transportistas, es posible que no cubra todas las opciones de envío disponibles en cada región. Esto puede ser una limitación si una empresa trabaja con transportistas específicos que no están integrados en AfterShip.
- Soporte al cliente limitado en planes gratuitos: Si se requiere soporte adicional o atención personalizada, es posible que se necesite actualizar al plan de pago correspondiente.

Ambas soluciones también comparten ciertas limitaciones:

- Planes de subscripción costosos:

Si bien es cierto que para utilizar las funciones principales ambas opciones son de pago, el hecho de vincularse con muchos transportistas hace aumentar sus requisitos de infraestructura, así como el precio de dicha vinculación para poder recibir datos en tiempo real.

- Métodos de notificación tradicionales y unidireccionales: Estas plataformas utilizan métodos de notificación como el correo electrónico, los cuales aportan una forma poco directa, moderna y atractiva de presentar los datos. Además, no permiten la comunicación bidireccional, es decir, que el cliente pueda responder, ya que la inmensa mayoría son mensajes automáticos.

Con este análisis detallado se pretende cimentar las bases para presentar la solución propuesta en este trabajo. Al aprovechar las debilidades identificadas en estas herramientas similares, se presenta una nueva opción que supera las limitaciones anteriores y ofrece un enfoque más completo y eficiente en la gestión de envíos. A través de este análisis comparativo, también podemos destacar las ventajas distintivas de la herramienta propuesta y demostrar cómo puede ayudar a las empresas a optimizar sus procesos de envío y mejorar significativamente la experiencia del cliente

3.2 Tecnologías

La mayoría de las veces, por no decir todas, es necesario imaginar el peor contexto al que puede estar expuesta una aplicación en un ambiente de producción. Un software de esta magnitud debe amortiguar una constante avalancha de solicitudes de creación de trackings, lidiar con la autenticación [6] de distintas empresas, custodiar comunicaciones y datos sensibles de millones de clientes, y notificar en tiempo real a todos los usuarios que esperan sus pedidos.

Por otro lado, es fácil pensar que el poder de contención, estabilidad y mantenibilidad solo recae en el escalado horizontal, cuantos más y mejores servidores, todo irá bien. La realidad es que este no es el único talón de Aquiles; aunque con una infraestructura física envidiable puede aligerar la carga de procesamiento, los principales retos son de la arquitectura del software y las tecnologías implementadas.

3.2.1 Infraestructura

Para poder ofrecer a las empresas el mejor software posible, es necesario identificar qué tipo de solución tecnológica debemos brindarle para que se adecúe a sus necesidades.

Existen muchos tipos de plataformas para ofrecer, desde FaaS, BaaS, IaaS, PaaS y SaaS, entre otros. La elección en este caso es clara, SaaS nos ofrece:

- Acceso inmediato
- Menor complejidad técnica
- Escalabilidad simplificada
- Costos reducidos
- Actualizaciones y mejoras continuas
- Mayor enfoque en su negocio principal
- Implementación más rápida
- Soporte técnico especializado
- Mejoras en la seguridad de los datos

¿Por qué se elige esta solución y no se realiza un desglose exhaustivo de las demás? La respuesta es clara. Las opciones restantes delegan demasiado trabajo al cliente. Esta opción ofrece una solución rápida, escalable y rentable sin que la empresa que contrata la herramienta tenga que preocuparse por la infraestructura y la gestión técnica; de todo esto se ocupa TrackingLab.

3.2.2 Software

Para construir el software se deben seleccionar una combinación de lenguajes de programación, herramientas y marcos en base a las necesidades a cubrir.

Lo primero es valorar ante qué tipo de proyecto nos encontramos, ¿Es un proyecto pequeño, de rango medio o complejo? Como se ha visto anteriormente, la aplicación puede lidiar con muchas compañías con cientos o miles de clientes. Así pues, la complejidad brilla por si sola. En este tipo de proyectos, es crucial seleccionar un motor de ejecución potente. **Node.js** se ejecuta en el servidor y proporciona una arquitectura orientada a eventos y basada en el modelo de I/O (Input/Output) no bloqueante, lo que le permite manejar múltiples solicitudes de manera eficiente y escalable. Node.js se usa para crear aplicaciones web en tiempo real, API, microservicios y aplicaciones de red.

El lenguaje de programación utilizado es **Typescript**, debido a su poderoso tipado asegura una buena calidad y robustez en el código, además, es muy escalable y tiene una muy buena integración con librerías y frameworks de JavaScript.

Como todo pastel, necesita una guinda, y aquí es donde entra el framework de desarrollo **NestJS**. Este marco combina la potencia de TypeScript y Node.js para ofrecer un desarrollo backend altamente escalable, modular y mantenible. Su enfoque basado en TypeScript proporciona beneficios como tipado estático, mejor productividad y mayor claridad del código, mientras que su integración con Node.js aprovecha las capacidades de desarrollo de aplicaciones backend eficientes y de alto rendimiento. Además, ofrece decoradores y metadatos, así como como inyección de dependencias, útiles para proporcionar una sintaxis limpia y un ecosistema modular, respectivamente.

Por otro lado, hay que valorar distintas bases de datos dependiendo de las necesidades. En ocasiones nos enfrentamos a datos cambiantes y que no guardan una estructura estática que los relaciona, para ellos tenemos la base de datos relacional **MySQL**. Otras veces topamos con datos dinámicos o ausentes de atomicidad, para ellos usamos una base de datos no-relacional como **MongoDB**. Por último, existen datos que necesitan estar disponibles por un corto o medio periodo de tiempo, pero que son “volátiles” y no necesitan una larga persistencia, aquí entran en juego las bases de datos en memoria como **Redis**.

Es importante destacar el manejo de contenedores **Docker** para poder generar un ecosistema de aplicaciones encapsulado, portable, aislado del sistema operativo, escalable y centralizado. Sin esta herramienta, sería muchísimo más tedioso generar una infraestructura de manera rápida y libre de problemas de compatibilidad. Por otra parte, se elige **Apache Kafka** como plataforma de datos en tiempo real, útil para el intercambio de mensajes entre productores y consumidores entre diferentes microservicios, aportando un desacoplamiento y múltiples ventajas que se analizan en los apartados siguientes.

Para finalizar, el sistema de pruebas viene de la mano de Jest. Es un paquete de prueba popular en Node.js que se utiliza para escribir y ejecutar pruebas automatizadas

4. Propuesta

Para abordar el problema planteado, se propone una solución tecnológica mixta en forma de SaaS que permita ayudar a las empresas de comercio electrónico a comunicarse eficazmente con sus clientes sobre el estado de sus pedidos para poder dirigir sus esfuerzos hacia otras ramas del negocio. Con esta fusión se puede brindar un control del trato de las empresas hacia sus clientes a la vez que le permite desentenderse, en cierta forma, del sistema de sincronización y notificación de seguimientos.

La plataforma permite personalizar varios parámetros que recibirán los usuarios en cada tipo de notificación, de esta manera, estos pueden percibir las notificaciones de una manera más cercana y confiable cuando hay un cambio en el estado de su pedido.

Esta aplicación solo está pensada para actuar en un ámbito nacional, es decir, en España, esto se ha decidido por un motivo principalmente. Si se quiere trabajar con más países hay que añadir una complejidad innecesaria al proyecto, ya que se tiene que tener en cuenta muchas variables como son los estándares ISO (International Organization for Standardization), notificaciones personalizables por idioma, lenguas habladas, distintas ramas de transportistas, entre otras; al ser problemas que no añadirían un valor en cuanto a tecnologías y herramientas utilizadas ni modificarían las funcionalidades que se quieren plasmar, se ha descartado la ampliación a otros países.

Por otra parte, con esta solución, las empresas de comercio electrónico pueden externalizar el seguimiento de sus pedidos y la posterior gestión de la comunicación con los clientes, lo que les permite centrarse en su negocio principal. A la vez, los clientes están siempre informados del estado de sus pedidos, lo que mejora su experiencia de compra y la confianza depositada no afecta [5].

La conexión con el servicio de Telegram presenta ventajas significativas [15] en comparación con los sistemas de notificación actuales basados en correo electrónico. A diferencia de estas (que suelen ser unidireccionales y no permiten la interacción del usuario), Telegram, a través de un bot, ofrece la posibilidad de una comunicación directa y en tiempo real. Esto significa que los usuarios pueden tener recibir los datos mediante notificaciones a través de una aplicación de uso cotidiano y menos formal. Esta capacidad de interacción mejora la experiencia del usuario al proporcionar un canal de comunicación más dinámico y efectivo, permitiendo una mayor satisfacción y un mejor servicio al cliente.

4.1 Requisitos

A continuación, se exponen los principales requisitos a nivel de usuario diferenciando entre las compañías que contratan nuestro SaaS, los desarrolladores de las compañías que usarán nuestra API para enviar los trackings generados y los clientes finales que recibirán los estados de sus pedidos.

- Compañía cliente y equipo de desarrollo

- Envío y procesamiento de trackings generados por empresas de comercio electrónico subscritas al SaaS.
- Sincronización de los trackings emitidos por parte de las empresas con las APIs de los transportistas correspondientes para obtener información sobre el estado de los envíos en tiempo real.
- Disponibilidad de una API pública para la integración correcta de la infraestructura cliente.
- Documentación clara e intuitiva de la API de TrackingLab para realizar la integración correctamente.

- Requisitos internos como SaaS:

- La aplicación SaaS debe recibir y procesar trackings de empresas cliente.
- La aplicación debe vincular los trackings con las APIs de los transportistas correspondientes.
- La plataforma de la empresa de logística debe notificar al SaaS de cada evento que se produce en el estado del tracking sincronizado.
- Se debe almacenar correctamente toda la información de los trackings, respetando la confidencialidad e integridad de los datos.
- El SaaS debe poder facilitar un registro de todas las operaciones de intercomunicación entre la empresa y el cliente final.
- Debe notificar a los usuarios finales mediante la aplicación de Telegram.

- Usuario final:

- Notificación en tiempo real de los eventos que se produzcan en los estados de los envíos a través de la aplicación de Telegram.
- Notificación personalizada con los datos del comprador para una mejor experiencia de usuario.

En resumen, la aplicación debe recibir, procesar, sincronizar y notificar el estado de los envíos a los clientes finales y a las empresas y sus desarrolladores.

4.2 Arquitectura general

Para que resulte más comprensible la estructura de la que se va a hablar, a continuación, se muestra la Ilustración 3. Desde el envío de seguimientos por parte de las empresas (Client Company), pasando por la plataforma desarrollada (TrackingLab) hasta el dispositivo de los compradores (Telegram API) encargado de recibir las notificaciones de sus pedidos. En posteriores apartados analizaremos cada uno de estos bloques.

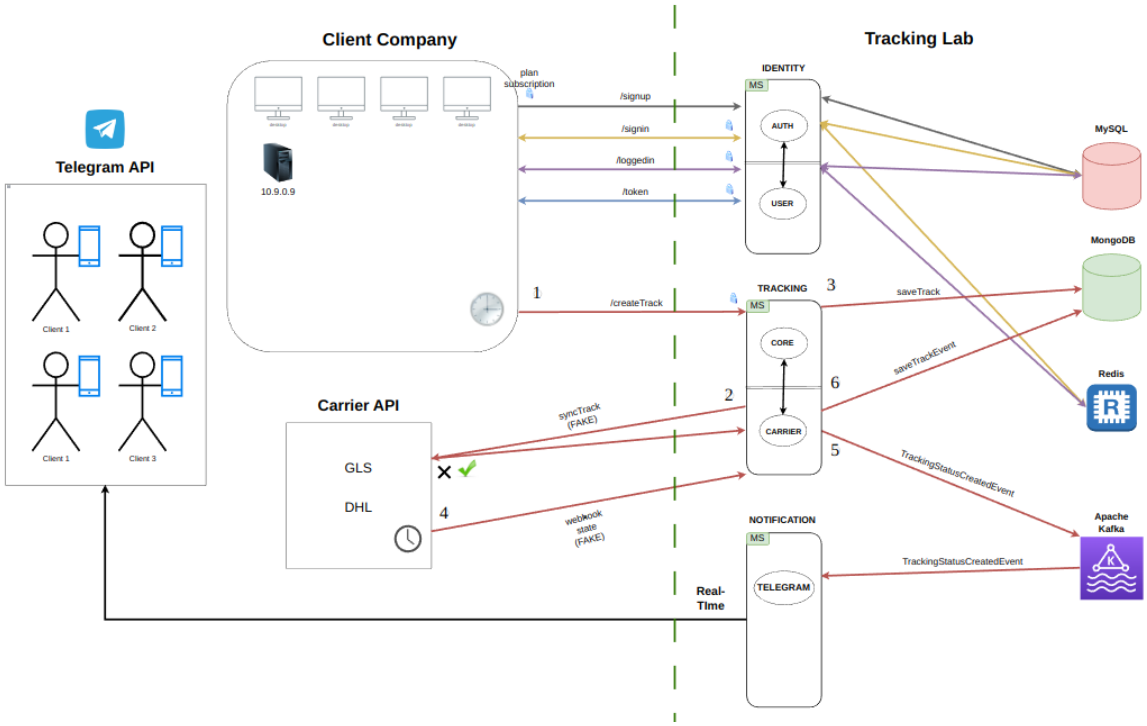

Ilustración 3. Diagrama de interacción general del proyecto

(Ilustración incluida en Anexos con mayor calidad)

Una vez analizado el ecosistema general, se exponen los requisitos tecnológicos para poder llevar a cabo la correcta implementación de la infraestructura con tal de cubrir todas las necesidades expuestas en el apartado anterior.

La aplicación se diseñará utilizando Node.js y NestJS, aprovechando las ventajas que ofrece NestJS en cuanto a la escalabilidad y modularidad. Node.js es adecuado cuando necesitas hacer muchas cosas al mismo tiempo, sobre todo muchas operaciones I/O (acceso a ficheros de configuración, bases de datos, “cron Jobs (Job scheduler)”, conexiones de clientes...) a la vez. También es especialmente bueno para aplicaciones “realtime”, que necesitan mantener una conexión persistente entre el navegador y el servidor además de un sistema de intercambio de mensajes por eventos. No hay que olvidar la gran comunidad que reside detrás de Node y JavaScript, algo muy a tener en cuenta de cara a un completo desarrollo con opción a solventar todos los problemas que pueden ir surgiendo.

Se utilizará una arquitectura basada en microservicios [4] utilizando el patrón arquitectónico hexagonal [8], siguiendo el enfoque de desarrollo guiado por **DDD** (Domain Driven Design). Como se puede observar en la Ilustración 4, una arquitectura monolítica centraliza toda la lógica bajo el mismo paraguas a diferencia de una basada en microservicios.

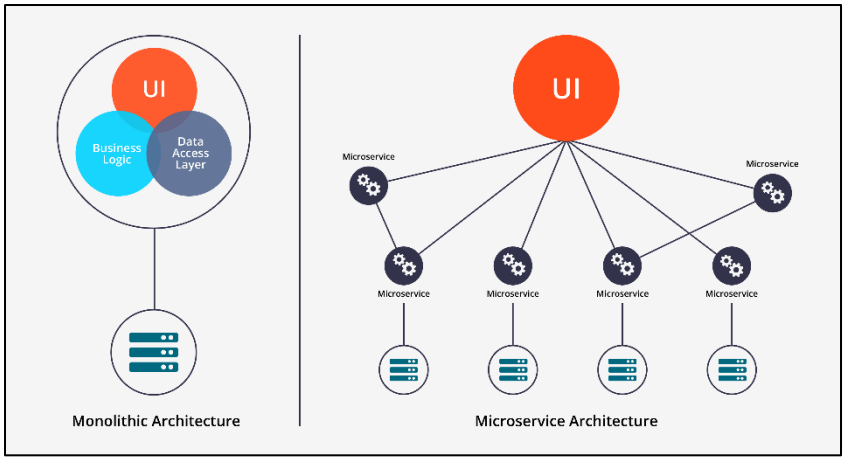

Ilustración 4. Diferencia entre arquitectura monolítica y basada en microservicios

La decisión de usar una arquitectura de microservicios es clara, es el tipo de estructura de desarrollo perfecta para satisfacer las necesidades de implementar y hacer cambios en el software de forma rápida y sencilla. Al separar todo en pequeñas cajas modulares, se pueden replicar e integrar con facilidad, añadiendo, modificando o eliminando, dependiendo necesidades. En definitiva, los microservicios permiten la integración continua, agilidad en los cambios y un menor riesgo de afectar a todo el sistema cuando algo falla.

El patrón arquitectónico hexagonal promueve la separación de asuntos mediante el encapsulamiento de la lógica en diferentes capas de la aplicación [10]. Además, este patrón permite que se pueda hacer test más fácilmente, un mejor aislamiento de los componentes y mayor control sobre el código de negocio específico.

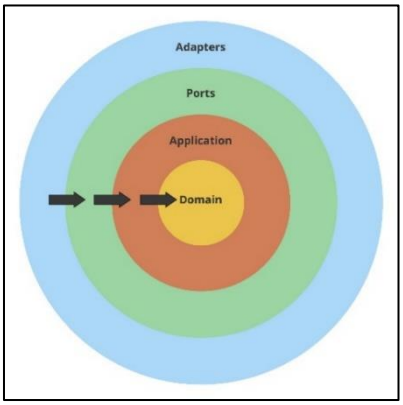

Ilustración 5. Patrón arquitectónico hexagonal separado por capas

Como se muestra en la Ilustración 5, la separación de responsabilidades en distintas capas nos permite aislar y darle toda la relevancia al dominio, ya que este tipo de enfoque ofrece una profunda conexión entre la implementación y los conceptos del modelo y núcleo del negocio.

La ventaja que nos otorga el enfoque DDD es el de poder acercar más el software al dominio [12] [14] [16], y, por lo tanto, al cliente. La lógica de negocio reside en la capa de aplicación diferenciada por casos de uso.

Cabe destacar, que el enfoque que se acaba de mencionar facilita el uso de patrones de diseño como SOLID (Software Design Principles). Estos principios sirven de esquema para construir un código flexible, limpio, reutilizable y mantenible, es decir, un código de calidad. Los principios SOLID simplifican la creación de pruebas, ayudan a entender la arquitectura, mejoran la cohesión disminuyendo el acoplamiento entre microservicios, módulos, clases, etc.

La aplicación contará con tres bases de datos: MySQL, MongoDB y Redis, según las necesidades. MySQL se utilizará para almacenar información estática sobre las empresas subscritas a las ofertadas por el SaaS, como los perfiles y credenciales de la empresa, y cierta configuración del software para realizar funcionalidades como la sincronización de trackings con los transportistas.

MongoDB almacena información sobre los trackings recibidos por las empresas, los estados recibidos de las APIs de transportistas y registros variados.

Redis se utiliza para almacenar información en caché como ciertos tokens de acceso o para aliviar la carga de las bases de datos principales.

El motivo de usar estos tres tipos de bases de datos tan diferentes entre sí no es porque sea la mejor solución en todos los casos, de hecho, se podría llegar a prescindir de alguna; el motivo principal es que permite ver su funcionamiento y ayuda a separar conceptos y responsabilidades de persistencia aun existiendo mínimas diferencias que nos hagan decantarnos por una u otra.

Para las pruebas se hace uso de **Jest**. Permite verificar que todas las clases y métodos cumplen su función además de asegurar que los casos de uso están completamente cubiertos, o, al menos, la cobertura roza el 100%.

Para evitar la instalación local de diferentes bases de datos y de Apache Kafka junto a su interfaz gráfica, Docker permite integrar todo en un único contenedor [13] y así poder trabajar en diferentes máquinas sin preocuparse por la configuración del entorno. Además, proporciona un alto nivel de aislamiento y seguridad.

Por último, se hace uso de un sistema de control de versiones como **Git** para poder llevar un control y seguimiento de todo el proyecto mediante un historial de manera local y remota.

4.3 Diseños

El diseño principal consta de tres microservicios independientes junto a un paquete auxiliar que contiene todas las herramientas comunes entre ellos. Cada microservicio puede o no estar formado por uno o varios módulos que representan distintas funcionalidades, pero relacionadas, dentro de su propio ecosistema. En la Ilustración 6 podemos observar de qué forma están distribuidos.

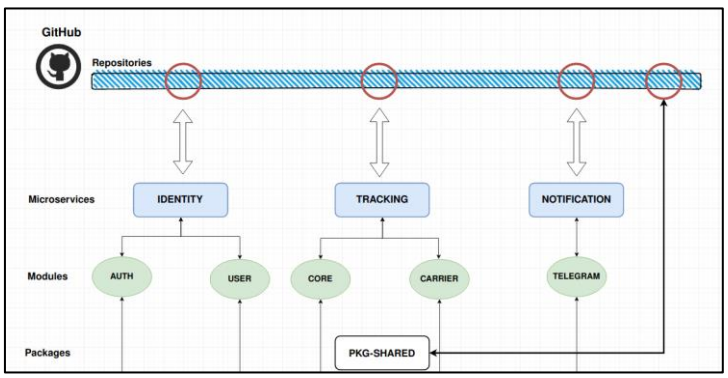

Ilustración 6. Distribución de repositorios en GitHub

La rama superior (Git) está compuesta por cuatro repositorios en GitHub, uno por cada microservicio y otro para el paquete de herramientas auxiliar.
Como se ve, todos los microservicios se nutren de las funcionalidades del paquete compartido, por lo tanto, al extraerlo y hacerlo público, desacoplamos cualquier lógica auxiliar necesaria entre los microservicios, ganando en flexibilidad, escalabilidad, mantenibilidad y mejora en el despliegue continuo.

A continuación, se muestra el esquema general, tanto de la aplicación desarrollada como de los servicios externos que interactúan de forma activa o pasiva con ella.

Esta infraestructura de software permite que los desarrolladores de la compañía cliente se conecten a través de varios puntos de entrada de la API expuesta, que tienen rutas para registrar usuarios, iniciar sesión y obtener tokens de acceso, así como una ruta posterior al acceso, donde se pueden crear los seguimientos que luego se utilizarán [11].

Respecto a los tokens de acceso, ofrecen a los desarrolladores la posibilidad de realizar llamadas a acciones y/o recursos protegidos, lo que garantiza la seguridad y la privacidad de los datos [7].

Hay que tener en cuenta que los puntos de acceso o “endpoints (Api procedure call)”, no tienen por qué ser llamados siempre desde una interfaz gráfica, pueden ser llamados por procesos automatizados (sin interacción humana) en los servidores alojados en las empresas cliente. Por poner un ejemplo, la llamada al endpoint de registro de usuario o (“/signup”) es llamado desde la interfaz gráfica de la página web del SaaS una vez el cliente se ha subscrito a un plan de este. Por otro lado, la llamada al endpoint de “/createTrack” se llama desde un proceso automatizado en el momento que la compañía emite un nuevo número de seguimiento desde su infraestructura y este va a ser rastreado por el SaaS.

Después de visualizar la infraestructura completa, se analiza en detalle cada entrada de la API para posteriormente visualizar el conjunto de secuencia mediante diagramas de interacción.

1. Registro de usuario.
2. Inicio de sesión
3. Inicio de sesión con token
4. Generar token de autenticación
5. Crear tracking
6. Eventos
7. Notificación Telegram

1. Registro de usuario

El primer punto es el registro de usuario (“/signup”), para poder registrarse es necesario subscribirse a un plan de TrackingLab.

Una vez hecho, los datos de registro llegan al microservicio de “Identity” donde son tratados y almacenados en una tabla de clientes de MySQL. Cabe destacar que, en este proceso, se generan un “client_id” y “client_secret” aleatorios y únicos. El primero sirve para identificar inequívocamente al usuario y el segundo como credencial encriptada. Ambos se utilizan para realizar llamadas posteriores de obtención de tokens de acceso.

2. Inicio de sesión

El siguiente paso corresponde al inicio de sesión, el cual recibe el nombre de usuario y la contraseña usados en el registro.

Una vez se llama a este endpoint, el microservicio de Identity inicia el proceso de autenticación [9], se llama al módulo de usuarios y este cumple su función de obtener el usuario (si existe) que se está solicitando; una vez obtenido, se comprueba que el hash de contraseña guardado es igual a la contraseña introducida como parámetro después ser sometida al mismo método de hashing.

Si la comparación es exitosa, se ejecuta la lógica del servicio de inicio de sesión, generando y guardando un “session_token” en memoria en la base de datos Redis. Este token se utiliza para substituir a la contraseña en posteriores inicios de sesión automáticos y tiene una duración de un año desde el momento de su creación.

Una vez concluye esta tarea, tanto el “client_id” como el “client_secret” son visibles y están disponibles en la parte del cliente. Así como el “session_token” es almacenado en el navegador sin que el usuario tenga que preocuparse de ello.

3. Inicio de sesión con token

Como se ha comentado, en futuros inicios de sesión y mientras haya un “session_token” disponible, se llamará al endpoint de “/loggedin”. El parámetro de entrada de la contraseña se sustituye por el token de sesión. La única diferencia interna es que, al autenticar el usuario, primero se obtiene el token de sesión de Redis asociado al usuario, se compara, y si todo es correcto, entonces el guardián otorga el acceso y el flujo de datos puede continuar. En el caso contrario, se arroja una excepción de autenticación. Por último, se devuelven las mismas credenciales salvo el token de sesión, el cual ya lo tiene guardado el usuario.

4. Generar token de autenticación

El siguiente endpoint recibe dos parámetros que también se utilizan para autenticar al usuario y comprobar que es quién dice ser. Tanto el “client_id” como el “client_secret” son visibles y están disponibles en la parte del cliente.

La función principal de esta llamada es obtener un token de acceso con tiempo limitado, que usaremos para hacer llamadas a recursos o acciones protegidas de la API. Esto proporciona un extra de seguridad de cara a llamadas no autorizadas, puesto que el tiempo de expiración es bajo, un atacante que logre capturar este token no tendrá mucho tiempo antes de que expire.

Para que el guardián nos otorgue el acceso, es necesario que el “client_secret” aportado por el usuario sea igual que el secreto desencriptado de la base de datos de MySQL. Una vez realizada esta validación, siempre se generará un nuevo token de acceso de tipo “Bearer” con un tiempo de expiración de cinco minutos; más tarde este será devuelto al cliente para que pueda utilizarlo.

5. Crear tracking

En posesión del token de acceso a la API de TrackingLab, el equipo de desarrollo de cada empresa puede utilizarlo para enviar la información de seguimiento a través de la plataforma. Mediante una petición HTTP, se añaden algunos parámetros obligatorios y otros opcionales.

Una vez la solicitud es recibida en la plataforma, se comprueba que el usuario es quién dice ser mediante una petición HTTP interna entre los microservicios de Tracking e Identity. Si todo está en orden, da comienzo el proceso de creación de trackings mediante estos pasos:

a) Desde el módulo Core, se obtiene la configuración de transportistas para validar que el transportista es válido y soportado por la plataforma. Además, se validan todos los parámetros recibidos para evitar inconsistencias posteriores.
b) Se genera una llamada de sincronización del módulo Core hacia el módulo Carrier facilitándole todos los datos de seguimiento necesarios para la conexión entre la plataforma y la API de transportista objetivo.
c) Se inicia la petición HTTP entre TrackingLab y la API de terceros. Independientemente de que haya una respuesta y se complete la sincronización, los datos del seguimiento son guardados en la base de datos de MongoDB, marcando el seguimiento con una bandera de estado, indicando verdadero o falso dependiendo de si se completó la acción o hubo algún error. Esto permite llevar un control sobre los seguimientos para reintentar reencaminarlos y evitar la pérdida de datos en el sistema.

6. Eventos

Mediante webhooks (User-Defined HTTP callbacks), las plataformas de transportistas envían notificaciones HTTP a un endpoint expuesto por la API de TrackingLab. De esta manera nos transfieren los datos necesarios de cambios en los estados de los pedidos.

Estos datos son procesados por el microservicio de Tracking y formateados para generar un evento “TrackingStatusCreatedEvent”, el cual sirve de contenedor para transportarlos de forma asincrónica hasta los consumidores que estén subscritos al topic (Data stream) en donde se depositará dicho evento. Después se guardan esos eventos con datos para llevar un registro actualizado.

7. Notificación Telegram

El microservicio Notification procesa todos los eventos que escucha a través del topic, los descompone y genera un mensaje personalizado que se envía a la API de Telegram a través de un bot llamado TrackingLabBot.

Automáticamente, el cliente final recibe en su dispositivo un mensaje automático de Telegram facilitándole todos los detalles del estado de su pedido.

En los anexos se pueden encontrar todos los diagramas de interacción de inicio a fin.

4.4 Implementación

En primer lugar, se agruparán en varios contenedores Docker los distintos servicios, se analizará la estructura de los microservicios, después se indagará sobre la implementación del microservicio de Identity, desglosando todo el sistema de autenticación y el funcionamiento de los guardianes; más tarde se expondrá el microservicio de Tracking junto a la recepción de seguimientos, la sincronización con las APIs de transportistas y la publicación de eventos de cambios de estado. Después de esto, se explicará como el microservicio Notification consume los eventos internos y se comunica con la interfaz pública de Telegram para notificar a los usuarios finales. Por último, se hará un análisis de las herramientas comunes utilizadas en todos los microservicios y de la estructura de la base de datos que se ha utilizado.

- Implantación de servicios con Docker
- Análisis de la estructura de un microservicio
- Creación de los microservicios y sus respectivas API
- Creación de la base de datos y sus tablas
- Integración con las APIs de los transportistas
- Implementación de Apache Kafka para responder a eventos
- Configuración de la notificación a los usuarios finales mediante Telegram

4.4.1 Implantación de servicios con Docker

Mediante un fichero de configuración llamado "*docker-compose.yml*", se declaran y orquestan todos los servicios que se usarán posteriormente en la aplicación. En este archivo, se pueden definir varios servicios que representan contenedores individuales y establecer la configuración necesaria para cada uno de ellos, como su imagen base, los puertos expuestos, los volúmenes, etc. También se puede definir dependencias y relaciones entre los servicios. Cuando se ejecuta el comando "*docker-compose up*", Docker utilizará la información proporcionada en el archivo de configuración para crear y ejecutar los contenedores necesarios. Como ejemplo, en la Ilustración 7 se presenta una parte del fichero de configuración, adjuntado en su totalidad en los anexos de este proyecto.

```yaml
#KAFKA
kafka:
  image: confluentinc/cp-kafka:latest
  hostname: kafka
  container_name: kafka
  depends_on:
    - zookeeper
  ports:
    - "9094:9094"
  environment:
    KAFKA_BROKER_ID: 1
    KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
    KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka:9092,EXTERNAL://localhost:9094
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
    KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
    KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
```

Ilustración 7. Configuración de Kafka en el fichero docker-compose.yml

4.4.2 Estructura de un microservicio

La estructura general de un microservicio con Node.js en TypeScript usando el framework de desarrollo NestJS se ve de la siguiente manera:

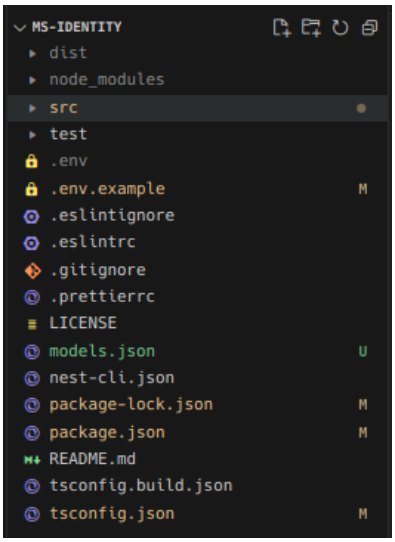

Ilustración 8. Estructura general de un proyecto en Node.js

Sin entrar en detalle del funcionamiento de un proyecto en Node.js y su configuración, la estructura dispone de dos directorios (Ilustración 8) principales donde se desarrolla la mayor parte del código.

Test:

En este directorio se realizarán las pruebas. Aquí se define toda la lógica referente a las pruebas end-to-end, integrales y unitarias. Este apartado lo veremos más adelante.

Src:

Esta es la ubicación para todo el código desarrollado. Es la ruta principal para ficheros de configuración usados en la compilación y ejecución del proyecto.

Cada microservicio tiene definida una estructura basada en capas dentro de esta carpeta. Como se ha comentado en apartados anteriores, se utiliza un enfoque por capas basado en una arquitectura hexagonal y centrada en el dominio del negocio (DDD); por lo tanto, la estructura de esta carpeta se declara de esta manera en la Ilustración 9.

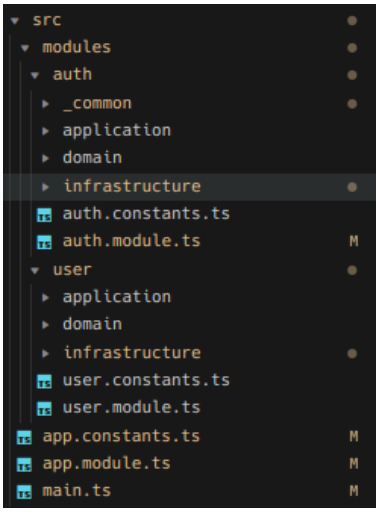

Ilustración 9. Estructura de un microservicio basado en la arquitectura hexagonal

Se pueden diferenciar dos módulos, *auth* y *user*, respectivamente. El primero se encarga de toda la lógica de autenticación de la aplicación, mientras que el segundo se encarga de la administración de usuarios.

Cabe destacar que siendo puristas y ciñéndonos al primer principio SOLID sobre responsabilidad única o (*Single-Responsibility Principle*), estos módulos deberían estar separados en dos microservicios totalmente independientes ya que cada microservicio es responsable de una sola funcionalidad del sistema. El motivo no se hace así porque las principales funcionalidades del SaaS ya reflejan el sentido de este tipo de diseño separándose en microservicios, por lo que fusionando estos módulos en uno solo evitamos una complejidad añadida no necesaria para el propósito de este trabajo.

Cada módulo siempre contiene tres directorios inamovibles, estos son *application*, *domain* e *infraestructure*, pudiendo haber directorios extra como es el caso de *_common*, el cual se encarga de albergar los guardianes en el módulo de *auth*.

Al adentrarnos en la jerarquía del módulo, mediante la Ilustración 10 se puede ver la importancia del flujo de la información entre las diferentes capas:

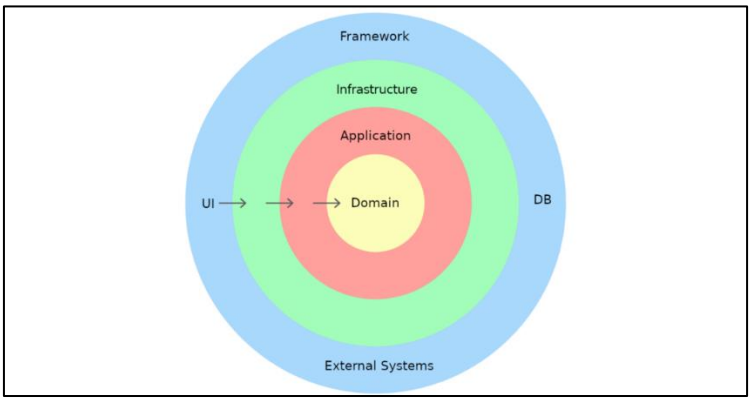

Ilustración 10. Flujo de datos basado en aquitectura hexagonal y centrado en el dominio

La capa de infraestructura es el cajón de sastre, la que interactúa con el exterior. Esta capa es la encargada de recibir todas las peticiones HTTP, las peticiones originadas en la interfaz del usuario del cliente, eventos, peticiones a través del framework de NestJS, la encargada de conectarse a *APIs* de terceros o a las diferentes bases de datos. Cualquier conexión entrante o saliente es obligación de la infraestructura. Es la encargada de utilizar los puertos y adaptadores necesarios para manejar la comunicación con la infraestructura interna.

Al desglosar esta capa encontraremos las siguientes carpetas:

- ***Controller:*** se encarga de definir los puntos de entrada del microservicio mediante el protocolo *HTTP.* Aquí se definen cada una de las rutas de acceso hacia la capa de aplicación y sus respectivos servicios.

- ***Dto* (*Data Transfer Object*):** declara el cuerpo de la respuesta que será devuelta por el *endpoint.*
- ***Persistence:*** se ocupa de definir los puertos de comunicación del microservicio hacia las *APIs* de terceros y bases de datos, entre otros. Aquí dentro se definen los modelos o esquemas que utilizará el software al que desee conectarse el microservicio. Por ejemplo, si necesitamos una conexión a *MySQL*, se definen los modelos que reflejan la estructura de las tablas en la base de datos, así como la lógica que se usa para conectarse a ella y realizar las acciones pertinentes.
- ***Subscriber***: declara los puntos de entrada de eventos a través de los consumidores asociados a algún topic de *Kafka*.

Cada petición o respuesta que llega por la capa de infraestructura se transfiere hacia la capa de aplicación. Esta capa contiene lo siguiente:

- ***Service*:** es la carpeta encargada de administrar la lógica principal de la aplicación. La capa intermedia entre el dominio del negocio y el exterior. Aquí se definen los diferentes casos de uso necesarios para realizar las tareas necesarias al responder a cualquier petición.
- ***Dto*:** declara el cuerpo de la respuesta que será recibida por el endpoint y a su vez por el servicio.

La capa más interna corresponde al dominio. En DDD, la capa de dominio es una capa que representa el núcleo del negocio y define las reglas de negocio, la lógica y el comportamiento del sistema. Los objetos de la capa de dominio representan las entidades del negocio y su comportamiento, y son independientes de la implementación técnica del sistema.

- ***Entity*:** definen las entidades y/o agregados del dominio. Los agregados son una forma de agrupar y organizar las entidades de un dominio en unidades lógicas coherentes. Un agregado suele contener una entidad raíz y uno o más objetos relacionados, y es una unidad coherente e independiente de otras unidades de agregados.
- ***Exception:*** Aquí se crean las excepciones propias del módulo con tal de lanzar errores personalizados con información relevante para trazas o monitorización.

- ***Repository:*** Contiene las abstracciones (interfaces/clases abstractas) de las que depende la infraestructura. Esto viene dado por el principio de inversión de dependencias que propone que los módulos de alto nivel (que contienen la lógica de negocio) no deben depender directamente de los módulos de bajo nivel (que contienen implementaciones concretas). En su lugar, ambos deben depender de abstracciones comunes y genéricas.

4.4.3 Microservicios

Microservicio Identity:

Este microservicio consta de dos módulos independientes, el de autenticación, y el de usuarios.

El módulo de autenticación es el punto de acceso principal para todas las llamadas que se dirigen a nuestro SaaS. Cada petición necesita autenticar al usuario y validar que es quién dice ser y que, a su vez, disponga de los permisos necesarios para obtener la acción o recurso solicitado.

Como se ha comentado antes, el controlador recibe las solicitudes entrantes. A continuación, se muestra la Ilustración 11 del controlador del módulo de autenticación para visualizar como se reciben los *DTOs*, como se llama al respectivo servicio en la capa de aplicación y finalmente como se devuelve el *DTO* de respuesta.

```
@Controller('auth')
export class AuthController extends BaseHttpResponse {
  constructor(
    private readonly signupService: SignupService,
    private readonly signinService: SigninService,
    private readonly loggedinService: LoggedinService,
    private readonly generateTokenService: GenerateTokenService
  ) {
    super();
  }

  @Post('signup')
  public async signup(@Body() dto: SignupDto) {
    const response: boolean = await this.signupService.run(dto);
    return this.success(response);
  }

  @UseGuards(SigninGuard)
  @Post('signin')
  public async signin(@Req() req) {
    const response: LoggedinResponseDto = await this.signinService.run(req.user);
    return this.success(response);
  }

  @UseGuards(LoggedinGuard)
  @Post('loggedin')
  public async loggedin(@Req() req) {
    const response: LoggedinResponseDto = await this.loggedinService.run(req.user);
    return this.success(response);
  }

  @UseGuards(ApiAuthGuard)
  @Post('token')
  public async generateToken(@Req() req) {
    const response: GenerateTokenResponseDto = await this.generateTokenService.run(req.user);
    return this.success(response);
  }

  @UseGuards(JwtGuard)
  @Post('checkAccess')
  public async checkUserAccess() {
    return true;
  }
```

Ilustración 11. Controlador de autenticación del microservicio Identiy

Dado que el flujo de datos completo es parecido en todos los *endpoints,* vamos a fijar el objetivo de la explicación sobre un único endpoint, en este caso el de Signin.

Cuando una petición llega a la ruta *www.trackinglab.com/api/v1/identity/auth/<x>,* si existe un decorador *@UseGuards()*, este intercepta la petición antes de que llegue al controlador. El guardián se encarga de aplicar la lógica necesaria para autenticar al usuario y dejar que la solicitud continue o en su defecto, lanzar una excepción de autorización fallida.

```
@Injectable()
export class SigninStrategy extends PassportStrategy(Strategy, 'local-signin') {
  constructor(private authService: AuthService) {
    super();
  }

  async validate(username: string, password: string): Promise<any> {
    const user = await this.authService.validateUserSignin(
      username,
      password
    );
    if (!user) throw new UnauthorizedException();
    return user;
  }
}
```

Ilustración 12. Estrategia del guardián para inicios de sesión

Como se puede observar en la Ilustración 12 y haciendo uso del paquete *passport* de *NestJS,* el método *validate()* recibe dos parámetros que posteriormente traslada al servicio de authService donde se aplica la lógica para obtener el usuario guardado en la base de datos y comprobar que el password es correcto.

```
@Injectable()
export class AuthService {
  constructor(
    private readonly getUserService: GetUserService,
    private readonly tokenRepository: TokenRepository
  ) {}

  public async validateUserSignin(username: string,password: string): Promise<User | undefined> {
    const user: User = await this.getUserService.run({username});
    const isValidPassword: boolean = await compareStrWithStrHashed(password,user.password);
    if (!isValidPassword) return undefined;
    return user;
  }
```

Ilustración 13. Servicio aplicación encargado de la lógica de autenticación del usuario

En la Ilustración 13, el método *validateUserSignin()* del servicio *AuthService* conecta con el módulo *User* ya importado en la configuración del microservicio *de NestJS,* que a su vez exporta el servicio *getUserService().* Este método permite delegar la funcionalidad de obtención de usuarios al módulo correspondiente y con ello, disponer del usuario en el servicio de *AuthService.*

Una vez recibido, se llama a la función *compareStrWithStrHashed()* para comparar dos cadenas hasheadas bajo la misma semilla. Si son iguales, el guardián dejará que la petición llegue al controlador y con ello, los datos al *DTO* de entrada, en el caso contrario, se devolverá una excepción.

Si la petición ha llegado satisfactoriamente al controlador, se puede ver en la Ilustración 14 como se llama al servicio *SigninService* encargado de dar sesión al usuario.

```
@Injectable()
export class SigninService {
  constructor(private readonly tokenRepository: TokenRepository) {}

  public async run(user: User): Promise<LoggedinResponseType> {
    const sessionToken: string = generateRandomHex(32);
    await this.tokenRepository.saveSessionToken(user.username, sessionToken);

    return {
      client_id: user.clientId,
      client_secret: decryptStr(user.clientSecret, user.password),
      session_token: sessionToken,
    };
  }
}
```

Ilustración 14. Método principal del caso de uso de Inicio de sesión

En este punto estamos en la capa de aplicación, es entonces cuando se aplica la lógica del caso de uso en sí mismo. En este ejemplo consta de obtener un token de sesión aleatorio y guardarlo en el repositorio de *Redis* para poder compararlo en futuros inicios de sesión.

Para llevar a cabo esta función, se ha inyectado el TokenRepository, el cual es una abstracción definida en la capa de dominio de nuestro microservicio. En la Ilustración 15, en el array de providers, se puede observar cómo esta abstracción se vincula al repositorio ubicado en la capa de infraestructura mediante las propiedades “provide” y “useClass”.

```
@Module({
  imports: [
    UserModule,
    KafkaModule,
    PassportModule,
    JwtModule.register({
      secret: jwtConstants.secret,
      signOptions: { expiresIn: `${jwtConstants.expires_in}s` },
    }),
  ],
  controllers: [AuthController],
  providers: [
    SigninStrategy,
    LoggedinStrategy,
    ApiAuthStrategy,
    JwtStrategy,
    AuthService,
    SignupService,
    SigninService,
    LoggedinService,
    GenerateTokenService,
    {
      provide: TokenRepository,
      useClass: TokenRedisRepository,
    },
  ],
  exports: [],
})
export class AuthModule implements OnModuleInit {
  constructor(
```

Ilustración 15. Configuración del módulo de autenticación en el microservicio Identiy

Dentro de la configuración del módulo de auth sobre NestJS, concretamente en la propiedad "*providers*", se puede definir la vinculación entre una abstracción y la clase que la usa, de tal forma que el servicio *SigninService* de la capa de aplicación (alto nivel) no dependerá de *TokenRedisRepository* ubicado en la capa de infraestructura (bajo nivel), sino que dependerá de la abstracción implementada (*provide: TokenRepository*) que facilita un contrato para poder comunicarse entre ambas capas sin acoplamiento.

Es este punto en el que debemos recordar que el flujo de datos en una arquitectura hexagonal siempre fluye hacia el dominio, hacia el interior. Por eso evitamos dependencias de niveles interiores hacia los exteriores.

Una vez se ha guardado el token de sesión en la base de datos en memoria, solo queda devolver los datos que queremos que el cliente reciba al iniciar su sesión.

Microservicio Tracking:

Lo forman dos módulos, *core* y *carrier*.

El módulo *core* es el encargado de recibir todos los *trackings* que la compañía cliente genera. Un ejemplo de un tracking entrante sería el de la Ilustración 16:

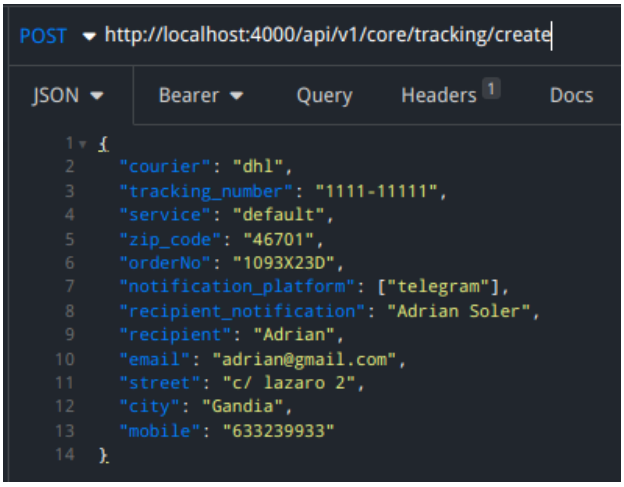

Ilustración 16. Ejemplo de solicitud HTTP para la creación de un tracking en los sistemas de seguimiento de TrackingLab

Cuando un tracking llega al endpoint del controlador del módulo de *Tracking*, lo primero es llamar al microservicio de *Identity* para verificar la validez del token de acceso mediante el *JwtGuard* alojado en el módulo de *auth.*

Si el token es válido y no ha expirado, el flujo de llamadas puede continuar.

Una vez el tracking llega a la capa de aplicación, se ejecutan estas acciones:

- Comprobación de parámetros de la petición. Se analiza que no haya errores, que existan los campos obligatorios y que no estén vacíos.
- Se crea la entidad de dominio con la que vamos a trabajar.
- Se verifica que el nombre del transportista corresponda con los nombres de transportistas guardados en la base de datos de configuración de *tracking.*
- Se envía el *tracking* al módulo de *carrier.*
- Si la respuesta del servicio de sincronización de tracking se completa satisfactoriamente, entonces se guarda el seguimiento en la base de datos de *MongoDB* indicándolo en la propiedad de sincronizado, en su defecto, se guarda como no sincronizado, de esta manera es posible “rescatar” los *trackings* no sincronizados más adelante mediante un proceso que se ejecuta cada *x* tiempo (*CRON JOB*).

Por otro lado, el módulo *carrier* se ocupa de la comunicación entre el *SaaS* y las APIs de terceros para sincronizar todos los seguimientos y recibir los cambios de estados cuando sucedan. También se encarga de notificar al sistema de comunicación por eventos *Apache Kafka* que veremos más adelante.

Estos son los pasos que se ejecutan a nivel de capa de aplicación:

- Filtrar el *tracking* diferenciando entre transportista.
- Crear la entidad de *tracking* a sincronizar.
- Conectar con el repositorio y enviar el tracking hacia la *API* del transportista.

Microservicio Notification:

Está formado por un único módulo, el de Telegram. Mientras que el microservicio de *Tracking*, concretamente el módulo de *carrier*, se ocupa de emitir (hacia un topic de *Kafka*) los eventos de cambio de estado de los diferentes transportistas. Este módulo se encarga de recibirlos a partir del mismo topic al cual está subscrito.

Una vez la información del estado ha llegado al microservicio de *Notification* y se ha redirigido al servicio de envío de notificaciones de usuarios, esta se prepara para ser formateada y enviada a la *API* de Telegram a través de un bot llamado TrackingLabBot.

Este bot solo necesita conocer el id de usuario o el id del chat del usuario (si ya existía una conversación anterior) que vienen dentro del evento de Kafka recibido.

Por último, el usuario recibe en su móvil una notificación vía Telegram en nombre de TrackingLabBot donde se le indica la información detallada del seguimiento de su pedido. Ver 5.4.7. Notificación vía Telegram.

4.4.4 Lógica compartida

Paquete pkg-shared:

Independientemente de trabajar sobre una infraestructura monolítica, monolítica distribuida o completamente distribuida (microservicios), en la mayoría de las ocasiones es necesario centralizar cierta lógica compartida con la intención de ser reutilizada por distintos módulos, clases o métodos.

Además, crear paquetes de código reutilizables permite optimizar el desarrollo, simplificar el mantenimiento, promover la coherencia, facilitar la escalabilidad y flexibilidad, y fomentar la colaboración entre equipos. Es una práctica efectiva para maximizar la eficiencia y la calidad del software en un entorno de microservicios.

Para ello, se ha creado un paquete que centraliza las siguientes funcionalidades compartidas por los tres microservicios:

- Proveedor de conexión y configuración para:
  - MySQL
  - MongoDB
  - Redis
  - Apache Kafka

- Clases y métodos de utilidades para:
  - Criptografía
  - Conversiones
  - Formato

- Paquete de definición de eventos para:
  - Apache Kafka

- Paquetes de tipado
  - HTTP
  - Excepciones

Debido a que estas funcionalidades son basadas en lógica de programación y no tanto en lógica de negocio, se obvia el desglose de estas. El código se encuentra disponible en los anexos de este trabajo.

4.4.5 Bases de datos

En este apartado se mostrará la estructura de las bases de datos utilizadas en el proyecto y el porqué de su uso.

Disponemos de tres bases de datos, MySQL, MongoDB y Redis.

MySQL se usa principalmente para transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), aunque esta base de datos relacional es buena para manejar datos estáticos y dinámicos, se usa para almacenar y gestionar datos de registros de usuarios y configuraciones. Ofrece un buen rendimiento, seguridad y robustez.

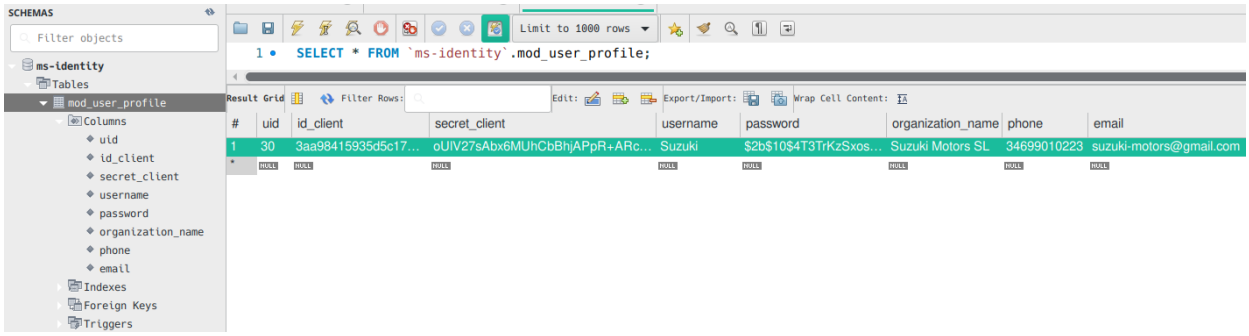

Ilustración 17. Esquema de las compañías subscritas a TrackingLab en MySQL

MongoDB, por el contrario, es no relacional y se utiliza para la escalabilidad y la flexibilidad de la aplicación, lo que lo hace ideal cuando se manejan grandes volúmenes de datos no estructurados, como, por ejemplo, los trackings generados y los cambios de estado. Además, no utiliza tablas y no requiere una estructura de datos predefinida, sino que usa documentos que permiten agregar y modificar campos fácilmente sin tener que cambiar el esquema de la base de datos.

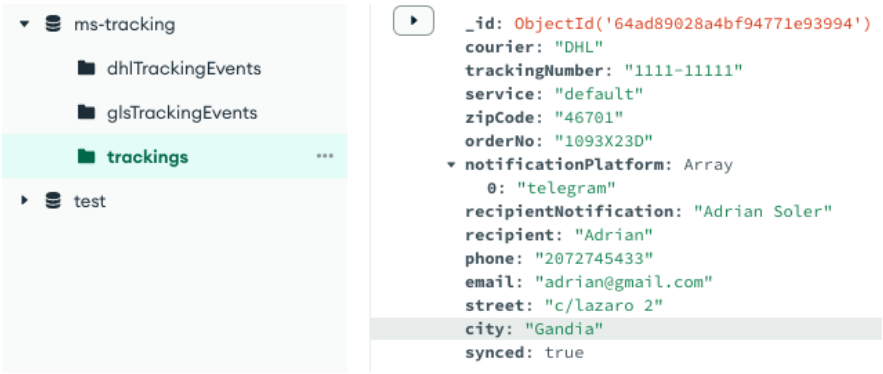

Ilustración 18. Esquema de los trackings creados en MongoDB

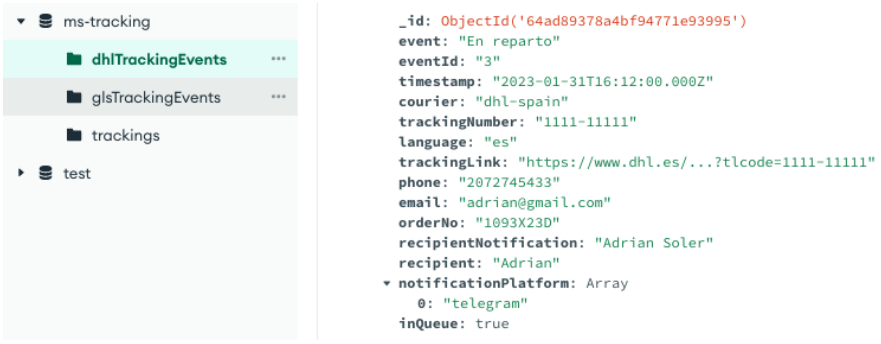

Ilustración 19. Esquema de los cambios de estado en los trackings en MongoDB

Redis es una base de datos en memoria, lo que le permite realizar operaciones rápidamente. Esto lo hace ideal para nuestra aplicación ya que en ocasiones requiere una alta velocidad y baja latencia. Es utilizada también para aligerar la carga cuando hay una demanda de datos muy elevada en las otras bases de datos. Como en esta aplicación no se necesita cumplir con estos requisitos ya que no existe tal carga de trabajo, Redis solo se usa como auxiliar para almacenar los tokens de sesión.

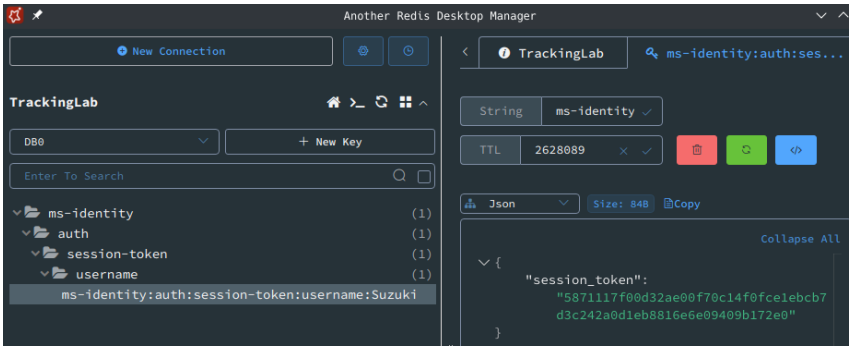

Ilustración 20. Esquema clave-valor de los tokens de sesión temporales en Redis

4.4.6 Integración con las APIs de transportistas

Debido a que no es posible, por motivos obvios, contratar los servicios de la API del transportista para realizar las pruebas, se ha simulado tanto el envío de información de tracking a un *endpoint* de terceros ficticio como la recepción de actualizaciones de los estados de estos seguimientos (Ilustración 21).

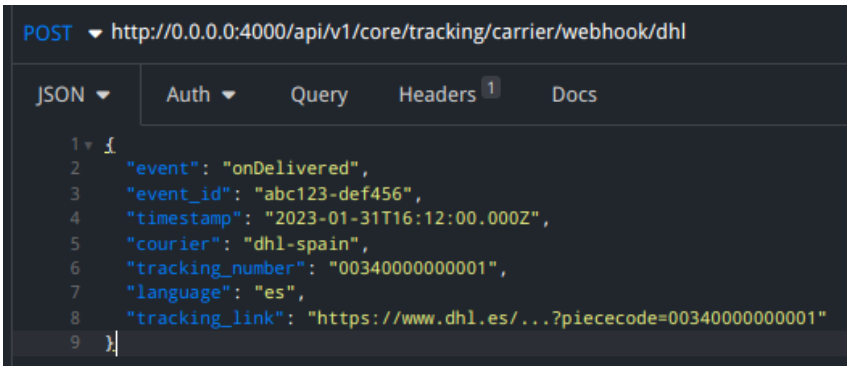

Ilustración 21. Simulación de webhook lanzado por la API de la empresa DHL

4.4.7 Apache Kafka y eventos de tracking

A veces no depende de una respuesta al realizar una petición a algún microservicio, el resultado de una acción no es relevante para seguir ejecutando las siguientes. Para estos casos, se ha optado por un sistema de comunicación asíncrono basado en eventos. *Kafka* es una plataforma de streaming distribuida que se utiliza para la transmisión de datos en tiempo real, además, es ideal para aplicaciones y sistemas que requieren una alta velocidad, una gran escalabilidad y una alta disponibilidad.

Los productores y los consumidores pueden interactuar con uno o varios topics. Los productores pueden enviar mensajes a varios topics, mientras que los consumidores pueden suscribirse a uno o varios topics.

En el microservicio de *Tracking*, concretamente en el módulo de *carrier*, se instancia un productor que envia sus mensajes a un topic en Kafka. Si en algún momento los nodos del clúster de Kafka no están disponibles por inactividad o por fallo en la red, entonces no se guardarán los eventos en la cola de eventos de Kafka ni se podrán entregar a los consumidores, para esos casos, se lanza una excepción y automáticamente se guarda en base de datos con el valor de “inQueue” a falso. De esta manera la aplicación se asegura de mantener los eventos el tiempo necesario para re-encolarlos en caso de un fallo del sistema. En la Ilustración 22 es posible ver todo este proceso.

```typescript
@Injectable()
export class CarrierDhlTrackingEventService {
  constructor(
    @Inject(Provider.KafkaProducer) private readonly kafkaClient: ClientKafka,
    @Inject('GET_TRACKING')
    private readonly getTrackingService: ICoreGetTracking,
    private readonly carrierRepo: CarrierDbRepository
  ) {}

  public async run(dto: DhlTrackingEventDto): Promise<boolean> {
    const tracking: Tracking = await this.getTrackingService.run(dto.tracking_number)
    const dhlTrackingEvent: TrackingStatusCreatedEvent = this.createEvent(dto, tracking)
    try {
      this.kafkaClient.emit(process.env.KAFKA_TRACKING_CARRIER_TOPIC, JSON.stringify(dhlTrackingEvent))
    } catch (error) {
      dhlTrackingEvent.inQueue = false
      throw new CarrierException('The tracking has not been sent to kafka')
    } finally {
      await this.carrierRepo.saveDhlTrackingEvent(dhlTrackingEvent)
    }
    return true
  }
```

Ilustración 22. Emisión de eventos de tracking en el microservicio de Tracking

Una vez el evento alcanza el topic de Kafka (Ilustración 23), pasa a estar disponible para todos los consumidores que estén subscritos a él.

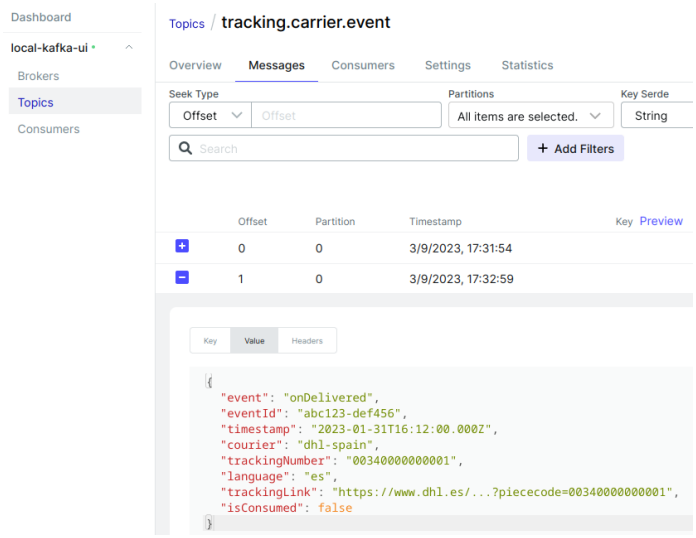

Ilustración 23: Mensaje recibido y encolado en el topic de Apache Kafka

En el microservicio de Notification, este subscriptor recibe el evento generado, obtiene los campos necesarios para realizar la llamada a la API de Telegram, construye y le da formato a la notificación, y, por último, lo envía.

4.4.8 Notificación vía Telegram

Aprovechando la interactividad que proporciona la aplicación de Telegram, es posible añadir un mensaje mucho más personalizado que un mensaje de texto simple, se expone la Ilustración 24 como ejemplo de una notificación recibida.

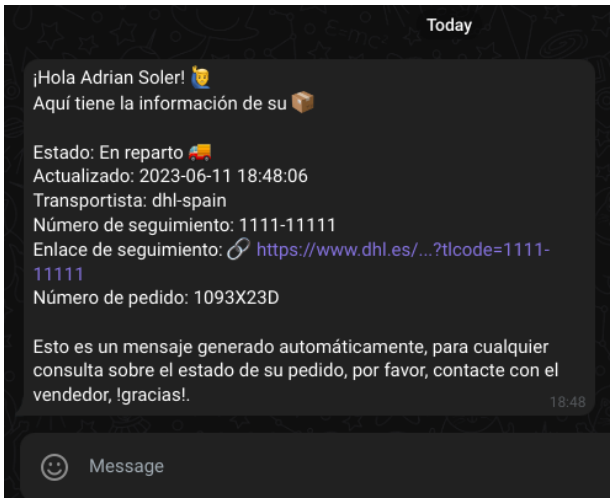

Ilustración 24. Mensaje recibido sobre el estado del envío en Telegram

Aunque para el objetivo del proyecto solo se ha personalizado un único tipo de mensaje genérico sobre la notificación de un cambio en el estado del pedido. Es posible dotar a los mensajes de muchas más características que pueden darle más legibilidad y dinamismo a la notificación, gracias a la potencia que ofrece el utilizar un bot de Telegram como mensajero.

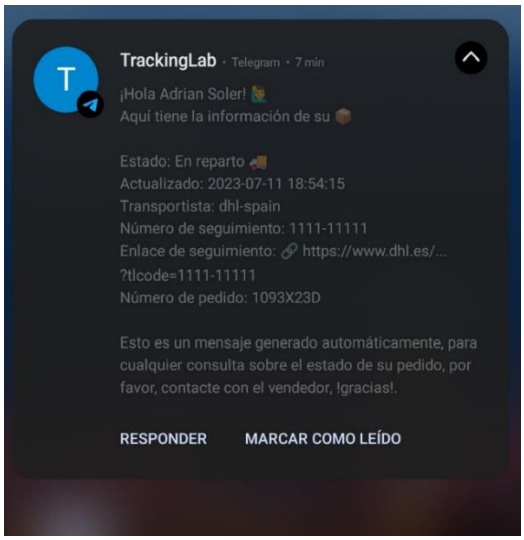

Ilustración 25. Notificación de Telegram recibida en un dispositivo móvil

4.5 Anexos

Esta sección ofrece una visión más detallada y completa de los datos recopilados y se adjuntan diferentes archivos, enlaces y diagramas para mejorar la interpretación y comprensión del proyecto.

- Manual de desarrollador
- Diagramas de secuencia
- Fichero Docker con todos los servicios necesarios para la ejecución del proyecto
- Enlaces a repositorios de GitHub

5. Pruebas

En este apartado se llevan a cabo algunas pruebas con Jest para verificar el comportamiento individual de las diversas unidades de código, como funciones, métodos, clases o módulos, asegurando que cada cual funcione correctamente de manera aislada. Estas pruebas nos permiten identificar y corregir errores de manera temprana, garantizando la calidad y confiabilidad del software que estamos desarrollando.

Por ejemplo, en el módulo de Notification, se han probado los métodos encargados de obtener la información de seguimiento, así como de darle formato y posteriormente, notificar a los usuarios de Telegram.

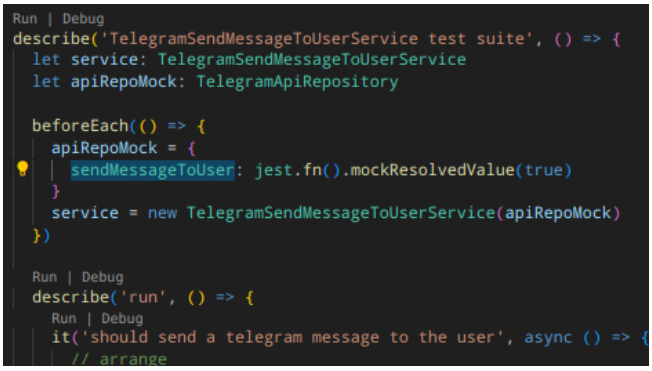

Ilustración 26. Mock del método sendMessageToUser para forzar el resultado

Como se puede ver en la Ilustración 26, se simula un repositorio devolviendo el valor booleano requerido para probar más adelante los métodos que lo usen; gracias a esta simulación, se evita que algún dato o acción externa del repositorio afecte al resultado buscado y, si ocurre un error, al desacoplar el caso de uso, es fácil encontrarlo.

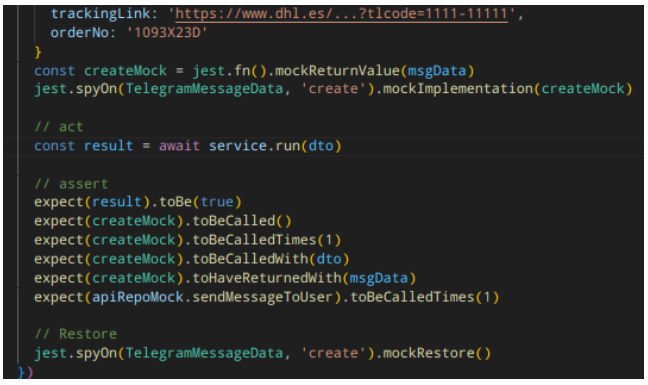

Ilustración 27. Organización, ejecución y comprobación del resultado de la prueba

Siguiendo el patrón AAA (Arrange, Act, Assert) de desarrollo de pruebas visible en la Ilustración 27, conseguimos, organizadamente, preparar los datos esperados y generar las simulaciones necesarias, llamar al método que se quiere testear y lanzar validaciones consecutivas para buscar cualquier problema y poder subsanarlo.

En la Ilustración 28 se incluye el resultado exitoso de la ejecución de las pruebas. Una vez se corrigen los errores encontrados, si es que los hubiera, la aplicación pasa a estar lista para su puesta en marcha.

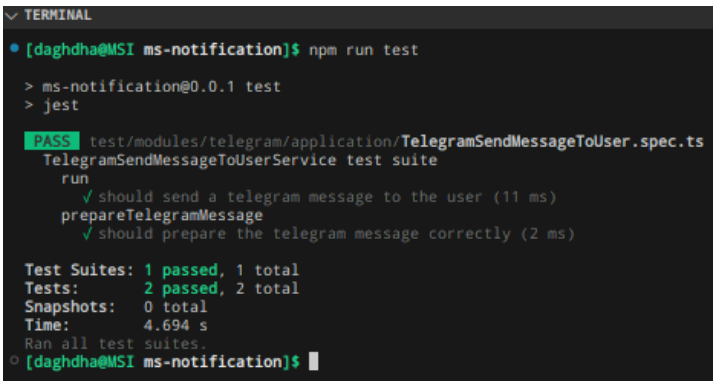

Ilustración 28. Resultados de la prueba finalizados con éxito

6. Conclusiones y futuras mejoras

En este trabajo se ha propuesto el desarrollo de una plataforma para mejorar y mitigar la carga de trabajo que sufren las empresas a la hora de enviar sus productos.

El principal objetivo del proyecto ha sido abordar directamente la falta de comunicación fluida y eficiente entre vendedor y comprador durante el proceso de envío y seguimiento de pedidos. Para solucionarlo, se ha creado esta solución que consigue externalizar el seguimiento de pedidos hacia diversos transportistas y centralizar la comunicación con el cliente a través de una plataforma de mensajería instantánea como Telegram. Se han utilizado la mayoría de las tecnologías que muchas empresas utilizan hoy en día para lograr una arquitectura escalable y robusta, así como buenas prácticas y novedosos estilos arquitectónicos de desarrollo de software.

Sin embargo, siempre existen limitaciones, algunas de las cuales podrían ser:

- La dependencia de las APIs de los transportistas, ya que cualquier cambio en sus sistemas o políticas podría afectar la funcionalidad de la aplicación.
- La disponibilidad y confiabilidad de los servicios de mensajería instantánea utilizados para la comunicación con los clientes.

Para mejorar la aplicación en el futuro, se podrían considerar las siguientes líneas de trabajo:

- Ampliar la integración con más transportistas y servicios de mensajería para brindar opciones flexibles a los comercios electrónicos en el ámbito nacional.
- Implementar análisis de datos y generación de informes para que las empresas puedan obtener información valiosa sobre el rendimiento de los envíos y la satisfacción del cliente.
- Mejorar la experiencia del usuario mediante mensajes más intuitivos y personalizables. Y poder generar una comunicación bidireccional y dar un trato más cercano al cliente, pudiendo recuperar la información actualizada de sus pedidos si lo requiera.

Adicionalmente, podría resultar de interés:

- Agregar funcionalidades de gestión de devoluciones y resolución de incidencias para ofrecer un servicio completo a los clientes.

Este proyecto ha involucrado y desarrollado varias competencias transversales:

- Diseño y proyecto.
- Conocimiento de problemas contemporáneos.
- Aprendizaje autónomo y adaptabilidad.
- Planificación y gestión del tiempo.
- Análisis y resolución de problemas.

Por último, los objetivos de desarrollo sostenible logrados han sido:

- ODS 9 (Industria, Innovación e Infraestructura) generando una mejora de la eficiencia operativa de las empresas y optimización de los procesos de envío.
- ODS 12: (Producción y Consumo Responsables) al proporcionar una comunicación clara y actualizada con los clientes, buscando mejorar su experiencia y satisfacción.
- ODS 13: (Acción por el Clima) al utilizar tecnologías y arquitecturas escalables, se fomenta reducir el impacto ambiental de las operaciones comerciales.

7. Referencias bibliográficas

[1] DiTendria, Digital Marketing Trends 2020. “Estadísticas sobre comercio electrónico en España y en el mundo” https://mktefa.ditrendia.es/blog/estadisticas-comercio-electronico [accessed Jan. 14, 2023]

[2] Palese, B. and Usai, A. (2018). The relative importance of service quality dimensions in E-commerce experiences. International Journal of Information Management. 40: 132-140. [accessed Jan. 12, 2023]

[3] Chnar Mustafa Mohammed & Subhi R.M Zeebaree, 2021. "Sufficient Comparison Among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review," International Journal of Science and Business, IJSAB International, vol. 5(2), pages 17-30. [accessed Jan. 16, 2023]

[4] Sergio Vergara (2019), API Gateway en tu arquitectura de microservicios. ITDO Agencia de desarrollo Web, APPs y Marketing en Barcelona [accessed Jan. 20, 2023]

[5] Bernal-Jiménez, M. C., & Rodríguez-Ibarra, D. L. (2019). Las tecnologías de la información y comunicación como factor de innovación y competitividad empresarial. Scientia Et Technica, 24(1), 85-96. [accessed Jan. 15, 2023]

[6] Sergio Vergara (2019), ¿Cuál es el mejor método de autentificación en un API REST? ITDO Agencia de desarrollo Web, APPs y Marketing en Barcelona [accessed Jan. 26, 2023]

[7] Daniel Estiven Rico Posada (2019), Guía práctica de autenticación basada en tokens para NodeJS, Medium [accessed Feb. 01, 2023]

[8] Abharian, Y. (2021). MECHANISMS FOR IMPLEMENTING HEXAGONAL ARCHITECTURE IN NODE JS. *Global Prosperity*, *1*(3-2), 41-48. [accessed Feb. 10, 2023]

[9] Marianna Rolfo (2022), ¿Cómo implementar la autenticación JWT de NestJS? Codigoencasa [accessed Feb. 01, 2023]

[10] Noel Rodríguez Calle (2020), Ejemplo de Arquitectura Hexagonal con Spring Data Refactorizando [accessed Feb. 04, 2023]

[11] Surwase, V. (2016). REST API modeling languages-a developer's perspective. *Int. J. Sci. Technol. Eng*, *2*(10), 634-637. [accessed Feb. 11, 2023]

[12] Petter Holmström (2020), DDD Part 1: Strategic Domain-Driven Design, Vaadin [accessed Feb. 08, 2023]

[13] Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, *17*(3), 228. [accessed Feb. 03, 2023]

[14] Petter Holmström (2020), DDD Part 2: Tactical Domain-Driven Design, Vaadin [accessed Feb. 08, 2023]

[15] Petter Holmström (2020), DDD Part 3: Domain-Driven Design and the Hexagonal Architecture, Vaadin, Vaadin [accessed Feb. 09, 2023]

[16] Nugraha, K. A., & Sebastian, D. (2021, November). Designing Consultation Chatbot Using Telegram API and Webhook-based NodeJS Applications. In the 7th *International Conference on Education and Technology (ICET 2021)* (pp. 119-122). Atlantis Press. [accessed Feb. 22, 2023]